Raster Vision Documentation

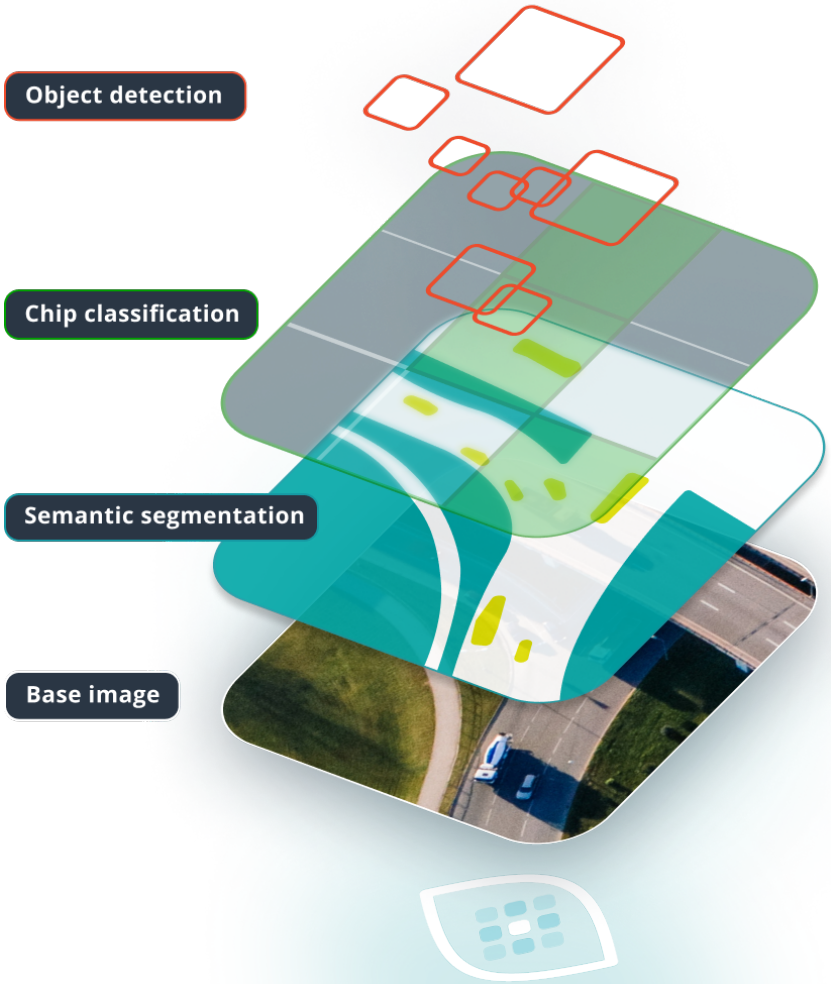
Release 0.20.1

Azavea

Jan 04, 2023

CONTENTS

Raster Vision is an open source library and framework for Python developers building computer vision models on satellite, aerial, and other large imagery sets (including oblique drone imagery). There is built-in support for chip classification, object detection, and semantic segmentation using PyTorch.


Object detection
Chip classification
Semantic segmentation
Base image

CHAPTER

ONE

WHY ANOTHER DEEP LEARNING LIBRARY?

Most machine learning libraries implement the core functionality needed to build and train models, but leave the "plumbing" to users to figure out. This plumbing is the work of implementing a repeatable, configurable workflow that creates training data, trains models, makes predictions, and computes evaluations, and runs locally and in the cloud. Not giving this work the engineering effort it deserves often results in a bunch of hacky, one-off scripts that are not reusable.

In addition, most machine learning libraries cannot work out-of-the-box with massive, geospatial imagery. This is because of the format of the data (eg. GeoTIFF and GeoJSON), the massive size of each scene (eg. 10,000 x 10,000 pixels), the use of map coordinates (eg. latitude and longitude), the use of more than three channels (eg. infrared), patches of missing data (eg. NODATA), and the need to focus on irregularly-shaped AOIs (areas of interest) within larger images.

CHAPTER

TWO

WHAT ARE THE BENEFITS OF USING RASTER VISION?

- Programmatically configure pipelines in a concise, modifiable, and reusable way, using abstractions such as *pipelines*, *backends*, *datasets*, and *scenes*.
- Let the framework handle the challenges and idiosyncrasies of doing machine learning on massive, geospatial imagery.
- Run pipelines and individual commands from the command line that execute in parallel, locally or on AWS Batch.
- Read files from HTTP, S3, the local filesystem, or anywhere with the pluggable *File Systems* architecture.
- Make predictions and build inference pipelines using a single "model bundle" which includes the trained model and associated metadata.
- *Customize* Raster Vision using the *plugins* architecture.

CHAPTER

THREE

WHO IS RASTER VISION FOR?

- Developers **new to deep learning** who want to get spun up on applying deep learning to imagery quickly or who want to leverage existing deep learning libraries like PyTorch for their projects simply.
- People who are **already applying deep learning** to problems and want to make their processes more robust, faster and scalable.
- Machine Learning engineers who are **developing new deep learning capabilities** they want to plug into a framework that allows them to focus on the interesting problems.
- **Teams building models collaboratively** that are in need of ways to share model configurations and create repeatable results in a consistent and maintainable way.

CHAPTER

FOUR

INSTALLATION

4.1 Configuration

Raster Vision is configured via the everett library, and will look for configuration in the following locations, in this order:

- Environment Variables
- A `.env` file in the working directory that holds environment variables.
- Raster Vision INI configuration files

By default, Raster Vision looks for a configuration file named `default` in the `${HOME}/.rastervision` folder.

4.1.1 Profiles

Profiles allow you to specify profile names from the command line or environment variables to determine which settings to use. The configuration file used will be named the same as the profile: if you had two profiles (the `default` and one named `myprofile`), your `${HOME}/.rastervision` would look like this:

```
> ls ~/.rastervision
default    myprofile
```

Use the `rastervision --profile` option in the *Command Line Interface* to set the profile.

4.1.2 Configuration-file Sections

AWS_S3

```
[AWS_S3]
requester_pays = False
```

- `requester_pays` - Set to True if you would like to allow using requester pays S3 buckets. The default value is False.

BATCH

See *Running on AWS Batch*.

4.1.3 Environment Variables

Any profile file option can also be stated in the environment. Just prepend the section name to the setting name, e.g. `export AWS_S3_REQUESTER_PAYS="False"`.

In addition to those environment variables that match the INI file values, there are the following environment variable options:

- `TMPDIR` - Setting this environment variable will cause all temporary directories to be created inside this folder. This is useful, for example, when you have a Docker container setup that mounts large network storage into a specific directory inside the Docker container. The tmp_dir can also be set on *Command Line Interface* as a root option.
- `RV_CONFIG` - Optional path to the specific Raster Vision Configuration file. These configurations will override configurations that exist in configurations files in the default locations, but will not cause those configurations to be ignored.
- `RV_CONFIG_DIR` - Optional path to the directory that contains Raster Vision configuration. Defaults to `${HOME}/.rastervision`

4.2 Using GPUs

To run Raster Vision on a realistic dataset in a reasonable amount of time, it is necessary to use a machine with a GPU. Note that Raster Vision will use a GPU if it detects that one is available. If you don't own a machine with a GPU, it is possible to rent one by the minute using a cloud provider such as AWS.

4.2.1 Check that GPU is available

Regardless of how you are running Raster Vision, we recommend you ensure that the GPUs are actually enabled. If you don't, you may run a training job that you think is using the GPU and isn't, and runs very slowly.

One way to check this is to make sure PyTorch can see the GPU(s). To do this, open up a `python` console and run the following:

```
import torch
torch.cuda.is_available()
torch.cuda.get_device_name(0)
```

This should print out something like:

```
True
Tesla K80
```

If you have nvidia-smi installed, you can also use this command to inspect GPU utilization while the training job is running:

```
> watch -d -n 0.5 nvidia-smi
```

4.2.2 GPUs and Docker

If you would like to run Raster Vision in a Docker container with GPUs, you'll need to check some things so that the Docker container can utilize the GPUs.

First, you'll need to install the nvidia-docker runtime on your system. Follow their Quickstart and installation instructions. Make sure that your GPU is supported by NVIDIA Docker - if not you might need to find another way to have your Docker container communicate with the GPU. If you figure out how to support more GPUs, please let us know so we can add the steps to this documentation!

When running your Docker container, be sure to include the `--runtime=nvidia` option, e.g.

```
> docker run --runtime=nvidia --rm -it quay.io/azavea/raster-vision:pytorch-0.20 /bin/
↪bash
```

or use the `--gpu` option with the `docker/run` script.

4.2.3 Running on AWS EC2

The simplest way to run Raster Vision on an AWS GPU is by starting a GPU-enabled EC2 instance such as a p3.2xlarge using the Deep Learning AMI. We have tested this using the "Deep Learning AMI GPU PyTorch 1.11.0 (Ubuntu 20.04)" with id `ami-0c968d7ef8a4b0c34`. After SSH'ing into the instance, Raster Vision can be installed with `pip`, and code can be transfered to this instance with a tool such as `rsync`.

4.2.4 Running on AWS Batch

AWS Batch is a service that makes it easier to run Dockerized computation pipelines in the cloud. It starts and stops the appropriate instances automatically and runs jobs sequentially or in parallel according to the dependencies between them. To run Raster Vision using AWS Batch, you'll need to setup your AWS account with a specific set of Batch resources, which you can do using *Setup AWS Batch using CloudFormation*. After creating the resources on AWS, set the following configuration in your Raster Vision config. Check the AWS Batch console to see the names of the resources that were created, as they vary depending on how CloudFormation was configured.

```
[BATCH]
gpu_job_queue=RasterVisionGpuJobQueue
gpu_job_def=RasterVisionHostedPyTorchGpuJobDefinition
cpu_job_queue=RasterVisionCpuJobQueue
cpu_job_def=RasterVisionHostedPyTorchCpuJobDefinition
attempts=5
```

- `gpu_job_queue` - job queue for GPU jobs
- `gpu_job_def` - job definition that defines the GPU Batch jobs
- `cpu_job_queue` - job queue for CPU-only jobs
- `cpu_job_def` - job definition that defines the CPU-only Batch jobs
- `attempts` - Optional number of attempts to retry failed jobs. It is good to set this to > 1 since Batch often kills jobs for no apparent reason.

See also:

For more information about how Raster Vision uses AWS Batch, see the section: *Running remotely*.

4.3 Installing via pip

You can directly install the library using `pip` (or `pip3` if you also have Python 2 installed).

```
> pip install rastervision==0.20
```

Note: You will also need to set a couple of environment variables required by `rasterio`. You can do it like so:

```
> export GDAL_DATA=$(pip show rasterio | grep Location | awk '{print $NF"/rasterio/gdal_
↪data/"}')
> export AWS_NO_SIGN_REQUEST=YES
```

This has been shown to work in the following environment. Variations on this environment may or may not work.

- Ubuntu Linux 20.04
- Python 3.9
- CUDA 11.6 and NVIDIA Driver 510.47.03 (for GPU support)

Raster Vision also runs on macOS version 12.1, except that the `num_workers` for the `DataLoader` will need to be set to 0 due to an issue with mulitprocessing on Macs with Python >= 3.8. It will also be necessary to install GDAL (check here for the exact version) prior to installing Raster Vision, which isn't necessary on Linux.

Warning: Raster Vision has not been tested with Windows, and will probably run into problems.

An alternative approach for running Raster Vision is to use the provided *docker images* which encapsulate a complete environment that is known to work.

4.3.1 Install individual pip packages

Raster Vision comprises a required *`rastervision.pipeline`* package, plus a number of optional plugin packages, as described in *Codebase Overview*. Each of these packages have their own dependencies, and can be installed individually. Running the following command:

```
> pip install rastervision==0.20
```

is equivalent to running the following sequence of commands:

```
> pip install rastervision_pipeline==0.20
> pip install rastervision_aws_s3==0.20
> pip install rastervision_aws_batch==0.20
> pip install rastervision_core==0.20
> pip install rastervision_pytorch_learner==0.20
> pip install rastervision_pytorch_backend==0.20
```

Another optional plugin that is available, but not installed by default, is `rastervision.gdal_vsi`.

```
> pip install rastervision_gdal_vsi==0.20
```

The command above will attempt to install GDAL via `pip`. If that fails, you can instead try installing via `conda` as shown below. Replace <`version`> with the version listed here.

```
> conda install -c conda-forge gdal==<version>
```

4.4 Docker Images

Using the Docker images published for Raster Vision makes it easy to use a fully set up environment. We have tested this with Docker 20, although you may be able to use a lower version.

The images we publish include plugins and dependencies for using Raster Vision with PyTorch, AWS S3, and Batch. These are published to quay.io/azavea/raster-vision. To run the container for the latest release, run:

```
> docker run --rm -it quay.io/azavea/raster-vision:pytorch-0.20 /bin/bash
```

There are also images with the *-latest* suffix for the latest commits on the `master` branch. You'll likely need to mount volumes and expose ports to make this container fully useful; see the docker/run script for an example usage.

You can also base your own Dockerfiles off the Raster Vision image to use with your own codebase. See *Bootstrap new projects with a template* for more information.

4.4.1 Docker Scripts

There are several scripts under docker/ in the Raster Vision repo that make it easier to build the Docker images from scratch, and run the container in various ways. These are useful if you are experimenting with changes to the Raster Vision source code, or writing *plugins*.

After cloning the repo, you can build the Docker image using:

```
> docker/build
```

To build an image that can run natively on an ARM64 chip, pass the `--arm64` flag. This won't be necessary for most users, but if you have an ARM64 chip, like in a recent Macbook, this will speed things up greatly.

Before running the container, set an environment variable to a local directory in which to store data.

```
> export RASTER_VISION_DATA_DIR="/path/to/data"
```

To run a Bash console in the PyTorch Docker container use:

```
> docker/run
```

This will mount the `$RASTER_VISION_DATA_DIR` local directory to to `/opt/data/` inside the container.

Warning: Users running under WSL2 in Windows will need to unset the `NAME` environment variable. For example, instead of `docker/run`, you would run `NAME='' docker/run`. By default, WSL2 sets a `NAME` variable that matches the network name of your computer. This environment variable collides with a variable in the `docker/run` script.

Warning: If you have built an ARM64 image, you should pass the `--arm64` flag to `docker/run`.

This script also has options for forwarding AWS credentials, and running Jupyter notebooks which can be seen below.

```
> docker/run --help

Usage: run <options> <command>

Run a console in a Raster Vision Docker image locally.
By default, the raster-vision-pytorch image is used in the CPU runtime.

Environment variables:
RASTER_VISION_DATA_DIR (directory for storing data; mounted to /opt/data)
RASTER_VISION_NOTEBOOK_DIR (optional directory for Jupyter notebooks; mounted to /opt/
↪notebooks)
AWS_PROFILE (optional AWS profile)

Options:
--aws forwards AWS credentials (sets AWS_PROFILE env var and mounts ~/.aws to /root/.aws)
--tensorboard maps port 6006
--name sets the name of the running container
--jupyter forwards port 8888, mounts RASTER_VISION_NOTEBOOK_DIR to /opt/notebooks, and␣
↪runs Jupyter
--docs runs the docs server and forwards port 8000
--debug forwards port 3000 for use with remote debugger
--gpu use nvidia runtime
--arm64 uses image built for arm64 architecture

All arguments after above options are passed to 'docker run'.
```

CHAPTER

FIVE

USAGE OVERVIEW

Users can use Raster Vision in a couple of different ways depending on their needs and level of experience:

- *As a library* of utilities for handling geospatial data and training deep learning models that you can incorporate into your own code.
- *As a low-code framework* in the form of the *Raster Vision Pipeline* that internally handles all aspects of the training workflow for you and only requires you to configure a few parameters.

5.1 As a library

This allows you to pick and choose which parts of Raster Vision to use. For example, to apply your existing PyTorch training code to geospatial imagery, simply create a *GeoDataset* like so:

```
from rastervision.pytorch_learner import SemanticSegmentationSlidingWindowGeoDataset

ds = SemanticSegmentationSlidingWindowGeoDataset.from_uris(
    class_config=class_config,
    image_uri=image_uri,
    label_raster_uri=label_uri,
    size=200,
    stride=100)
```

5.2 As a framework

This allows you to configure a full pipeline in one go like so:

```
def get_config(runner) -> SemanticSegmentationConfig:
    output_root_uri = '/opt/data/output/'
    class_config = ClassConfig(
        names=['building', 'background'], colors=['red', 'black'])

    base_uri = ('https://s3.amazonaws.com/azavea-research-public-data/'
                'raster-vision/examples/spacenet')
    train_image_uri = join(base_uri, 'RGB-PanSharpen_AOI_2_Vegas_img205.tif')
    train_label_uri = join(base_uri, 'buildings_AOI_2_Vegas_img205.geojson')
    val_image_uri = join(base_uri, 'RGB-PanSharpen_AOI_2_Vegas_img25.tif')
    val_label_uri = join(base_uri, 'buildings_AOI_2_Vegas_img25.geojson')
```

(continues on next page)

(continued from previous page)

```
    train_scene = make_scene('scene_205', train_image_uri, train_label_uri,
                             class_config)
    val_scene = make_scene('scene_25', val_image_uri, val_label_uri,
                           class_config)
    scene_dataset = DatasetConfig(
        class_config=class_config,
        train_scenes=[train_scene],
        validation_scenes=[val_scene])

    # Use the PyTorch backend for the SemanticSegmentation pipeline.
    chip_sz = 300

    backend = PyTorchSemanticSegmentationConfig(
        data=SemanticSegmentationGeoDataConfig(
            scene_dataset=scene_dataset,
            window_opts=GeoDataWindowConfig(
                # randomly sample training chips from scene
                method=GeoDataWindowMethod.random,
                # ... of size chip_sz x chip_sz
                size=chip_sz,
                # ... and at most 10 chips per scene
                max_windows=10)),
        model=SemanticSegmentationModelConfig(backbone=Backbone.resnet50),
        solver=SolverConfig(lr=1e-4, num_epochs=1, batch_sz=2))

    return SemanticSegmentationConfig(
        root_uri=output_root_uri,
        dataset=scene_dataset,
        backend=backend,
        train_chip_sz=chip_sz,
        predict_chip_sz=chip_sz)
```

And then run it from the command line:

```
> rastervision run local <path/to/file.py>
```

Read more about it here: *The Raster Vision Pipeline*.

CHAPTER

SIX

BASIC CONCEPTS

At a high-level, a typical machine learning workflow for geospatial data involves the following steps:

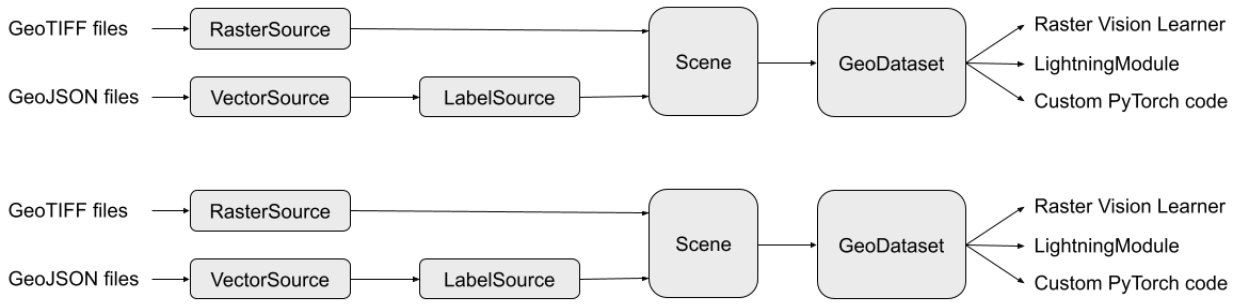
- Read geospatial data
- Train a model
- Make predictions
- Write predictions (as geospatial data)

Below, we describe various Raster Vision components that can be used to perform these steps.

6.1 Reading geospatial data

Raster Vision internally uses the following pipeline for reading geo-referenced data and coaxing it into a form suitable for training computer vision models.

When using Raster Vision *as a library*, users generally do not need to deal with all the individual components to arrive at a working *GeoDataset* (see the tutorial on *Sampling training data*), but certainly can if needed.



Below, we briefly describe each of the components shown in the diagram above.

6.1.1 RasterSource

Tutorial: *Reading raster data*

A *RasterSource* represents a source of raster data for a scene. It is used to retrieve small windows of raster data (or *chips*) from larger scenes. It can also be used to subset image channels (i.e. bands) as well as do more complex transformations using *RasterTransformers*. You can even combine bands from multiple sources using a *MultiRasterSource*.

See also:

- *RasterioSource*
- *MultiRasterSource*
- *RasterizedSource*
- *RasterTransformer*
  - *CastTransformer*
  - *MinMaxTransformer*
  - *NanTransformer*
  - *ReclassTransformer*
  - *RGBClassTransformer*
  - *StatsTransformer*

6.1.2 VectorSource

Tutorial: *Reading vector data*

Annotations for geospatial data are often represented as vector data such as polygons and lines. A *VectorSource* is Raster Vision's abstraction for a vector data reader. Just like *RasterSources*, *VectorSources* also allow transforming the data using *VectorTransformers*.

See also:

- *GeoJSONVectorSource*
- *RasterizedSource*
- *VectorTransformer*
  - *BufferTransformer*
  - *ClassInferenceTransformer*
  - *ShiftTransformer*

6.1.3 LabelSource

Tutorial: *Reading labels*

A `LabelSource` interprets the data read by raster or vector sources into a form suitable for machine learning. They can be queried for the labels that lie within a window and are used for creating training chips, as well as providing ground truth labels for evaluation against model predictions. There are different implementations available for `chip classification`, `semantic segmentation`, and `object detection`.

See also:

- `ChipClassificationLabelSource`
- `SemanticSegmentationLabelSource`
- `ObjectDetectionLabelSource`

6.1.4 Scene

Tutorial: *Scenes and AOIs*

A `Scene` is essentially a combination of a *RasterSource* and a *LabelSource* along with an optional AOI which can be specified as one or more polygons.

It can also

- hold a *LabelStore*; this is useful for evaluating predictions against ground truth labels
- just have a *RasterSource* without a *LabelSource* or *LabelStore*; this can be useful if you want to turn it into a dataset to be used for unsupervised or self-supervised learning

Scenes can also be more *conveniently initialized* using the factory functions defined in `rastervision.core.data.utils.factory`.

6.1.5 GeoDataset

Tutorial: *Sampling training data*

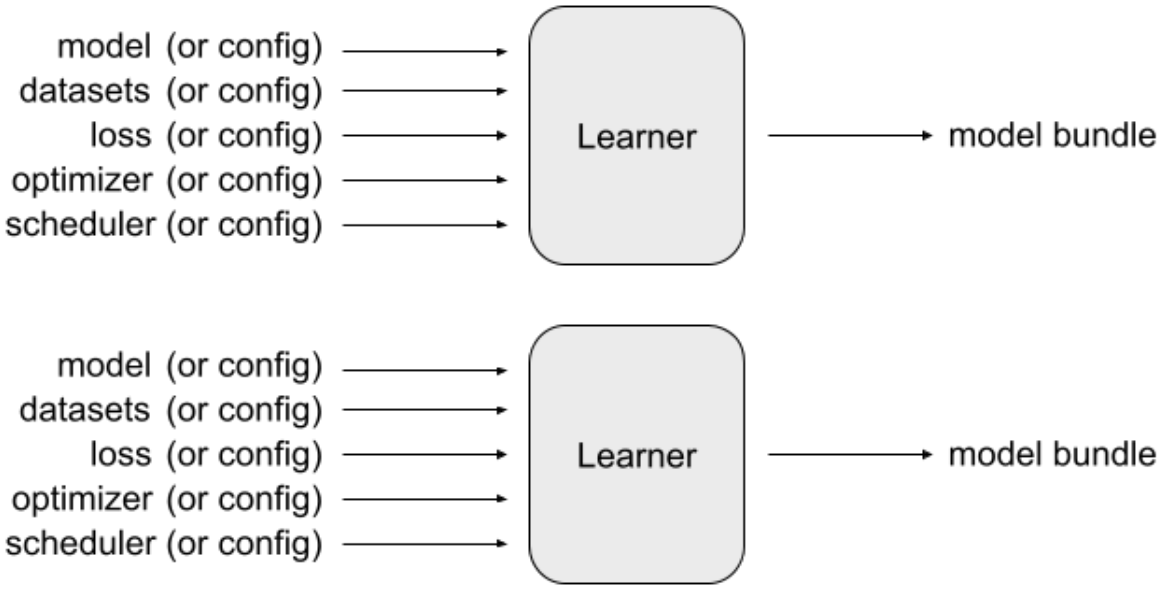
A `GeoDataset` (provided by Raster Vision's `pytorch_learner` plugin) is a PyTorch-compatible dataset that can readily be wrapped into a DataLoader and used by any PyTorch training code. Raster Vision provides a `Learner` class for training models, but you can also use GeoDatasets with either your own custom training code, or with a 3rd party library like *PyTorch Lightning*.

See also:

- `AlbumentationsDataset` (base dataset class)
- `GeoDataset`
  - `SlidingWindowGeoDataset`
    - `ClassificationSlidingWindowGeoDataset`
    - `SemanticSegmentationSlidingWindowGeoDataset`
    - `ObjectDetectionSlidingWindowGeoDataset`
    - `RegressionSlidingWindowGeoDataset`
  - `RandomWindowGeoDataset`
    - `ClassificationRandomWindowGeoDataset`

* *SemanticSegmentationRandomWindowGeoDataset*
* *ObjectDetectionRandomWindowGeoDataset*
* *RegressionRandomWindowGeoDataset*

6.2 Training a model



6.2.1 Learner

Tutorial: *Training a model*

Raster Vision's *pytorch_learner* plugin provides a *Learner* class that encapsulates the entire training process. It is highly configurable. You can either fill out a *LearnerConfig* and have the *Learner* set everything up (datasets, model, loss, optimizers, etc.) for you, or you can pass in your own models, datasets, etc. and have the *Learner* use them instead.

The main output of the *Learner* is a trained model. This is available as a `last-model.pth` file which is a serialized dictionary of model weights that can be loaded into a model via

```
model.load_state_dict(torch.load('last-model.pth'))
```

You can also make the *Learner* output a "model-bundle" (via *save_model_bundle()*), which outputs a zip file containing the model weights as well as a config file that can be used to re-create the *Learner* via *from_model_bundle()*.

There are Learner subclasses for *chip classification*, *semantic segmentation*, *object detection*, and *regression*.

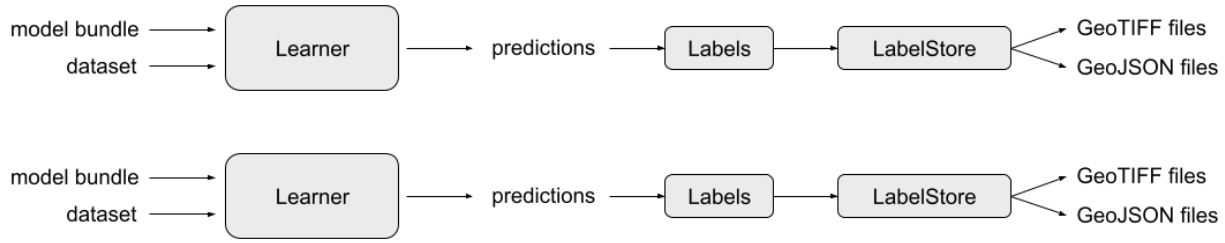
Note: The *Learners* are not limited to *GeoDatasets* and can work with any PyTorch-compatible image dataset. In fact, *pytorch_learner* also provides an *ImageDataset* class for dealing with non-geospatial datasets.

See also:

- *ClassificationLearner*
- *SemanticSegmentationLearner*
- *ObjectDetectionLearner*
- *RegressionLearner*
- *ImageDataset*
  - *ClassificationImageDataset*
  - *SemanticSegmentationImageDataset* (and *SemanticSegmentationDataReader*)
  - *ObjectDetectionImageDataset* (and *CocoDataset*)
  - *RegressionImageDataset* (and *RegressionDataReader*)

6.3 Making predictions and saving them

Tutorial: *Prediction and Evaluation*



Having trained a model, you would naturally want to use it to make predictions on new scenes. The usual workflow for this is:

1. Instantiate a *Learner* form a model-bundle (via *from_model_bundle()*)
2. Instantiate the appropriate *SlidingWindowGeoDataset* subclass e.g. *SemanticSegmentationSlidingWindowGeoDataset* (can be done easily using the convenience method *from_uris()*)
3. Pass the *SlidingWindowGeoDataset* to *Learner.predict_dataset()*
4. Convert predictions into the appropriate *Labels* subclass e.g. *SemanticSegmentationLabels* (via *from_predictions()*)
5. Save the *Labels* to file (via *save()*)
   - Alternatively, you can Instantiate an appropriate *LabelStore* subclass and pass the *Labels* to *LabelStore.save()*

6.3.1 Labels

The *Labels* class is an in-memory representation of labels. It can represent both ground truth labels and model predictions.

See also:

- *ChipClassificationLabels*
- *ObjectDetectionLabels*
- *SemanticSegmentationLabels*
  - *SemanticSegmentationDiscreteLabels*
  - *SemanticSegmentationSmoothLabels*

6.3.2 LabelStore

A *LabelStore* abstracts away the writing of *Labels* to file. It can also be used to read previously written predictions back as *Labels* which is useful for evaluating predictions.

See also:

- *ChipClassificationGeoJSONStore*
- *ObjectDetectionGeoJSONStore*
- *SemanticSegmentationLabelStore*

CHAPTER

SEVEN

TUTORIALS

The following tutorials highlight many of the things you can do with Raster Vision and demonstrate some best-practices.

Building blocks

7.1 Reading raster data

7.1.1 Reading rasters using `RasterSource`

The *`RasterSource`* is Raster Vision's abstraction for windowed reading from a raster image.

One concrete implementation of it is the *`RasterioSource`* which is a wrapper around the rasterio library and allows reading from all file formats supported by it.

We can create one from an image like shown below. `allow_streaming=True` allows us to take advantage of `rasterio`'s remote-file-reading capabilities; setting it to `False` will cause Raster Vision to download the image.

[1]:
```
from rastervision.core.data import RasterioSource

img_uri = 's3://azavea-research-public-data/raster-vision/examples/spacenet/RGB-
↪PanSharpen_AOI_2_Vegas_img205.tif'
raster_source = RasterioSource(img_uri, allow_streaming=True)
```

[2]:
```
raster_source.shape
```

[2]:
```
(650, 650, 3)
```

[3]:
```
raster_source.dtype
```

[3]:
```
dtype('uint16')
```

Reading chips

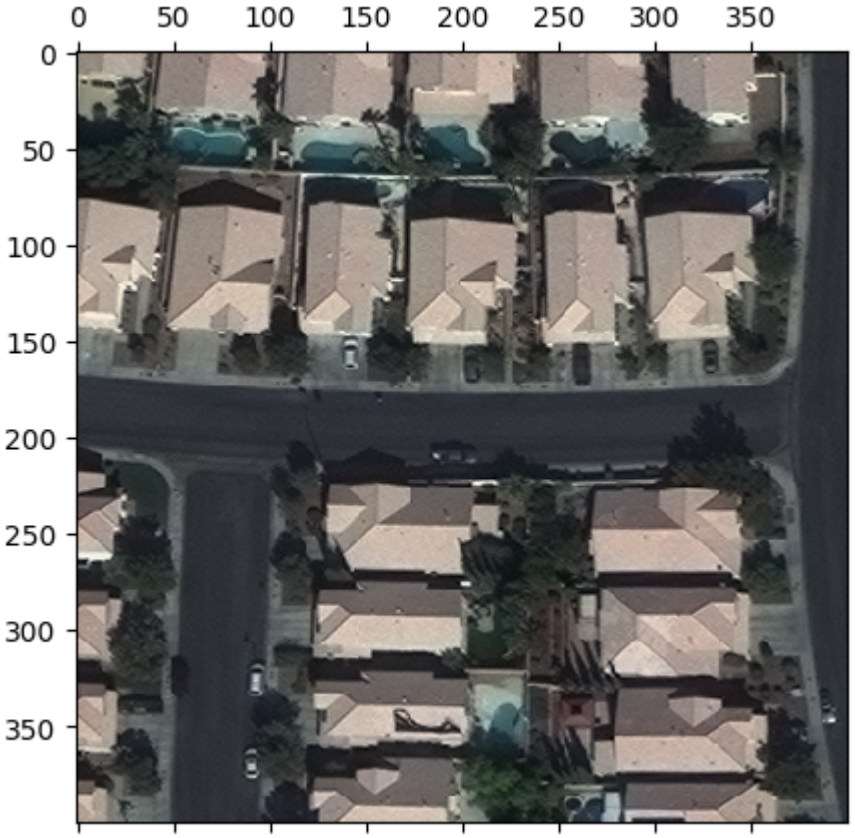
RasterSources support numpy-like array slicing, so we can read a smaller chip within the full raster like so:

[4]:
```
chip = raster_source[:400, :400]
chip.shape
```

[4]:
```
(400, 400, 3)
```

The returned chip is a numpy array which we can plot using matplotlib. Note that since the values are uint16, we first normalize them before plotting.

[5]:
```
from matplotlib import pyplot as plt

colors_mins = chip.reshape(-1, chip.shape[-1]).min(axis=0)
colors_maxs = chip.reshape(-1, chip.shape[-1]).max(axis=0)
chip_normalized = (chip - colors_mins) / (colors_maxs - colors_mins)

fig, ax = plt.subplots(figsize=(5, 5))
ax.matshow(chip_normalized)
plt.show()
```



nbsphinx-code-borderwhite

We can even do slightly fancier indexing. The example below reads a 400x400 chip from the the first band in the raster, subsampled to 100x100.

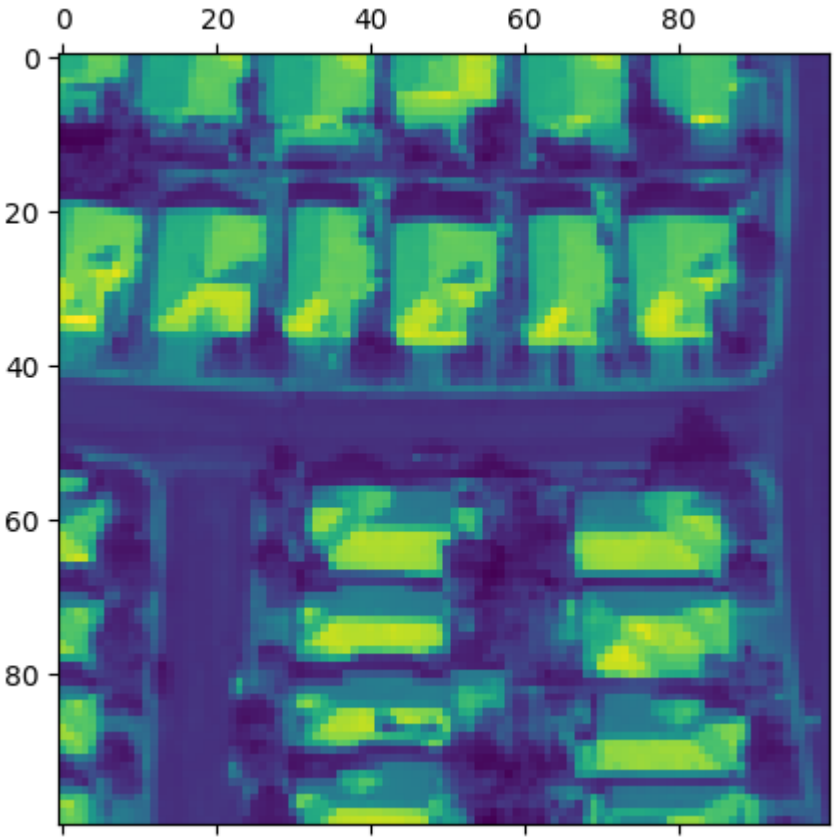
[6]:
```
chip = raster_source[:400:4, :400:4, [0]]
print('chip.shape:', chip.shape)
```

(continues on next page)

(continued from previous page)

```
fig, ax = plt.subplots(figsize=(5, 5))
ax.matshow(chip)
plt.show()
```

```
chip.shape: (100, 100, 1)
```



nbsphinx-code-borderwhite

The fancy slicing is just syntactic-sugar for the *RasterioSource.get_chip()* method. The last call is internally translated to the following call to the *get_chip()* method:

[7]:
```
from rastervision.core.box import Box

chip = raster_source.get_chip(
    window=Box(ymin=0, xmin=0, ymax=400, xmax=400),
    out_shape=(100, 100),
    bands=[0])
chip.shape
```

[7]:
```
(100, 100, 1)
```

Learn more about the handy *Box class here*.

7.1.2 Transforming rasters using `RasterTransformers`

RasterSources accept a list of *RasterTransformers*, all of which are automatically applied (in the order specified) to each chip sampled from that *RasterSource*.

Below we'll look at two such `RasterTransformer`s:

- *MinMaxTransformer*
- *StatsTransformer*

But Raster Vision also provides the following:

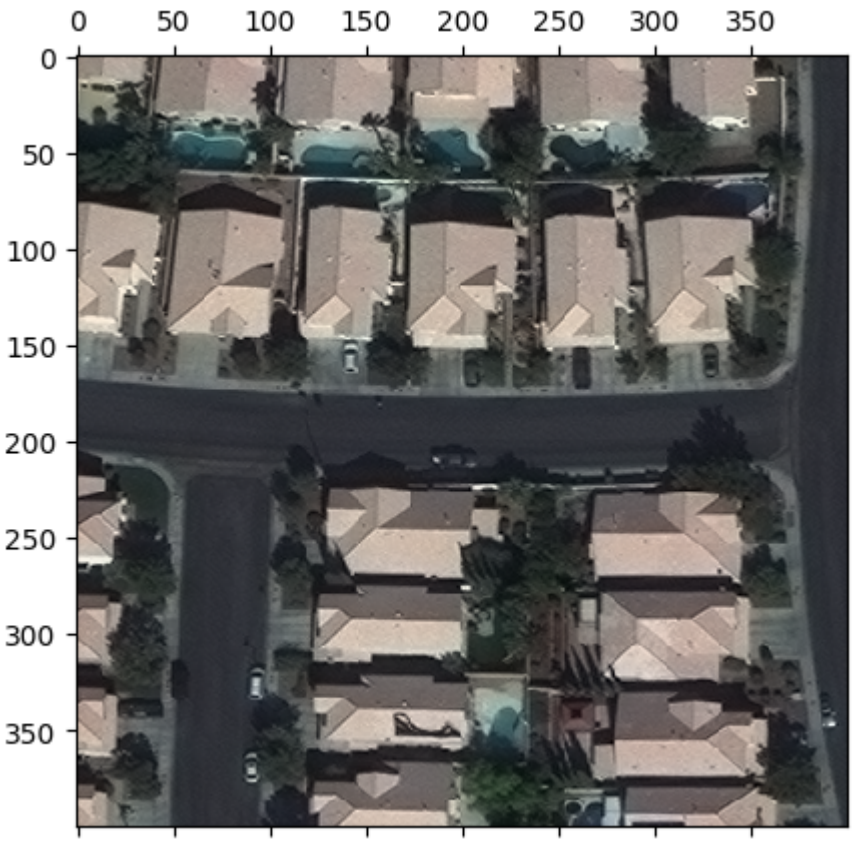
- *CastTransformer*: type-cast chip.
- *NanTransformer*: map NaN values to another value.
- *ReclassTransformer*: map values to other values using a given mapping; most useful for modifying class IDs in semantic segmentation labels.
- *RGBClassTransformer*: if your semantic segmentation labels are in the form of an RGB raster with different colors representing different classes, use this to map them to class IDs.

MinMaxTransformer

Above, we manually min-max normalized the `uint16` chip to 0-1. If we wanted this normalization to be automatically applied to all chips sampled from a `RasterSource`, we can simply attach a *MinMaxTransformer* to that `RasterSource`.

[8]:
```
from rastervision.core.data import RasterioSource, MinMaxTransformer

img_uri = 's3://azavea-research-public-data/raster-vision/examples/spacenet/RGB-
↪PanSharpen_AOI_2_Vegas_img205.tif'
raster_source = RasterioSource(img_uri, allow_streaming=True)

raster_source_normalized = RasterioSource(
    img_uri,
    allow_streaming=True,
    raster_transformers=[MinMaxTransformer()])
```

[9]:
```
from matplotlib import pyplot as plt

chip = raster_source_normalized[:400, :400]

fig, ax = plt.subplots(figsize=(5, 5))
ax.matshow(chip)
plt.show()
```



nbsphinx-code-borderwhite

StatsTransformer

Another useful `RasterTransformer` is the *StatsTransformer*.

Unlike *MinMaxTransformer*, the *StatsTransformer* is able to deal with outlier values. It works by using channel means and standard deviations to convert values to z-scores and then clipping them to some number of standard deviations before scaling to 0-255 and converting to `uint8`.

We can create and use one like so:

[10]:
```
from rastervision.core import RasterStats
from rastervision.core.data import RasterioSource, StatsTransformer

img_uri = 's3://azavea-research-public-data/raster-vision/examples/spacenet/RGB-
↪PanSharpen_AOI_2_Vegas_img205.tif'
raster_source = RasterioSource(img_uri, allow_streaming=True)

stats_transformer = StatsTransformer.from_raster_sources(
    raster_sources=[raster_source],
    max_stds=3)

raster_source_normalized = RasterioSource(
    img_uri,
```

(continues on next page)

(continued from previous page)

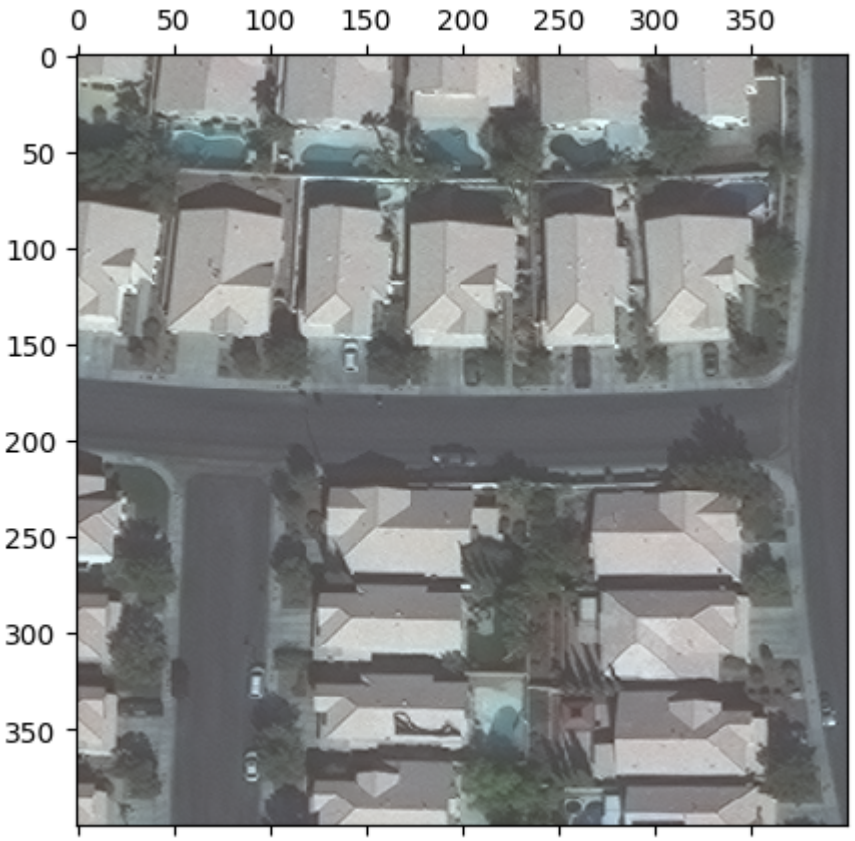
```
    allow_streaming=True,
    raster_transformers=[stats_transformer])
```

```
Analyzing chips: 0it [00:00, ?it/s]
```

[11]:
```
stats_transformer.means, stats_transformer.stds
```

[11]:
```
(array([[[577.23597778, 716.4319    , 519.44995556]]]),
 array([[[351.254233  , 342.98780916, 200.60693   ]]]))
```

[12]:
```
from matplotlib import pyplot as plt

chip_normalized = raster_source_normalized[:400, :400]

fig, ax = plt.subplots(figsize=(5, 5))
ax.matshow(chip_normalized)
plt.show()
```



nbsphinx-code-borderwhite

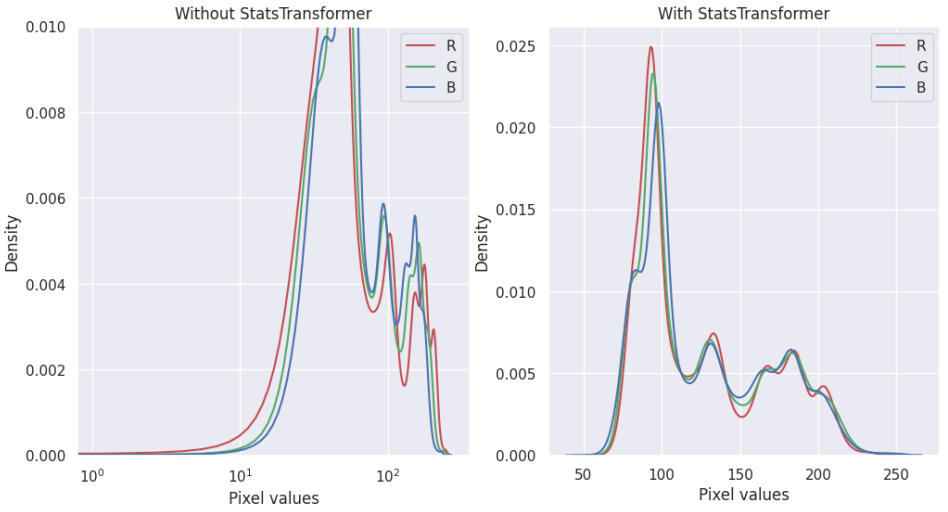
To get a closer look at StatsTransformer's work, we can visualize each band's pixel intensity distributions with and without it:

[13]:
```
import seaborn as sns
sns.reset_defaults()
sns.set_theme()
```

(continues on next page)

(continued from previous page)

```
band_names = 'RGB'

fig, (ax_l, ax_r) = plt.subplots(1, 2, squeeze=True, figsize=(12, 6))

# left
for i in range(chip.shape[-1]):
    sns.kdeplot(chip[..., i].flat, ax=ax_l, c=band_names[i].lower(), label=band_names[i])
ax_l.set_xscale('log')
ax_l.set_ylim((0, 0.01))
ax_l.set_xlabel('Pixel values')
ax_l.legend()
ax_l.set_title('Without StatsTransformer')

# right
for i in range(chip_normalized.shape[-1]):
    sns.kdeplot(
        chip_normalized[..., i].flat, ax=ax_r, c=band_names[i].lower(), label=band_
↪names[i])
ax_r.set_xlabel('Pixel values')
ax_r.legend()
ax_r.set_title('With StatsTransformer')

plt.show()
sns.reset_defaults()
```



nbsphinx-code-borderwhite

7.1.3 Subsetting and reordering bands

Dealing with multi-band imagery is a commonn use case in the GIS domain. All parts of Raster Vision work seamlessly with arbitrary number of bands.

The following example shows how we can use `RasterioSource` to sample 6 specific channels from a 191-band hyperspectral image.

Data from: https://engineering.purdue.edu/~biehl/MultiSpec/hyperspectral.html

[14]:
```
!wget "http://cobweb.ecn.purdue.edu/~biehl/Hyperspectral_Project.zip"
!apt-get install unzip
!unzip "Hyperspectral_Project.zip"
```

There are 2 ways to do this. We can either instantiate a `RasterioSource` like normal and sample the subset of bands using array indexing…

[15]:
```
from rastervision.core.data import RasterioSource, MinMaxTransformer

img_uri = 'Hyperspectral_Project/dc.tif'
raster_source_hsi = RasterioSource(img_uri)

raster_source_hsi.shape, raster_source_hsi.dtype
```

[15]:
```
((1280, 307, 191), dtype('int16'))
```

[16]:
```
chip = raster_source_hsi[:100, :100, [10, 30, 50, 70, 110, 130]]
chip.shape
```

[16]:
```
(100, 100, 6)
```

… Or we can specify the `channel_order` param while instantiating the `RasterioSource`, which will automatically restrict all sampled chips to these bands.

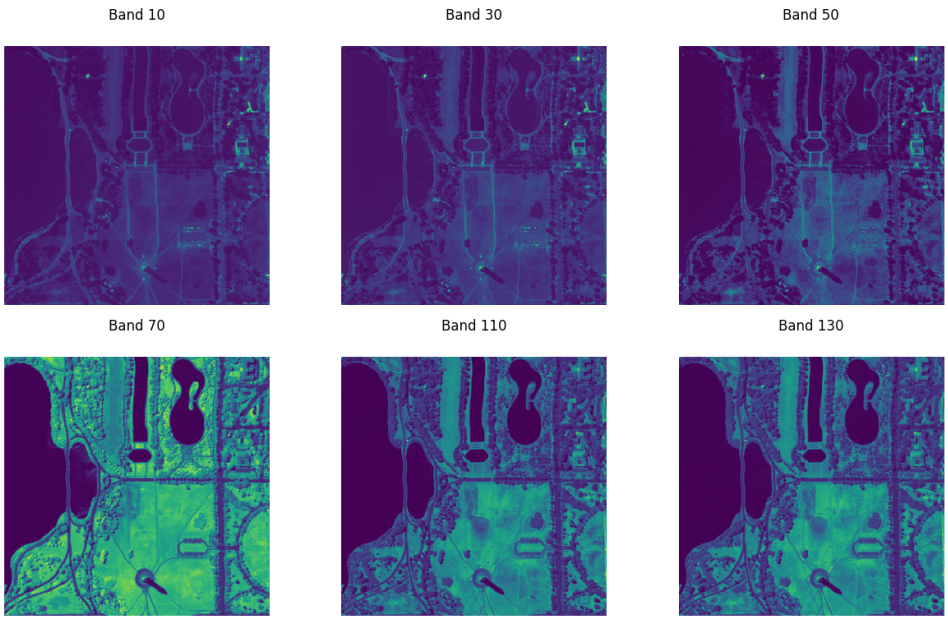
[17]:
```
raster_source_hsi = RasterioSource(
    img_uri,
    channel_order=[10, 30, 50, 70, 110, 130],
    raster_transformers=[MinMaxTransformer()])

raster_source_hsi.shape, raster_source_hsi.dtype
```

[17]:
```
((1280, 307, 6), dtype('uint8'))
```

[18]:
```
chip = raster_source_hsi[200:500, :]
chip.shape
```

[18]:
```
(300, 307, 6)
```

[19]:
```
fig, axs = plt.subplots(2, 3, squeeze=True, figsize=(15, 9))

for i, (ax, ch) in enumerate(zip(axs.flat, raster_source_hsi.channel_order)):
    ax.matshow(chip[..., i])
    ax.set_title(f'Band {ch}')
    ax.axis('off')
plt.show()
```



nbsphinx-code-borderwhite

7.1.4 Combining bands from different files using `MultiRasterSource`

Another common use case is combining bands from multiple sources. This can done using a *`MultiRasterSource`*.

The following example combines RGB, SWIR, and SAR bands into a single 8-band `RasterSource`.

Data from: Cloud to Street - Microsoft flood dataset

[20]:
```
uri_seninel_2_rgb = 'https://radiantearth.blob.core.windows.net/mlhub/c2smsfloods/chips/
↪e7d1917e-c069-45cf-a392-42e24aa2f4ac/s2/S2A_MSIL1C_20201022T100051_N0209_R122_T33UXP_
↪20201022T111023_07680-00512/RGB.png'
uri_seninel_2_swir = 'https://radiantearth.blob.core.windows.net/mlhub/c2smsfloods/chips/
↪e7d1917e-c069-45cf-a392-42e24aa2f4ac/s2/S2A_MSIL1C_20201022T100051_N0209_R122_T33UXP_
↪20201022T111023_07680-00512/SWIR.png'
uri_seninel_1_vh = 'https://radiantearth.blob.core.windows.net/mlhub/c2smsfloods/chips/
↪e7d1917e-c069-45cf-a392-42e24aa2f4ac/s1/S1B_IW_GRDH_1SDV_20201020T164222_
↪20201020T164247_023899_02D6C4_35D8_07680-00512/VH.tif'
uri_seninel_1_vv = 'https://radiantearth.blob.core.windows.net/mlhub/c2smsfloods/chips/
↪e7d1917e-c069-45cf-a392-42e24aa2f4ac/s1/S1B_IW_GRDH_1SDV_20201020T164222_
↪20201020T164247_023899_02D6C4_35D8_07680-00512/VV.tif'
```

First, we create `RasterSource`s for each individual source.

[21]:
```
from rastervision.core.data import RasterioSource, MultiRasterSource, MinMaxTransformer

rs_sentinel_2_rgb = RasterioSource(uri_seninel_2_rgb, allow_streaming=True)
rs_sentinel_2_swir = RasterioSource(uri_seninel_2_swir, allow_streaming=True)

rs_seninel_1_vh = RasterioSource(uri_seninel_1_vh, allow_streaming=True, raster_
↪transformers=[MinMaxTransformer()])
rs_seninel_1_vv = RasterioSource(uri_seninel_1_vv, allow_streaming=True, raster_
```

(continues on next page)

(continued from previous page)

```
↪transformers=[MinMaxTransformer()])

print('rs_sentinel_2_rgb', rs_sentinel_2_rgb.shape, rs_sentinel_2_rgb.dtype)
print('rs_sentinel_2_swir', rs_sentinel_2_swir.shape, rs_sentinel_2_swir.dtype)
print('rs_seninel_1_vh', rs_seninel_1_vh.shape, rs_seninel_1_vh.dtype)
print('rs_seninel_1_vv', rs_seninel_1_vv.shape, rs_seninel_1_vv.dtype)
```

```
rs_sentinel_2_rgb (512, 512, 3) uint8
rs_sentinel_2_swir (512, 512, 3) uint8
rs_seninel_1_vh (512, 512, 1) uint8
rs_seninel_1_vv (512, 512, 1) uint8
```

Next, we combine them into a *MultiRasterSource*.

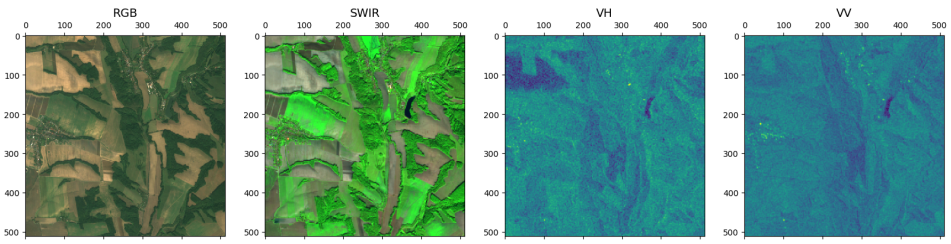
[22]:
```
raster_sources = [
    rs_sentinel_2_rgb,
    rs_sentinel_2_swir,
    rs_seninel_1_vh,
    rs_seninel_1_vv
]

raster_source_multi = MultiRasterSource(
    raster_sources=raster_sources, primary_source_idx=0)

raster_source_multi.shape, raster_source_multi.dtype
```

[22]:
```
((512, 512, 8), dtype('uint8'))
```

[23]:
```
chip = raster_source_multi[:, :]
chip.shape
```

[23]:
```
(512, 512, 8)
```

[24]:
```
from matplotlib import pyplot as plt

fig, axs = plt.subplots(1, 4, figsize=(20, 5))
(ax_rgb, ax_swir, ax_vh, ax_vv) = axs.flat

ax_rgb.matshow(chip[..., :3])
ax_rgb.set_title('RGB', fontsize=14)
ax_swir.matshow(chip[..., 3:6])
ax_swir.set_title('SWIR', fontsize=14)
ax_vh.matshow(chip[..., 6])
ax_vh.set_title('VH', fontsize=14)
ax_vv.matshow(chip[..., 7])
ax_vv.set_title('VV', fontsize=14)

plt.show()
```

nbsphinx-code-borderwhite 

7.1.5 Cropping the extent

Sometimes you might want to crop a `RasterSource`. For example, if you want to use one part of it for training and another for validation.

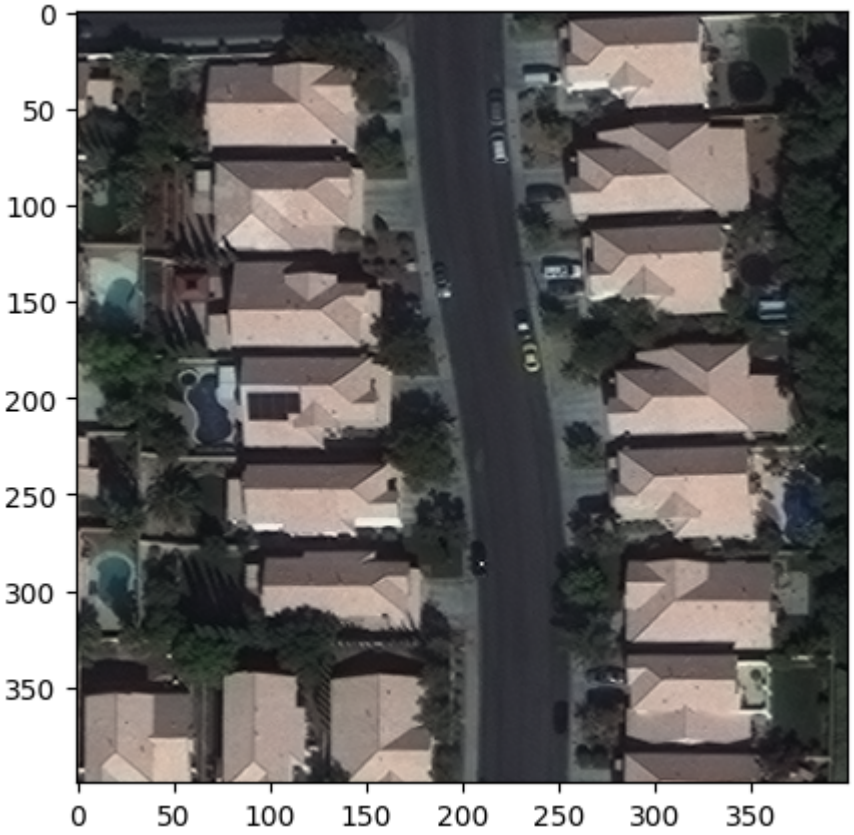
The following example shows how we can crop out the top 200 pixels and left 200 pixels of the full raster.

[25]:
```
from rastervision.core.data import RasterioSource, MinMaxTransformer
from rastervision.core.box import Box

img_uri = 's3://azavea-research-public-data/raster-vision/examples/spacenet/RGB-
↪PanSharpen_AOI_2_Vegas_img205.tif'
raster_source_cropped = RasterioSource(
    img_uri,
    allow_streaming=True,
    extent=Box(200, 200, 650, 650),
    raster_transformers=[MinMaxTransformer()])

raster_source_cropped.shape, raster_source_cropped.dtype
```

[25]:
```
((450, 450, 3), dtype('uint8'))
```

[26]:
```
from rastervision.core.box import Box
from matplotlib import pyplot as plt

chip = raster_source_cropped[:400, :400]
print(chip.shape)

fig, ax = plt.subplots(figsize=(5, 5))
ax.imshow(chip)
plt.show()
```

```
(400, 400, 3)
```



nbsphinx-code-borderwhite

7.2 Reading vector data

The *VectorSource* is Raster Vision's abstraction for reading from a source of vector data.

Besides reading the data, they can also convert geometries from map-coordinates to pixel-coordinates and perform some data cleaning such as removing empty geometries and splitting apart multi-part geometries (e.g. `MultiPolygon` etc.).

One concrete implementation of it is the *GeoJSONVectorSource* which can read vector data from a GeoJSON file.

[1]:
```
from rastervision.core.data import GeoJSONVectorSource, RasterioCRSTransformer

img_uri = 's3://azavea-research-public-data/raster-vision/examples/spacenet/RGB-
↪PanSharpen_AOI_2_Vegas_img205.tif'
label_uri = 's3://azavea-research-public-data/raster-vision/examples/spacenet/buildings_
↪AOI_2_Vegas_img205.geojson'

crs_transformer = RasterioCRSTransformer.from_uri(img_uri)
vector_source = GeoJSONVectorSource(
    label_uri, crs_transformer, ignore_crs_field=True)
```

We can read data from a *VectorSource* in three different formats:

1. as GeoJSON dict (*get_geojson()*)

2. as Shapely geoms (*get_geoms()*)
3. as a GeoPandas GeoDataFrame (*get_dataframe()*)

Each of these is shown in the following cells.

7.2.1 .get_geojson()

[2]:
```
geojson = vector_source.get_geojson()
geojson['features'][:3]
```

```
2022-09-13 12:06:41:rastervision.pipeline.file_system.utils: INFO - Using cached file /
↪opt/data/tmp/cache/s3/azavea-research-public-data/raster-vision/examples/spacenet/
↪buildings_AOI_2_Vegas_img205.geojson.
```

[2]:
```
[{'type': 'Feature',
  'geometry': {'type': 'Polygon',
   'coordinates': (((552.0, 587.0),
     (485.0, 587.0),
     (485.0, 604.0),
     (482.0, 604.0),
     (482.0, 621.0),
     (503.0, 621.0),
     (503.0, 624.0),
     (515.0, 624.0),
     (515.0, 633.0),
     (552.0, 633.0),
     (552.0, 587.0)),)},
  'properties': {'OBJECTID': 0,
   'FID_VEGAS_': 0,
   'Id': 0,
   'FID_Vegas': 0,
   'Name': 'None',
   'AREA': 0.0,
   'Shape_Leng': 0.0,
   'Shape_Le_1': 0.0,
   'SISL': 0.0,
   'OBJECTID_1': 0,
   'Shape_Le_2': 0.0,
   'Shape_Le_3': 0.000625,
   'Shape_Area': 0.0,
   'partialBuilding': 0.0,
   'partialDec': 1.0}},
 {'type': 'Feature',
  'geometry': {'type': 'Polygon',
   'coordinates': (((561.0, 533.0),
     (562.0, 487.0),
     (486.0, 486.0),
     (485.0, 527.0),
     (541.0, 528.0),
     (541.0, 532.0),
     (561.0, 533.0)),)},
```

(continues on next page)

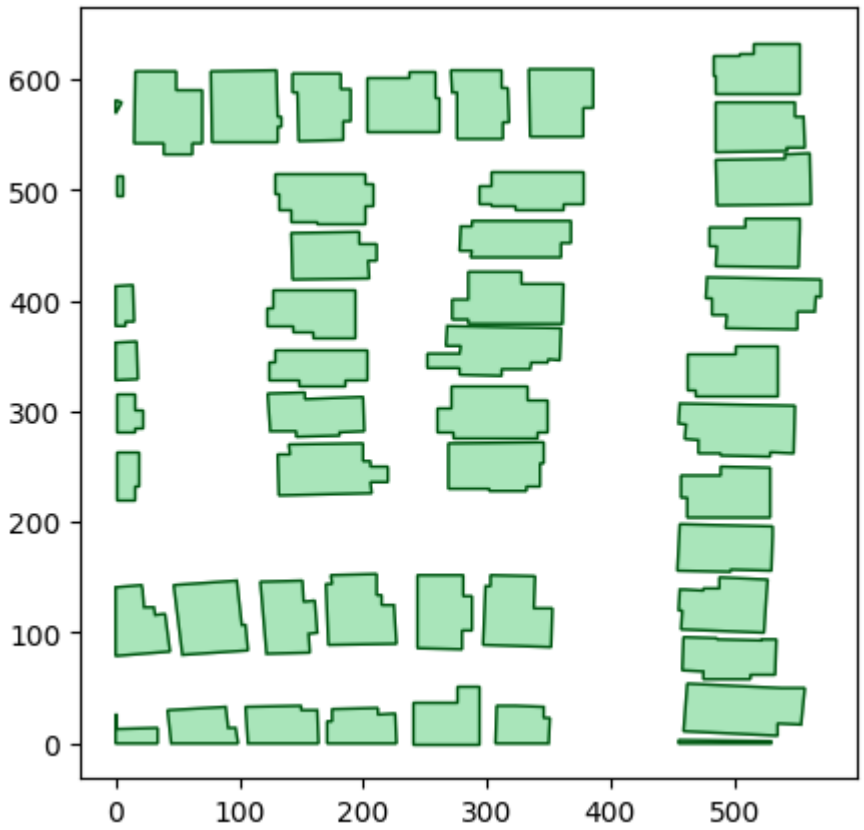
(continued from previous page)


```
  'properties': {'OBJECTID': 0,
   'FID_VEGAS_': 0,
   'Id': 0,
   'FID_Vegas': 0,
   'Name': 'None',
   'AREA': 0.0,
   'Shape_Leng': 0.0,
   'Shape_Le_1': 0.0,
   'SISL': 0.0,
   'OBJECTID_1': 0,
   'Shape_Le_2': 0.0,
   'Shape_Le_3': 0.000658,
   'Shape_Area': 0.0,
   'partialBuilding': 0.0,
   'partialDec': 1.0}},
 {'type': 'Feature',
  'geometry': {'type': 'Polygon',
   'coordinates': (((553.0, 465.0),
     (552.0, 430.0),
     (485.0, 431.0),
     (486.0, 449.0),
     (482.0, 449.0),
     (480.0, 449.0),
     (480.0, 466.0),
     (482.0, 466.0),
     (509.0, 466.0),
     (509.0, 474.0),
     (553.0, 474.0),
     (553.0, 465.0)),)},
  'properties': {'OBJECTID': 0,
   'FID_VEGAS_': 0,
   'Id': 0,
   'FID_Vegas': 0,
   'Name': 'None',
   'AREA': 0.0,
   'Shape_Leng': 0.0,
   'Shape_Le_1': 0.0,
   'SISL': 0.0,
   'OBJECTID_1': 0,
   'Shape_Le_2': 0.0,
   'Shape_Le_3': 0.000627,
   'Shape_Area': 0.0,
   'partialBuilding': 0.0,
   'partialDec': 1.0}}]
```

7.2.2 .get_geoms()

[3]:
```
def plot_geoms(geoms: list, title=''):
    from matplotlib import pyplot as plt
    from matplotlib import patches as patches
    import numpy as np

    fig, ax = plt.subplots(figsize=(5, 5))
    for g in geoms:
        if g.geom_type == 'Polygon':
            xy = np.array(g.exterior)
            patch = patches.Polygon(xy, color='#55cc77', alpha=0.5)
            ax.add_patch(patch)
            patch = patches.Polygon(xy, edgecolor='#005511', fill=None, alpha=1)
            ax.add_patch(patch)
        elif g.geom_type == 'LineString':
            xy = np.array(g.buffer(1).exterior)
            patch = patches.Polygon(xy, color='#005511', alpha=0.8)
            ax.add_patch(patch)
        else:
            raise NotImplementedError()
    ax.set_title(title, fontsize=14)
    ax.autoscale()
    plt.show()
```

[4]:
```
geoms = vector_source.get_geoms()
plot_geoms(geoms)
```



nbsphinx-code-borderwhite

7.2.3 .get_dataframe()

[5]:
```
df = vector_source.get_dataframe()
df.head()
```

[5]:
```
                                            geometry  OBJECTID  FID_VEGAS_  \
0  POLYGON ((552.000 587.000, 485.000 587.000, 48...         0           0
1  POLYGON ((561.000 533.000, 562.000 487.000, 48...         0           0
2  POLYGON ((553.000 465.000, 552.000 430.000, 48...         0           0
3  POLYGON ((551.000 374.000, 493.000 375.000, 49...         0           0
4  POLYGON ((535.000 315.000, 468.000 315.000, 46...         0           0

   Id  FID_Vegas  Name  AREA  Shape_Leng  Shape_Le_1  SISL  OBJECTID_1  \
0   0          0  None   0.0         0.0         0.0   0.0           0
1   0          0  None   0.0         0.0         0.0   0.0           0
2   0          0  None   0.0         0.0         0.0   0.0           0
3   0          0  None   0.0         0.0         0.0   0.0           0
4   0          0  None   0.0         0.0         0.0   0.0           0

   Shape_Le_2  Shape_Le_3  Shape_Area  partialBuilding  partialDec
0         0.0    0.000625         0.0              0.0         1.0
1         0.0    0.000658         0.0              0.0         1.0
```

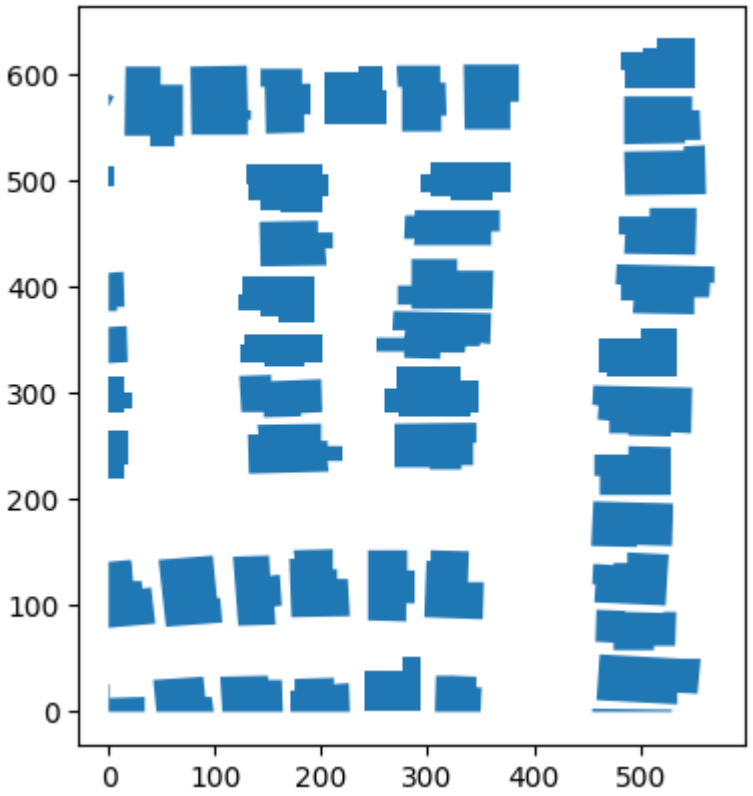
(continues on next page)

(continued from previous page)

```
2          0.0     0.000627          0.0           0.0          1.0
3          0.0     0.000744          0.0           0.0          1.0
4          0.0     0.000634          0.0           0.0          1.0
```

[6]:
```
df.plot()
```

[6]:
```
<AxesSubplot:>
```



nbsphinx-code-borderwhite

Transforming vector data using `VectorTransformers`

Just like we can tansform rasters by specifying a series of *`RasterTransformers`*, we can specify *`VectorTransformers`* to transform vector data.

7.2.4 Inferring class IDs for polygons

One very important `VectorTransformer` is the *`ClassInferenceTransformer`*.

When using vector data in machine learning, it is important that each polygon be labeled with an appropriate class ID. But often, your data will not have this property stored in the GeoJSON file.

The `ClassInferenceTransformer` can automatically infer and attach a `class_id` to each polygon read from the `VectorSource`. It can 1. Assign the same `class_id` to all the polygons (a very common use case). 2. Map class names to `class_id`s given a mapping. 3. Use a MapBox-style filter (see https://docs.mapbox.com/mapbox-gl-js/style-spec/other/#other-filter).

The example below shows how to use the first of the above methods.

[7]:
```
from rastervision.core.data import (
    GeoJSONVectorSource, RasterioCRSTransformer,
    RasterizedSource, ClassInferenceTransformer)

img_uri = 's3://azavea-research-public-data/raster-vision/examples/spacenet/RGB-
↪PanSharpen_AOI_2_Vegas_img205.tif'
label_uri = 's3://azavea-research-public-data/raster-vision/examples/spacenet/buildings_
↪AOI_2_Vegas_img205.geojson'

crs_transformer = RasterioCRSTransformer.from_uri(img_uri)
vector_source = GeoJSONVectorSource(
    label_uri,
    crs_transformer,
    ignore_crs_field=True,
    vector_transformers=[ClassInferenceTransformer(default_class_id=1)])
```

[8]:
```
df = vector_source.get_dataframe()
df[['geometry', 'class_id']].head()
```

```
2022-09-13 12:06:42:rastervision.pipeline.file_system.utils: INFO - Using cached file /
↪opt/data/tmp/cache/s3/azavea-research-public-data/raster-vision/examples/spacenet/
↪buildings_AOI_2_Vegas_img205.geojson.
```

[8]:

	geometry	class_id
0	POLYGON ((552.000 587.000, 485.000 587.000, 48...	1
1	POLYGON ((561.000 533.000, 562.000 487.000, 48...	1
2	POLYGON ((553.000 465.000, 552.000 430.000, 48...	1
3	POLYGON ((551.000 374.000, 493.000 375.000, 49...	1
4	POLYGON ((535.000 315.000, 468.000 315.000, 46...	1

7.2.5 Buffering Point and LineString geometries into polygons

`Point` and `LineString` geometries are not directly useable if doing, say, semantic segmentation. The cells below show an example of converting road geometries (given in the form of ``LineString``s) into polygons using the *`BufferTransformer`*.

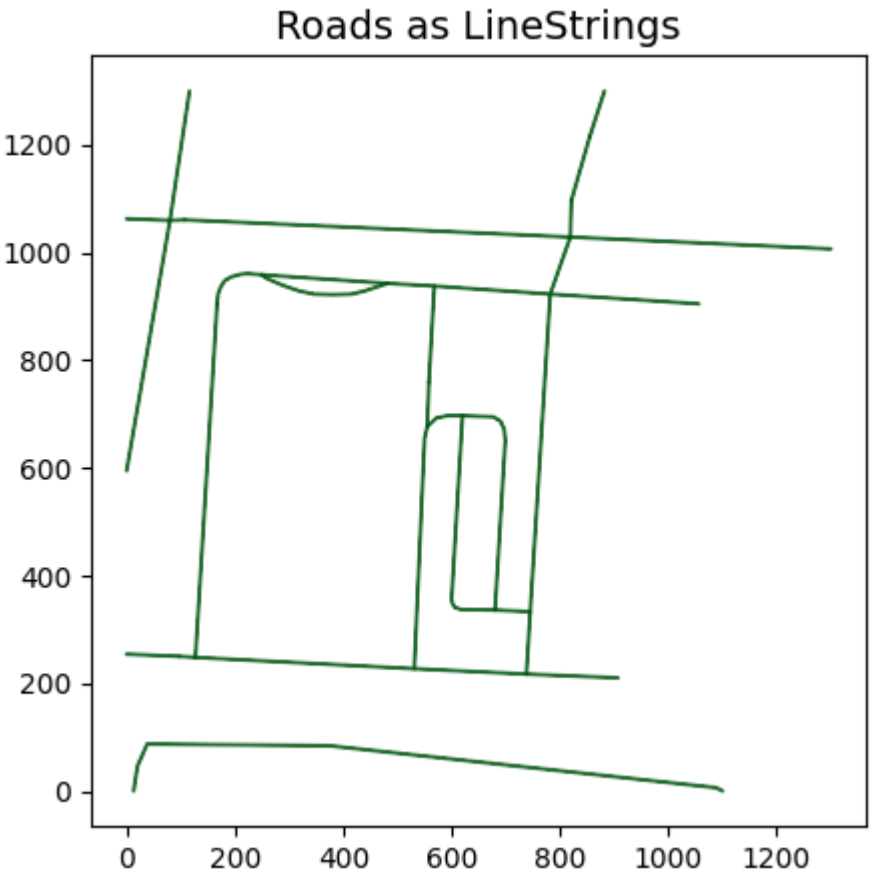
Data source: https://spacenet.ai/spacenet-roads-dataset/

[9]:
```
from rastervision.core.data import (
    GeoJSONVectorSource, RasterioCRSTransformer,
    RasterizedSource, BufferTransformer)

img_uri = 's3://spacenet-dataset/spacenet/SN3_roads/train/AOI_4_Shanghai/PS-RGB/SN3_
↪roads_train_AOI_4_Shanghai_PS-RGB_img999.tif'
label_uri = 's3://spacenet-dataset/spacenet/SN3_roads/train/AOI_4_Shanghai/geojson_roads/
↪SN3_roads_train_AOI_4_Shanghai_geojson_roads_img999.geojson'

crs_transformer = RasterioCRSTransformer.from_uri(img_uri)
```

[10]:
```
def plot_geoms(geoms: list, title=''):
    from matplotlib import pyplot as plt
    from matplotlib import patches as patches
    import numpy as np

    fig, ax = plt.subplots(figsize=(5, 5))
    for g in geoms:
        if g.geom_type == 'Polygon':
            xy = np.array(g.exterior)
            patch = patches.Polygon(xy, color='#55cc77', alpha=0.5)
            ax.add_patch(patch)
            patch = patches.Polygon(xy, edgecolor='#005511', fill=None, alpha=1)
            ax.add_patch(patch)
        elif g.geom_type == 'LineString':
            xy = np.array(g.buffer(1).exterior)
            patch = patches.Polygon(xy, color='#005511', alpha=0.8)
            ax.add_patch(patch)
        else:
            raise NotImplementedError()
    ax.set_title(title, fontsize=14)
    ax.autoscale()
    plt.show()
```

[11]:
```
vector_source = GeoJSONVectorSource(
    label_uri,
    crs_transformer,
    ignore_crs_field=True)
plot_geoms(vector_source.get_geoms(), title='Roads as LineStrings')
```

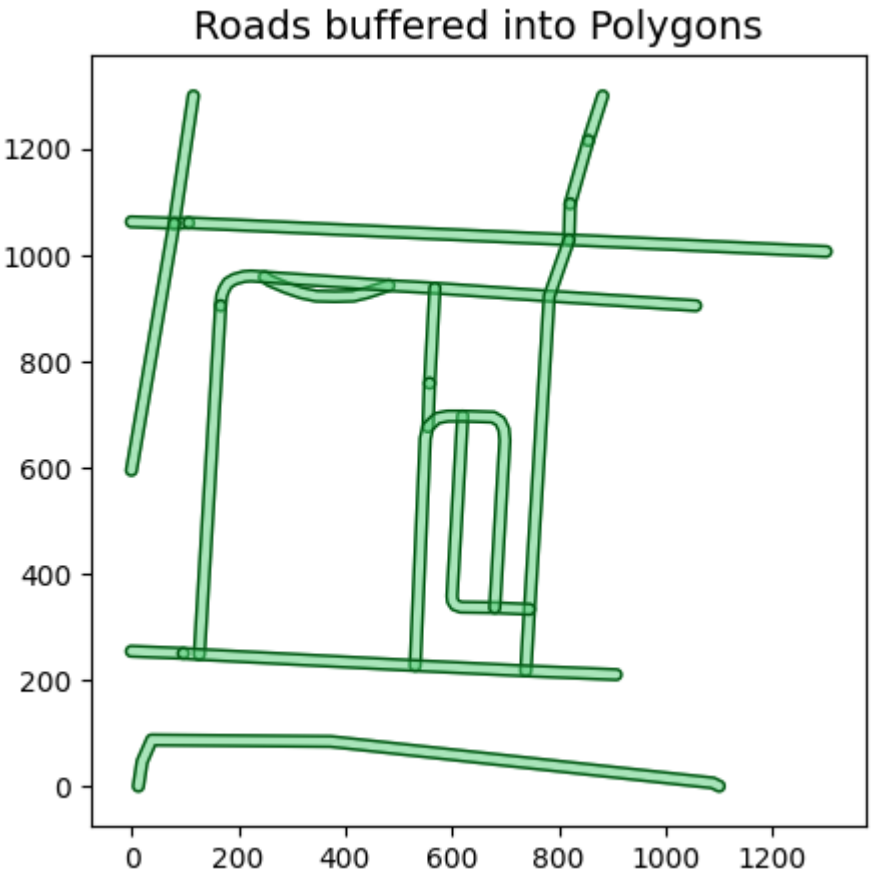
```
2022-09-13 12:06:48:rastervision.pipeline.file_system.utils: INFO - Using cached file /
↪opt/data/tmp/cache/s3/spacenet-dataset/spacenet/SN3_roads/train/AOI_4_Shanghai/geojson_
↪roads/SN3_roads_train_AOI_4_Shanghai_geojson_roads_img999.geojson.
```



nbsphinx-code-borderwhite

[12]:
```
vector_source_buffered = GeoJSONVectorSource(
    label_uri,
    crs_transformer,
    ignore_crs_field=True,
    vector_transformers=[BufferTransformer(geom_type='LineString', default_buf=10)])

plot_geoms(vector_source_buffered.get_geoms(), title='Roads buffered into Polygons')
```

```
2022-09-13 12:06:48:rastervision.pipeline.file_system.utils: INFO - Using cached file /
↪opt/data/tmp/cache/s3/spacenet-dataset/spacenet/SN3_roads/train/AOI_4_Shanghai/geojson_
↪roads/SN3_roads_train_AOI_4_Shanghai_geojson_roads_img999.geojson.
```



nbsphinx-code-borderwhite

Rasterizing vector data using `RasterizedSource`

Suppose we have semantic segmentation labels in the form of polygons. To use them for training, we will first need to convert them into rasters. Raster Vision allows accomplishing this using the *`RasterizedSource`* class.

The `RasterizedSource` is a `RasterSource` that reads data from a `VectorSource` (rather than an image file) and then converts it into rasters. It can be indexed like any other `RasterSource`.

[13]:
```
from rastervision.core.data import (
    GeoJSONVectorSource, RasterioCRSTransformer,
    RasterizedSource, ClassInferenceTransformer)

img_uri = 's3://azavea-research-public-data/raster-vision/examples/spacenet/RGB-
↪PanSharpen_AOI_2_Vegas_img205.tif'
label_uri = 's3://azavea-research-public-data/raster-vision/examples/spacenet/buildings_
↪AOI_2_Vegas_img205.geojson'

crs_transformer = RasterioCRSTransformer.from_uri(img_uri)
vector_source = GeoJSONVectorSource(
    label_uri,
    crs_transformer,
    ignore_crs_field=True,
    vector_transformers=[ClassInferenceTransformer(default_class_id=1)])
```

(continues on next page)

(continued from previous page)

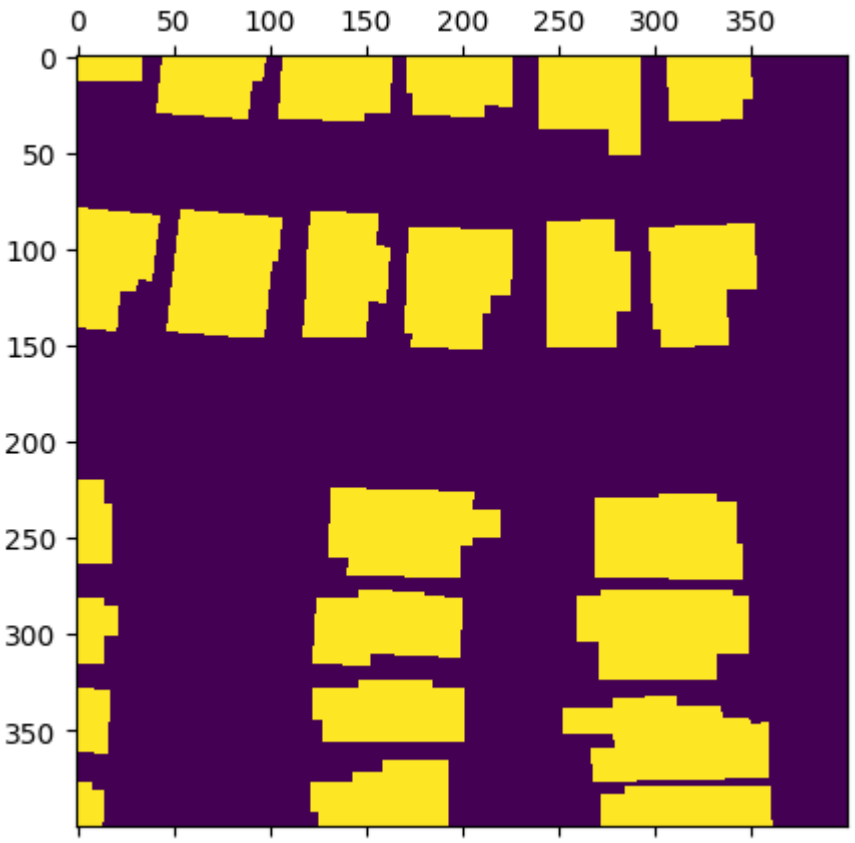
```
rasterized_source = RasterizedSource(
    vector_source,
    background_class_id=0,
    # Normally we'd pass in the RasterSource's extent, but we don't have that here.
    extent=vector_source.extent)
```

```
2022-09-13 12:06:49:rastervision.pipeline.file_system.utils: INFO - Using cached file /
↪opt/data/tmp/cache/s3/azavea-research-public-data/raster-vision/examples/spacenet/
↪buildings_AOI_2_Vegas_img205.geojson.
```

[14]:
```
chip = rasterized_source[:400, :400]
chip.shape
```

[14]:
```
(400, 400, 1)
```

[15]:
```
from matplotlib import pyplot as plt

fig, ax = plt.subplots(figsize=(5, 5))
ax.matshow(chip)
plt.show()
```



nbsphinx-code-borderwhite

7.3 Reading labels

7.3.1 ClassConfig

Before we can work with labels, we first want to define what our target classes are – their names, their IDs, and possibly, their colors (to use for visualization).

Raster Vision makes all this simple to do using the handy *ClassConfig* class.

[1]:
```
from rastervision.core.data import ClassConfig

class_config = ClassConfig(names=['background', 'foreground'])
class_config
```

[1]:
```
ClassConfig(names=['background', 'foreground'], colors=[(244, 153, 180), (1, 235, 86)],␣
↪null_class=None)
```

(Note how a unique color was generated for each class.)

The numeric ID of each class is its index in the *names* list.

We can query the ID of a class like so:

[2]:
```
print('background', class_config.get_class_id('background'))
print('foreground', class_config.get_class_id('foreground'))
```

```
background 0
foreground 1
```

In the example above, ClassConfig automatically generated a color for each class. We could have instead manually specified them. The colors can be any color-string recognized by PIL.

[3]:
```
class_config = ClassConfig(
    names=['background', 'foreground'],
    colors=['lightgray', 'darkred'])
class_config
```

[3]:
```
ClassConfig(names=['background', 'foreground'], colors=['lightgray', 'darkred'], null_
↪class=None)
```

The "null" class

Geospatial rasters can have NODATA pixels. This is relevant for semantic segmentation where you need to assign a label to every pixel.

You may either want to assign them a class of their own or consider them a part of a catch-all "background" class.

ClassConfig's name for this class is the "null class". Below are some ways that you can define it.

1. Designate one class as the "null" class.

[4]:
```
class_config = ClassConfig(names=['background', 'foreground'], null_class='background')
class_config
```

[4]:
```
ClassConfig(names=['background', 'foreground'], colors=[(235, 58, 174), (150, 29, 74)],␣
↪null_class='background')
```

2. Specify a class called "null".

[5]:
```
class_config = ClassConfig(names=['null', 'foreground'])
class_config
```

[5]:
```
ClassConfig(names=['null', 'foreground'], colors=[(1, 53, 240), (81, 165, 180)], null_
↪class='null')
```

3. Call `.ensure_null_class()` to automatically add an additional "null" class.

[6]:
```
class_config = ClassConfig(names=['background', 'foreground'])
class_config.ensure_null_class()
class_config
```

[6]:
```
ClassConfig(names=['background', 'foreground', 'null'], colors=[(58, 135, 29), (211, 60,␣
↪215), 'black'], null_class='null')
```

Normalized colors

Another nifty functionality is the *color_triples* property that returns the colors in a normalized form that can be directly used with `matplotlib`.

The example below shows how we can easily create a color-map from our class colors.

[7]:
```
from matplotlib.colors import ListedColormap

class_config = ClassConfig(names=['a', 'b', 'c', 'd', 'e', 'f'])
cmap = ListedColormap(class_config.color_triples)
cmap
```

nbsphinx-code-borderwidth

7.3.2 LabelSource

While *RasterSources* and *VectorSources* allow us to read raw data, the *LabelSources* take this data and convert them into a form suitable for machine learning.

We have 3 kinds of pre-defined *LabelSources* – one for each of the 3 main computer vision tasks: semantic segmentation, object detection, and chip classification.

Semantic Segmentation - SemanticSegmentationLabelSource

SemanticSegmentationLabelSource is perhaps the simplest of *LabelSources*. Since semantic segmentation labels are rasters themselves, *SemanticSegmentationLabelSource* takes in a *RasterSource* and allows querying chips from it using array-indexing or *Box's*.

The main added service it provides is ensuring that if a chip overflows the extent, the overflowing pixels are assigned the ID of the "null class".

The example below shows how to create a SemanticSegmentationLabelSource from rasterized vector labels.

[8]:
```
img_uri = 's3://azavea-research-public-data/raster-vision/examples/spacenet/RGB-
↪PanSharpen_AOI_2_Vegas_img205.tif'
label_uri = 's3://azavea-research-public-data/raster-vision/examples/spacenet/buildings_
↪AOI_2_Vegas_img205.geojson'
```

Define ClassConfig:

[9]:
```
from rastervision.core.data import ClassConfig

class_config = ClassConfig(
    names=['background', 'building'],
    colors=['lightgray', 'darkred'],
    null_class='background')
```

Create

- a *RasterSource* to get the image extent and a *CRSTransformer*
- a *VectorSource* to read the vector labels
- a *RasterizedSource* to rasterize the *VectorSource*

[10]:
```
from rastervision.core.data import (
    ClassInferenceTransformer, GeoJSONVectorSource,
    RasterioSource, RasterizedSource)

img_raster_source = RasterioSource(img_uri, allow_streaming=True)

vector_source = GeoJSONVectorSource(
    label_uri,
    img_raster_source.crs_transformer,
    ignore_crs_field=True,
    vector_transformers=[
        ClassInferenceTransformer(
            default_class_id=class_config.get_class_id('building'))])

label_raster_source = RasterizedSource(
    vector_source,
    background_class_id=class_config.null_class_id,
    extent=img_raster_source.extent)
```

```
2022-09-13 12:08:55:rastervision.pipeline.file_system.utils: INFO - Using cached file /
↪opt/data/tmp/cache/s3/azavea-research-public-data/raster-vision/examples/spacenet/
↪buildings_AOI_2_Vegas_img205.geojson.
```

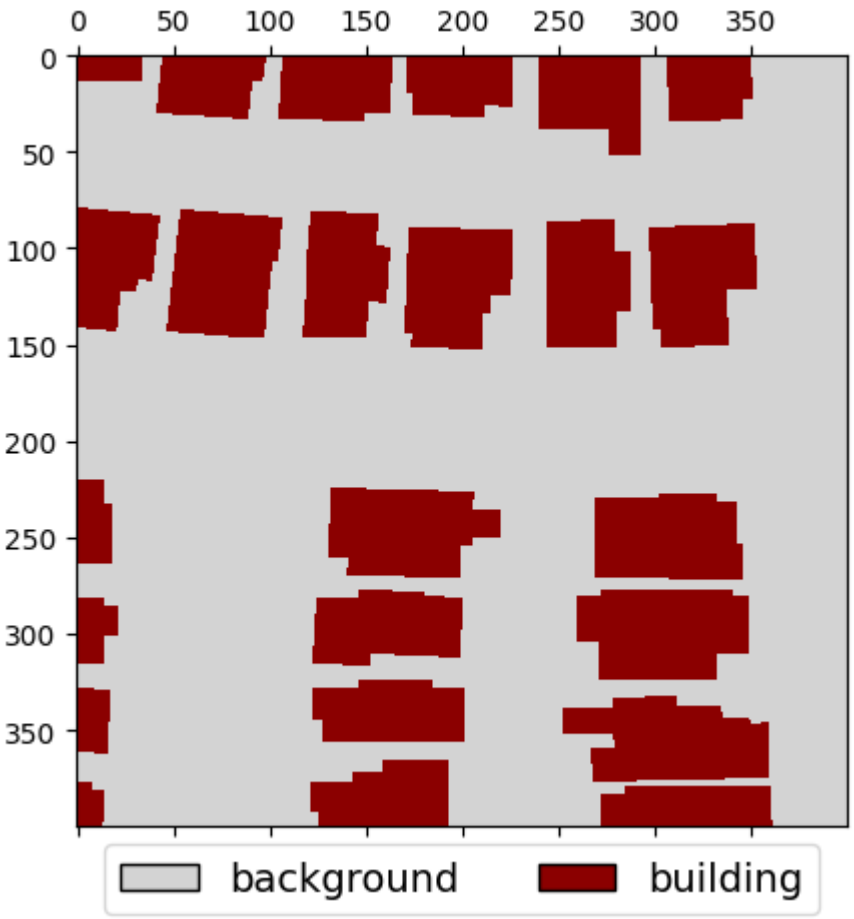
Create a SemanticSegmentationLabelSource from the RasterizedSource.

[11]:
```
from rastervision.core.data import SemanticSegmentationLabelSource

label_source = SemanticSegmentationLabelSource(
    label_raster_source, class_config=class_config)
```

We can sample label-chips like so:

[12]:
```
label_chip = label_source[:400, :400]

## equivalent to:
#
# from rastervision.core.box import Box
# label_chip = label_source.get_label_arr(Box(0, 0, 400, 400))
```

[13]:
```
from matplotlib import pyplot as plt
from matplotlib.colors import ListedColormap
from matplotlib import patches

fig, ax = plt.subplots(figsize=(5, 5))
cmap = ListedColormap(class_config.color_triples)
ax.matshow(label_chip, cmap=cmap)

legend_items = [
    patches.Patch(facecolor=cmap(i), edgecolor='black', label=cname)
    for i, cname in enumerate(class_config.names)]
ax.legend(
    handles=legend_items,
    ncol=len(class_config),
    loc='upper center',
    fontsize=14,
    bbox_to_anchor=(0.5, 0))
plt.show()
```



nbsphinx-code-borderwhite

Object Detection - `ObjectDetectionLabelSource`

The *`ObjectDetectionLabelSource`* allows querying all the label bounding boxes and their corresponding class IDs that fall inside a window. It also transforms the coordinates of the returned bounding boxes so that they represent points inside the window rather than the global extent.

The bounding-box-to-window matching behavior can be further controlled by specifying an `ioa_thresh` (intersection-over-area threshold for considering a bounding box a part of a window) and a `clip` flag which ensures that bounding boxes do not overflow the window.

Download (~500 KB) and unzip data:

[14]:
```
!wget "https://github.com/azavea/raster-vision-data/releases/download/v0.0.1/cowc-
↪potsdam-labels.zip"
!apt-get install unzip
!unzip "cowc-potsdam-labels.zip" -d "cowc-potsdam-labels/"
```

[15]:
```
img_uri = 's3://azavea-research-public-data/raster-vision/examples/tutorials-data/top_
↪potsdam_2_10_RGBIR.tif'
label_uri = 'cowc-potsdam-labels/labels/train/top_potsdam_2_10_RGBIR.json'
```

Create - a `RasterSource` to get the image extent and a `CRSTRansformer` - a `VectorSource` to read the vector labels - an `ObjectDetectionLabelSource` to convert data from `VectorSource` into object detection labels

[16]:
```
from rastervision.core.data import (
    ClassConfig, ClassInferenceTransformer, GeoJSONVectorSource,
    ObjectDetectionLabelSource, RasterioSource)

class_config = ClassConfig(names=['car'], colors=['red'])

raster_source = RasterioSource(img_uri, allow_streaming=True)

vector_source = GeoJSONVectorSource(
    label_uri,
    crs_transformer=raster_source.crs_transformer,
    ignore_crs_field=True,
    vector_transformers=[
        ClassInferenceTransformer(
            default_class_id=class_config.get_class_id('car'))])

label_source = ObjectDetectionLabelSource(
    vector_source, extent=raster_source.extent, ioa_thresh=0.2, clip=False)
```

We can sample label data like shown below. Here, we are querying a 1000x1000 chip downscaled by a factor of 2 from the raster source. Using the same index for the `ObjectDetectionLabelSource` gets us all the bounding boxes that fall within the 1000x1000 window and are automatically downscaled by a factor of 2 so that they still align correctly with the image chip.

[17]:
```
chip = raster_source[:1000:2, :1000:2]
bboxes, _, _ = label_source[:1000:2, :1000:2]
```

[18]:
```
from matplotlib import pyplot as plt
from matplotlib import patches as patches
import numpy as np
from shapely.geometry import Polygon

fig, ax = plt.subplots(figsize=(5, 5))

ax.matshow(chip[..., :3])

bbox_color = class_config.color_triples[class_config.get_class_id('car')]
bbox_polygons = [Polygon.from_bounds(*bounds) for bounds in bboxes[:, [1, 0, 3, 2]]]
for p in bbox_polygons:
    xy = np.array(p.exterior)
    patch = patches.Polygon(xy, fill=None, color=bbox_color)
    ax.add_patch(patch)
ax.autoscale()

plt.show()
```

nbsphinx-code-borderwhite

Chip Classification - `ChipClassificationLabelSource`

The *`ChipClassificationLabelSource`* can do the following:

1. The trivial case: given a rectangular polygons (representing chips) with associated class IDs, allow querying the class ID for those chips.
2. The more interesting case: given polygons representing the members of the target class(es), infer the class ID for a window based on how much it overlaps with any label-polygon.

The example below showcases the latter.

Data source: SpaceNet 1

> **Warning:** This example will download a 150 MB file.

[20]:
```
img_uri = 's3://spacenet-dataset/AOIs/AOI_1_Rio/srcData/mosaic_3band/013022232020.tif'
label_uri = 's3://spacenet-dataset/AOIs/AOI_1_Rio/srcData/buildingLabels/Rio_Buildings_
↪Public_AOI_v2.geojson'
```

Create - a `RasterSource` to get the image extent and a `CRSTRansformer` - a `VectorSource` to read the vector labels

[20]:
```
from rastervision.core.data import (
    ClassConfig, ClassInferenceTransformer,
    GeoJSONVectorSource, RasterioSource)

class_config = ClassConfig(
    names=['background', 'building'], null_class='background')

raster_source = RasterioSource(img_uri, allow_streaming=True)

vector_source = GeoJSONVectorSource(
    label_uri,
    crs_transformer=raster_source.crs_transformer,
    ignore_crs_field=True,
    vector_transformers=[
        ClassInferenceTransformer(
            default_class_id=class_config.get_class_id('building'))])
```

:class:‘ chip$_{c}lassification_{l}abel_{s}ource.ChipClassificationLabelSource‘accepts a number of options for configuring the class ID class : ‘ chip_{c}lassification_{l}abel_{s}ource_{c}onfig.ChipClassificationLabelSourceConfig‘.$

[21]:
```
from rastervision.core.data import (
    ChipClassificationLabelSource, ChipClassificationLabelSourceConfig)

cfg = ChipClassificationLabelSourceConfig(
    ioa_thresh=0.5,
    use_intersection_over_cell=False,
    pick_min_class_id=False,
    background_class_id=class_config.null_class_id,
    infer_cells=True)

label_source = ChipClassificationLabelSource(
    cfg, vector_source, extent=raster_source.extent, lazy=True)
```

```
2022-09-13 12:09:08:rastervision.pipeline.file_system.utils: INFO - Using cached file /
↪opt/data/tmp/cache/s3/spacenet-dataset/AOIs/AOI_1_Rio/srcData/buildingLabels/Rio_
↪Buildings_Public_AOI_v2.geojson.
```

```
Transforming to pixel coords:   4%|4         | 9364/220049 [00:05<01:52, 1872.72it/s]
Splitting multi-part geoms:   8%|8         | 18662/220049 [00:05<00:53, 3732.25it/s]
Simplifying polygons:   8%|7         | 16784/220070 [00:05<01:00, 3356.64it/s]
Splitting multi-part geoms:  18%|#8        | 40611/220057 [00:05<00:22, 8122.11it/s]
```

We can query `ChipClassificationLabelSourceConfig` in the usual way. For a given index/window, `ChipClassificationLabelSourceConfig` retuns a single `int` representing the class ID of the inferred class of that chip.

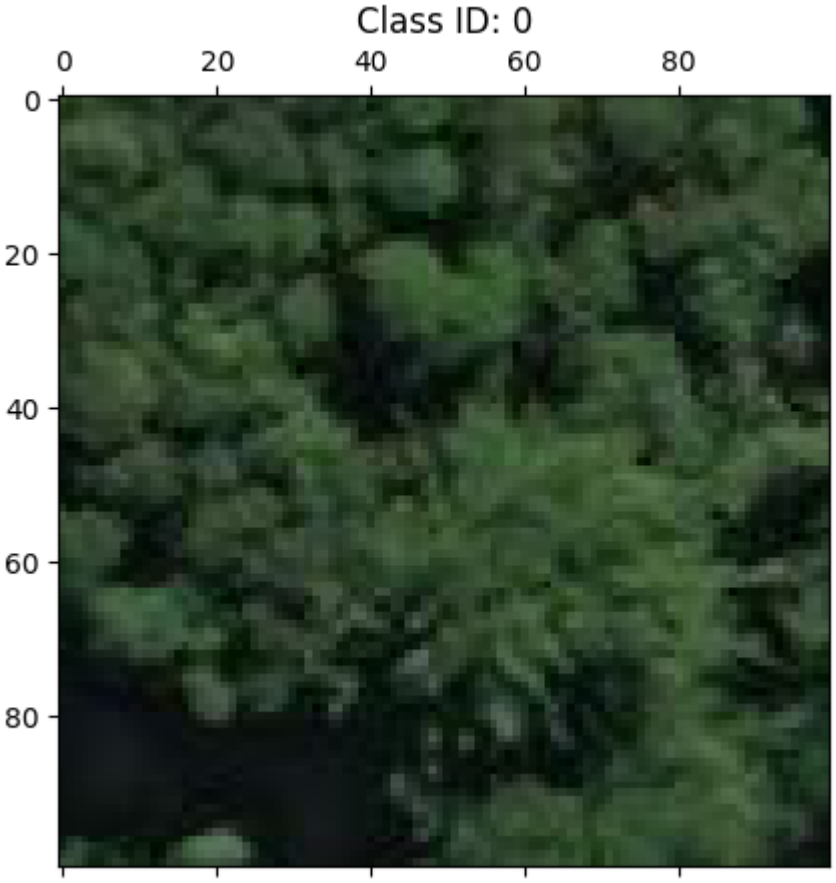
Querying a 100x100 chip from the corner of the raster, we see that there are no buildings, and indeed the label returned corresponds to the `"background"` class.

[22]:
```
from matplotlib import pyplot as plt
```

(continues on next page)

(continued from previous page)

```
chip = raster_source[:100, :100]
label = label_source[:100, :100]

fig, ax = plt.subplots(figsize=(5, 5))
ax.matshow(chip)
ax.set_title(f'Class ID: {label}')
plt.show()
```



nbsphinx-code-borderwhite

Querying a different chip, we see that it does in fact contain buildings, and the label source agrees – returning the label corresponding to the "building" class.

[23]:
```
from matplotlib import pyplot as plt

chip = raster_source[:200, 200:400]
label = label_source[:200, 200:400]

fig, ax = plt.subplots(figsize=(5, 5))
ax.matshow(chip)
ax.set_title(f'Class ID: {label}')
plt.show()
```

nbsphinx-code-borderwhite

To get a better sense of `ChipClassificationLabelSource`'s working we will now query the class for many 100x100 chips within a 1500x1500 chunk of the full raster.

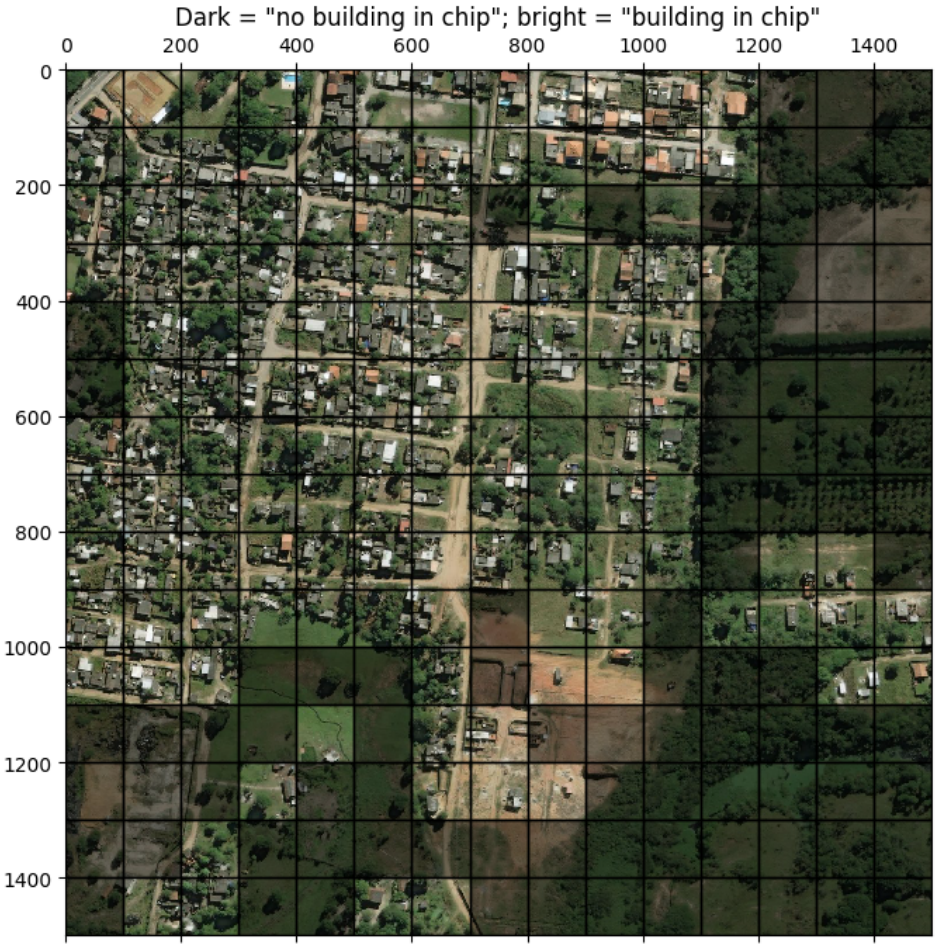
[24]:
```
from matplotlib import pyplot as plt
from matplotlib import patches as patches
import numpy as np
from shapely.geometry import Polygon
from rastervision.core.box import Box

chip = raster_source[500:2000, 500:2000]

extent = Box(500, 500, 2000, 2000)
windows = extent.get_windows(100, stride=100)
label_source.populate_labels(cells=windows)
```

[25]:
```
fig, ax = plt.subplots(figsize=(8, 8))
ax.matshow(chip[..., :3])

for w in windows:
    p = w.translate(-500, -500).to_shapely()
    xy = np.array(p.exterior)
    label = label_source[w] # equivalent to label_source[w.ymin:w.ymax, w.xmin:w.xmax]
    if label == 0:
```

(continues on next page)

(continued from previous page)

```
        patch = patches.Polygon(xy, fc='k', alpha=0.45)
        ax.add_patch(patch)
        patch = patches.Polygon(xy, fill=None, ec='k', alpha=1, lw=1)
    else:
        patch = patches.Polygon(xy, fill=None, ec='k', alpha=1, lw=1)
    ax.add_patch(patch)
ax.set_title('Dark = "no building in chip"; bright = "building in chip"')
plt.show()
```



nbsphinx-code-borderwhite

7.3.3 Labels

The *Labels* class is a source-agnostic, in-memory representation of the labels in a scene.

We can get all or a subset of a *LabelSource* as a *Labels* instance, by calling *LabelSource.get_labels()*. E.g.

```
labels = label_source.get_labels(window)
```

Crucially, the predictions produced by a model can also be converted to `Labels` instance, allowing them to be compared agains the `Labels` from the `LabelSource` (i.e, the ground truth) to get performance metrics. This is essentially what `Evaluator` does.

There is one for each of the three tasks:

- *ChipClassificationLabels*
- *SemanticSegmentationLabels*
- *ObjectDetectionLabels*

7.4 Sampling training data

7.4.1 The GeoDataset class

The *GeoDataset* is a PyTorch-compatible `Dataset` implementation that allows sampling images and labels from a *Scene*.

It comes in two flavors:

1. *SlidingWindowGeoDataset*
2. *RandomWindowGeoDataset*

Below we explore both in the context of semantic segmentation.

First, let's define a handy plotting function:

[1]:
```
def show_windows(img, windows, title=''):
    from matplotlib import pyplot as plt
    import matplotlib.patches as patches

    fig, ax = plt.subplots(1, 1, squeeze=True, figsize=(8, 8))
    ax.imshow(img)
    ax.axis('off')
    # draw windows on top of the image
    for w in windows:
        p = patches.Polygon(w.to_points(), color='r', linewidth=1, fill=False)
        ax.add_patch(p)
    ax.autoscale()
    ax.set_title(title)
    plt.show()
```

SlidingWindowGeoDataset

The *SlidingWindowGeoDataset* allows reading the scene left-to-right, top-to-bottom, using a sliding window.

[2]:
```
image_uri = 's3://spacenet-dataset/spacenet/SN7_buildings/train/L15-0331E-1257N_1327_
↪3160_13/images/global_monthly_2018_01_mosaic_L15-0331E-1257N_1327_3160_13.tif'
label_uri = 's3://spacenet-dataset/spacenet/SN7_buildings/train/L15-0331E-1257N_1327_
↪3160_13/labels/global_monthly_2018_01_mosaic_L15-0331E-1257N_1327_3160_13_Buildings.
↪geojson'
```

Here we make use of the convenience API, *from_uris()* (specifically, *from_uris()*), but we can also use the normal constructor if we want to manually define the *RasterSource* and *LabelSource*.

[3]:
```
from rastervision.core.data import ClassConfig
from rastervision.pytorch_learner import (
    SemanticSegmentationSlidingWindowGeoDataset, SemanticSegmentationVisualizer)

import albumentations as A

class_config = ClassConfig(
    names=['background', 'building'],
    colors=['lightgray', 'darkred'],
    null_class='background')

ds = SemanticSegmentationSlidingWindowGeoDataset.from_uris(
    class_config=class_config,
    image_uri=image_uri,
    label_vector_uri=label_uri,
    label_vector_default_class_id=class_config.get_class_id('building'),
    image_raster_source_kw=dict(allow_streaming=True),
    size=200,
    stride=200,
    transform=A.Resize(256, 256)
)
```

```
2022-12-02 12:39:05:rastervision.core.data.raster_source.rasterio_source: WARNING -␣
↪Raster block size (2, 1024) is too non-square. This can slow down reading. Consider re-
↪tiling using GDAL.
2022-12-02 12:39:05:rastervision.pipeline.file_system.utils: INFO - Using cached file /
↪opt/data/tmp/cache/s3/spacenet-dataset/spacenet/SN7_buildings/train/L15-0331E-1257N_
↪1327_3160_13/labels/global_monthly_2018_01_mosaic_L15-0331E-1257N_1327_3160_13_
↪Buildings.geojson.
```

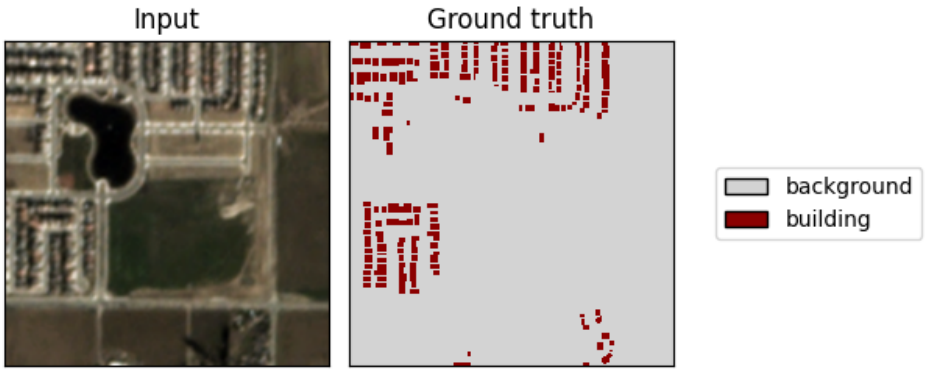
We can read a data sample and the corresponding ground truth from the Dataset like so:

[4]:
```
x, y = ds[0]
x.shape, y.shape
```

[4]:
```
(torch.Size([3, 256, 256]), torch.Size([256, 256]))
```

And then plot it using the *SemanticSegmentationVisualizer*:

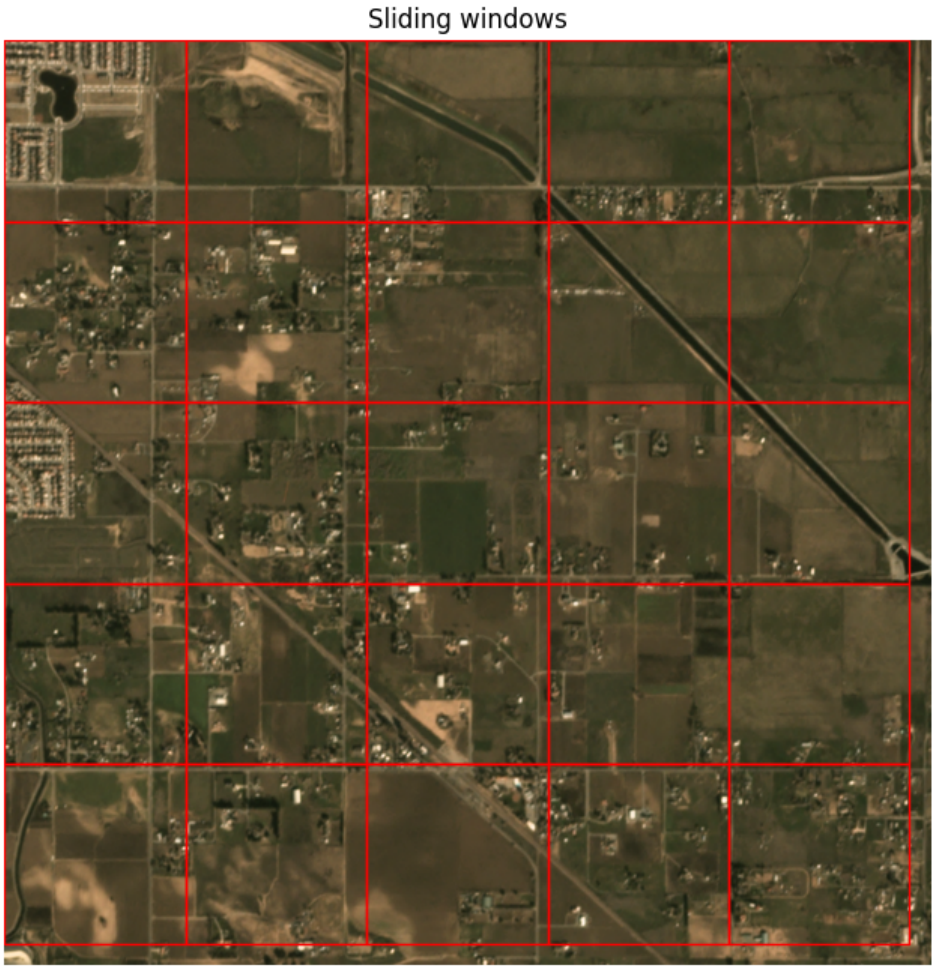
[5]:
```
viz = SemanticSegmentationVisualizer(
    class_names=class_config.names, class_colors=class_config.colors)
viz.plot_batch(x.unsqueeze(0), y.unsqueeze(0), show=True)
```



nbsphinx-code-borderwhite

The above was the first sliding window in the dataset. We can visualize what the full set of windows looks like like so:

[6]:
```
img_full = ds.scene.raster_source[:, :]
show_windows(img_full, ds.windows, title='Sliding windows')
```



nbsphinx-code-borderwhite

RandomWindowGeoDataset

The *RandomWindowGeoDataset* allows reading the scene by sampling random window sizes and locations.

[8]:
```
image_uri = 's3://spacenet-dataset/spacenet/SN7_buildings/train/L15-0331E-1257N_1327_
↪3160_13/images/global_monthly_2018_01_mosaic_L15-0331E-1257N_1327_3160_13.tif'
label_uri = 's3://spacenet-dataset/spacenet/SN7_buildings/train/L15-0331E-1257N_1327_
↪3160_13/labels/global_monthly_2018_01_mosaic_L15-0331E-1257N_1327_3160_13_Buildings.
↪geojson'
```

As before, we make use of the convenience API, *from_uris()* (specifically, *from_uris()*), but we can also use the normal constructor if we want to manually define the *RasterSource* and *LabelSource*.

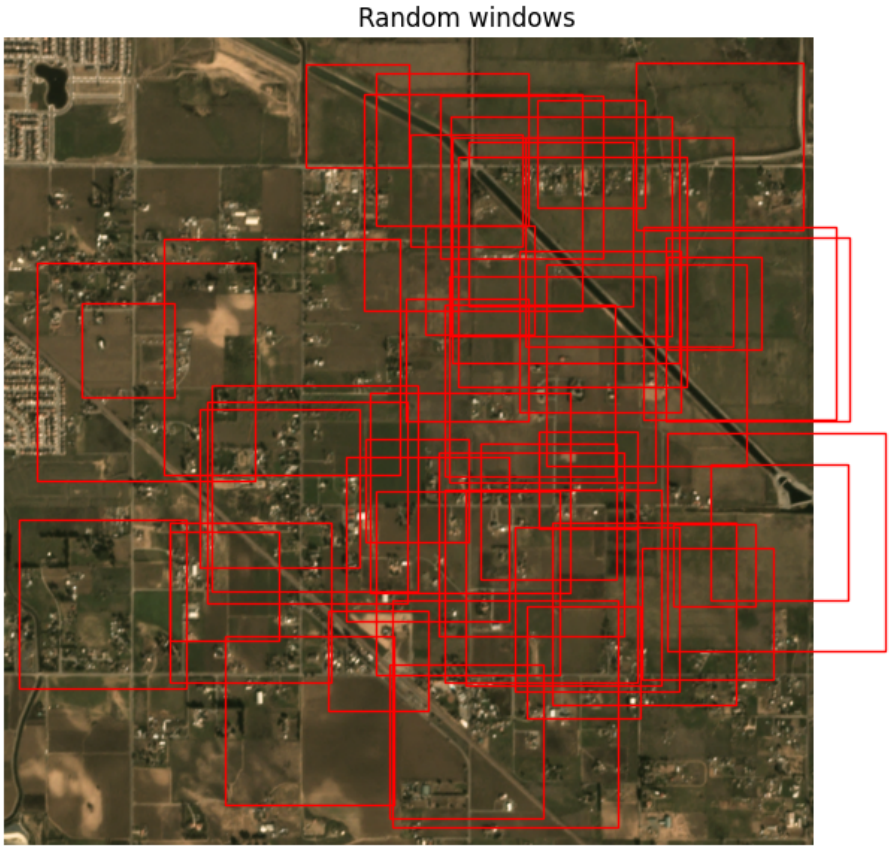
[9]:
```
from rastervision.core.data import ClassConfig
from rastervision.pytorch_learner import SemanticSegmentationRandomWindowGeoDataset
```

(continues on next page)

(continued from previous page)

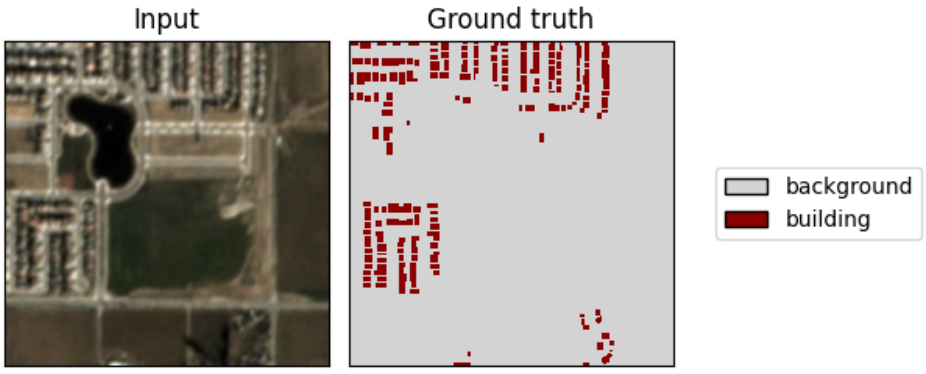
```
import albumentations as A

class_config = ClassConfig(
    names=['background', 'building'],
    colors=['lightgray', 'darkred'],
    null_class='background')

ds = SemanticSegmentationRandomWindowGeoDataset.from_uris(
    class_config=class_config,
    image_uri=image_uri,
    label_vector_uri=label_uri,
    label_vector_default_class_id=class_config.get_class_id('building'),
    image_raster_source_kw=dict(allow_streaming=True),
    # window sizes will randomly vary from 100x100 to 300x300
    size_lims=(100, 300),
    # resize chips to 256x256 before returning
    out_size=256,
    # allow windows to overflow the extent by 100 pixels
    padding=100
)

img_full = ds.scene.raster_source[:, :]
windows = [ds.sample_window() for _ in range(50)]
show_windows(img_full, windows, title='Random windows')
```

```
2022-09-13 12:52:22:rastervision.pipeline.file_system.utils: INFO - Using cached file /
↪opt/data/tmp/cache/s3/spacenet-dataset/spacenet/SN7_buildings/train/L15-0331E-1257N_
↪1327_3160_13/labels/global_monthly_2018_01_mosaic_L15-0331E-1257N_1327_3160_13_
↪Buildings.geojson.
```



nbsphinx-code-borderwhite

[10]:
```
x, y = ds[0]
viz.plot_batch(x.unsqueeze(0), y.unsqueeze(0), show=True)
```



nbsphinx-code-borderwhite

7.5 Scenes and AOIs

We have seen how to use *RasterSources* and *LabelSources* in other tutorials.

This tutorial introduces the *Scene* abstraction, which bundles them together. Additionally, it allows specifying one or more "Areas of Interest" (AOIs) if we are only interested in a subset of the scene.

7.5.1 Example

[1]:
```
image_uri = 's3://spacenet-dataset/spacenet/SN7_buildings/train/L15-0331E-1257N_1327_
↪3160_13/images/global_monthly_2018_01_mosaic_L15-0331E-1257N_1327_3160_13.tif'
label_uri = 's3://spacenet-dataset/spacenet/SN7_buildings/train/L15-0331E-1257N_1327_
↪3160_13/labels/global_monthly_2018_01_mosaic_L15-0331E-1257N_1327_3160_13_Buildings.
↪geojson'
```

[2]:
```
from rastervision.core.data import RasterioSource

raster_source = RasterioSource(image_uri, allow_streaming=True)
```

[3]:
```
from rastervision.core.data import (
    ClassConfig, ClassInferenceTransformer, GeoJSONVectorSource,
    RasterizedSource, Scene)

class_config = ClassConfig(
    names=['background', 'building'],
    colors=['lightgray', 'darkred'],
    null_class='background')

class_inf_tf = ClassInferenceTransformer(
    default_class_id=class_config.get_class_id('building'))

vector_source = GeoJSONVectorSource(
    uri=label_uri,
    ignore_crs_field=True,
    crs_transformer=raster_source.crs_transformer,
    vector_transformers=[class_inf_tf])

label_raster_source = RasterizedSource(
    vector_source=vector_source,
    background_class_id=class_config.null_class_id,
    extent=raster_source.extent)
```

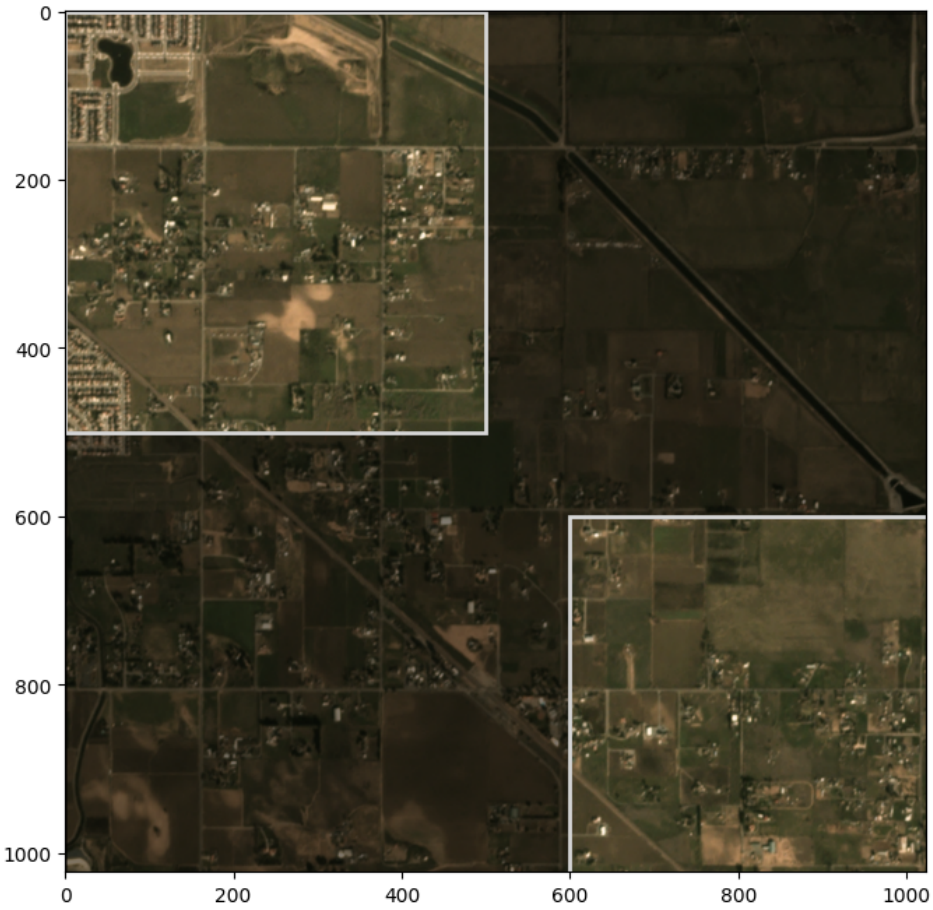
```
2022-10-20 08:59:30:rastervision.pipeline.file_system.utils: INFO - Using cached file /
↪opt/data/tmp/cache/s3/spacenet-dataset/spacenet/SN7_buildings/train/L15-0331E-1257N_
↪1327_3160_13/labels/global_monthly_2018_01_mosaic_L15-0331E-1257N_1327_3160_13_
↪Buildings.geojson.
```

Define some AOI using polygons:

[4]:
```
from shapely.geometry import Polygon

aoi_polygons = [
    Polygon.from_bounds(xmin=0, ymin=0, xmax=500, ymax=500),
    Polygon.from_bounds(xmin=600, ymin=600, xmax=1024, ymax=1024),
]
```

Visualize the AOI:

[5]:
```
import numpy as np
from shapely.ops import unary_union
from matplotlib import pyplot as plt
from matplotlib import patches as mpatches

img = raster_source[:, :]

H, W = img.shape[:2]
extent = Polygon.from_bounds(0, 0, W, H)
bg = extent.difference(unary_union(aoi_polygons))
bg = bg if bg.geom_type == 'MultiPolygon' else [bg]

fig, ax = plt.subplots(1, 1, squeeze=True, figsize=(8, 8))
ax.imshow(img)

for p in bg:
    p = mpatches.Polygon(np.array(p.exterior), color='k', linewidth=2, alpha=0.5)
    ax.add_patch(p)

for aoi in aoi_polygons:
    p = mpatches.Polygon(
        np.array(aoi.exterior), color='#d9d9d9', linewidth=2, fill=False)
    ax.add_patch(p)

plt.show()
```



nbsphinx-code-borderwhite

Finally, define a *Scene*:

[6]:
```
from rastervision.core.data import Scene

scene = Scene(
    id='my_scene',
    raster_source=raster_source,
    label_source=label_raster_source,
    aoi_polygons=aoi_polygons)
```

We can now index the scene like so:

[7]:
```
x, y = scene[100:200, 100:200]

x.shape, y.shape
```

[7]:
```
((100, 100, 3), (100, 100, 1))
```

Note that the `Scene` itself does not prevent you from reading windows outside the AOI.

A simple check to make sure you're inside the AOI before reading is to use the *`within_aoi()`* method:

[8]:
```
from rastervision.core.box import Box

window_inside_aoi = Box(ymin=100, xmin=100, ymax=200, xmax=200)
```

(continues on next page)

(continued from previous page)

```
window_not_inside_aoi = Box(ymin=500, xmin=500, ymax=600, xmax=600)

print(window_inside_aoi, Box.within_aoi(window_inside_aoi, aoi_polygons))
print(window_not_inside_aoi, Box.within_aoi(window_not_inside_aoi, aoi_polygons))
```

```
Box(ymin=100, xmin=100, ymax=200, xmax=200) True
Box(ymin=500, xmin=500, ymax=600, xmax=600) False
```

7.5.2 Easier initialization

If you found the above steps to be tedious, there is an alternative, simpler way of creating a scene:

[9]:
```
from rastervision.core.data.utils import make_ss_scene

scene = make_ss_scene(
    class_config=class_config,
    image_uri=image_uri,
    label_vector_uri=label_uri,
    label_vector_default_class_id=class_config.get_class_id('building'),
    image_raster_source_kw=dict(allow_streaming=True))
```

```
2022-10-20 08:59:38:rastervision.pipeline.file_system.utils: INFO - Using cached file /
↪opt/data/tmp/cache/s3/spacenet-dataset/spacenet/SN7_buildings/train/L15-0331E-1257N_
↪1327_3160_13/labels/global_monthly_2018_01_mosaic_L15-0331E-1257N_1327_3160_13_
↪Buildings.geojson.
```

make_ss_scene() is for creating semantic segmentation scenes. There is also *make_cc_scene()* for chip classificaiton and *make_od_scene()* for object detection.

7.6 Plot samples from Datasets using Visualizers

This notebook shows how to use *Visualizer* objects to plot image/label samples for computer vision PyTorch Datasets. There are examples for *semantic segmentation*, *object detection*, and *image classification*. We use Raster Vision's *GeoDataset* functionality to read the data, but the *Visualizer* classes can be used with any images and labels as long as they are in the expected format.

7.6.1 Setup

[2]:
```
from os.path import join

import matplotlib.pyplot as plt
import torch

from rastervision.pytorch_learner.dataset import (
    SemanticSegmentationSlidingWindowGeoDataset,
    ObjectDetectionSlidingWindowGeoDataset,
    ClassificationSlidingWindowGeoDataset)
from rastervision.pytorch_learner.dataset.visualizer import (
    SemanticSegmentationVisualizer,
```

(continues on next page)

(continued from previous page)

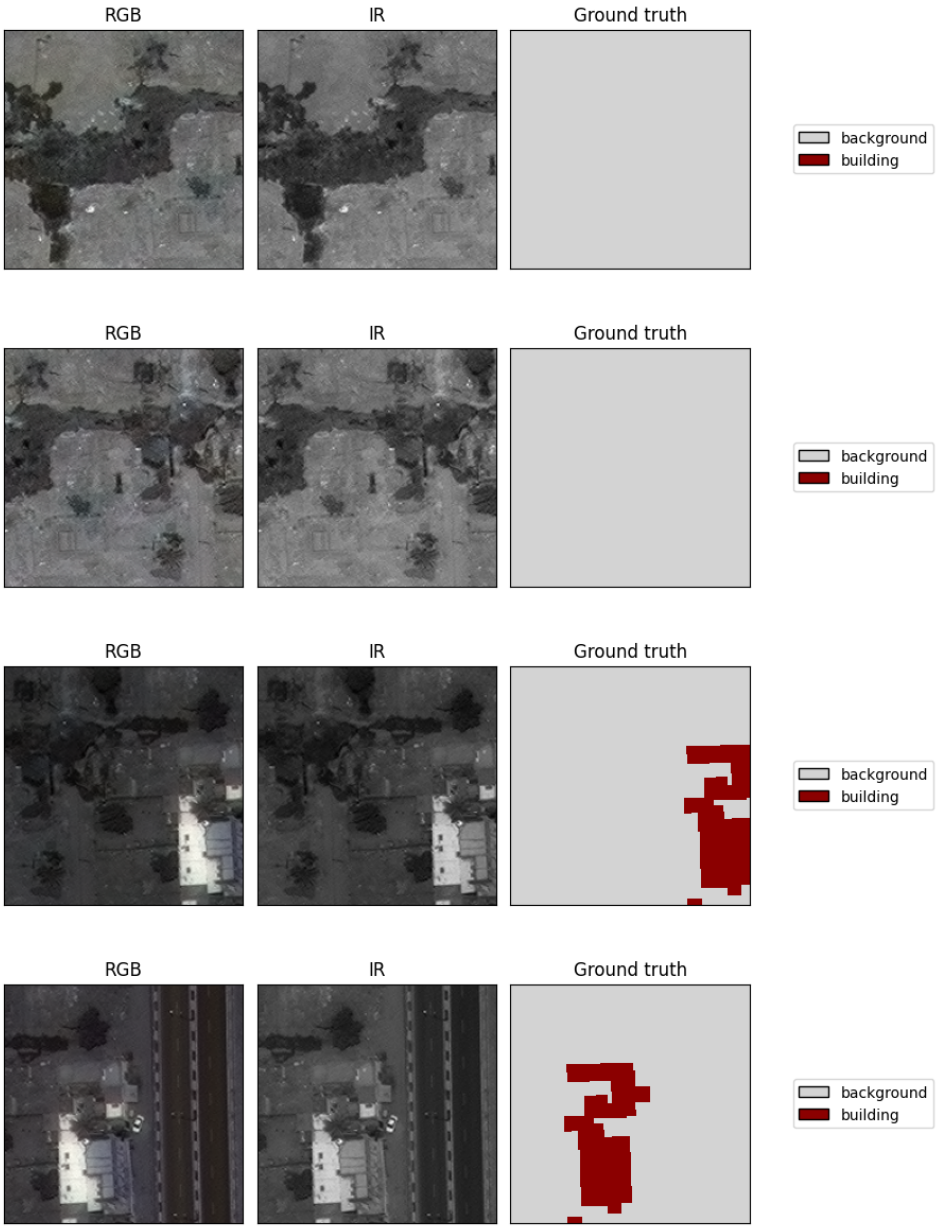
```
    ObjectDetectionVisualizer,
    ClassificationVisualizer)
from rastervision.core.data import ClassConfig
```

[3]:
```
# These examples all use a scene from the SpaceNet 2 buildings dataset.
image_uri = 's3://spacenet-dataset/spacenet/SN2_buildings/train/AOI_5_Khartoum/PS-MS/SN2_
↪buildings_train_AOI_5_Khartoum_PS-MS_img1004.tif'
label_uri = 's3://spacenet-dataset/spacenet/SN2_buildings/train/AOI_5_Khartoum/geojson_
↪buildings/SN2_buildings_train_AOI_5_Khartoum_geojson_buildings_img1004.geojson'

class_config = ClassConfig(
    names=['background', 'building'],
    colors=['lightgray', 'darkred'],
    null_class='background')
chip_sz = 200
chip_stride = chip_sz // 2
# This describes how to group different input channels when plotting images.
# It's helpful when dealing with multiband imagery.
channel_display_groups = {'RGB': (0, 1, 2), 'IR': (3, )}
```

7.6.2 Semantic Segmentation – `SemanticSegmentationVisualizer`

[5]:
```
ds = SemanticSegmentationSlidingWindowGeoDataset.from_uris(
    class_config=class_config,
    image_uri=image_uri,
    label_vector_uri=label_uri,
    label_vector_default_class_id=class_config.get_class_id('building'),
    image_raster_source_kw=dict(allow_streaming=True),
    size=chip_sz,
    stride=chip_stride)

vis = SemanticSegmentationVisualizer(
    class_names=class_config.names, class_colors=class_config.colors,
    channel_display_groups=channel_display_groups)
x, y = vis.get_batch(ds, 4)
vis.plot_batch(x, y, show=True)
```

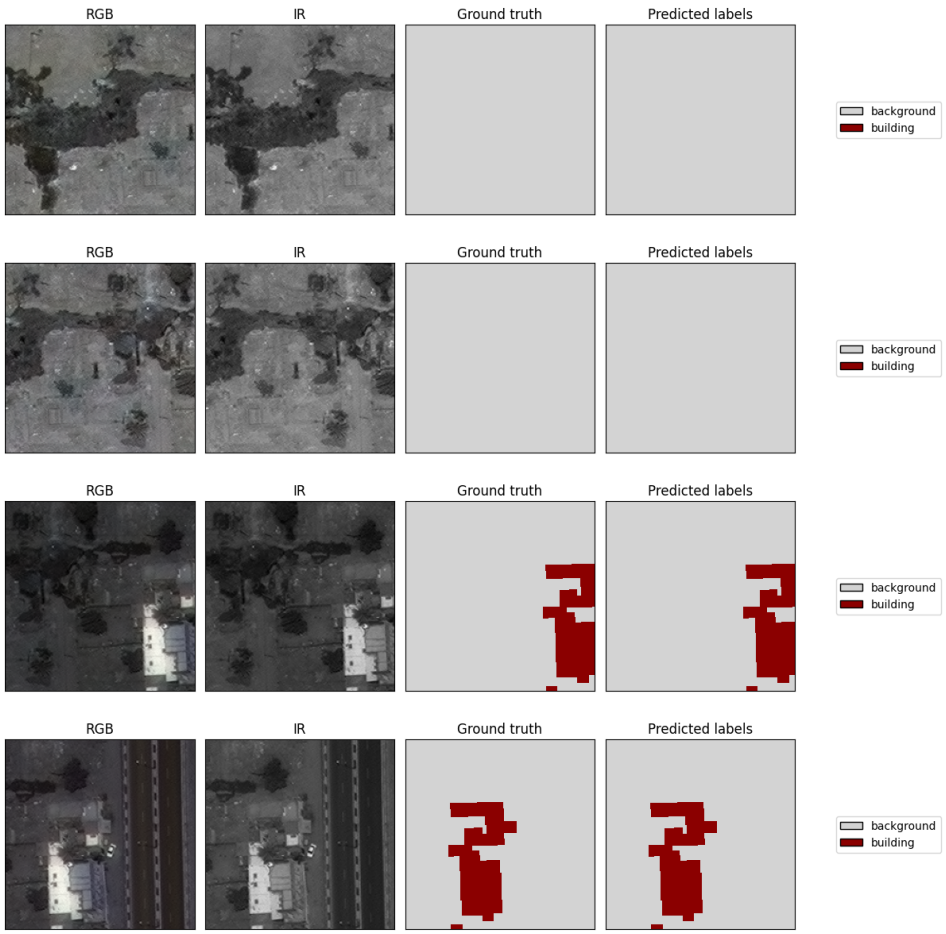
```
2022-09-22 19:55:13:rastervision.pipeline.file_system.utils: INFO - Using cached file /
↪opt/data/tmp/cache/s3/spacenet-dataset/spacenet/SN2_buildings/train/AOI_5_Khartoum/PS-
↪MS/SN2_buildings_train_AOI_5_Khartoum_PS-MS_img1004.tif.
2022-09-22 19:55:14:rastervision.pipeline.file_system.utils: INFO - Using cached file /
↪opt/data/tmp/cache/s3/spacenet-dataset/spacenet/SN2_buildings/train/AOI_5_Khartoum/
↪geojson_buildings/SN2_buildings_train_AOI_5_Khartoum_geojson_buildings_img1004.geojson.
```



nbsphinx-code-borderwhite

The `Visualizer` can also display predictions alongside ground truth labels. Here we will use the ground truth labels as mock predictions for testing purposes, simulating a model with perfect accuracy.

[17]:
```
z = torch.zeros((4, 3, 200, 200))
z[:, 1, :, :] = y
vis.plot_batch(x, y, z=z, show=True)
```



nbsphinx-code-borderwhite

7.6.3 Object Detection – `ObjectDetectionVisualizer`

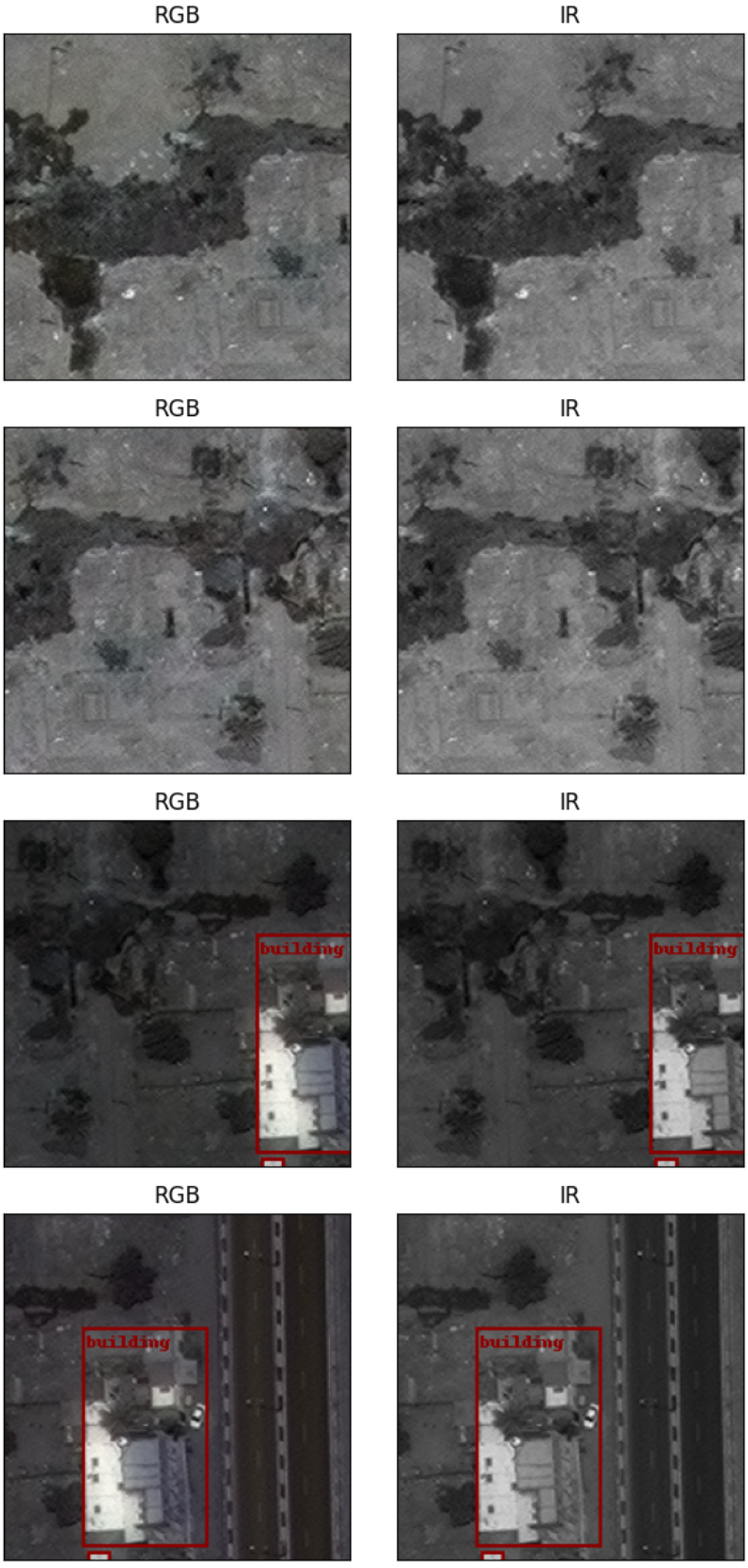
[6]:
```
ds = ObjectDetectionSlidingWindowGeoDataset.from_uris(
    class_config=class_config,
    image_uri=image_uri,
    label_vector_uri=label_uri,
    size=chip_sz,
    stride=chip_stride,
    label_vector_default_class_id=class_config.get_class_id('building'),
    image_raster_source_kw=dict(allow_streaming=True))

vis = ObjectDetectionVisualizer(
    class_names=class_config.names, class_colors=class_config.colors,
    channel_display_groups=channel_display_groups)
x, y = vis.get_batch(ds, 4)
vis.plot_batch(x, y, show=True)
```

```
2022-09-14 18:54:38:rastervision.pipeline.file_system.utils: INFO - Using cached file /
↪opt/data/tmp/cache/s3/spacenet-dataset/spacenet/SN2_buildings/train/AOI_5_Khartoum/PS-
↪MS/SN2_buildings_train_AOI_5_Khartoum_PS-MS_img1004.tif.
2022-09-14 18:54:38:rastervision.pipeline.file_system.utils: INFO - Using cached file /
```

(continues on next page)

(continued from previous page)

```
↪opt/data/tmp/cache/s3/spacenet-dataset/spacenet/SN2_buildings/train/AOI_5_Khartoum/
↪geojson_buildings/SN2_buildings_train_AOI_5_Khartoum_geojson_buildings_img1004.geojson.
```

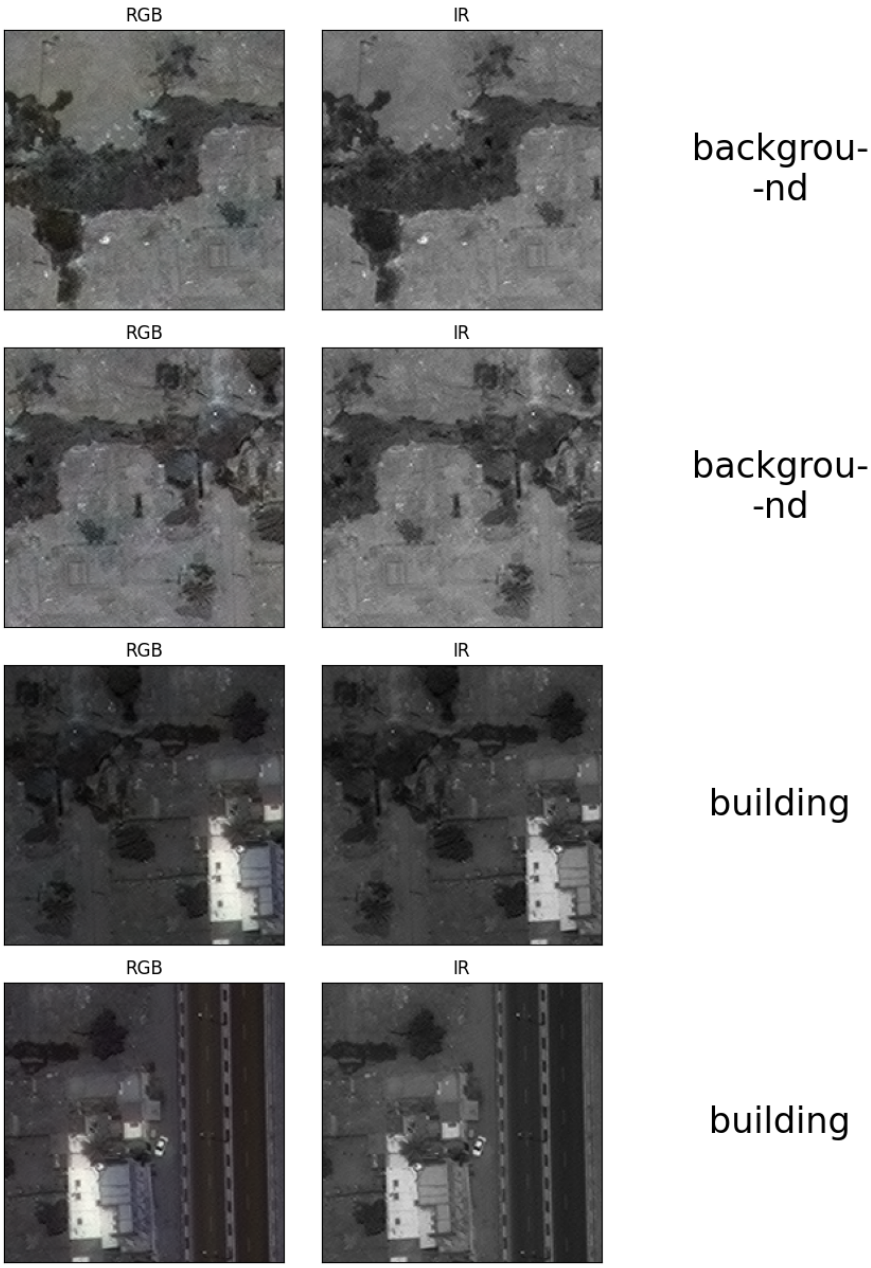


nbsphinx-code-borderwhite

7.6.4 Image Classification – `ClassificationVisualizer`

[ ]:
```
ds = ClassificationSlidingWindowGeoDataset.from_uris(
    class_config=class_config,
    image_uri=image_uri,
    label_vector_uri=label_uri,
    label_vector_default_class_id=class_config.get_class_id('building'),
    size=chip_sz,
    stride=chip_stride,
    label_source_kw=dict(
        ioa_thresh=0.5,
        use_intersection_over_cell=False,
        pick_min_class_id=False,
        background_class_id=class_config.get_class_id('background'),
        infer_cells=True,
        cell_sz=chip_sz))
vis = ClassificationVisualizer(
    class_names=class_config.names, class_colors=class_config.colors,
    channel_display_groups=channel_display_groups)
x, y = vis.get_batch(ds, 4)
vis.plot_batch(x, y, show=True)
```

```
2022-09-14 17:49:50:rastervision.pipeline.file_system.utils: INFO - Using cached file /
↪opt/data/tmp/cache/s3/spacenet-dataset/spacenet/SN2_buildings/train/AOI_5_Khartoum/PS-
↪MS/SN2_buildings_train_AOI_5_Khartoum_PS-MS_img1004.tif.
2022-09-14 17:49:50:rastervision.pipeline.file_system.utils: INFO - Using cached file /
↪opt/data/tmp/cache/s3/spacenet-dataset/spacenet/SN2_buildings/train/AOI_5_Khartoum/
↪geojson_buildings/SN2_buildings_train_AOI_5_Khartoum_geojson_buildings_img1004.geojson.
```



nbsphinx-code-borderwhite

7.7 Training a model

7.7.1 Define ClassConfig

[1]:
```
from rastervision.core.data import ClassConfig

class_config = ClassConfig(
    names=['background', 'building'],
    colors=['lightgray', 'darkred'],
```

(continues on next page)

(continued from previous page)

```
    null_class='background')
```

7.7.2 Define training and validation datasets

To keep things simple, we use one scene for training and one for validation. In a real workflow, we would normally use many more scenes.

[2]:
```
train_image_uri = 's3://spacenet-dataset/spacenet/SN7_buildings/train/L15-0331E-1257N_
↪1327_3160_13/images/global_monthly_2018_01_mosaic_L15-0331E-1257N_1327_3160_13.tif'
train_label_uri = 's3://spacenet-dataset/spacenet/SN7_buildings/train/L15-0331E-1257N_
↪1327_3160_13/labels/global_monthly_2018_01_mosaic_L15-0331E-1257N_1327_3160_13_
↪Buildings.geojson'
```

[3]:
```
val_image_uri = 's3://spacenet-dataset/spacenet/SN7_buildings/train/L15-0357E-1223N_1429_
↪3296_13/images/global_monthly_2018_01_mosaic_L15-0357E-1223N_1429_3296_13.tif'
val_label_uri = 's3://spacenet-dataset/spacenet/SN7_buildings/train/L15-0357E-1223N_1429_
↪3296_13/labels/global_monthly_2018_01_mosaic_L15-0357E-1223N_1429_3296_13_Buildings.
↪geojson'
```

[4]:
```
import albumentations as A

from rastervision.pytorch_learner import (
    SemanticSegmentationRandomWindowGeoDataset,
    SemanticSegmentationSlidingWindowGeoDataset,
    SemanticSegmentationVisualizer)

viz = SemanticSegmentationVisualizer(
    class_names=class_config.names, class_colors=class_config.colors)
```

Training dataset with random-window sampling and data augmentation

[5]:
```
data_augmentation_transform = A.Compose([
    A.Flip(),
    A.ShiftScaleRotate(),
    A.OneOf([
        A.HueSaturationValue(hue_shift_limit=10),
        A.RGBShift(),
        A.ToGray(),
        A.ToSepia(),
        A.RandomBrightness(),
        A.RandomGamma(),
    ]),
    A.CoarseDropout(max_height=32, max_width=32, max_holes=5)
])

train_ds = SemanticSegmentationRandomWindowGeoDataset.from_uris(
    class_config=class_config,
    image_uri=train_image_uri,
    label_vector_uri=train_label_uri,
```

(continues on next page)

(continued from previous page)

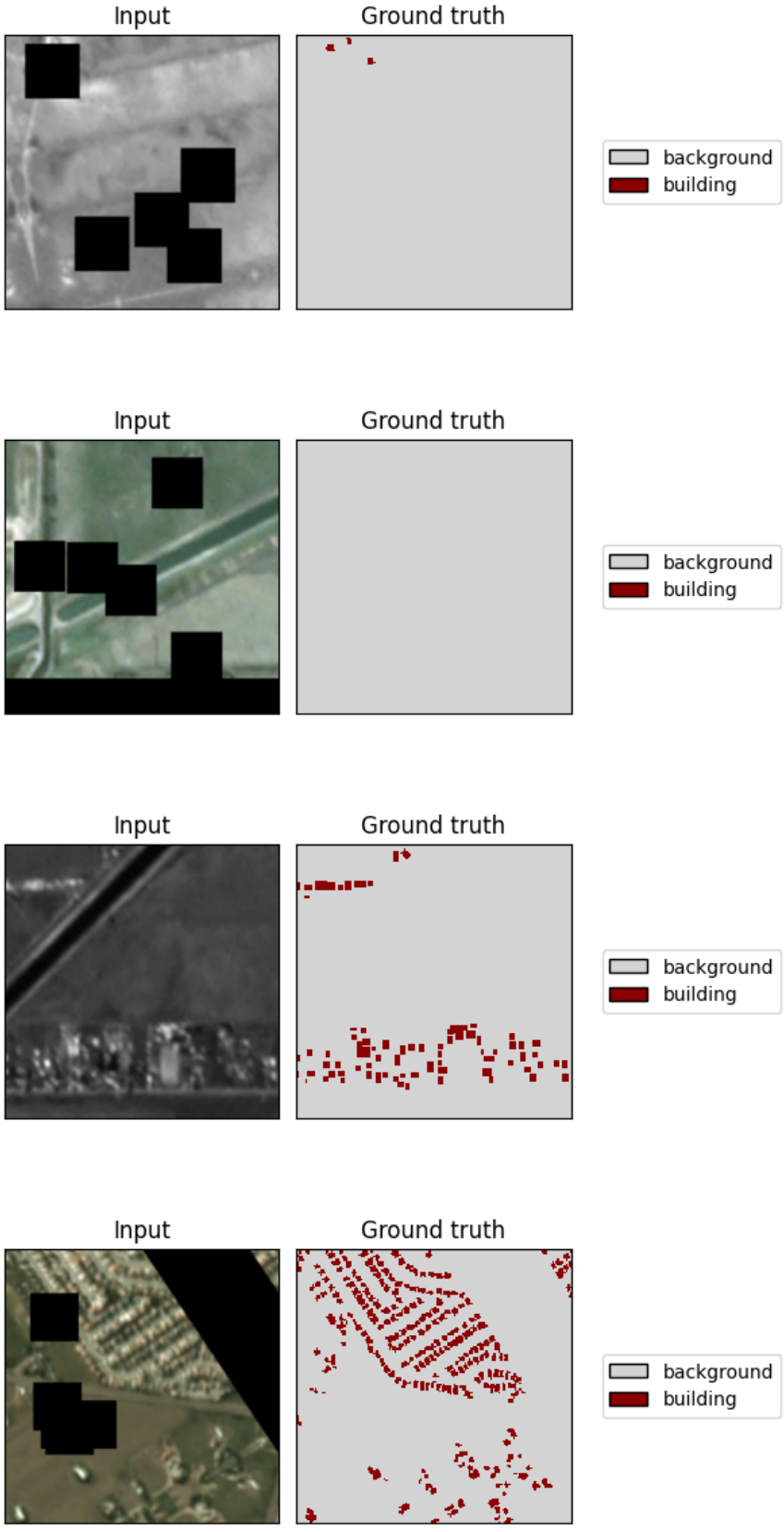
```
    label_vector_default_class_id=class_config.get_class_id('building'),
    size_lims=(150, 200),
    out_size=256,
    max_windows=400,
    transform=data_augmentation_transform)

len(train_ds)
```

```
2022-12-15 14:26:01:rastervision.pipeline.file_system.utils: INFO - Using cached file /
↪opt/data/tmp/cache/s3/spacenet-dataset/spacenet/SN7_buildings/train/L15-0331E-1257N_
↪1327_3160_13/images/global_monthly_2018_01_mosaic_L15-0331E-1257N_1327_3160_13.tif.
2022-12-15 14:26:01:rastervision.core.data.raster_source.rasterio_source: WARNING -␣
↪Raster block size (2, 1024) is too non-square. This can slow down reading. Consider re-
↪tiling using GDAL.
2022-12-15 14:26:01:rastervision.pipeline.file_system.utils: INFO - Using cached file /
↪opt/data/tmp/cache/s3/spacenet-dataset/spacenet/SN7_buildings/train/L15-0331E-1257N_
↪1327_3160_13/labels/global_monthly_2018_01_mosaic_L15-0331E-1257N_1327_3160_13_
↪Buildings.geojson.
```

[5]:
```
400
```

Visualize:

[8]:
```
x, y = viz.get_batch(train_ds, 4)
viz.plot_batch(x, y, show=True)
```



nbsphinx-code-borderwhite

Validation dataset with sliding-window sampling (and no data augmentation)

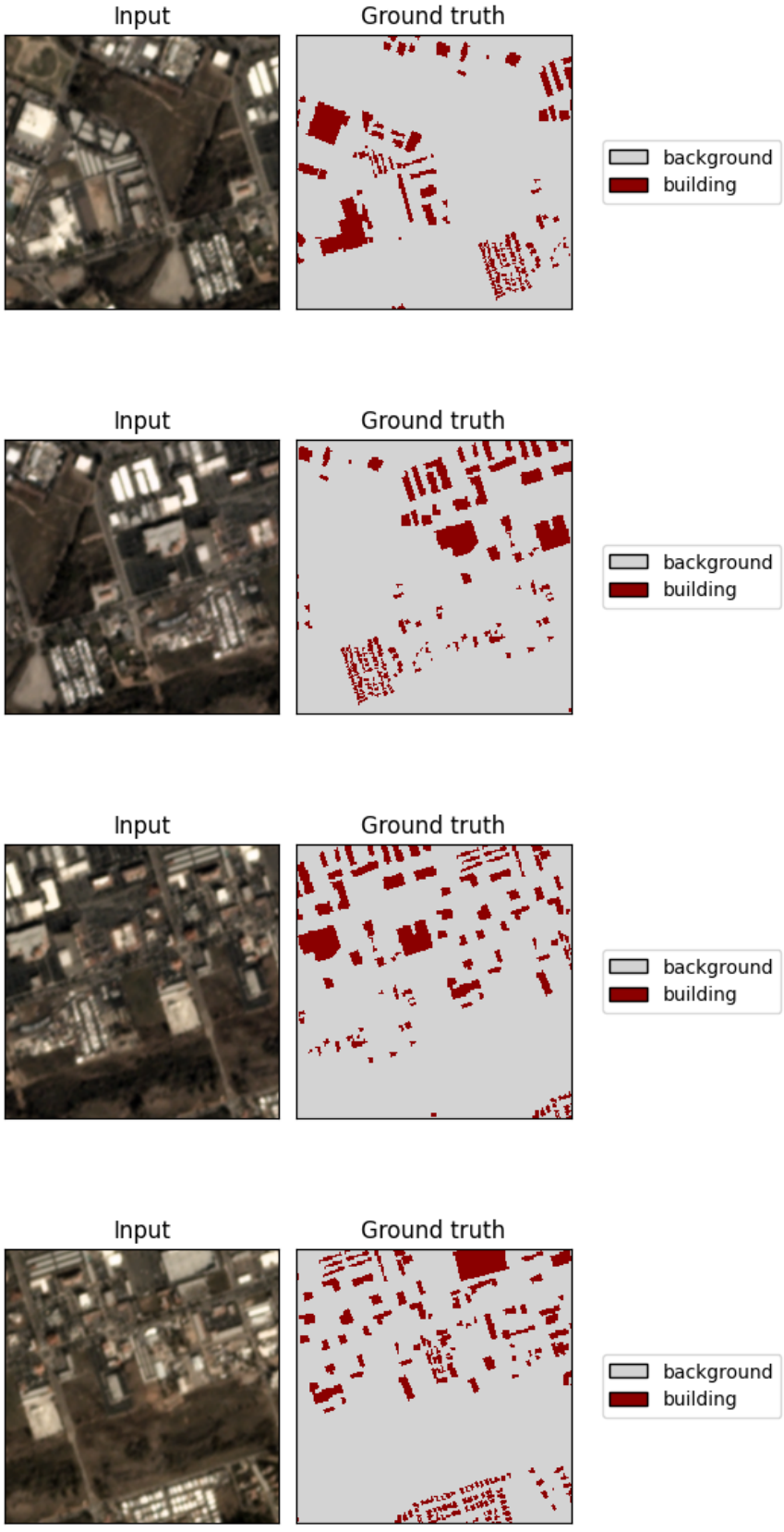
[6]:
```
val_ds = SemanticSegmentationSlidingWindowGeoDataset.from_uris(
    class_config=class_config,
    image_uri=val_image_uri,
    label_vector_uri=val_label_uri,
    label_vector_default_class_id=class_config.get_class_id('building'),
    size=200,
    stride=100,
    transform=A.Resize(256, 256))

len(val_ds)
```

```
2022-12-15 14:26:05:rastervision.pipeline.file_system.utils: INFO - Using cached file /
↪opt/data/tmp/cache/s3/spacenet-dataset/spacenet/SN7_buildings/train/L15-0357E-1223N_
↪1429_3296_13/images/global_monthly_2018_01_mosaic_L15-0357E-1223N_1429_3296_13.tif.
2022-12-15 14:26:05:rastervision.core.data.raster_source.rasterio_source: WARNING -␣
↪Raster block size (2, 1024) is too non-square. This can slow down reading. Consider re-
↪tiling using GDAL.
2022-12-15 14:26:05:rastervision.pipeline.file_system.utils: INFO - Using cached file /
↪opt/data/tmp/cache/s3/spacenet-dataset/spacenet/SN7_buildings/train/L15-0357E-1223N_
↪1429_3296_13/labels/global_monthly_2018_01_mosaic_L15-0357E-1223N_1429_3296_13_
↪Buildings.geojson.
```

[6]:
```
100
```

Visualize:

[10]:
```
x, y = viz.get_batch(val_ds, 4)
viz.plot_batch(x, y, show=True)
```



nbsphinx-code-borderwhite

7.7.3 Define model

Use a light-weight panoptic FPN model with a ResNet-18 backbone.

[7]:
```
import torch

model = torch.hub.load(
    'AdeelH/pytorch-fpn:0.3',
    'make_fpn_resnet',
    name='resnet18',
    fpn_type='panoptic',
    num_classes=len(class_config),
    fpn_channels=128,
    in_channels=3,
    out_size=(256, 256),
    pretrained=True)
```

```
Using cache found in /root/.cache/torch/hub/AdeelH_pytorch-fpn_0.3
```

7.7.4 Configure the training

`DataConfig` – Configure data-related hyper-parameters

[9]:
```
from rastervision.pytorch_learner import SemanticSegmentationGeoDataConfig

data_cfg = SemanticSegmentationGeoDataConfig(
    class_names=class_config.names,
    class_colors=class_config.colors,
    num_workers=0, # increase to use multi-processing
)
```

`SolverConfig` – Configure the loss, optimizer, and scheduler(s)

[10]:
```
from rastervision.pytorch_learner import SolverConfig

solver_cfg = SolverConfig(
    batch_sz=8,
    lr=3e-2,
    class_loss_weights=[1., 10.]
)
```

LearnerConfig – Combine DataConfig, SolverConfig (and optionally, ModelConfig)

[11]:
```
from rastervision.pytorch_learner import SemanticSegmentationLearnerConfig

learner_cfg = SemanticSegmentationLearnerConfig(data=data_cfg, solver=solver_cfg)
```

7.7.5 Initialize Learner

[12]:
```
from rastervision.pytorch_learner import SemanticSegmentationLearner

learner = SemanticSegmentationLearner(
    cfg=learner_cfg,
    output_dir='./train-demo/',
    model=model,
    train_ds=train_ds,
    valid_ds=val_ds,
)
```

[13]:
```
learner.log_data_stats()
```

```
2022-12-15 14:26:26:rastervision.pytorch_learner.learner: INFO - train_ds: 400 items
2022-12-15 14:26:26:rastervision.pytorch_learner.learner: INFO - valid_ds: 100 items
```

7.7.6 Run Tensorboard for monitoring

Note:

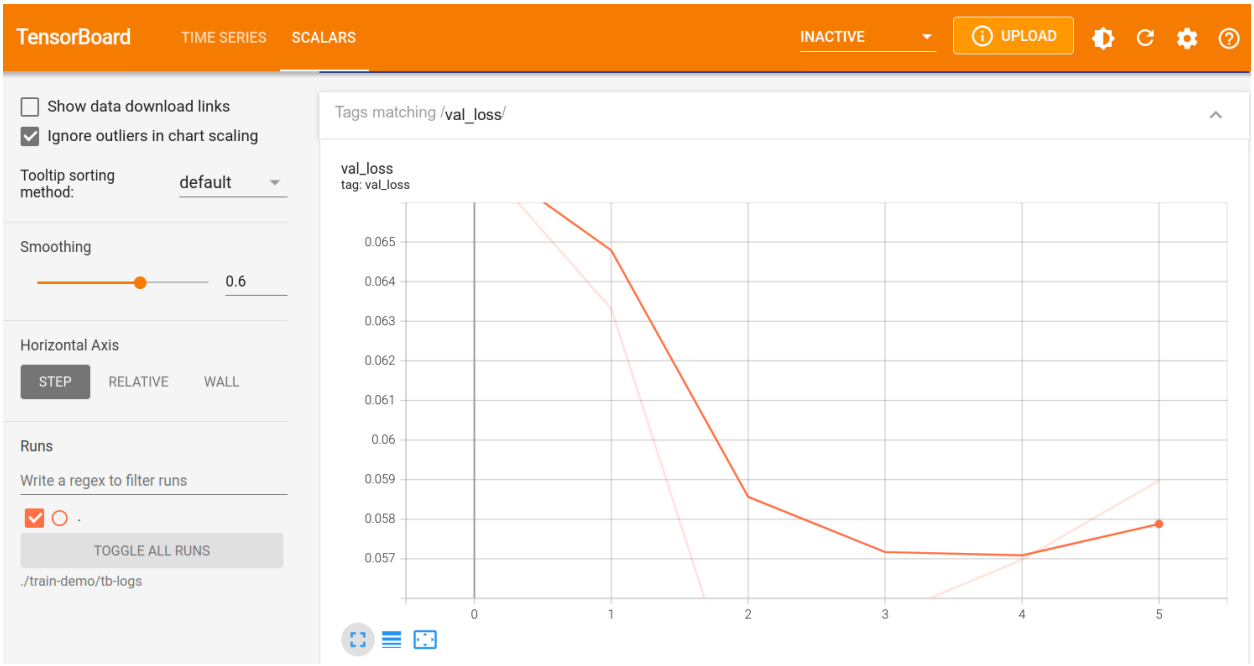
- If running inside the Raster Vision docker image, you will need to pass *–tensorboard* to *docker/run* for this to work.
- If the dashboard doen't auto-reload, you can click the reload button on the top-right.

---

[13]:
```
%load_ext tensorboard
```

This will start an instance of tensorboard and embed it in the output of the cell:

[ ]:
```
%tensorboard --bind_all --logdir "./train-demo/tb-logs" --reload_interval 10
```



7.7.7 Train – `Learner.train()`

[14]:
```
learner.train(epochs=3)
```

```
2022-12-15 14:26:42:rastervision.pytorch_learner.learner: INFO - epoch: 0
Training:   0%|          | 0/50 [00:00<?, ?it/s]
Validating:   0%|          | 0/13 [00:00<?, ?it/s]
2022-12-15 14:26:57:rastervision.pytorch_learner.learner: INFO - metrics:
{'avg_f1': 0.9074727892875671,
 'avg_precision': 0.9226699471473694,
 'avg_recall': 0.8927680850028992,
 'background_f1': 0.9416702389717102,
 'background_precision': 0.9641342163085938,
 'background_recall': 0.9202292561531067,
 'building_f1': 0.3365422785282135,
 'building_precision': 0.2660379707813263,
 'building_recall': 0.45789051055908203,
 'epoch': 0,
 'train_loss': 0.06389201164245606,
 'train_time': '0:00:10.922799',
 'val_loss': 0.06789381802082062,
 'valid_time': '0:00:03.880362'}
2022-12-15 14:26:57:rastervision.pytorch_learner.learner: INFO - epoch: 1
Training:   0%|          | 0/50 [00:00<?, ?it/s]
Validating:   0%|          | 0/13 [00:00<?, ?it/s]
2022-12-15 14:27:12:rastervision.pytorch_learner.learner: INFO - metrics:
{'avg_f1': 0.9250732064247131,
 'avg_precision': 0.916088879108429,
```

(continues on next page)

(continued from previous page)

```
 'avg_recall': 0.9342355132102966,
 'background_f1': 0.9656270146369934,
 'background_precision': 0.9497168064117432,
 'background_recall': 0.9820793271064758,
 'building_f1': 0.2418244332075119,
 'building_precision': 0.3835538327693939,
 'building_recall': 0.17657652497291565,
 'epoch': 1,
 'train_loss': 0.032022590637207034,
 'train_time': '0:00:10.916714',
 'val_loss': 0.12407467514276505,
 'valid_time': '0:00:04.199710'}
2022-12-15 14:27:13:rastervision.pytorch_learner.learner: INFO - epoch: 2
```

```
Training:   0%|          | 0/50 [00:00<?, ?it/s]
```

```
Validating:   0%|          | 0/13 [00:00<?, ?it/s]
```

```
2022-12-15 14:27:28:rastervision.pytorch_learner.learner: INFO - metrics:
{'avg_f1': 0.8449840545654297,
 'avg_precision': 0.9309152960777283,
 'avg_recall': 0.7735764980316162,
 'background_f1': 0.8657280802726746,
 'background_precision': 0.9788642525672913,
 'background_recall': 0.7760347723960876,
 'building_f1': 0.27820268273353577,
 'building_precision': 0.1715911626815796,
 'building_recall': 0.7346470952033997,
 'epoch': 2,
 'train_loss': 0.03501858711242676,
 'train_time': '0:00:11.307989',
 'val_loss': 0.05712978541851044,
 'valid_time': '0:00:04.134225'}
```

7.7.8 Train some more

[15]:
```
learner.train(epochs=3)
```

```
2022-12-15 14:27:28:rastervision.pytorch_learner.learner: INFO - Resuming training from␣
↪epoch 3
2022-12-15 14:27:28:rastervision.pytorch_learner.learner: INFO - epoch: 3
```

```
Training:   0%|          | 0/50 [00:00<?, ?it/s]
```

```
Validating:   0%|          | 0/13 [00:00<?, ?it/s]
```

```
2022-12-15 14:27:44:rastervision.pytorch_learner.learner: INFO - metrics:
{'avg_f1': 0.8460741639137268,
 'avg_precision': 0.9367697834968567,
 'avg_recall': 0.7713902592658997,
 'background_f1': 0.8635478019714355,
 'background_precision': 0.984494686126709,
 'background_recall': 0.7690666317939758,
 'building_f1': 0.29575374722480774,
```

(continues on next page)

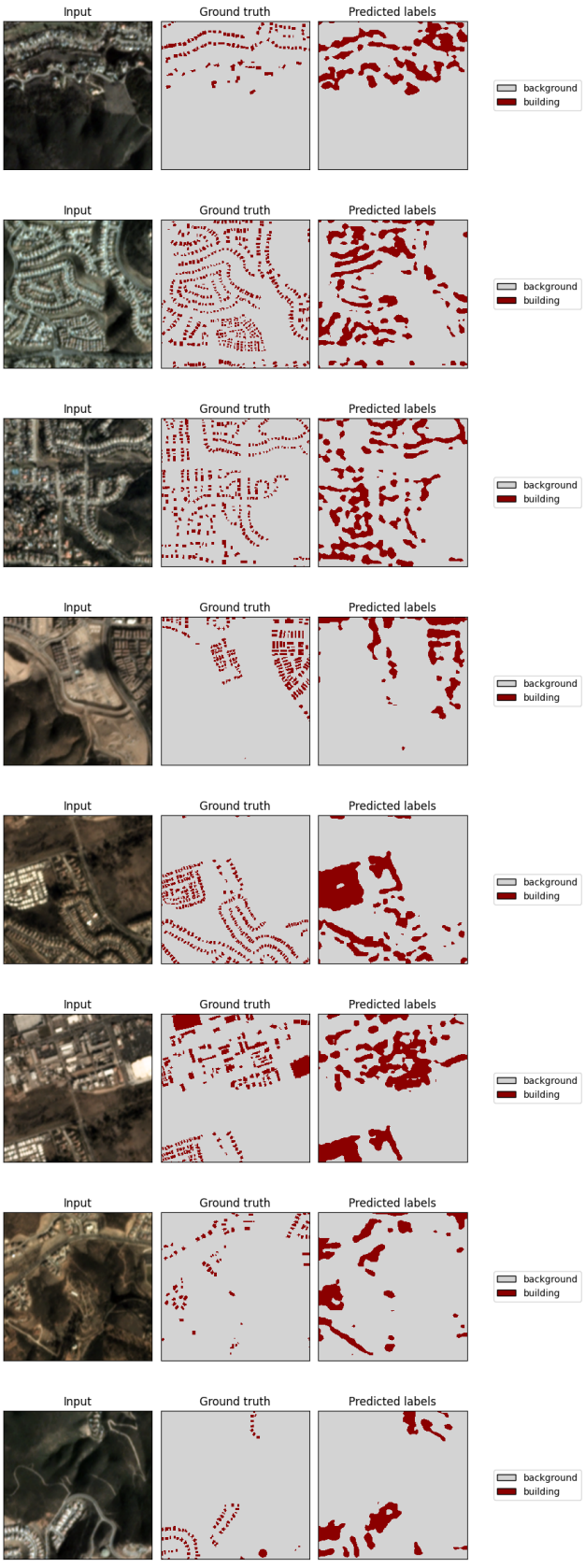
(continued from previous page)

```
 'building_precision': 0.18099400401115417,
 'building_recall': 0.8081868290901184,
 'epoch': 3,
 'train_loss': 0.03204505920410156,
 'train_time': '0:00:11.553398',
 'val_loss': 0.05406069755554199,
 'valid_time': '0:00:04.054214'}
2022-12-15 14:27:44:rastervision.pytorch_learner.learner: INFO - epoch: 4
Training:   0%|          | 0/50 [00:00<?, ?it/s]
Validating:   0%|          | 0/13 [00:00<?, ?it/s]
2022-12-15 14:28:00:rastervision.pytorch_learner.learner: INFO - metrics:
{'avg_f1': 0.832942545413971,
 'avg_precision': 0.9333340525627136,
 'avg_recall': 0.7520503401756287,
 'background_f1': 0.8505723476409912,
 'background_precision': 0.9818631410598755,
 'background_recall': 0.7502516508102417,
 'building_f1': 0.272173136472702,
 'building_precision': 0.16482366621494293,
 'building_recall': 0.7805342674255371,
 'epoch': 4,
 'train_loss': 0.030616183280944825,
 'train_time': '0:00:11.438546',
 'val_loss': 0.058418720960617065,
 'valid_time': '0:00:04.163509'}
2022-12-15 14:28:00:rastervision.pytorch_learner.learner: INFO - epoch: 5
Training:   0%|          | 0/50 [00:00<?, ?it/s]
Validating:   0%|          | 0/13 [00:00<?, ?it/s]
2022-12-15 14:28:16:rastervision.pytorch_learner.learner: INFO - metrics:
{'avg_f1': 0.8777509331703186,
 'avg_precision': 0.930891215801239,
 'avg_recall': 0.8303500413894653,
 'background_f1': 0.9030481576919556,
 'background_precision': 0.9763485193252563,
 'background_recall': 0.8399853110313416,
 'building_f1': 0.32184305787086487,
 'building_precision': 0.2110251784324646,
 'building_recall': 0.6777646541595459,
 'epoch': 5,
 'train_loss': 0.028649821281433105,
 'train_time': '0:00:11.639080',
 'val_loss': 0.053958915174007416,
 'valid_time': '0:00:04.503397'}
```

7.7.9 Examine predictions – `Learner.plot_predictions()`

[19]:
```
learner.plot_predictions(split='valid', show=True)
```

```
2022-10-21 14:00:43:rastervision.pytorch_learner.learner: INFO - Plotting predictions...
```



nbsphinx-code-borderwhite

```
<Figure size 640x480 with 0 Axes>
```

7.7.10 Save as a model-bundle – `Learner.save_model_bundle()`

Note the warning about `ModelConfig`. This is relevant when loading from from the bundle as we will see *below*.

[16]:
```
learner.save_model_bundle()
```

```
2022-12-15 14:28:17:rastervision.pytorch_learner.learner: WARNING - Model was not␣
↪configured via ModelConfig, and therefore, will not be reconstructable form the model-
↪bundle. You will need to initialize the model yourself and pass it to from_model_
↪bundle().
2022-12-15 14:28:17:rastervision.pytorch_learner.learner: INFO - Creating bundle.
2022-12-15 14:28:18:rastervision.pytorch_learner.learner: INFO - Saving bundle to ./
↪train-demo/model-bundle.zip.
```

7.7.11 Examine learner output

The trained model weights are saved at `./train-demo/last-model.pth` as well as inside the model-bundle.

[20]:
```
!apt-get install tree > "/dev/null"
!tree "./train-demo/"
```

```
./train-demo/
├── last-model.pth
├── learner-config.json
├── log.csv
├── model-bundle.zip
├── tb-logs
│   └── events.out.tfevents.1670411413.8a5ee9f3ced0.102.0
└── valid_preds.png

1 directory, 6 files
```

7.7.12 Using model-bundles

For predictions – `Learner.from_model_bundle()`

We can use the model-bundle to re-construct our `Learner` and then use it to make predictions.

Note: Since we used a custom model instead of using `ModelConfig`, the model-bundle does not know how to construct the model; therefore, we need to pass in the model again.

---

[17]:
```
from rastervision.pytorch_learner import SemanticSegmentationLearner

learner = SemanticSegmentationLearner.from_model_bundle(
    model_bundle_uri='./train-demo/model-bundle.zip',
    output_dir='./train-demo/',
```

(continues on next page)

(continued from previous page)

```
    model=model,
)
```

```
2022-12-15 14:28:21:rastervision.pytorch_learner.learner: INFO - Loading learner from␣
↪bundle ./train-demo/model-bundle.zip.
2022-12-15 14:28:21:rastervision.pytorch_learner.learner: INFO - Unzipping model-bundle␣
↪to /opt/data/tmp/tmp_9hxtbek/model-bundle
2022-12-15 14:28:21:rastervision.pytorch_learner.learner: INFO - Loading model weights␣
↪from: /opt/data/tmp/tmp_9hxtbek/model-bundle/model.pth
```

For next steps, see the *"Prediction and Evaluation" tutorial*.

For fine-tuning – `Learner.from_model_bundle()`

We can also re-construct the `Learner` in order to continue training, perhaps on a different dataset. To do this, we pass in `train_ds` and `val_ds` and set `training=True`

Note: Since we used a custom model instead of using `ModelConfig`, the model-bundle does not know how to construct the model; therefore, we need to pass in the model again.

Note: Optimizers and schedulers are (currently) not stored in model-bundles.

[18]:
```
from rastervision.pytorch_learner import SemanticSegmentationLearner

learner = SemanticSegmentationLearner.from_model_bundle(
    model_bundle_uri='./train-demo/model-bundle.zip',
    output_dir='./train-demo/',
    model=model,
    train_ds=train_ds,
    valid_ds=val_ds,
    training=True,
)
```

```
2022-12-15 14:28:22:rastervision.pytorch_learner.learner: INFO - Loading learner from␣
↪bundle ./train-demo/model-bundle.zip.
2022-12-15 14:28:22:rastervision.pytorch_learner.learner: INFO - Unzipping model-bundle␣
↪to /opt/data/tmp/tmpcx15mo9q/model-bundle
2022-12-15 14:28:22:rastervision.pytorch_learner.learner: INFO - Loading model weights␣
↪from: /opt/data/tmp/tmpcx15mo9q/model-bundle/model.pth
2022-12-15 14:28:22:rastervision.pytorch_learner.learner: INFO - Loading checkpoint from␣
↪./train-demo/last-model.pth
```

Continue training:

[19]:
```
learner.train(epochs=1)
```

```
2022-12-15 14:28:23:rastervision.pytorch_learner.learner: INFO - Resuming training from␣
↪epoch 6
2022-12-15 14:28:23:rastervision.pytorch_learner.learner: INFO - epoch: 6
```

```
Training:   0%|          | 0/50 [00:00<?, ?it/s]
Validating:   0%|          | 0/13 [00:00<?, ?it/s]
2022-12-15 14:28:39:rastervision.pytorch_learner.learner: INFO - metrics:
{'avg_f1': 0.812635600566864,
 'avg_precision': 0.9350194334983826,
 'avg_recall': 0.7185811996459961,
 'background_f1': 0.8263904452323914,
 'background_precision': 0.9844221472740173,
 'background_recall': 0.7120786905288696,
 'building_f1': 0.2574964761734009,
 'building_precision': 0.15267422795295715,
 'building_recall': 0.8215558528900146,
 'epoch': 6,
 'train_loss': 0.04115571975708008,
 'train_time': '0:00:11.777954',
 'val_loss': 0.05978838726878166,
 'valid_time': '0:00:04.738656'}
```

7.8 Prediction and Evaluation

7.8.1 Load a Learner with a trained model from bundle – Learner.from_model_bundle()

[1]:
```
bundle_uri = 'https://s3.amazonaws.com/azavea-research-public-data/raster-vision/
↪examples/model-zoo-0.20/spacenet-vegas-buildings-ss/train/model-bundle.zip'
```

[2]:
```
from rastervision.pytorch_learner import SemanticSegmentationLearner

learner = SemanticSegmentationLearner.from_model_bundle(bundle_uri, training=False)
```

```
2022-11-09 13:01:57:rastervision.pytorch_learner.learner: INFO - Loading learner from␣
↪bundle https://s3.amazonaws.com/azavea-research-public-data/raster-vision/examples/
↪model-zoo-0.13/spacenet-vegas-buildings-ss/train/model-bundle.zip.
2022-11-09 13:01:57:rastervision.pipeline.file_system.utils: INFO - Using cached file /
↪opt/data/tmp/cache/http/s3.amazonaws.com/azavea-research-public-data/raster-vision/
↪examples/model-zoo-0.13/spacenet-vegas-buildings-ss/train/model-bundle.zip.
2022-11-09 13:02:03:rastervision.pytorch_learner.learner: INFO - Local output dir: /opt/
↪data/tmp/tmpyt368dek/s3/raster-vision/examples/0.13/output/spacenet-vegas-buildings-ss/
↪train
2022-11-09 13:02:03:rastervision.pytorch_learner.learner: INFO - Remote output dir: s3://
↪raster-vision/examples/0.13/output/spacenet-vegas-buildings-ss/train
2022-11-09 13:02:06:rastervision.pytorch_learner.learner: INFO - Loading model weights␣
↪from: /opt/data/tmp/tmpyt368dek/model-bundle/model.pth
```

Note: If you used a custom model instead of using *ModelConfig* while training, you will need to initialize that model again and pass it to *Learner.from_model_bundle()*. See the *Training a model* tutorial for an example.

7.8.2 Get scene to predict

[3]:
```
scene_id = 5631
image_uri = f's3://spacenet-dataset/spacenet/SN2_buildings/train/AOI_2_Vegas/PS-RGB/SN2_
↪buildings_train_AOI_2_Vegas_PS-RGB_img{scene_id}.tif'
label_uri = f's3://spacenet-dataset/spacenet/SN2_buildings/train/AOI_2_Vegas/geojson_
↪buildings/SN2_buildings_train_AOI_2_Vegas_geojson_buildings_img{scene_id}.geojson'
```

[4]:
```
from rastervision.core.data import ClassConfig

class_config = ClassConfig(
    names=['building', 'background'],
    colors=['orange', 'black'])
class_config.ensure_null_class()
```

[5]:
```
from rastervision.core.data import ClassConfig
from rastervision.pytorch_learner import SemanticSegmentationSlidingWindowGeoDataset

import albumentations as A

ds = SemanticSegmentationSlidingWindowGeoDataset.from_uris(
    class_config=class_config,
    image_uri=image_uri,
    size=325,
    stride=325,
    transform=A.Resize(325, 325))
```

```
2022-11-09 13:02:35:rastervision.pipeline.file_system.utils: INFO - Using cached file /
↪opt/data/tmp/cache/s3/spacenet-dataset/spacenet/SN2_buildings/train/AOI_2_Vegas/PS-RGB/
↪SN2_buildings_train_AOI_2_Vegas_PS-RGB_img5631.tif.
2022-11-09 13:02:36:rastervision.core.data.raster_source.rasterio_source: WARNING -␣
↪Raster block size (2, 650) is too non-square. This can slow down reading. Consider re-
↪tiling using GDAL.
```

7.8.3 Predict – `Learner.predict_dataset()`

Make predictions via *`Learner.predict_dataset()`* and then turn them into *Labels* via *`Labels.from_predictions`* (specifically, *`SemanticSegmentationLabels.from_predictions`*).

[6]:
```
from rastervision.core.data import SemanticSegmentationLabels

predictions = learner.predict_dataset(
    ds,
    raw_out=True,
    numpy_out=True,
    predict_kw=dict(out_shape=(325, 325)),
    progress_bar=True)

pred_labels = SemanticSegmentationLabels.from_predictions(
    ds.windows,
    predictions,
    smooth=True,
```

(continues on next page)

(continued from previous page)

```
    extent=ds.scene.extent,
    num_classes=len(class_config))
```

```
Predicting:   0%|          | 0/4 [00:00<?, ?it/s]
```

7.8.4 Visualize predictions

`pred_labels` is an instance of *SemanticSegmentationSmoothLabels* which is a raster of probability distributions for each pixel for the entire scene. We can get these probabilities via *get_score_arr()*.

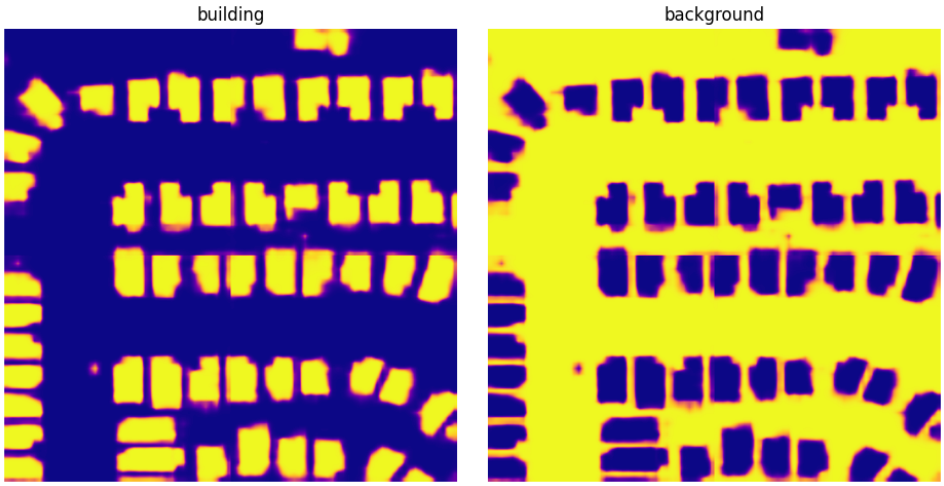
Note: There is also a *get_label_arr()* method that will return a 2D raster of class IDs representing the most probable class for each pixel.

---

[7]:
```
scores = pred_labels.get_score_arr(pred_labels.extent)
```

[8]:
```
from matplotlib import pyplot as plt

scores_building = scores[0]
scores_background = scores[1]

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5))
fig.tight_layout(w_pad=-2)
ax1.imshow(scores_building, cmap='plasma')
ax1.axis('off')
ax1.set_title('building')
ax2.imshow(scores_background, cmap='plasma')
ax2.axis('off')
ax2.set_title('background')
plt.show()
```

nbsphinx-code-borderwhite



7.8.5 Save predictions to file – SemanticSegmentationSmoothLabels.save()

[7]:
```
pred_labels.save(
    uri=f'./spacenet-vegas-buildings-ss/predict/{scene_id}',
    crs_transformer=ds.scene.raster_source.crs_transformer,
    class_config=class_config)
```

```
Saving pixel labels:   0%|          | 0/2 [00:00<?, ?it/s]
```

```
Saving pixel scores:   0%|          | 0/2 [00:00<?, ?it/s]
```

7.8.6 Evaluate predictions

We now want to evaluate the predictions against the ground truth labels.

Raster Vision allows us to do this via an *Evaluator*. In our case, this would be the *SemanticSegmentationEvaluator*. We are going to use its *evaluate_predictions()* method, which takes both ground truth labels and predictions as *Labels* objects.

We already have the predictions as a *SemanticSegmentationLabels* object, so we just need to load the ground truth labels as *SemanticSegmentationLabels* too. We do that by using *make_ss_scene()* factory function to create a scene and then accessing `scene.label_source.get_labels()`. Alternatively, we could have directly created a *SemanticSegmentationLabelSource*.

[10]:
```
from rastervision.core.data.utils import make_ss_scene

scene = make_ss_scene(
    class_config=class_config,
    image_uri=image_uri,
    label_vector_uri=label_uri,
    label_vector_default_class_id=class_config.get_class_id('building'),
    label_raster_source_kw=dict(
        background_class_id=class_config.get_class_id('background')),
    image_raster_source_kw=dict(allow_streaming=True))

gt_labels = scene.label_source.get_labels()
```

```
2022-10-21 10:06:26:rastervision.pipeline.file_system.utils: INFO - Using cached file /
↪opt/data/tmp/cache/s3/spacenet-dataset/spacenet/SN2_buildings/train/AOI_2_Vegas/
↪geojson_buildings/SN2_buildings_train_AOI_2_Vegas_geojson_buildings_img5631.geojson.
```

Note: `gt_labels` is an instance of *SemanticSegmentationDiscreteLabels*. You can convert it to a label raster (for visualization or some other analysis) via *get_label_arr()*.

---

[11]:
```
from rastervision.core.evaluation import SemanticSegmentationEvaluator

evaluator = SemanticSegmentationEvaluator(class_config)

evaluation = evaluator.evaluate_predictions(
    ground_truth=gt_labels, predictions=pred_labels)
```

SemanticSegmentationEvaluator.evaluate_predictions() returns a *SemanticSegmentationEvaluation* object which contains evaluations for each class as *ClassEvaluationItem* objects.

We can examine these evaluations as shown below.

Evaluation for the building class:

[12]:
```
evaluation.class_to_eval_item[0]
```

[12]:
```
{'class_id': 0,
 'class_name': 'building',
 'conf_mat': [[289042.0, 12212.0], [9351.0, 111895.0]],
 'conf_mat_dict': {'FN': 9351.0, 'FP': 12212.0, 'TN': 289042.0, 'TP': 111895.0},
 'conf_mat_frac': [[0.684123076923077, 0.02890414201183432],
                   [0.022132544378698226, 0.2648402366863905]],
 'conf_mat_frac_dict': {'FN': 0.022132544378698226,
                        'FP': 0.02890414201183432,
                        'TN': 0.684123076923077,
                        'TP': 0.2648402366863905},
 'count_error': 2861.0,
 'gt_count': 121246.0,
 'metrics': {'f1': 0.9121143821351279,
             'precision': 0.9016010378141442,
             'recall': 0.9228758062121637,
             'sensitivity': 0.9228758062121637,
             'specificity': 0.9594627789174583},
 'pred_count': 124107.0,
 'relative_frequency': 0.2869727810650888}
```

Evaluation for the background class:

[13]:
```
evaluation.class_to_eval_item[1]
```

[13]:
```
{'class_id': 1,
 'class_name': 'background',
 'conf_mat': [[111895.0, 9351.0], [12212.0, 289042.0]],
 'conf_mat_dict': {'FN': 12212.0, 'FP': 9351.0, 'TN': 111895.0, 'TP': 289042.0},
 'conf_mat_frac': [[0.2648402366863905, 0.022132544378698226],
                   [0.02890414201183432, 0.684123076923077]],
 'conf_mat_frac_dict': {'FN': 0.02890414201183432,
                        'FP': 0.022132544378698226,
                        'TN': 0.2648402366863905,
                        'TP': 0.684123076923077},
 'count_error': 2861.0,
 'gt_count': 301254.0,
 'metrics': {'f1': 0.9640405105003443,
             'precision': 0.9686621334950887,
             'recall': 0.9594627789174583,
             'sensitivity': 0.9594627789174583,
             'specificity': 0.9228758062121637},
 'pred_count': 298393.0,
 'relative_frequency': 0.7130272189349113}
```

Save evaluation

We can also save the evaluations as a JSON via *SemanticSegmentationEvaluation.save()*

[14]:
```
evaluation.save(f'eval-{scene_id}.json')
```

Advanced

7.9 Using Raster Vision with Lightning

Lightning (formerly known as PyTorch Lightning) is a high-level library for training PyTorch models. In this tutorial, we demonstrate a complete workflow for doing semantic segmentation on SpaceNet Vegas using a combination of Raster Vision and Lightning. We use Raster Vision for reading data, Lightning for training a model, and then Raster Vision again for making predictions and evaluations on whole scenes.

Raster Vision has easy-to-use, built-in model training functionality implemented by the `Learner` class which is shown in the *"Training a model" tutorial*. However, some users may prefer to use Lightning for training models, either because they already know how to use it, and like it, or because they desire more flexibility than the `Learner` class offers. This notebook shows how these libraries can be used together, but does not attempt to use either library in a particularly sophisticated manner.

First, we need to install `pytorch-lightning` since it is not a dependency of Raster Vision.

[ ]:
```
! pip install pytorch-lightning==1.8
```

7.9.1 Define training and validation datasets

We use Raster Vision to create training and validation `Dataset` objects. To keep things simple, we use a single scene for training and the same for validation. In a real workflow we would use many more scenes.

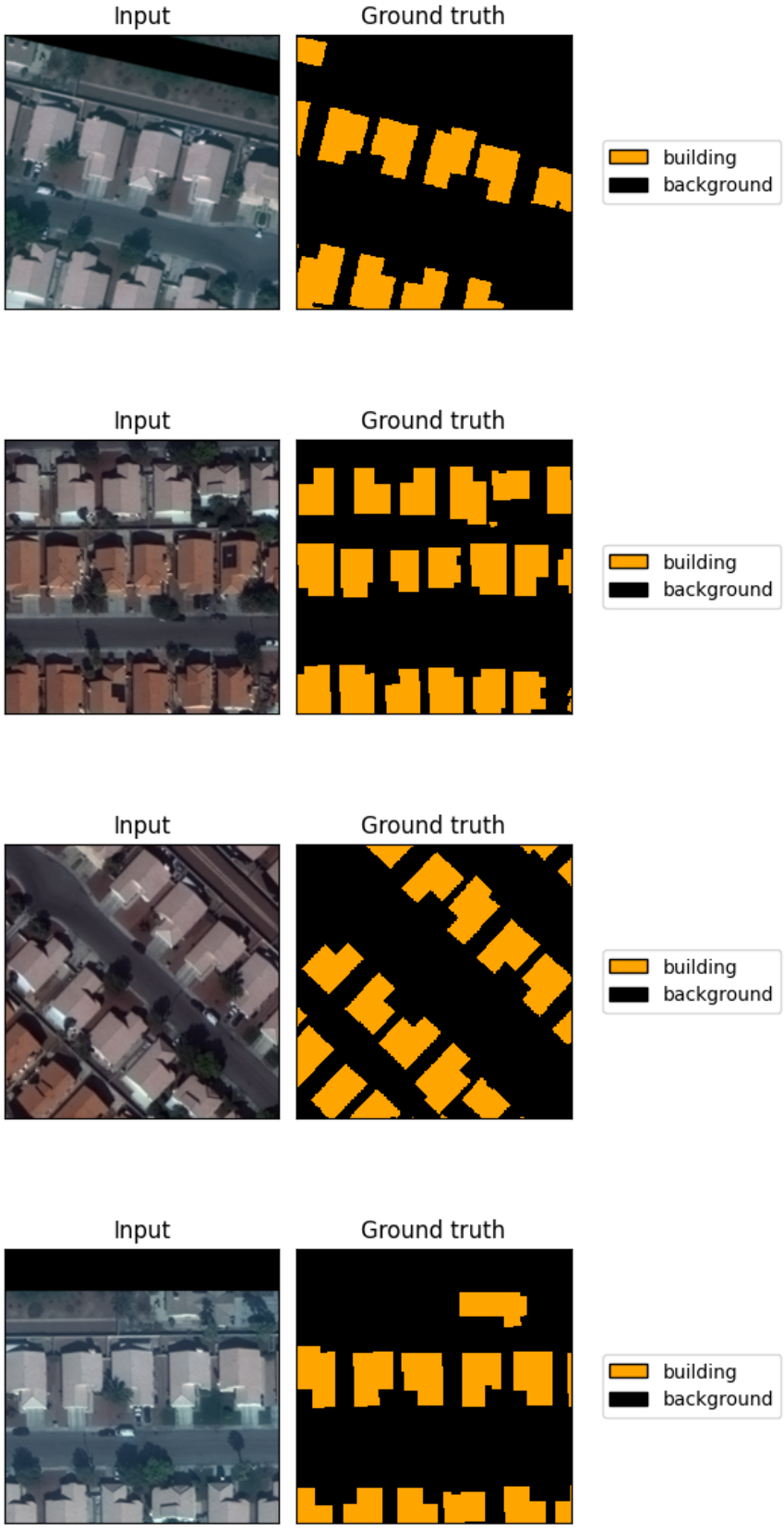
[15]:
```
import albumentations as A

from rastervision.pytorch_learner import (
    SemanticSegmentationRandomWindowGeoDataset,
    SemanticSegmentationSlidingWindowGeoDataset,
    SemanticSegmentationVisualizer)
from rastervision.core.data import ClassConfig
```

[20]:
```
scene_id = 5631
train_image_uri = f's3://spacenet-dataset/spacenet/SN2_buildings/train/AOI_2_Vegas/PS-
↪RGB/SN2_buildings_train_AOI_2_Vegas_PS-RGB_img{scene_id}.tif'
train_label_uri = f's3://spacenet-dataset/spacenet/SN2_buildings/train/AOI_2_Vegas/
↪geojson_buildings/SN2_buildings_train_AOI_2_Vegas_geojson_buildings_img{scene_id}.
↪geojson'

class_config = ClassConfig(
    names=['building', 'background'],
    colors=['orange', 'black'],
    null_class='background')

data_augmentation_transform = A.Compose([
    A.Flip(),
    A.ShiftScaleRotate(),
    A.RGBShift()
])

train_ds = SemanticSegmentationRandomWindowGeoDataset.from_uris(
    class_config=class_config,
    image_uri=train_image_uri,
    label_vector_uri=train_label_uri,
    label_vector_default_class_id=class_config.get_class_id('building'),
    size_lims=(300, 350),
    out_size=325,
    max_windows=10,
    transform=data_augmentation_transform)
```

```
2022-11-21 22:27:37:rastervision.pipeline.file_system.utils: INFO - Using cached file /
↪opt/data/tmp/cache/s3/spacenet-dataset/spacenet/SN2_buildings/train/AOI_2_Vegas/PS-RGB/
↪SN2_buildings_train_AOI_2_Vegas_PS-RGB_img5631.tif.
2022-11-21 22:27:37:rastervision.core.data.raster_source.rasterio_source: WARNING -␣
↪Raster block size (2, 650) is too non-square. This can slow down reading. Consider re-
↪tiling using GDAL.
2022-11-21 22:27:37:rastervision.pipeline.file_system.utils: INFO - Using cached file /
↪opt/data/tmp/cache/s3/spacenet-dataset/spacenet/SN2_buildings/train/AOI_2_Vegas/
↪geojson_buildings/SN2_buildings_train_AOI_2_Vegas_geojson_buildings_img5631.geojson.
```

To check that data is being read correctly, we use the `Visualizer` to plot a batch.

[21]:
```
viz = SemanticSegmentationVisualizer(
    class_names=class_config.names, class_colors=class_config.colors)
x, y = viz.get_batch(train_ds, 4)
viz.plot_batch(x, y, show=True)
```



nbsphinx-code-borderwhite

[22]:
```
scene_id = 5632
val_image_uri = f's3://spacenet-dataset/spacenet/SN2_buildings/train/AOI_2_Vegas/PS-RGB/
↪SN2_buildings_train_AOI_2_Vegas_PS-RGB_img{scene_id}.tif'
val_label_uri = f's3://spacenet-dataset/spacenet/SN2_buildings/train/AOI_2_Vegas/geojson_
↪buildings/SN2_buildings_train_AOI_2_Vegas_geojson_buildings_img{scene_id}.geojson'

val_ds = SemanticSegmentationSlidingWindowGeoDataset.from_uris(
    class_config=class_config,
    image_uri=val_image_uri,
    label_vector_uri=val_label_uri,
    label_vector_default_class_id=class_config.get_class_id('building'),
    size=325,
    stride=325)
```

```
2022-11-21 22:28:31:rastervision.pipeline.file_system.utils: INFO - Using cached file /
↪opt/data/tmp/cache/s3/spacenet-dataset/spacenet/SN2_buildings/train/AOI_2_Vegas/PS-RGB/
↪SN2_buildings_train_AOI_2_Vegas_PS-RGB_img5632.tif.
2022-11-21 22:28:31:rastervision.core.data.raster_source.rasterio_source: WARNING -␣
↪Raster block size (2, 650) is too non-square. This can slow down reading. Consider re-
↪tiling using GDAL.
2022-11-21 22:28:31:rastervision.pipeline.file_system.utils: INFO - Using cached file /
↪opt/data/tmp/cache/s3/spacenet-dataset/spacenet/SN2_buildings/train/AOI_2_Vegas/
↪geojson_buildings/SN2_buildings_train_AOI_2_Vegas_geojson_buildings_img5632.geojson.
```

7.9.2 Train Model using Lightning

Here we build a DeepLab-ResNet50 model, and then train it using Lightning. We only train for 3 epochs so this can run in a minute or so on a CPU. In a real workflow we would train for 10-100 epochs on GPU. Because of this, the model will not be accurate at all.

[27]:
```
from tqdm.autonotebook import tqdm
import torch
from torch.utils.data import DataLoader
from torch.nn import functional as F
from torchvision.models.segmentation import deeplabv3_resnet50
import pytorch_lightning as pl

from rastervision.pipeline.file_system import make_dir
```

[28]:
```
batch_size = 8
lr = 1e-4
epochs = 3
output_dir = './lightning-demo/'
make_dir(output_dir)
fast_dev_run = False
```

[29]:
```
train_dl = DataLoader(train_ds, batch_size=batch_size, shuffle=True, num_workers=4)
val_dl = DataLoader(val_ds, batch_size=batch_size, num_workers=4)
```

One of the main abstractions in Lightning is the `LightningModule` which extends a PyTorch `nn.Module` with extra methods that define how to train and validate the model. Here we define a `LightningModule` that does the bare minimum to train a DeepLab semantic segmentation model.

[8]:
```
class SemanticSegmentation(pl.LightningModule):
    def __init__(self, deeplab, lr=1e-4):
        super().__init__()
        self.deeplab = deeplab
        self.lr = lr

    def forward(self, img):
        return self.deeplab(img)['out']

    def training_step(self, batch, batch_idx):
        img, mask = batch
        img = img.float()
        mask = mask.long()
        out = self.forward(img)
        loss = F.cross_entropy(out, mask)
        log_dict = {'train_loss': loss}
        self.log_dict(log_dict, on_step=True, on_epoch=True, prog_bar=True, logger=True)
        return loss

    def validation_step(self, batch, batch_idx):
        img, mask = batch
        img = img.float()
        mask = mask.long()
        out = self.forward(img)
        loss = F.cross_entropy(out, mask)
        log_dict = {'validation_loss': loss}
        self.log_dict(log_dict, on_step=True, on_epoch=True, prog_bar=True, logger=True)
        return loss

    def configure_optimizers(self):
        optimizer = torch.optim.Adam(
            self.parameters(), lr=self.lr)
        return optimizer
```

The other main abstraction in Lighting is the `Trainer` which is responsible for actually training a `LightningModule`. This is configured to log metrics to Tensorboard.

[30]:
```
from pytorch_lightning.loggers import CSVLogger, TensorBoardLogger

deeplab = deeplabv3_resnet50(num_classes=len(class_config) + 1)
model = SemanticSegmentation(deeplab, lr=lr)
tb_logger = TensorBoardLogger(save_dir=output_dir, flush_secs=10)
trainer = pl.Trainer(
    accelerator='auto',
    min_epochs=1,
    max_epochs=epochs+1,
    default_root_dir=output_dir,
    logger=[tb_logger],
    fast_dev_run=fast_dev_run,
    log_every_n_steps=1,
)
```

```
GPU available: False, used: False
```

(continues on next page)

(continued from previous page)

```
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs
```

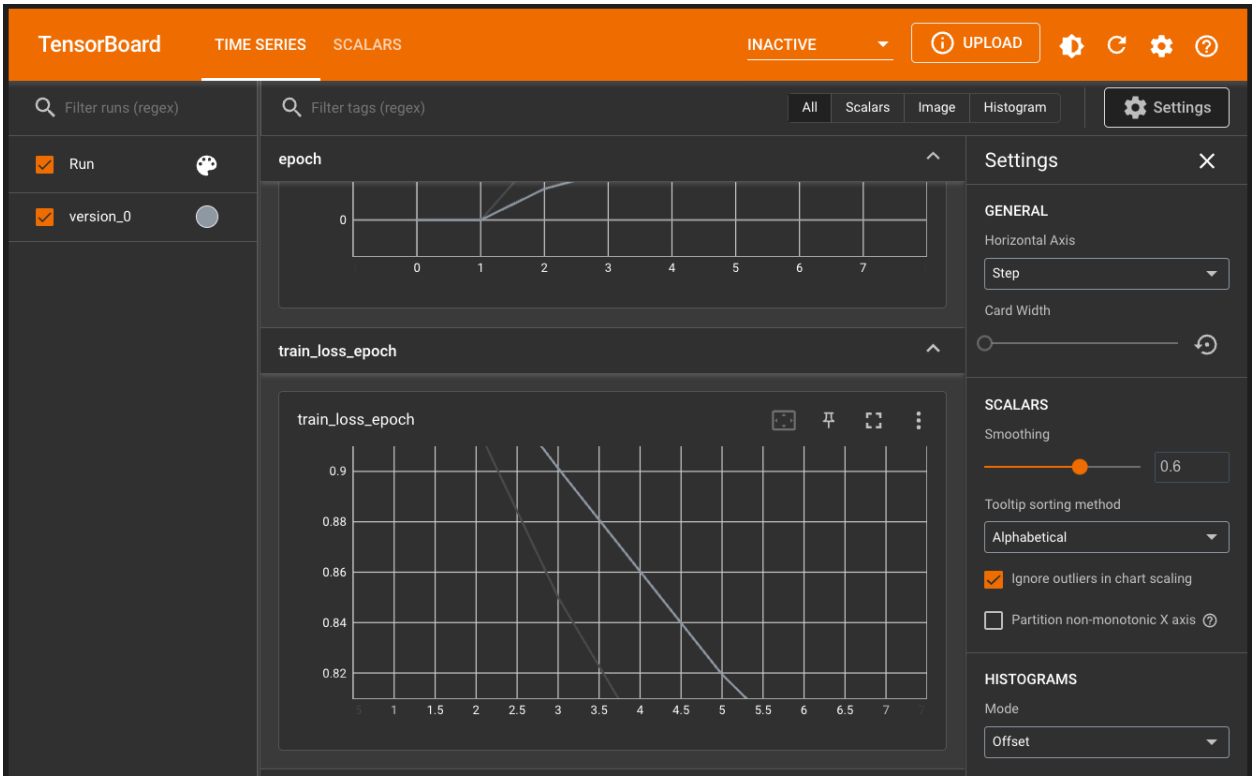
7.9.3 Monitor training using Tensorboard

This runs an instance of Tensorboard inside this notebook.

Note:

- If running inside the Raster Vision docker image, you will need to pass *–tensorboard* to *docker/run* for this to work.
- If the dashboard doen't auto-reload, you can click the reload button on the top-right.

[ ]:
```
%reload_ext tensorboard
%tensorboard --bind_all --logdir "./lightning-demo/lightning_logs" --reload_interval 10
```



[15]:
```
trainer.fit(model, train_dl, val_dl)
```

```
Missing logger folder: ./lightning-demo/lightning_logs

  | Name    | Type       | Params
-------------------------------------
0 | deeplab | DeepLabV3 | 39.6 M
-------------------------------------
```

(continues on next page)

(continued from previous page)

```
39.6 M    Trainable params
0         Non-trainable params
39.6 M    Total params
158.537   Total estimated model params size (MB)
```

```
A Jupyter Widget
```

```
A Jupyter Widget
```

```
A Jupyter Widget
```

```
A Jupyter Widget
```

```
A Jupyter Widget
```

```
A Jupyter Widget
```

```
`Trainer.fit` stopped: `max_epochs=4` reached.
```

7.9.4 Load saved model

After training the model for only 3 epochs, it will not make good predictions. In order to have some sensible looking output, we will loads weights from a model that was fully trained on SpaceNet Vegas.

[31]:
```
weights_uri = 'https://s3.amazonaws.com/azavea-research-public-data/raster-vision/
↪examples/model-zoo-0.20/spacenet-vegas-buildings-ss/model.pth'
deeplab.load_state_dict(torch.hub.load_state_dict_from_url(weights_uri, map_
↪location=torch.device('cpu')))
```

[31]:
```
<All keys matched successfully>
```

7.9.5 Make predictions for scene

We can now use Raster Vision's *SemanticSegmentationLabels* class to make predictions over a whole scene. The *SemanticSegmentationLabels.from_predictions()* method takes an iterator over predictions. We create this using a `get_predictions()` helper function defined below.

[32]:
```
def get_predictions(dataloader):
    for x, _ in tqdm(dataloader):
        with torch.inference_mode():
            out_batch = model(x)
        # This needs to yield a single prediction, not a whole batch of them.
        for out in out_batch:
            yield out.numpy()
```

[33]:
```
from rastervision.core.data import SemanticSegmentationLabels

model.eval()
predictions = get_predictions(val_dl)
pred_labels = SemanticSegmentationLabels.from_predictions(
    val_ds.windows,
    predictions,
    smooth=True,
```

(continues on next page)

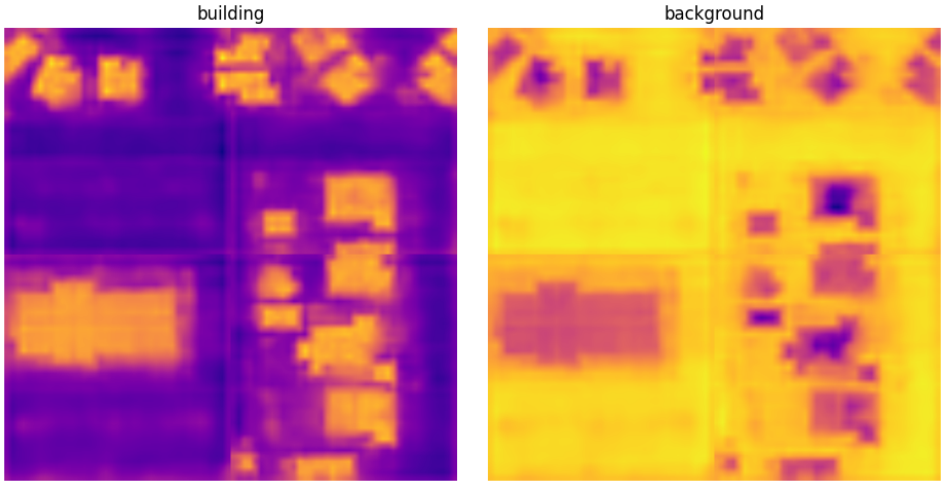
(continued from previous page)

```
    extent=val_ds.scene.extent,
    num_classes=len(class_config) + 1)
scores = pred_labels.get_score_arr(pred_labels.extent)
```

```
A Jupyter Widget
```

7.9.6 Visualize and then save predictions

[34]:
```
from matplotlib import pyplot as plt

scores_building = scores[0]
scores_background = scores[1]

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5))
fig.tight_layout(w_pad=-2)
ax1.imshow(scores_building, cmap='plasma')
ax1.axis('off')
ax1.set_title('building')
ax2.imshow(scores_background, cmap='plasma')
ax2.axis('off')
ax2.set_title('background')
plt.show()
```



nbsphinx-code-borderwhite

[19]:
```
from os.path import join

pred_labels.save(
    uri=join(output_dir, 'predictions'),
    crs_transformer=val_ds.scene.raster_source.crs_transformer,
    class_config=class_config)
```

```
A Jupyter Widget
```

```
A Jupyter Widget
```

7.9.7 Evaluate predictions for a scene

Now that we have predictions for the validation scene, we can evaluate them by comparing to ground truth using *SemanticSegmentationEvaluator*.

[20]:
```
from rastervision.core.evaluation import SemanticSegmentationEvaluator

evaluator = SemanticSegmentationEvaluator(class_config)

evaluation = evaluator.evaluate_predictions(
    ground_truth=val_ds.scene.label_source.get_labels(),
    predictions=pred_labels)
```

SemanticSegmentationEvaluator.evaluate_predictions() returns a *SemanticSegmentationEvaluation* object which contains evaluations for each class as *ClassEvaluationItem* objects.

Here are the metrics for the building and background classes.

[21]:
```
evaluation.class_to_eval_item[0]
```

[21]:
```
{'class_id': 0,
 'class_name': 'building',
 'conf_mat': [[336253.0, 5163.0], [3586.0, 77498.0]],
 'conf_mat_dict': {'FN': 3586.0, 'FP': 5163.0, 'TN': 336253.0, 'TP': 77498.0},
 'conf_mat_frac': [[0.7958650887573965, 0.012220118343195266],
                   [0.008487573964497041, 0.18342721893491123]],
 'conf_mat_frac_dict': {'FN': 0.008487573964497041,
                        'FP': 0.012220118343195266,
                        'TN': 0.7958650887573965,
                        'TP': 0.18342721893491123},
 'count_error': 1577.0,
 'gt_count': 81084.0,
 'metrics': {'f1': 0.946569360896516,
             'precision': 0.937540073311477,
             'recall': 0.955742587933501,
             'sensitivity': 0.955742587933501,
             'specificity': 0.9848776858729527},
 'pred_count': 82661.0,
 'relative_frequency': 0.19191479289940827}
```

[22]:
```
evaluation.class_to_eval_item[1]
```

[22]:
```
{'class_id': 1,
 'class_name': 'background',
 'conf_mat': [[77498.0, 3586.0], [5163.0, 336253.0]],
 'conf_mat_dict': {'FN': 5163.0, 'FP': 3586.0, 'TN': 77498.0, 'TP': 336253.0},
 'conf_mat_frac': [[0.18342721893491123, 0.008487573964497041],
                   [0.012220118343195266, 0.7958650887573965]],
 'conf_mat_frac_dict': {'FN': 0.012220118343195266,
                        'FP': 0.008487573964497041,
                        'TN': 0.18342721893491123,
                        'TP': 0.7958650887573965},
 'count_error': 1577.0,
 'gt_count': 341416.0,
 'metrics': {'f1': 0.9871575254493545,
```

(continues on next page)

(continued from previous page)

```
            'precision': 0.9894479444678215,
            'recall': 0.9848776858729527,
            'sensitivity': 0.9848776858729527,
            'specificity': 0.9557742587933501},
 'pred_count': 339839.0,
 'relative_frequency': 0.8080852071005917}
```

[24]:
```
evaluation.save(join(output_dir, 'evaluation.json'))
```

7.10 Working with pre-chipped datasets

It is not uncommon for geospatial datasets to be released in a "pre-chipped" form; i.e., not as large GeoTIFFs, but as non-georeferenced chips/tiles/patches in ordinary image formats such as PNG or JPEG.

In such scenarios, you do not necessarily need to use Raster Vision to read this data and train a model. In fact, you can train a model outside of Raster Vision and then use Raster Vision to run it over GeoTIFFs, as shown in the *"Using Raster Vision with Lightning"* tutorial.

Nevertheless, Raster Vision *is* capable of training with ordinary images and this tutorial notebook will walk you through examples for each of the three supported computer vision tasks. Note that this notebook only demonstrates how to read these datasets, but that is all that is needed; once you have instantiated these dataset classes, the rest of the *training* and *prediction* procedure is identical to that for non-chipped geo-referenced data.

7.10.1 The `ImageDataset` class

The *`ImageDataset`* is a PyTorch-compatible `Dataset` implementation that allows reading non-geospatial image datasets.

Just like the *`GeoDataset`* class, it is based on the *`AlbumentationsDataset`* class.

Supported image formats

Each ImageDataset subclass is capable of reading all image formats supported by pillow (`.png`, `.jp[e]g`, `.tif[f]`, `.bmp`, and more) plus the `numpy` format, `.npy`.

7.10.2 Semantic segmentation – `SemanticSegmentationImageDataset`

The *`SemanticSegmentationImageDataset`* (which internally uses a *`SemanticSegmentationDataReader`*) expects a path to a directory containing images and a path to a directory containing segmentation masks. Images are matched to their respective label-masks based on their filename (excluding the extension). The masks can be in any *supported image format*.

Dataset

For this example, we will use the Extended Optical Remote Sensing Saliency Detection (EORSSD) Dataset.

Zhang, Qijian, Runmin Cong, Chongyi Li, Ming-Ming Cheng, Yuming Fang, Xiaochun Cao, Yao Zhao, and Sam Kwong. "Dense attention fluid network for salient object detection in optical remote sensing images." IEEE Transactions on Image Processing 30 (2020): 1305-1317.

Below we download (63 MB) and unzip the data.

[14]:
```
!wget "https://github.com/rmcong/EORSSD-dataset/raw/master/EORSSD.zip"
!apt-get install unzip -y
!unzip -q "EORSSD.zip" -d "EORSSD"
!ls "EORSSD"
```

```
test-images  test-labels  train-images  train-labels
```

Usage

[2]:
```
import albumentations as A

from rastervision.pytorch_learner import SemanticSegmentationImageDataset

ds = SemanticSegmentationImageDataset(
    img_dir='EORSSD/train-images/',
    label_dir='EORSSD/train-labels/',
    transform=A.Resize(256, 256),
)
len(ds)
```

[2]:
```
1400
```

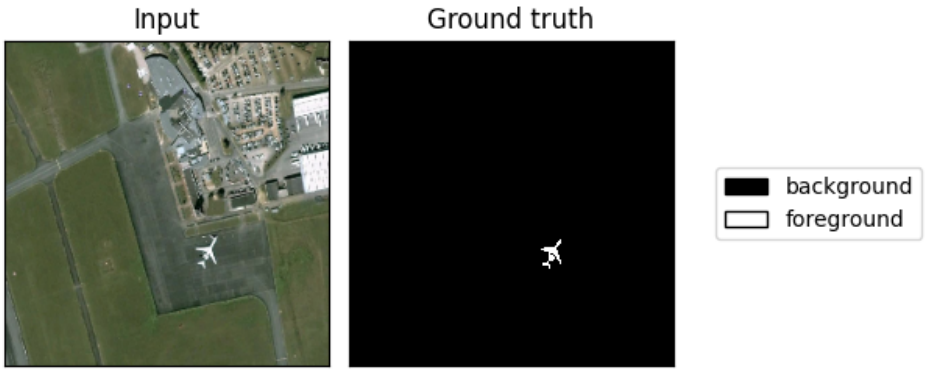
We can read a data sample and the corresponding ground truth from the Dataset like so:

[3]:
```
x, y = ds[0]
x.shape, y.shape
```

[3]:
```
(torch.Size([3, 256, 256]), torch.Size([256, 256]))
```

And then plot it using the *SemanticSegmentationVisualizer*:

[4]:
```
from rastervision.pytorch_learner import SemanticSegmentationVisualizer

viz = SemanticSegmentationVisualizer(
    class_names=['background', 'foreground'], class_colors=['black', 'white'])
viz.plot_batch(x.unsqueeze(0), y.unsqueeze(0), show=True)
```



nbsphinx-code-borderwhite

7.10.3 Object detection – `ObjectDetectionImageDataset`

The *`ObjectDetectionImageDataset`* (which internally uses a *`CocoDataset`*) expects a path to a directory containing images and a URI to a JSON file containing annotations in the COCO format.

Dataset

For this example, we will use the Airbus Aircraft Detection dataset.

You will need to manually download the dataset (92 MB) from here: https://www.kaggle.com/datasets/airbusgeo/airbus-aircrafts-sample-dataset.

The cells below assume that the data has been unzipped into an `airbus/` directory.

[1]:
```
!ls "airbus/"
```

```
LICENSE.txt  README.md  annotations.csv  annotations.json  extras  images
```

Note: This dataset cannot, strictly speaking, be called "pre-chipped" since the individual images are still pretty large and warrant further chipping. Nevertheless, the images are provided as non-georeferenced JPEGs and thus serve the purpose of this tutorial.

Transform annotations into COCO format

- Add a `bbox` column representing bounding boxes in `xywh` format and delete the old `geometry` column.
- Add a `category_id` column representing class IDs.

[1]:
```
import pandas as pd
from shapely.geometry import Polygon

from rastervision.core.box import Box
from rastervision.pipeline.file_system.utils import json_to_file
```

[31]:
```
class_names = ['Airplane', 'Truncated_airplane']
ann_df = pd.read_csv('airbus/annotations.csv')
ann_df.loc[:, 'bbox'] = [Box.from_shapely(Polygon(eval(g))).to_xywh() for g in ann_df.
↪geometry]
ann_df = ann_df.drop(columns='geometry')
ann_df.loc[:, 'category_id'] = [class_names.index(c) for c in ann_df['class']]
```

Convert to JSON and save to file:

[24]:
```
ann_json = {
    'images': [dict(id=image_id, file_name=image_id) for image_id in ann_df.image_id.
↪unique()],
    'annotations': ann_df.to_dict(orient='records'),
}
json_to_file(ann_json, 'airbus/annotations.json')
```

Usage

The annotations are now ready to be used:

[4]:
```
import albumentations as A

from rastervision.pytorch_learner import ObjectDetectionImageDataset

ds = ObjectDetectionImageDataset(
    img_dir='airbus/images/',
    annotation_uri='airbus/annotations.json',
    transform=A.Resize(1024, 1024),
)
len(ds)
```

[4]:
```
103
```

We can read a data sample and the corresponding ground truth from the Dataset like so:

[5]:
```
x, y = ds[0]
x.shape, y
```

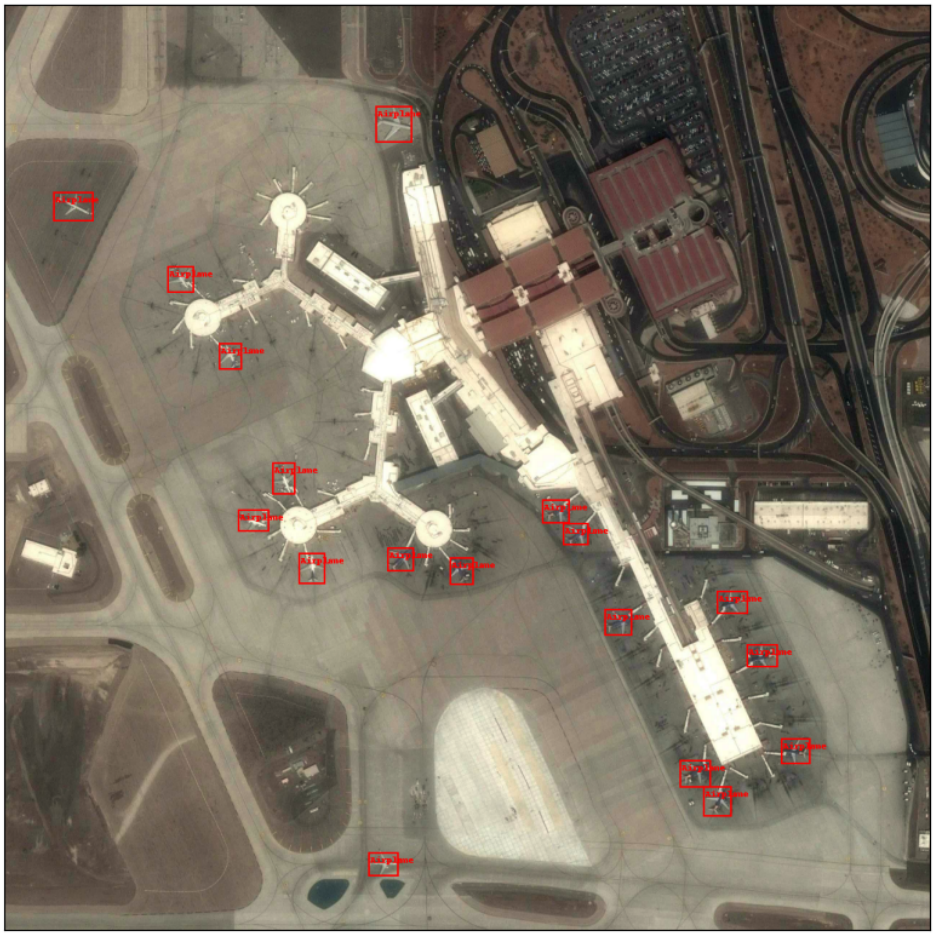
[5]:
```
(torch.Size([3, 1024, 1024]),
 <rastervision.pytorch_learner.object_detection_utils.BoxList at 0x7f22380d5460>)
```

Note that `y` is a *BoxList*.

And then plot it using the *ObjectDetectionVisualizer*:

[6]:
```
from rastervision.pytorch_learner import ObjectDetectionVisualizer

viz = ObjectDetectionVisualizer(
    class_names=['Airplane', 'Truncated_airplane'],
    class_colors=['red', 'green'])
viz.scale = 8
viz.plot_batch(x.unsqueeze(0), [y], show=True)
```

Input



nbsphinx-code-borderwhite

7.10.4 Classification – `ClassificationImageDataset`

The *`ClassificationImageDataset`* subclasses torchvision's `DatasetFolder` and expects a path to a directory containing images in the following structure:

```
<data_dir>/
    class_1/
        <images>
    class_2/
        <images>
    ...
    class_N/
        <images>
```

Dataset

For this example, we will use the EuroSat Dataset.

Helber, Patrick, Benjamin Bischke, Andreas Dengel, and Damian Borth. "Eurosat: A novel dataset and deep learning benchmark for land use and land cover classification." IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing 12, no. 7 (2019): 2217-2226.

Below we download (94 MB) and unzip the data.

[11]:
```
!wget "https://madm.dfki.de/files/sentinel/EuroSAT.zip"
!apt-get install unzip -y
!unzip -q "EuroSAT.zip" -d "EuroSAT"
!mv "EuroSAT/2750/"* "EuroSAT/" && rm -rf "EuroSAT/2750"
!ls "EuroSAT"
```

```
AnnualCrop  HerbaceousVegetation  Industrial  PermanentCrop  River
Forest      Highway               Pasture     Residential    SeaLake
```

Usage

[13]:
```
import albumentations as A

from rastervision.pytorch_learner import ClassificationImageDataset

ds = ClassificationImageDataset(
    data_dir='EuroSAT',
    # You can pass in a list explicitly if you want to enforce a specific
    # class-name to class-ID mapping.
    class_names=None,
    transform=A.Resize(256, 256),
)
len(ds)
```

[13]:
```
27000
```

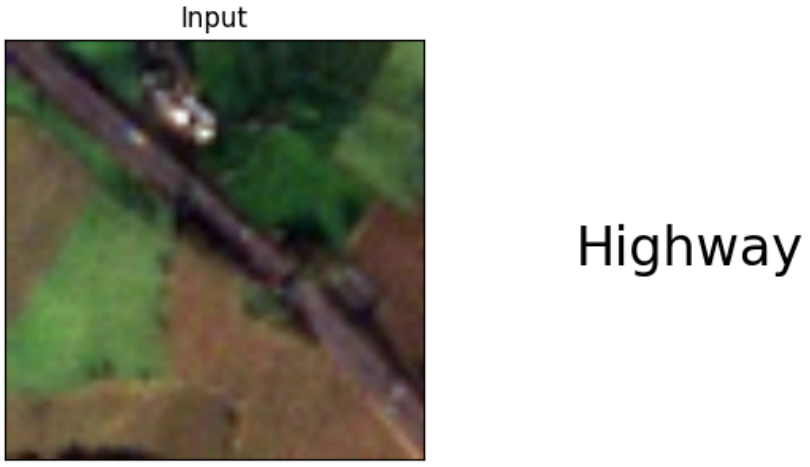
We can read a data sample and the corresponding ground truth from the Dataset like so:

[22]:
```
x, y = ds[10_000]
x.shape, y
```

[22]:
```
(torch.Size([3, 256, 256]), tensor(3))
```

And then plot it using the *ClassificationVisualizer*:

[23]:
```
from rastervision.pytorch_learner import ClassificationVisualizer

viz = ClassificationVisualizer(class_names=ds.orig_dataset.classes)
viz.plot_batch(x.unsqueeze(0), y.unsqueeze(0), show=True)
```



nbsphinx-code-borderwhite

CHAPTER

EIGHT

THE RASTER VISION PIPELINE

8.1 Quickstart

In this Quickstart, we'll train a semantic segmentation model on SpaceNet data. Don't get too excited - we'll only be training for a very short time on a very small training set! So the model that is created here will be pretty much worthless. But! These steps will show how Raster Vision pipelines are set up and run, so when you are ready to run against a lot of training data for a longer time on a GPU, you'll know what you have to do. Also, we'll show how to make predictions on the data using a model we've already trained on GPUs to show what you can expect to get out of Raster Vision.

For the Quickstart we are going to be using one of the published *Docker Images* as it has an environment with all necessary dependencies already installed.

See also:

It is also possible to install Raster Vision using `pip`, but it can be time-consuming and error-prone to install all the necessary dependencies. See *Installing via pip* for more details.

Note: This Quickstart requires a Docker installation. We have tested this with Docker 19, although you may be able to use a lower version. See Get Started with Docker for installation instructions.

You'll need to choose two directories, one for keeping your configuration source file and another for holding experiment output. Make sure these directories exist:

```
> export RV_QUICKSTART_CODE_DIR=`pwd`/code
> export RV_QUICKSTART_OUT_DIR=`pwd`/output
> mkdir -p ${RV_QUICKSTART_CODE_DIR} ${RV_QUICKSTART_OUT_DIR}
```

Now we can run a console in the the Docker container by doing

```
> docker run --rm -it \
    -v ${RV_QUICKSTART_CODE_DIR}:/opt/src/code  \
    -v ${RV_QUICKSTART_OUT_DIR}:/opt/data/output \
    quay.io/azavea/raster-vision:pytorch-0.20 /bin/bash
```

See also:

See *Docker Images* for more information about setting up Raster Vision with Docker images.

8.1.1 The Data

8.1.2 Configuring a semantic segmentation pipeline

Create a Python file in the `${RV_QUICKSTART_CODE_DIR}` named `tiny_spacenet.py`. Inside, you're going to write a function called `get_config` that returns a *SemanticSegmentationConfig* object. This object's type is a subclass of *PipelineConfig*, and configures a semantic segmentation pipeline which (optionally) analyzes the imagery, (optionally) creates training chips, trains a model, makes predictions on validation scenes, evaluates the predictions, and saves a model bundle.

Listing 1: tiny_spacenet.py

```
from os.path import join
from rastervision.core.rv_pipeline import *
from rastervision.core.backend import *
from rastervision.core.data import *
from rastervision.pytorch_backend import *
from rastervision.pytorch_learner import *


def get_config(runner) -> SemanticSegmentationConfig:
    output_root_uri = '/opt/data/output/'
    class_config = ClassConfig(
        names=['building', 'background'], colors=['red', 'black'])

    base_uri = ('https://s3.amazonaws.com/azavea-research-public-data/'
                'raster-vision/examples/spacenet')
    train_image_uri = join(base_uri, 'RGB-PanSharpen_AOI_2_Vegas_img205.tif')
    train_label_uri = join(base_uri, 'buildings_AOI_2_Vegas_img205.geojson')
    val_image_uri = join(base_uri, 'RGB-PanSharpen_AOI_2_Vegas_img25.tif')
    val_label_uri = join(base_uri, 'buildings_AOI_2_Vegas_img25.geojson')

    train_scene = make_scene('scene_205', train_image_uri, train_label_uri,
                             class_config)
    val_scene = make_scene('scene_25', val_image_uri, val_label_uri,
                           class_config)
    scene_dataset = DatasetConfig(
        class_config=class_config,
        train_scenes=[train_scene],
        validation_scenes=[val_scene])

    # Use the PyTorch backend for the SemanticSegmentation pipeline.
    chip_sz = 300

    backend = PyTorchSemanticSegmentationConfig(
        data=SemanticSegmentationGeoDataConfig(
            scene_dataset=scene_dataset,
            window_opts=GeoDataWindowConfig(
                # randomly sample training chips from scene
                method=GeoDataWindowMethod.random,
                # ... of size chip_sz x chip_sz
                size=chip_sz,
                # ... and at most 10 chips per scene
```

(continues on next page)

(continued from previous page)

```
                max_windows=10)),
        model=SemanticSegmentationModelConfig(backbone=Backbone.resnet50),
        solver=SolverConfig(lr=1e-4, num_epochs=1, batch_sz=2))

    return SemanticSegmentationConfig(
        root_uri=output_root_uri,
        dataset=scene_dataset,
        backend=backend,
        train_chip_sz=chip_sz,
        predict_chip_sz=chip_sz)


def make_scene(scene_id: str, image_uri: str, label_uri: str,
               class_config: ClassConfig) -> SceneConfig:
    """Define a Scene with image and labels from the given URIs."""

    raster_source = RasterioSourceConfig(
        uris=image_uri,
        # use only the first 3 bands
        channel_order=[0, 1, 2],
    )

    # configure GeoJSON reading
    vector_source = GeoJSONVectorSourceConfig(
        uri=label_uri,
        # This assumes the CRS is WGS-84 and ignores whatever the CRS specified
        # in the file is.
        ignore_crs_field=True,
        # The geoms in the label GeoJSON do not have a "class_id" property, so
        # classes must be inferred. Since all geoms are for the building class,
        # this is easy to do: we just assing the building class ID to all of
        # them.
        transformers=[
            ClassInferenceTransformerConfig(
                default_class_id=class_config.get_class_id('building'))
        ])
    # configure transformation of vector data into semantic segmentation labels
    label_source = SemanticSegmentationLabelSourceConfig(
        # semantic segmentation labels must be rasters, so rasterize the geoms
        raster_source=RasterizedSourceConfig(
            vector_source=vector_source,
            rasterizer_config=RasterizerConfig(
                # What about pixels outside of any geoms? Mark them as
                # background.
                background_class_id=class_config.get_class_id('background'))))

    return SceneConfig(
        id=scene_id,
        raster_source=raster_source,
        label_source=label_source,
    )
```

8.1.3 Running the pipeline

We can now run the pipeline by invoking the following command inside the container.

```
> rastervision run local code/tiny_spacenet.py
```

8.1.4 Seeing Results

If you go to `${RV_QUICKSTART_OUT_DIR}` you should see a directory structure like this.

Note: This uses the `tree` command which you may need to install first.

```
> tree -L 3
├── Makefile
├── bundle
│   └── model-bundle.zip
├── eval
│   └── validation_scenes
│       └── eval.json
├── pipeline-config.json
├── predict
│   └── scene_25
│       └── labels.tif
└── train
    ├── dataloaders
    │   ├── train.png
    │   └── valid.png
    ├── last-model.pth
    ├── learner-config.json
    ├── log.csv
    ├── model-bundle.zip
    ├── tb-logs
    │   ├── events.out.tfevents.1670510483.c5c1c7621fb7.1830.0
    │   ├── events.out.tfevents.1670511197.c5c1c7621fb7.2706.0
    │   └── events.out.tfevents.1670595249.986730ccbe70.7507.0
    ├── valid_metrics.json
    └── valid_preds.png
```

Note: The numbers in your `events.out.tfevents` filename will not necessarily match the ones above.

The root directory contains a serialized JSON version of the configuration at `pipeline-config.json`, and each subdirectory with a command name contains output for that command. You can see test predictions on a batch of data in `train/test_preds.png`, and evaluation metrics in `eval/eval.json`, but don't get too excited! We trained a model for 1 epoch on a tiny dataset, and the model is likely making random predictions at this point. We would need to train on a lot more data for a lot longer for the model to become good at this task.

8.1.5 Model Bundles

To immediately use Raster Vision with a fully trained model, one can make use of the pretrained models in our *Model Zoo*. However, be warned that these models probably won't work well on imagery taken in a different city, with a different ground sampling distance, or different sensor.

For example, to use a DeepLab/Resnet50 model that has been trained to do building segmentation on Las Vegas, one can type:

```
> rastervision predict \
    https://s3.amazonaws.com/azavea-research-public-data/raster-vision/examples/model-
↪zoo-0.20/spacenet-vegas-buildings-ss/model-bundle.zip \
    https://s3.amazonaws.com/azavea-research-public-data/raster-vision/examples/model-
↪zoo-0.20/spacenet-vegas-buildings-ss/sample-predictions/sample-img-spacenet-vegas-
↪buildings-ss.tif \
    prediction
```

This will make predictions on the image `sample-img-spacenet-vegas-buildings-ss.tif` using the provided model bundle. The predictions will be stored in the `prediction/` directory and will comprise a raster of predicted labels (`prediction/labels.tif`) and vectorized labels as GeoJSON files for each class (in `prediction/vector_outputs/`). These raster predictions are in the GeoTiff format, and you will need a GIS viewer such as QGIS to open them correctly on your device. Notice that the prediction package and the input raster are transparently downloaded via HTTP.

The input image (false color) and predictions are reproduced below.

See also:

You can read more about the *model bundles* and the *predict* CLI command in the documentation.

8.1.6 Next Steps

This is just a quick example of a Raster Vision pipeline. For several complete example of how to train models on open datasets (including SpaceNet), optionally using GPUs on AWS Batch, see the *Examples*.

8.2 Command Line Interface

The Raster Vision command line utility, `rastervision`, is installed with a `pip install` of `rastervision`, which is installed by default in the *Docker Images*. It has a main command, with some top level options, and several subcommands.

```
> rastervision --help

Usage: rastervision [OPTIONS] COMMAND [ARGS]...

The main click command.

Sets the profile, verbosity, and tmp_dir in RVConfig.

Options:
-p, --profile TEXT  Sets the configuration profile name to use.
-v, --verbose       Increment the verbosity level.
--tmpdir TEXT       Root of temporary directories to use.
--help              Show this message and exit.

Commands:
predict      Use a model bundle to predict on new images.
```

(continues on next page)

(continued from previous page)

```
run          Run sequence of commands within pipeline(s).
run_command  Run an individual command within a pipeline.
```

8.2.1 Subcommands

run

Run is the main interface into running pipelines.

```
> rastervision run --help

Usage: rastervision run [OPTIONS] RUNNER CFG_MODULE [COMMANDS]...

Run COMMANDS within pipelines in CFG_MODULE using RUNNER.

RUNNER: name of the Runner to use

CFG_MODULE: the module with `get_configs` function that returns
PipelineConfigs. This can either be a Python module path or a local path
to a .py file.

COMMANDS: space separated sequence of commands to run within pipeline. The
order in which to run them is based on the Pipeline.commands attribute. If
this is omitted, all commands will be run.

Options:
-a, --arg            KEY VALUE   Arguments to pass to get_config function
-s, --splits         INTEGER     Number of processes to run in parallel for splittable
                                 commands
--pipeline-run-name  TEXT        The name for this run of the pipeline.
--help                           Show this message and exit.
```

Some specific parameters to call out:

-\-splits

Use `-s N` or `--splits N`, where `N` is the number of splits to create, to parallelize commands that can be split into parallelizable chunks. See *Running Commands in Parallel* for more information.

run_command

The `run_command` is used to run a specific command from a serialized `PipelineConfig` JSON file. This is likely only interesting to people writing *custom runners*.

```
> rastervision run_command --help

Usage: rastervision run_command [OPTIONS] CFG_JSON_URI COMMAND

Run a single COMMAND using a serialized PipelineConfig in CFG_JSON_URI.
```

(continues on next page)

(continued from previous page)

```
Options:
--split-ind INTEGER   The process index of a split command
--num-splits INTEGER  The number of processes to use for running splittable
                        commands
--runner TEXT         Name of runner to use
--help                Show this message and exit.
```

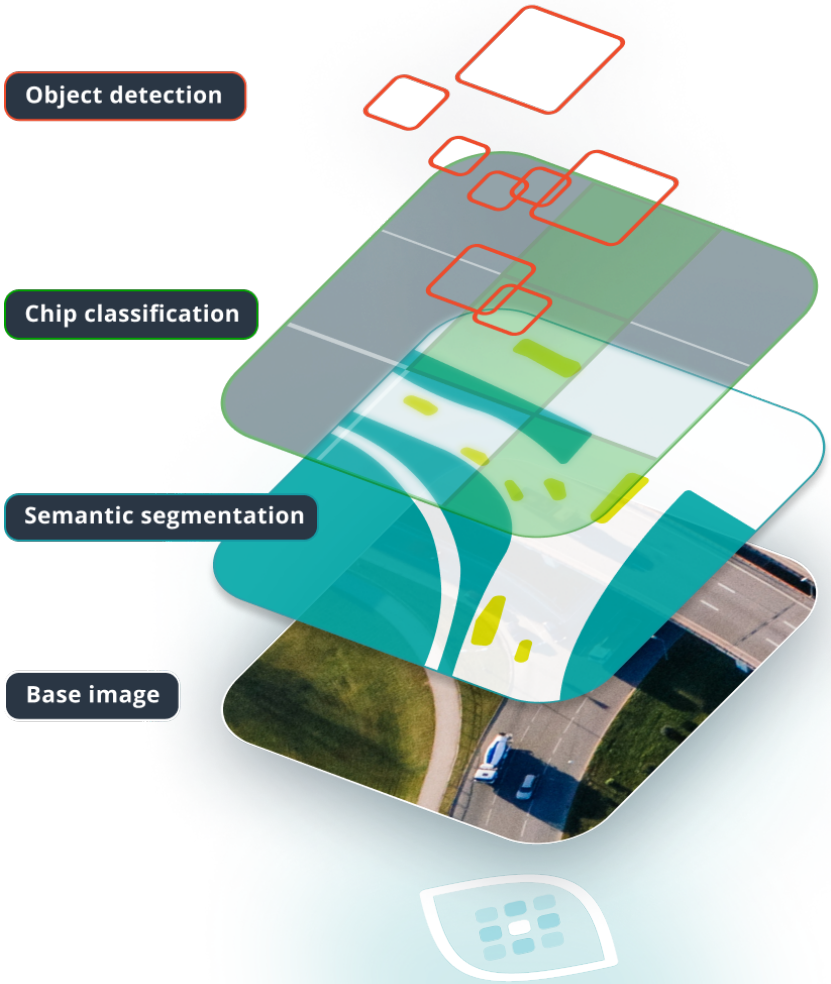
predict

Use `predict` to make predictions on new imagery given a *model bundle*.

```
> rastervision predict --help

Usage: rastervision predict [OPTIONS] MODEL_BUNDLE IMAGE_URI LABEL_URI

Make predictions on the images at IMAGE_URI using MODEL_BUNDLE and store
the prediction output at LABEL_URI.

Options:
-a, --update-stats       Run an analysis on this individual image, as
                        opposed to using any analysis like statistics that
                        exist in the prediction package
--channel-order TEXT     List of indices comprising channel_order. Example:
                        2 1 0
--help                   Show this message and exit.
```

8.3 Pipelines and Commands

In addition to providing abstract *pipeline* functionality, Raster Vision provides a set of concrete pipelines for deep learning on remote sensing imagery including *ChipClassification*, *SemanticSegmentation*, and *ObjectDetection*. These pipelines all derive from *RVPipeline*, and are provided by the *rastervision.core* package. It's possible to customize these pipelines as well as create new ones from scratch, which is discussed in *Customizing Raster Vision*.


Object detection
Chip classification
Semantic segmentation
Base image

8.3.1 Chip Classification

In chip classification, the goal is to divide the scene up into a grid of cells and classify each cell. This task is good for getting a rough idea of where certain objects are located, or where indiscrete "stuff" (such as grass) is located. It requires relatively low labeling effort, but also produces spatially coarse predictions. In our experience, this task trains the fastest, and is easiest to configure to get "decent" results.

8.3.2 Object Detection

In object detection, the goal is to predict a bounding box and a class around each object of interest. This task requires higher labeling effort than chip classification, but has the ability to localize and individuate objects. Object detection models require more time to train and also struggle with objects that are very close together. In theory, it is straightforward to use object detection for counting objects.

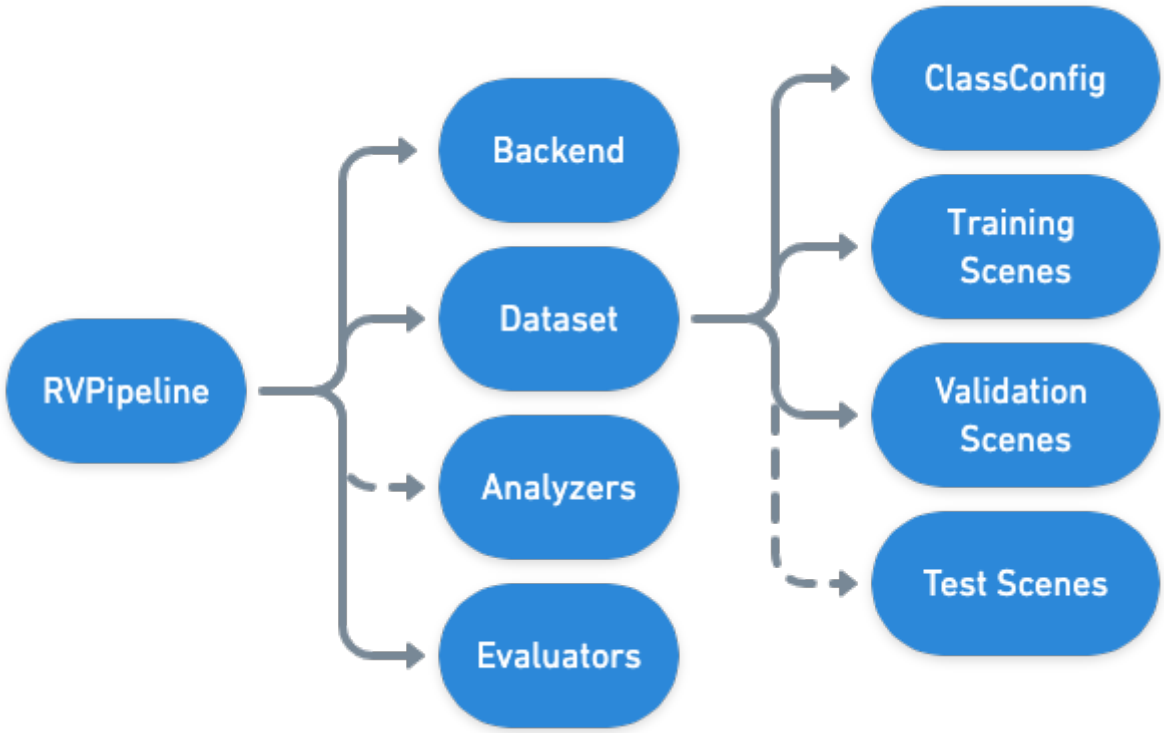
8.3.3 Semantic Segmentation

In semantic segmentation, the goal is to predict the class of each pixel in a scene. This task requires the highest labeling effort, but also provides the most spatially precise predictions. Like object detection, these models take longer to train than chip classification models.

8.3.4 Configuring RVPipelines

Each (subclass of) *RVPipeline* is configured by returning an instance of (a subclass of) *RVPipelineConfigs* from a `get_config()` function in a Python module. It's also possible to return a list of *RVPipelineConfigs* from `get_configs()`, which will be executed in parallel.

Each *RVPipelineConfig* object specifies the details about how the commands within the pipeline will execute (ie. which files, what methods, what hyperparameters, etc.). In contrast, the *pipeline runner*, which actually executes the commands in the pipeline, is specified as an argument to the *Command Line Interface*. The following diagram shows the hierarchy of the high level components comprising an *RVPipeline*:



In the tiny_spacenet.py example, the *SemanticSegmentationConfig* is the last thing constructed and returned from the `get_config` function.

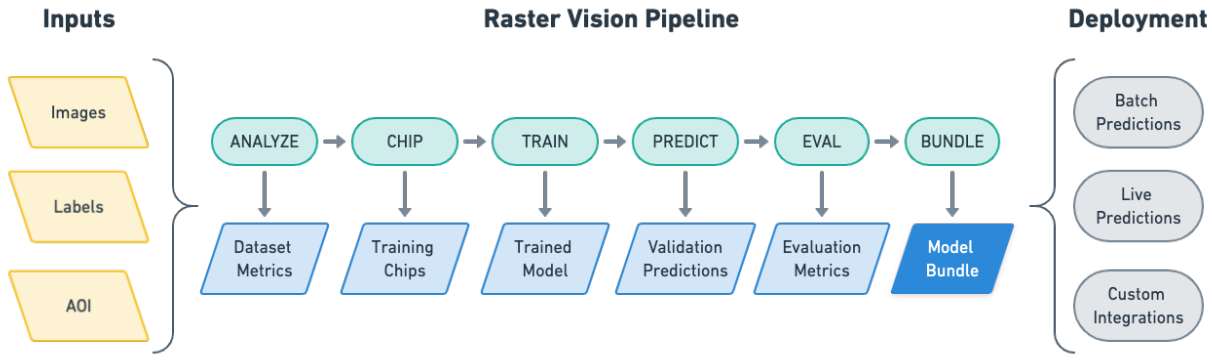
```
return SemanticSegmentationConfig(
    root_uri=output_root_uri,
    dataset=scene_dataset,
    backend=backend,
    train_chip_sz=chip_sz,
    predict_chip_sz=chip_sz)
```

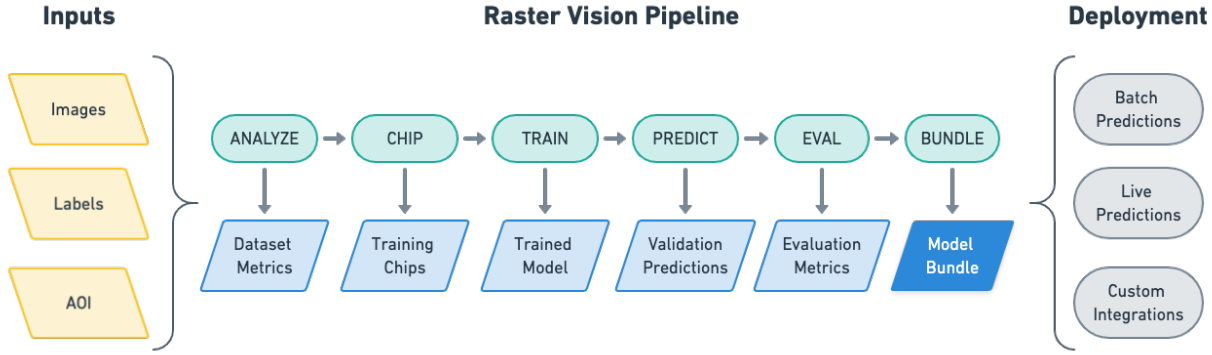
See also:

The *ChipClassificationConfig*, *SemanticSegmentationConfig*, and *ObjectDetectionConfig* API docs have more information on configuring pipelines.

8.3.5 Commands

The *RVPipelines* provide a sequence of commands, which are described below.





ANALYZE

The ANALYZE command is used to analyze scenes that are part of an experiment and produce some output that can be consumed by later commands. Geospatial raster sources such as GeoTIFFs often contain 16- and 32-bit pixel color values, but many deep learning libraries expect 8-bit values. In order to perform this transformation, we need to know the distribution of pixel values. So one usage of the ANALYZE command is to compute statistics of the raster sources and save them to a JSON file which is later used by the StatsTransformer (one of the available *RasterTransformers*) to do the conversion.

CHIP

Scenes comprise large geospatial raster sources (e.g. GeoTIFFs) and geospatial label sources (e.g. GeoJSONs), but models can only consume small images (i.e. chips) and labels in pixel based-coordinates. In addition, each *Backend* has its own dataset format. The CHIP command solves this problem by converting scenes into training chips and into a format the backend can use for training.

TRAIN

The TRAIN command is used to train a model using the dataset generated by the CHIP command. The command uses the *Backend* to run a training loop that saves the model and other artifacts each epoch. If the training command is interrupted, it will resume at the last epoch when restarted.

PREDICT

The PREDICT command makes predictions for a set of scenes using a model produced by the TRAIN command. To do this, a sliding window is used to feed small images into the model, and the predictions are transformed from image-centric, pixel-based coordinates into scene-centric, map-based coordinates.

EVAL

The EVAL command evaluates the quality of models by comparing the predictions generated by the PREDICT command to ground truth labels. A variety of metrics including F1, precision, and recall are computed for each class (as well as overall) and are written to a JSON file.

BUNDLE

The BUNDLE command generates a model bundle from the output of the previous commands which contains a model file plus associated configuration data. A model bundle can be used to make predictions on new imagery using the *predict* command.

8.3.6 Pipeline components

Below we describe some components of the pipeline that you might directly or indirectly configure.

A lot of these will be familiar from *Basic Concepts*, but note that when configuring a pipeline, instead of dealing with the classes directly, you will instead be configuring their `Config` counterparts.

The following table shows the corresponding `Configs` for various commonly used classes.

Class	Config	Notes
Scene	*SceneConfig*	*notes*
RasterSource • *RasterioSource* • *MultiRasterSource* • *RasterizedSource*	*RasterSourceConfig* • *RasterioSourceConfig* • *MultiRasterSourceConfig* • *RasterizedSourceConfig*	
RasterTransformer • *CastTransformer* • *MinMaxTransformer* • *NanTransformer* • *ReclassTransformer* • *RGBClassTransformer* • *StatsTransformer*	*RasterTransformerConfig* • *CastTransformerConfig* • *MinMaxTransformerConfig* • *NanTransformerConfig* • *ReclassTransformerConfig* • *RGBClassTransformerConfig* • *StatsTransformerConfig*	
VectorSource • *GeoJSONVectorSource*	*VectorSourceConfig* • *GeoJSONVectorSourceConfig*	
VectorTransformer • *BufferTransformer* • *ClassInferenceTransformer* • *ShiftTransformer*	*VectorTransformerConfig* • *BufferTransformerConfig* • *ClassInferenceTransformerConfig* • *ShiftTransformerConfig*	
LabelSource • *ChipClassificationLabelSource* • *SemanticSegmentationLabelSource* • *ObjectDetectionLabelSource*	*LabelSourceConfig* • *ChipClassificationLabelSourceConfig* • *SemanticSegmentationLabelSourceConfig* • *ObjectDetectionLabelSourceConfig*	
LabelStore • *ChipClassificationGeoJSONStore* • *ObjectDetectionGeoJSONStore* • *SemanticSegmentationLabelStore*	*LabelStoreConfig* • *ChipClassificationGeoJSONStoreConfig* • *ObjectDetectionGeoJSONStoreConfig* • *SemanticSegmentationLabelStoreConfig*	*notes*
Analyzer • *StatsAnalyzer*	*AnalyzerConfig* • *StatsAnalyzerConfig*	*notes*
Evaluator • *ChipClassificationEvaluator* • *SemanticSegmentationEvaluator* • *ObjectDetectionEvaluator*	*EvaluatorConfig* • *ChipClassificationEvaluatorConfig* • *SemanticSegmentationEvaluatorConfig* • *ObjectDetectionEvaluatorConfig*	*notes*

Backend

RVPipelines use a "backend" abstraction inspired by Keras, which makes it easier to customize the code for building and training models (including using Raster Vision with arbitrary deep learning libraries). Each backend is a subclass of *Backend* and has methods for saving training chips, training models, and making predictions, and is configured with a *~Backend*.

The *rastervision.pytorch_backend* plugin provides backends that are thin wrappers around the *rastervision.pytorch_learner* package, which does most of the heavy lifting of building and training models using torch and torchvision. (Note that *rastervision.pytorch_learner* is decoupled from *rastervision.pytorch_backend* so that it can be used in conjunction with *rastervision.pipeline* to write arbitrary computer vision pipelines that have nothing to do with remote sensing.)

Here are the PyTorch backends:

- The *PyTorchChipClassification* backend trains classification models from torchvision.
- The *PyTorchObjectDetection* backend trains the Faster-RCNN model in torchvision.
- The *PyTorchSemanticSegmentation* backend trains the DeepLabV3 model in torchvision.

In our tiny_spacenet.py example, we configured the PyTorch semantic segmentation backend using:

```
# Use the PyTorch backend for the SemanticSegmentation pipeline.
chip_sz = 300

backend = PyTorchSemanticSegmentationConfig(
    data=SemanticSegmentationGeoDataConfig(
        scene_dataset=scene_dataset,
        window_opts=GeoDataWindowConfig(
            # randomly sample training chips from scene
            method=GeoDataWindowMethod.random,
            # ... of size chip_sz x chip_sz
            size=chip_sz,
            # ... and at most 10 chips per scene
            max_windows=10)),
    model=SemanticSegmentationModelConfig(backbone=Backbone.resnet50),
    solver=SolverConfig(lr=1e-4, num_epochs=1, batch_sz=2))
```

See also:

rastervision.pytorch_backend and *rastervision.pytorch_learner* API docs for more information on configuring backends.

DatasetConfig

A *DatasetConfig* defines the training, validation, and test splits needed to train and evaluate a model. Each dataset split is a list of SceneConfigs.

In our tiny_spacenet.py example, we configured the dataset with single scenes, though more often in real use cases you would use multiple scenes per split:

```
train_scene = make_scene('scene_205', train_image_uri, train_label_uri,
                         class_config)
val_scene = make_scene('scene_25', val_image_uri, val_label_uri,
                       class_config)
scene_dataset = DatasetConfig(
```

(continues on next page)

(continued from previous page)

```
    class_config=class_config,
    train_scenes=[train_scene],
    validation_scenes=[val_scene])
```

Scene

A *Scene* is configured via a *SceneConfig* which is composed of the following elements:

- *Imagery*: a *RasterSourceConfig* represents a large scene image, which can be made up of multiple sub-images or a single file.
- *Ground truth labels*: a *LabelSourceConfig* represents ground-truth task-specific labels.
- *Predicted labels* (Optional): a *LabelStoreConfig* specifies how to store and retrieve the predictions from a scene.
- *AOIs* (Optional): An optional list of areas of interest that describes which sections of the scene imagery are exhaustively labeled. It is important to only create training chips from parts of the scenes that have been exhaustively labeled – in other words, that have no missing labels.

In our tiny_spacenet.py example, we configured the one training scene with a GeoTIFF URI and a GeoJSON URI.

```
def make_scene(scene_id: str, image_uri: str, label_uri: str,
               class_config: ClassConfig) -> SceneConfig:
    """Define a Scene with image and labels from the given URIs."""

    raster_source = RasterioSourceConfig(
        uris=image_uri,
        # use only the first 3 bands
        channel_order=[0, 1, 2],
    )

    # configure GeoJSON reading
    vector_source = GeoJSONVectorSourceConfig(
        uri=label_uri,
        # This assumes the CRS is WGS-84 and ignores whatever the CRS specified
        # in the file is.
        ignore_crs_field=True,
        # The geoms in the label GeoJSON do not have a "class_id" property, so
        # classes must be inferred. Since all geoms are for the building class,
        # this is easy to do: we just assing the building class ID to all of
        # them.
        transformers=[
            ClassInferenceTransformerConfig(
                default_class_id=class_config.get_class_id('building'))
        ])
    # configure transformation of vector data into semantic segmentation labels
    label_source = SemanticSegmentationLabelSourceConfig(
        # semantic segmentation labels must be rasters, so rasterize the geoms
        raster_source=RasterizedSourceConfig(
            vector_source=vector_source,
            rasterizer_config=RasterizerConfig(
                # What about pixels outside of any geoms? Mark them as
```

(continues on next page)

(continued from previous page)

```
                # background.
                background_class_id=class_config.get_class_id('background'))))

    return SceneConfig(
        id=scene_id,
        raster_source=raster_source,
        label_source=label_source,
    )
```

LabelStore

A *LabelStore* is configured via a *LabelStoreConfig*.

In the tiny_spacenet.py example, there is no explicit *LabelStore* configured on the validation scene, because it can be inferred from the type of *RVPipelineConfig* it is part of. In the isprs_potsdam.py example, the following code is used to explicitly create a *LabelStoreConfig* that, in turn, will be used to create a *LabelStore* that writes out the predictions in "RGB" format, where the color of each pixel represents the class, and predictions of class 0 (ie. car) are also written out as polygons.

```
label_store = SemanticSegmentationLabelStoreConfig(
    rgb=True, vector_output=[PolygonVectorOutputConfig(class_id=0)])

scene = SceneConfig(
    id=id,
    raster_source=raster_source,
    label_source=label_source,
    label_store=label_store)
```

Analyzer

Analyzers, configured via *AnalyzerConfigs*, are used to gather dataset-level statistics and metrics for use in downstream processes. Typically, you won't need to explicitly configure any.

Evaluator

For each computer vision task, there is an *Evaluator* (configured via the corresponding *EvaluatorConfig*) that computes metrics for a trained model. It does this by measuring the discrepancy between ground truth and predicted labels for a set of validation scenes. Typically, you won't need to explicitly configure any.

8.4 Examples

This page contains examples of using Raster Vision on open datasets. Unless otherwise stated, all commands should be run inside the Raster Vision Docker container. See *Docker Images* for info on how to do this.

8.4.1 How to Run an Example

There is a common structure across all of the examples which represents a best practice for defining experiments. Running an example involves the following steps.

- Acquire raw dataset.
- (Optional) Get processed dataset which is derived from the raw dataset, either using a Jupyter notebook, or by downloading the processed dataset.
- (Optional) Do an abbreviated test run of the experiment on a small subset of data locally.
- Run full experiment on GPU.
- Inspect output
- (Optional) Make predictions on new imagery

Each of the examples has several arguments that can be set on the command line:

- The input data for each experiment is divided into two directories: the raw data which is publicly distributed, and the processed data which is derived from the raw data. These two directories are set using the `raw_uri` and `processed_uri` arguments.
- The output generated by the experiment is stored in the directory set by the `root_uri` argument.
- The `raw_uri`, `processed_uri`, and `root_uri` can each be local or remote (on S3), and don't need to agree on whether they are local or remote.
- Experiments have a `test` argument which runs an abbreviated experiment for testing/debugging purposes.

In the next section, we describe in detail how to run one of the examples, SpaceNet Rio Chip Classification. For other examples, we only note example-specific details.

8.4.2 Chip Classification: SpaceNet Rio Buildings

This example performs chip classification to detect buildings on the Rio AOI of the SpaceNet dataset.

Step 1: Acquire Raw Dataset

The dataset is stored on AWS S3 at `s3://spacenet-dataset`. You will need an AWS account to access this dataset, but it will not be charged for accessing it. (To forward you AWS credentials into the container, use `docker/run --aws`).

Optional: to run this example with the data stored locally, first copy the data using something like the following inside the container.

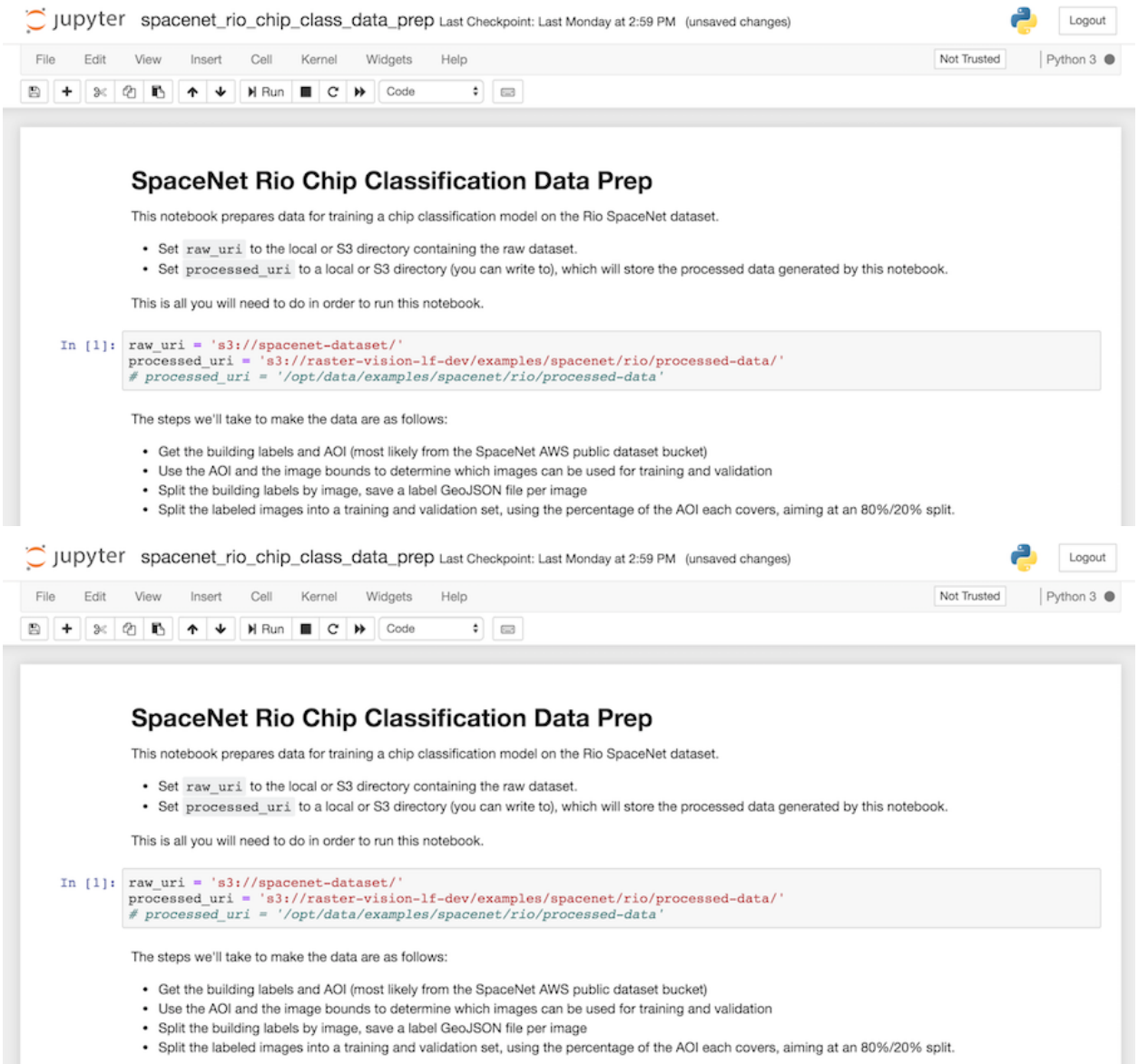
```
aws s3 sync s3://spacenet-dataset/AOIs/AOI_1_Rio/ /opt/data/spacenet-dataset/AOIs/AOI_1_
↪Rio/
```

Step 2: Run the Jupyter Notebook

You'll need to do some data preprocessing, which we can do in the Jupyter notebook supplied.

```
docker/run --jupyter [--aws]
```

The `--aws` option is only needed if pulling data from S3. In Jupyter inside the browser, navigate to the rastervision/examples/chip_classification/spacenet_rio_data_prep.ipynb notebook. Set the URIs in the first cell and then run the rest of the notebook. Set the `processed_uri` to a local or S3 URI depending on where you want to run the experiment.



jupyter spacenet_rio_chip_class_data_prep Last Checkpoint: Last Monday at 2:59 PM (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Not Trusted
Python 3
Run
Code
SpaceNet Rio Chip Classification Data Prep
This notebook prepares data for training a chip classification model on the Rio SpaceNet dataset.
Set raw_uri to the local or S3 directory containing the raw dataset.
Set processed_uri to a local or S3 directory (you can write to), which will store the processed data generated by this notebook.
This is all you will need to do in order to run this notebook.
In [1]:
raw_uri = 's3://spacenet-dataset/'
processed_uri = 's3://raster-vision-lf-dev/examples/spacenet/rio/processed-data/'
# processed_uri = '/opt/data/examples/spacenet/rio/processed-data'
The steps we'll take to make the data are as follows:
Get the building labels and AOI (most likely from the SpaceNet AWS public dataset bucket)
Use the AOI and the image bounds to determine which images can be used for training and validation
Split the building labels by image, save a label GeoJSON file per image
Split the labeled images into a training and validation set, using the percentage of the AOI each covers, aiming at an 80%/20% split.

Step 3: Do a test run locally

The experiment we want to run is in spacenet_rio.py. To run this, first get to the Docker console using:

```
docker/run [--aws] [--gpu] [--tensorboard]
```

The `--aws` option is only needed if running experiments on AWS or using data stored on S3. The `--gpu` option should only be used if running on a local GPU. The `--tensorboard` option should be used if running locally and you would like to view Tensorboard. The test run can be executed using something like:

```
export RAW_URI="s3://spacenet-dataset/"
export PROCESSED_URI="/opt/data/examples/spacenet/rio/processed-data"
export ROOT_URI="/opt/data/examples/spacenet/rio/local-output"

rastervision run local rastervision.examples.chip_classification.spacenet_rio \
    -a raw_uri $RAW_URI -a processed_uri $PROCESSED_URI -a root_uri $ROOT_URI \
    -a test True --splits 2
```

The sample above assumes that the raw data is on S3, and the processed data and output are stored locally. The `raw_uri` directory is assumed to contain an `AOIs/AOI_1_Rio` subdirectory. This runs two parallel jobs for the `chip` and `predict` commands via `--splits 2`. See `rastervision --help` and `rastervision run --help` for more usage information.

Note that when running with `-a test True`, some crops of the test scenes are created and stored in `processed_uri/crops/`. All of the examples that use big image files use this trick to make the experiment run faster in test mode.

After running this, the main thing to check is that it didn't crash, and that the visualization of training and validation chips look correct. These "debug chips" for each of the data splits can be found in `$ROOT_URI/train/dataloaders/`.

Step 4: Run full experiment

To run the full experiment on GPUs using AWS Batch, use something like the following. Note that all the URIs are on S3 since remote instances will not have access to your local file system.

```
export RAW_URI="s3://spacenet-dataset/"
export PROCESSED_URI="s3://mybucket/examples/spacenet/rio/processed-data"
export ROOT_URI="s3://mybucket/examples/spacenet/rio/remote-output"

rastervision run batch rastervision.examples.chip_classification.spacenet_rio \
    -a raw_uri $RAW_URI -a processed_uri $PROCESSED_URI -a root_uri $ROOT_URI \
    -a test False --splits 8
```

For instructions on setting up AWS Batch resources and configuring Raster Vision to use them, see *Running on AWS Batch*. To monitor the training process using Tensorboard, visit `<public dns>:6006` for the EC2 instance running the training job.

If you would like to run on a local GPU, replace `batch` with `local`, and use local URIs. To monitor the training process using Tensorboard, visit `localhost:6006`, assuming you used `docker/run --tensorboard`.

Step 5: Inspect results

After everything completes, which should take about 1.5 hours if you're running on AWS using a `p3.2xlarge` instance for training and 8 splits, you should be able to find the predictions over the validation scenes in `$root_uri/predict/`. The imagery and predictions are best viewed in QGIS, an example of which can be seen below. Cells that are predicted to contain buildings are red, and background are green.

The evaluation metrics can be found in `$root_uri/eval/eval.json`. This is an example of the scores from a run, which show an F1 score of 0.97 for detecting chips with buildings.


```
[
    {
        "precision": 0.9802512682554008,
        "recall": 0.9865974924340684,
        "f1": 0.9833968183611386,
        "count_error": 0.0,
        "gt_count": 2313.0,
        "class_id": 0,
        "class_name": "no_building"
    },
    {
```


(continues on next page)

(continued from previous page)


```
        "precision": 0.9789227645464389,
        "recall": 0.9685147159479809,
        "f1": 0.9736038795756798,
        "count_error": 0.0,
        "gt_count": 1461.0,
        "class_id": 1,
        "class_name": "building"
    },
    {
        "precision": 0.9797369746892128,
        "recall": 0.9795972443031267,
        "f1": 0.9796057522335405,
        "count_error": 0.0,
        "gt_count": 3774.0,
        "class_id": null,
        "class_name": "average"
    }
]
```


More evaluation details can be found here.

Step 6: Predict on new imagery

After running an experiment, a **model bundle** is saved into `$root_uri/bundle/`. This can be used to make predictions on new images. See the *Model Zoo* section for more details.

8.4.3 Semantic Segmentation: SpaceNet Vegas

This experiment contains an example of doing semantic segmentation using the SpaceNet Vegas dataset which has labels in vector form. It allows for training a model to predict buildings or roads. Note that for buildings, polygon output in the form of GeoJSON files will be saved to the `predict` directory alongside the GeoTIFF files.

Arguments:

- `raw_uri` should be set to the root of the SpaceNet data repository, which is at `s3://spacenet-dataset`, or a local copy of it. A copy only needs to contain the `AOIs/AOI_2_Vegas` subdirectory.
- `target` can be `buildings` or `roads`
- `processed_uri` should not be set because there is no processed data in this example.

Buildings

After training a model, the building F1 score is 0.91. More evaluation details can be found here.

Roads

After training a model, the road F1 score was 0.83. More evaluation details can be found here.

8.4.4 Semantic Segmentation: ISPRS Potsdam

This experiment performs semantic segmentation on the ISPRS Potsdam dataset. The dataset consists of 5cm aerial imagery over Potsdam, Germany, segmented into six classes including building, tree, low vegetation, impervious, car, and clutter. For more info see our blog post.

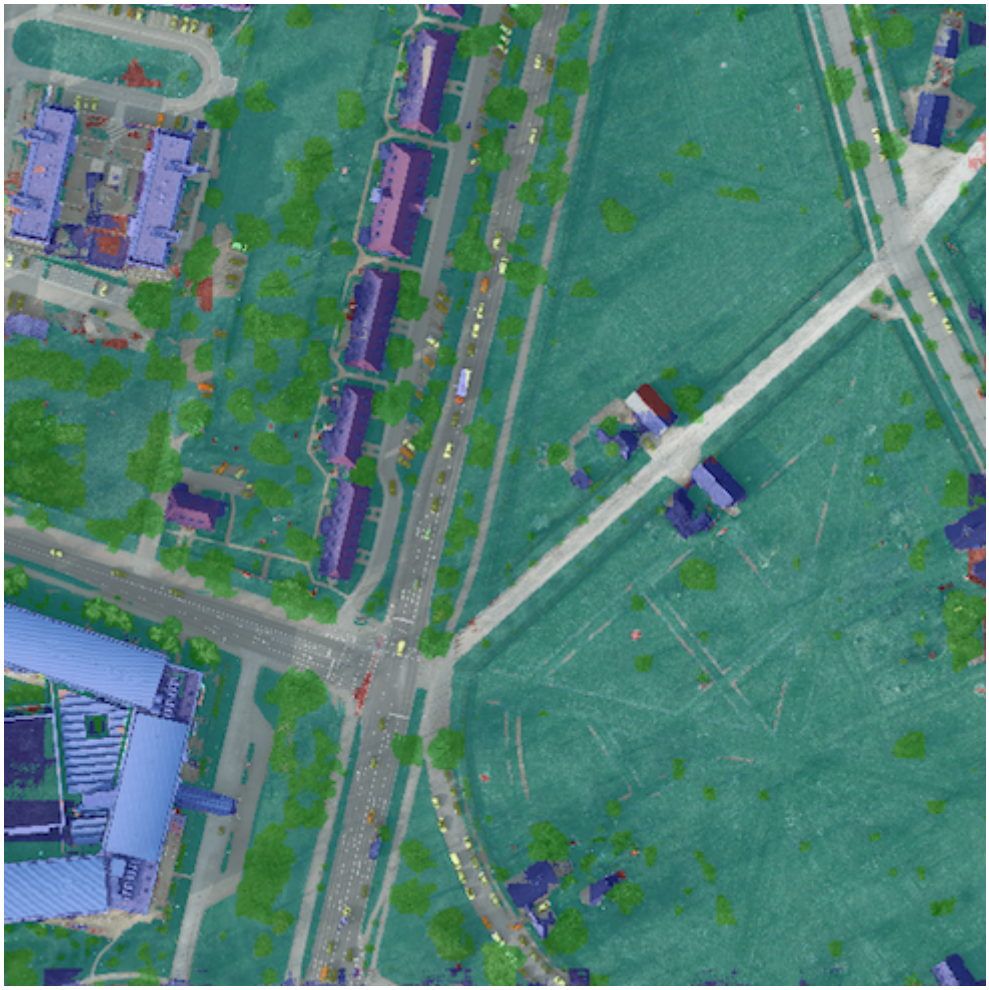
Data:

- The dataset can be downloaded from here. Access to files is password protected, but the password is provided on the same site. After downloading, unzip `4_Ortho_RGBIR.zip` and `5_Labels_for_participants.zip` into a directory, and then upload to S3 if desired.

Arguments:

- `raw_uri` should contain `4_Ortho_RGBIR` and `5_Labels_for_participants` subdirectories.
- `processed_uri` should be set to a directory which will be used to store test crops.

After training a model, the average F1 score was 0.89. More evaluation details can be found here.



8.4.5 Object Detection: COWC Potsdam Cars

This experiment performs object detection on cars with the Cars Overhead With Context dataset over Potsdam, Germany.

Data:

- The dataset can be downloaded from here. After downloading, unzip `4_Ortho_RGBIR.zip` into a directory, and then upload to S3 if desired. (This example uses the same imagery as *Semantic Segmentation: ISPRS Potsdam*.)
- Download the processed labels and unzip. These files were generated from the COWC car detection dataset using some scripts. TODO: Get these scripts into runnable shape.

Arguments:

- `raw_uri` should point to the imagery directory created above, and should contain the `4_Ortho_RGBIR` subdirectory.
- `processed_uri` should point to the labels directory created above. It should contain the `labels/all` subdirectory.

After training a model, the car F1 score was 0.95. More evaluation details can be found here.

8.4.6 Object Detection: xView Vehicles

This experiment performs object detection to find vehicles using the DIUx xView Detection Challenge dataset.

Data:

- Sign up for an account for the DIUx xView Detection Challenge. Navigate to the downloads page and download the zipped training images and labels. Unzip both of these files and place their contents in a directory, and upload to S3 if desired.
- Run the xview-data-prep.ipynb Jupyter notebook, pointing the `raw_uri` to the directory created above.

Arguments:

- The `raw_uri` should point to the directory created above, and contain a labels GeoJSON file named `xView_train.geojson`, and a directory named `train_images`.
- The `processed_uri` should point to the processed data generated by the notebook.

After training a model, the vehicle F1 score was 0.61. More evaluation details can be found here.

8.4.7 Model Zoo

Using the Model Zoo, you can download model bundles which contain pre-trained models and meta-data, and then run them on sample test images that the model wasn't trained on.

```
rastervision predict <model bundle> <infile> <outfile>
```

Note that the input file is assumed to have the same channel order and statistics as the images the model was trained on. See `rastervision predict --help` to see options for manually overriding these. It shouldn't take more than a minute on a CPU to make predictions for each sample. For some of the examples, there are also model files that can be used for fine-tuning on another dataset.

Disclaimer: These models are provided for testing and demonstration purposes and aren't particularly accurate. As is usually the case for deep learning models, the accuracy drops greatly when used on input that is outside the training distribution. In other words, a model trained on one city probably won't work well on another city (unless they are very similar) or at a different imagery resolution.

When unzipped, the model bundle contains a `model.pth` file which can be used for fine-tuning.

Note: The model bundles linked below are only compatible with Raster Vision version `0.20` or greater.

Table 1: Model Zoo

Dataset	Task	Model Type	Model Bundle	Sample Image
SpaceNet Rio Buildings	Chip Classification	Resnet 50	link	link
SpaceNet Vegas Buildings	Semantic Segmentation	DeeplabV3 / Resnet50	link	link
SpaceNet Vegas Roads	Semantic Segmentation	DeeplabV3 / Resnet50	link	link
ISPRS Potsdam	Semantic Segmentation	Panoptic FPN / Resnet50	link	link
COWC Potsdam (Cars)	Object Detection	Faster-RCNN / Resnet18	link	link
xView (Vehicles)	Object Detection	Faster-RCNN / Resnet50	link	link

8.5 Running Pipelines

Running pipelines in Raster Vision is done using the `rastervision` *run* command. This generates a pipeline configuration, serializes it, and then uses a runner to actually execute the commands, locally or remotely.

See also:

Pipelines and Commands explains more of the details of how `Pipelines` are implemented.

8.5.1 Running locally

local

A `rastervision run local ...` command will use the `LocalRunner`, which builds a `Makefile` based on the pipeline and executes it on the host machine. This will run multiple pipelines in parallel, as well as splittable commands in parallel, by spawning new processes for each command, where each process runs `rastervision run_command ...`.

inprocess

For debugging purposes, using `rastervision run inprocess` will run everything sequentially within a single process.

8.5.2 Running remotely

batch

Running `rastervision run batch ...` will submit a DAG (directed acyclic graph) of jobs to be run on AWS Batch, which will increase the instance count to meet the workload with low-cost spot instances, and terminate the instances when the queue of commands is finished. It can also run some commands on CPU instances (like `chip`), and others on GPU (like `train`), and will run multiple experiments in parallel, as well as splittable commands in parallel.

The `AWSBatchRunner` executes each command by submitting a job to Batch, which executes the `rastervision run_command` inside the Docker image configured in the Batch job definition. Commands that are dependent on an upstream command are submitted as a job after the upstream command's job, with the `jobId` of the upstream command

job as the parent `jobId`. This way Batch knows to wait to execute each command until all upstream commands are finished executing, and will fail the command if any upstream commands fail.

If you are running on AWS Batch or any other remote runner, you will not be able to use your local file system to store any of the data associated with an experiment.

Note: To run on AWS Batch, you'll need the proper setup. See *Running on AWS Batch* for instructions.

8.5.3 Running Commands in Parallel

Raster Vision can run certain commands in parallel, such as the *CHIP* and *PREDICT* commands. These commands are designated as `split_commands` in the corresponding `Pipeline` class. To run split commands in parallel, use the `--split` option to the *run* CLI command.

Splittable commands can be run in parallel, with each instance doing its share of the workload. For instance, using `--splits 5` on a `CHIP` command over 50 training scenes and 25 validation scenes will result in 5 CHIP commands running in parallel, that will each create chips for 15 scenes.

The command DAG that is given to the runner is constructed such that each split command can be run in parallel if the runner supports parallelization, and that any command that is dependent on the output of the split command will be dependent on each of the splits. So that means, in the above example, a `TRAIN` command, which was dependent on a single `CHIP` command pre-split, will be dependent each of the 5 individual `CHIP` commands after the split.

Each runner will handle parallelization differently. For instance, the local runner will run each of the splits simultaneously, so be sure the split number is in relation to the number of CPUs available. The AWS Batch runner will use array jobs to run commands in parallel, and the Batch Compute Environment will determine how many resources are available to run jobs simultaneously.

8.6 Architecture and Customization

8.6.1 Codebase Overview

The Raster Vision codebase is designed with modularity and flexibility in mind. There is a main, required package, *rastervision.pipeline*, which contains functionality for defining and configuring computational pipelines, running them in different environments using parallelism and GPUs, reading and writing to different file systems, and adding and customizing pipelines via a plugin mechanism. In contrast, the "domain logic" of geospatial deep learning using PyTorch, and running on AWS is contained in a set of optional plugin packages. All plugin packages must be under the `rastervision` native namespace package.

Each of these packages is contained in a separate `setuptools`/`pip` package with its own dependencies, including dependencies on other Raster Vision packages. This means that it's possible to install and use subsets of the functionality in Raster Vision. A short summary of the packages is as follows:

- *rastervision.pipeline*: define and run pipelines
- *rastervision.aws_s3*: read and write files on S3
- *rastervision.aws_batch*: run pipelines on Batch
- *rastervision.core*: chip classification, object detection, and semantic segmentation pipelines that work on geospatial data along with abstractions for running with different *backends* and data formats
- *rastervision.pytorch_learner*: model building and training code using `torch` and `torchvision`, which can be used independently of *rastervision.core*.

- *rastervision.pytorch_backend*: adds backends for the pipelines in *rastervision.core* using *rastervision.pytorch_learner* to do the heavy lifting

The figure below shows the packages, the dependencies between them, and important base classes within each package.

8.6.2 Writing pipelines and plugins

In this section, we explain the most important aspects of the *rastervision.pipeline* package through a series of examples which incrementally build on one another. These examples show how to write custom pipelines and configuration schemas, how to customize an existing pipeline, and how to package the code as a plugin.

The full source code for Examples 1 and 2 is in rastervision.pipeline_example_plugin1 and Example 3 is in rastervision.pipeline_example_plugin2 and they can be run from inside the Raster Vision Docker image. However, **note that new plugins are typically created in a separate repo and Docker image**, and *Bootstrap new projects with a template* shows how to do this.

Example 1: a simple pipeline

A *Pipeline* in Raster Vision is a class which represents a sequence of commands with a shared configuration in the form of a *PipelineConfig*. Here is a toy example of these two classes that saves a set of messages to disk, and then prints them all out.

Listing 2: rastervision.pipeline_example_plugin1.sample_pipeline

```
from typing import List, Optional
from os.path import join

from rastervision.pipeline.pipeline import Pipeline
from rastervision.pipeline.file_system import str_to_file, file_to_str
from rastervision.pipeline.pipeline_config import PipelineConfig
from rastervision.pipeline.config import register_config
from rastervision.pipeline.utils import split_into_groups


# Each Config needs to be registered with a type hint which is used for
# serializing and deserializing to JSON.
@register_config('pipeline_example_plugin1.sample_pipeline')
class SamplePipelineConfig(PipelineConfig):
    # Config classes are configuration schemas. Each field is an attributes
    # with a type and optional default value.
    names: List[str] = ['alice', 'bob']
    message_uris: Optional[List[str]] = None

    def build(self, tmp_dir):
        # The build method is used to instantiate the corresponding object
        # using this configuration.
        return SamplePipeline(self, tmp_dir)

    def update(self):
        # The update method is used to set default values as a function of
        # other values.
        if self.message_uris is None:
            self.message_uris = [
                join(self.root_uri, '{}.txt'.format(name))
                for name in self.names
            ]
```

(continues on next page)

(continued from previous page)

```
class SamplePipeline(Pipeline):
    # The order in which commands run. Each command correspond to a method.
    commands: List[str] = ['save_messages', 'print_messages']

    # Split commands can be split up and run in parallel.
    split_commands = ['save_messages']

    # GPU commands are run using GPUs if available. There are no commands worth running
    # on a GPU in this pipeline.
    gpu_commands = []

    def save_messages(self, split_ind=0, num_splits=1):
        # Save a file for each name with a message.

        # The num_splits is the number of parallel jobs to use and
        # split_ind tracks the index of the parallel job. In this case
        # we are splitting on the names/message_uris.
        split_groups = split_into_groups(
            list(zip(self.config.names, self.config.message_uris)), num_splits)
        split_group = split_groups[split_ind]

        for name, message_uri in split_group:
            message = 'hello {}!'.format(name)
            # str_to_file and most functions in the file_system package can
            # read and write transparently to different file systems based on
            # the URI pattern.
            str_to_file(message, message_uri)
            print('Saved message to {}'.format(message_uri))

    def print_messages(self):
        # Read all the message files and print them.
        for message_uri in self.config.message_uris:
            message = file_to_str(message_uri)
            print(message)
```

In order to run this, we need a separate Python file with a `get_config()` function which provides an instantiation of the `SamplePipelineConfig`.

Listing 3: rastervision.pipeline_example_plugin1.config1

```
from rastervision.pipeline_example_plugin1.sample_pipeline import (
    SamplePipelineConfig)


def get_config(runner, root_uri):
    # The get_config function returns an instantiated PipelineConfig and
    # plays a similar role as a typical "config file" used in other systems.
    # It's different in that it can have loops, conditionals, local variables,
    # etc. The runner argument is the name of the runner used to run the
    # pipeline (eg. local or batch). Any other arguments are passed from the
    # CLI using the -a option.
    names = ['alice', 'bob', 'susan']
```

(continues on next page)

(continued from previous page)

```
    # Note that root_uri is a field that is inherited from PipelineConfig,
    # the parent class of SamplePipelineConfig, and specifies the root URI
    # where any output files are saved.
    return SamplePipelineConfig(root_uri=root_uri, names=names)
```

Finally, in order to package this code as a plugin, and make it usable within the Raster Vision framework, it needs to be in a package directly under the `rastervision` namespace package, and have a top-level `__init__.py` file with a certain structure.

Listing 4: rastervision.pipeline_example_plugin1.__init__

```
# flake8: noqa


def register_plugin(registry):
    # Can be used to manually update the registry. Useful
    # for adding new FileSystems and Runners.
    pass


# Must import pipeline package first.
import rastervision.pipeline

# Then import any modules that add Configs so that the register_config decorators
# get called.
import rastervision.pipeline_example_plugin1.sample_pipeline
import rastervision.pipeline_example_plugin1.sample_pipeline2
```

We can invoke the Raster Vision CLI to run the pipeline using:

```
> rastervision run inprocess rastervision.pipeline_example_plugin1.config1 -a root_uri /
↪opt/data/pipeline-example/1/ -s 2

Running save_messages command split 1/2...
Saved message to /opt/data/pipeline-example/1/alice.txt
Saved message to /opt/data/pipeline-example/1/bob.txt
Running save_messages command split 2/2...
Saved message to /opt/data/pipeline-example/1/susan.txt
Running print_messages command...
hello alice!
hello bob!
hello susan!
```

This uses the `inprocess` runner, which executes all the commands in a single process locally (which is good for debugging), and uses the `LocalFileSystem` to read and write files. The `-s 2` option says to use two splits for splittable commands, and the `-a root_uri /opt/data/sample-pipeline` option says to pass the `root_uri` argument to the `get_config` function.

Example 2: hierarchical config

This example makes some small changes to the previous example, and shows how configurations can be built up hierarchically. However, the main purpose here is to lay the foundation for *Example 3: customizing an existing pipeline* which shows how to customize the configuration schema and behavior of this pipeline using a plugin. The changes to the previous example are highlighted with comments, but the overall effect is to delegate making messages to a `MessageMaker` class with its own `MessageMakerConfig` including a `greeting` field.

Listing 5: rastervision.pipeline_example_plugin1.sample_pipeline2

```
from typing import List, Optional
from os.path import join

from rastervision.pipeline.pipeline import Pipeline
from rastervision.pipeline.file_system import str_to_file, file_to_str
from rastervision.pipeline.pipeline_config import PipelineConfig
from rastervision.pipeline.config import register_config, Config
from rastervision.pipeline.utils import split_into_groups


@register_config('pipeline_example_plugin1.message_maker')
class MessageMakerConfig(Config):
    greeting: str = 'hello'

    def build(self):
        return MessageMaker(self)


class MessageMaker():
    def __init__(self, config):
        self.config = config

    def make_message(self, name):
        # Use the greeting field to make the message.
        return '{} {}!'.format(self.config.greeting, name)


@register_config('pipeline_example_plugin1.sample_pipeline2')
class SamplePipeline2Config(PipelineConfig):
    names: List[str] = ['alice', 'bob']
    message_uris: Optional[List[str]] = None
    # Fields can have other Configs as types.
    message_maker: MessageMakerConfig = MessageMakerConfig()

    def build(self, tmp_dir):
        return SamplePipeline2(self, tmp_dir)

    def update(self):
        if self.message_uris is None:
            self.message_uris = [
                join(self.root_uri, '{}.txt'.format(name))
                for name in self.names
            ]
```

(continues on next page)

(continued from previous page)

```
class SamplePipeline2(Pipeline):
    commands: List[str] = ['save_messages', 'print_messages']
    split_commands = ['save_messages']
    gpu_commands = []

    def save_messages(self, split_ind=0, num_splits=1):
        message_maker = self.config.message_maker.build()

        split_groups = split_into_groups(
            list(zip(self.config.names, self.config.message_uris)), num_splits)
        split_group = split_groups[split_ind]

        for name, message_uri in split_group:
            # Unlike before, we use the message_maker to make the message.
            message = message_maker.make_message(name)
            str_to_file(message, message_uri)
            print('Saved message to {}'.format(message_uri))

    def print_messages(self):
        for message_uri in self.config.message_uris:
            message = file_to_str(message_uri)
            print(message)
```

We can configure the pipeline using:

Listing 6: rastervision.pipeline_example_plugin1.config2

```
from rastervision.pipeline_example_plugin1.sample_pipeline2 import (
    SamplePipeline2Config, MessageMakerConfig)


def get_config(runner, root_uri):
    names = ['alice', 'bob', 'susan']
    # Same as before except we can set the greeting to be
    # 'hola' instead of 'hello'.
    message_maker = MessageMakerConfig(greeting='hola')
    return SamplePipeline2Config(
        root_uri=root_uri, names=names, message_maker=message_maker)
```

The pipeline can then be run with the above configuration using:

```
> rastervision run inprocess rastervision.pipeline_example_plugin1.config2 -a root_uri /
↪opt/data/pipeline-example/2/ -s 2

Running save_messages command split 1/2...
Saved message to /opt/data/pipeline-example/2/alice.txt
Saved message to /opt/data/pipeline-example/2/bob.txt
Running save_messages command split 2/2...
Saved message to /opt/data/pipeline-example/2/susan.txt
Running print_messages command...
```

(continues on next page)

(continued from previous page)

```
hola alice!
hola bob!
hola susan!
```

Example 3: customizing an existing pipeline

This example shows how to customize the behavior of an existing pipeline, namely the `SamplePipeline2` developed in *Example 2: hierarchical config*. That pipeline delegates printing messages to a `MessageMaker` class which is configured by `MessageMakerConfig`. Our goal here is to make it possible to control the number of exclamation points at the end of the message.

By writing a plugin (ie. a plugin to the existing plugin that was developed in the previous two examples), we can add new behavior without modifying any of the original source code from *Example 2: hierarchical config*. This mimics the situation plugin writers will be in when they want to modify the behavior of one of the *geospatial deep learning pipelines* without modifying the source code in the main Raster Vision repo.

The code to implement the new configuration and behavior, and a sample configuration are below. (We omit the `__init__.py` file since it is similar to the one in the previous plugin.) Note that the new `DeluxeMessageMakerConfig` uses inheritance to extend the configuration schema.

Listing 7: rastervision.pipeline_example_plugin2.deluxe_message_maker

```
from rastervision.pipeline.config import register_config
from rastervision.pipeline_example_plugin1.sample_pipeline2 import (
    MessageMakerConfig, MessageMaker)


# You always need to use the register_config decorator.
@register_config('pipeline_example_plugin2.deluxe_message_maker')
class DeluxeMessageMakerConfig(MessageMakerConfig):
    # Note that this inherits the greeting field from MessageMakerConfig.
    level: int = 1

    def build(self):
        return DeluxeMessageMaker(self)


class DeluxeMessageMaker(MessageMaker):
    def make_message(self, name):
        # Uses the level field to determine the number of exclamation marks.
        exclamation_marks = '!' * self.config.level
        return '{} {}{}'.format(self.config.greeting, name, exclamation_marks)
```

Listing 8: rastervision.pipeline_example_plugin2.config3

```
from rastervision.pipeline_example_plugin1.sample_pipeline2 import (
    SamplePipeline2Config)
from rastervision.pipeline_example_plugin2.deluxe_message_maker import (
    DeluxeMessageMakerConfig)


def get_config(runner, root_uri):
```

(continues on next page)

(continued from previous page)

```
    names = ['alice', 'bob', 'susan']
    # Note that we use the DeluxeMessageMakerConfig and set the level to 3.
    message_maker = DeluxeMessageMakerConfig(greeting='hola', level=3)
    return SamplePipeline2Config(
        root_uri=root_uri, names=names, message_maker=message_maker)
```

We can run the pipeline as follows:

```
> rastervision run inprocess rastervision.pipeline_example_plugin2.config3 -a root_uri /
↪opt/data/pipeline-example/3/ -s 2

Running save_messages command split 1/2...
Saved message to /opt/data/pipeline-example/3/alice.txt
Saved message to /opt/data/pipeline-example/3/bob.txt
Running save_messages command split 2/2...
Saved message to /opt/data/pipeline-example/3/susan.txt
Running print_messages command...
hola alice!!!
hola bob!!!
hola susan!!!
```

The output in `/opt/data/sample-pipeline` contains a `pipeline-config.json` file which is the serialized version of the `SamplePipeline2Config` created in `config3.py`. The serialized configuration is used to transmit the configuration when running a pipeline remotely. It also is a programming language-independent record of the fully-instantiated configuration that was generated by the `run` command in conjunction with any command line arguments. Below is the partial contents of this file. The interesting thing to note here is the `type_hint` field that appears twice. This is what allows the JSON to be deserialized back into the Python classes that were originally used.(Recall that the `register_config` decorator is what tells the `Registry` the type hint for each *`Config`* class.)

```
{
    "root_uri": "/opt/data/sample-pipeline",
    "type_hint": "sample_pipeline2",
    "names": [
        "alice",
        "bob",
        "susan"
    ],
    "message_uris": [
        "/opt/data/sample-pipeline/alice.txt",
        "/opt/data/sample-pipeline/bob.txt",
        "/opt/data/sample-pipeline/susan.txt"
    ],
    "message_maker": {
        "greeting": "hola",
        "type_hint": "deluxe_message_maker",
        "level": 3
    }
}
```

We now have a plugin that customizes an existing pipeline! Being a toy example, this may all seem like overkill. Hopefully, the real power of the `pipeline` package becomes more apparent when considering the standard set of plugins distributed with Raster Vision, and how this functionality can be customized with user-created plugins.

8.6.3 Customizing Raster Vision

When approaching a new problem or dataset with Raster Vision, you may get lucky and be able to apply Raster Vision "off-the-shelf". In other cases, Raster Vision can be used after writing scripts to convert data into the appropriate format.

However, sometimes you will need to modify the functionality of Raster Vision to suit your problem. In this case, you could modify the Raster Vision source code (ie. any of the code in the *packages* in the main Raster Vision repo). In some cases, this may be necessary, as the right extension points don't exist. In other cases, the functionality may be very widely-applicable, and you would like to *contributing* it to the main repo. Most of the time, however, the functionality will be problem-specific, or is in an embryonic stage of development, and should be implemented in a plugin that resides outside the main repo.

General information about plugins can be found in *Bootstrap new projects with a template* and *Writing pipelines and plugins*. The following are some brief pointers on how to write plugins for different scenarios. In the future, we would like to enhance this section.

- To add commands to an existing *Pipeline*: write a plugin with subclasses of the *Pipeline* and its corresponding *PipelineConfig* class. The new *Pipeline* should add a method for the new command, and modify the list of `commands`. Any new configuration should be added to the subclass of the *PipelineConfig*. Example: running some data pre- or post-processing code in a pipeline.
- To modify commands of an existing *Pipeline*: same as above except you will override command methods. If a new configuration field is required, you can subclass the *Config* class that field resides within. Example: custom chipping functionality for semantic segmentation. You will need to create subclasses of *SemanticSegmentationChipOptions*, *SemanticSegmentation*, and *SemanticSegmentationConfig*.
- To create a substantially new *Pipeline*: write a new plugin that adds a new *Pipeline*. See the *rastervision.core* plugin, in particular, the contents of the *rastervision.core.rv_pipeline* package. If you want to add a new geospatial deep learning pipeline (eg. for chip regression), you may want to override the *rastervision.core.rv_pipeline* class. In other cases that deviate more from `Raster VisionPipeline`, you may want to write a new *Pipeline* class with arbitrary commands and logic, but that uses the core model building and training functionality in the *rastervision.pytorch_learner* plugin.
- To add the ability to use new file systems or run in new cloud environments: write a plugin that adds a new *FileSystem* or *Runner*. See the *rastervision.aws_s3* and *rastervision.aws_batch* plugins for examples.
- To use an existing *rastervision.core.rv_pipeline* with a new *Backend*: write a plugin that adds a subclass of *Backend* and *BackendConfig*. See the *rastervision.pytorch_backend* plugin for an example.
- To override model building or training routines in an existing *PyTorchLearnerBackend*: write a plugin that adds a subclass of *Learner* (and *LearnerConfig*) that overrides *build_model()* and *train_step()*, and a subclass of *PyTorchLearnerBackend* (and *PyTorchLearnerBackendConfig*) that overrides the backend so it uses the *Learner* subclass.

8.7 Bootstrap new projects with a template

When using Raster Vision on a new project, the best practice is to create a new repo with its own Docker image based on the Raster Vision image. This involves a fair amount of boilerplate code which has a few things that vary between projects. To facilitate bootstrapping new projects, there is a cookiecutter template. Assuming that you cloned the Raster Vision repo and ran `pip install cookiecutter==1.7.0`, you can instantiate the template as follows (after adjusting paths appropriately for your particular setup).

```
[lfishgold@monoshone ~/projects]
$ cookiecutter raster-vision/cookiecutter_template/
```

(continues on next page)

(continued from previous page)

```
caps_project_name [MY_PROJECT]:
project_name [my_project]:
docker_image [my_project]:
parent_docker_image [quay.io/azavea/raster-vision:pytorch-0.20]:
version [0.20]:
description [A Raster Vision plugin]:
url [https://github.com/azavea/raster-vision]:
author [Azavea]:
author_email [info@azavea.com]:

[lfishgold@monoshone ~/projects]
$ tree my_project/
my_project/
├── Dockerfile
├── README.md
├── docker
│   ├── build
│   ├── ecr_publish
│   └── run
└── rastervision_my_project
    ├── rastervision
    │   └── my_project
    │       ├── __init__.py
    │       ├── configs
    │       │   ├── __init__.py
    │       │   └── test.py
    │       ├── test_pipeline.py
    │       └── test_pipeline_config.py
    ├── requirements.txt
    └── setup.py

5 directories, 12 files
```

The output is a repo structure with the skeleton of a Raster Vision plugin that can be pip installed, and everything needed to build, run, and publish a Docker image with the plugin. The resulting `README.md` file contains setup and usage information for running locally and on Batch, which makes use of the *CloudFormation setup* for creating new user/project-specific job defs.

8.8 Setup AWS Batch using CloudFormation

This describes the deployment code that sets up the necessary AWS resources to utilize the AWS Batch runner. Using Batch is advantageous because it starts and stops instances automatically and runs jobs sequentially or in parallel according to the dependencies between them. In addition, this deployment sets up distinct CPU and GPU resources and utilizes spot instances, which is more cost-effective than always using a GPU on-demand instance. Deployment is driven via the AWS console using a CloudFormation template.

This AWS Batch setup is an "advanced" option that assumes some familiarity with Docker, AWS IAM, named profiles, availability zones, EC2, ECR, CloudFormation, and Batch.

8.8.1 AWS Account Setup

In order to setup Batch using this repo, you will need to setup your AWS account so that:

- you have either root access to your AWS account, or an IAM user with admin permissions. It is probably possible with less permissions, but we haven't figured out how to do this yet after some experimentation.
- you have the ability to launch P2 or P3 instances which have GPUs.
- you have requested permission from AWS to use availability zones outside the USA if you would like to use them. (New AWS accounts can't launch EC2 instances in other AZs by default.) If you are in doubt, just use `us-east-1`.

8.8.2 AWS Credentials

Using the AWS CLI, create an AWS profile for the target AWS environment. An example, naming the profile `raster-vision`:

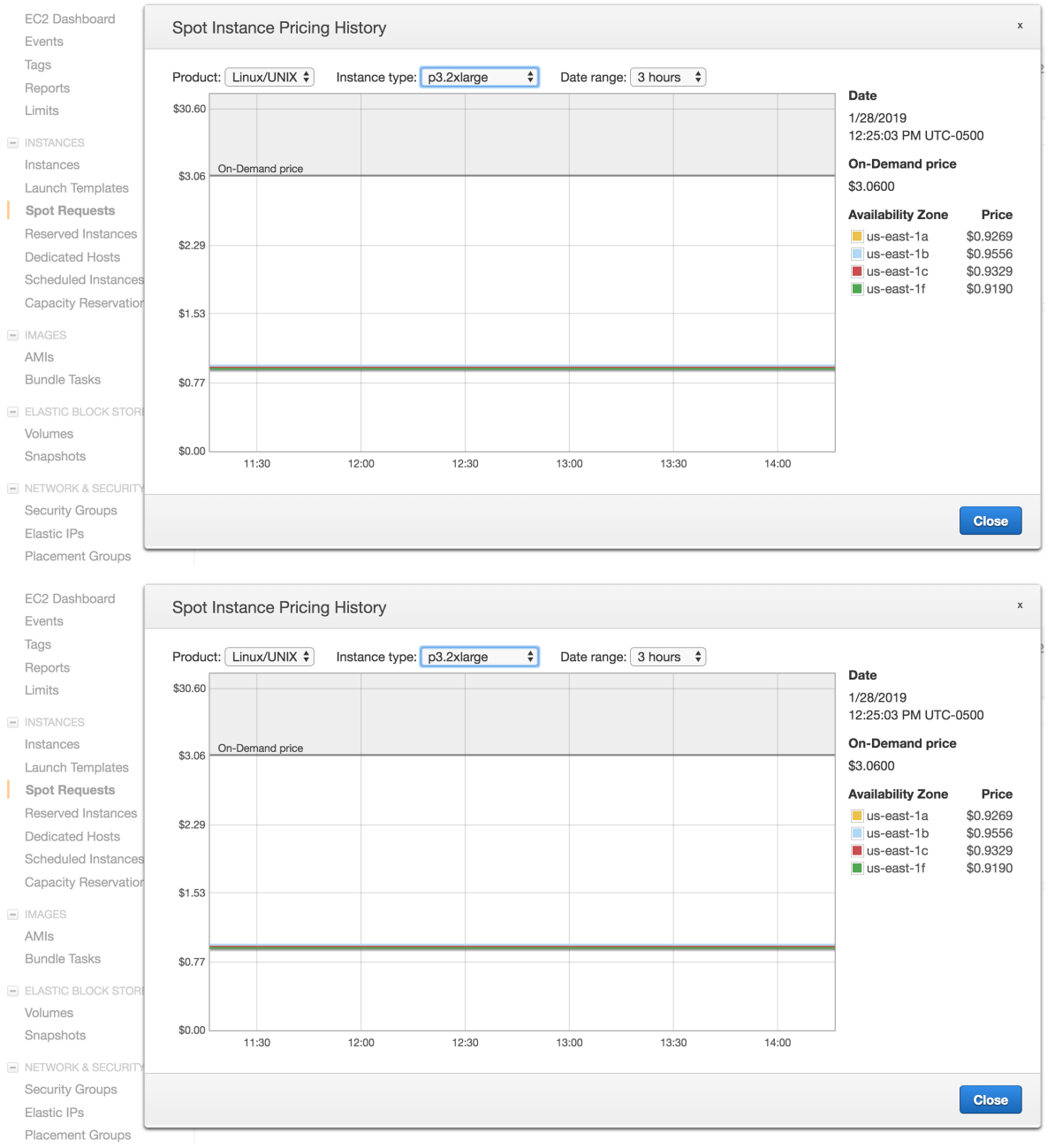
```
$ aws --profile raster-vision configure
AWS Access Key ID [****************F2DQ]:
AWS Secret Access Key [****************TLJ/]:
Default region name [us-east-1]: us-east-1
Default output format [None]:
```

You will be prompted to enter your AWS credentials, along with a default region. The Access Key ID and Secret Access Key can be retrieved from the IAM console. These credentials will be used to authenticate calls to the AWS API when using Packer and the AWS CLI.

8.8.3 Deploying Batch resources

To deploy AWS Batch resources using AWS CloudFormation, start by logging into your AWS console. Then, follow the steps below:

- Navigate to `CloudFormation > Create Stack`
- In the `Choose a template field`, select `Upload a template to Amazon S3` and upload the template in cloudformation/template.yml.
- `Prefix`: If you are setting up multiple RV stacks within an AWS account, you need to set a prefix for namespacing resources. Otherwise, there will be name collisions with any resources that were created as part of another stack.
- Specify the following required parameters:
  - `Stack Name`: The name of your CloudFormation stack
  - `VPC`: The ID of the Virtual Private Cloud in which to deploy your resource. Your account should have at least one by default.
  - `Subnets`: The ID of any subnets that you want to deploy your resources into. Your account should have at least two by default; make sure that the subnets you select are in the VPC that you chose by using the AWS VPC console, or else CloudFormation will throw an error. (Subnets are tied to availability zones, and so affect spot prices.) In addition, you need to choose subnets that are available for the instance type you have chosen. To find which subnets are available, go to Spot Pricing History in the EC2 console and select the instance type. Then look up the availability zones that are present in the VPC console to find the corresponding subnets. Your spot requests will be more likely to be successful and your savings will be greater if you have subnets in more availability zones.



  - `SSH Key Name`: The name of the SSH key pair you want to be able to use to shell into your Batch instances. If you've created an EC2 instance before, you should already have one you can use; otherwise, you can create one in the EC2 console. *Note: If you decide to create a new one, you will need to log out and then back in to the console before creating a Cloudformation stack using this key.*
  - `Instance Types`: Provide the instance types you would like to use. (For GPUs, `p3.2xlarge` is approximately 4 times the speed for 4 times the price.)

- Adjust any preset parameters that you want to change (the defaults should be fine for most users) and click `Next`.

Note: Advanced users: If you plan on modifying Raster Vision and would like to publish a custom image to run on Batch, you will need to specify an ECR repo name and a tag name. Note that the repo names cannot be the same as the Stack name (the first field in the UI) and cannot be the same as any existing ECR repo names. If you

are in a team environment where you are sharing the AWS account, the repo names should contain an identifier such as your username.

- Accept all default options on the `Options` page and click `Next`.
- Accept "I acknowledge that AWS CloudFormation might create IAM resources with custom names" on the `Review` page and click `Create`.
- Watch your resources get deployed!

8.8.4 Publish local Raster Vision images to ECR

If you setup ECR repositories during the CloudFormation setup (the "advanced user" option), then you will need to follow this step, which publishes local Raster Vision images to those ECR repositories. Every time you make a change to your local Raster Vision images and want to use those on Batch, you will need to run these steps:

- Run `./docker/build` in the Raster Vision repo to build a local copy of the Docker image.
- Run `./docker/ecr_publish` in the Raster Vision repo to publish the Docker images to ECR. Note that this requires setting the `RV_ECR_IMAGE` environment variable to be set to `<ecr_repo_name>:<tag_name>`.

8.8.5 Update Raster Vision configuration

Finally, make sure to update your *Running on AWS Batch* with the Batch resources that were created.

8.8.6 Deploy new job definitions

When a user starts working on a new RV-based project (or a new user starts working on an existing RV-based project), they will often want to publish a custom Docker image to ECR and use it when running on Batch. To facilitate this, there is a separate cloudformation/job_def_template.yml. The idea is that for each user/project pair which is identified by a `Namespace` string, a CPU and GPU job definition is created which point to a specified ECR repo using that `Namespace` as the tag. After creating these new resources, the image should be published to `<repo>:<namespace>` on ECR, and the new job definitions should be placed in a project-specific RV profile file.

8.9 Miscellaneous Topics

8.9.1 File Systems

The *`FileSystem`* architecture supports multiple file systems through an interface that is chosen by URI. There is built-in support for: local and HTTP file systems in the *`rastervision.pipeline`* package, AWS S3 in the *`rastervision.aws_s3`* plugin, and any file that can be opened using GDAL VSI in the `rastervision.gdal_vsi` plugin. Some file systems support read only (HTTP), while others are read/write. If you need to support other file storage systems, you can add new *`FileSystem`* subclasses via a plugin.

8.9.2 Viewing Tensorboard

The PyTorch backends included in the *rastervision.pytorch_backend* plugin will start an instance of TensorBoard while training if *log_tensorboard=True* and *run_tensorboard=True* in the *PyTorchLearnerBackendConfig*.

To view TensorBoard, go to `https://<domain>:6006/`. If you're running locally, then `<domain>` should be `localhost`, and if you are running remotely (for example AWS), `<domain>` is the public DNS of the machine running the training command. If running locally, make sure to forward port 6006 using the `--tensorboard` option to `docker/run` if you are using it. At the moment, basic metrics are logged each epoch, but more interesting visualization could be added in the future.

8.9.3 Transfer learning using models trained by RV

To use a model trained by Raster Vision for transfer learning or fine tuning, you can use output of the TRAIN command of the experiment as a pretrained model of further experiments. The `last-model.pth` model file in the `train` directory can be used as a pretrained model in a new pipeline. To do so, set the *init_weights* field to the model file in the *ModelConfig* in the new pipeline.

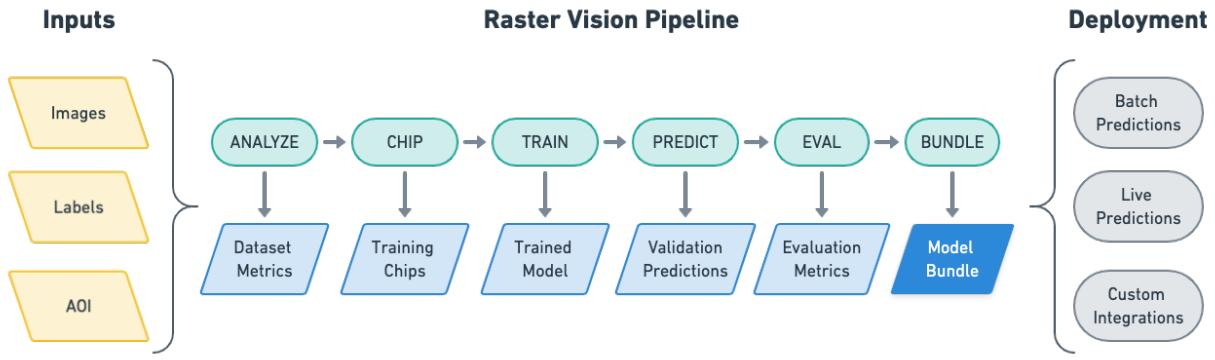
8.9.4 Making Predictions with Model Bundles

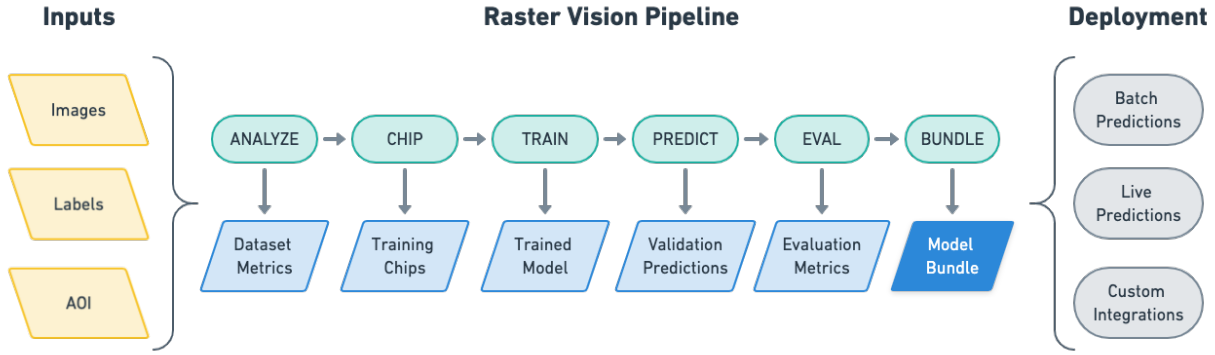
To make predictions on new imagery, the *bundle* command generates a "model bundle" which can be used with the *predict* command. This loads the model and saves the predictions for a single scene. If you need to call this for a large number of scenes, consider using the *Predictor* class programmatically, as this will allow you to load the model once and use it many times. This can matter a lot if you want the time-to-prediction to be as fast as possible - the model load time can be orders of magnitudes slower than the prediction time of a loaded model.

The model bundle is a zip file containing the model weights and the configuration necessary for Raster Vision to use the model. This configuration includes the configuration of the model architecture, how the training data was processed by *RasterTransformers* (if any), the subset of bands used by the *RasterSource*, and potentially other things. The model bundle holds all of this necessary information, so that a prediction call only needs to know what imagery it is predicting against.

This works generically over all models produced by Raster Vision, without additional client considerations, and therefore abstracts away the specifics of every model when considering how to deploy prediction software. Note that this means that by default, predictions will be made according to the configuration of the pipeline that produced the model bundle. Some of this configuration might be inappropriate for the new imagery (such as the *channel_order*), and can be overridden by options to the *predict* command.

Raster Vision allows engineers to quickly and repeatably configure **pipelines** that go through core components of a machine learning workflow: analyzing training data, creating training chips, training models, creating predictions, evaluating models, and bundling the model files and configuration for easy deployment.





The input to a Raster Vision pipeline is a set of images and training data, optionally with Areas of Interest (AOIs) that describe where the images are labeled. The output of a Raster Vision pipeline is a model bundle that allows you to easily utilize models in various deployment scenarios.

The pipelines include running the following commands:

- **ANALYZE**: Gather dataset-level statistics and metrics for use in downstream processes.
- **CHIP**: Create training chips from a variety of image and label sources.
- **TRAIN**: Train a model using a "backend" such as PyTorch.
- **PREDICT**: Make predictions using trained models on validation and test data.
- **EVAL**: Derive evaluation metrics such as F1 score, precision and recall against the model's predictions on validation datasets.
- **BUNDLE**: Bundle the trained model and associated configuration into a *model bundle*, which can be deployed in batch processes, live servers, and other workflows.

Pipelines are configured using a compositional, programmatic approach that makes configuration easy to read, reuse, and maintain. Below, we show the `tiny_spacenet` example.

Listing 9: tiny_spacenet.py

```
from os.path import join
from rastervision.core.rv_pipeline import *
from rastervision.core.backend import *
from rastervision.core.data import *
from rastervision.pytorch_backend import *
```

(continues on next page)

(continued from previous page)

```
from rastervision.pytorch_learner import *


def get_config(runner) -> SemanticSegmentationConfig:
    output_root_uri = '/opt/data/output/'
    class_config = ClassConfig(
        names=['building', 'background'], colors=['red', 'black'])

    base_uri = ('https://s3.amazonaws.com/azavea-research-public-data/'
                'raster-vision/examples/spacenet')
    train_image_uri = join(base_uri, 'RGB-PanSharpen_AOI_2_Vegas_img205.tif')
    train_label_uri = join(base_uri, 'buildings_AOI_2_Vegas_img205.geojson')
    val_image_uri = join(base_uri, 'RGB-PanSharpen_AOI_2_Vegas_img25.tif')
    val_label_uri = join(base_uri, 'buildings_AOI_2_Vegas_img25.geojson')

    train_scene = make_scene('scene_205', train_image_uri, train_label_uri,
                             class_config)
    val_scene = make_scene('scene_25', val_image_uri, val_label_uri,
                           class_config)
    scene_dataset = DatasetConfig(
        class_config=class_config,
        train_scenes=[train_scene],
        validation_scenes=[val_scene])

    # Use the PyTorch backend for the SemanticSegmentation pipeline.
    chip_sz = 300

    backend = PyTorchSemanticSegmentationConfig(
        data=SemanticSegmentationGeoDataConfig(
            scene_dataset=scene_dataset,
            window_opts=GeoDataWindowConfig(
                # randomly sample training chips from scene
                method=GeoDataWindowMethod.random,
                # ... of size chip_sz x chip_sz
                size=chip_sz,
                # ... and at most 10 chips per scene
                max_windows=10)),
        model=SemanticSegmentationModelConfig(backbone=Backbone.resnet50),
        solver=SolverConfig(lr=1e-4, num_epochs=1, batch_sz=2))

    return SemanticSegmentationConfig(
        root_uri=output_root_uri,
        dataset=scene_dataset,
        backend=backend,
        train_chip_sz=chip_sz,
        predict_chip_sz=chip_sz)


def make_scene(scene_id: str, image_uri: str, label_uri: str,
               class_config: ClassConfig) -> SceneConfig:
    """Define a Scene with image and labels from the given URIs."""
```

(continues on next page)

(continued from previous page)

```
    raster_source = RasterioSourceConfig(
        uris=image_uri,
        # use only the first 3 bands
        channel_order=[0, 1, 2],
    )

    # configure GeoJSON reading
    vector_source = GeoJSONVectorSourceConfig(
        uri=label_uri,
        # This assumes the CRS is WGS-84 and ignores whatever the CRS specified
        # in the file is.
        ignore_crs_field=True,
        # The geoms in the label GeoJSON do not have a "class_id" property, so
        # classes must be inferred. Since all geoms are for the building class,
        # this is easy to do: we just assing the building class ID to all of
        # them.
        transformers=[
            ClassInferenceTransformerConfig(
                default_class_id=class_config.get_class_id('building'))
        ])
    # configure transformation of vector data into semantic segmentation labels
    label_source = SemanticSegmentationLabelSourceConfig(
        # semantic segmentation labels must be rasters, so rasterize the geoms
        raster_source=RasterizedSourceConfig(
            vector_source=vector_source,
            rasterizer_config=RasterizerConfig(
                # What about pixels outside of any geoms? Mark them as
                # background.
                background_class_id=class_config.get_class_id('background'))))

    return SceneConfig(
        id=scene_id,
        raster_source=raster_source,
        label_source=label_source,
    )
```

Raster Vision uses a `unittest`-like method for executing pipelines. For instance, if the above was defined in *tiny_spacenet.py*, with the proper setup you could run the experiment on AWS Batch by running:

```
> rastervision run batch tiny_spacenet.py
```

See the *Quickstart* for a more complete description of using this example.

CHAPTER

NINE

API REFERENCE

Raster Vision Plugins

rastervision.pipeline

rastervision.core

rastervision.pytorch_learner

rastervision.pytorch_backend

rastervision.aws_s3

rastervision.aws_batch

9.1 pipeline

Modules

cli	
config	
file_system	
pipeline	
pipeline_config	
registry	
runner	
rv_config	
utils	
verbosity	
version	Library verison

9.1.1 cli

Functions

convert_bool_args(args)	Convert boolean CLI arguments from string to bool.
get_configs(cfg_module_path, runner, args)	Get PipelineConfigs from a module.
print_error(msg)	Print error message to console in red.

convert_bool_args

convert_bool_args(*args: dict*) → dict

Convert boolean CLI arguments from string to bool.

Parameters
args (*dict*) – a mapping from CLI argument names to values

Returns
copy of args with boolean string values convert to bool

Return type
dict

get_configs

get_configs(*cfg_module_path: str*, *runner: str*, *args: Dict[str, any]*) → List[*PipelineConfig*]

Get PipelineConfigs from a module.

Calls a get_config(s) function with some arguments from the CLI to get a list of PipelineConfigs.

Parameters

- **cfg_module_path** (*str*) – the module with *get_configs* function that returns PipelineConfigs. This can either be a Python module path or a local path to a .py file.
- **runner** (*str*) – name of the runner
- **args** (*Dict[str, any]*) – CLI args to pass to the get_config(s) function that comes from the –args option

Return type

List[PipelineConfig]

print_error

print_error(*msg*)

Print error message to console in red.

9.1.2 config

Configs

Config	Base class that can be extended to provide custom configurations.

Config

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model Config

Base class that can be extended to provide custom configurations.

This adds some extra methods to Pydantic BaseModel. See https://pydantic-docs.helpmanual.io/

The general idea is that configuration schemas can be defined by subclassing this and adding class attributes with types and default values for each field. Configs can be defined hierarchically, ie. a Config can have fields which are of type Config. Validation, serialization, deserialization, and IDE support is provided automatically based on this schema.


```
{
   "title": "Config",
   "description": "Base class that can be extended to provide custom configurations.
↪\n\nThis adds some extra methods to Pydantic BaseModel.\nSee https://pydantic-
↪docs.helpmanual.io/\n\nThe general idea is that configuration schemas can be␣
↪defined by\nsubclassing this and adding class attributes with types and\ndefault␣
```


(continues on next page)

(continued from previous page)

```
↪values for each field. Configs can be defined hierarchically,\nie. a Config can␣
↪have fields which are of type Config.\nValidation, serialization, deserialization,
↪ and IDE support is\nprovided automatically based on this schema.",
   "type": "object",
   "properties": {},
   "additionalProperties": false
}
```

Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

build()

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(**args*, ***kwargs*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

Functions

build_config(x)	Build a Config from various types of input.
get_plugin(config_cls)	Infer the module path of the plugin where a Config class is defined.
register_config(type_hint[, plugin, upgrader])	Class decorator used to register Config classes with registry.
save_pipeline_config(cfg, output_uri)	Save a PipelineConfig to JSON file.
upgrade_config(config_dict)	Upgrade serialized Config(s) to the latest version.
upgrade_plugin_versions(plugin_versions)	Update the names of the plugins using the plugin aliases in the registry.

build_config

build_config(*x: Union[dict, List[Union[dict,* Config*]],* Config*]*) → Union[*Config*, List[*Config*]]

Build a Config from various types of input.

This is useful for deserializing from JSON. It implements polymorphic deserialization by using the *type_hint* in each dict to get the corresponding Config class from the registry.

Parameters
x (*Union[dict, List[Union[dict, Config]], Config]*) – some representation of Config(s)

Returns
the corresponding Config(s)

Return type
Config

get_plugin

get_plugin(*config_cls: Type*) → str

Infer the module path of the plugin where a Config class is defined.

This only works correctly if the plugin is in a module under rastervision.

Parameters
config_cls (*Type*) –

Return type
str

register_config

register_config(*type_hint: str*, *plugin: Optional[str] = None*, *upgrader: Optional[Callable] = None*) → Callable

Class decorator used to register Config classes with registry.

All Configs must be registered! Registering a Config does the following:

1. Associates Config classes with type_hint, plugin, and upgrader, which is necessary for polymorphic deserialization. See build_config() for more details.
2. Adds a constant *type_hint* field to the Config which is set to type_hint.

Parameters

- **type_hint** (*str*) – a type hint used to deserialize Configs. Must be unique across all registered Configs.
- **plugin** (*Optional[str], optional*) – the module path of the plugin where the Config is defined. If None, will be inferred. Defauilts to None.
- **upgrader** (*Optional[Callable], optional*) – a function of the form upgrade(config_dict, version) which returns the corresponding config dict of version = version + 1. This can be useful for maintaining backward compatibility by allowing old configs using an outdated schema to be upgraded to the current schema. Defaults to None.

Returns

A function that returns a new class that is identical to the input Config with an additional `type_hint` field.

Return type
Callable

save_pipeline_config

save_pipeline_config(*cfg:* PipelineConfig, *output_uri: str*) → None

Save a PipelineConfig to JSON file.

Inject rv_config and plugin_versions before saving.

Parameters

- **cfg** (PipelineConfig) –
- **output_uri** (*str*) –

Return type
None

upgrade_config

upgrade_config(*config_dict: Union[dict, List[dict]]*) → Union[dict, List[dict]]

Upgrade serialized Config(s) to the latest version.

Used to implement backward compatibility of Configs using upgraders stored in the registry.

Parameters
config_dict (*Union[dict, List[dict]]*) – serialized PipelineConfig(s) which are potentially of a non-current version

Returns

the corresponding serialized PipelineConfig(s) that have been upgraded to the current version

Return type
Union[dict, *List*[dict]]

upgrade_plugin_versions

upgrade_plugin_versions(*plugin_versions: Dict[str, int]*) → Dict[str, int]

Update the names of the plugins using the plugin aliases in the registry.

This allows changing the names of plugins over time and maintaining backward compatibility of serialized PipelineConfigs.

Parameters

- **plugin_version** – maps from plugin name to version
- **plugin_versions** (*Dict[str, int]*) –

Return type
Dict[str, int]

Exceptions

ConfigError	Exception raised for invalid configuration.

rastervision.pipeline.config.ConfigError

exception ConfigError

Exception raised for invalid configuration.

__init__(**args*, ***kwargs*)

__new__(***kwargs*)

9.1.3 file_system

Modules

file_system
http_file_system
local_file_system
utils

file_system

Classes

FileSystem	Abstraction for a local or remote file system.

FileSystem

class FileSystem

Bases: ABC

Abstraction for a local or remote file system.

This can be subclassed to handle different cloud storage providers, etc.

__init__()

Methods

__init__()

copy_from(src_uri, dst_path)	Copy a source file to a local destination.
copy_to(src_path, dst_uri)	Copy a local source file to a destination.
file_exists(uri[, include_dir])	Check if a file exists.
get_file_system(uri[, mode])	Return FileSystem that should be used for the given URI/mode pair.
last_modified(uri)	Get the last modified date of a file.
list_paths(uri[, ext])	List paths rooted at URI.
local_path(uri, download_dir)	Return the path where a local copy should be stored.
matches_uri(uri, mode)	Returns True if this FS can be used for the given URI/mode pair.
read_bytes(uri)	Read contents of URI to bytes.
read_str(uri)	Read contents of URI to a string.
sync_from_dir(src_dir_uri, dst_dir[, delete])	Syncs a source directory to a local destination directory.
sync_to_dir(src_dir, dst_dir_uri[, delete])	Syncs a local source directory to a destination directory.
write_bytes(uri, data)	Write bytes in data to URI.
write_str(uri, data)	Write string in data to URI.

abstract static copy_from(*src_uri: str*, *dst_path: str*)

Copy a source file to a local destination.

If the FileSystem is remote, this involves downloading.

Parameters

- **src_uri** (*str*) – uri of source that can be copied from by this FileSystem
- **dst_path** (*str*) – local path to destination file

abstract static copy_to(*src_path: str*, *dst_uri: str*)

Copy a local source file to a destination.

If the FileSystem is remote, this involves uploading.

Parameters

- **src_path** (*str*) – local path to source file
- **dst_uri** (*str*) – uri of destination that can be copied to by this FileSystem

abstract static file_exists(*uri: str*, *include_dir: bool = True*) → bool

Check if a file exists.

Parameters

- **uri** (*str*) – The URI to check
- **include_dir** (*bool*) – Include directories in check, if this file_system supports directory reads. Otherwise only return true if a single file exists at the URI.

Return type
bool

static get_file_system(*uri: str*, *mode: str = 'r'*) → *FileSystem*

Return FileSystem that should be used for the given URI/mode pair.

Parameters

- **uri** (*str*) – URI of file
- **mode** (*str*) – mode to open file in, 'r' or 'w'

Return type
FileSystem

abstract static last_modified(*uri: str*) → Optional[datetime]

Get the last modified date of a file.

Parameters
uri (*str*) – the URI of the file

Returns
the last modified date in UTC of a file or None if this FileSystem does not support this operation.

Return type
Optional[*datetime*]

abstract static list_paths(*uri: str*, *ext: Optional[str] = None*) → List[str]

List paths rooted at URI.

Optionally only includes paths with a certain file extension.

Parameters

- **uri** (*str*) – the URI of a directory
- **ext** (*Optional[str]*) – the optional file extension to filter by

Return type
List[str]

abstract static local_path(*uri: str*, *download_dir: str*) → str

Return the path where a local copy should be stored.

Parameters

- **uri** (*str*) – the URI of the file to be copied
- **download_dir** (*str*) – path of the local directory in which files should be copied

Return type
str

abstract static matches_uri(*uri: str*, *mode: str*) → bool

Returns True if this FS can be used for the given URI/mode pair.

Parameters

- **uri** (*str*) – URI of file
- **mode** (*str*) – mode to open file in, 'r' or 'w'

Return type
bool

abstract static read_bytes(*uri: str*) → bytes

Read contents of URI to bytes.

Parameters
uri (*str*) –

Return type
bytes

abstract static read_str(*uri: str*) → str

Read contents of URI to a string.

Parameters
uri (*str*) –

Return type
str

abstract static sync_from_dir(*src_dir_uri: str*, *dst_dir: str*, *delete: bool = False*)

Syncs a source directory to a local destination directory.

If the FileSystem is remote, this involves downloading.

Parameters

- **src_dir_uri** (*str*) – source directory that can be synced from by this FileSystem
- **dst_dir** (*str*) – A local destination directory
- **delete** (*bool*) – True if the destination should be deleted first.

abstract static sync_to_dir(*src_dir: str*, *dst_dir_uri: str*, *delete: bool = False*)

Syncs a local source directory to a destination directory.

If the FileSystem is remote, this involves uploading.

Parameters

- **src_dir** (*str*) – local source directory to sync from
- **dst_dir_uri** (*str*) – A destination directory that can be synced to by this FileSystem

- **delete** (*bool*) – True if the destination should be deleted first.

abstract static write_bytes(*uri: str*, *data: bytes*)

Write bytes in data to URI.

Parameters

- **uri** (*str*) –
- **data** (*bytes*) –

abstract static write_str(*uri: str*, *data: str*)

Write string in data to URI.

Parameters

- **uri** (*str*) –
- **data** (*str*) –

Exceptions

NotReadableError	Exception raised when files are not readable.
NotWritableError	Exception raised when files are not writable.

rastervision.pipeline.file_system.file_system.NotReadableError

exception NotReadableError

Exception raised when files are not readable.

__init__(**args*, ***kwargs*)

__new__(***kwargs*)

rastervision.pipeline.file_system.file_system.NotWritableError

exception NotWritableError

Exception raised when files are not writable.

__init__(**args*, ***kwargs*)

__new__(***kwargs*)

http_file_system

Classes

HttpFileSystem	A FileSystem for downloading files over HTTP.

HttpFileSystem

class HttpFileSystem

Bases: *FileSystem*

A FileSystem for downloading files over HTTP.

__init__()

Methods

__init__()	

copy_from(src_uri, dst_path)	Copy a source file to a local destination.
copy_to(src_path, dst_uri)	Copy a local source file to a destination.
file_exists(uri[, include_dir])	Check if a file exists.
get_file_system(uri[, mode])	Return FileSystem that should be used for the given URI/mode pair.
last_modified(uri)	Get the last modified date of a file.
list_paths(uri[, suffix])	List paths rooted at URI.
local_path(uri, download_dir)	Return the path where a local copy should be stored.
matches_uri(uri, mode)	Returns True if this FS can be used for the given URI/mode pair.
read_bytes(uri)	Read contents of URI to bytes.
read_str(uri)	Read contents of URI to a string.
sync_from_dir(src_dir_uri, dst_dir[, delete])	Syncs a source directory to a local destination directory.
sync_to_dir(src_dir, dst_dir_uri[, delete])	Syncs a local source directory to a destination directory.
write_bytes(uri, data)	Write bytes in data to URI.
write_str(uri, data)	Write string in data to URI.

static copy_from(*src_uri: str*, *dst_path: str*) → None

Copy a source file to a local destination.

If the FileSystem is remote, this involves downloading.

Parameters

- **src_uri** (*str*) – uri of source that can be copied from by this FileSystem
- **dst_path** (*str*) – local path to destination file

Return type
None

static copy_to(*src_path: str*, *dst_uri: str*) → None

Copy a local source file to a destination.

If the FileSystem is remote, this involves uploading.

Parameters

- **src_path** (*str*) – local path to source file
- **dst_uri** (*str*) – uri of destination that can be copied to by this FileSystem

Return type
None

static file_exists(*uri: str*, *include_dir: bool = True*) → bool

Check if a file exists.

Parameters

- **uri** (*str*) – The URI to check
- **include_dir** (*bool*) – Include directories in check, if this file_system supports directory reads. Otherwise only return true if a single file exists at the URI.

Return type
bool

static get_file_system(*uri: str*, *mode: str = 'r'*) → *FileSystem*

Return FileSystem that should be used for the given URI/mode pair.

Parameters

- **uri** (*str*) – URI of file
- **mode** (*str*) – mode to open file in, 'r' or 'w'

Return type
FileSystem

static last_modified(*uri: str*) → datetime

Get the last modified date of a file.

Parameters
uri (*str*) – the URI of the file

Returns
the last modified date in UTC of a file or None if this FileSystem does not support this operation.

Return type
datetime

static list_paths(*uri*, *suffix=None*)

List paths rooted at URI.

Optionally only includes paths with a certain file extension.

Parameters

- **uri** – the URI of a directory
- **ext** – the optional file extension to filter by

static local_path(*uri: str*, *download_dir: str*) → None

Return the path where a local copy should be stored.

Parameters

- **uri** (*str*) – the URI of the file to be copied
- **download_dir** (*str*) – path of the local directory in which files should be copied

Return type
None

static matches_uri(*uri: str*, *mode: str*) → bool

Returns True if this FS can be used for the given URI/mode pair.

Parameters

- **uri** (*str*) – URI of file
- **mode** (*str*) – mode to open file in, 'r' or 'w'

Return type
bool

static read_bytes(*uri: str*) → bytes

Read contents of URI to bytes.

Parameters
uri (*str*) –

Return type
bytes

static read_str(*uri: str*) → str

Read contents of URI to a string.

Parameters
uri (*str*) –

Return type
str

static sync_from_dir(*src_dir_uri: str*, *dst_dir: str*, *delete: bool = False*) → None

Syncs a source directory to a local destination directory.

If the FileSystem is remote, this involves downloading.

Parameters

- **src_dir_uri** (*str*) – source directory that can be synced from by this FileSystem
- **dst_dir** (*str*) – A local destination directory
- **delete** (*bool*) – True if the destination should be deleted first.

Return type
None

static sync_to_dir(*src_dir: str*, *dst_dir_uri: str*, *delete: bool = False*) → None

Syncs a local source directory to a destination directory.

If the FileSystem is remote, this involves uploading.

Parameters

- **src_dir** (*str*) – local source directory to sync from
- **dst_dir_uri** (*str*) – A destination directory that can be synced to by this FileSystem
- **delete** (*bool*) – True if the destination should be deleted first.

Return type
None

static write_bytes(*uri: str*, *data: bytes*) → None

Write bytes in data to URI.

Parameters

- **uri** (*str*) –
- **data** (*bytes*) –

Return type
None

static write_str(*uri: str*, *data: str*) → None

Write string in data to URI.

Parameters

- **uri** (*str*) –
- **data** (*str*) –

Return type
None

Functions

get_file_obj(uri[, with_progress])	Returns a context manager for a file-like object that supports buffered reads.

get_file_obj

get_file_obj(*uri: str*, *with_progress: bool = True*) → ContextManager

Returns a context manager for a file-like object that supports buffered reads. If with_progress is True, wraps the read() method of the object in a function that updates a tqdm progress bar.

Usage:

```
with get_file_obj(uri) as f:
    ...
```

Adapted from https://stackoverflow.com/a/63831344/5908685.

Parameters

- **uri** (*str*) –
- **with_progress** (*bool*) –

Return type
ContextManager

local_file_system

Classes

LocalFileSystem	A FileSystem for interacting with the local file system.

LocalFileSystem

class LocalFileSystem

Bases: *FileSystem*

A FileSystem for interacting with the local file system.

__init__()

Methods

__init__()	
copy_from(src_uri, dst_path)	Copy a source file to a local destination.
copy_to(src_path, dst_uri)	Copy a local source file to a destination.
file_exists(uri[, include_dir])	Check if a file exists.
get_file_system(uri[, mode])	Return FileSystem that should be used for the given URI/mode pair.
last_modified(uri)	Get the last modified date of a file.
list_paths(uri[, ext])	List paths rooted at URI.
local_path(uri, download_dir)	Return the path where a local copy should be stored.
matches_uri(uri, mode)	Returns True if this FS can be used for the given URI/mode pair.
read_bytes(file_uri)	Read contents of URI to bytes.
read_str(file_uri)	Read contents of URI to a string.
sync_from_dir(src_dir_uri, dst_dir[, delete])	Syncs a source directory to a local destination directory.
sync_to_dir(src_dir, dst_dir_uri[, delete])	Syncs a local source directory to a destination directory.
write_bytes(file_uri, data)	Write bytes in data to URI.
write_str(file_uri, data)	Write string in data to URI.

static copy_from(*src_uri: str*, *dst_path: str*) → None

Copy a source file to a local destination.

If the FileSystem is remote, this involves downloading.

Parameters

- **src_uri** (*str*) – uri of source that can be copied from by this FileSystem
- **dst_path** (*str*) – local path to destination file

Return type
None

static copy_to(*src_path: str*, *dst_uri: str*) → None

Copy a local source file to a destination.

If the FileSystem is remote, this involves uploading.

Parameters

- **src_path** (*str*) – local path to source file
- **dst_uri** (*str*) – uri of destination that can be copied to by this FileSystem

Return type
None

static file_exists(*uri: str*, *include_dir: bool = True*) → bool

Check if a file exists.

Parameters

- **uri** (*str*) – The URI to check
- **include_dir** (*bool*) – Include directories in check, if this file_system supports directory reads. Otherwise only return true if a single file exists at the URI.

Return type
bool

static get_file_system(*uri: str*, *mode: str = 'r'*) → *FileSystem*

Return FileSystem that should be used for the given URI/mode pair.

Parameters

- **uri** (*str*) – URI of file
- **mode** (*str*) – mode to open file in, 'r' or 'w'

Return type
FileSystem

static last_modified(*uri: str*) → datetime

Get the last modified date of a file.

Parameters
uri (*str*) – the URI of the file

Returns
the last modified date in UTC of a file or None if this FileSystem does not support this operation.

Return type
datetime

static list_paths(*uri*, *ext=None*)

List paths rooted at URI.

Optionally only includes paths with a certain file extension.

Parameters

- **uri** – the URI of a directory
- **ext** – the optional file extension to filter by

static local_path(*uri: str*, *download_dir: str*) → None

Return the path where a local copy should be stored.

Parameters

- **uri** (*str*) – the URI of the file to be copied
- **download_dir** (*str*) – path of the local directory in which files should be copied

Return type
None

static matches_uri(*uri: str*, *mode: str*) → bool

Returns True if this FS can be used for the given URI/mode pair.

Parameters

- **uri** (*str*) – URI of file
- **mode** (*str*) – mode to open file in, 'r' or 'w'

Return type
bool

static read_bytes(*file_uri: str*) → bytes

Read contents of URI to bytes.

Parameters
file_uri (*str*) –

Return type
bytes

static read_str(*file_uri: str*) → str

Read contents of URI to a string.

Parameters
file_uri (*str*) –

Return type
str

static sync_from_dir(*src_dir_uri: str*, *dst_dir: str*, *delete: bool = False*) → None

Syncs a source directory to a local destination directory.

If the FileSystem is remote, this involves downloading.

Parameters

- **src_dir_uri** (*str*) – source directory that can be synced from by this FileSystem
- **dst_dir** (*str*) – A local destination directory
- **delete** (*bool*) – True if the destination should be deleted first.

Return type
None

static sync_to_dir(*src_dir: str*, *dst_dir_uri: str*, *delete: bool = False*) → None

Syncs a local source directory to a destination directory.

If the FileSystem is remote, this involves uploading.

Parameters

- **src_dir** (*str*) – local source directory to sync from

- **dst_dir_uri** (*str*) – A destination directory that can be synced to by this FileSystem
- **delete** (*bool*) – True if the destination should be deleted first.

Return type
None

static write_bytes(*file_uri: str*, *data: bytes*) → None

Write bytes in data to URI.

Parameters

- **file_uri** (*str*) –
- **data** (*bytes*) –

Return type
None

static write_str(*file_uri: str*, *data: str*) → None

Write string in data to URI.

Parameters

- **file_uri** (*str*) –
- **data** (*str*) –

Return type
None

Functions

make_dir(path[, check_empty, force_empty, ...])	Make a local directory.
progressbar(file_obj, method, size, desc)	

make_dir

make_dir(*path*, *check_empty=False*, *force_empty=False*, *use_dirname=False*)

Make a local directory.

Parameters

- **path** – path to directory
- **check_empty** – if True, check that directory is empty
- **force_empty** – if True, delete files if necessary to make directory empty
- **use_dirname** – if True, use the the parent directory as path

Raises
ValueError if check_empty is True and directory is not empty –

progressbar

progressbar(*file_obj*, *method: str*, *size: int*, *desc: str*)

Parameters

- **method** (*str*) –
- **size** (*int*) –
- **desc** (*str*) –

utils

Functions

download_if_needed(uri[, download_dir, fs, ...])	Download a file into a directory if it's remote.
download_or_copy(uri, target_dir[, ...])	Downloads or copies a file to a directory.
extract(uri[, target_dir, download_dir])	Extract a compressed file.
file_exists(uri[, fs, include_dir])	Check if file exists.
file_to_json(uri)	Return JSON dict based on file at uri.
file_to_str(uri[, fs])	Load contents of text file into a string.
get_local_path(uri, download_dir[, fs])	Return the path where a local copy of URI should be stored.
get_tmp_dir()	Return temporary directory given by the RVConfig.
is_archive(uri)	Check if the URI's extension represents an archived file.
is_local(uri)	
json_to_file(content_dict, uri)	Upload JSON file to uri based on content_dict.
list_paths(uri[, ext, fs])	List paths rooted at URI.
start_sync(src_dir, dst_dir_uri[, ...])	Repeatedly sync a local source directory to a destination on a schedule.
str_to_file(content_str, uri[, fs])	Writes string to text file.
sync_from_dir(src_dir_uri, dst_dir[, delete, fs])	Synchronize a source directory to local destination directory.
sync_to_dir(src_dir, dst_dir_uri[, delete, fs])	Synchronize a local source directory to destination directory.
unzip(zip_path, target_dir)	Unzip contents of zip file at zip_path into target_dir.
upload_or_copy(src_path, dst_uri[, fs])	Upload or copy a file.
zipdir(dir, zip_path)	Save a zip file with contents of directory.

download_if_needed

download_if_needed(*uri: str*, *download_dir: Optional[str] = None*, *fs: Optional[FileSystem] = None*, *use_cache: bool = True*) → str

Download a file into a directory if it's remote.

If uri is local, there is no need to download the file.

Parameters

- **uri** (*str*) – URI of file to download.

- **download_dir** (*Optional[str], optional*) – Local directory to download file into. If None, the file will be downloaded to cache dir as defined by RVConfig. Defaults to None.
- **fs** (*Optional[FileSystem], optional*) – If provided, use fs instead of the automatically chosen FileSystem for uri. Defaults to None.
- **use_cache** (*bool, optional*) – If False and the file is remote, download it regardless of whether it exists in cache. Defaults to True.

Returns
Path to local file.

Return type
str

Raises
NotReadableError if URI cannot be read from –

download_or_copy

download_or_copy(*uri: str*, *target_dir: str*, *delete_tmp: bool = False*, *fs: Optional[FileSystem] = None*) → str

Downloads or copies a file to a directory.

Downloads or copies URI into target_dir.

Parameters

- **uri** (*str*) – URI of file.
- **target_dir** (*str*) – Local directory to download or copy file to.
- **delete_tmp** (*bool*) – Delete temporary download dir after copying file.
- **fs** (*Optional[FileSystem]*) – If supplied, use fs instead of automatically chosen FileSystem for uri.

Returns
the local path of file

Return type
str

extract

extract(*uri: str*, *target_dir: Optional[str] = None*, *download_dir: Optional[str] = None*) → str

Extract a compressed file.

Parameters

- **uri** (*str*) –
- **target_dir** (*Optional[str]*) –
- **download_dir** (*Optional[str]*) –

Return type
str

file_exists

file_exists(*uri*, *fs=None*, *include_dir=True*) → bool

Check if file exists.

Parameters

- **uri** – URI of file
- **fs** – if supplied, use fs instead of automatically chosen FileSystem for uri

Return type
bool

file_to_json

file_to_json(*uri: str*) → dict

Return JSON dict based on file at uri.

Parameters
uri (*str*) –

Return type
dict

file_to_str

file_to_str(*uri: str*, *fs: Optional[FileSystem] = None*) → str

Load contents of text file into a string.

Parameters

- **uri** (*str*) – URI of file
- **fs** (*Optional[FileSystem]*) – if supplied, use fs instead of automatically chosen FileSystem

Returns
contents of text file

Raises
NotReadableError if URI cannot be read –

Return type
str

get_local_path

get_local_path(*uri: str*, *download_dir: str*, *fs: Optional[FileSystem] = None*) → str

Return the path where a local copy of URI should be stored.

If URI is local, return it. If it's remote, we generate a path for it within download_dir.

Parameters

- **uri** (*str*) – the URI of the file to be copied
- **download_dir** (*str*) – path of the local directory in which files should be copied

- **fs** (*Optional[FileSystem]*) – if supplied, use fs instead of automatically chosen FileSystem for URI

Returns
a local path

Return type
str

get_tmp_dir

get_tmp_dir() → TemporaryDirectory

Return temporary directory given by the RVConfig.

Returns
A context manager.

Return type
TemporaryDirectory

is_archive

is_archive(*uri: str*) → bool

Check if the URI's extension represents an archived file.

Parameters
uri (*str*) –

Return type
bool

is_local

is_local(*uri: str*) → bool

Parameters
uri (*str*) –

Return type
bool

json_to_file

json_to_file(*content_dict: dict*, *uri: str*)

Upload JSON file to uri based on content_dict.

Parameters

- **content_dict** (*dict*) –
- **uri** (*str*) –

list_paths

list_paths(*uri: str*, *ext: str = ''*, *fs: Optional[FileSystem] = None*) → List[str]

List paths rooted at URI.

Optionally only includes paths with a certain file extension.

Parameters

- **uri** (*str*) – the URI of a directory
- **ext** (*str*) – the optional file extension to filter by
- **fs** (*Optional[FileSystem]*) – if supplied, use fs instead of automatically chosen FileSystem for uri

Return type
List[str]

start_sync

start_sync(*src_dir: str*, *dst_dir_uri: str*, *sync_interval: int = 600*, *fs: Optional[FileSystem] = None*)

Repeatedly sync a local source directory to a destination on a schedule.

Calls sync_to_dir on a schedule.

Parameters

- **src_dir** (*str*) – path of the local source directory
- **dst_dir_uri** (*str*) – URI of destination directory
- **sync_interval** (*int*) – period in seconds for syncing
- **fs** (*Optional[FileSystem]*) – if supplied, use fs instead of automatically chosen FileSystem

str_to_file

str_to_file(*content_str: str*, *uri: str*, *fs: Optional[FileSystem] = None*)

Writes string to text file.

Parameters

- **content_str** (*str*) – string to write
- **uri** (*str*) – URI of file to write
- **fs** (*Optional[FileSystem]*) – if supplied, use fs instead of automatically chosen FileSystem

Raises
NotWritableError if uri cannot be written –

sync_from_dir

sync_from_dir(*src_dir_uri: str*, *dst_dir: str*, *delete: bool = False*, *fs: Optional[FileSystem] = None*)

Synchronize a source directory to local destination directory.

Transfers files from source to destination directories so that the destination has all the source files. If FileSystem is remote, this involves downloading.

Parameters

- **src_dir_uri** (*str*) – URI of source directory
- **dst_dir** (*str*) – path of local destination directory
- **delete** (*bool*) – if True, delete files in the destination to match those in the source directory
- **fs** (*Optional[FileSystem]*) – if supplied, use fs instead of automatically chosen FileSystem for dst_dir_uri

sync_to_dir

sync_to_dir(*src_dir: str*, *dst_dir_uri: str*, *delete: bool = False*, *fs: Optional[FileSystem] = None*)

Synchronize a local source directory to destination directory.

Transfers files from source to destination directories so that the destination has all the source files. If FileSystem is remote, this involves uploading.

Parameters

- **src_dir** (*str*) – path of local source directory
- **dst_dir_uri** (*str*) – URI of destination directory
- **delete** (*bool*) – if True, delete files in the destination to match those in the source directory
- **fs** (*Optional[FileSystem]*) – if supplied, use fs instead of automatically chosen FileSystem for dst_dir_uri

unzip

unzip(*zip_path: str*, *target_dir: str*)

Unzip contents of zip file at zip_path into target_dir.

Creates target_dir if needed.

Parameters

- **zip_path** (*str*) –
- **target_dir** (*str*) –

upload_or_copy

upload_or_copy(*src_path: str*, *dst_uri: str*, *fs: Optional[FileSystem] = None*) → None

Upload or copy a file.

If dst_uri is local, the file is copied. Otherwise, it is uploaded.

Parameters

- **src_path** (*str*) – path to source file
- **dst_uri** (*str*) – URI of destination for file
- **fs** (*Optional[FileSystem]*) – if supplied, use fs instead of automatically chosen FileSystem for dst_uri

Raises

NotWritableError if dst_uri cannot be written to –

Return type

None

zipdir

zipdir(*dir: str*, *zip_path: str*)

Save a zip file with contents of directory.

Contents of directory will be at root of zip file.

Parameters

- **dir** (*str*) – directory to zip
- **zip_path** (*str*) – path to zip file to create

9.1.4 pipeline

Classes

Pipeline	A pipeline of commands to run sequentially.

Pipeline

class Pipeline

Bases: object

A pipeline of commands to run sequentially.

This is an abstraction over a sequence of commands. Each command is represented by a method. This base class has two test commands, and new pipelines should be created by subclassing this.

Note that any split command methods should have the following signature: def my_command(self, split_ind: int = 0, num_splits: int = 1) The num_splits represents how many parallel jobs should be created, and the split_ind is the index of the current job within that set.

commands

command names listed in the order in which they should run

Type
List[str]

split_commands

names of commands that can be split and run in parallel

Type
List[str]

gpu_commands

names of commands that should be executed on GPUs if available

Type
List[str]

Attributes

commands
gpu_commands
split_commands

__init__(*config:* PipelineConfig, *tmp_dir: str*)

Constructor

Parameters

- **config** (PipelineConfig) – the configuration of this pipeline
- **tmp_dir** (*str*) – the root any temporary directories created by running this pipeline

Methods

__init__(config, tmp_dir)	Constructor
test_cpu([split_ind, num_splits])	A command to test the ability to run split jobs on CPU.
test_gpu()	A command to test the ability to run on GPU.

__init__(*config:* PipelineConfig, *tmp_dir: str*)

Constructor

Parameters

- **config** (PipelineConfig) – the configuration of this pipeline
- **tmp_dir** (*str*) – the root any temporary directories created by running this pipeline

test_cpu(*split_ind: int = 0*, *num_splits: int = 1*)

A command to test the ability to run split jobs on CPU.

Parameters

- **split_ind** (*int*) –
- **num_splits** (*int*) –

test_gpu()

A command to test the ability to run on GPU.

commands: List[str] = ['test_cpu', 'test_gpu']

gpu_commands: List[str] = ['test_gpu']

split_commands: List[str] = ['test_cpu']

9.1.5 pipeline_config

Configs

PipelineConfig	Base class for configuring *Pipelines*.

PipelineConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model PipelineConfig

Base class for configuring *Pipelines*.

This should be subclassed to configure new Pipelines.


```
{
   "title": "PipelineConfig",
   "description": "Base class for configuring :class:`Pipelines <.Pipeline>`.\n\
↪nThis should be subclassed to configure new Pipelines.",
   "type": "object",
   "properties": {
      "root_uri": {
         "title": "Root Uri",
         "description": "The root URI for output generated by the pipeline",
         "type": "string"
      },
      "rv_config": {
         "title": "Rv Config",
         "description": "Used to store serialized RVConfig so pipeline can run in␣
↪remote environment with the local RVConfig. This should not be set explicitly by␣
↪users -- it is only used by the runner when running a remote pipeline.",
         "type": "object"
      },
      "plugin_versions": {
```


(continues on next page)

(continued from previous page)

```
        "title": "Plugin Versions",
        "description": "Used to store a mapping of plugin module paths to the␣
↪latest version number. This should not be set explicitly by users -- it is set␣
↪automatically when serializing and saving the config to disk.",
        "type": "object",
        "additionalProperties": {
           "type": "integer"
        }
     },
     "type_hint": {
        "title": "Type Hint",
        "default": "pipeline",
        "enum": [
           "pipeline"
        ],
        "type": "string"
     }
  },
  "additionalProperties": false
}
```

Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *plugin_versions (Optional[Dict[str, int]])*
- *root_uri (str)*
- *rv_config (dict)*
- *type_hint (Literal['pipeline'])*

field plugin_versions: Optional[Dict[str, int]] = None

Used to store a mapping of plugin module paths to the latest version number. This should not be set explicitly by users – it is set automatically when serializing and saving the config to disk.

field root_uri: str = None

The root URI for output generated by the pipeline

field rv_config: dict = None

Used to store serialized RVConfig so pipeline can run in remote environment with the local RVConfig. This should not be set explicitly by users – it is only used by the runner when running a remote pipeline.

field type_hint: Literal['pipeline'] = 'pipeline'

build(*tmp_dir: str*) → *Pipeline*

Return a pipeline based on this configuration.

Subclasses should override this to return an instance of the corresponding subclass of Pipeline.

Parameters

tmp_dir (*str*) – root of any temporary directory to pass to pipeline

Return type
Pipeline

get_config_uri() → str

Get URI of serialized version of this PipelineConfig.

Return type
str

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(**args*, ***kwargs*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises
ConfigError – if field is invalid

9.1.6 registry

Classes

Registry	A registry for resources that are built-in or contributed by plugins.

Registry

class Registry

Bases: object

A registry for resources that are built-in or contributed by plugins.

__init__()

Methods

__init__()	
add_config(type_hint, config, plugin[, upgrader])	Add a Config.
add_file_system(file_system)	Add a FileSystem.
add_plugin_command(cmd)	Add a click command contributed by a plugin.
add_runner(runner_name, runner)	Add a Runner.
add_rv_config_schema(config_section, ...)	Add section of schema used by RVConfig.
discover_plugins()	Discover all raster vision plugins.
get_config(type_hint)	Get a Config class associated with a type_hint.
get_file_system(uri[, mode])	Get a FileSystem used to handle the file type of a URI.
get_plugin(type_hint)	Get module path of plugin when Config class with type_hint is defined.
get_plugin_commands()	Get the click commands contributed by plugins.
get_plugin_from_alias(alias)	
get_plugin_version(plugin)	Get latest version of plugin.
get_runner(runner_name)	Return a Runner class based on its name.
get_rv_config_schema()	Return RVConfig schema.
get_type_hint_lineage(type_hint)	Get the lineage for a type hint.
get_upgrader(type_hint)	Get function that upgrades config dicts for type_hint.
load_builtins()	Add all builtin resources.
load_plugins([plugin_names])	Load plugins and register their resources.
set_plugin_aliases(plugin, aliases)	
set_plugin_version(plugin, version)	Set the latest version of a plugin.
update_config_info()	

__init__()

add_config(*type_hint: str*, *config: Type[*Config*]*, *plugin: str*, *upgrader=None*)

Add a Config.

Parameters

- **type_hint** (*str*) – the type hint used for deserialization of dict to an instance of config
- **config** (*Type[Config]*) – Config class
- **plugin** (*str*) –

add_file_system(*file_system:* FileSystem)

Add a FileSystem.

Parameters
file_system (FileSystem) – the FileSystem class to add

add_plugin_command(*cmd: Command*)

Add a click command contributed by a plugin.

Parameters
cmd (*Command*) –

add_runner(*runner_name: str*, *runner: Type[*Runner*]*)

Add a Runner.

Parameters

- **runner_name** (*str*) – the name of the runner that is passed to the CLI
- **runner** (*Type[*Runner*]*) – the Runner class

add_rv_config_schema(*config_section: str*, *config_fields: List[str]*)

Add section of schema used by RVConfig.

Parameters

- **config_section** (*str*) – name of section
- **config_fields** (*List[str]*) – list of field names within section

discover_plugins()

Discover all raster vision plugins.

get_config(*type_hint: str*) → Type[*Config*]

Get a Config class associated with a type_hint.

Parameters
type_hint (*str*) –

Return type
Type[*Config*]

get_file_system(*uri: str*, *mode: str = 'r'*) → Type[*FileSystem*]

Get a FileSystem used to handle the file type of a URI.

Parameters

- **uri** (*str*) – a URI to be opened by a registered FileSystem
- **mode** (*str*) – mode for opening file (eg. r or w)

Returns
the first FileSystem class which can handle opening the uri

Return type
Type[*FileSystem*]

get_plugin(*type_hint: str*) → str

Get module path of plugin when Config class with type_hint is defined.

Parameters
type_hint (*str*) –

Return type
str

get_plugin_commands() → List[Command]

Get the click commands contributed by plugins.

Return type
List[*Command*]

get_plugin_from_alias(*alias: str*) → Optional[str]

Parameters
alias (*str*) –

Return type
Optional[str]

get_plugin_version(*plugin: str*) → int

Get latest version of plugin.

Parameters
plugin (*str*) – the module path of the plugin

Return type
int

get_runner(*runner_name: str*) → Type[*Runner*]

Return a Runner class based on its name.

Parameters
runner_name (*str*) –

Return type
Type[*Runner*]

get_rv_config_schema()

Return RVConfig schema.

get_type_hint_lineage(*type_hint: str*) → List[str]

Get the lineage for a type hint.

Returns
List of type hints of all Config classes in the subclass is-a "lineage" from Config to the class with type hint type_hint.

Parameters
type_hint (*str*) –

Return type
List[str]

get_upgrader(*type_hint: str*) → Optional[Callable]

Get function that upgrades config dicts for type_hint.

Parameters
type_hint (*str*) –

Return type
Optional[*Callable*]

load_builtins()

Add all builtin resources.

load_plugins(*plugin_names: Optional[Iterable[str]] = None*) → None

Load plugins and register their resources.

Import each Python module within the rastervision namespace package and call the register_plugin function at its root (if it exists).

Parameters
plugin_names (*Optional[Iterable[str]]*) –

Return type
None

set_plugin_aliases(*plugin: str*, *aliases: List[str]*)

Parameters

- **plugin** (*str*) –
- **aliases** (*List[str]*) –

set_plugin_version(*plugin: str*, *version: int*)

Set the latest version of a plugin.

Parameters

- **plugin** (*str*) – module path of plugin (eg. rastervision.core)
- **version** (*int*) – a non-negative integer version number that should be incremented each time a config schema changes

update_config_info()

Exceptions

RegistryError	Exception raised for invalid use of registry.

rastervision.pipeline.registry.RegistryError

exception RegistryError

Exception raised for invalid use of registry.

__init__(**args*, ***kwargs*)

__new__(***kwargs*)

9.1.7 runner

Modules

inprocess_runner
local_runner
runner

inprocess_runner

Classes

InProcessRunner	Runs each command sequentially within a single process.

InProcessRunner

class InProcessRunner

Bases: *Runner*

Runs each command sequentially within a single process.

Useful for testing and debugging.

__init__()

Methods

__init__()

get_split_ind()	Get the split_ind for the process.
run(cfg_json_uri, pipeline, commands[, ...])	Run commands in a Pipeline using a serialized PipelineConfig.

get_split_ind() → Optional[int]

Get the split_ind for the process.

For split commands, the split_ind determines which split of work to perform within the current OS process. The CLI has a –split-ind option, but some runners may have their own means of communicating the split_ind, and this method should be overridden in such cases. If this method returns None, then the –split-ind option will be used. If both are null, then it won't be possible to run the command.

Return type
Optional[int]

run(*cfg_json_uri*, *pipeline*, *commands*, *num_splits=1*, *pipeline_run_name: str = 'raster-vision'*)

Run commands in a Pipeline using a serialized PipelineConfig.

Parameters

- **cfg_json_uri** – URI of a JSON file with a serialized PipelineConfig
- **pipeline** – the Pipeline to run
- **commands** – names of commands to run
- **num_splits** – number of splits to use for splittable commands
- **pipeline_run_name** (*str*) –

local_runner

Classes

LocalRunner	Runs each command locally using different processes for each command/split.

LocalRunner

class LocalRunner

Bases: *Runner*

Runs each command locally using different processes for each command/split.

This is implemented by generating a Makefile and then running it using make.

__init__()

Methods

__init__()

get_split_ind()	Get the split_ind for the process.
run(cfg_json_uri, pipeline, commands[, ...])	Run commands in a Pipeline using a serialized PipelineConfig.

get_split_ind() → Optional[int]

Get the split_ind for the process.

For split commands, the split_ind determines which split of work to perform within the current OS process. The CLI has a –split-ind option, but some runners may have their own means of communicating the split_ind, and this method should be overridden in such cases. If this method returns None, then the –split-ind option will be used. If both are null, then it won't be possible to run the command.

Return type
Optional[int]

run(*cfg_json_uri*, *pipeline*, *commands*, *num_splits=1*, *pipeline_run_name: str = 'raster-vision'*)

Run commands in a Pipeline using a serialized PipelineConfig.

Parameters

- **cfg_json_uri** – URI of a JSON file with a serialized PipelineConfig
- **pipeline** – the Pipeline to run
- **commands** – names of commands to run
- **num_splits** – number of splits to use for splittable commands
- **pipeline_run_name** (*str*) –

runner

Classes

Runner	A method for running a Pipeline.

Runner

class Runner

Bases: object

A method for running a Pipeline.

This can be subclassed to provide the ability to run on different cloud providers, etc.

__init__()

Methods

__init__()	
get_split_ind()	Get the split_ind for the process.
run(cfg_json_uri, pipeline, commands[, ...])	Run commands in a Pipeline using a serialized PipelineConfig.

get_split_ind() → Optional[int]

Get the split_ind for the process.

For split commands, the split_ind determines which split of work to perform within the current OS process. The CLI has a –split-ind option, but some runners may have their own means of communicating the split_ind, and this method should be overridden in such cases. If this method returns None, then the –split-ind option will be used. If both are null, then it won't be possible to run the command.

Return type
Optional[int]

abstract run(*cfg_json_uri: str*, *pipeline:* Pipeline, *commands: List[str]*, *num_splits: int = 1*, *pipeline_run_name: str = 'raster-vision'*)

Run commands in a Pipeline using a serialized PipelineConfig.

Parameters

- **cfg_json_uri** (*str*) – URI of a JSON file with a serialized PipelineConfig
- **pipeline** (Pipeline) – the Pipeline to run
- **commands** (*List[str]*) – names of commands to run
- **num_splits** (*int*) – number of splits to use for splittable commands
- **pipeline_run_name** (*str*) –

9.1.8 rv_config

Classes

RVConfig	A store of global user-specific configuration not tied to particular pipelines.

RVConfig

class RVConfig

Bases: object

A store of global user-specific configuration not tied to particular pipelines.

This is used to store user-specific configuration like the root temporary directory, verbosity, and other system-wide configuration handled by Everett (eg. which AWS Batch job queue to use).

DEFAULT_PROFILE

the default RV configuration profile name

Type
str

DEFAULT_TMP_DIR_ROOT

the default location for root of temporary directories

Type
str

Attributes

DEFAULT_PROFILE
DEFAULT_TMP_DIR_ROOT

__init__()

Methods

__init__()

get_cache_dir()	Return the cache directory.
get_config_dict(rv_config_schema)	Get all Everett configuration.
get_namespace_config(namespace)	Get the key-val pairs associated with a namespace.
get_tmp_dir()	Return a new TemporaryDirectory object.
get_tmp_dir_root()	Return the root of all temp dirs.
get_verbosity()	Returns verbosity level for logging.
set_everett_config([profile, rv_home, ...])	Set Everett config.
set_tmp_dir_root([tmp_dir_root])	Set root of all temporary directories.
set_verbosity([verbosity])	Set verbosity level for logging.

__init__()

get_cache_dir() → TemporaryDirectory

Return the cache directory.

Return type
TemporaryDirectory

get_config_dict(*rv_config_schema: Dict[str, List[str]]*) → Dict[str, str]

Get all Everett configuration.

This method is used to serialize an Everett configuration so it can be used on a remote instance.

Parameters
rv_config_schema (*Dict[str, List[str]]*) – each key is a namespace; each value is list of keys within that namespace

Returns
Each key is of form namespace_i_key_ij with corresponding value val_ij.

Return type
Dict[str, str]

get_namespace_config(*namespace: str*) → Dict[str, str]

Get the key-val pairs associated with a namespace.

Parameters
namespace (*str*) –

Return type
Dict[str, str]

get_tmp_dir() → TemporaryDirectory

Return a new TemporaryDirectory object.

Return type
TemporaryDirectory

get_tmp_dir_root() → str

Return the root of all temp dirs.

Return type
str

get_verbosity() → *Verbosity*

Returns verbosity level for logging.

Return type
Verbosity

set_everett_config(*profile: Optional[str] = None*, *rv_home: Optional[str] = None*, *config_overrides: Optional[Dict[str, str]] = None*)

Set Everett config.

This sets up any other configuration using the Everett library. See https://everett.readthedocs.io/

It roughly mimics the behavior of how the AWS CLI is configured, if that is a helpful analogy. Configuration can be specified through configuration files, environment variables, and the config_overrides argument in increasing order of precedence.

Configuration files are in the following format: ``` [namespace_1] key_11=val_11 … key_1n=val_1n

…

[namespace_m] key_m1=val_m1 ... key_mn=val_mn ```

Each namespace can be used for the configuration of a different plugin. Each configuration file is a "profile" with the name of the file being the name of the profile. This supports switching between different configuration sets. The corresponding environment variable setting for namespace_i and key_ij is *namespace_i_key_ij=val_ij*.

Parameters

- **profile** (*Optional[str]*) – name of the RV configuration profile to use. If not set, defaults to value of RV_PROFILE env var, or DEFAULT_PROFILE.
- **rv_home** (*Optional[str]*) – a local dir with RV configuration files. If not set, attempts to use ~/.rastervision.
- **config_overrides** (*Optional[Dict[str, str]]*) – any configuration to override. Each key is of form namespace_i_key_ij with corresponding value val_ij.

set_tmp_dir_root(*tmp_dir_root: Optional[str] = None*)

Set root of all temporary directories.

To set the value, the following rules are used in decreasing priority:

1) the tmp_dir_root argument if it is not None
2) an environment variable (TMPDIR, TEMP, or TMP)
3) a default temporary directory which is
4) a directory returned by tempfile.TemporaryDirectory()

Parameters
tmp_dir_root (*Optional[str]*) –

set_verbosity(*verbosity: Verbosity = 1*)

Set verbosity level for logging.

Parameters
verbosity (*Verbosity*) –

DEFAULT_PROFILE: str = 'default'

DEFAULT_TMP_DIR_ROOT: str = '/opt/data/tmp'

Functions

load_conf_list(s)	Loads a list of items from the config.

load_conf_list

load_conf_list(*s*)

Loads a list of items from the config.

Lists should be comma separated.

This takes into account that previous versions of Raster Vision allowed for a *["module"]* like syntax, even though that didn't work for multi-value lists.

9.1.9 utils

Functions

grouped(lst, size)	Returns a list of lists of length 'size'.
split_into_groups(lst, num_groups)	Attempts to split a list into a given number of groups.
terminate_at_exit(process)	

grouped

grouped(*lst*, *size*)

Returns a list of lists of length 'size'. The last list will have size <= 'size'.

split_into_groups

split_into_groups(*lst*, *num_groups*)

Attempts to split a list into a given number of groups. The number of groups will be at least 1 and at most num_groups.

Parameters

- **lst** – The list to split
- **num_groups** – The number of groups to create.

Returns

A list of size between 1 and num_groups containing lists of items of l.

terminate_at_exit

terminate_at_exit(*process*)

9.1.10 verbosity

Classes

Verbosity	Verbosity level for the sake of logging.

Verbosity

class Verbosity

Bases: object

Verbosity level for the sake of logging.

Attributes

DEBUG
NORMAL
QUIET
VERBOSE
VERY_VERBOSE

__init__()

Methods

__init__()	
get()	Get the verbosity from RVConfig.

static get() → *Verbosity*

Get the verbosity from RVConfig.

Return type
Verbosity

DEBUG = 4

NORMAL = 1

QUIET = 0

VERBOSE = 2

VERY_VERBOSE = 3

9.1.11 version

Library verison

9.2 core

Modules

9.2.1 analyzer

Modules

analyzer

Classes

Analyzer	Analyzes scenes and writes some output while running the analyze command.

Analyzer

class Analyzer

Bases: ABC

Analyzes scenes and writes some output while running the analyze command.

This output can be used to normalize images, for example.

__init__()

Methods

__init__()	
process(scenes, tmp_dir)	Process scenes and save result.

abstract process(*scenes: List[*Scene*]*, *tmp_dir: str*)

Process scenes and save result.

Parameters

- **scenes** (*List[*Scene*]*) –
- **tmp_dir** (*str*) –

analyzer_config

Configs

AnalyzerConfig	Configure an *Analyzer*.

AnalyzerConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model AnalyzerConfig

Configure an *Analyzer*.


```
{
   "title": "AnalyzerConfig",
   "description": "Configure an :class:`.Analyzer`.",
   "type": "object",
   "properties": {
      "type_hint": {
         "title": "Type Hint",
         "default": "analyzer",
         "enum": [
            "analyzer"
```


(continues on next page)

(continued from previous page)

```
        ],
        "type": "string"
      }
    },
    "additionalProperties": false
}
```

Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *type_hint (Literal['analyzer'])*

field type_hint: Literal['analyzer'] = 'analyzer'

build(*scene_group: Optional[Tuple[str, Iterable[str]]] = None*) → *Analyzer*

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

Parameters
scene_group (*Optional[Tuple[str, Iterable[str]]]*) –

Return type
Analyzer

get_bundle_filenames() → List[str]

Returns the names of files that should be included in a model bundle.

The files are assumed to be in the analyze/ directory generated by the analyze command. Note that only the names, not the full paths should be returned.

Return type
List[str]

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(**args*, ***kwargs*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

stats_analyzer

Classes

StatsAnalyzer	Compute imagery statistics of scenes.

StatsAnalyzer

class StatsAnalyzer

Bases: *Analyzer*

Compute imagery statistics of scenes.

__init__(*stats_uri: Optional[str] = None*, *sample_prob: float = 0.1*)

Parameters

- **stats_uri** (*Optional[str]*) –
- **sample_prob** (*float*) –

Methods

__init__([stats_uri, sample_prob])	
compute_stats(scenes)	
process(scenes, tmp_dir)	Process scenes and save result.

__init__(*stats_uri: Optional[str] = None*, *sample_prob: float = 0.1*)

Parameters

- **stats_uri** (*Optional[str]*) –
- **sample_prob** (*float*) –

compute_stats(*scenes: Iterable[Scene]*) → *RasterStats*

Parameters
scenes (*Iterable[Scene]*) –

Return type
RasterStats

process(*scenes: Iterable[Scene]*, *tmp_dir: str*) → None

Process scenes and save result.

Parameters

- **scenes** (*Iterable[Scene]*) –
- **tmp_dir** (*str*) –

Return type
None

stats_analyzer_config

Configs

StatsAnalyzerConfig	Configure a *StatsAnalyzer*.

StatsAnalyzerConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model StatsAnalyzerConfig

Configure a *StatsAnalyzer*.

A *StatsAnalyzer* computes imagery statistics of scenes which can be used to normalize chips read from them.


```
{
   "title": "StatsAnalyzerConfig",
   "description": "Configure a :class:`.StatsAnalyzer`.\n\nA :class:`.
↪StatsAnalyzer` computes imagery statistics of scenes which can\nbe used to␣
↪normalize chips read from them.",
   "type": "object",
   "properties": {
      "type_hint": {
         "title": "Type Hint",
         "default": "stats_analyzer",
         "enum": [
            "stats_analyzer"
         ],
         "type": "string"
      },
      "output_uri": {
         "title": "Output Uri",
```


(continues on next page)

(continued from previous page)

```
        "description": "URI of directory where stats will be saved. Stats for a␣
↪scene-group will be save in a JSON file at <output_uri>/<scene-group-name>/stats.
↪json. If None, and this Config is part of an RVPipeline, this field will be auto-
↪generated.",
        "type": "string"
      },
      "sample_prob": {
        "title": "Sample Prob",
        "description": "The probability of using a random window for computing␣
↪statistics. If None, will use a sliding window.",
        "default": 0.1,
        "type": "number"
      }
   },
   "additionalProperties": false
}
```

Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *output_uri (Optional[str])*
- *sample_prob (Optional[float])*
- *type_hint (Literal['stats_analyzer'])*

field output_uri: Optional[str] = None

URI of directory where stats will be saved. Stats for a scene-group will be save in a JSON file at <output_uri>/<scene-group-name>/stats.json. If None, and this Config is part of an RVPipeline, this field will be auto-generated.

field sample_prob: Optional[float] = 0.1

The probability of using a random window for computing statistics. If None, will use a sliding window.

field type_hint: Literal['stats_analyzer'] = 'stats_analyzer'

build(*scene_group: Optional[Tuple[str, Iterable[str]]] = None*) → *StatsAnalyzer*

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

Parameters

scene_group (*Optional[Tuple[str, Iterable[str]]]*) –

Return type

StatsAnalyzer

get_bundle_filenames()

Returns the names of files that should be included in a model bundle.

The files are assumed to be in the analyze/ directory generated by the analyze command. Note that only the names, not the full paths should be returned.

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*pipeline: Optional[RVPipelineConfig] = None*) → None

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

Parameters
pipeline (*Optional[RVPipelineConfig]*) –

Return type
None

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises
ConfigError – if field is invalid

9.2.2 backend

Modules

backend

backend_config

backend

Classes

Backend	Abstraction around core ML functionality used by an RVPipeline.
SampleWriter	Writes DataSamples in a streaming fashion.

Backend

class Backend

Bases: ABC

Abstraction around core ML functionality used by an RVPipeline.

This should be subclassed to enable use of a third party ML library with an RVPipeline. There is a one-to-many relationship from RVPipeline to Backend.

__init__()

Methods

__init__()	
get_sample_writer()	Returns a SampleWriter for this Backend.
load_model()	Load the model in preparation for one or more prediction calls.
predict_scene(scene, chip_sz, stride)	Return predictions for an entire scene using the model.
train()	Train a model.

abstract get_sample_writer()

Returns a SampleWriter for this Backend.

abstract load_model()

Load the model in preparation for one or more prediction calls.

abstract predict_scene(*scene:* Scene, *chip_sz: int*, *stride: int*) → *Labels*

Return predictions for an entire scene using the model.

Parameters

- **scene** (Scene) – Scene to run inference on.
- **chip_sz** (*int*) –
- **stride** (*int*) –

Returns

Labels object containing predictions

Return type

Labels

abstract train()

Train a model.

This should download chips created by the SampleWriter, train the model, and then saving it to disk.

SampleWriter

class SampleWriter

Bases: AbstractContextManager

Writes DataSamples in a streaming fashion.

This is a context manager used for creating training and validation chips, and should be subclassed for each Backend.

__init__()

Methods

__init__()	
write_sample(sample)	Writes a single sample.

abstract write_sample(*sample: DataSample*)

Writes a single sample.

Parameters

sample (*DataSample*) –

backend_config

Configs

BackendConfig	Configure a *Backend*.

BackendConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model BackendConfig

Configure a *Backend*.


```
{
   "title": "BackendConfig",
   "description": "Configure a :class:`.Backend`.",
   "type": "object",
   "properties": {
      "type_hint": {
```


(continues on next page)

(continued from previous page)


```
            "title": "Type Hint",
            "default": "backend",
            "enum": [
               "backend"
            ],
            "type": "string"
         }
      },
      "additionalProperties": false
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *type_hint (Literal['backend'])*

field type_hint: Literal['backend'] = 'backend'

build(*pipeline:* RVPipeline, *tmp_dir: str*) → *Backend*

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

Parameters

- **pipeline** (RVPipeline) –
- **tmp_dir** (*str*) –

Return type

Backend

filter_commands(*commands: List[str]*) → List[str]

Filter out any commands that are not needed or supported.

Parameters

commands (*List[str]*) –

Return type

List[str]

get_bundle_filenames() → List[str]

Returns the names of files that should be included in a model bundle.

The files are assumed to be in the train/ directory generated by the train command. Note that only the names, not the full paths should be returned.

Return type

List[str]

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*pipeline: Optional[RVPipeline] = None*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

Parameters

pipeline (*Optional[RVPipeline]*) –

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

9.2.3 box

Classes

Box	A multi-purpose box (ie.
NonNegInt	alias of ConstrainedIntValue

Box

class Box

Bases: object

A multi-purpose box (ie. rectangle) representation .

Attributes

area	Return area of Box.
height	Return height of Box.
size	
width	Return width of Box.

__init__(*ymin*, *xmin*, *ymax*, *xmax*)

Construct a bounding box.

Unless otherwise stated, the convention is that these coordinates are in pixel coordinates and represent boxes that lie within a RasterSource.

Parameters

- **ymin** – minimum y value (y is row)
- **xmin** – minimum x value (x is column)
- **ymax** – maximum y value
- **xmax** – maximum x value

Methods

__init__(ymin, xmin, ymax, xmax)	Construct a bounding box.
buffer(buffer_sz, max_extent)	Return new Box whose sides are buffered by buffer_sz.
center_crop(edge_offset_y, edge_offset_x)	Return Box whose sides are eroded by the given offsets.
copy()	
erode(erosion_sz)	Return new Box whose sides are eroded by erosion_sz.
filter_by_aoi(windows, aoi_polygons[, within])	Filters windows by a list of AOI polygons
from_dict(d)	
from_npbox(npbox)	Return new Box based on npbox format.
from_rasterio(rio_window)	
from_shapely(shape)	
geojson_coordinates()	Return Box as GeoJSON coordinates.
get_windows(size, stride[, padding, ...])	Returns a list of boxes representing windows generated using a sliding window traversal with the specified size, stride, and padding.
intersection(other)	Return the intersection of this Box and the other.
intersects(other)	
make_random_box_container(out_h, out_w)	Return a new rectangular Box that contains this Box.
make_random_square(size)	Return new randomly positioned square Box that lies inside this Box.

continues on next page

Table 1 – continued from previous page

make_random_square_container(size)	Return a new square Box that contains this Box.
make_square(ymin, xmin, size)	Return new square Box.
npbox_format()	Return Box in npbox format used by TF Object Detection API.
pad(ymin, xmin, ymax, xmax)	Pad sides by the given amount.
rasterio_format()	Return Box in Rasterio format: ((ymin, ymax), (xmin, xmax)).
reproject(transform_fn)	Reprojects this box based on a transform function.
shapely_format()	
shift_origin(extent)	Shift origin of window coords to (extent.xmin, extent.ymin).
to_dict()	
to_int()	
to_npboxes(boxes)	Return nx4 numpy array from list of Box.
to_offsets(container)	Convert coords to offsets from (container.xmin, container.ymin).
to_points()	Get (x, y) coords of each vertex as a 4x2 numpy array.
to_rasterio()	Convert to a Rasterio Window.
to_shapely()	Convert to shapely Polygon.
to_slices()	Convert to slices: ymin:ymax, xmin:xmax
to_xywh()	
to_xyxy()	
translate(dy, dx)	Translate window along y and x axes by the given distances.
tuple_format()	
within_aoi(window, aoi_polygons)	Check if window is within a list of AOI polygons.

__init__(*ymin*, *xmin*, *ymax*, *xmax*)

Construct a bounding box.

Unless otherwise stated, the convention is that these coordinates are in pixel coordinates and represent boxes that lie within a RasterSource.

Parameters

- **ymin** – minimum y value (y is row)
- **xmin** – minimum x value (x is column)
- **ymax** – maximum y value
- **xmax** – maximum x value

buffer(*buffer_sz: float*, *max_extent:* Box) → *Box*

Return new Box whose sides are buffered by buffer_sz.

The resulting box is clipped so that the values of the corners are always greater than zero and less than the height and width of max_extent.

Parameters

- **buffer_sz** (*float*) –
- **max_extent** (Box) –

Return type
Box

center_crop(*edge_offset_y: int*, *edge_offset_x: int*) → *Box*

Return Box whose sides are eroded by the given offsets.

Box(0, 0, 10, 10).center_crop(2, 4) == Box(2, 4, 8, 6)

Parameters

- **edge_offset_y** (*int*) –
- **edge_offset_x** (*int*) –

Return type
Box

copy() → *Box*

Return type
Box

erode(*erosion_sz*) → *Box*

Return new Box whose sides are eroded by erosion_sz.

Return type
Box

static filter_by_aoi(*windows: List[*Box*]*, *aoi_polygons: List[Polygon]*, *within: bool = True*) → List[*Box*]

Filters windows by a list of AOI polygons

Parameters

- **within** (*bool*) – if True, windows are only kept if they lie fully within an AOI polygon. Otherwise, windows are kept if they intersect an AOI polygon.
- **windows** (*List[*Box*]*) –
- **aoi_polygons** (*List[Polygon]*) –

Return type
List[Box]

classmethod from_dict(*d: dict*) → *Box*

Parameters
d (*dict*) –

Return type
Box

static from_npbox(*npbox*)

Return new Box based on npbox format.

Parameters
npbox – Numpy array of form [ymin, xmin, ymax, xmax] with float type

classmethod from_rasterio(*rio_window: Window*) → *Box*

Parameters
rio_window (*Window*) –

Return type
Box

static from_shapely(*shape*)

geojson_coordinates() → List[Tuple[int, int]]

Return Box as GeoJSON coordinates.

Return type
List[*Tuple*[int, int]]

get_windows(*size: Union[PositiveInt, Tuple[PositiveInt, PositiveInt]]*, *stride: Union[PositiveInt, Tuple[PositiveInt, PositiveInt]]*, *padding: Optional[Union[ConstrainedIntValue, Tuple[ConstrainedIntValue, ConstrainedIntValue]]] = None*, *pad_direction: Literal['both', 'start', 'end'] = 'end'*) → List[*Box*]

Returns a list of boxes representing windows generated using a sliding window traversal with the specified size, stride, and padding.

Each of size, stride, and padding can be either a positive int or a tuple *(vertical-componet, horizontal-component)* of positive ints.

Padding currently only applies to the right and bottom edges.

Parameters

- **size** (*Union[PosInt, Tuple[PosInt, PosInt]]*) – Size (h, w) of the windows.
- **stride** (*Union[PosInt, Tuple[PosInt, PosInt]]*) – Step size between windows. Can be 2-tuple (h_step, w_step) or positive int.
- **padding** (*Optional[Union[PosInt, Tuple[PosInt, PosInt]]], optional*) – Optional padding to accomodate windows that overflow the extent. Can be 2-tuple (h_pad, w_pad) or non-negative int. If None, will be set to (size[0]//2, size[1]//2). Defaults to None.
- **pad_direction** (*Literal['both', 'start', 'end']*) – If 'end', only pad ymax and xmax (bottom and right). If 'start', only pad ymin and xmin (top and left). If 'both', pad all sides. Has no effect if paddiong is zero. Defaults to 'end'.

Returns
List of Box objects.

Return type
List[*Box*]

intersection(*other:* Box) → *Box*

Return the intersection of this Box and the other.

Parameters
other (Box) – The box to intersect with this one.

Returns
The intersection of this box and the other one.

Return type
Box

intersects(*other:* Box) → bool

Parameters
other (Box) –

Return type
bool

make_random_box_container(*out_h: int*, *out_w: int*) → *Box*

Return a new rectangular Box that contains this Box.

Parameters

- **out_h** (*int*) – the height of the new Box
- **out_w** (*int*) – the width of the new Box

Return type
Box

make_random_square(*size: int*) → *Box*

Return new randomly positioned square Box that lies inside this Box.

Parameters
size (*int*) – the height and width of the new Box

Return type
Box

make_random_square_container(*size*)

Return a new square Box that contains this Box.

Parameters
size – the width and height of the new Box

static make_square(*ymin*, *xmin*, *size*) → *Box*

Return new square Box.

Return type
Box

npbox_format()

Return Box in npbox format used by TF Object Detection API.

Returns
Numpy array of form [ymin, xmin, ymax, xmax] with float type

pad(*ymin: int*, *xmin: int*, *ymax: int*, *xmax: int*) → *Box*

Pad sides by the given amount.

Parameters

- **ymin** (*int*) –
- **xmin** (*int*) –
- **ymax** (*int*) –
- **xmax** (*int*) –

Return type
Box

rasterio_format() → Tuple[Tuple[int, int], Tuple[int, int]]

Return Box in Rasterio format: ((ymin, ymax), (xmin, xmax)).

Return type
Tuple[*Tuple*[int, int], *Tuple*[int, int]]

reproject(*transform_fn: Callable*) → *Box*

Reprojects this box based on a transform function.

Parameters

- **tuple** (*transform_fn - A function that takes in a*) – and reprojects that point to the target coordinate reference system.
- **transform_fn** (*Callable*) –

Return type
Box

shapely_format() → Tuple[int, int, int, int]

Return type
Tuple[int, int, int, int]

shift_origin(*extent:* Box) → *Box*

Shift origin of window coords to (extent.xmin, extent.ymin).

Parameters
extent (Box) –

Return type
Box

to_dict() → Dict[str, int]

Return type
Dict[str, int]

to_int()

static to_npboxes(*boxes*)

Return nx4 numpy array from list of Box.

to_offsets(*container:* Box) → *Box*

Convert coords to offsets from (container.xmin, container.ymin).

Parameters
container (Box) –

Return type
Box

to_points() → ndarray

Get (x, y) coords of each vertex as a 4x2 numpy array.

Return type
ndarray

to_rasterio() → Window

Convert to a Rasterio Window.

Return type
Window

to_shapely() → Polygon

Convert to shapely Polygon.

Return type
Polygon

to_slices() → Tuple[slice, slice]

Convert to slices: ymin:ymax, xmin:xmax

Return type
Tuple[slice, slice]

to_xywh() → Tuple[int, int, int, int]

Return type
Tuple[int, int, int, int]

to_xyxy() → Tuple[int, int, int, int]

Return type
Tuple[int, int, int, int]

translate(*dy: int*, *dx: int*) → *Box*

Translate window along y and x axes by the given distances.

Parameters

- **dy** (*int*) –
- **dx** (*int*) –

Return type
Box

tuple_format() → Tuple[int, int, int, int]

Return type
Tuple[int, int, int, int]

static within_aoi(*window:* Box, *aoi_polygons: List[Polygon]*) → bool

Check if window is within a list of AOI polygons.

Parameters

- **window** (Box) –
- **aoi_polygons** (*List[Polygon]*) –

Return type
bool

property area: int

Return area of Box.

property height: int

Return height of Box.

property size: Tuple[int, int]

property width: int

Return width of Box.

NonNegInt

NonNegInt

alias of ConstrainedIntValue

Exceptions

BoxSizeError

rastervision.core.box.BoxSizeError

exception BoxSizeError

__init__(**args*, ***kwargs*)

__new__(***kwargs*)

9.2.4 cli

Classes

OptionEatAll

OptionEatAll

class OptionEatAll

Bases: Option

Attributes

human_readable_name	Returns the human readable name of this parameter.
param_type_name	

__init__(**args*, ***kwargs*)

Methods

__init__(*args, **kwargs)	
add_to_parser(parser, ctx)	
consume_value(ctx, opts)	
get_default(ctx[, call])	Get the default for the parameter.
get_error_hint(ctx)	Get a stringified version of the param for use in error messages to indicate which param caused the error.
get_help_record(ctx)	
get_usage_pieces(ctx)	
handle_parse_result(ctx, opts, args)	
make_metavar()	
process_value(ctx, value)	
prompt_for_value(ctx)	This is an alternative flow that can be activated in the full value processing if a value does not exist.
resolve_envvar_value(ctx)	
shell_complete(ctx, incomplete)	Return a list of completions for the incomplete value.
to_info_dict()	Gather information that could be useful for a tool generating user-facing documentation.
type_cast_value(ctx, value)	Convert and validate a value against the option's `type`, `multiple`, and `nargs`.
value_from_envvar(ctx)	
value_is_missing(value)	

__init__(**args*, ***kwargs*)

add_to_parser(*parser*, *ctx*)

consume_value(*ctx: Context*, *opts: Mapping[str, Parameter]*) → Tuple[Any, ParameterSource]

Parameters

- **ctx** (*Context*) –
- **opts** (*Mapping[str, Parameter]*) –

Return type
Tuple[*Any*, *ParameterSource*]

get_default(*ctx: Context*, *call: bool = True*) → Optional[Union[Any, Callable[[], Any]]]

Get the default for the parameter. Tries `Context.lookup_default()` first, then the local default.

Parameters

- **ctx** (*Context*) – Current context.

- **call** (*bool*) – If the default is a callable, call it. Disable to return the callable instead.

Return type
Optional[*Union*[*Any*, *Callable*[[], *Any*]]]

Changed in version 8.0.2: Type casting is no longer performed when getting a default.

Changed in version 8.0.1: Type casting can fail in resilient parsing mode. Invalid defaults will not prevent showing help text.

Changed in version 8.0: Looks at `ctx.default_map` first.

Changed in version 8.0: Added the `call` parameter.

get_error_hint(*ctx: Context*) → str

Get a stringified version of the param for use in error messages to indicate which param caused the error.

Parameters
ctx (*Context*) –

Return type
str

get_help_record(*ctx: Context*) → Optional[Tuple[str, str]]

Parameters
ctx (*Context*) –

Return type
Optional[*Tuple*[str, str]]

get_usage_pieces(*ctx: Context*) → List[str]

Parameters
ctx (*Context*) –

Return type
List[str]

handle_parse_result(*ctx: Context*, *opts: Mapping[str, Any]*, *args: List[str]*) → Tuple[Any, List[str]]

Parameters

- **ctx** (*Context*) –
- **opts** (*Mapping[str, Any]*) –
- **args** (*List[str]*) –

Return type
Tuple[*Any*, *List*[str]]

make_metavar() → str

Return type
str

process_value(*ctx: Context*, *value: Any*) → Any

Parameters

- **ctx** (*Context*) –
- **value** (*Any*) –

Return type
Any

prompt_for_value(*ctx: Context*) → Any

This is an alternative flow that can be activated in the full value processing if a value does not exist. It will prompt the user until a valid value exists and then returns the processed value as result.

Parameters
ctx (*Context*) –

Return type
Any

resolve_envvar_value(*ctx: Context*) → Optional[str]

Parameters
ctx (*Context*) –

Return type
Optional[str]

shell_complete(*ctx: Context*, *incomplete: str*) → List[CompletionItem]

Return a list of completions for the incomplete value. If a `shell_complete` function was given during init, it is used. Otherwise, the `type` `shell_complete()` function is used.

Parameters

- **ctx** (*Context*) – Invocation context for this command.
- **incomplete** (*str*) – Value being completed. May be empty.

Return type
List[CompletionItem]

New in version 8.0.

to_info_dict() → Dict[str, Any]

Gather information that could be useful for a tool generating user-facing documentation.

Use `click.Context.to_info_dict()` to traverse the entire CLI structure.

New in version 8.0.

Return type
Dict[str, *Any*]

type_cast_value(*ctx: Context*, *value: Any*) → Any

Convert and validate a value against the option's `type`, `multiple`, and `nargs`.

Parameters

- **ctx** (*Context*) –
- **value** (*Any*) –

Return type
Any

value_from_envvar(*ctx: Context*) → Optional[Any]

Parameters
ctx (*Context*) –

Return type
Optional[*Any*]

value_is_missing(*value: Any*) → bool

Parameters
value (*Any*) –

Return type
bool

property human_readable_name: str

Returns the human readable name of this parameter. This is the same as the name for options, but the metavar for arguments.

param_type_name = 'option'

9.2.5 data

Modules

class_config

crs_transformer

dataset_config

label

label_source

label_store

raster_source

raster_transformer

scene

scene_config

utils

vector_source

vector_transformer

class_config

Configs

ClassConfig	Configure class information for a machine learning task.

ClassConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model ClassConfig

Configure class information for a machine learning task.


```
{
   "title": "ClassConfig",
   "description": "Configure class information for a machine learning task.",
   "type": "object",
   "properties": {
      "names": {
         "title": "Names",
         "description": "Names of classes. The i-th class in this list will have␣
↪class ID = i.",
         "type": "array",
         "items": {
            "type": "string"
         }
      },
      "colors": {
         "title": "Colors",
         "description": "Colors used to visualize classes. Can be color strings␣
↪accepted by matplotlib or RGB tuples. If None, a random color will be auto-
↪generated for each class.",
         "type": "array",
         "items": {
            "anyOf": [
               {
                  "type": "string"
               },
               {
                  "type": "array",
                  "items": {}
               }
            ]
         }
      },
      "null_class": {
         "title": "Null Class",
         "description": "Optional name of class in `names` to use as the null class.
↪ This is used in semantic segmentation to represent the label for imagery pixels␣
↪that are NODATA or that are missing a label. If None and the class names include \
```


(continues on next page)

(continued from previous page)


```
↪"null\", it will automatically be used as the null class. If None, and this␣
↪Config is part of a SemanticSegmentationConfig, a null class will be added␣
↪automatically.",
         "type": "string"
      },
      "type_hint": {
         "title": "Type Hint",
         "default": "class_config",
         "enum": [
            "class_config"
         ],
         "type": "string"
      }
   },
   "required": [
      "names"
   ],
   "additionalProperties": false
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *colors (Optional[List[Union[str, Tuple]]])*
- *names (List[str])*
- *null_class (Optional[str])*
- *type_hint (Literal['class_config'])*

Validators

- *validate_colors* » *colors*
- *validate_null_class* » *null_class*

field colors: Optional[List[Union[str, Tuple]]] = None

Colors used to visualize classes. Can be color strings accepted by matplotlib or RGB tuples. If None, a random color will be auto-generated for each class.

Validated by

- *validate_colors*

field names: List[str] [Required]

Names of classes. The i-th class in this list will have class ID = i.

field null_class: Optional[str] = None

Optional name of class in *names* to use as the null class. This is used in semantic segmentation to represent the label for imagery pixels that are NODATA or that are missing a label. If None and the class names include "null", it will automatically be used as the null class. If None, and this Config is part of a SemanticSegmentationConfig, a null class will be added automatically.

Validated by

- *validate_null_class*

field type_hint: Literal['class_config'] = 'class_config'

build()

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

ensure_null_class() → None

Add a null class if one isn't set. This method is idempotent.

Return type
None

get_class_id(*name: str*) → int

Parameters
name (*str*) –

Return type
int

get_color_to_class_id() → dict

Return type
dict

get_name(*id: int*) → str

Parameters
id (*int*) –

Return type
str

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(**args*, ***kwargs*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validator validate_colors » colors

Compare length w/ names. Also auto-generate if not specified.

Parameters

- **v** (*Optional[List[Union[str, Tuple]]]*) –

- **values** (*dict*) –

Return type
Optional[*List*[*Union*[str, *Tuple*]]]

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises
ConfigError – if field is invalid

validator validate_null_class » null_class

Check if in names. If 'null' in names, use it as null class.

Parameters

- **v** (*Optional[str]*) –
- **values** (*dict*) –

Return type
Optional[str]

property color_triples: List[Tuple[float, float, float]]

Class colors in a normalized form.

property null_class_id: int

crs_transformer

Modules

crs_transformer

identity_crs_transformer

rasterio_crs_transformer

crs_transformer

Classes

CRSTransformer	Transforms map points in some CRS into pixel coordinates.

CRSTransformer

class CRSTransformer

Bases: ABC

Transforms map points in some CRS into pixel coordinates.

Each transformer is associated with a particular *RasterSource*.

__init__(*transform: Optional[Any] = None*, *image_crs: Optional[str] = None*, *map_crs: Optional[str] = None*)

Parameters

- **transform** (*Optional[Any]*) –
- **image_crs** (*Optional[str]*) –
- **map_crs** (*Optional[str]*) –

Methods

__init__([transform, image_crs, map_crs])	

map_to_pixel()	Transform input from pixel to map coords.
pixel_to_map()	Transform input from pixel to map coords.

__init__(*transform: Optional[Any] = None*, *image_crs: Optional[str] = None*, *map_crs: Optional[str] = None*)

Parameters

- **transform** (*Optional[Any]*) –
- **image_crs** (*Optional[str]*) –
- **map_crs** (*Optional[str]*) –

map_to_pixel(*inp: Tuple[float, float]*) → Tuple[int, int]

map_to_pixel(*inp: Tuple['np.array', 'np.array']*) → Tuple['np.array', 'np.array']

map_to_pixel(*inp:* Box) → *Box*

map_to_pixel(*inp: BaseGeometry*) → BaseGeometry

Transform input from pixel to map coords.

Parameters

inp – (x, y) tuple or Box or rasterio Window or shapely geometry in pixel coordinates. If tuple, x and y can be single values or array-like.

Returns
Coordinate-transformed input in the same format.

pixel_to_map(*inp: Tuple[float, float]*) → Tuple[float, float]

pixel_to_map(*inp: Tuple['np.array', 'np.array']*) → Tuple['np.array', 'np.array']

pixel_to_map(*inp:* Box) → *Box*

pixel_to_map(*inp: BaseGeometry*) → BaseGeometry

Transform input from pixel to map coords.

Parameters
inp – (x, y) tuple or Box or rasterio Window or shapely geometry in pixel coordinates. If tuple, x and y can be single values or array-like.

Returns
Coordinate-transformed input in the same format.

identity_crs_transformer

Classes

IdentityCRSTransformer	Transformer for when map coordinates are already in pixel coordinates.

IdentityCRSTransformer

class IdentityCRSTransformer

Bases: *CRSTransformer*

Transformer for when map coordinates are already in pixel coordinates.

This is useful for non-georeferenced imagery.

__init__(*transform: Optional[Any] = None*, *image_crs: Optional[str] = None*, *map_crs: Optional[str] = None*)

Parameters

- **transform** (*Optional[Any]*) –
- **image_crs** (*Optional[str]*) –
- **map_crs** (*Optional[str]*) –

Methods

__init__([transform, image_crs, map_crs])

map_to_pixel(inp)	Transform input from pixel to map coords.
pixel_to_map(inp)	Transform input from pixel to map coords.

__init__(*transform: Optional[Any] = None*, *image_crs: Optional[str] = None*, *map_crs: Optional[str] = None*)

Parameters

- **transform** (*Optional[Any]*) –
- **image_crs** (*Optional[str]*) –
- **map_crs** (*Optional[str]*) –

map_to_pixel(*inp*)

Transform input from pixel to map coords.

Parameters
inp – (x, y) tuple or Box or rasterio Window or shapely geometry in pixel coordinates. If tuple, x and y can be single values or array-like.

Returns
Coordinate-transformed input in the same format.

pixel_to_map(*inp*)

Transform input from pixel to map coords.

Parameters
inp – (x, y) tuple or Box or rasterio Window or shapely geometry in pixel coordinates. If tuple, x and y can be single values or array-like.

Returns
Coordinate-transformed input in the same format.

rasterio_crs_transformer

Classes

RasterioCRSTransformer	Transformer for a RasterioRasterSource.

RasterioCRSTransformer

class RasterioCRSTransformer

Bases: *CRSTransformer*

Transformer for a RasterioRasterSource.

__init__(*transform: Affine*, *image_crs: Any*, *map_crs: Any = 'epsg:4326'*, *round_pixels: bool = True*)

Constructor.

Parameters

- **transform** (*Affine*) – Rasterio affine transform.
- **image_crs** (*Any*) – CRS of image in format that PyProj can handle eg. wkt or init string.
- **map_crs** (*Any*) – CRS of the labels. Defaults to "epsg:4326".
- **round_pixels** (*bool*) – If True, round outputs of map_to_pixel and inputs of pixel_to_map to integers. Defaults to False.

Methods

__init__(transform, image_crs[, map_crs, ...])	Constructor.
from_dataset(dataset[, map_crs])	
from_uri(uri[, map_crs])	
map_to_pixel(inp)	Transform input from pixel to map coords.
pixel_to_map(inp)	Transform input from pixel to map coords.

__init__(*transform: Affine*, *image_crs: Any*, *map_crs: Any = 'epsg:4326'*, *round_pixels: bool = True*)

Constructor.

Parameters

- **transform** (*Affine*) – Rasterio affine transform.
- **image_crs** (*Any*) – CRS of image in format that PyProj can handle eg. wkt or init string.
- **map_crs** (*Any*) – CRS of the labels. Defaults to "epsg:4326".
- **round_pixels** (*bool*) – If True, round outputs of map_to_pixel and inputs of pixel_to_map to integers. Defaults to False.

classmethod from_dataset(*dataset*, *map_crs: Optional[str] = 'epsg:4326'*, ***kwargs*) → *RasterioCRSTransformer*

Parameters

map_crs (*Optional[str]*) –

Return type

RasterioCRSTransformer

classmethod from_uri(*uri: str*, *map_crs: Optional[str] = 'epsg:4326'*, ***kwargs*) → *RasterioCRSTransformer*

Parameters

- **uri** (*str*) –
- **map_crs** (*Optional[str]*) –

Return type

RasterioCRSTransformer

map_to_pixel(*inp*)

Transform input from pixel to map coords.

Parameters

inp – (x, y) tuple or Box or rasterio Window or shapely geometry in pixel coordinates. If tuple, x and y can be single values or array-like.

Returns

Coordinate-transformed input in the same format.

pixel_to_map(*inp*)

Transform input from pixel to map coords.

Parameters

inp – (x, y) tuple or Box or rasterio Window or shapely geometry in pixel coordinates. If tuple, x and y can be single values or array-like.

Returns
Coordinate-transformed input in the same format.

dataset_config

Configs

DatasetConfig	Configure train, validation, and test splits for a dataset.

DatasetConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model DatasetConfig

Configure train, validation, and test splits for a dataset.


```
{
   "title": "DatasetConfig",
   "description": "Configure train, validation, and test splits for a dataset.",
   "type": "object",
   "properties": {
      "class_config": {
         "$ref": "#/definitions/ClassConfig"
      },
      "train_scenes": {
         "title": "Train Scenes",
         "type": "array",
         "items": {
            "$ref": "#/definitions/SceneConfig"
         }
      },
      "validation_scenes": {
         "title": "Validation Scenes",
         "type": "array",
         "items": {
            "$ref": "#/definitions/SceneConfig"
         }
      },
      "test_scenes": {
         "title": "Test Scenes",
         "default": [],
         "type": "array",
         "items": {
            "$ref": "#/definitions/SceneConfig"
         }
      },
      "scene_groups": {
         "title": "Scene Groups",
         "description": "Groupings of scenes. Should be a dict of the form: {<group-
```


(continues on next page)

(continued from previous page)

```
↪name>: Set(scene_id_1, scene_id_2, ...)}. Three groups are added by default: \
↪"train_scenes\", \"validation_scenes\", and \"test_scenes\"",
         "default": {},
         "type": "object",
         "additionalProperties": {
            "type": "array",
            "items": {
               "type": "string"
            },
            "uniqueItems": true
         }
      },
      "type_hint": {
         "title": "Type Hint",
         "default": "dataset",
         "enum": [
            "dataset"
         ],
         "type": "string"
      }
   },
   "required": [
      "class_config",
      "train_scenes",
      "validation_scenes"
   ],
   "additionalProperties": false,
   "definitions": {
      "ClassConfig": {
         "title": "ClassConfig",
         "description": "Configure class information for a machine learning task.",
         "type": "object",
         "properties": {
            "names": {
               "title": "Names",
               "description": "Names of classes. The i-th class in this list will␣
↪have class ID = i.",
               "type": "array",
               "items": {
                  "type": "string"
               }
            },
            "colors": {
               "title": "Colors",
               "description": "Colors used to visualize classes. Can be color␣
↪strings accepted by matplotlib or RGB tuples. If None, a random color will be␣
↪auto-generated for each class.",
               "type": "array",
               "items": {
                  "anyOf": [
                     {
                        "type": "string"
```

(continues on next page)

(continued from previous page)


```
                    },
                    {
                        "type": "array",
                        "items": {}
                    }
                ]
            }
        },
        "null_class": {
            "title": "Null Class",
            "description": "Optional name of class in `names` to use as the null␣
↪class. This is used in semantic segmentation to represent the label for imagery␣
↪pixels that are NODATA or that are missing a label. If None and the class names␣
↪include \"null\", it will automatically be used as the null class. If None, and␣
↪this Config is part of a SemanticSegmentationConfig, a null class will be added␣
↪automatically.",
            "type": "string"
        },
        "type_hint": {
            "title": "Type Hint",
            "default": "class_config",
            "enum": [
                "class_config"
            ],
            "type": "string"
        }
    },
    "required": [
        "names"
    ],
    "additionalProperties": false
},
"RasterTransformerConfig": {
    "title": "RasterTransformerConfig",
    "description": "Configure a :class:`.RasterTransformer`.",
    "type": "object",
    "properties": {
        "type_hint": {
            "title": "Type Hint",
            "default": "raster_transformer",
            "enum": [
                "raster_transformer"
            ],
            "type": "string"
        }
    },
    "additionalProperties": false
},
"RasterSourceConfig": {
    "title": "RasterSourceConfig",
    "description": "Configure a :class:`.RasterSource`.",
    "type": "object",
```


(continues on next page)

(continued from previous page)

```
        "properties": {
           "channel_order": {
              "title": "Channel Order",
              "description": "The sequence of channel indices to use when reading␣
↪imagery.",
              "type": "array",
              "items": {
                 "type": "integer"
              }
           },
           "transformers": {
              "title": "Transformers",
              "default": [],
              "type": "array",
              "items": {
                 "$ref": "#/definitions/RasterTransformerConfig"
              }
           },
           "extent": {
              "title": "Extent",
              "description": "Use-specified extent in pixel coords in the form␣
↪(ymin, xmin, ymax, xmax). Useful for cropping the raster source so that only part␣
↪of the raster is read from.",
              "type": "array",
              "minItems": 4,
              "maxItems": 4,
              "items": [
                 {
                    "type": "integer"
                 },
                 {
                    "type": "integer"
                 },
                 {
                    "type": "integer"
                 },
                 {
                    "type": "integer"
                 }
              ]
           },
           "type_hint": {
              "title": "Type Hint",
              "default": "raster_source",
              "enum": [
                 "raster_source"
              ],
              "type": "string"
           }
        },
        "additionalProperties": false
     },
```

(continues on next page)

(continued from previous page)

```
      "LabelSourceConfig": {
         "title": "LabelSourceConfig",
         "description": "Configure a :class:`.LabelSource`.",
         "type": "object",
         "properties": {
            "type_hint": {
               "title": "Type Hint",
               "default": "label_source",
               "enum": [
                  "label_source"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      },
      "LabelStoreConfig": {
         "title": "LabelStoreConfig",
         "description": "Configure a :class:`.LabelStore`.",
         "type": "object",
         "properties": {
            "type_hint": {
               "title": "Type Hint",
               "default": "label_store",
               "enum": [
                  "label_store"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      },
      "SceneConfig": {
         "title": "SceneConfig",
         "description": "Configure a :class:`.Scene` comprising raster data &␣
↪labels for an AOI.\n    ",
         "type": "object",
         "properties": {
            "id": {
               "title": "Id",
               "type": "string"
            },
            "raster_source": {
               "$ref": "#/definitions/RasterSourceConfig"
            },
            "label_source": {
               "$ref": "#/definitions/LabelSourceConfig"
            },
            "label_store": {
               "$ref": "#/definitions/LabelStoreConfig"
            },
            "aoi_uris": {
```

(continues on next page)

(continued from previous page)

```
                "title": "Aoi Uris",
                "description": "List of URIs of GeoJSON files that define the AOIs␣
↪for the scene. Each polygon defines an AOI which is a piece of the scene that is␣
↪assumed to be fully labeled and usable for training or validation. The AOIs are␣
↪assumed to be in EPSG:4326 coordinates.",
                "type": "array",
                "items": {
                   "type": "string"
                }
             },
             "type_hint": {
                "title": "Type Hint",
                "default": "scene",
                "enum": [
                   "scene"
                ],
                "type": "string"
             }
          },
          "required": [
             "id",
             "raster_source"
          ],
          "additionalProperties": false
       }
    }
}
```

Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *class_config (rastervision.core.data.class_config.ClassConfig)*
- *scene_groups (Dict[str, Set[str]])*
- *test_scenes (List[rastervision.core.data.scene_config.SceneConfig])*
- *train_scenes (List[rastervision.core.data.scene_config.SceneConfig])*
- *type_hint (Literal['dataset'])*
- *validation_scenes (List[rastervision.core.data.scene_config.SceneConfig])*

field class_config: *ClassConfig* [Required]

field scene_groups: Dict[str, Set[str]] = {}

Groupings of scenes. Should be a dict of the form: {<group-name>: Set(scene_id_1, scene_id_2, ...)}. Three groups are added by default: "train_scenes", "validation_scenes", and "test_scenes"

field test_scenes: List[*SceneConfig*] = []

field train_scenes: List[*SceneConfig*] [Required]

field type_hint: Literal['dataset'] = 'dataset'

field validation_scenes: List[*SceneConfig*] [Required]

build()

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

get_split_config(*split_ind*, *num_splits*)

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*pipeline=None*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

property all_scenes: List[*SceneConfig*]

label

Modules

chip_classification_labels	
labels	Defines the abstract Labels class.
object_detection_labels	
semantic_segmentation_labels	
tfod_utils	
utils	

chip_classification_labels

Classes

ChipClassificationLabels	Represents a spatial grid of cells associated with classes.
ClassificationLabel	ClassificationLabel(class_id: int, scores: Optional[Sequence[float]] = None)

ChipClassificationLabels

class ChipClassificationLabels

Bases: *Labels*

Represents a spatial grid of cells associated with classes.

__init__(*cell_to_label: Optional[Dict[*Box*, Tuple[int, Optional[Sequence[float]]]]] = None*)

Parameters

cell_to_label (*Optional[Dict[*Box*, Tuple[int, Optional[Sequence[float]]]]]*) –

Methods

__init__([cell_to_label])	

extend(labels)	Adds cells contained in labels.
filter_by_aoi(aoi_polygons)	Returns a copy of these labels filtered by a given set of AOI polygons
from_predictions(windows, predictions)	Overrid to convert predictions to (class_id, scores) pairs.
get_cell_class_id(cell)	Return class_id for a cell.
get_cell_scores(cell)	Return scores for a cell.
get_cells()	Return list of all cells (list of Box).
get_class_ids()	Return list of class_ids for all cells.
get_scores()	Return list of scores for all cells.
get_singleton_labels(cell)	Return Labels object representing a single cell.
get_values()	Return list of class_ids and scores for all cells.
make_empty()	Instantiate an empty instance of this class.
save(uri, class_config, crs_transformer)	Save labels as a GeoJSON file.
set_cell(cell, class_id[, scores])	Set cell and its class_id.

__init__(*cell_to_label: Optional[Dict[*Box*, Tuple[int, Optional[Sequence[float]]]]] = None*)

Parameters
cell_to_label (*Optional[Dict[*Box*, Tuple[int, Optional[Sequence[float]]]]]*) –

extend(*labels:* ChipClassificationLabels) → None

Adds cells contained in labels.

Parameters
labels (ChipClassificationLabels) – ChipClassificationLabels

Return type
None

filter_by_aoi(*aoi_polygons: Iterable[Polygon]*)

Returns a copy of these labels filtered by a given set of AOI polygons

Parameters

- **by** (*aoi_polygons - A list of AOI polygons to filter*) –
- **coordinates.** (*in pixel*) –
- **aoi_polygons** (*Iterable[Polygon]*) –

classmethod from_predictions(*windows: Iterable[*Box*], predictions: Iterable[Any]*) → *Labels*

Overrid to convert predictions to (class_id, scores) pairs.

Parameters

- **windows** (*Iterable[*Box*]*) –
- **predictions** (*Iterable[Any]*) –

Return type
Labels

get_cell_class_id(*cell:* Box) → int

Return class_id for a cell.

Parameters
cell (Box) – (Box)

Return type
int

get_cell_scores(*cell:* Box) → Optional[Sequence[float]]

Return scores for a cell.

Parameters
cell (Box) – (Box)

Return type
Optional[*Sequence*[float]]

get_cells() → List[*Box*]

Return list of all cells (list of Box).

Return type
List[Box]

get_class_ids() → List[int]

Return list of class_ids for all cells.

Return type
List[int]

get_scores() → List[Optional[Sequence[float]]]

Return list of scores for all cells.

Return type
List[*Optional*[*Sequence*[float]]]

get_singleton_labels(*cell:* Box)

Return Labels object representing a single cell.

Parameters
cell (Box) – (Box)

get_values() → List[*ClassificationLabel*]

Return list of class_ids and scores for all cells.

Return type
List[ClassificationLabel]

classmethod make_empty() → *ChipClassificationLabels*

Instantiate an empty instance of this class.

Returns

An object of the Label subclass on which this method is called.

Return type
Labels

save(*uri: str*, *class_config:* ClassConfig, *crs_transformer:* CRSTransformer) → None

Save labels as a GeoJSON file.

Parameters

- **uri** (*str*) – URI of the output file.
- **class_config** (ClassConfig) – ClassConfig to map class IDs to names.
- **crs_transformer** (CRSTransformer) – CRSTransformer to convert from pixel-coords to map-coords before saving.

Return type
None

set_cell(*cell: Box*, *class_id: int*, *scores: Optional[ndarray] = None*) → None

Set cell and its class_id.

Parameters

- **cell** (Box) – (Box)
- **class_id** (*int*) – int
- **scores** (*Optional[ndarray]*) – 1d numpy array of probabilities for each class

Return type
None

ClassificationLabel

class ClassificationLabel

Bases: object

ClassificationLabel(class_id: int, scores: Optional[Sequence[float]] = None)

Attributes

scores

class_id

__init__(*class_id: int*, *scores: Optional[Sequence[float]] = None*) → None

Parameters

- **class_id** (*int*) –
- **scores** (*Optional[Sequence[float]]*) –

Return type
None

Methods

__init__(class_id[, scores])	

__init__(*class_id: int*, *scores: Optional[Sequence[float]] = None*) → None

Parameters

- **class_id** (*int*) –
- **scores** (*Optional[Sequence[float]]*) –

Return type
None

class_id: int

scores: Optional[Sequence[float]] = None

labels

Defines the abstract Labels class.

Classes

Labels	A source-agnostic, in-memory representation of labels in a scene.

Labels

class Labels

Bases: ABC

A source-agnostic, in-memory representation of labels in a scene.

This class can represent labels obtained via a `LabelSource`, a `LabelStore`, or directly from model predictions.

__init__()

Methods

__init__()	
filter_by_aoi(aoi_polygons)	Returns a copy of these labels filtered by a given set of AOI polygons
from_predictions(windows, predictions)	Instantiate from windows and their corresponding predictions.
make_empty()	Instantiate an empty instance of this class.
save(uri)	Save to file.

abstract filter_by_aoi(*aoi_polygons*)

Returns a copy of these labels filtered by a given set of AOI polygons

Parameters

- **by** (*aoi_polygons - A list of AOI polygons to filter*) –
- **coordinates.** (*in pixel*) –

classmethod from_predictions(*windows: Iterable[Box]*, *predictions: Iterable[Any]*) → *Labels*

Instantiate from windows and their corresponding predictions.

This makes no assumptions about the type or format of the predictions. Subclasses should implement the __setitem__ method to correctly handle the predictions.

Parameters

- **windows** (*Iterable[Box]*) – Boxes in pixel coords, specifying chips in the raster.
- **predictions** (*Iterable[Any]*) – The model predictions for each chip specified by the windows.

Returns

An object of the Label subclass on which this method is called.

Return type
Labels

abstract classmethod make_empty() → *Labels*

Instantiate an empty instance of this class.

Returns

An object of the Label subclass on which this method is called.

Return type
Labels

abstract save(*uri: str*) → None

Save to file.

Parameters
uri (*str*) –

Return type
None

object_detection_labels

Classes

ObjectDetectionLabels	A set of boxes and associated class_ids and scores.

ObjectDetectionLabels

class ObjectDetectionLabels

Bases: *Labels*

A set of boxes and associated class_ids and scores.

Implemented using the Tensorflow Object Detection API's BoxList class.

__init__(*npboxes: array*, *class_ids: array*, *scores: Optional[array] = None*)

Construct a set of object detection labels.

Parameters

- **npboxes** (*array*) – float numpy array of size nx4 with cols ymin, xmin, ymax, xmax. Should be in pixel coordinates within the global frame of reference.
- **class_ids** (*array*) – int numpy array of size n with class ids
- **scores** (*Optional[array]*) – float numpy array of size n

Methods

__init__(npboxes, class_ids[, scores])	Construct a set of object detection labels.
assert_equal(expected_labels)	
concatenate(labels1, labels2)	Return concatenation of labels.
filter_by_aoi(aoi_polygons)	Returns a copy of these labels filtered by a given set of AOI polygons
from_boxlist(boxlist)	Make ObjectDetectionLabels from BoxList object.
from_geojson(geojson[, extent])	Convert GeoJSON to ObjectDetectionLabels object.
from_predictions(windows, predictions)	Instantiate from windows and their corresponding predictions.
get_boxes()	Return list of Boxes.
get_class_ids()	
get_npboxes()	
get_overlapping(labels, window[, ...])	Return subset of labels that overlap with window.
get_scores()	
global_to_local(npboxes, window)	Convert from global to local coordinates.
local_to_global(npboxes, window)	Convert from local to global coordinates.
local_to_normalized(npboxes, window)	Convert from local to normalized coordinates.
make_empty()	Instantiate an empty instance of this class.
normalized_to_local(npboxes, window)	Convert from normalized to local coordinates.
prune_duplicates(labels, score_thresh, ...)	Remove duplicate boxes.
save(uri, class_config, crs_transformer)	Save labels as a GeoJSON file.
to_boxlist()	
to_dict()	Returns a dict version of these labels.

__init__(*npboxes: array*, *class_ids: array*, *scores: Optional[array] = None*)

Construct a set of object detection labels.

Parameters

- **npboxes** (*array*) – float numpy array of size nx4 with cols ymin, xmin, ymax, xmax. Should be in pixel coordinates within the global frame of reference.
- **class_ids** (*array*) – int numpy array of size n with class ids
- **scores** (*Optional[array]*) – float numpy array of size n

assert_equal(*expected_labels:* ObjectDetectionLabels)

Parameters
expected_labels (ObjectDetectionLabels) –

static concatenate(*labels1:* ObjectDetectionLabels, *labels2:* ObjectDetectionLabels) → *ObjectDetectionLabels*

Return concatenation of labels.

Parameters

- **labels1** (ObjectDetectionLabels) – ObjectDetectionLabels
- **labels2** (ObjectDetectionLabels) – ObjectDetectionLabels

Return type
ObjectDetectionLabels

filter_by_aoi(*aoi_polygons: Iterable[Polygon]*)

Returns a copy of these labels filtered by a given set of AOI polygons

Parameters

- **by** (*aoi_polygons - A list of AOI polygons to filter*) –
- **coordinates.** (*in pixel*) –
- **aoi_polygons** (*Iterable[Polygon]*) –

static from_boxlist(*boxlist:* BoxList)

Make ObjectDetectionLabels from BoxList object.

Parameters
boxlist (BoxList) –

static from_geojson(*geojson: dict*, *extent: Optional[*Box*] = None*) → *ObjectDetectionLabels*

Convert GeoJSON to ObjectDetectionLabels object.

If extent is provided, filter out the boxes that lie "more than a little bit" outside the extent.

Parameters

- **geojson** (*dict*) – (dict) normalized GeoJSON (see VectorSource)
- **extent** (*Optional[Box]*) – (Box) in pixel coords

Returns
ObjectDetectionLabels

Return type
ObjectDetectionLabels

classmethod from_predictions(*windows: Iterable[*Box*]*, *predictions: Iterable[Any]*) → *Labels*

Instantiate from windows and their corresponding predictions.

This makes no assumptions about the type or format of the predictions. Subclasses should implement the __setitem__ method to correctly handle the predictions.

Parameters

- **windows** (*Iterable[Box]*) – Boxes in pixel coords, specifying chips in the raster.
- **predictions** (*Iterable[Any]*) – The model predictions for each chip specified by the windows.

Returns

An object of the Label subclass on which this method is called.

Return type
Labels

get_boxes() → List[*Box*]

Return list of Boxes.

Return type
List[Box]

get_class_ids() → ndarray

Return type
ndarray

get_npboxes() → ndarray

Return type
ndarray

static get_overlapping(*labels:* ObjectDetectionLabels, *window:* Box, *ioa_thresh: float = 1e-06*, *clip: bool = False*) → *ObjectDetectionLabels*

Return subset of labels that overlap with window.

Parameters

- **labels** (ObjectDetectionLabels) – ObjectDetectionLabels
- **window** (Box) – Box
- **ioa_thresh** (*float*) – the minimum IOA for a box to be considered as overlapping
- **clip** (*bool*) – if True, clip label boxes to the window

Return type
ObjectDetectionLabels

get_scores() → ndarray

Return type
ndarray

static global_to_local(*npboxes: ndarray*, *window:* Box)

Convert from global to local coordinates.

The global coordinates are row/col within the extent of a RasterSource. The local coordinates are row/col within the window frame of reference.

Parameters

- **npboxes** (*ndarray*) –
- **window** (Box) –

static local_to_global(*npboxes: ndarray*, *window:* Box)

Convert from local to global coordinates.

The local coordinates are row/col within the window frame of reference. The global coordinates are row/col within the extent of a RasterSource.

Parameters

- **npboxes** (*ndarray*) –
- **window** (Box) –

static local_to_normalized(*npboxes: ndarray*, *window:* Box)

Convert from local to normalized coordinates.

The local coordinates are row/col within the window frame of reference. Normalized coordinates range from 0 to 1 on each (height/width) axis.

Parameters

- **npboxes** (*ndarray*) –
- **window** (Box) –

classmethod make_empty() → *ObjectDetectionLabels*

Instantiate an empty instance of this class.

Returns

An object of the Label subclass on which this method is called.

Return type
Labels

static normalized_to_local(*npboxes: ndarray*, *window:* Box)

Convert from normalized to local coordinates.

Normalized coordinates range from 0 to 1 on each (height/width) axis. The local coordinates are row/col within the window frame of reference.

Parameters

- **npboxes** (*ndarray*) –
- **window** (Box) –

static prune_duplicates(*labels:* ObjectDetectionLabels, *score_thresh: float*, *merge_thresh: float*) → *ObjectDetectionLabels*

Remove duplicate boxes.

Runs non-maximum suppression to remove duplicate boxes that result from sliding window prediction algorithm.

Parameters

- **labels** (ObjectDetectionLabels) – ObjectDetectionLabels
- **score_thresh** (*float*) – the minimum allowed score of boxes
- **merge_thresh** (*float*) – the minimum IOA allowed when merging two boxes together

Returns
ObjectDetectionLabels

Return type
ObjectDetectionLabels

save(*uri: str*, *class_config: ClassConfig*, *crs_transformer: CRSTransformer*) → None

Save labels as a GeoJSON file.

Parameters

- **uri** (*str*) – URI of the output file.
- **class_config** (*ClassConfig*) – ClassConfig to map class IDs to names.
- **crs_transformer** (*CRSTransformer*) – CRSTransformer to convert from pixel-coords to map-coords before saving.

Return type
None

to_boxlist() → *BoxList*

Return type
BoxList

to_dict() → dict

Returns a dict version of these labels.

The Dict has a Box as a key, and a tuple of (class_id, score) as the values.

Return type
dict

semantic_segmentation_labels

Classes

SemanticSegmentationDiscreteLabels	Vote-counts for each pixel belonging to each class.
SemanticSegmentationLabels	Representation of Semantic Segmentation labels.
SemanticSegmentationSmoothLabels	Membership-scores for each pixel for each class.

SemanticSegmentationDiscreteLabels

class SemanticSegmentationDiscreteLabels

Bases: *SemanticSegmentationLabels*

Vote-counts for each pixel belonging to each class.

Maintains a num_classes x H x W array where value_{ijk} represents how many times pixel_{jk} has been classified as class i. A label array can be obtained from this by argmax'ing along the class dimension. Can also be turned into a score converting counts to probabilities.

__init__(*extent: ~rastervision.core.box.Box*, *num_classes: int*, *dtype: ~typing.Any = <class 'numpy.uint8'>*)

Constructor.

Parameters

- **extent** (*Box*) – The extent of the region to which the labels belong, in global coordinates.
- **num_classes** (*int*) – Number of classes.

- **dtype** (*Any*) – dtype of the counts array. Defaults to np.uint8.

Methods

__init__(extent, num_classes[, dtype])	Constructor.
add_predictions(windows, predictions[, crop_sz])	Populate predictions.
add_window(window, pixel_class_ids)	Set labels for the given window.
filter_by_aoi(aoi_polygons, null_class_id, ...)	Keep only the values that lie inside the AOI.
from_predictions(windows, predictions, ...)	Instantiate from windows and their corresponding predictions.
get_label_arr(window[, null_class_id])	Get labels as array of class IDs.
get_score_arr(window)	Get array of pixel scores.
get_windows(**kwargs)	Generate sliding windows over the local extent.
make_empty(extent, num_classes)	Instantiate an empty instance.
mask_fill(window, mask, fill_value)	Set fill_value'th class ID's count to 1 and all others to zero.
save(uri, crs_transformer, class_config[, ...])	Save labels as a raster and/or vectors.

__init__(*extent: ~rastervision.core.box.Box*, *num_classes: int*, *dtype: ~typing.Any = <class 'numpy.uint8'>*)

Constructor.

Parameters

- **extent** (Box) – The extent of the region to which the labels belong, in global coordinates.
- **num_classes** (*int*) – Number of classes.
- **dtype** (*Any*) – dtype of the counts array. Defaults to np.uint8.

add_predictions(*windows: Iterable[Box]*, *predictions: Iterable[Any]*, *crop_sz: Optional[int] = None*) → None

Populate predictions.

Parameters

- **windows** (*Iterable[Box]*) – Boxes in pixel coords, specifying chips in the raster.
- **predictions** (*Iterable[Any]*) – The model predictions for each chip specified by the windows.
- **crop_sz** (*Optional[int]*) – Number of rows/columns of pixels from the edge of prediction windows to discard. This is useful because predictions near edges tend to be lower quality and can result in very visible artifacts near the edges of chips. This should only be used if the given windows represent a sliding-window grid over the scene extent with overlap between adjacent windows. Defaults to None.

Return type
None

add_window(*window: Box*, *pixel_class_ids: ndarray*) → None

Set labels for the given window.

Parameters

- **window** (Box) –
- **pixel_class_ids** (*ndarray*) –

Return type
None

filter_by_aoi(*aoi_polygons: list*, *null_class_id: int*, ***kwargs*) → *SemanticSegmentationLabels*

Keep only the values that lie inside the AOI.

Parameters

- **aoi_polygons** (*list*) –
- **null_class_id** (*int*) –

Return type
SemanticSegmentationLabels

classmethod from_predictions(*windows: Iterable[Box]*, *predictions: Iterable[Any]*, *extent: Box*, *num_classes: int*, *crop_sz: Optional[int] = None*) → Union[*SemanticSegmentationDiscreteLabels*, *SemanticSegmentationSmoothLabels*]

Instantiate from windows and their corresponding predictions.

Parameters

- **windows** (*Iterable[Box]*) – Boxes in pixel coords, specifying chips in the raster.
- **predictions** (*Iterable[Any]*) – The model predictions for each chip specified by the windows.
- **extent** (Box) – The extent of the region to which the labels belong, in global coordinates.
- **num_classes** (*int*) – Number of classes.
- **crop_sz** (*Optional[int]*) – Number of rows/columns of pixels from the edge of prediction windows to discard. This is useful because predictions near edges tend to be lower quality and can result in very visible artifacts near the edges of chips. This should only be used if the given windows represent a sliding-window grid over the scene extent with overlap between adjacent windows. Defaults to None.

Returns

Union[SemanticSegmentationDiscreteLabels, SemanticSegmentationSmoothLabels]: If smooth=True, returns a

SemanticSegmentationSmoothLabels. Otherwise, a SemanticSegmentationDiscreteLabels.

Return type
Union[SemanticSegmentationDiscreteLabels, SemanticSegmentationSmoothLabels]

get_label_arr(*window: Box*, *null_class_id: int = -1*) → ndarray

Get labels as array of class IDs.

Returns null_class_id for pixels for which there is no data.

Parameters

- **window** (Box) –
- **null_class_id** (*int*) –

Return type
ndarray

get_score_arr(*window:* Box) → ndarray

Get array of pixel scores.

Parameters
window (Box) –

Return type
ndarray

get_windows(***kwargs*) → List[*Box*]

Generate sliding windows over the local extent. The keyword args are passed to Box.get_windows() and can therefore be used to control the specifications of the windows.

If the keyword args do not contain size, a list of length 1, containing the full extent is returned.

Return type
List[Box]

classmethod make_empty(*extent:* Box, *num_classes: int*) → *SemanticSegmentationDiscreteLabels*

Instantiate an empty instance.

Parameters

- **extent** (Box) –
- **num_classes** (*int*) –

Return type
SemanticSegmentationDiscreteLabels

mask_fill(*window:* Box, *mask: ndarray*, *fill_value: Any*) → None

Set fill_value'th class ID's count to 1 and all others to zero.

Parameters

- **window** (Box) –
- **mask** (*ndarray*) –
- **fill_value** (*Any*) –

Return type
None

save(*uri: str*, *crs_transformer:* CRSTransformer, *class_config:* ClassConfig, *tmp_dir: Optional[str] = None*, *save_as_rgb: bool = False*, *raster_output: bool = True*, *rasterio_block_size: int = 512*, *vector_outputs: Optional[Sequence[*VectorOutputConfig*]] = None*, *profile_overrides: Optional[dict] = None*) → None

Save labels as a raster and/or vectors.

If URI is remote, all files will be first written locally and then uploaded to the URI.

Parameters

- **uri** (*str*) – URI of directory in which to save all output files.
- **crs_transformer** (CRSTransformer) – CRSTransformer to configure CRS and affine transform of the output GeoTiff.
- **class_config** (ClassConfig) – The ClassConfig.
- **tmp_dir** (*Optional[str], optional*) – Temporary directory to use. If None, will be auto-generated. Defaults to None.
- **save_as_rgb** (*bool, optional*) – If True, Saves labels as an RGB image, using the class-color mapping in the class_config. Defaults to False.

- **raster_output** (*bool, optional*) – If True, saves labels as a raster of class IDs (one band). Defaults to True.
- **rasterio_block_size** (*int, optional*) – Value to set blockxsize and blockysize to. Defaults to 512.
- **vector_outputs** (*Optional[Sequence[VectorOutputConfig]], optional*) – List of VectorOutputConfig's containing vectorization configuration information. Only classes for which a VectorOutputConfig is specified will be saved as vectors. If None, no vector outputs will be produced. Defaults to None.
- **profile_overrides** (*Optional[dict], optional*) – This can be used to arbitrarily override properties in the profile used to create the output GeoTiff. Defaults to None.

Return type
None

SemanticSegmentationLabels

class SemanticSegmentationLabels

Bases: *Labels*

Representation of Semantic Segmentation labels.

__init__(*extent: Box, num_classes: int, dtype: dtype*)

Constructor.

Parameters

- **extent** (Box) – The extent of the region to which the labels belong, in global coordinates.
- **num_classes** (*int*) – Number of classes.
- **dtype** (*dtype*) –

Methods

__init__(extent, num_classes, dtype)	Constructor.
add_predictions(windows, predictions[, crop_sz])	Populate predictions.
add_window(window, values)	Set labels for the given window.
filter_by_aoi(aoi_polygons, null_class_id, ...)	Keep only the values that lie inside the AOI.
from_predictions(windows, predictions, ...)	Instantiate from windows and their corresponding predictions.
get_label_arr(window[, null_class_id])	Get labels as array of class IDs.
get_windows(**kwargs)	Generate sliding windows over the local extent.
make_empty(extent, num_classes[, smooth])	Instantiate an empty instance.
mask_fill(window, mask, fill_value)	Given a window and a binary mask, set all the pixels in the window for which the mask is ON to the fill_value.
save(uri)	Save to file.

__init__(*extent: Box, num_classes: int, dtype: dtype*)

Constructor.

Parameters

- **extent** (Box) – The extent of the region to which the labels belong, in global coordinates.
- **num_classes** (*int*) – Number of classes.
- **dtype** (*dtype*) –

add_predictions(*windows: Iterable[Box]*, *predictions: Iterable[Any]*, *crop_sz: Optional[int] = None*) → None

Populate predictions.

Parameters

- **windows** (*Iterable[Box]*) – Boxes in pixel coords, specifying chips in the raster.
- **predictions** (*Iterable[Any]*) – The model predictions for each chip specified by the windows.
- **crop_sz** (*Optional[int]*) – Number of rows/columns of pixels from the edge of prediction windows to discard. This is useful because predictions near edges tend to be lower quality and can result in very visible artifacts near the edges of chips. This should only be used if the given windows represent a sliding-window grid over the scene extent with overlap between adjacent windows. Defaults to None.

Return type
None

abstract add_window(*window: Box*, *values: ndarray*) → List[*Box*]

Set labels for the given window.

Parameters

- **window** (Box) –
- **values** (*ndarray*) –

Return type
List[Box]

filter_by_aoi(*aoi_polygons: list*, *null_class_id: int*, ***kwargs*) → *SemanticSegmentationLabels*

Keep only the values that lie inside the AOI.

Parameters

- **aoi_polygons** (*list*) –
- **null_class_id** (*int*) –

Return type
SemanticSegmentationLabels

classmethod from_predictions(*windows: Iterable[Box]*, *predictions: Iterable[Any]*, *extent: Box*, *num_classes: int*, *smooth: bool = False*, *crop_sz: Optional[int] = None*) → Union[*SemanticSegmentationDiscreteLabels*, *SemanticSegmentationSmoothLabels*]

Instantiate from windows and their corresponding predictions.

Parameters

- **windows** (*Iterable[Box]*) – Boxes in pixel coords, specifying chips in the raster.
- **predictions** (*Iterable[Any]*) – The model predictions for each chip specified by the windows.
- **extent** (Box) – The extent of the region to which the labels belong, in global coordinates.

- **num_classes** (*int*) – Number of classes.
- **crop_sz** (*Optional[int]*) – Number of rows/columns of pixels from the edge of prediction windows to discard. This is useful because predictions near edges tend to be lower quality and can result in very visible artifacts near the edges of chips. This should only be used if the given windows represent a sliding-window grid over the scene extent with overlap between adjacent windows. Defaults to None.
- **smooth** (*bool*) –

Returns

Union[SemanticSegmentationDiscreteLabels, SemanticSegmentationSmoothLabels]: If smooth=True, returns a

SemanticSegmentationSmoothLabels. Otherwise, a SemanticSegmentationDiscreteLabels.

Return type
Union[SemanticSegmentationDiscreteLabels, SemanticSegmentationSmoothLabels]

abstract get_label_arr(*window:* Box, *null_class_id:* *int* = *-1*) → ndarray

Get labels as array of class IDs.

Note: The returned array is not guaranteed to be the same size as the input window.

Parameters

- **window** (Box) –
- **null_class_id** (*int*) –

Return type
ndarray

get_windows(***kwargs*) → List[*Box*]

Generate sliding windows over the local extent. The keyword args are passed to Box.get_windows() and can therefore be used to control the specifications of the windows.

If the keyword args do not contain size, a list of length 1, containing the full extent is returned.

Return type
List[Box]

classmethod make_empty(*extent:* Box, *num_classes:* *int*, *smooth:* *bool* = *False*) → Union[*SemanticSegmentationDiscreteLabels*, *SemanticSegmentationSmoothLabels*]

Instantiate an empty instance.

Parameters

- **extent** (Box) – The extent of the region to which the labels belong, in global coordinates.
- **num_classes** (*int*) – Number of classes.
- **smooth** (*bool, optional*) – If True, creates a SemanticSegmentationSmoothLabels object. If False, creates a SemanticSegmentationDiscreteLabels object. Defaults to False.

Returns

Union[SemanticSegmentationDiscreteLabels, SemanticSegmentationSmoothLabels]: If smooth=True, returns a

SemanticSegmentationSmoothLabels. Otherwise, a SemanticSegmentationDiscreteLabels.

Raises
ValueError – if num_classes and extent are not specified, but smooth=True.

Return type
Union[SemanticSegmentationDiscreteLabels, SemanticSegmentationSmoothLabels]

abstract mask_fill(*window:* Box, *mask:* *ndarray*, *fill_value:* *Any*) → None

Given a window and a binary mask, set all the pixels in the window for which the mask is ON to the fill_value.

Parameters
- **window** (Box) –
- **mask** (*ndarray*) –
- **fill_value** (*Any*) –

Return type
None

abstract save(*uri:* *str*) → None

Save to file.

Parameters
uri (*str*) –

Return type
None

SemanticSegmentationSmoothLabels

class SemanticSegmentationSmoothLabels

Bases: *SemanticSegmentationLabels*

Membership-scores for each pixel for each class.

Maintains a num_classes x H x W array where value_{ijk} represents the probability (or some other measure) of pixel_{jk} belonging to class i. A discrete label array can be obtained from this by argmax'ing along the class dimension.

__init__(*extent: ~rastervision.core.box.Box*, *num_classes: int*, *dtype: ~typing.Any = <class 'numpy.float16'>*, *dtype_hits: ~typing.Any = <class 'numpy.uint8'>*)

Constructor.

Parameters
- **extent** (Box) – The extent of the region to which the labels belong, in global coordinates.
- **num_classes** (*int*) – Number of classes.
- **dtype** (*Any*) – dtype of the scores array. Defaults to np.float16.
- **dtype_hits** (*Any*) – dtype of the hits array. Defaults to np.uint8.

Methods

__init__(extent, num_classes[, dtype, ...])	Constructor.
add_predictions(windows, predictions[, crop_sz])	Populate predictions.
add_window(window, pixel_class_scores)	Set labels for the given window.
filter_by_aoi(aoi_polygons, null_class_id, ...)	Keep only the values that lie inside the AOI.
from_predictions(windows, predictions, ...)	Instantiate from windows and their corresponding predictions.
get_label_arr(window[, null_class_id])	Get labels as array of class IDs.
get_score_arr(window)	Get array of pixel scores.
get_windows(**kwargs)	Generate sliding windows over the local extent.
make_empty(extent, num_classes)	Instantiate an empty instance.
mask_fill(window, mask, fill_value)	Set fill_value'th class ID's score to 1 and all others to zero.
save(uri, crs_transformer, class_config[, ...])	Save labels as rasters and/or vectors.

__init__(*extent: ~rastervision.core.box.Box*, *num_classes: int*, *dtype: ~typing.Any = <class 'numpy.float16'>*, *dtype_hits: ~typing.Any = <class 'numpy.uint8'>*)

Constructor.

Parameters

- **extent** (Box) – The extent of the region to which the labels belong, in global coordinates.
- **num_classes** (*int*) – Number of classes.
- **dtype** (*Any*) – dtype of the scores array. Defaults to np.float16.
- **dtype_hits** (*Any*) – dtype of the hits array. Defaults to np.uint8.

add_predictions(*windows: Iterable[Box]*, *predictions: Iterable[Any]*, *crop_sz: Optional[int] = None*) → None

Populate predictions.

Parameters

- **windows** (*Iterable[Box]*) – Boxes in pixel coords, specifying chips in the raster.
- **predictions** (*Iterable[Any]*) – The model predictions for each chip specified by the windows.
- **crop_sz** (*Optional[int]*) – Number of rows/columns of pixels from the edge of prediction windows to discard. This is useful because predictions near edges tend to be lower quality and can result in very visible artifacts near the edges of chips. This should only be used if the given windows represent a sliding-window grid over the scene extent with overlap between adjacent windows. Defaults to None.

Return type
None

add_window(*window: Box*, *pixel_class_scores: ndarray*) → None

Set labels for the given window.

Parameters

- **window** (Box) –
- **pixel_class_scores** (*ndarray*) –

Return type
None

filter_by_aoi(*aoi_polygons: list*, *null_class_id: int*, ***kwargs*) → *SemanticSegmentationLabels*

Keep only the values that lie inside the AOI.

Parameters

- **aoi_polygons** (*list*) –
- **null_class_id** (*int*) –

Return type
SemanticSegmentationLabels

classmethod from_predictions(*windows: Iterable[Box]*, *predictions: Iterable[Any]*, *extent: Box*, *num_classes: int*, *crop_sz: Optional[int] = None*) → Union[*SemanticSegmentationDiscreteLabels*, *SemanticSegmentationSmoothLabels*]

Instantiate from windows and their corresponding predictions.

Parameters

- **windows** (*Iterable[Box]*) – Boxes in pixel coords, specifying chips in the raster.
- **predictions** (*Iterable[Any]*) – The model predictions for each chip specified by the windows.
- **extent** (Box) – The extent of the region to which the labels belong, in global coordinates.
- **num_classes** (*int*) – Number of classes.
- **crop_sz** (*Optional[int]*) – Number of rows/columns of pixels from the edge of prediction windows to discard. This is useful because predictions near edges tend to be lower quality and can result in very visible artifacts near the edges of chips. This should only be used if the given windows represent a sliding-window grid over the scene extent with overlap between adjacent windows. Defaults to None.

Returns

Union[SemanticSegmentationDiscreteLabels, SemanticSegmentationSmoothLabels]: If smooth=True, returns a

SemanticSegmentationSmoothLabels. Otherwise, a SemanticSegmentationDiscreteLabels.

Return type
Union[SemanticSegmentationDiscreteLabels, SemanticSegmentationSmoothLabels]

get_label_arr(*window: Box*, *null_class_id: int = -1*) → ndarray

Get labels as array of class IDs.

Returns null_class_id for pixels for which there is no data.

Parameters

- **window** (Box) –
- **null_class_id** (*int*) –

Return type
ndarray

get_score_arr(*window:* Box) → ndarray

Get array of pixel scores.

Parameters
window (Box) –

Return type
ndarray

get_windows(***kwargs*) → List[*Box*]

Generate sliding windows over the local extent. The keyword args are passed to Box.get_windows() and can therefore be used to control the specifications of the windows.

If the keyword args do not contain size, a list of length 1, containing the full extent is returned.

Return type
List[Box]

classmethod make_empty(*extent:* Box, *num_classes: int*) → *SemanticSegmentationSmoothLabels*

Instantiate an empty instance.

Parameters

- **extent** (Box) –
- **num_classes** (*int*) –

Return type
SemanticSegmentationSmoothLabels

mask_fill(*window:* Box, *mask: ndarray*, *fill_value: Any*) → None

Set fill_value'th class ID's score to 1 and all others to zero.

Parameters

- **window** (Box) –
- **mask** (*ndarray*) –
- **fill_value** (*Any*) –

Return type
None

save(*uri: str*, *crs_transformer:* CRSTransformer, *class_config:* ClassConfig, *tmp_dir: Optional[str] = None*, *save_as_rgb: bool = False*, *discrete_output: bool = True*, *smooth_output: bool = True*, *smooth_as_uint8: bool = False*, *rasterio_block_size: int = 512*, *vector_outputs: Optional[Sequence[*VectorOutputConfig*]] = None*, *profile_overrides: Optional[dict] = None*) → None

Save labels as rasters and/or vectors.

If URI is remote, all files will be first written locally and then uploaded to the URI.

Parameters

- **uri** (*str*) – URI of directory in which to save all output files.
- **crs_transformer** (CRSTransformer) – CRSTransformer to configure CRS and affine transform of the output GeoTiff(s).
- **class_config** (ClassConfig) – The ClassConfig.
- **tmp_dir** (*Optional[str], optional*) – Temporary directory to use. If None, will be auto-generated. Defaults to None.

- **save_as_rgb** (*bool, optional*) – If True, saves labels as an RGB image, using the class-color mapping in the class_config. Defaults to False.
- **discrete_output** (*bool, optional*) – If True, saves labels as a raster of class IDs (one band). Defaults to True.
- **smooth_output** (*bool, optional*) – If True, saves labels as a raster of class scores (one band for each class). Defaults to True.
- **smooth_as_uint8** (*bool, optional*) – If True, stores smooth class scores as np.uint8 (0-255) values rather than as np.float32 discrete labels, to help save memory/disk space. Defaults to False.
- **rasterio_block_size** (*int, optional*) – Value to set blockxsize and blockysize to. Defaults to 512.
- **vector_outputs** (*Optional[Sequence[VectorOutputConfig]], optional*) – List of VectorOutputConfig's containing vectorization configuration information. Only classes for which a VectorOutputConfig is specified will be saved as vectors. If None, no vector outputs will be produced. Defaults to None.
- **profile_overrides** (*Optional[dict], optional*) – This can be used to arbitrarily override properties in the profile used to create the output GeoTiff(s). Defaults to None.

Return type
None

tfod_utils

Modules

np_box_list	Numpy BoxList classes and functions.
np_box_list_ops	Bounding Box List operations for Numpy BoxLists.
np_box_ops	Operations for [N, 4] numpy arrays representing bounding boxes.

np_box_list

Numpy BoxList classes and functions.

Classes

BoxList	A list of bounding boxes as a [y_min, x_min, y_max, x_max] numpy array.

BoxList

class BoxList

Bases: object

A list of bounding boxes as a [y_min, x_min, y_max, x_max] numpy array.

It is assumed that all bounding boxes within a given list correspond to a single image. Optionally, users can add additional related fields (such as objectness/classification scores).

__init__(*data: ndarray*)

Constructor.

Parameters
data (*np.ndarray*) – Box coords as a [N, 4] numpy array.

Raises

- **ValueError** – If bbox data is not a numpy array.
- **ValueError** – If invalid dimensions for bbox data.

Methods

__init__(data)	Constructor.
add_field(name, data)	Add data to a specified field.
get()	Shorthand for get_field('boxes').
get_coordinates()	Get corner coordinates of boxes.
get_extra_fields()	Return all non-box fields.
get_field(name)	Get data for field.
has_field(field)	
num_boxes()	Return number of boxes held in collections.

__init__(*data: ndarray*)

Constructor.

Parameters
data (*np.ndarray*) – Box coords as a [N, 4] numpy array.

Raises

- **ValueError** – If bbox data is not a numpy array.
- **ValueError** – If invalid dimensions for bbox data.

add_field(*name: str*, *data: ndarray*) → None

Add data to a specified field.

Parameters

- **name** (*str*) – Field name.
- **data** (*np.ndarray*) – Field data: box coords as a [N, 4] numpy array.

Raises

- **ValueError** – If name already exists.
- **ValueError** – If the dimension of the field data does not matche the number of boxes.

Return type
None

`get()`

Shorthand for get_field('boxes').

`get_coordinates()` → Tuple[ndarray, ndarray, ndarray, ndarray]

Get corner coordinates of boxes.

Returns

a 4-tuple
of 1-d numpy arrays [y_min, x_min, y_max, x_max].

Return type
Tuple[np.ndarray, np.ndarray, np.ndarray, np.ndarray]

`get_extra_fields()` → List[str]

Return all non-box fields.

Return type
List[str]

`get_field`(*name: str*) → ndarray

Get data for field.

Parameters
name (*str*) – Field name.

Returns
The data associated with the field.

Return type
np.ndarray

Raises
ValueError – if invalid field.

`has_field`(*field*) → bool

Return type
bool

`num_boxes()` → int

Return number of boxes held in collections.

Return type
int

np_box_list_ops

Bounding Box List operations for Numpy BoxLists.

Classes

SortOrder	Enum class for sort order.

SortOrder

class SortOrder

Bases: object

Enum class for sort order.

ascend

ascend order.

descend

descend order.

Attributes

ASCEND

DESCEND

__init__()

Methods

__init__()

ASCEND = 1

DESCEND = 2

Functions

area(boxlist)	Computes area of boxes.
change_coordinate_frame(boxlist, window)	Change coordinate frame of the boxlist to be relative to window's frame.
clip_to_window(boxlist, window)	Clip bounding boxes to a window.
concatenate(boxlists[, fields])	Concatenate list of BoxLists.
filter_scores_greater_than(boxlist, thresh)	Filter to keep only boxes with score exceeding a given threshold.
gather(boxlist, indices[, fields])	Gather boxes from BoxList according to indices and return new BoxList.
intersection(boxlist1, boxlist2)	Compute pairwise intersection areas between boxes.
ioa(boxlist1, boxlist2)	Computes pairwise intersection-over-area between box collections.
iou(boxlist1, boxlist2)	Computes pairwise intersection-over-union between box collections.
multi_class_non_max_suppression(boxlist, ...)	Multi-class version of non maximum suppression.
non_max_suppression(boxlist[, ...])	Non maximum suppression.
prune_non_overlapping_boxes(boxlist1, boxlist2)	Prunes boxes with insufficient overlap b/w boxlists.
prune_outside_window(boxlist, window)	Prunes bounding boxes that fall outside a given window.
scale(boxlist, y_scale, x_scale)	Scale box coordinates in x and y dimensions.
sort_by_field(boxlist, field[, order])	Sort boxes and associated fields according to a scalar field.

area

area(*boxlist:* BoxList) → ndarray

Computes area of boxes.

Parameters
boxlist (BoxList) – BoxList holding N boxes.

Returns
A numpy array with shape [N*1] representing box areas.

Return type
np.ndarray

change_coordinate_frame

change_coordinate_frame(*boxlist:* BoxList, *window: ndarray*) → *BoxList*

Change coordinate frame of the boxlist to be relative to window's frame.

Given a window of the form [ymin, xmin, ymax, xmax], changes bounding box coordinates from boxlist to be relative to this window (e.g., the min corner maps to (0,0) and the max corner maps to (1,1)). An example use case is data augmentation: where we are given groundtruth boxes (boxlist) and would like to randomly crop the image to some window (window). In this case we need to change the coordinate frame of each groundtruth box to be relative to this new window.

Parameters

- **boxlist** (BoxList) – A BoxList object holding N boxes.

- **window** (*ndarray*) – a size 4 1-D numpy array.

Returns
Returns a BoxList object with N boxes.

Return type
BoxList

clip_to_window

clip_to_window(*boxlist:* BoxList, *window:* *ndarray*) → *BoxList*

Clip bounding boxes to a window.

This op clips input bounding boxes (represented by bounding box corners) to a window, optionally filtering out boxes that do not overlap at all with the window.

Parameters

- **boxlist** (BoxList) – A BoxList holding M_in boxes
- **window** (*ndarray*) – a numpy array of shape [4] representing the [y_min, x_min, y_max, x_max] window to which the op should clip boxes.

Returns
A BoxList holding M_out boxes where M_out <= M_in

Return type
BoxList

concatenate

concatenate(*boxlists: List[*BoxList*], fields: Optional[List[str]] = None*) → *BoxList*

Concatenate list of BoxLists.

This op concatenates a list of input BoxLists into a larger BoxList. It also handles concatenation of BoxList fields as long as the field tensor shapes are equal except for the first dimension.

Parameters

- **boxlists** (*List[*BoxList*]*) – List of BoxList objects.
- **fields** (*Optional[List[str]]*) – Optional list of fields to also concatenate. If None, all fields from the first BoxList in the list are included in the concatenation. Defaults to None.

Returns

A BoxList with number of boxes equal to
sum([boxlist.num_boxes() for boxlist in BoxList])

Return type
BoxList

Raises
ValueError – If boxlists is invalid (i.e., is not a list, is empty, or contains non BoxList objects), or if requested fields are not contained in all boxlists

filter_scores_greater_than

filter_scores_greater_than(*boxlist:* BoxList, *thresh: float*) → *BoxList*

Filter to keep only boxes with score exceeding a given threshold.

This op keeps the collection of boxes whose corresponding scores are greater than the input threshold.

Parameters

- **boxlist** (BoxList) – A BoxList holding N boxes. Must contain a 'scores' field representing detection scores.
- **thresh** (*float*) – scalar threshold

Returns
A BoxList holding M boxes. where M <= N

Return type
BoxList

Raises
ValueError – If boxlist not a BoxList object or if it does not have a scores field.

gather

gather(*boxlist:* BoxList, *indices: ndarray*, *fields: Optional[List[str]] = None*) → *BoxList*

Gather boxes from BoxList according to indices and return new BoxList.

By default, gather returns boxes corresponding to the input index list, as well as all additional fields stored in the boxlist (indexing into the first dimension). However one can optionally only gather from a subset of fields.

Parameters

- **boxlist** (BoxList) – BoxList holding N boxes.
- **indices** (*np.ndarray*) – A 1-d numpy array of type int.
- **fields** (*Optional[List[str]]*) – List of fields to also gather from. If None, all fields are gathered from. Pass an empty fields list to only gather the box coordinates. Defaults to None.

Returns

a BoxList corresponding to the subset of the input BoxList
specified by indices

Return type
BoxList

Raises
ValueError – If specified field is not contained in boxlist or if the indices are not of type int.

intersection

intersection(*boxlist1:* BoxList, *boxlist2:* BoxList) → ndarray

Compute pairwise intersection areas between boxes.

Parameters

- **boxlist1** (BoxList) – BoxList holding N boxes.
- **boxlist2** (BoxList) – BoxList holding M boxes.

Returns

A numpy array with shape [N*M] representing pairwise
intersection area.

Return type
np.ndarray

ioa

ioa(*boxlist1:* BoxList, *boxlist2:* BoxList) → ndarray

Computes pairwise intersection-over-area between box collections.

Intersection-over-area (ioa) between two boxes box1 and box2 is defined as their intersection area over box2's area. Note that ioa is not symmetric, that is, IOA(box1, box2) != IOA(box2, box1).

Parameters

- **boxlist1** (BoxList) – BoxList holding N boxes.
- **boxlist2** (BoxList) – BoxList holding M boxes.

Returns

A numpy array with shape [N, M] representing pairwise ioa
scores.

Return type
np.ndarray

iou

iou(*boxlist1:* BoxList, *boxlist2:* BoxList) → ndarray

Computes pairwise intersection-over-union between box collections.

Parameters

- **boxlist1** (BoxList) – BoxList holding N boxes.
- **boxlist2** (BoxList) – BoxList holding M boxes.

Returns
A numpy array with shape [N, M] representing pairwise iou scores.

Return type
np.ndarray

multi_class_non_max_suppression

multi_class_non_max_suppression(*boxlist:* BoxList, *score_thresh: float*, *iou_thresh: float*, *max_output_size: int*) → *BoxList*

Multi-class version of non maximum suppression.

This op greedily selects a subset of detection bounding boxes, pruning away boxes that have high IOU (intersection over union) overlap (> thresh) with already selected boxes. It operates independently for each class for which scores are provided (via the scores field of the input box_list), pruning boxes with score less than a provided threshold prior to applying NMS.

Parameters

- **boxlist** (BoxList) – A BoxList holding N boxes. Must contain a 'scores' field representing detection scores. This scores field is a tensor that can be 1 dimensional (in the case of a single class) or 2-dimensional, which which case we assume that it takes the shape [num_boxes, num_classes]. We further assume that this rank is known statically and that scores.shape[1] is also known (i.e., the number of classes is fixed and known at graph construction time).
- **score_thresh** (*float*) – Scalar threshold for score (low scoring boxes are removed).
- **iou_thresh** (*float*) – Scalar threshold for IOU (boxes that that high IOU overlap with previously selected boxes are removed).
- **max_output_size** (*int*) – maximum number of retained boxes per class.

Returns

BoxList with M boxes with a rank-1 scores field representing
corresponding scores for each box with scores sorted in decreasing order and a rank-1 classes field representing a class label for each box.

Return type
BoxList

Raises
ValueError – If iou_thresh is not in [0, 1] or if input boxlist does not have a valid scores field.

non_max_suppression

non_max_suppression(*boxlist:* BoxList, *max_output_size: int = 10000*, *iou_threshold: float = 1.0*, *score_threshold: float = -10.0*) → *BoxList*

Non maximum suppression. This op greedily selects a subset of detection bounding boxes, pruning away boxes that have high IOU (intersection over union) overlap (> thresh) with already selected boxes. In each iteration, the detected bounding box with highest score in the available pool is selected.

Parameters

- **boxlist** (BoxList) – BoxList holding N boxes. Must contain a 'scores' field representing detection scores. All scores belong to the same class.
- **max_output_size** (*int*) – Maximum number of retained boxes. Defaults to 10_000.
- **iou_threshold** (*float*) – Intersection over union threshold. Defaults to 1.0.
- **score_threshold** (*float*) – Minimum score threshold. Remove the boxes with scores less than this value. Default value is set to -10. A very low threshold to pass pretty much all the boxes, unless the user sets a different score threshold. Defaults to -10.0.

Returns
A BoxList holding M boxes. where M <= max_output_size.

Return type
BoxList

Raises

- **ValueError** – If 'scores' field does not exist.
- **ValueError** – If threshold is not in [0, 1].
- **ValueError** – If max_output_size < 0.

prune_non_overlapping_boxes

prune_non_overlapping_boxes(*boxlist1:* BoxList, *boxlist2:* BoxList, *minoverlap:* *float* = *0.0*) → *BoxList*

Prunes boxes with insufficient overlap b/w boxlists.

Prunes the boxes in boxlist1 that overlap less than thresh with boxlist2. For each box in boxlist1, we want its IOA to be more than minoverlap with at least one of the boxes in boxlist2. If it does not, we remove it.

Parameters

- **boxlist1** (BoxList) – BoxList holding N boxes.
- **boxlist2** (BoxList) – BoxList holding M boxes.
- **minoverlap** (*float*) – float: Minimum required overlap between boxes, to count them as overlapping. Defaults to 0.0.

Returns
A pruned boxlist with size [N', 4].

Return type
BoxList

prune_outside_window

prune_outside_window(*boxlist:* BoxList, *window:* *ndarray*) → Tuple[*BoxList*, ndarray]

Prunes bounding boxes that fall outside a given window.

This function prunes bounding boxes that even partially fall outside the given window. See also ClipToWindow which only prunes bounding boxes that fall completely outside the window, and clips any bounding boxes that partially overflow.

Parameters

- **boxlist** (BoxList) – A BoxList holding M_in boxes.
- **window** (*np.ndaarray*) – A numpy array of size 4, representing [ymin, xmin, ymax, xmax] of the window.

Returns

Pruned Boxlist of length <= M_in and
an array of shape [M_out] indexing the valid bounding boxes in the input tensor.

Return type
Tuple[*BoxList*, np.ndarray]

scale

scale(*boxlist:* BoxList, *y_scale: float*, *x_scale: float*) → *BoxList*

Scale box coordinates in x and y dimensions.

Parameters

- **boxlist** (BoxList) – A BoxList holding N boxes.
- **y_scale** (*float*) –
- **x_scale** (*float*) –

Returns
A BoxList holding N boxes.

Return type
BoxList

sort_by_field

sort_by_field(*boxlist:* BoxList, *field: str*, *order:* SortOrder = *2*)

Sort boxes and associated fields according to a scalar field. A common use case is reordering the boxes according to descending scores.

Parameters

- **boxlist** (BoxList) – A BoxList holding N boxes.
- **field** (*str*) – A BoxList field for sorting and reordering the BoxList.
- **order** (SortOrder, *optional*) – 'descend' or 'ascend'. Default is descend.

Returns
A sorted BoxList with the field in the specified order.

Return type
BoxList

Raises

- **ValueError** – If specified field does not exist or is not of single dimension.
- **ValueError** – If the order is not either descend or ascend.

np_box_ops

Operations for [N, 4] numpy arrays representing bounding boxes.

Functions

area(boxes)	Computes area of boxes.
intersection(boxes1, boxes2)	Compute pairwise intersection areas between boxes.
ioa(boxes1, boxes2)	Computes pairwise intersection-over-area between box collections.
iou(boxes1, boxes2)	Computes pairwise intersection-over-union between box collections.

area

area(*boxes*)

Computes area of boxes.

Parameters
boxes (*np.ndarray*) – A numpy array with shape [N, 4] holding N boxes.

Returns
A numpy array with shape [N*1] representing box areas.

Return type
np.ndarray

intersection

intersection(*boxes1*, *boxes2*)

Compute pairwise intersection areas between boxes.

Parameters

- **boxes1** (*np.ndarray*) – A numpy array with shape [N, 4] holding N boxes.
- **boxes2** (*np.ndarray*) – A numpy array with shape [M, 4] holding M boxes.

Returns

A numpy array with shape [N*M] representing pairwise
intersection area.

Return type
np.ndarray

ioa

ioa(*boxes1*, *boxes2*)

Computes pairwise intersection-over-area between box collections.

Intersection-over-area (ioa) between two boxes box1 and box2 is defined as their intersection area over box2's area. Note that ioa is not symmetric, that is, IOA(box1, box2) != IOA(box2, box1).

Parameters

- **boxes1** (*np.ndarray*) – A numpy array with shape [N, 4] holding N boxes.
- **boxes2** (*np.ndarray*) – A numpy array with shape [M, 4] holding N boxes.

Returns

A numpy array with shape [N, M] representing pairwise ioa
scores.

Return type
np.ndarray

iou

iou(*boxes1*, *boxes2*)

Computes pairwise intersection-over-union between box collections.

Parameters

- **boxes1** (*np.ndarray*) – A numpy array with shape [N, 4] holding N boxes.
- **boxes2** (*np.ndarray*) – A numpy array with shape [M, 4] holding M boxes.

Returns

A numpy array with shape [N, M] representing pairwise iou
scores.

Return type
np.ndarray

utils

Functions

discard_prediction_edges(windows, ...)	Discard the edges of predicted chips.

discard_prediction_edges

discard_prediction_edges(*windows: Iterable[*Box*]*, *predictions: Iterable[np.ndarray]*, *crop_sz: int*) → Tuple[List[*Box*], Iterator[np.ndarray]]

Discard the edges of predicted chips.

Parameters

- **windows** (*Iterable[*Box*]*) – The windows corresponding to the chips.
- **predictions** (*Iterable[np.ndarray]*) – The predicted chips.
- **crop_sz** (*int*) – Number of pixel rows/cols to discard.

Returns
Cropped windows and chips.

Return type
Tuple[Iterator[*Box*], Iterator[np.ndarray]]

label_source

Modules

chip_classification_label_source
chip_classification_label_source_config
label_source
label_source_config
object_detection_label_source
object_detection_label_source_config
semantic_segmentation_label_source
semantic_segmentation_label_source_config

chip_classification_label_source

Classes

ChipClassificationLabelSource	A source of chip classification labels.

ChipClassificationLabelSource

class ChipClassificationLabelSource

Bases: *LabelSource*

A source of chip classification labels.

Ideally the vector_source contains a square for each cell in the grid. But in reality, it can be difficult to label imagery in such an exhaustive way. So, this can also handle sources with non-overlapping polygons that do not necessarily cover the entire extent. It infers the grid of cells and associated class_ids using the extent and options if infer_cells is set to True.

Attributes

extent

__init__(*label_source_config:* ChipClassificationLabelSourceConfig, *vector_source:* VectorSource, *extent:* Box = *None*, *lazy:* *bool* = *False*)

Constructs a LabelSource for chip classification.

Parameters

- **label_source_config** (ChipClassificationLabelSourceConfig) – Config for class inference.
- **vector_source** (VectorSource) – Source of vector labels.
- **extent** (Box) – Box used to filter the labels by extent or compute grid.
- **lazy** (*bool, optional*) – If True, labels are not populated during initialization. Defaults to False.

Methods

__init__(label_source_config, vector_source)	Constructs a LabelSource for chip classification.
get_labels([window])	Return labels overlapping with window.
infer_cells([cells])	Infer labels for a list of cells.
populate_labels([cells])	Populate self.labels by either reading or inferring.
validate_labels(df)	

__init__(*label_source_config: ChipClassificationLabelSourceConfig*, *vector_source: VectorSource*, *extent: Box = None*, *lazy: bool = False*)

Constructs a LabelSource for chip classification.

Parameters

- **label_source_config** (ChipClassificationLabelSourceConfig) – Config for class inference.
- **vector_source** (VectorSource) – Source of vector labels.
- **extent** (Box) – Box used to filter the labels by extent or compute grid.
- **lazy** (*bool, optional*) – If True, labels are not populated during initialization. Defaults to False.

get_labels(*window: Optional[Box] = None*) → *ChipClassificationLabels*

Return labels overlapping with window.

Parameters
window (*Optional[Box]*) – Box

Returns

Labels overlapping with window. If window is None,
returns all labels.

Return type
ChipClassificationLabels

infer_cells(*cells: Optional[Iterable[Box]] = None*) → *ChipClassificationLabels*

Infer labels for a list of cells. Only cells whose labels are not already known are inferred.

Parameters
cells (*Optional[Iterable[Box]], optional*) – Cells whose labels are to be inferred. Defaults to None.

Returns
labels

Return type
ChipClassificationLabels

populate_labels(*cells: Optional[Iterable[Box]] = None*) → None

Populate self.labels by either reading or inferring.

If cfg.infer_cells is True or specific cells are given, the labels are inferred. Otherwise, they are read from the geojson.

Parameters
cells (*Optional[Iterable[Box]]*) –

Return type
None

validate_labels(*df: geopandas.GeoDataFrame*) → None

Parameters
df (*geopandas.GeoDataFrame*) –

Return type
None

property extent: *Box*

Functions

infer_cells(cells, labels_df, ioa_thresh, ...)	Infer ChipClassificationLabels grid from GeoJSON containing polygons.
read_labels(labels_df[, extent])	Convert GeoDataFrame to ChipClassificationLabels.

infer_cells

infer_cells(*cells: List[Box]*, *labels_df: geopandas.GeoDataFrame*, *ioa_thresh: float*, *use_intersection_over_cell: bool*, *pick_min_class_id: bool*, *background_class_id: int*) → *ChipClassificationLabels*

Infer ChipClassificationLabels grid from GeoJSON containing polygons.

Given GeoJSON with polygons associated with class_ids, infer a grid of cells and class_ids that best captures the contents of each cell.

For each cell, the problem is to infer the class_id that best captures the content of the cell. This is non-trivial since there can be multiple polygons of differing classes overlapping with the cell. Any polygons that sufficiently overlaps with the cell are in the running for setting the class_id. If there are none in the running, the cell is either considered null or background.

Parameters

- **ioa_thresh** (*float*) – (float) the minimum IOA of a polygon and cell for that polygon to be a candidate for setting the class_id
- **use_intersection_over_cell** (*bool*) – (bool) If true, then use the area of the cell as the denominator in the IOA. Otherwise, use the area of the polygon.
- **background_class_id** (*int*) – (None or int) If not None, class_id to use as the background class; ie. the one that is used when a window contains no boxes.
- **pick_min_class_id** (*bool*) – If true, the class_id for a cell is the minimum class_id of the boxes in that cell. Otherwise, pick the class_id of the box covering the greatest area.

- **cells** (*List[Box]*) –
- **labels_df** (*geopandas.GeoDataFrame*) –

Returns
ChipClassificationLabels

Return type
ChipClassificationLabels

read_labels

read_labels(*labels_df: geopandas.GeoDataFrame*, *extent: Optional[Box] = None*) → *ChipClassificationLabels*

Convert GeoDataFrame to ChipClassificationLabels.

If the GeoDataFrame already contains a grid of cells, then ChipClassificationLabels can be constructed in a straightforward manner without having to infer the class of cells.

If extent is given, only labels that intersect with it are returned.

Parameters

- **geojson** – dict in normalized GeoJSON format (see VectorSource)
- **extent** (*Optional[Box]*) – Box in pixel coords
- **labels_df** (*geopandas.GeoDataFrame*) –

Returns
ChipClassificationLabels

Return type
ChipClassificationLabels

chip_classification_label_source_config

Configs

ChipClassificationLabelSourceConfig	Configure a *ChipClassificationLabelSource*.

ChipClassificationLabelSourceConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model ChipClassificationLabelSourceConfig

Configure a *ChipClassificationLabelSource*.

This can be provided explicitly as a grid of cells, or a grid of cells can be inferred from arbitrary polygons.

```
{
   "title": "ChipClassificationLabelSourceConfig",
   "description": "Configure a :class:`.ChipClassificationLabelSource`.\n\nThis can␣
↪be provided explicitly as a grid of cells, or a grid of cells can\nbe inferred␣
```

(continues on next page)

(continued from previous page)


```
↪from arbitrary polygons.",
   "type": "object",
   "properties": {
      "type_hint": {
         "title": "Type Hint",
         "default": "chip_classification_label_source",
         "enum": [
            "chip_classification_label_source"
         ],
         "type": "string"
      },
      "vector_source": {
         "$ref": "#/definitions/VectorSourceConfig"
      },
      "ioa_thresh": {
         "title": "Ioa Thresh",
         "description": "Minimum IOA of a polygon and cell for that polygon to be a␣
↪candidate for setting the class_id.",
         "default": 0.5,
         "type": "number"
      },
      "use_intersection_over_cell": {
         "title": "Use Intersection Over Cell",
         "description": "If True, then use the area of the cell as the denominator␣
↪in the IOA. Otherwise, use the area of the polygon.",
         "default": false,
         "type": "boolean"
      },
      "pick_min_class_id": {
         "title": "Pick Min Class Id",
         "description": "If True, the class_id for a cell is the minimum class_id␣
↪of the boxes in that cell. Otherwise, pick the class_id of the box covering the␣
↪greatest area.",
         "default": false,
         "type": "boolean"
      },
      "background_class_id": {
         "title": "Background Class Id",
         "description": "If not None, class_id to use as the background class; ie.␣
↪the one that is used when a window contains no boxes. Cannot be None if infer_
↪cells=True.",
         "type": "integer"
      },
      "infer_cells": {
         "title": "Infer Cells",
         "description": "If True, infers a grid of cells based on the cell_sz.",
         "default": false,
         "type": "boolean"
      },
      "cell_sz": {
         "title": "Cell Sz",
         "description": "Size of a cell to use in pixels. If None, and this Config␣
```


(continues on next page)

(continued from previous page)


```
↪is part of an RVPipeline, this field will be set from RVPipeline.train_chip_sz.",
        "type": "integer"
      },
      "lazy": {
        "title": "Lazy",
        "description": "If True, labels will not be populated automatically during␣
↪initialization of the label source.",
        "default": false,
        "type": "boolean"
      }
   },
   "additionalProperties": false,
   "definitions": {
      "VectorTransformerConfig": {
        "title": "VectorTransformerConfig",
        "description": "Configure a :class:`.VectorTransformer`.",
        "type": "object",
        "properties": {
          "type_hint": {
            "title": "Type Hint",
            "default": "vector_transformer",
            "enum": [
              "vector_transformer"
            ],
            "type": "string"
          }
        },
        "additionalProperties": false
      },
      "VectorSourceConfig": {
        "title": "VectorSourceConfig",
        "description": "Configure a :class:`.VectorSource`.",
        "type": "object",
        "properties": {
          "transformers": {
            "title": "Transformers",
            "description": "List of VectorTransformers.",
            "default": [],
            "type": "array",
            "items": {
              "$ref": "#/definitions/VectorTransformerConfig"
            }
          },
          "type_hint": {
            "title": "Type Hint",
            "default": "vector_source",
            "enum": [
              "vector_source"
            ],
            "type": "string"
          }
        },
```


(continues on next page)

(continued from previous page)

```
            "additionalProperties": false
        }
    }
}
```

Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *background_class_id (Optional[int])*
- *cell_sz (Optional[int])*
- *infer_cells (bool)*
- *ioa_thresh (float)*
- *lazy (bool)*
- *pick_min_class_id (bool)*
- *type_hint (Literal['chip_classification_label_source'])*
- *use_intersection_over_cell (bool)*
- *vector_source (Optional[rastervision.core.data.vector_source.vector_source_config.VectorSourceConfig])*

Validators

- *ensure_bg_class_id_if_inferring* » `all fields`
- *ensure_required_transformers* » *vector_source*

field background_class_id: Optional[int] = None

If not None, class_id to use as the background class; ie. the one that is used when a window contains no boxes. Cannot be None if infer_cells=True.

Validated by

- *ensure_bg_class_id_if_inferring*

field cell_sz: Optional[int] = None

Size of a cell to use in pixels. If None, and this Config is part of an RVPipeline, this field will be set from RVPipeline.train_chip_sz.

Validated by

- *ensure_bg_class_id_if_inferring*

field infer_cells: bool = False

If True, infers a grid of cells based on the cell_sz.

Validated by

- *ensure_bg_class_id_if_inferring*

field ioa_thresh: float = 0.5

Minimum IOA of a polygon and cell for that polygon to be a candidate for setting the class_id.

Validated by

- *ensure_bg_class_id_if_inferring*

field lazy: bool = False

If True, labels will not be populated automatically during initialization of the label source.

Validated by

- *ensure_bg_class_id_if_inferring*

field pick_min_class_id: bool = False

If True, the class_id for a cell is the minimum class_id of the boxes in that cell. Otherwise, pick the class_id of the box covering the greatest area.

Validated by

- *ensure_bg_class_id_if_inferring*

field type_hint: Literal['chip_classification_label_source'] = 'chip_classification_label_source'

Validated by

- *ensure_bg_class_id_if_inferring*

field use_intersection_over_cell: bool = False

If True, then use the area of the cell as the denominator in the IOA. Otherwise, use the area of the polygon.

Validated by

- *ensure_bg_class_id_if_inferring*

field vector_source: Optional[*VectorSourceConfig*] = None

Validated by

- *ensure_bg_class_id_if_inferring*
- *ensure_required_transformers*

build(*class_config*, *crs_transformer*, *extent=None*, *tmp_dir=None*)

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

validator ensure_bg_class_id_if_inferring » *all fields*

Parameters
values (*dict*) –

Return type
dict

validator ensure_required_transformers » vector_source

Add class-inference and buffer transformers if absent.

Parameters
v (VectorSourceConfig) –

Return type
VectorSourceConfig

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*pipeline=None*, *scene=None*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises
ConfigError – if field is invalid

label_source

Classes

LabelSource	An interface for storage of labels for a scene.

LabelSource

class LabelSource

Bases: ABC

An interface for storage of labels for a scene.

An LabelSource is a read source of labels for a scene that could be backed by a file, a database, an API, etc. The difference between LabelSources and Labels can be understood by analogy to the difference between a database and result sets queried from a database.

Attributes

extent

__init__()

Methods

__init__()

get_labels([window])	Return labels overlapping with window.

abstract get_labels(*window=None*)

Return labels overlapping with window.

Parameters
window – Box

Returns

Labels overlapping with window. If window is None,
returns all labels.

abstract property extent: *Box*

label_source_config

Configs

LabelSourceConfig	Configure a *LabelSource*.

LabelSourceConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model LabelSourceConfig

Configure a *LabelSource*.


```
{
   "title": "LabelSourceConfig",
   "description": "Configure a :class:`.LabelSource`.",
   "type": "object",
   "properties": {
      "type_hint": {
         "title": "Type Hint",
         "default": "label_source",
```


(continues on next page)

(continued from previous page)

```
            "enum": [
               "label_source"
            ],
            "type": "string"
         }
      },
      "additionalProperties": false
}
```

Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *type_hint (Literal['label_source'])*

field type_hint: Literal['label_source'] = 'label_source'

build(*class_config*, *crs_transformer*, *extent*, *tmp_dir*)

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*pipeline=None*, *scene=None*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises
ConfigError – if field is invalid

object_detection_label_source

Classes

ObjectDetectionLabelSource	A read-only label source for object detection.

ObjectDetectionLabelSource

class ObjectDetectionLabelSource

Bases: *LabelSource*

A read-only label source for object detection.

Attributes

extent

__init__(*vector_source: VectorSource*, *extent: Box*, *ioa_thresh: Optional[float] = None*, *clip: bool = False*)

Constructor.

Parameters

- **vector_source** (*VectorSource*) – A VectorSource.
- **extent** (*Box*) – Box used to filter the labels by extent.
- **ioa_thresh** (*Optional[float], optional*) – IOA threshold to apply when retieving labels for a window. Defaults to None.
- **clip** (*bool, optional*) – Clip bounding boxes to window limits when retrieving labels for a window. Defaults to False.

Methods

__init__(vector_source, extent[, ...])	Constructor.
get_labels([window, ioa_thresh, clip])	Get labels (in global coords) for a window.
validate_geojson(geojson)	

__init__(*vector_source: VectorSource*, *extent: Box*, *ioa_thresh: Optional[float] = None*, *clip: bool = False*)

Constructor.

Parameters

- **vector_source** (*VectorSource*) – A VectorSource.
- **extent** (*Box*) – Box used to filter the labels by extent.

- **ioa_thresh** (*Optional[float], optional*) – IOA threshold to apply when retieving labels for a window. Defaults to None.
- **clip** (*bool, optional*) – Clip bounding boxes to window limits when retrieving labels for a window. Defaults to False.

get_labels(*window: Optional[*Box*] = None*, *ioa_thresh: float = 1e-06*, *clip: bool = False*) → *ObjectDetectionLabels*

Get labels (in global coords) for a window.

Parameters

- **window** (Box) – Window coords.
- **ioa_thresh** (*float*) –
- **clip** (*bool*) –

Returns

Labels with sufficient overlap with the window. The returned labels are in global coods (i.e. coords wihtin the full extent).

Return type
ObjectDetectionLabels

validate_geojson(*geojson: dict*) → None

Parameters
geojson (*dict*) –

Return type
None

property extent: *Box*

object_detection_label_source_config

Configs

ObjectDetectionLabelSourceConfig	Configure an *ObjectDetectionLabelSource*.

ObjectDetectionLabelSourceConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model ObjectDetectionLabelSourceConfig

Configure an *ObjectDetectionLabelSource*.


```
{
   "title": "ObjectDetectionLabelSourceConfig",
   "description": "Configure an :class:`.ObjectDetectionLabelSource`.",
   "type": "object",
   "properties": {
```


(continues on next page)

(continued from previous page)


```
      "type_hint": {
         "title": "Type Hint",
         "default": "object_detection_label_source",
         "enum": [
            "object_detection_label_source"
         ],
         "type": "string"
      },
      "vector_source": {
         "$ref": "#/definitions/VectorSourceConfig"
      }
   },
   "required": [
      "vector_source"
   ],
   "additionalProperties": false,
   "definitions": {
      "VectorTransformerConfig": {
         "title": "VectorTransformerConfig",
         "description": "Configure a :class:`.VectorTransformer`.",
         "type": "object",
         "properties": {
            "type_hint": {
               "title": "Type Hint",
               "default": "vector_transformer",
               "enum": [
                  "vector_transformer"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      },
      "VectorSourceConfig": {
         "title": "VectorSourceConfig",
         "description": "Configure a :class:`.VectorSource`.",
         "type": "object",
         "properties": {
            "transformers": {
               "title": "Transformers",
               "description": "List of VectorTransformers.",
               "default": [],
               "type": "array",
               "items": {
                  "$ref": "#/definitions/VectorTransformerConfig"
               }
            },
            "type_hint": {
               "title": "Type Hint",
               "default": "vector_source",
               "enum": [
                  "vector_source"
```


(continues on next page)

(continued from previous page)

```
                ],
                "type": "string"
            }
        },
        "additionalProperties": false
    }
}
```

Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *type_hint (Literal['object_detection_label_source'])*
- *vector_source (rastervision.core.data.vector_source.vector_source_config.VectorSourceConfig)*

Validators

- *ensure_required_transformers* » *vector_source*

field type_hint: Literal['object_detection_label_source'] = 'object_detection_label_source'

field vector_source: *VectorSourceConfig* [Required]

Validated by

- *ensure_required_transformers*

build(*class_config*, *crs_transformer*, *extent*, *tmp_dir=None*) → *ObjectDetectionLabelSource*

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

Return type
ObjectDetectionLabelSource

validator ensure_required_transformers » vector_source

Add class-inference and buffer transformers if absent.

Parameters
v (VectorSourceConfig) –

Return type
VectorSourceConfig

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*pipeline=None*, *scene=None*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

semantic_segmentation_label_source

Classes

SemanticSegmentationLabelSource	A read-only label source for semantic segmentation.

SemanticSegmentationLabelSource

class SemanticSegmentationLabelSource

Bases: *LabelSource*

A read-only label source for semantic segmentation.

Attributes

extent

__init__(*raster_source: RasterSource*, *class_config: ClassConfig*)

Constructor.

Parameters

- **raster_source** (RasterSource) – A raster source that returns a single channel raster with class_ids as values.
- **null_class_id** (*int*) – the null class id used as fill values for when windows go over the edge of the label array. This can be retrieved using class_config.null_class_id.
- **class_config** (ClassConfig) –

Methods

__init__(raster_source, class_config)	Constructor.
enough_target_pixels(window, ...)	Check if window contains enough pixels of the given target classes.
get_label_arr([window])	Get labels for a window.
get_labels([window])	Get labels for a window.

__init__(*raster_source:* RasterSource, *class_config:* ClassConfig)

Constructor.

Parameters

- **raster_source** (RasterSource) – A raster source that returns a single channel raster with class_ids as values.
- **null_class_id** (*int*) – the null class id used as fill values for when windows go over the edge of the label array. This can be retrieved using class_config.null_class_id.
- **class_config** (ClassConfig) –

enough_target_pixels(*window:* Box, *target_count_threshold:* *int*, *target_classes:* *List[int]*) → bool

Check if window contains enough pixels of the given target classes.

Parameters

- **window** (Box) – The larger window from-which the sub-window will be clipped.
- **target_count_threshold** (*int*) – Minimum number of target pixels.
- **target_classes** (*List[int]*) – The classes of interest. The given window is examined to make sure that it contains a sufficient number of target pixels.

Returns

True (the window does contain interesting pixels) or False.

Return type

bool

get_label_arr(*window:* *Optional[*Box*]* = *None*) → ndarray

Get labels for a window.

The returned array will be the same size as the input window. If window overflows the extent, the overflowing region will be filled with the ID of the null class as defined by the class_config.

Parameters

- **window** (*Optional[Box], optional*) – Window to get labels for. If
- **None** –
- **scene.** (*returns a label array covering the full extent of the*) –

Returns
Label array.

Return type
np.ndarray

`get_labels`(*window: Optional[Box] = None*) → *SemanticSegmentationLabels*

Get labels for a window.

Parameters

- **window** (*Optional[Box], optional*) – Window to get labels for. If
- **None** –
- **scene.** (*returns labels covering the full extent of the*) –

Returns
The labels.

Return type
SemanticSegmentationLabels

`property extent: Box`

Functions

fill_edge(label_arr, window, extent, fill_value)	If window goes over the edge of the extent, buffer with fill_value.

fill_edge

`fill_edge`(*label_arr: ndarray*, *window:* Box, *extent:* Box, *fill_value: int*) → ndarray

If window goes over the edge of the extent, buffer with fill_value.

Parameters

- **label_arr** (*ndarray*) –
- **window** (Box) –
- **extent** (Box) –
- **fill_value** (*int*) –

Return type
ndarray

semantic_segmentation_label_source_config

Configs

SemanticSegmentationLabelSourceConfig	Configure a *SemanticSegmentationLabelSource*.

SemanticSegmentationLabelSourceConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model SemanticSegmentationLabelSourceConfig

Configure a *SemanticSegmentationLabelSource*.


```
{
   "title": "SemanticSegmentationLabelSourceConfig",
   "description": "Configure a :class:`.SemanticSegmentationLabelSource`.",
   "type": "object",
   "properties": {
      "type_hint": {
         "title": "Type Hint",
         "default": "semantic_segmentation_label_source",
         "enum": [
            "semantic_segmentation_label_source"
         ],
         "type": "string"
      },
      "raster_source": {
         "title": "Raster Source",
         "description": "The labels in the form of rasters.",
         "anyOf": [
            {
               "$ref": "#/definitions/RasterSourceConfig"
            },
            {
               "$ref": "#/definitions/RasterizedSourceConfig"
            }
         ]
      }
   },
   "required": [
      "raster_source"
   ],
   "additionalProperties": false,
   "definitions": {
      "RasterTransformerConfig": {
         "title": "RasterTransformerConfig",
         "description": "Configure a :class:`.RasterTransformer`.",
         "type": "object",
         "properties": {
```


(continues on next page)

(continued from previous page)


```
            "type_hint": {
               "title": "Type Hint",
               "default": "raster_transformer",
               "enum": [
                  "raster_transformer"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      },
      "RasterSourceConfig": {
         "title": "RasterSourceConfig",
         "description": "Configure a :class:`.RasterSource`.",
         "type": "object",
         "properties": {
            "channel_order": {
               "title": "Channel Order",
               "description": "The sequence of channel indices to use when reading␣
↪imagery.",
               "type": "array",
               "items": {
                  "type": "integer"
               }
            },
            "transformers": {
               "title": "Transformers",
               "default": [],
               "type": "array",
               "items": {
                  "$ref": "#/definitions/RasterTransformerConfig"
               }
            },
            "extent": {
               "title": "Extent",
               "description": "Use-specified extent in pixel coords in the form␣
↪(ymin, xmin, ymax, xmax). Useful for cropping the raster source so that only part␣
↪of the raster is read from.",
               "type": "array",
               "minItems": 4,
               "maxItems": 4,
               "items": [
                  {
                     "type": "integer"
                  },
                  {
                     "type": "integer"
                  },
                  {
                     "type": "integer"
                  },
                  {
```


(continues on next page)

(continued from previous page)


```
                    "type": "integer"
                }
            ]
        },
        "type_hint": {
            "title": "Type Hint",
            "default": "raster_source",
            "enum": [
                "raster_source"
            ],
            "type": "string"
        }
    },
    "additionalProperties": false
},
"VectorTransformerConfig": {
    "title": "VectorTransformerConfig",
    "description": "Configure a :class:`.VectorTransformer`.",
    "type": "object",
    "properties": {
        "type_hint": {
            "title": "Type Hint",
            "default": "vector_transformer",
            "enum": [
                "vector_transformer"
            ],
            "type": "string"
        }
    },
    "additionalProperties": false
},
"VectorSourceConfig": {
    "title": "VectorSourceConfig",
    "description": "Configure a :class:`.VectorSource`.",
    "type": "object",
    "properties": {
        "transformers": {
            "title": "Transformers",
            "description": "List of VectorTransformers.",
            "default": [],
            "type": "array",
            "items": {
                "$ref": "#/definitions/VectorTransformerConfig"
            }
        },
        "type_hint": {
            "title": "Type Hint",
            "default": "vector_source",
            "enum": [
                "vector_source"
            ],
            "type": "string"
```


(continues on next page)

(continued from previous page)

```
            }
         },
         "additionalProperties": false
      },
      "RasterizerConfig": {
         "title": "RasterizerConfig",
         "description": "Configure rasterization params for a :class:`.
↪RasterizedSource`.",
         "type": "object",
         "properties": {
            "background_class_id": {
               "title": "Background Class Id",
               "description": "The class_id to use for any background pixels, i.e.␣
↪pixels not covered by a polygon.",
               "type": "integer"
            },
            "all_touched": {
               "title": "All Touched",
               "description": "If True, all pixels touched by geometries will be␣
↪burned in. If false, only pixels whose center is within the polygon or that are␣
↪selected by Bresenham's line algorithm will be burned in. (See rasterio.features.
↪rasterize for more details).",
               "default": false,
               "type": "boolean"
            },
            "type_hint": {
               "title": "Type Hint",
               "default": "rasterizer",
               "enum": [
                  "rasterizer"
               ],
               "type": "string"
            }
         },
         "required": [
            "background_class_id"
         ],
         "additionalProperties": false
      },
      "RasterizedSourceConfig": {
         "title": "RasterizedSourceConfig",
         "description": "Configure a :class:`.RasterizedSource`.",
         "type": "object",
         "properties": {
            "vector_source": {
               "$ref": "#/definitions/VectorSourceConfig"
            },
            "rasterizer_config": {
               "$ref": "#/definitions/RasterizerConfig"
            },
            "type_hint": {
               "title": "Type Hint",
```

(continues on next page)

(continued from previous page)

```
                "default": "rasterized_source",
                "enum": [
                    "rasterized_source"
                ],
                "type": "string"
            }
        },
        "required": [
            "vector_source",
            "rasterizer_config"
        ],
        "additionalProperties": false
    }
}
}
```

Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *raster_source (Union[rastervision.core.data.raster_source.raster_source_config.RasterSourceConfig, rastervision.core.data.raster_source.rasterized_source_config.RasterizedSourceConfig])*
- *type_hint (Literal['semantic_segmentation_label_source'])*

field raster_source: Union[*RasterSourceConfig*, *RasterizedSourceConfig*] [Required]

The labels in the form of rasters.

field type_hint: Literal['semantic_segmentation_label_source'] = 'semantic_segmentation_label_source'

build(*class_config*, *crs_transformer*, *extent*, *tmp_dir*)

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*pipeline=None*, *scene=None*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

label_store

Modules

chip_classification_geojson_store
chip_classification_geojson_store_config
label_store
label_store_config
object_detection_geojson_store
object_detection_geojson_store_config
semantic_segmentation_label_store
semantic_segmentation_label_store_config
utils

chip_classification_geojson_store

Classes

ChipClassificationGeoJSONStore	Storage for chip classification predictions.

ChipClassificationGeoJSONStore

class ChipClassificationGeoJSONStore

Bases: *LabelStore*

Storage for chip classification predictions.

__init__(*uri: str*, *class_config:* ClassConfig, *crs_transformer:* CRSTransformer)

Constructor.

Parameters

- **uri** (*str*) – uri of GeoJSON file containing labels
- **class_config** (ClassConfig) – ClassConfig
- **crs_transformer** (CRSTransformer) – CRSTransformer to convert from map coords in label in GeoJSON file to pixel coords.

Methods

__init__(uri, class_config, crs_transformer)	Constructor.
empty_labels()	Produces an empty Labels
get_labels()	Loads Labels from this label store.
save(labels)	Save labels to URI if writable.

__init__(*uri: str*, *class_config:* ClassConfig, *crs_transformer:* CRSTransformer)

Constructor.

Parameters

- **uri** (*str*) – uri of GeoJSON file containing labels
- **class_config** (ClassConfig) – ClassConfig
- **crs_transformer** (CRSTransformer) – CRSTransformer to convert from map coords in label in GeoJSON file to pixel coords.

empty_labels() → *ChipClassificationLabels*

Produces an empty Labels

Return type
ChipClassificationLabels

get_labels() → *ChipClassificationLabels*

Loads Labels from this label store.

Return type
ChipClassificationLabels

save(*labels:* ChipClassificationLabels) → None

Save labels to URI if writable.

Note that if the grid is inferred from polygons, only the grid will be written, not the original polygons.

Parameters
labels (ChipClassificationLabels) –

Return type
None

chip_classification_geojson_store_config

Configs

ChipClassificationGeoJSONStoreConfig	Configure a *ChipClassificationGeoJSONStore*.

ChipClassificationGeoJSONStoreConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model ChipClassificationGeoJSONStoreConfig

Configure a *ChipClassificationGeoJSONStore*.


```
{
   "title": "ChipClassificationGeoJSONStoreConfig",
   "description": "Configure a :class:`.ChipClassificationGeoJSONStore`.",
   "type": "object",
   "properties": {
      "type_hint": {
         "title": "Type Hint",
         "default": "chip_classification_geojson_store",
         "enum": [
            "chip_classification_geojson_store"
         ],
         "type": "string"
      },
      "uri": {
         "title": "Uri",
         "description": "URI of GeoJSON file with predictions. If None, and this␣
↪Config is part of a SceneConfig inside an RVPipelineConfig, it will be auto-
↪generated.",
         "type": "string"
      }
   },
   "additionalProperties": false
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *type_hint (Literal['chip_classification_geojson_store'])*
- *uri (Optional[str])*

field type_hint: Literal['chip_classification_geojson_store'] = 'chip_classification_geojson_store'

field uri: Optional[str] = None

URI of GeoJSON file with predictions. If None, and this Config is part of a SceneConfig inside an RVPipelineConfig, it will be auto-generated.

build(*class_config*, *crs_transformer*, *extent=None*, *tmp_dir=None*)

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*pipeline=None*, *scene=None*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

label_store

Classes

LabelStore	This defines how to store prediction labels for a scene.

LabelStore

class LabelStore

Bases: ABC

This defines how to store prediction labels for a scene.

__init__()

Methods

__init__()

empty_labels()	Produces an empty Labels
get_labels()	Loads Labels from this label store.
save(labels)	Save.

abstract empty_labels()

Produces an empty Labels

abstract get_labels()

Loads Labels from this label store.

abstract save(*labels*)

Save.

Parameters

- **saved** (*labels - Labels to be*) – of pipeline.
- **type** (*the type of which will be dependant on the*) – of pipeline.

label_store_config

Configs

LabelStoreConfig	Configure a *LabelStore*.

LabelStoreConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model LabelStoreConfig

Configure a *LabelStore*.

```
{
   "title": "LabelStoreConfig",
   "description": "Configure a :class:`.LabelStore`.",
   "type": "object",
```

(continues on next page)

(continued from previous page)

```
   "properties": {
      "type_hint": {
         "title": "Type Hint",
         "default": "label_store",
         "enum": [
            "label_store"
         ],
         "type": "string"
      }
   },
   "additionalProperties": false
}
```

Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *type_hint (Literal['label_store'])*

field type_hint: Literal['label_store'] = 'label_store'

build(*class_config*, *crs_transformer*, *extent*, *tmp_dir*)

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*pipeline=None*, *scene=None*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises
ConfigError – if field is invalid

object_detection_geojson_store

Classes

ObjectDetectionGeoJSONStore	Storage for object detection predictions.

ObjectDetectionGeoJSONStore

class ObjectDetectionGeoJSONStore

Bases: *LabelStore*

Storage for object detection predictions.

__init__(*uri: str*, *class_config:* ClassConfig, *crs_transformer:* CRSTransformer)

Constructor.

Parameters

- **uri** (*str*) – uri of GeoJSON file containing labels
- **class_config** (ClassConfig) – ClassConfig used to infer class_ids from class_name (or label) field
- **crs_transformer** (CRSTransformer) – CRSTransformer to convert from map coords in label in GeoJSON file to pixel coords.

Methods

__init__(uri, class_config, crs_transformer)	Constructor.
empty_labels()	Produces an empty Labels
get_labels()	Loads Labels from this label store.
save(labels)	Save labels to URI.

__init__(*uri: str*, *class_config:* ClassConfig, *crs_transformer:* CRSTransformer)

Constructor.

Parameters

- **uri** (*str*) – uri of GeoJSON file containing labels
- **class_config** (ClassConfig) – ClassConfig used to infer class_ids from class_name (or label) field
- **crs_transformer** (CRSTransformer) – CRSTransformer to convert from map coords in label in GeoJSON file to pixel coords.

empty_labels() → *ObjectDetectionLabels*

Produces an empty Labels

Return type
ObjectDetectionLabels

get_labels() → *ObjectDetectionLabels*

Loads Labels from this label store.

Return type
ObjectDetectionLabels

save(*labels:* ObjectDetectionLabels) → None

Save labels to URI.

Parameters
labels (ObjectDetectionLabels) –

Return type
None

object_detection_geojson_store_config

Configs

ObjectDetectionGeoJSONStoreConfig	Configure an *ObjectDetectionGeoJSONStore*.

ObjectDetectionGeoJSONStoreConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model ObjectDetectionGeoJSONStoreConfig

Configure an *ObjectDetectionGeoJSONStore*.


```
{
   "title": "ObjectDetectionGeoJSONStoreConfig",
   "description": "Configure an :class:`.ObjectDetectionGeoJSONStore`.",
   "type": "object",
   "properties": {
      "type_hint": {
         "title": "Type Hint",
         "default": "object_detection_geojson_store",
         "enum": [
            "object_detection_geojson_store"
         ],
         "type": "string"
      },
      "uri": {
         "title": "Uri",
         "description": "URI of GeoJSON file with predictions. If None, and this␣
↪Config is part of a SceneConfig inside an RVPipelineConfig, it will be auto-
```


(continues on next page)

(continued from previous page)

```
↪generated.",
            "type": "string"
        }
    },
    "additionalProperties": false
}
```

Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *type_hint (Literal['object_detection_geojson_store'])*
- *uri (Optional[str])*

field type_hint: Literal['object_detection_geojson_store'] = 'object_detection_geojson_store'

field uri: Optional[str] = None

URI of GeoJSON file with predictions. If None, and this Config is part of a SceneConfig inside an RVPipelineConfig, it will be auto-generated.

build(*class_config*, *crs_transformer*, *extent=None*, *tmp_dir=None*)

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*pipeline=None*, *scene=None*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

semantic_segmentation_label_store

Classes

SemanticSegmentationLabelStore	Storage for semantic segmentation predictions.

SemanticSegmentationLabelStore

class SemanticSegmentationLabelStore

Bases: *LabelStore*

Storage for semantic segmentation predictions.

Can store predicted class ID raster and class scores raster as GeoTIFFs, and can optionally vectorize predictions and store them as GeoJSON files.

__init__(*uri: str*, *extent: Box*, *crs_transformer: CRSTransformer*, *class_config: ClassConfig*, *tmp_dir: Optional[str] = None*, *vector_outputs: Optional[Sequence[VectorOutputConfig]] = None*, *save_as_rgb: bool = False*, *discrete_output: bool = True*, *smooth_output: bool = False*, *smooth_as_uint8: bool = False*, *rasterio_block_size: int = 512*)

Constructor.

Parameters

- **uri** (*str*) – Path to directory where the predictions are/will be stored. Smooth scores will be saved as "uri/scores.tif", discrete labels will be stored as "uri/labels.tif", and vector outputs will be saved in "uri/vector_outputs/".
- **extent** (*Box*) – The extent of the scene.
- **crs_transformer** (*CRSTransformer*) – CRS transformer for correctly mapping from pixel coords to map coords.
- **tmp_dir** (*Optional[str], optional*) – Temporary directory to use. If None, will be auto-generated. Defaults to None.
- **vector_outputs** (*Optional[Sequence[VectorOutputConfig]], optional*) – List of VectorOutputConfig's containing vectorization configuration information. Only classes for which a VectorOutputConfig is specified will be saved as vectors. If None, no vector outputs will be produced. Defaults to None.
- **class_config** (*ClassConfig*) – Class config.
- **save_as_rgb** (*bool, optional*) – If True, saves labels as an RGB image, using the class-color mapping in the class_config. Defaults to False.
- **discrete_output** (*bool, optional*) – If True, saves labels as a raster of class IDs (one band). Defaults to False.
- **smooth_output** (*bool, optional*) – If True, saves labels as a raster of class scores (one band for each class). Defaults to False.

- **smooth_as_uint8** (*bool, optional*) – If True, stores smooth class scores as np.uint8 (0-255) values rather than as np.float32 discrete labels, to help save memory/disk space. Defaults to False.
- **rasterio_block_size** (*int, optional*) – Value to set blockxsize and blockysize to. Defaults to 512.

Methods

__init__(uri, extent, crs_transformer, ...)	Constructor.
empty_labels(**kwargs)	Returns an empty SemanticSegmentationLabels object.
get_discrete_labels()	Get all labels.
get_labels()	Get all labels.
get_scores()	Get all scores.
save(labels[, profile])	Save labels to disk.
write_discrete_raster_output(out_profile, ...)	
write_smooth_raster_output(out_profile, ...)	
write_vector_outputs(labels)	Write vectorized outputs for all configs in self.vector_outputs.

__init__(*uri: str*, *extent: Box*, *crs_transformer: CRSTransformer*, *class_config: ClassConfig*, *tmp_dir: Optional[str] = None*, *vector_outputs: Optional[Sequence[VectorOutputConfig]] = None*, *save_as_rgb: bool = False*, *discrete_output: bool = True*, *smooth_output: bool = False*, *smooth_as_uint8: bool = False*, *rasterio_block_size: int = 512*)

Constructor.

Parameters

- **uri** (*str*) – Path to directory where the predictions are/will be stored. Smooth scores will be saved as "uri/scores.tif", discrete labels will be stored as "uri/labels.tif", and vector outputs will be saved in "uri/vector_outputs/".
- **extent** (*Box*) – The extent of the scene.
- **crs_transformer** (*CRSTransformer*) – CRS transformer for correctly mapping from pixel coords to map coords.
- **tmp_dir** (*Optional[str], optional*) – Temporary directory to use. If None, will be auto-generated. Defaults to None.
- **vector_outputs** (*Optional[Sequence[VectorOutputConfig]], optional*) – List of VectorOutputConfig's containing vectorization configuration information. Only classes for which a VectorOutputConfig is specified will be saved as vectors. If None, no vector outputs will be produced. Defaults to None.
- **class_config** (*ClassConfig*) – Class config.
- **save_as_rgb** (*bool, optional*) – If True, saves labels as an RGB image, using the class-color mapping in the class_config. Defaults to False.
- **discrete_output** (*bool, optional*) – If True, saves labels as a raster of class IDs (one band). Defaults to False.

- **smooth_output** (*bool, optional*) – If True, saves labels as a raster of class scores (one band for each class). Defaults to False.
- **smooth_as_uint8** (*bool, optional*) – If True, stores smooth class scores as np.uint8 (0-255) values rather than as np.float32 discrete labels, to help save memory/disk space. Defaults to False.
- **rasterio_block_size** (*int, optional*) – Value to set blockxsize and blockysize to. Defaults to 512.

empty_labels(***kwargs*) → *SemanticSegmentationLabels*

Returns an empty SemanticSegmentationLabels object.

Return type
SemanticSegmentationLabels

get_discrete_labels() → *SemanticSegmentationDiscreteLabels*

Get all labels.

Returns
SemanticSegmentationDiscreteLabels

Return type
SemanticSegmentationDiscreteLabels

get_labels() → *SemanticSegmentationLabels*

Get all labels.

Returns
SemanticSegmentationLabels

Return type
SemanticSegmentationLabels

get_scores() → *SemanticSegmentationSmoothLabels*

Get all scores.

Returns
SemanticSegmentationSmoothLabels

Return type
SemanticSegmentationSmoothLabels

save(*labels:* SemanticSegmentationLabels, *profile: Optional[dict] = None*) → None

Save labels to disk.

More info on rasterio IO: - https://github.com/mapbox/rasterio/blob/master/docs/quickstart.rst - https://rasterio.readthedocs.io/en/latest/topics/windowed-rw.html

Parameters

- **-** (*labels*) –
- **labels** (SemanticSegmentationLabels) –
- **profile** (*Optional[dict]*) –

Return type
None

write_discrete_raster_output(*out_profile: dict, path: str, labels:* SemanticSegmentationLabels) → None

Parameters

- **out_profile** (*dict*) –
- **path** (*str*) –
- **labels** (SemanticSegmentationLabels) –

Return type
None

write_smooth_raster_output(*out_profile: dict*, *scores_path: str*, *hits_path: str*, *labels:* SemanticSegmentationSmoothLabels) → None

Parameters

- **out_profile** (*dict*) –
- **scores_path** (*str*) –
- **hits_path** (*str*) –
- **labels** (SemanticSegmentationSmoothLabels) –

Return type
None

write_vector_outputs(*labels:* SemanticSegmentationLabels) → None

Write vectorized outputs for all configs in self.vector_outputs.

Parameters
labels (SemanticSegmentationLabels) –

Return type
None

semantic_segmentation_label_store_config

Configs

BuildingVectorOutputConfig	Config for vectorized semantic segmentation predictions.
PolygonVectorOutputConfig	Config for vectorized semantic segmentation predictions.
SemanticSegmentationLabelStoreConfig	Configure a *SemanticSegmentationLabelStore*.
VectorOutputConfig	Config for vectorized semantic segmentation predictions.

BuildingVectorOutputConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model BuildingVectorOutputConfig

Config for vectorized semantic segmentation predictions.

Intended to break up clusters of buildings.


```
{
   "title": "BuildingVectorOutputConfig",
   "description": "Config for vectorized semantic segmentation predictions.\n\
↪nIntended to break up clusters of buildings.",
   "type": "object",
   "properties": {
      "uri": {
         "title": "Uri",
         "description": "URI of vector output. If None, and this Config is part of␣
↪a SceneConfig and RVPipeline, this field will be auto-generated.",
         "type": "string"
      },
      "class_id": {
         "title": "Class Id",
         "description": "The prediction class that is to turned into vectors.",
         "type": "integer"
      },
      "denoise": {
         "title": "Denoise",
         "description": "Radius of the structural element used to remove high-
↪frequency signals from the image. Smaller values will reduce less noise and make␣
↪vectorization slower (especially for large images). Larger values will remove␣
↪more noise and make vectorization faster but might also remove legitimate␣
↪detections.",
         "default": 8,
         "type": "integer"
      },
      "type_hint": {
         "title": "Type Hint",
         "default": "building_vector_output",
         "enum": [
            "building_vector_output"
         ],
         "type": "string"
      },
      "min_area": {
         "title": "Min Area",
         "description": "Minimum area (in pixels^2) of anything that can be␣
↪considered to be a building or a cluster of buildings. The goal is to distinguish␣
↪between buildings and artifacts.",
         "default": 0.0,
         "type": "number"
      },
```


(continues on next page)

(continued from previous page)

```
      "element_width_factor": {
         "title": "Element Width Factor",
         "description": "Width of the structural element used to break building␣
↪clusters as a fraction of the width of the cluster.",
         "default": 0.5,
         "type": "number"
      },
      "element_thickness": {
         "title": "Element Thickness",
         "description": "Thickness of the structural element that is used to break␣
↪building clusters.",
         "default": 0.001,
         "type": "number"
      }
   },
   "required": [
      "class_id"
   ],
   "additionalProperties": false
}
```

Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *class_id (int)*
- *denoise (int)*
- *element_thickness (float)*
- *element_width_factor (float)*
- *min_area (float)*
- *type_hint (Literal['building_vector_output'])*
- *uri (Optional[str])*

field class_id: int [Required]

The prediction class that is to turned into vectors.

field denoise: int = 8

Radius of the structural element used to remove high-frequency signals from the image. Smaller values will reduce less noise and make vectorization slower (especially for large images). Larger values will remove more noise and make vectorization faster but might also remove legitimate detections.

field element_thickness: float = 0.001

Thickness of the structural element that is used to break building clusters.

field element_width_factor: float = 0.5

Width of the structural element used to break building clusters as a fraction of the width of the cluster.

field min_area: float = 0.0

Minimum area (in pixels^2) of anything that can be considered to be a building or a cluster of buildings. The goal is to distinguish between buildings and artifacts.

field type_hint: Literal['building_vector_output'] = 'building_vector_output'

field uri: Optional[str] = None

URI of vector output. If None, and this Config is part of a SceneConfig and RVPipeline, this field will be auto-generated.

build()

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

get_mode() → str

Return type
str

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*pipeline: Optional[RVPipelineConfig] = None*, *scene: Optional[SceneConfig] = None*, *uri_prefix: Optional[str] = None*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

Parameters

- **pipeline** (*Optional[RVPipelineConfig]*) –
- **scene** (*Optional[SceneConfig]*) –
- **uri_prefix** (*Optional[str]*) –

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises
ConfigError – if field is invalid

PolygonVectorOutputConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model PolygonVectorOutputConfig

Config for vectorized semantic segmentation predictions.


```
{
   "title": "PolygonVectorOutputConfig",
   "description": "Config for vectorized semantic segmentation predictions.",
   "type": "object",
   "properties": {
      "uri": {
         "title": "Uri",
         "description": "URI of vector output. If None, and this Config is part of␣
↪a SceneConfig and RVPipeline, this field will be auto-generated.",
         "type": "string"
      },
      "class_id": {
         "title": "Class Id",
         "description": "The prediction class that is to turned into vectors.",
         "type": "integer"
      },
      "denoise": {
         "title": "Denoise",
         "description": "Radius of the structural element used to remove high-
↪frequency signals from the image. Smaller values will reduce less noise and make␣
↪vectorization slower (especially for large images). Larger values will remove␣
↪more noise and make vectorization faster but might also remove legitimate␣
↪detections.",
         "default": 8,
         "type": "integer"
      },
      "type_hint": {
         "title": "Type Hint",
         "default": "polygon_vector_output",
         "enum": [
            "polygon_vector_output"
         ],
         "type": "string"
      }
   },
   "required": [
      "class_id"
   ],
   "additionalProperties": false
}
```

Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *class_id (int)*
- *denoise (int)*
- *type_hint (Literal['polygon_vector_output'])*
- *uri (Optional[str])*

field class_id: int [Required]

The prediction class that is to turned into vectors.

field denoise: int = 8

Radius of the structural element used to remove high-frequency signals from the image. Smaller values will reduce less noise and make vectorization slower (especially for large images). Larger values will remove more noise and make vectorization faster but might also remove legitimate detections.

field type_hint: Literal['polygon_vector_output'] = 'polygon_vector_output'

field uri: Optional[str] = None

URI of vector output. If None, and this Config is part of a SceneConfig and RVPipeline, this field will be auto-generated.

build()

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

get_mode() → str

Return type
str

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*pipeline: Optional[RVPipelineConfig] = None*, *scene: Optional[SceneConfig] = None*, *uri_prefix: Optional[str] = None*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

Parameters

- **pipeline** (*Optional[RVPipelineConfig]*) –
- **scene** (*Optional[SceneConfig]*) –
- **uri_prefix** (*Optional[str]*) –

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

SemanticSegmentationLabelStoreConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model SemanticSegmentationLabelStoreConfig

Configure a *SemanticSegmentationLabelStore*.

Stores class raster as GeoTIFF, and can optionally vectorizes predictions and stores them in GeoJSON files.


```
{
   "title": "SemanticSegmentationLabelStoreConfig",
   "description": "Configure a :class:`.SemanticSegmentationLabelStore`.\n\nStores␣
↪class raster as GeoTIFF, and can optionally vectorizes predictions and stores\
↪nthem in GeoJSON files.",
   "type": "object",
   "properties": {
      "type_hint": {
         "title": "Type Hint",
         "default": "semantic_segmentation_label_store",
         "enum": [
            "semantic_segmentation_label_store"
         ],
         "type": "string"
      },
      "uri": {
         "title": "Uri",
         "description": "URI of file with predictions. If None, and this Config is␣
↪part of a SceneConfig inside an RVPipelineConfig, this fiend will be auto-
↪generated.",
         "type": "string"
      },
      "vector_output": {
```


(continues on next page)

(continued from previous page)


```
        "title": "Vector Output",
        "default": [],
        "type": "array",
        "items": {
           "$ref": "#/definitions/VectorOutputConfig"
        }
     },
     "rgb": {
        "title": "Rgb",
        "description": "If True, save prediction class_ids in RGB format using the␣
↪colors in class_config.",
        "default": false,
        "type": "boolean"
     },
     "smooth_output": {
        "title": "Smooth Output",
        "description": "If True, expects labels to be continuous values␣
↪representing class scores and stores both scores and discrete labels.",
        "default": false,
        "type": "boolean"
     },
     "smooth_as_uint8": {
        "title": "Smooth As Uint8",
        "description": "If True, stores smooth scores as uint8, resulting in loss␣
↪of precision, but reduced file size. Only used if smooth_output=True.",
        "default": false,
        "type": "boolean"
     },
     "rasterio_block_size": {
        "title": "Rasterio Block Size",
        "description": "blockxsize and blockysize params in rasterio.open() will␣
↪be set to this.",
        "default": 256,
        "type": "integer"
     }
  },
  "additionalProperties": false,
  "definitions": {
     "VectorOutputConfig": {
        "title": "VectorOutputConfig",
        "description": "Config for vectorized semantic segmentation predictions.",
        "type": "object",
        "properties": {
           "uri": {
              "title": "Uri",
              "description": "URI of vector output. If None, and this Config is␣
↪part of a SceneConfig and RVPipeline, this field will be auto-generated.",
              "type": "string"
           },
           "class_id": {
              "title": "Class Id",
              "description": "The prediction class that is to turned into vectors.
```


(continues on next page)

(continued from previous page)


```
↪",
                "type": "integer"
            },
            "denoise": {
                "title": "Denoise",
                "description": "Radius of the structural element used to remove high-
↪frequency signals from the image. Smaller values will reduce less noise and make␣
↪vectorization slower (especially for large images). Larger values will remove␣
↪more noise and make vectorization faster but might also remove legitimate␣
↪detections.",
                "default": 8,
                "type": "integer"
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "vector_output",
                "enum": [
                    "vector_output"
                ],
                "type": "string"
            }
        },
        "required": [
            "class_id"
        ],
        "additionalProperties": false
    }
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *rasterio_block_size (int)*
- *rgb (bool)*
- *smooth_as_uint8 (bool)*
- *smooth_output (bool)*
- *type_hint (Literal['semantic_segmentation_label_store'])*
- *uri (Optional[str])*
- *vector_output (List[rastervision.core.data.label_store.semantic_segmentation_label_store_config.VectorOutputConfig])*

field rasterio_block_size: int = 256

blockxsize and blockysize params in rasterio.open() will be set to this.

field rgb: bool = False

If True, save prediction class_ids in RGB format using the colors in class_config.

field smooth_as_uint8: bool = False

If True, stores smooth scores as uint8, resulting in loss of precision, but reduced file size. Only used if smooth_output=True.

field smooth_output: bool = False

If True, expects labels to be continuous values representing class scores and stores both scores and discrete labels.

field type_hint: Literal['semantic_segmentation_label_store'] = 'semantic_segmentation_label_store'

field uri: Optional[str] = None

URI of file with predictions. If None, and this Config is part of a SceneConfig inside an RVPipelineConfig, this fiend will be auto-generated.

field vector_output: List[*VectorOutputConfig*] = []

build(*class_config*, *crs_transformer*, *extent*, *tmp_dir*)

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*pipeline: Optional[RVPipelineConfig] = None*, *scene: Optional[SceneConfig] = None*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

Parameters

- **pipeline** (*Optional[RVPipelineConfig]*) –
- **scene** (*Optional[SceneConfig]*) –

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate

- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

VectorOutputConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model VectorOutputConfig

Config for vectorized semantic segmentation predictions.


```
{
   "title": "VectorOutputConfig",
   "description": "Config for vectorized semantic segmentation predictions.",
   "type": "object",
   "properties": {
      "uri": {
         "title": "Uri",
         "description": "URI of vector output. If None, and this Config is part of␣
↪a SceneConfig and RVPipeline, this field will be auto-generated.",
         "type": "string"
      },
      "class_id": {
         "title": "Class Id",
         "description": "The prediction class that is to turned into vectors.",
         "type": "integer"
      },
      "denoise": {
         "title": "Denoise",
         "description": "Radius of the structural element used to remove high-
↪frequency signals from the image. Smaller values will reduce less noise and make␣
↪vectorization slower (especially for large images). Larger values will remove␣
↪more noise and make vectorization faster but might also remove legitimate␣
↪detections.",
         "default": 8,
         "type": "integer"
      },
      "type_hint": {
         "title": "Type Hint",
         "default": "vector_output",
         "enum": [
            "vector_output"
         ],
         "type": "string"
      }
   },
   "required": [
      "class_id"
   ],
   "additionalProperties": false
```


(continues on next page)

(continued from previous page)

```
}
```

Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *class_id (int)*
- *denoise (int)*
- *type_hint (Literal['vector_output'])*
- *uri (Optional[str])*

field class_id: int [Required]

The prediction class that is to turned into vectors.

field denoise: int = 8

Radius of the structural element used to remove high-frequency signals from the image. Smaller values will reduce less noise and make vectorization slower (especially for large images). Larger values will remove more noise and make vectorization faster but might also remove legitimate detections.

field type_hint: Literal['vector_output'] = 'vector_output'

field uri: Optional[str] = None

URI of vector output. If None, and this Config is part of a SceneConfig and RVPipeline, this field will be auto-generated.

build()

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

get_mode() → str

Return type
str

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*pipeline: Optional[RVPipelineConfig] = None*, *scene: Optional[SceneConfig] = None*, *uri_prefix: Optional[str] = None*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

Parameters

- **pipeline** (*Optional[RVPipelineConfig]*) –
- **scene** (*Optional[SceneConfig]*) –
- **uri_prefix** (*Optional[str]*) –

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

utils

Functions

boxes_to_geojson(boxes, class_ids, ...[, scores])	Convert boxes and associated data into a GeoJSON dict.

boxes_to_geojson

boxes_to_geojson(*boxes: Sequence[Box]*, *class_ids: Sequence[int]*, *crs_transformer:* CRSTransformer, *class_config:* ClassConfig, *scores: Optional[Sequence[Union[float, Sequence[float]]]] = None*) → dict

Convert boxes and associated data into a GeoJSON dict.

Parameters

- **boxes** (*Sequence[Box]*) – List of Box in pixel row/col format.
- **class_ids** (*Sequence[int]*) – List of int (one for each box)
- **crs_transformer** (*CRSTransformer*) – CRSTransformer used to convert pixel coords to map coords in the GeoJSON.
- **class_config** (*ClassConfig*) – ClassConfig
- **scores** (*Optional[Sequence[Union[float, Sequence[float]]]], optional*) – Optional list of score or scores. If floats (one for each box), property name will be "score". If lists of floats, property name will be "scores". Defaults to None.

Returns

Serialized GeoJSON.

Return type
dict

raster_source

Modules

multi_raster_source
multi_raster_source_config
raster_source
raster_source_config
rasterio_source
rasterio_source_config
rasterized_source
rasterized_source_config

multi_raster_source

Classes

MultiRasterSource	Merge multiple `RasterSources` by concatenating along channel dim.

MultiRasterSource

class MultiRasterSource

Bases: *RasterSource*

Merge multiple `RasterSources` by concatenating along channel dim.

Attributes

crs_transformer	Associated *CRSTransformer*.
dtype	`numpy.dtype` of the chips read from this source.
extent	Extent of the RasterSource.
num_channels	Number of channels in the chips read from this source.
primary_source	Primary sub-`RasterSource`
shape	Shape of the raster as a (height, width, num_channels) tuple.

__init__(*raster_sources: Sequence[RasterSource], primary_source_idx: ConstrainedIntValue = 0, force_same_dtype: bool = False, channel_order: Optional[Sequence[ConstrainedIntValue]] = None, raster_transformers: Sequence = [], extent: Optional[Box] = None*)

Constructor.

Parameters

- **raster_sources** (*Sequence[RasterSource]*) – Sequence of RasterSources.
- **primary_source_idx** (*0 <= int < len(raster_sources)*) – Index of the raster source whose CRS, dtype, and other attributes will override those of the other raster sources.
- **force_same_dtype** (*bool*) – If true, force all sub-chips to have the same dtype as the primary_source_idx-th sub-chip. No careful conversion is done, just a quick cast. Use with caution.
- **channel_order** (*Sequence[conint(ge=0)], optional*) – Channel ordering that will be used by .get_chip(). Defaults to None.
- **raster_transformers** (*Sequence, optional*) – Sequence of transformers. Defaults to [].
- **extent** (*Optional[Box]*) –

Methods

__init__(raster_sources[, ...])	Constructor.
get_chip(window)	Return the transformed chip in the window.
get_image_array()	Return entire transformed image array.
get_raw_chip(window)	Return raw chip without applying channel_order or transforms.
get_raw_image_array()	Return raw image for the full extent.
validate_raster_sources()	Validate sub-RasterSources.

__init__(*raster_sources: Sequence[RasterSource], primary_source_idx: ConstrainedIntValue = 0, force_same_dtype: bool = False, channel_order: Optional[Sequence[ConstrainedIntValue]] = None, raster_transformers: Sequence = [], extent: Optional[Box] = None*)

Constructor.

Parameters

- **raster_sources** (*Sequence[RasterSource]*) – Sequence of RasterSources.
- **primary_source_idx** (*0 <= int < len(raster_sources)*) – Index of the raster source whose CRS, dtype, and other attributes will override those of the other raster sources.
- **force_same_dtype** (*bool*) – If true, force all sub-chips to have the same dtype as the primary_source_idx-th sub-chip. No careful conversion is done, just a quick cast. Use with caution.
- **channel_order** (*Sequence[conint(ge=0)], optional*) – Channel ordering that will be used by .get_chip(). Defaults to None.
- **raster_transformers** (*Sequence, optional*) – Sequence of transformers. Defaults to [].

- **extent** (*Optional[Box]*) –

get_chip(*window:* Box) → ndarray

Return the transformed chip in the window.

Get processed chips from sub raster sources (with their respective channel orders and transformations applied), concatenate them along the channel dimension, apply channel_order, followed by transformations.

Parameters
window (Box) – Box

Returns
np.ndarray with shape [height, width, channels]

Return type
ndarray

get_image_array() → np.ndarray

Return entire transformed image array.

> **Warning:** Not safe to call on very large RasterSources.

Returns
Array of shape (height, width, channels).

Return type
np.ndarray

get_raw_chip(*window:* Box) → np.ndarray

Return raw chip without applying channel_order or transforms.

Parameters
window (Box) – The window for which to get the chip.

Returns
Array of shape (height, width, channels).

Return type
np.ndarray

get_raw_image_array() → np.ndarray

Return raw image for the full extent.

> **Warning:** Not safe to call on very large RasterSources.

Returns
Array of shape (height, width, channels).

Return type
np.ndarray

validate_raster_sources() → None

Validate sub-RasterSources.

Checks if:

- dtypes are same or force_same_dtype is True.

- each sub-RasterSource is a *RasterioSource* if extents not identical.

Return type
None

property crs_transformer: *CRSTransformer*

Associated *CRSTransformer*.

property dtype: dtype

numpy.dtype of the chips read from this source.

property extent: *Box*

Extent of the RasterSource.

property num_channels: int

Number of channels in the chips read from this source.

property primary_source: *RasterSource*

Primary sub-RasterSource

property shape: Tuple[int, int, int]

Shape of the raster as a (height, width, num_channels) tuple.

multi_raster_source_config

Configs

MultiRasterSourceConfig	Configure a *MultiRasterSource*.

MultiRasterSourceConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model MultiRasterSourceConfig

Configure a *MultiRasterSource*.


```
{
   "title": "MultiRasterSourceConfig",
   "description": "Configure a :class:`.MultiRasterSource`.",
   "type": "object",
   "properties": {
      "channel_order": {
         "title": "Channel Order",
         "description": "The sequence of channel indices to use when reading␣
↪imagery.",
         "type": "array",
         "items": {
            "type": "integer"
         }
      },
```


(continues on next page)

(continued from previous page)

```
      "transformers": {
         "title": "Transformers",
         "default": [],
         "type": "array",
         "items": {
            "$ref": "#/definitions/RasterTransformerConfig"
         }
      },
      "extent": {
         "title": "Extent",
         "description": "Use-specified extent in pixel coords in the form (ymin,␣
↪xmin, ymax, xmax). Useful for cropping the raster source so that only part of the␣
↪raster is read from.",
         "type": "array",
         "minItems": 4,
         "maxItems": 4,
         "items": [
            {
               "type": "integer"
            },
            {
               "type": "integer"
            },
            {
               "type": "integer"
            },
            {
               "type": "integer"
            }
         ]
      },
      "type_hint": {
         "title": "Type Hint",
         "default": "multi_raster_source",
         "enum": [
            "multi_raster_source"
         ],
         "type": "string"
      },
      "raster_sources": {
         "title": "Raster Sources",
         "description": "List of RasterSourceConfig to combine.",
         "minItems": 1,
         "type": "array",
         "items": {
            "$ref": "#/definitions/RasterSourceConfig"
         }
      },
      "primary_source_idx": {
         "title": "Primary Source Idx",
         "description": "Index of the raster source whose CRS, dtype, and other␣
↪attributes will override those of the other raster sources. Defaults to 0.",
```

(continues on next page)

(continued from previous page)

```
            "default": 0,
            "minimum": 0,
            "type": "integer"
         },
         "force_same_dtype": {
            "title": "Force Same Dtype",
            "description": "Force all subchips to be of the same dtype as the primary_
↪source_idx-th subchip.",
            "default": false,
            "type": "boolean"
         }
      },
      "required": [
         "raster_sources"
      ],
      "additionalProperties": false,
      "definitions": {
         "RasterTransformerConfig": {
            "title": "RasterTransformerConfig",
            "description": "Configure a :class:`.RasterTransformer`.",
            "type": "object",
            "properties": {
               "type_hint": {
                  "title": "Type Hint",
                  "default": "raster_transformer",
                  "enum": [
                     "raster_transformer"
                  ],
                  "type": "string"
               }
            },
            "additionalProperties": false
         },
         "RasterSourceConfig": {
            "title": "RasterSourceConfig",
            "description": "Configure a :class:`.RasterSource`.",
            "type": "object",
            "properties": {
               "channel_order": {
                  "title": "Channel Order",
                  "description": "The sequence of channel indices to use when reading␣
↪imagery.",
                  "type": "array",
                  "items": {
                     "type": "integer"
                  }
               },
               "transformers": {
                  "title": "Transformers",
                  "default": [],
                  "type": "array",
                  "items": {
```

(continues on next page)

(continued from previous page)


```
                "$ref": "#/definitions/RasterTransformerConfig"
            }
        },
        "extent": {
            "title": "Extent",
            "description": "Use-specified extent in pixel coords in the form␣
↪(ymin, xmin, ymax, xmax). Useful for cropping the raster source so that only part␣
↪of the raster is read from.",
            "type": "array",
            "minItems": 4,
            "maxItems": 4,
            "items": [
                {
                    "type": "integer"
                },
                {
                    "type": "integer"
                },
                {
                    "type": "integer"
                },
                {
                    "type": "integer"
                }
            ]
        },
        "type_hint": {
            "title": "Type Hint",
            "default": "raster_source",
            "enum": [
                "raster_source"
            ],
            "type": "string"
        }
    },
    "additionalProperties": false
  }
 }
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *channel_order (Optional[List[int]])*
- *extent (Optional[Tuple[int, int, int, int]])*
- *force_same_dtype (bool)*
- *primary_source_idx (rastervision.core.data.raster_source.multi_raster_source_config.ConstrainedIntValue)*

- *raster_sources (types.ConstrainedListValue[rastervision.core.data.raster_source.raster_source_config.RasterSourceConfig])*
- *transformers (List[rastervision.core.data.raster_transformer.raster_transformer_config.RasterTransformerConfig])*
- *type_hint (Literal['multi_raster_source'])*

Validators

- validate_extent » *extent*
- *validate_primary_source_idx* » *primary_source_idx*

field channel_order: Optional[List[int]] = None

The sequence of channel indices to use when reading imagery.

field extent: Optional[Tuple[int, int, int, int]] = None

Use-specified extent in pixel coords in the form (ymin, xmin, ymax, xmax). Useful for cropping the raster source so that only part of the raster is read from.

Validated by

- validate_extent

field force_same_dtype: bool = False

Force all subchips to be of the same dtype as the primary_source_idx-th subchip.

field primary_source_idx: conint(ge=0) = 0

Index of the raster source whose CRS, dtype, and other attributes will override those of the other raster sources. Defaults to 0.

Constraints

- **minimum** = 0

Validated by

- *validate_primary_source_idx*

field raster_sources: conlist(RasterSourceConfig, min_items=1) [Required]

List of RasterSourceConfig to combine.

Constraints

- **minItems** = 1

field transformers: List[*RasterTransformerConfig*] = []

field type_hint: Literal['multi_raster_source'] = 'multi_raster_source'

build(*tmp_dir: str*, *use_transformers: bool = True*) → *MultiRasterSource*

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

Parameters

- **tmp_dir** (*str*) –
- **use_transformers** (*bool*) –

Return type

MultiRasterSource

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*pipeline=None*, *scene=None*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validator validate_extent » extent

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

validator validate_primary_source_idx » primary_source_idx

Parameters

- **v** (*int*) –
- **values** (*dict*) –

raster_source

Classes

RasterSource	A source of raster data.

RasterSource

class RasterSource

Bases: ABC

A source of raster data.

This should be subclassed when adding a new source of raster data such as a set of files, an API, a TMS URI schema, etc.

Attributes

crs_transformer	Associated *CRSTransformer*.
dtype	`numpy.dtype` of the chips read from this source.
extent	Extent of the RasterSource.
num_channels	Number of channels in the chips read from this source.
shape	Shape of the raster as a (height, width, num_channels) tuple.

__init__(*channel_order: Optional[List[int]]*, *num_channels_raw: int*, *raster_transformers: List[RasterTransformer] = []*, *extent: Optional[Box] = None*)

Constructor.

Parameters

- **channel_order** (*Optional[List[int]]*) – list of channel indices to use when extracting chip from raw imagery.
- **num_channels_raw** (*int*) – Number of channels in the raw imagery before applying channel_order.
- **raster_transformers** (*List[RasterTransformer]*) – `RasterTransformers` for transforming chips whenever they are retrieved. Defaults to `[]`.
- **extent** (*Optional[Box]*) – Use-specified extent. If None, the full extent of the raster source is used.

Methods

__init__(channel_order, num_channels_raw[, ...])	Constructor.
get_chip(window)	Return the transformed chip in the window.
get_image_array()	Return entire transformed image array.
get_raw_chip(window)	Return raw chip without applying channel_order or transforms.
get_raw_image_array()	Return raw image for the full extent.

__init__(*channel_order: Optional[List[int]]*, *num_channels_raw: int*, *raster_transformers: List[RasterTransformer] = []*, *extent: Optional[Box] = None*)

Constructor.

Parameters

- **channel_order** (*Optional[List[int]]*) – list of channel indices to use when extracting chip from raw imagery.
- **num_channels_raw** (*int*) – Number of channels in the raw imagery before applying channel_order.
- **raster_transformers** (*List[RasterTransformer]*) – RasterTransformers for transforming chips whenever they are retrieved. Defaults to [].
- **extent** (*Optional[Box]*) – Use-specified extent. If None, the full extent of the raster source is used.

get_chip(*window:* Box) → np.ndarray

Return the transformed chip in the window.

Get a raw chip, extract subset of channels using channel_order, and then apply transformations.

Parameters
window (Box) – The window for which to get the chip.

Returns
Array of shape (height, width, channels).

Return type
np.ndarray

get_image_array() → np.ndarray

Return entire transformed image array.

> **Warning:** Not safe to call on very large RasterSources.

Returns
Array of shape (height, width, channels).

Return type
np.ndarray

get_raw_chip(*window:* Box) → np.ndarray

Return raw chip without applying channel_order or transforms.

Parameters
window (Box) – The window for which to get the chip.

Returns
Array of shape (height, width, channels).

Return type
np.ndarray

get_raw_image_array() → np.ndarray

Return raw image for the full extent.

> **Warning:** Not safe to call on very large RasterSources.

Returns
Array of shape (height, width, channels).

Return type
np.ndarray

abstract property crs_transformer: *CRSTransformer*

Associated *CRSTransformer*.

abstract property dtype: np.dtype

`numpy.dtype` of the chips read from this source.

property extent: *Box*

Extent of the RasterSource.

property num_channels: int

Number of channels in the chips read from this source.

property shape: Tuple[int, int, int]

Shape of the raster as a (height, width, num_channels) tuple.

Exceptions

ChannelOrderError	

rastervision.core.data.raster_source.raster_source.ChannelOrderError

exception ChannelOrderError

__init__(*channel_order: List[int]*, *num_channels_raw: int*)

Parameters

- **channel_order** (*List[int]*) –
- **num_channels_raw** (*int*) –

__new__(***kwargs*)

raster_source_config

Configs

RasterSourceConfig	Configure a *RasterSource*.

RasterSourceConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model RasterSourceConfig

Configure a *RasterSource*.


```
{
   "title": "RasterSourceConfig",
   "description": "Configure a :class:`.RasterSource`.",
   "type": "object",
   "properties": {
      "channel_order": {
         "title": "Channel Order",
         "description": "The sequence of channel indices to use when reading␣
↪imagery.",
         "type": "array",
         "items": {
            "type": "integer"
         }
      },
      "transformers": {
         "title": "Transformers",
         "default": [],
         "type": "array",
         "items": {
            "$ref": "#/definitions/RasterTransformerConfig"
         }
      },
      "extent": {
         "title": "Extent",
         "description": "Use-specified extent in pixel coords in the form (ymin,␣
↪xmin, ymax, xmax). Useful for cropping the raster source so that only part of the␣
↪raster is read from.",
         "type": "array",
         "minItems": 4,
         "maxItems": 4,
         "items": [
            {
               "type": "integer"
            },
            {
               "type": "integer"
            },
            {
               "type": "integer"
            },
            {
               "type": "integer"
            }
         ]
```


(continues on next page)

(continued from previous page)


```
      },
      "type_hint": {
         "title": "Type Hint",
         "default": "raster_source",
         "enum": [
            "raster_source"
         ],
         "type": "string"
      }
   },
   "additionalProperties": false,
   "definitions": {
      "RasterTransformerConfig": {
         "title": "RasterTransformerConfig",
         "description": "Configure a :class:`.RasterTransformer`.",
         "type": "object",
         "properties": {
            "type_hint": {
               "title": "Type Hint",
               "default": "raster_transformer",
               "enum": [
                  "raster_transformer"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      }
   }
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *channel_order (Optional[List[int]])*
- *extent (Optional[Tuple[int, int, int, int]])*
- *transformers (List[rastervision.core.data.raster_transformer.raster_transformer_config.RasterTransformerConfig])*
- *type_hint (Literal['raster_source'])*

Validators

- *validate_extent* » *extent*

field channel_order: Optional[List[int]] = None

The sequence of channel indices to use when reading imagery.

field extent: Optional[Tuple[int, int, int, int]] = None

Use-specified extent in pixel coords in the form (ymin, xmin, ymax, xmax). Useful for cropping the raster source so that only part of the raster is read from.

Validated by

- *validate_extent*

field transformers: List[*RasterTransformerConfig*] = []

field type_hint: Literal['raster_source'] = 'raster_source'

build(*tmp_dir*, *use_transformers=True*)

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*pipeline=None*, *scene=None*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validator validate_extent » extent

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

rasterio_source

Classes

RasterioSource	A rasterio-based *RasterSource*.

RasterioSource

class RasterioSource

Bases: *RasterSource*

A rasterio-based *RasterSource*.

This RasterSource can read any file that can be opened by Rasterio/GDAL.

If there are multiple image files that cover a single scene, you can pass the corresponding list of URIs, and read them as if it were a single stitched-together image.

It can also read non-georeferenced images such as `.tif`, `.png`, and `.jpg` files. This is useful for oblique drone imagery, biomedical imagery, and any other (potentially massive!) non-georeferenced images.

If channel_order is None, then use non-alpha channels. This also sets any masked or NODATA pixel values to be zeros.

Attributes

crs_transformer	Associated *CRSTransformer*.
dtype	`numpy.dtype` of the chips read from this source.
extent	Extent of the RasterSource.
num_channels	Number of channels in the chips read from this source.
shape	Shape of the raster as a (height, width, num_channels) tuple.

__init__(*uris: Union[str, List[str]]*, *raster_transformers: List[RasterTransformer] = []*, *allow_streaming: bool = False*, *channel_order: Optional[Sequence[int]] = None*, *extent: Optional[Box] = None*, *tmp_dir: Optional[str] = None*)

Constructor.

Parameters

- **uris** (*Union[str, List[str]]*) – One or more URIs of images. If more than one, the images will be mosaiced together using GDAL.
- **raster_transformers** (*List['RasterTransformer']*) – RasterTransformers to use to trasnform chips after they are read.
- **allow_streaming** (*bool*) – If True, read data without downloading the entire file first. Defaults to False.
- **channel_order** (*Optional[Sequence[int]]*) – List of indices of channels to extract from raw imagery. Can be a subset of the available channels. If None, all channels available in the image will be read. Defaults to None.

- **extent** (*Optional[Box], optional*) – Use-specified extent. If None, the full extent of the raster source is used.
- **tmp_dir** (*Optional[str]*) – Directory to use for storing the VRT (needed if multiple uris or allow_streaming=True). If None, will be auto-generated. Defaults to None.

Methods

__init__(uris[, raster_transformers, ...])	Constructor.
download_data([vrt_dir, stream])	Download any data needed for this raster source.
get_chip(window[, bands, out_shape])	Read a chip specified by a window from the file.
get_image_array()	Return entire transformed image array.
get_raw_chip(window)	Return raw chip without applying channel_order or transforms.
get_raw_image_array()	Return raw image for the full extent.

__init__(*uris: Union[str, List[str]]*, *raster_transformers: List[RasterTransformer] = []*, *allow_streaming: bool = False*, *channel_order: Optional[Sequence[int]] = None*, *extent: Optional[Box] = None*, *tmp_dir: Optional[str] = None*)

Constructor.

Parameters

- **uris** (*Union[str, List[str]]*) – One or more URIs of images. If more than one, the images will be mosaiced together using GDAL.
- **raster_transformers** (*List['RasterTransformer']*) – RasterTransformers to use to trasnform chips after they are read.
- **allow_streaming** (*bool*) – If True, read data without downloading the entire file first. Defaults to False.
- **channel_order** (*Optional[Sequence[int]]*) – List of indices of channels to extract from raw imagery. Can be a subset of the available channels. If None, all channels available in the image will be read. Defaults to None.
- **extent** (*Optional[Box], optional*) – Use-specified extent. If None, the full extent of the raster source is used.
- **tmp_dir** (*Optional[str]*) – Directory to use for storing the VRT (needed if multiple uris or allow_streaming=True). If None, will be auto-generated. Defaults to None.

download_data(*vrt_dir: Optional[str] = None*, *stream: bool = False*) → str

Download any data needed for this raster source.

Return a single local path representing the image or a VRT of the data.

Parameters

- **vrt_dir** (*Optional[str]*) –
- **stream** (*bool*) –

Return type

str

get_chip(*window: Box*, *bands: Optional[Union[Sequence[int], slice]] = None*, *out_shape: Optional[Tuple[int, ...]] = None*) → ndarray

Read a chip specified by a window from the file.

Parameters

- **window** (Box) – Bounding box of chip in pixel coordinates.
- **bands** (*Optional[Union[Sequence[int], slice]], optional*) – Subset of bands to read. Note that this will be applied on top of the channel_order (if specified). So if this is an RGB image and channel_order=[2, 1, 0], then using bands=[0] will return the B-channel. Defaults to None.
- **out_shape** (*Optional[Tuple[int, ...]], optional*) – (hieght, width) of
- **None** (*the output chip. If*) –
- **None.** (*no resizing is done. Defaults to*) –

Returns
A chip of shape (height, width, channels).

Return type
np.ndarray

get_image_array() → np.ndarray
Return entire transformed image array.

> **Warning:** Not safe to call on very large RasterSources.

Returns
Array of shape (height, width, channels).

Return type
np.ndarray

get_raw_chip(*window:* Box) → np.ndarray
Return raw chip without applying channel_order or transforms.

Parameters
window (Box) – The window for which to get the chip.

Returns
Array of shape (height, width, channels).

Return type
np.ndarray

get_raw_image_array() → np.ndarray
Return raw image for the full extent.

> **Warning:** Not safe to call on very large RasterSources.

Returns
Array of shape (height, width, channels).

Return type
np.ndarray

property crs_transformer: *RasterioCRSTransformer*
Associated *CRSTransformer*.

property dtype: Tuple[int, int, int]

numpy.dtype of the chips read from this source.

property extent: *Box*

Extent of the RasterSource.

property num_channels: int

Number of channels in the chips read from this source.

Note: Unlike the parent class, RasterioSource applies channel_order (via bands_to_read) before raster_transformers. So the number of output channels is not guaranteed to be equal to len(channel_order).

---

property shape: Tuple[int, int, int]

Shape of the raster as a (height, width, num_channels) tuple.

Functions

build_vrt(vrt_path, image_uris)	Build a VRT for a set of TIFF files.
download_and_build_vrt(image_uris, vrt_dir)	Download images (if needed) and build a VRT for a set of TIFF files.
fill_overflow(extent, window, chip[, fill_value])	Where chip's window overflows extent, fill with fill_value.
get_channel_order_from_dataset(image_dataset)	Get channel order from rasterio image dataset.
load_window(image_dataset[, bands, window, ...])	Load a window of an image using Rasterio.

build_vrt

build_vrt(*vrt_path: str*, *image_uris: List[str]*) → None

Build a VRT for a set of TIFF files.

Parameters

- **vrt_path** (*str*) – Local path for the VRT to be created.
- **image_uris** (*List[str]*) – Image URIs.

Return type
None

download_and_build_vrt

download_and_build_vrt(*image_uris: List[str]*, *vrt_dir: str*, *stream: bool = False*) → str

Download images (if needed) and build a VRT for a set of TIFF files.

Parameters

- **image_uris** (*List[str]*) – Image URIs.
- **vrt_dir** (*str*) – Dir where the VRT will be created.
- **stream** (*bool, optional*) – If true, do not download images. Defaults to False.

Returns
The path to the created VRT file.

Return type
str

fill_overflow

fill_overflow(*extent:* Box, *window:* Box, *chip:* *ndarray*, *fill_value:* *int = 0*) → ndarray

Where `chip`'s `window` overflows extent, fill with `fill_value`.

Parameters

- **extent** (Box) – Extent.
- **window** (Box) – Window from which `chip` was read.
- **chip** (*np.ndarray*) – (H, W, C) array.
- **fill_value** (*int, optional*) – Value to set oveflowing pixels to. Defaults to 0.

Returns
Chip with overflowing regions filled with `fill_value`.

Return type
np.ndarray

get_channel_order_from_dataset

get_channel_order_from_dataset(*image_dataset: DatasetReader*) → List[int]

Get channel order from rasterio image dataset.

Accounts for dataset's `colorinterp` if defined.

Parameters
image_dataset (*DatasetReader*) – Rasterio image dataset.

Returns
List of channel indices.

Return type
List[int]

load_window

load_window(*image_dataset: DatasetReader*, *bands: Optional[Union[int, Sequence[int]]] = None*, *window: Optional[Tuple[Tuple[int, int], Tuple[int, int]]] = None*, *is_masked: bool = False*, *out_shape: Optional[Tuple[int, ...]] = None*) → ndarray

Load a window of an image using Rasterio.

Parameters

- **image_dataset** (*DatasetReader*) – a Rasterio dataset.
- **bands** (*Optional[Union[int, Sequence[int]]]*) – Band index or indices to read. Must be 1-indexed.

- **window** (*Optional[Tuple[Tuple[int, int], Tuple[int, int]]]*) – ((row_start, row_stop), (col_start, col_stop)) or ((y_min, y_max), (x_min, x_max)). If None, reads the entire raster. Defaults to None.
- **is_masked** (*bool*) – If True, read a masked array from rasterio. Defaults to False.
- **out_shape** (*Optional[Tuple[int, int]]*) – (hieght, width) of the output chip. If None, no resizing is done. Defaults to None.

Returns
array of shape (height, width, channels).

Return type
np.ndarray

rasterio_source_config

Configs

RasterioSourceConfig	Configure a *RasterioSource*.

RasterioSourceConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model RasterioSourceConfig

Configure a *RasterioSource*.


```
{
   "title": "RasterioSourceConfig",
   "description": "Configure a :class:`.RasterioSource`.",
   "type": "object",
   "properties": {
      "channel_order": {
         "title": "Channel Order",
         "description": "The sequence of channel indices to use when reading␣
↪imagery.",
         "type": "array",
         "items": {
            "type": "integer"
         }
      },
      "transformers": {
         "title": "Transformers",
         "default": [],
         "type": "array",
         "items": {
            "$ref": "#/definitions/RasterTransformerConfig"
         }
      },
      "extent": {
```


(continues on next page)

(continued from previous page)

```
        "title": "Extent",
        "description": "Use-specified extent in pixel coords in the form (ymin,␣
↪xmin, ymax, xmax). Useful for cropping the raster source so that only part of the␣
↪raster is read from.",
        "type": "array",
        "minItems": 4,
        "maxItems": 4,
        "items": [
           {
              "type": "integer"
           },
           {
              "type": "integer"
           },
           {
              "type": "integer"
           },
           {
              "type": "integer"
           }
        ]
     },
     "type_hint": {
        "title": "Type Hint",
        "default": "rasterio_source",
        "enum": [
           "rasterio_source"
        ],
        "type": "string"
     },
     "uris": {
        "title": "Uris",
        "description": "One or more image URIs that comprise the imagery for a␣
↪scene. The format of each file can be any that can be read by Rasterio/GDAL. If >␣
↪1 URI is provided, a VRT will be created to mosaic together the individual images.
↪",
        "anyOf": [
           {
              "type": "string"
           },
           {
              "type": "array",
              "items": {
                 "type": "string"
              }
           }
        ]
     },
     "allow_streaming": {
        "title": "Allow Streaming",
        "description": "Stream assets as needed rather than downloading them.",
        "default": false,
```

(continues on next page)

(continued from previous page)


```
            "type": "boolean"
         }
      },
      "required": [
         "uris"
      ],
      "additionalProperties": false,
      "definitions": {
         "RasterTransformerConfig": {
            "title": "RasterTransformerConfig",
            "description": "Configure a :class:`.RasterTransformer`.",
            "type": "object",
            "properties": {
               "type_hint": {
                  "title": "Type Hint",
                  "default": "raster_transformer",
                  "enum": [
                     "raster_transformer"
                  ],
                  "type": "string"
               }
            },
            "additionalProperties": false
         }
      }
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *allow_streaming (bool)*
- *channel_order (Optional[List[int]])*
- *extent (Optional[Tuple[int, int, int, int]])*
- *transformers (List[rastervision.core.data.raster_transformer.raster_transformer_config.RasterTransformerConfig])*
- *type_hint (Literal['rasterio_source'])*
- *uris (Union[str, List[str]])*

Validators

- validate_extent » *extent*

field allow_streaming: bool = False

Stream assets as needed rather than downloading them.

field channel_order: Optional[List[int]] = None

The sequence of channel indices to use when reading imagery.

field extent: Optional[Tuple[int, int, int, int]] = None

Use-specified extent in pixel coords in the form (ymin, xmin, ymax, xmax). Useful for cropping the raster source so that only part of the raster is read from.

Validated by

- validate_extent

field transformers: List[*RasterTransformerConfig*] = []

field type_hint: Literal['rasterio_source'] = 'rasterio_source'

field uris: Union[str, List[str]] [Required]

One or more image URIs that comprise the imagery for a scene. The format of each file can be any that can be read by Rasterio/GDAL. If > 1 URI is provided, a VRT will be created to mosaic together the individual images.

build(*tmp_dir*, *use_transformers=True*)

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*pipeline=None*, *scene=None*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validator validate_extent » extent

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

rasterized_source

Classes

RasterizedSource	A *RasterSource* based on the rasterization of a VectorSource.

RasterizedSource

class RasterizedSource

Bases: *RasterSource*

A *RasterSource* based on the rasterization of a VectorSource.

Attributes

crs_transformer	Associated *CRSTransformer*.
dtype	`numpy.dtype` of the chips read from this source.
extent	Extent of the RasterSource.
num_channels	Number of channels in the chips read from this source.
shape	Shape of the raster as a (height, width, num_channels) tuple.

__init__(*vector_source:* VectorSource, *background_class_id:* *int*, *extent:* Box, *all_touched:* *bool* = *False*, *raster_transformers:* *List*[RasterTransformer] = *[]*)

Constructor.

Parameters

- **vector_source** (*VectorSource*) – The VectorSource to rasterize.
- **background_class_id** (*int*) – The class_id to use for any background pixels, ie. pixels not covered by a polygon.
- **extent** (*Box*) – Extent of corresponding imagery RasterSource.
- **all_touched** (*bool, optional*) – If True, all pixels touched by geometries will be burned in. If false, only pixels whose center is within the polygon or that are selected by Bresenham's line algorithm will be burned in. (See `rasterize()` for more details). Defaults to False.
- **raster_transformers** (*List*[*RasterTransformer*]) –

Methods

__init__(vector_source, background_class_id, ...)	Constructor.
get_chip(window)	Return the transformed chip in the window.
get_image_array()	Return entire transformed image array.
get_raw_chip(window)	Return raw chip without applying channel_order or transforms.
get_raw_image_array()	Return raw image for the full extent.
validate_labels(df)	Validate label geometries.

__init__(*vector_source:* VectorSource, *background_class_id: int*, *extent:* Box, *all_touched: bool = False*, *raster_transformers: List[*RasterTransformer*] = []*)

Constructor.

Parameters

- **vector_source** (*VectorSource*) – The VectorSource to rasterize.
- **background_class_id** (*int*) – The class_id to use for any background pixels, ie. pixels not covered by a polygon.
- **extent** (*Box*) – Extent of corresponding imagery RasterSource.
- **all_touched** (*bool, optional*) – If True, all pixels touched by geometries will be burned in. If false, only pixels whose center is within the polygon or that are selected by Bresenham's line algorithm will be burned in. (See *rasterize()* for more details). Defaults to False.
- **raster_transformers** (*List[RasterTransformer]*) –

get_chip(*window:* Box) → np.ndarray

Return the transformed chip in the window.

Get a raw chip, extract subset of channels using channel_order, and then apply transformations.

Parameters
window (*Box*) – The window for which to get the chip.

Returns
Array of shape (height, width, channels).

Return type
np.ndarray

get_image_array() → np.ndarray

Return entire transformed image array.

> **Warning:** Not safe to call on very large RasterSources.

Returns
Array of shape (height, width, channels).

Return type
np.ndarray

get_raw_chip(*window:* Box) → np.ndarray

Return raw chip without applying channel_order or transforms.

Parameters
window (Box) – The window for which to get the chip.

Returns
Array of shape (height, width, channels).

Return type
np.ndarray

get_raw_image_array() → np.ndarray

Return raw image for the full extent.

> **Warning:** Not safe to call on very large RasterSources.

Returns
Array of shape (height, width, channels).

Return type
np.ndarray

validate_labels(*df: geopandas.GeoDataFrame*) → None

Validate label geometries.

Parameters
df (*gpd.GeoDataFrame*) – Label geometries.

Raises

- **ValueError** – If `Point` or `LineString` geometries found.
- **ValueError** – If geometries are missing class IDs.

Return type
None

property crs_transformer

Associated *CRSTransformer*.

property dtype: dtype

`numpy.dtype` of the chips read from this source.

property extent: *Box*

Extent of the RasterSource.

property num_channels: int

Number of channels in the chips read from this source.

property shape: Tuple[int, int, int]

Shape of the raster as a (height, width, num_channels) tuple.

Functions

geoms_to_raster(df, window, ...)	Rasterize geometries that intersect with the window.

geoms_to_raster

geoms_to_raster(*df: geopandas.GeoDataFrame*, *window:* Box, *background_class_id: int*, *all_touched: bool*) → ndarray

Rasterize geometries that intersect with the window.

Parameters

- **df** (*gpd.GeoDataFrame*) – All label geometries in the scene.
- **window** (Box) – The part of the scene to rasterize.
- **background_class_id** (*int*) – Class ID to use for pixels that don't fall under any label geometry.
- **all_touched** (*bool*) – If True, all pixels touched by geometries will be burned in. If false, only pixels whose center is within the polygon or that are selected by Bresenham's line algorithm will be burned in. (See `rasterize()` for more details). Defaults to False.

Returns
A raster.

Return type
np.ndarray

rasterized_source_config

Configs

RasterizedSourceConfig	Configure a *RasterizedSource*.
RasterizerConfig	Configure rasterization params for a *RasterizedSource*.

RasterizedSourceConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model RasterizedSourceConfig

Configure a *RasterizedSource*.


```
{
   "title": "RasterizedSourceConfig",
   "description": "Configure a :class:`.RasterizedSource`.",
   "type": "object",
   "properties": {
      "vector_source": {
```


(continues on next page)

(continued from previous page)


```
            "$ref": "#/definitions/VectorSourceConfig"
        },
        "rasterizer_config": {
            "$ref": "#/definitions/RasterizerConfig"
        },
        "type_hint": {
            "title": "Type Hint",
            "default": "rasterized_source",
            "enum": [
                "rasterized_source"
            ],
            "type": "string"
        }
    },
    "required": [
        "vector_source",
        "rasterizer_config"
    ],
    "additionalProperties": false,
    "definitions": {
        "VectorTransformerConfig": {
            "title": "VectorTransformerConfig",
            "description": "Configure a :class:`.VectorTransformer`.",
            "type": "object",
            "properties": {
                "type_hint": {
                    "title": "Type Hint",
                    "default": "vector_transformer",
                    "enum": [
                        "vector_transformer"
                    ],
                    "type": "string"
                }
            },
            "additionalProperties": false
        },
        "VectorSourceConfig": {
            "title": "VectorSourceConfig",
            "description": "Configure a :class:`.VectorSource`.",
            "type": "object",
            "properties": {
                "transformers": {
                    "title": "Transformers",
                    "description": "List of VectorTransformers.",
                    "default": [],
                    "type": "array",
                    "items": {
                        "$ref": "#/definitions/VectorTransformerConfig"
                    }
                },
                "type_hint": {
                    "title": "Type Hint",
```


(continues on next page)

(continued from previous page)


```
                "default": "vector_source",
                "enum": [
                    "vector_source"
                ],
                "type": "string"
            }
        },
        "additionalProperties": false
    },
    "RasterizerConfig": {
        "title": "RasterizerConfig",
        "description": "Configure rasterization params for a :class:`.
↪RasterizedSource`.",
        "type": "object",
        "properties": {
            "background_class_id": {
                "title": "Background Class Id",
                "description": "The class_id to use for any background pixels, i.e.␣
↪pixels not covered by a polygon.",
                "type": "integer"
            },
            "all_touched": {
                "title": "All Touched",
                "description": "If True, all pixels touched by geometries will be␣
↪burned in. If false, only pixels whose center is within the polygon or that are␣
↪selected by Bresenham's line algorithm will be burned in. (See rasterio.features.
↪rasterize for more details).",
                "default": false,
                "type": "boolean"
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "rasterizer",
                "enum": [
                    "rasterizer"
                ],
                "type": "string"
            }
        },
        "required": [
            "background_class_id"
        ],
        "additionalProperties": false
    }
  }
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *rasterizer_config (rastervision.core.data.raster_source.rasterized_source_config.RasterizerConfig)*
- *type_hint (Literal['rasterized_source'])*
- *vector_source (rastervision.core.data.vector_source.vector_source_config.VectorSourceConfig)*

Validators

- *ensure_required_transformers* » *vector_source*

field rasterizer_config: *RasterizerConfig* [Required]

field type_hint: Literal['rasterized_source'] = 'rasterized_source'

field vector_source: *VectorSourceConfig* [Required]

Validated by

- *ensure_required_transformers*

build(*class_config*, *crs_transformer*, *extent*)

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

validator ensure_required_transformers » vector_source

Add class-inference and buffer transformers if absent.

Parameters
v (VectorSourceConfig) –

Return type
VectorSourceConfig

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*pipeline=None*, *scene=None*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

RasterizerConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model RasterizerConfig

Configure rasterization params for a *RasterizedSource*.


```
{
   "title": "RasterizerConfig",
   "description": "Configure rasterization params for a :class:`.RasterizedSource`.
↪",
   "type": "object",
   "properties": {
      "background_class_id": {
         "title": "Background Class Id",
         "description": "The class_id to use for any background pixels, i.e. pixels␣
↪not covered by a polygon.",
         "type": "integer"
      },
      "all_touched": {
         "title": "All Touched",
         "description": "If True, all pixels touched by geometries will be burned␣
↪in. If false, only pixels whose center is within the polygon or that are selected␣
↪by Bresenham's line algorithm will be burned in. (See rasterio.features.rasterize␣
↪for more details).",
         "default": false,
         "type": "boolean"
      },
      "type_hint": {
         "title": "Type Hint",
         "default": "rasterizer",
         "enum": [
            "rasterizer"
         ],
         "type": "string"
      }
   },
   "required": [
      "background_class_id"
   ],
```


(continues on next page)

(continued from previous page)

```
    "additionalProperties": false
}
```

Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *all_touched (bool)*
- *background_class_id (int)*
- *type_hint (Literal['rasterizer'])*

field all_touched: bool = False

If True, all pixels touched by geometries will be burned in. If false, only pixels whose center is within the polygon or that are selected by Bresenham's line algorithm will be burned in. (See rasterio.features.rasterize for more details).

field background_class_id: int [Required]

The class_id to use for any background pixels, i.e. pixels not covered by a polygon.

field type_hint: Literal['rasterizer'] = 'rasterizer'

build()

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(**args*, ***kwargs*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

raster_transformer

Modules

cast_transformer

Classes

CastTransformer	Casts chips to the specified dtype.

CastTransformer

class CastTransformer

Bases: *RasterTransformer*

Casts chips to the specified dtype.

__init__(*to_dtype: str*)

Constructor.

Parameters
to_dtype (*str*) – (str) dtype to cast the chips to.

Methods

__init__(to_dtype)	Constructor.
transform(chip[, channel_order])	Cast chip to self.to_dtype.

__init__(*to_dtype: str*)

Constructor.

Parameters
to_dtype (*str*) – (str) dtype to cast the chips to.

transform(*chip: ndarray*, *channel_order: Optional[list] = None*) → ndarray

Cast chip to self.to_dtype.

Parameters

- **chip** (*ndarray*) – ndarray of shape [height, width, channels]
- **channel_order** (*Optional[list]*) –

Returns
[height, width, channels] numpy array

Return type
ndarray

cast_transformer_config

Configs

CastTransformerConfig	Configure a *CastTransformer*.

CastTransformerConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model CastTransformerConfig

Configure a *CastTransformer*.


```
{
   "title": "CastTransformerConfig",
   "description": "Configure a :class:`.CastTransformer`.",
   "type": "object",
   "properties": {
      "type_hint": {
         "title": "Type Hint",
         "default": "cast_transformer",
         "enum": [
            "cast_transformer"
         ],
         "type": "string"
      },
      "to_dtype": {
         "title": "To Dtype",
         "description": "dtype to cast raster to. Must be a valid Numpy dtype e.g. \
↪"uint8\", \"float32\", etc.",
         "type": "string"
      }
   },
   "required": [
      "to_dtype"
   ],
   "additionalProperties": false
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *to_dtype (str)*
- *type_hint (Literal['cast_transformer'])*

field to_dtype: str [Required]

dtype to cast raster to. Must be a valid Numpy dtype e.g. "uint8", "float32", etc.

field type_hint: Literal['cast_transformer'] = 'cast_transformer'

build()

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*pipeline=None*, *scene=None*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

update_root(*root_dir*)

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

min_max_transformer

Classes

MinMaxTransformer	Transforms chips by scaling values in each channel to span 0-255.

MinMaxTransformer

class MinMaxTransformer

Bases: *RasterTransformer*

Transforms chips by scaling values in each channel to span 0-255.

__init__()

Methods

__init__()	
transform(chip[, channel_order])	Transform a chip of a raster source.

transform(*chip: ndarray*, *channel_order: Optional[List[int]] = None*) → ndarray

Transform a chip of a raster source.

Parameters

- **chip** (*ndarray*) – ndarray of shape [height, width, channels] This is assumed to already have the channel_order applied to it if channel_order is set. In other words, channels should be equal to len(channel_order).
- **channel_order** (*Optional[List[int]]*) – list of indices of channels that were extracted from the raw imagery.

Returns

[height, width, channels] numpy array

Return type

ndarray

min_max_transformer_config

Configs

MinMaxTransformerConfig	Configure a *MinMaxTransformer*.

MinMaxTransformerConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model MinMaxTransformerConfig

Configure a *MinMaxTransformer*.


```
{
   "title": "MinMaxTransformerConfig",
   "description": "Configure a :class:`.MinMaxTransformer`.",
```


(continues on next page)

(continued from previous page)

```
   "type": "object",
   "properties": {
      "type_hint": {
         "title": "Type Hint",
         "default": "min_max_transformer",
         "enum": [
            "min_max_transformer"
         ],
         "type": "string"
      }
   },
   "additionalProperties": false
}
```

Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *type_hint (Literal['min_max_transformer'])*

field type_hint: Literal['min_max_transformer'] = 'min_max_transformer'

build()

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*pipeline=None*, *scene=None*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

update_root(*root_dir*)

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

nan_transformer

Classes

NanTransformer	Removes NaN values from float raster.

NanTransformer

class NanTransformer

Bases: *RasterTransformer*

Removes NaN values from float raster.

__init__(*to_value: float = 0.0*)

Construct a new NanTransformer.

Parameters

to_value (*float*) – (float) NaN values are replaced with this

Methods

__init__([to_value])	Construct a new NanTransformer.
transform(chip[, channel_order])	Transform a chip.

__init__(*to_value: float = 0.0*)

Construct a new NanTransformer.

Parameters

to_value (*float*) – (float) NaN values are replaced with this

transform(*chip*, *channel_order=None*)

Transform a chip.

Removes NaN values.

Parameters

chip – ndarray of shape [height, width, channels] This is assumed to already have the channel_order applied to it if channel_order is set. In other words, channels should be equal to len(channel_order).

Returns

[height, width, channels] numpy array

nan_transformer_config

Configs

NanTransformerConfig	Configure a *NanTransformer*.

NanTransformerConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model NanTransformerConfig

Configure a *NanTransformer*.


```
{
   "title": "NanTransformerConfig",
   "description": "Configure a :class:`.NanTransformer`.",
   "type": "object",
   "properties": {
      "type_hint": {
         "title": "Type Hint",
         "default": "nan_transformer",
         "enum": [
            "nan_transformer"
         ],
         "type": "string"
      },
      "to_value": {
         "title": "To Value",
         "description": "Turn all NaN values into this value.",
         "default": 0.0,
         "type": "number"
      }
   },
   "additionalProperties": false
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *to_value (Optional[float])*
- *type_hint (Literal['nan_transformer'])*

field to_value: Optional[float] = 0.0

Turn all NaN values into this value.

field type_hint: Literal['nan_transformer'] = 'nan_transformer'

build()

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*pipeline=None*, *scene=None*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

update_root(*root_dir*)

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

raster_transformer

Classes

RasterTransformer	Transforms raw chips to be input to a neural network.

RasterTransformer

class RasterTransformer

Bases: ABC

Transforms raw chips to be input to a neural network.

__init__()

Methods

__init__()	
transform(chip[, channel_order])	Transform a chip of a raster source.

abstract transform(*chip*, *channel_order=None*)

Transform a chip of a raster source.

Parameters

- **chip** – ndarray of shape [height, width, channels] This is assumed to already have the channel_order applied to it if channel_order is set. In other words, channels should be equal to len(channel_order).
- **channel_order** – list of indices of channels that were extracted from the raw imagery.

Returns

[height, width, channels] numpy array

raster_transformer_config

Configs

RasterTransformerConfig	Configure a *RasterTransformer*.

RasterTransformerConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model RasterTransformerConfig

Configure a *RasterTransformer*.


```
{
   "title": "RasterTransformerConfig",
   "description": "Configure a :class:`.RasterTransformer`.",
   "type": "object",
   "properties": {
      "type_hint": {
         "title": "Type Hint",
```


(continues on next page)

(continued from previous page)

```
            "default": "raster_transformer",
            "enum": [
               "raster_transformer"
            ],
            "type": "string"
         }
      },
      "additionalProperties": false
}
```

Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *type_hint (Literal['raster_transformer'])*

field type_hint: Literal['raster_transformer'] = 'raster_transformer'

build()

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*pipeline=None*, *scene=None*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

update_root(*root_dir*)

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate

- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

reclass_transformer

Classes

ReclassTransformer	Maps class IDs in a label raster to other values.

ReclassTransformer

class ReclassTransformer

Bases: *RasterTransformer*

Maps class IDs in a label raster to other values.

__init__(*mapping: Dict[int, int]*)

Construct a new ReclassTransformer.

Parameters

mapping (*Dict[int, int]*) – (dict) Remapping dictionary

Methods

__init__(mapping)	Construct a new ReclassTransformer.
transform(chip[, channel_order])	Transform a chip.

__init__(*mapping: Dict[int, int]*)

Construct a new ReclassTransformer.

Parameters

mapping (*Dict[int, int]*) – (dict) Remapping dictionary

transform(*chip: np.ndarray*, *channel_order: Optional[List[int]] = None*)

Transform a chip.

Reclassify a label raster using the given mapping.

Parameters

- **chip** (*np.ndarray*) – ndarray of shape [height, width, channels] This is assumed to already have the channel_order applied to it if channel_order is set. In other words, channels should be equal to len(channel_order).
- **channel_order** (*Optional[List[int]]*) – list of indices of channels that were extracted from the raw imagery.

Returns

[height, width, channels] numpy array

reclass_transformer_config

Configs

ReclassTransformerConfig	Configure a *ReclassTransformer*.

ReclassTransformerConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model ReclassTransformerConfig

Configure a *ReclassTransformer*.


```
{
   "title": "ReclassTransformerConfig",
   "description": "Configure a :class:`.ReclassTransformer`.",
   "type": "object",
   "properties": {
      "type_hint": {
         "title": "Type Hint",
         "default": "reclass_transformer",
         "enum": [
            "reclass_transformer"
         ],
         "type": "string"
      },
      "mapping": {
         "title": "Mapping",
         "description": "The reclassification mapping.",
         "type": "object",
         "additionalProperties": {
            "type": "integer"
         }
      }
   },
   "required": [
      "mapping"
   ],
   "additionalProperties": false
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *mapping (Dict[int, int])*
- *type_hint (Literal['reclass_transformer'])*

field mapping: Dict[int, int] [Required]

The reclassification mapping.

field type_hint: Literal['reclass_transformer'] = 'reclass_transformer'

build() → *ReclassTransformer*

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

Return type
ReclassTransformer

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*pipeline=None*, *scene=None*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

update_root(*root_dir*)

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises
ConfigError – if field is invalid

rgb_class_transformer

Classes

RGBClassTransformer	Maps RGB values to class IDs.

RGBClassTransformer

class RGBClassTransformer

Bases: *RasterTransformer*

Maps RGB values to class IDs. Can also do the reverse.

__init__(*class_config:* ClassConfig)

Parameters
class_config (ClassConfig) –

Methods

__init__(class_config)	
class_to_rgb(class_labels)	
rgb_to_class(array_rgb)	
transform(chip[, channel_order])	Transform RGB array to array of class IDs or vice versa.

__init__(*class_config:* ClassConfig)

Parameters
class_config (ClassConfig) –

class_to_rgb(*class_labels: ndarray*) → ndarray

Parameters
class_labels (*ndarray*) –

Return type
ndarray

rgb_to_class(*array_rgb: ndarray*) → ndarray

Parameters
array_rgb (*ndarray*) –

Return type
ndarray

transform(*chip: ndarray*, *channel_order: Optional[List[int]] = None*) → ndarray

Transform RGB array to array of class IDs or vice versa.

Parameters

- **chip** (*np.ndarray*) – Numpy array of shape (H, W, 3).
- **channel_order** (*Optional[List[int]], optional*) – List of indices of channels that were extracted from the raw imagery. Defaults to None.

Returns
An array of class IDs.

Return type
np.ndarray

rgb_class_transformer_config

Configs

RGBClassTransformerConfig	Configure a *RGBClassTransformer*.

RGBClassTransformerConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model RGBClassTransformerConfig

Configure a *RGBClassTransformer*.


```
{
   "title": "RGBClassTransformerConfig",
   "description": "Configure a :class:`.RGBClassTransformer`.",
   "type": "object",
   "properties": {
      "type_hint": {
         "title": "Type Hint",
         "default": "rgb_class_transformer",
         "enum": [
            "rgb_class_transformer"
         ],
         "type": "string"
      },
      "class_config": {
         "title": "Class Config",
         "description": "The class config defining the mapping between classes and colors.",
         "allOf": [
            {
               "$ref": "#/definitions/ClassConfig"
            }
         ]
      }
   },
   "required": [
      "class_config"
```


(continues on next page)

(continued from previous page)


```
  ],
  "additionalProperties": false,
  "definitions": {
     "ClassConfig": {
        "title": "ClassConfig",
        "description": "Configure class information for a machine learning task.",
        "type": "object",
        "properties": {
           "names": {
              "title": "Names",
              "description": "Names of classes. The i-th class in this list will␣
↪have class ID = i.",
              "type": "array",
              "items": {
                 "type": "string"
              }
           },
           "colors": {
              "title": "Colors",
              "description": "Colors used to visualize classes. Can be color␣
↪strings accepted by matplotlib or RGB tuples. If None, a random color will be␣
↪auto-generated for each class.",
              "type": "array",
              "items": {
                 "anyOf": [
                    {
                       "type": "string"
                    },
                    {
                       "type": "array",
                       "items": {}
                    }
                 ]
              }
           },
           "null_class": {
              "title": "Null Class",
              "description": "Optional name of class in `names` to use as the null␣
↪class. This is used in semantic segmentation to represent the label for imagery␣
↪pixels that are NODATA or that are missing a label. If None and the class names␣
↪include \"null\", it will automatically be used as the null class. If None, and␣
↪this Config is part of a SemanticSegmentationConfig, a null class will be added␣
↪automatically.",
              "type": "string"
           },
           "type_hint": {
              "title": "Type Hint",
              "default": "class_config",
              "enum": [
                 "class_config"
              ],
              "type": "string"
```


(continues on next page)

(continued from previous page)

```
            }
         },
         "required": [
            "names"
         ],
         "additionalProperties": false
      }
   }
}
```

Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *class_config (rastervision.core.data.class_config.ClassConfig)*
- *type_hint (Literal['rgb_class_transformer'])*

field class_config: *ClassConfig* [Required]

The class config defining the mapping between classes and colors.

field type_hint: Literal['rgb_class_transformer'] = 'rgb_class_transformer'

build() → *RGBClassTransformer*

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

Return type
RGBClassTransformer

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*pipeline=None*, *scene=None*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

update_root(*root_dir*)

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

stats_transformer

Classes

StatsTransformer	Transforms non-uint8 to uint8 values using channel statistics.

StatsTransformer

class StatsTransformer

Bases: *RasterTransformer*

Transforms non-uint8 to uint8 values using channel statistics.

This works as follows:

- Convert pixel values to z-scores using channel means and standard deviations.
- Clip z-scores to the specified number of standard deviations (default 3) on each side.
- Scale values to 0-255 and cast to uint8.

This transformation is not applied to NODATA pixels (assumed to be pixels with all values equal to zero).

__init__(*means: Sequence[float]*, *stds: Sequence[float]*, *max_stds: float = 3.0*)

Construct a new StatsTransformer.

Parameters

- **means** (*np.ndarray*) – Channel means.
- **means** – Channel standard deviations.
- **max_stds** (*float*) – Number of standard deviations to clip the distribution to on both sides. Defaults to 3.
- **stds** (*Sequence[float]*) –

Methods

__init__(means, stds[, max_stds])	Construct a new StatsTransformer.
from_raster_sources(raster_sources[, ...])	Create a StatsTransformer with stats from the given raster sources.
transform(chip[, channel_order])	Transform a chip.

__init__(*means: Sequence[float]*, *stds: Sequence[float]*, *max_stds: float = 3.0*)

Construct a new StatsTransformer.

Parameters

- **means** (*np.ndarray*) – Channel means.
- **means** – Channel standard deviations.
- **max_stds** (*float*) – Number of standard deviations to clip the distribution to on both sides. Defaults to 3.
- **stds** (*Sequence[float]*) –

classmethod from_raster_sources(*raster_sources: List[*RasterSource*]*, *sample_prob: float = 0.1*, *max_stds: float = 3.0*) → *StatsTransformer*

Create a StatsTransformer with stats from the given raster sources.

Parameters

- **raster_sources** (*List['RasterSource']*) – List of raster sources to compute stats from.
- **sample_prob** (*float, optional*) – Fraction of each raster to sample for computing stats. For details see docs for RasterStats.compute(). Defaults to 0.1.
- **max_stds** (*float, optional*) – Number of standard deviations to clip the distribution to on both sides. Defaults to 3.

Returns

A StatsTransformer.

Return type

StatsTransformer

transform(*chip: ndarray*, *channel_order: Optional[Sequence[int]] = None*) → ndarray

Transform a chip.

Transforms non-uint8 to uint8 values using raster_stats.

Parameters

- **chip** (*ndarray*) – ndarray of shape [height, width, channels] This is assumed to already have the channel_order applied to it if channel_order is set. In other words, channels should be equal to len(channel_order).
- **channel_order** (*Optional[Sequence[int]]*) – list of indices of channels that were extracted from the raw imagery.

Returns

[height, width, channels] uint8 numpy array

Return type

ndarray

stats_transformer_config

Configs

StatsTransformerConfig	Configure a *StatsTransformer*.

StatsTransformerConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model StatsTransformerConfig

Configure a *StatsTransformer*.


```
{
   "title": "StatsTransformerConfig",
   "description": "Configure a :class:`.StatsTransformer`.",
   "type": "object",
   "properties": {
      "type_hint": {
         "title": "Type Hint",
         "default": "stats_transformer",
         "enum": [
            "stats_transformer"
         ],
         "type": "string"
      },
      "stats_uri": {
         "title": "Stats Uri",
         "description": "The URI of the output of the StatsAnalyzer. If None, and␣
↪this Config is inside an RVPipeline, this field will be auto-generated.",
         "type": "string"
      },
      "scene_group": {
         "title": "Scene Group",
         "description": "Name of the group of scenes whose stats to use. Defaultsto␣
↪\"train_scenes\".",
         "default": "train_scenes",
         "type": "string"
      }
   },
   "additionalProperties": false
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *scene_group (str)*

- *stats_uri (Optional[str])*
- *type_hint (Literal['stats_transformer'])*

field scene_group: str = 'train_scenes'

Name of the group of scenes whose stats to use. Defaultsto "train_scenes".

field stats_uri: Optional[str] = None

The URI of the output of the StatsAnalyzer. If None, and this Config is inside an RVPipeline, this field will be auto-generated.

field type_hint: Literal['stats_transformer'] = 'stats_transformer'

build()

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*pipeline: Optional[RVPipelineConfig] = None*, *scene: Optional[SceneConfig] = None*) → None

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

Parameters

- **pipeline** (*Optional[RVPipelineConfig]*) –
- **scene** (*Optional[SceneConfig]*) –

Return type
None

update_root(*root_dir: str*) → None

Parameters
root_dir (*str*) –

Return type
None

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

scene

Classes

Scene	The raster data and labels associated with an area of interest.

Scene

class Scene

Bases: object

The raster data and labels associated with an area of interest.

Attributes

extent	Extent of the associated *RasterSource*.

__init__(*id: str*, *raster_source: RasterSource*, *label_source: Optional[LabelSource] = None*, *label_store: Optional[LabelStore] = None*, *aoi_polygons: Optional[list] = None*)

Construct a new Scene.

Parameters

- **id** (*str*) – ID for this scene
- **raster_source** (*RasterSource*) – RasterSource for this scene
- **ground_truth_label_store** – optional LabelSource
- **label_store** (*Optional[LabelStore]*) – optional LabelStore
- **aoi** – Optional list of AOI polygons in pixel coordinates
- **label_source** (*Optional[LabelSource]*) –
- **aoi_polygons** (*Optional[list]*) –

Methods

__init__(id, raster_source[, label_source, ...])	Construct a new Scene.

__init__(*id: str*, *raster_source:* RasterSource, *label_source: Optional[*LabelSource*] = None*, *label_store: Optional[*LabelStore*] = None*, *aoi_polygons: Optional[list] = None*)

Construct a new Scene.

Parameters

- **id** (*str*) – ID for this scene
- **raster_source** (RasterSource) – RasterSource for this scene
- **ground_truth_label_store** – optional LabelSource
- **label_store** (*Optional[*LabelStore*]*) – optional LabelStore
- **aoi** – Optional list of AOI polygons in pixel coordinates
- **label_source** (*Optional[*LabelSource*]*) –
- **aoi_polygons** (*Optional[list]*) –

property extent: *Box*

Extent of the associated *RasterSource*.

scene_config

Configs

SceneConfig	Configure a *Scene* comprising raster data & labels for an AOI.

SceneConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model SceneConfig

Configure a *Scene* comprising raster data & labels for an AOI.


```
{
   "title": "SceneConfig",
   "description": "Configure a :class:`.Scene` comprising raster data & labels for␣
↪an AOI.\n    ",
   "type": "object",
   "properties": {
      "id": {
         "title": "Id",
         "type": "string"
      },
      "raster_source": {
```


(continues on next page)

(continued from previous page)


```
            "$ref": "#/definitions/RasterSourceConfig"
         },
         "label_source": {
            "$ref": "#/definitions/LabelSourceConfig"
         },
         "label_store": {
            "$ref": "#/definitions/LabelStoreConfig"
         },
         "aoi_uris": {
            "title": "Aoi Uris",
            "description": "List of URIs of GeoJSON files that define the AOIs for the␣
↪scene. Each polygon defines an AOI which is a piece of the scene that is assumed␣
↪to be fully labeled and usable for training or validation. The AOIs are assumed␣
↪to be in EPSG:4326 coordinates.",
            "type": "array",
            "items": {
               "type": "string"
            }
         },
         "type_hint": {
            "title": "Type Hint",
            "default": "scene",
            "enum": [
               "scene"
            ],
            "type": "string"
         }
      },
      "required": [
         "id",
         "raster_source"
      ],
      "additionalProperties": false,
      "definitions": {
         "RasterTransformerConfig": {
            "title": "RasterTransformerConfig",
            "description": "Configure a :class:`.RasterTransformer`.",
            "type": "object",
            "properties": {
               "type_hint": {
                  "title": "Type Hint",
                  "default": "raster_transformer",
                  "enum": [
                     "raster_transformer"
                  ],
                  "type": "string"
               }
            },
            "additionalProperties": false
         },
         "RasterSourceConfig": {
            "title": "RasterSourceConfig",
```


(continues on next page)

(continued from previous page)


```
        "description": "Configure a :class:`.RasterSource`.",
        "type": "object",
        "properties": {
           "channel_order": {
              "title": "Channel Order",
              "description": "The sequence of channel indices to use when reading␣
↪imagery.",
              "type": "array",
              "items": {
                 "type": "integer"
              }
           },
           "transformers": {
              "title": "Transformers",
              "default": [],
              "type": "array",
              "items": {
                 "$ref": "#/definitions/RasterTransformerConfig"
              }
           },
           "extent": {
              "title": "Extent",
              "description": "Use-specified extent in pixel coords in the form␣
↪(ymin, xmin, ymax, xmax). Useful for cropping the raster source so that only part␣
↪of the raster is read from.",
              "type": "array",
              "minItems": 4,
              "maxItems": 4,
              "items": [
                 {
                    "type": "integer"
                 },
                 {
                    "type": "integer"
                 },
                 {
                    "type": "integer"
                 },
                 {
                    "type": "integer"
                 }
              ]
           },
           "type_hint": {
              "title": "Type Hint",
              "default": "raster_source",
              "enum": [
                 "raster_source"
              ],
              "type": "string"
           }
        },
```


(continues on next page)

(continued from previous page)


```
            "additionalProperties": false
        },
        "LabelSourceConfig": {
            "title": "LabelSourceConfig",
            "description": "Configure a :class:`.LabelSource`.",
            "type": "object",
            "properties": {
                "type_hint": {
                    "title": "Type Hint",
                    "default": "label_source",
                    "enum": [
                        "label_source"
                    ],
                    "type": "string"
                }
            },
            "additionalProperties": false
        },
        "LabelStoreConfig": {
            "title": "LabelStoreConfig",
            "description": "Configure a :class:`.LabelStore`.",
            "type": "object",
            "properties": {
                "type_hint": {
                    "title": "Type Hint",
                    "default": "label_store",
                    "enum": [
                        "label_store"
                    ],
                    "type": "string"
                }
            },
            "additionalProperties": false
        }
    }
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *aoi_uris (Optional[List[str]])*
- *id (str)*
- *label_source (Optional[rastervision.core.data.label_source.label_source_config.LabelSourceConfig])*
- *label_store (Optional[rastervision.core.data.label_store.label_store_config.LabelStoreConfig])*

- *raster_source (rastervision.core.data.raster_source.raster_source_config.RasterSourceConfig)*
- *type_hint (Literal['scene'])*

field aoi_uris: Optional[List[str]] = None

List of URIs of GeoJSON files that define the AOIs for the scene. Each polygon defines an AOI which is a piece of the scene that is assumed to be fully labeled and usable for training or validation. The AOIs are assumed to be in EPSG:4326 coordinates.

field id: str [Required]

field label_source: Optional[*LabelSourceConfig*] = None

field label_store: Optional[*LabelStoreConfig*] = None

field raster_source: *RasterSourceConfig* [Required]

field type_hint: Literal['scene'] = 'scene'

build(*class_config*, *tmp_dir*, *use_transformers=True*) → *Scene*

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

Return type
Scene

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*pipeline=None*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises
ConfigError – if field is invalid

utils

Modules

factory
geojson
misc
vectorization

factory

Functions

make_cc_scene(image_uri[, label_vector_uri, ...])	Create a chip classification scene from image and label URIs.
make_od_scene(image_uri[, label_vector_uri, ...])	Create an object detection scene from image and label URIs.
make_ss_scene(image_uri[, label_raster_uri, ...])	Create a semantic segmentation scene from image and label URIs.

make_cc_scene

make_cc_scene(*image_uri: Union[str, List[str]]*, *label_vector_uri: Optional[str] = None*, *class_config: Optional[*ClassConfig*] = None*, *aoi_uri: Union[str, List[str]] = []*, *label_vector_default_class_id: Optional[int] = None*, *image_raster_source_kw: dict = {}*, *label_vector_source_kw: dict = {}*, *label_source_kw: dict = {}*, *scene_id: Optional[str] = None*) → *Scene*

Create a chip classification scene from image and label URIs.

This is a convenience method. For more fine-grained control, it is recommended to use the default constructor.

Parameters

- **image_uri** (*Union[str, List[str]]*) – URI or list of URIs of GeoTIFFs to use as the source of image data.
- **label_vector_uri** (*Optional[str], optional*) – URI of GeoJSON file to use as the source of segmentation label data. Defaults to None.
- **class_config** (*Optional[*ClassConfig*]*) – The ClassConfig. Must be non-None if creating a scene without a LabelSource. Defaults to None.
- **aoi_uri** (*Union[str, List[str]], optional*) – URI or list of URIs of GeoJSONs that specify the area-of-interest. If provided, the dataset will only access data from this area. Defaults to [].

- **label_vector_default_class_id** (*Optional[int], optional*) – If using label_vector_uri and all polygons in that file belong to the same class and they do not contain a *class_id* property, then use this argument to map all of the polgons to the appropriate class ID. See docs for ClassInferenceTransformer for more details. Defaults to None.
- **image_raster_source_kw** (*dict, optional*) – Additional arguments to pass to the RasterioSource used for image data. See docs for RasterioSource for more details. Defaults to {}.
- **label_vector_source_kw** (*dict, optional*) – Additional arguments to pass to the GeoJSONVectorSourceConfig used for label data, if label_vector_uri is set. See docs for GeoJSONVectorSourceConfig for more details. Defaults to {}.
- **label_source_kw** (*dict, optional*) – Additional arguments to pass to the ChipClassificationLabelSourceConfig used for label data, if label_vector_uri is set. See docs for ChipClassificationLabelSourceConfig for more details. Defaults to {}.
- **scene_id** (*Optional[str]*) – Optional scene ID. If None, will be randomly generated. Defaults to None.

Returns
A chip classification scene.

Return type
Scene

make_od_scene

make_od_scene(*image_uri: Union[str, List[str]]*, *label_vector_uri: Optional[str] = None*, *class_config: Optional[ClassConfig] = None*, *aoi_uri: Union[str, List[str]] = []*, *label_vector_default_class_id: Optional[int] = None*, *image_raster_source_kw: dict = {}*, *label_vector_source_kw: dict = {}*, *label_source_kw: dict = {}*, *scene_id: Optional[str] = None*) → *Scene*

Create an object detection scene from image and label URIs.

This is a convenience method. For more fine-grained control, it is recommended to use the default constructor.

Parameters

- **image_uri** (*Union[str, List[str]]*) – URI or list of URIs of GeoTIFFs to use as the source of image data.
- **label_vector_uri** (*Optional[str], optional*) – URI of GeoJSON file to use as the source of segmentation label data. Defaults to None.
- **class_config** (*Optional[ClassConfig]*) – The ClassConfig. Must be non-None if creating a scene without a LabelSource. Defaults to None.
- **aoi_uri** (*Union[str, List[str]], optional*) – URI or list of URIs of GeoJSONs that specify the area-of-interest. If provided, the dataset will only access data from this area. Defaults to [].
- **label_vector_default_class_id** (*Optional[int], optional*) – If using label_vector_uri and all polygons in that file belong to the same class and they do not contain a *class_id* property, then use this argument to map all of the polgons to the appropriate class ID. See docs for ClassInferenceTransformer for more details. Defaults to None.
- **image_raster_source_kw** (*dict, optional*) – Additional arguments to pass to the RasterioSource used for image data. See docs for RasterioSource for more details. Defaults to {}.

- **label_vector_source_kw** (*dict, optional*) – Additional arguments to pass to the GeoJSONVectorSourceConfig used for label data, if label_vector_uri is set. See docs for GeoJSONVectorSourceConfig for more details. Defaults to {}.
- **label_source_kw** (*dict, optional*) – Additional arguments to pass to the ObjectDetectionLabelSourceConfig used for label data, if label_vector_uri is set. See docs for ObjectDetectionLabelSourceConfig for more details. Defaults to {}.
- **scene_id** (*Optional[str]*) – Optional scene ID. If None, will be randomly generated. Defaults to None.

Returns
An object detection scene.

Return type
Scene

make_ss_scene

make_ss_scene(*image_uri: Union[str, List[str]]*, *label_raster_uri: Optional[Union[str, List[str]]] = None*, *class_config: Optional[ClassConfig] = None*, *label_vector_uri: Optional[str] = None*, *aoi_uri: Union[str, List[str]] = []*, *label_vector_default_class_id: Optional[int] = None*, *image_raster_source_kw: dict = {}*, *label_raster_source_kw: dict = {}*, *label_vector_source_kw: dict = {}*, *scene_id: Optional[str] = None*) → *Scene*

Create a semantic segmentation scene from image and label URIs.

This is a convenience method. For more fine-grained control, it is recommended to use the default constructor.

Parameters

- **image_uri** (*Union[str, List[str]]*) – URI or list of URIs of GeoTIFFs to use as the source of image data.
- **label_raster_uri** (*Optional[Union[str, List[str]]], optional*) – URI or list of URIs of GeoTIFFs to use as the source of segmentation label data. If the labels are in the form of GeoJSONs, use label_vector_uri instead. Defaults to None.
- **label_vector_uri** (*Optional[str], optional*) – URI of GeoJSON file to use as the source of segmentation label data. If the labels are in the form of GeoTIFFs, use label_raster_uri instead. Defaults to None.
- **class_config** (*Optional[ClassConfig]*) – The ClassConfig. Must be non-None if creating a scene without a LabelSource. Defaults to None.
- **aoi_uri** (*Union[str, List[str]], optional*) – URI or list of URIs of GeoJSONs that specify the area-of-interest. If provided, the dataset will only access data from this area. Defaults to [].
- **label_vector_default_class_id** (*Optional[int], optional*) – If using label_vector_uri and all polygons in that file belong to the same class and they do not contain a *class_id* property, then use this argument to map all of the polgons to the appropriate class ID. See docs for ClassInferenceTransformer for more details. Defaults to None.
- **image_raster_source_kw** (*dict, optional*) – Additional arguments to pass to the RasterioSource used for image data. See docs for RasterioSource for more details. Defaults to {}.
- **label_raster_source_kw** (*dict, optional*) – Additional arguments to pass to the RasterioSource used for label data, if label_raster_uri is used. See docs for RasterioSource for more details. Defaults to {}.

- **label_vector_source_kw** (*dict, optional*) – Additional arguments to pass to the GeoJSONVectorSource used for label data, if label_vector_uri is used. See docs for GeoJSONVectorSource for more details. Defaults to {}.
- **scene_id** (*Optional[str]*) – Optional scene ID. If None, will be randomly generated. Defaults to None.

Raises
ValueError – If both label_raster_uri and label_vector_uri are specified.

Returns
A semantic segmentation scene.

Return type
Scene

geojson

Functions

all_geoms_valid(geojson)	Check if there are any invalid geometries in the GeoJSON.
buffer_geoms(geojson, geom_type[, ...])	Buffer geometries.
features_to_geojson(features)	Convert GeoJSON-like mapping of Features to a FeatureCollection.
filter_features(func, geojson[, progressbar_kw])	Filter GeoJSON features.
geojson_to_geoms(geojson)	Return the shapely geometry for each feature in the GeoJSON.
geom_to_feature(geom[, properties])	Serialize a single shapely geomety to a GeoJSON Feature.
geometries_to_geojson(geometries)	Convert serialized geometries to a serialized GeoJSON FeatureCollection.
geometry_to_feature(mapping[, properties])	Convert a serialized geometry to a serialized GeoJSON feature.
geoms_to_geojson(geoms[, properties])	Serialize shapely geometries to GeoJSON.
get_polygons_from_uris(uris, crs_transformer)	Load and return polygons (in pixel coords) from one or more URIs.
is_empty_feature(f)	Check if a GeoJSON Feature lacks geometry info.
map_features(func, geojson[, ...])	Map GeoJSON features to new features.
map_geoms(func, geojson[, ...])	Map GeoJSON features to new features by applying func to geometries.
map_to_pixel_coords(geojson, crs_transformer)	Convert a GeoJSON dict from map to pixel coordinates.
pixel_to_map_coords(geojson, crs_transformer)	Convert a GeoJSON dict from pixel to map coordinates.
remove_empty_features(geojson)	Remove Features from a FeatureCollection that lack geometry info.
simplify_polygons(geojson)	Simplify polygon geometries by applying `.buffer(0)`.
split_multi_geometries(geojson)	Split any Features with multi-part geometries into multiple Features.

all_geoms_valid

all_geoms_valid(*geojson: dict*)

Check if there are any invalid geometries in the GeoJSON.

Parameters
geojson (*dict*) –

buffer_geoms

buffer_geoms(*geojson: dict*, *geom_type: str*, *class_bufs: Dict[int, Optional[float]] = {}*, *default_buf: Optional[float] = 1*) → dict

Buffer geometries.

Geometries in features without a class_id property will be ignored.

Parameters

- **geojson** (*dict*) – A GeoJSON-like mapping of a FeatureCollection.
- **geom_type** (*str*) – Shapely geometry type to apply the buffering to. Other types of geometries will not be affected.
- **class_bufs** (*Dict[int, Optional[float]]*) – Optional mapping from class ID to buffer distance (in pixel units) for geom_type geometries.
- **default_buf** (*Optional[float]*) –

Returns
FeatureCollection with buffered geometries.

Return type
dict

features_to_geojson

features_to_geojson(*features: List[dict]*) → dict

Convert GeoJSON-like mapping of Features to a FeatureCollection.

Parameters
features (*List[dict]*) –

Return type
dict

filter_features

filter_features(*func: Callable*, *geojson: dict*, *progressbar_kw: Optional[dict] = None*) → dict

Filter GeoJSON features. Returns a new GeoJSON dict.

Parameters

- **func** (*Callable*) –
- **geojson** (*dict*) –
- **progressbar_kw** (*Optional[dict]*) –

Return type
dict

geojson_to_geoms

geojson_to_geoms(*geojson: dict*) → Iterator[BaseGeometry]

Return the shapely geometry for each feature in the GeoJSON.

Parameters
geojson (*dict*) –

Return type
Iterator[BaseGeometry]

geom_to_feature

geom_to_feature(*geom: BaseGeometry*, *properties: Optional[dict] = None*) → dict

Serialize a single shapely geomety to a GeoJSON Feature.

Parameters

- **geom** (*BaseGeometry*) –
- **properties** (*Optional[dict]*) –

Return type
dict

geometries_to_geojson

geometries_to_geojson(*geometries: Iterable[dict]*) → dict

Convert serialized geometries to a serialized GeoJSON FeatureCollection.

Parameters
geometries (*Iterable[dict]*) –

Return type
dict

geometry_to_feature

geometry_to_feature(*mapping: dict*, *properties: Optional[dict] = None*) → dict

Convert a serialized geometry to a serialized GeoJSON feature.

Parameters

- **mapping** (*dict*) –
- **properties** (*Optional[dict]*) –

Return type
dict

geoms_to_geojson

geoms_to_geojson(*geoms: Iterable[BaseGeometry]*, *properties: Optional[Iterable[dict]] = None*) → dict

Serialize shapely geometries to GeoJSON.

Parameters

- **geoms** (*Iterable[BaseGeometry]*) –
- **properties** (*Optional[Iterable[dict]]*) –

Return type
dict

get_polygons_from_uris

get_polygons_from_uris(*uris: Union[str, List[str]]*, *crs_transformer:* CRSTransformer) → List[BaseGeometry]

Load and return polygons (in pixel coords) from one or more URIs.

Parameters

- **uris** (*Union[str, List[str]]*) –
- **crs_transformer** (*CRSTransformer*) –

Return type
List[BaseGeometry]

is_empty_feature

is_empty_feature(*f: dict*) → bool

Check if a GeoJSON Feature lacks geometry info.

This was added to handle empty geoms which appear when using OSM vector tiles.

Parameters
f (*dict*) – A GeoJSON-like mapping of a Feature.

Returns
Whether the feature contains any geometry.

Return type
bool

map_features

map_features(*func: Callable*, *geojson: dict*, *include_geom_types: Iterable[str] = []*, *progressbar_kw: Optional[dict] = None*) → dict

Map GeoJSON features to new features. Returns a new GeoJSON dict.

Parameters

- **func** (*Callable*) –
- **geojson** (*dict*) –
- **include_geom_types** (*Iterable[str]*) –

- **progressbar_kw** (*Optional[dict]*) –

Return type
dict

map_geoms

map_geoms(*func: Callable*, *geojson: dict*, *include_geom_types: Iterable[str] = []*, *progressbar_kw: Optional[dict] = None*) → dict

Map GeoJSON features to new features by applying func to geometries.

Returns a new GeoJSON dict.

Parameters

- **func** (*Callable*) –
- **geojson** (*dict*) –
- **include_geom_types** (*Iterable[str]*) –
- **progressbar_kw** (*Optional[dict]*) –

Return type
dict

map_to_pixel_coords

map_to_pixel_coords(*geojson: dict*, *crs_transformer:* CRSTransformer) → dict

Convert a GeoJSON dict from map to pixel coordinates.

Parameters

- **geojson** (*dict*) –
- **crs_transformer** (CRSTransformer) –

Return type
dict

pixel_to_map_coords

pixel_to_map_coords(*geojson: dict*, *crs_transformer:* CRSTransformer) → dict

Convert a GeoJSON dict from pixel to map coordinates.

Parameters

- **geojson** (*dict*) –
- **crs_transformer** (CRSTransformer) –

Return type
dict

remove_empty_features

remove_empty_features(*geojson: dict*) → dict

Remove Features from a FeatureCollection that lack geometry info.

Parameters
geojson (*dict*) – A GeoJSON-like mapping of a FeatureCollection.

Returns
Filtered FeatureCollection.

Return type
dict

simplify_polygons

simplify_polygons(*geojson: dict*) → dict

Simplify polygon geometries by applying `.buffer(0)`.

For Polygon geomtries, `.buffer(0)` can do the following:

1. *Sometimes* break up a polygon with "bowties" into multiple polygons. (See https://github.com/shapely/shapely/issues/599.)
2. *Sometimes* "simplify" polygons. See shapely documentation for `BaseGeometry.buffer()`.

Parameters
geojson (*dict*) – A GeoJSON-like mapping of a FeatureCollection.

Returns
FeatureCollection with simplified geometries.

Return type
dict

split_multi_geometries

split_multi_geometries(*geojson: dict*) → dict

Split any Features with multi-part geometries into multiple Features.

Parameters
geojson (*dict*) – A GeoJSON-like mapping of a FeatureCollection.

Returns
FeatureCollection without multi-part geometries.

Return type
dict

misc

Functions

all_equal(it)	Returns true if all elements are equal to each other
color_to_integer(color)	Given a PIL ImageColor string, return a packed integer.
color_to_triple([color])	Given a PIL ImageColor string, return a triple of integers representing the red, green, and blue values.
listify_uris(uris)	Convert to URI to list if needed.
normalize_color(color)	Convert color representation to a float 3-tuple with values in [0-1].
rgb_to_int_array(rgb_array)	

all_equal

all_equal(*it: list*)

Returns true if all elements are equal to each other

Parameters
it (*list*) –

color_to_integer

color_to_integer(*color: str*) → int

Given a PIL ImageColor string, return a packed integer.

Parameters
color (*str*) – A PIL ImageColor string

Returns
An integer containing the packed RGB values.

Return type
int

color_to_triple

color_to_triple(*color: Optional[Union[str, Sequence]] = None*) → Tuple[int, int, int]

Given a PIL ImageColor string, return a triple of integers representing the red, green, and blue values.

If color is None, return a random color.

Parameters
color (*Optional[Union[str, Sequence]]*) – A PIL ImageColor string

Returns
An triple of integers

Return type
Tuple[int, int, int]

listify_uris

listify_uris(*uris: Union[str, List[str]]*) → List[str]

Convert to URI to list if needed.

Parameters
uris (*Union[str, List[str]]*) –

Return type
List[str]

normalize_color

normalize_color(*color: Union[str, tuple, list]*) → Tuple[float, float, float]

Convert color representation to a float 3-tuple with values in [0-1].

Parameters
color (*Union[str, tuple, list]*) –

Return type
Tuple[float, float, float]

rgb_to_int_array

rgb_to_int_array(*rgb_array: ndarray*) → ndarray

Parameters
rgb_array (*ndarray*) –

Return type
ndarray

vectorization

Functions

denoise(mask, radius)	Apply morphological opening /w circular kernel to remove hi-freq noise.
get_kernel(rectangle[, width_factor, thickness])	
get_rectangle(buildings)	
mask_to_building_polygons(mask[, transform, ...])	Try to break up building clusters and then convert to polygons.
mask_to_polygons(mask[, transform])	Polygonize a raster mask.

denoise

denoise(*mask: ndarray*, *radius: int*) → ndarray

Apply morphological opening /w circular kernel to remove hi-freq noise.

Parameters

- **mask** (*np.ndarray*) – the binary mask to remove noise from.
- **radius** (*int*) – size in pixels of kernel for morphological op.

Returns
The mask after applying denoising.

Return type
np.ndarray

get_kernel

get_kernel(*rectangle: Tuple[Tuple[float, float], Tuple[float, float], float]*, *width_factor: float = 0.5*, *thickness: float = 0.001*) → Optional[ndarray]

Parameters

- **rectangle** (*Tuple[Tuple[float, float], Tuple[float, float], float]*) –
- **width_factor** (*float*) –
- **thickness** (*float*) –

Return type
Optional[*ndarray*]

get_rectangle

get_rectangle(*buildings: ndarray*) → Optional[Tuple[Tuple[float, float], Tuple[float, float], float]]

Parameters
buildings (*ndarray*) –

Return type
Optional[*Tuple*[*Tuple*[float, float], *Tuple*[float, float], float]]

mask_to_building_polygons

mask_to_building_polygons(*mask: ndarray*, *transform: Optional[Affine] = None*, *min_area: float = 100*, *width_factor: float = 0.5*, *thickness: float = 0.001*) → Iterator[BaseGeometry]

Try to break up building clusters and then convert to polygons.

Perofrms the following steps:

1. Identify connected components in `mask`.
2. For each connected component, if >= `min_area`:
   a. Generate a kernel based on its dimensions and orientation and the `width_factor` and `thickness` params.

b. Use the kernel to apply morphological erosion to component- mask.

c. Identify connected sub-components in component-mask.

d. For each connected sub-component, if >= `min_area`:

1. Apply morphological dilation using the kernel from above.
2. Polygonize using `mask_to_polygons()`.

Parameters

- **mask** (*np.ndarray*) – The mask containing buildings to polygonize.
- **transform** (*Optional[rio.Affine]*) – Affine transform to use during polygonization. Deafults to None (i.e. identity transform).
- **min_area** (*float*) – Minimum area (in pixels^2) of anything that can be considered to be a building or cluster of buildings. The goal is to distinguish between buildings and artifacts. Components with area less than this value will be discarded. Defaults to 100.
- **width_factor** (*float*) – Width of the structural element used to break building clusters as a fraction of the width of the cluster.
- **thickness** (*float*) – Thickness of the structural element that is used to break building clusters. Defaults to 0.001.

Returns
Generator of shapely polygons.

Return type
Iterator[BaseGeometry]

mask_to_polygons

mask_to_polygons(*mask: ndarray*, *transform: Optional[Affine] = None*) → Iterator[BaseGeometry]

Polygonize a raster mask. Wrapper around rasterio.features.shapes.

Parameters

- **mask** (*np.ndarray*) – The mask containing buildings to polygonize.
- **transform** (*Optional[rio.Affine]*) – Affine transform to use during polygonization. Deafults to None (i.e. identity transform).

Returns
Generator of shapely polygons.

Return type
Iterator[BaseGeometry]

vector_source

Modules

geojson_vector_source
geojson_vector_source_config
vector_source
vector_source_config

geojson_vector_source

Classes

GeoJSONVectorSource	A *VectorSource* for reading GeoJSON files.

GeoJSONVectorSource

class GeoJSONVectorSource

Bases: *VectorSource*

A *VectorSource* for reading GeoJSON files.

Attributes

extent	Envelope of union of all geoms.

__init__(*uri: str*, *crs_transformer:* CRSTransformer, *vector_transformers: List[*VectorTransformer*] = []*, *ignore_crs_field: bool = False*)

Constructor.

Parameters

- **uri** (*str*) – URI of the GeoJSON file.
- **crs_transformer** (CRSTransformer) – A CRSTransformer to convert between map and pixel coords. Normally this is obtained from a *RasterSource*.
- **vector_transformers** (*List[*VectorTransformer*]*) – VectorTransformers for transforming geometries. Defaults to [].
- **ignore_crs_field** (*bool*) – Ignore the CRS specified in the file and assume WGS84 (EPSG:4326) coords. Only WGS84 is supported currently. If False, and the file contains a CRS, will throw an exception on read. Defaults to False.

Methods

__init__(uri, crs_transformer[, ...])	Constructor.
get_dataframe([to_map_coords])	Return geometries as a GeoDataFrame.
get_geojson([to_map_coords])	Return transformed GeoJSON.
get_geoms([to_map_coords])	Returns all geometries in the transformed GeoJSON as Shapely geoms.

__init__(*uri: str*, *crs_transformer:* CRSTransformer, *vector_transformers: List[*VectorTransformer*] = []*, *ignore_crs_field: bool = False*)

Constructor.

Parameters

- **uri** (*str*) – URI of the GeoJSON file.
- **crs_transformer** (CRSTransformer) – A CRSTransformer to convert between map and pixel coords. Normally this is obtained from a *RasterSource*.
- **vector_transformers** (*List[*VectorTransformer*]*) – VectorTransformers for transforming geometries. Defaults to [].
- **ignore_crs_field** (*bool*) – Ignore the CRS specified in the file and assume WGS84 (EPSG:4326) coords. Only WGS84 is supported currently. If False, and the file contains a CRS, will throw an exception on read. Defaults to False.

get_dataframe(*to_map_coords: bool = False*) → geopandas.GeoDataFrame

Return geometries as a GeoDataFrame.

Parameters
to_map_coords (*bool*) –

Return type
geopandas.GeoDataFrame

get_geojson(*to_map_coords: bool = False*) → dict

Return transformed GeoJSON.

This makes the following transformations to the raw geojson:

- converts to pixels coords (by default)
- removes empty features
- splits apart multi-geoms and geom collections into single geometries
- buffers lines and points into Polygons

Additionally, the transformations specified by all the VectorTransformers in vector_transformers are also applied.

Parameters
to_map_coords (*bool*) – If true, will return GeoJSON in map coordinates.

Returns
dict in GeoJSON format

Return type
dict

get_geoms(*to_map_coords: bool = False*) → List[BaseGeometry]

Returns all geometries in the transformed GeoJSON as Shapely geoms.

Parameters
to_map_coords (*bool*) – If true, will return geoms in map coordinates.

Returns
List of Shapely geoms.

Return type
List['BaseGeometry']

property extent: *Box*

Envelope of union of all geoms.

geojson_vector_source_config

Configs

GeoJSONVectorSourceConfig	Configure a *GeoJSONVectorSource*.

GeoJSONVectorSourceConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model GeoJSONVectorSourceConfig

Configure a *GeoJSONVectorSource*.


```
{
   "title": "GeoJSONVectorSourceConfig",
   "description": "Configure a :class:`.GeoJSONVectorSource`.",
   "type": "object",
   "properties": {
      "transformers": {
         "title": "Transformers",
         "description": "List of VectorTransformers.",
         "default": [],
         "type": "array",
         "items": {
            "$ref": "#/definitions/VectorTransformerConfig"
         }
      },
      "type_hint": {
         "title": "Type Hint",
         "default": "geojson_vector_source",
         "enum": [
            "geojson_vector_source"
         ],
         "type": "string"
      },
```


(continues on next page)

(continued from previous page)


```
      "uri": {
         "title": "Uri",
         "description": "The URI of a GeoJSON file.",
         "type": "string"
      },
      "ignore_crs_field": {
         "title": "Ignore Crs Field",
         "default": false,
         "type": "boolean"
      }
   },
   "required": [
      "uri"
   ],
   "additionalProperties": false,
   "definitions": {
      "VectorTransformerConfig": {
         "title": "VectorTransformerConfig",
         "description": "Configure a :class:`.VectorTransformer`.",
         "type": "object",
         "properties": {
            "type_hint": {
               "title": "Type Hint",
               "default": "vector_transformer",
               "enum": [
                  "vector_transformer"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      }
   }
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *ignore_crs_field (bool)*
- *transformers (List[rastervision.core.data.vector_transformer.vector_transformer_config.VectorTransformerConfig])*
- *type_hint (Literal['geojson_vector_source'])*
- *uri (str)*

field ignore_crs_field: bool = False

field transformers: List[*VectorTransformerConfig*] = []

List of VectorTransformers.

field type_hint: Literal['geojson_vector_source'] = 'geojson_vector_source'

field uri: str [Required]

The URI of a GeoJSON file.

build(*class_config:* ClassConfig, *crs_transformer:* CRSTransformer, *use_transformers:* *bool* = *True*) → *GeoJSONVectorSource*

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

Parameters

- **class_config** (ClassConfig) –
- **crs_transformer** (CRSTransformer) –
- **use_transformers** (*bool*) –

Return type
GeoJSONVectorSource

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*pipeline:* *Optional*[RVPipelineConfig] = *None*, *scene:* *Optional*[SceneConfig] = *None*) → None

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

Parameters

- **pipeline** (*Optional*[RVPipelineConfig]) –
- **scene** (*Optional*[SceneConfig]) –

Return type
None

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field:* *str*, *valid_options:* *List*[*str*])

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List*[*str*]) – values that field is allowed to take

Raises
ConfigError – if field is invalid

vector_source

Classes

VectorSource	A source of vector data.

VectorSource

class VectorSource

Bases: ABC

A source of vector data.

Attributes

extent	Envelope of union of all geoms.

__init__(*crs_transformer:* CRSTransformer, *vector_transformers: List[*VectorTransformer*] = []*)

Constructor.

Parameters

- **crs_transformer** (CRSTransformer) – A CRSTransformer to convert between map and pixel coords. Normally this is obtained from a *RasterSource*.
- **vector_transformers** (*List[VectorTransformer]*) – VectorTransformers for transforming geometries. Defaults to [].

Methods

__init__(crs_transformer[, vector_transformers])	Constructor.
get_dataframe([to_map_coords])	Return geometries as a GeoDataFrame.
get_geojson([to_map_coords])	Return transformed GeoJSON.
get_geoms([to_map_coords])	Returns all geometries in the transformed GeoJSON as Shapely geoms.

__init__(*crs_transformer:* CRSTransformer, *vector_transformers: List[*VectorTransformer*] = []*)

Constructor.

Parameters

- **crs_transformer** (CRSTransformer) – A CRSTransformer to convert between map and pixel coords. Normally this is obtained from a *RasterSource*.
- **vector_transformers** (*List[VectorTransformer]*) – VectorTransformers for transforming geometries. Defaults to [].

get_dataframe(*to_map_coords: bool = False*) → geopandas.GeoDataFrame

Return geometries as a `GeoDataFrame`.

Parameters
to_map_coords (*bool*) –

Return type
geopandas.GeoDataFrame

get_geojson(*to_map_coords: bool = False*) → dict

Return transformed GeoJSON.

This makes the following transformations to the raw geojson:

- converts to pixels coords (by default)
- removes empty features
- splits apart multi-geoms and geom collections into single geometries
- buffers lines and points into Polygons

Additionally, the transformations specified by all the VectorTransformers in vector_transformers are also applied.

Parameters
to_map_coords (*bool*) – If true, will return GeoJSON in map coordinates.

Returns
dict in GeoJSON format

Return type
dict

get_geoms(*to_map_coords: bool = False*) → List[BaseGeometry]

Returns all geometries in the transformed GeoJSON as Shapely geoms.

Parameters
to_map_coords (*bool*) – If true, will return geoms in map coordinates.

Returns
List of Shapely geoms.

Return type
List['BaseGeometry']

property extent: *Box*

Envelope of union of all geoms.

Functions

sanitize_geojson(geojson, crs_transformer[, ...])	Apply some basic transformations (listed below) to a GeoJSON.

sanitize_geojson

sanitize_geojson(*geojson: dict*, *crs_transformer:* CRSTransformer, *to_map_coords: bool = False*) → dict

Apply some basic transformations (listed below) to a GeoJSON.

The following transformations are applied:

1. Removal of features without geometries.
2. Coordinate transformation to pixel coordinates.
3. Splitting of multi-part geometries e.g. MultiPolygon –> Polygons.
4. (Optional) If to_map_coords=true, transformation back to map coordinates.

Parameters

- **geojson** (*dict*) – A GeoJSON-like mapping of a FeatureCollection.
- **crs_transformer** (CRSTransformer) – A CRS transformer for coordinate transformation.
- **to_map_coords** (*bool, optional*) – If True, transform geometries back to map coordinates before returning. Defaults to False.

Returns
Transformed FeatureCollection.

Return type
dict

vector_source_config

Configs

VectorSourceConfig	Configure a *VectorSource*.

VectorSourceConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model VectorSourceConfig

Configure a *VectorSource*.


```
{
   "title": "VectorSourceConfig",
   "description": "Configure a :class:`.VectorSource`.",
   "type": "object",
   "properties": {
      "transformers": {
         "title": "Transformers",
         "description": "List of VectorTransformers.",
         "default": [],
         "type": "array",
```


(continues on next page)

(continued from previous page)


```
            "items": {
               "$ref": "#/definitions/VectorTransformerConfig"
            }
         },
         "type_hint": {
            "title": "Type Hint",
            "default": "vector_source",
            "enum": [
               "vector_source"
            ],
            "type": "string"
         }
      },
      "additionalProperties": false,
      "definitions": {
         "VectorTransformerConfig": {
            "title": "VectorTransformerConfig",
            "description": "Configure a :class:`.VectorTransformer`.",
            "type": "object",
            "properties": {
               "type_hint": {
                  "title": "Type Hint",
                  "default": "vector_transformer",
                  "enum": [
                     "vector_transformer"
                  ],
                  "type": "string"
               }
            },
            "additionalProperties": false
         }
      }
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *transformers (List[rastervision.core.data.vector_transformer.vector_transformer_config.VectorTransformerConfig])*
- *type_hint (Literal['vector_source'])*

field transformers: List[*VectorTransformerConfig*] = []

List of VectorTransformers.

field type_hint: Literal['vector_source'] = 'vector_source'

abstract build(*class_config:* ClassConfig, *crs_transformer:* CRSTransformer) → *VectorSource*

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

Parameters

- **class_config** (ClassConfig) –
- **crs_transformer** (CRSTransformer) –

Return type
VectorSource

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*pipeline: Optional[RVPipelineConfig] = None*, *scene: Optional[SceneConfig] = None*) → None

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

Parameters

- **pipeline** (*Optional[RVPipelineConfig]*) –
- **scene** (*Optional[SceneConfig]*) –

Return type
None

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises
ConfigError – if field is invalid

vector_transformer

Modules

buffer_transformer
buffer_transformer_config
class_inference_transformer
class_inference_transformer_config
label_maker
shift_transformer
shift_transformer_config
vector_transformer
vector_transformer_config

buffer_transformer

Classes

BufferTransformer	Buffers geometries.

BufferTransformer

class BufferTransformer

Bases: *VectorTransformer*

Buffers geometries.

__init__(*geom_type: str*, *class_bufs: Optional[Dict[int, Optional[float]]] = None*, *default_buf: Optional[float] = None*)

Constructor.

Parameters

- **geom_type** (*str*) – The geometry type to apply this transform to. E.g. "LineString", "Point", "Polygon".
- **class_bufs** (*Dict[int, Optional[float]], optional*) – Mapping from class IDs to buffer amounts (in pixels). If a class ID is not found in the mapping, the value specified by the default_buf field will be used. If the buffer value for a class is None, then no buffering will be applied to the geoms of that class. Defaults to {}.

- **default_buf** (*Optional[float], optional*) – Default buffer to apply to classes not in class_bufs. If None, no buffering will be applied to the geoms of those missing classes. Defaults to None.

Methods

__init__(geom_type[, class_bufs, default_buf])	Constructor.
transform(geojson[, crs_transformer])	Transform a GeoJSON mapping of vector data.

__init__(*geom_type: str*, *class_bufs: Optional[Dict[int, Optional[float]]] = None*, *default_buf: Optional[float] = None*)

Constructor.

Parameters

- **geom_type** (*str*) – The geometry type to apply this transform to. E.g. "LineString", "Point", "Polygon".
- **class_bufs** (*Dict[int, Optional[float]], optional*) – Mapping from class IDs to buffer amounts (in pixels). If a class ID is not found in the mapping, the value specified by the default_buf field will be used. If the buffer value for a class is None, then no buffering will be applied to the geoms of that class. Defaults to {}.
- **default_buf** (*Optional[float], optional*) – Default buffer to apply to classes not in class_bufs. If None, no buffering will be applied to the geoms of those missing classes. Defaults to None.

transform(*geojson: dict*, *crs_transformer: Optional[CRSTransformer] = None*) → dict

Transform a GeoJSON mapping of vector data.

Parameters

- **geojson** (*dict*) – A GeoJSON-like mapping of a FeatureCollection.
- **crs_transformer** (*Optional[CRSTransformer]*) –

Returns
Transformed GeoJSON.

Return type
dict

buffer_transformer_config

Configs

BufferTransformerConfig	Configure a *BufferTransformer*.

BufferTransformerConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model BufferTransformerConfig

Configure a *BufferTransformer*.

This is useful, for example, for buffering lines representing roads so that their width roughly matches the width of roads in the imagery.


```
{
   "title": "BufferTransformerConfig",
   "description": "Configure a :class:`.BufferTransformer`.\n\nThis is useful, for␣
↪example, for buffering lines representing roads so that\ntheir width roughly␣
↪matches the width of roads in the imagery.",
   "type": "object",
   "properties": {
      "type_hint": {
         "title": "Type Hint",
         "default": "buffer_transformer",
         "enum": [
            "buffer_transformer"
         ],
         "type": "string"
      },
      "geom_type": {
         "title": "Geom Type",
         "description": "The geometry type to apply this transform to. E.g. \
↪"LineString\", \"Point\", \"Polygon\".",
         "type": "string"
      },
      "class_bufs": {
         "title": "Class Bufs",
         "description": "Mapping from class IDs to buffer amounts (in pixels). If a␣
↪class ID is not found in the mapping, the value specified by the default_buf␣
↪field will be used. If the buffer value for a class is None, then no buffering␣
↪will be applied to the geoms of that class and the geom won't get converted to a␣
↪Polygon. Not converting to Polygon is incompatible with the currently available␣
↪LabelSources, but may be useful in the future.",
         "default": {},
         "type": "object",
         "additionalProperties": {
            "type": "number"
         }
      },
      "default_buf": {
         "title": "Default Buf",
         "description": "Default buffer to apply to classes not in class_bufs. If␣
↪None, no buffering will be applied to the geoms of those classes.",
         "default": 1,
         "type": "number"
      }
```


(continues on next page)

(continued from previous page)

```
    },
    "required": [
        "geom_type"
    ],
    "additionalProperties": false
}
```

Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *class_bufs (Dict[int, Optional[float]])*
- *default_buf (Optional[float])*
- *geom_type (str)*
- *type_hint (Literal['buffer_transformer'])*

field class_bufs: Dict[int, Optional[float]] = {}

Mapping from class IDs to buffer amounts (in pixels). If a class ID is not found in the mapping, the value specified by the default_buf field will be used. If the buffer value for a class is None, then no buffering will be applied to the geoms of that class and the geom won't get converted to a Polygon. Not converting to Polygon is incompatible with the currently available LabelSources, but may be useful in the future.

field default_buf: Optional[float] = 1

Default buffer to apply to classes not in class_bufs. If None, no buffering will be applied to the geoms of those classes.

field geom_type: str [Required]

The geometry type to apply this transform to. E.g. "LineString", "Point", "Polygon".

field type_hint: Literal['buffer_transformer'] = 'buffer_transformer'

build(*class_config: Optional[*ClassConfig*] = None*) → *BufferTransformer*

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

Parameters
class_config (*Optional[*ClassConfig*]*) –

Return type
BufferTransformer

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*pipeline: Optional[RVPipelineConfig] = None*, *scene: Optional[SceneConfig] = None*) → None

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

Parameters

- **pipeline** (*Optional[RVPipelineConfig]*) –
- **scene** (*Optional[SceneConfig]*) –

Return type
None

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises
ConfigError – if field is invalid

class_inference_transformer

Classes

ClassInferenceTransformer	Infers missing class_ids from GeoJSON features.

ClassInferenceTransformer

class ClassInferenceTransformer

Bases: *VectorTransformer*

Infers missing class_ids from GeoJSON features.

Rules:

1) If class_id is in feature['properties'], use it.
2) **If class_config is set and class_name or label are in**
   feature['properties'] and in class_config, use corresponding class_id.
3) **If class_id_to_filter is set and filter is true when applied to**
   feature, use corresponding class_id.
4) Otherwise, return the default_class_id

__init__(*default_class_id: Optional[int]*, *class_config: Optional[ClassConfig] = None*, *class_id_to_filter: Optional[Dict[int, list]] = None*)

Parameters

- **default_class_id** (*Optional[int]*) –
- **class_config** (*Optional[ClassConfig]*) –
- **class_id_to_filter** (*Optional[Dict[int, list]]*) –

Methods

__init__(default_class_id[, class_config, ...])	

infer_feature_class_id(feature, default_class_id)	Infer the class_id for a GeoJSON feature.
transform(geojson[, crs_transformer])	Add class_id to feature properties and drop features with no class.

__init__(*default_class_id: Optional[int]*, *class_config: Optional[ClassConfig] = None*, *class_id_to_filter: Optional[Dict[int, list]] = None*)

Parameters

- **default_class_id** (*Optional[int]*) –
- **class_config** (*Optional[ClassConfig]*) –
- **class_id_to_filter** (*Optional[Dict[int, list]]*) –

static infer_feature_class_id(*feature: dict*, *default_class_id: Optional[int]*, *class_config: Optional[ClassConfig] = None*, *class_id_to_filter: Optional[Dict[int, list]] = None*) → Optional[int]

Infer the class_id for a GeoJSON feature.

Rules:

1) If class_id is in feature['properties'], use it.
2) **If class_config is set and class_name or label are in** feature['properties'] and in class_config, use corresponding class_id.
3) **If class_id_to_filter is set and filter is true when applied to** feature, use corresponding class_id.
4) Otherwise, return the default_class_id.

Parameters

- **feature** (*dict*) – GeoJSON feature.
- **default_class_id** (*Optional[int]*) –
- **class_config** (*Optional[ClassConfig]*) –
- **class_id_to_filter** (*Optional[Dict[int, list]]*) –

Returns

Inferred class ID.

Return type
Optional[int]

transform(*geojson: dict*, *crs_transformer: Optional[CRSTransformer] = None*) → dict

Add class_id to feature properties and drop features with no class.

For each feature in geojson, the class_id is inferred and is set into feature['properties']. If the class_id is None (because none of the rules apply and the default_class_id is None), the feature is dropped.

Parameters

- **geojson** (*dict*) –
- **crs_transformer** (*Optional[CRSTransformer]*) –

Return type
dict

class_inference_transformer_config

Configs

ClassInferenceTransformerConfig	Configure a *ClassInferenceTransformer*.

ClassInferenceTransformerConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model ClassInferenceTransformerConfig

Configure a *ClassInferenceTransformer*.


```
{
   "title": "ClassInferenceTransformerConfig",
   "description": "Configure a :class:`.ClassInferenceTransformer`.",
   "type": "object",
   "properties": {
      "type_hint": {
         "title": "Type Hint",
         "default": "class_inference_transformer",
         "enum": [
            "class_inference_transformer"
         ],
         "type": "string"
      },
      "default_class_id": {
         "title": "Default Class Id",
         "description": "The default class_id to use if class cannot be inferred␣
↪using other mechanisms. If a feature has an inferred class_id of None, then it␣
↪will be deleted.",
         "type": "integer"
      },
```


(continues on next page)

(continued from previous page)

```
      "class_id_to_filter": {
         "title": "Class Id To Filter",
         "description": "Map from class_id to JSON filter used to infer missing␣
↪class_ids. Each key should be a class id, and its value should be a boolean␣
↪expression which is run against the property field for each feature. This allows␣
↪matching different features to different class IDs based on its properties. The␣
↪expression schema is that described by https://docs.mapbox.com/mapbox-gl-js/style-
↪spec/other/#other-filter",
         "type": "object",
         "additionalProperties": {
            "type": "array",
            "items": {}
         }
      }
   },
   "additionalProperties": false
}
```

Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *class_id_to_filter (Optional[Dict[int, list]])*
- *default_class_id (Optional[int])*
- *type_hint (Literal['class_inference_transformer'])*

field class_id_to_filter: Optional[Dict[int, list]] = None

Map from class_id to JSON filter used to infer missing class_ids. Each key should be a class id, and its value should be a boolean expression which is run against the property field for each feature. This allows matching different features to different class IDs based on its properties. The expression schema is that described by https://docs.mapbox.com/mapbox-gl-js/style-spec/other/#other-filter

field default_class_id: Optional[int] = None

The default class_id to use if class cannot be inferred using other mechanisms. If a feature has an inferred class_id of None, then it will be deleted.

field type_hint: Literal['class_inference_transformer'] = 'class_inference_transformer'

build(*class_config: Optional[*ClassConfig*] = None*) → *ClassInferenceTransformer*

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

Parameters
class_config (*Optional[*ClassConfig*]*) –

Return type
ClassInferenceTransformer

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*pipeline: Optional[RVPipelineConfig] = None*, *scene: Optional[SceneConfig] = None*) → None

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

Parameters

- **pipeline** (*Optional[RVPipelineConfig]*) –
- **scene** (*Optional[SceneConfig]*) –

Return type
None

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises
ConfigError – if field is invalid

label_maker

Modules

filter	Create a feature filtering function from a Mapbox GL Filter.

filter

Create a feature filtering function from a Mapbox GL Filter.

Functions

create_filter(filt)	Create a feature filtering function from a Mapbox GL Filter.

create_filter

create_filter(*filt*)

Create a feature filtering function from a Mapbox GL Filter.

Given a filter expressed as nested lists, return a new function that evaluates whether a given feature (with a .properties or .tags property) passes its test. More information: - https://www.mapbox.com/mapbox-gl-js/style-spec/#other-filter - https://github.com/mapbox/mapbox-gl-js/tree/master/src/style-spec/feature_filter

Parameters
filt (*list*) – Mapbox GL filter

Returns
func – A function which evaluates whether a GeoJSON feature meets the input filter criteria

Return type
function

shift_transformer

Classes

ShiftTransformer	Shift geometries by some distance specified in meters.

ShiftTransformer

class ShiftTransformer

Bases: *VectorTransformer*

Shift geometries by some distance specified in meters.

__init__(*x_shift: float = 0.0*, *y_shift: float = 0.0*, *round_pixels: bool = True*)

Constructor.

Args:

Parameters

- **x_shift** (*float*) –
- **y_shift** (*float*) –
- **round_pixels** (*bool*) –

Methods

__init__([x_shift, y_shift, round_pixels])	Constructor.
make_wgs84_transformer(crs_transformer)	
transform(geojson[, crs_transformer])	Transform a GeoJSON mapping of vector data.

__init__(*x_shift: float = 0.0*, *y_shift: float = 0.0*, *round_pixels: bool = True*)

Constructor.

Args:

Parameters

- **x_shift** (*float*) –
- **y_shift** (*float*) –
- **round_pixels** (*bool*) –

make_wgs84_transformer(*crs_transformer: CRSTransformer*)

Parameters
crs_transformer (*CRSTransformer*) –

transform(*geojson: dict*, *crs_transformer: Optional[CRSTransformer] = None*) → dict

Transform a GeoJSON mapping of vector data.

Parameters

- **geojson** (*dict*) – A GeoJSON-like mapping of a FeatureCollection.
- **crs_transformer** (*Optional[CRSTransformer]*) –

Returns
Transformed GeoJSON.

Return type
dict

shift_transformer_config

Configs

ShiftTransformerConfig	Configure a *ShiftTransformer*.

ShiftTransformerConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model ShiftTransformerConfig

Configure a *ShiftTransformer*.

```
{
   "title": "ShiftTransformerConfig",
   "description": "Configure a :class:`.ShiftTransformer`.",
   "type": "object",
   "properties": {
      "type_hint": {
         "title": "Type Hint",
         "default": "shift_transformer",
         "enum": [
            "shift_transformer"
         ],
         "type": "string"
      },
      "x_shift": {
         "title": "X Shift",
         "default": 0.0,
         "descriptions": "Distance in meters to shift along the x-axis. Postive␣
↪values shift eastward.",
         "type": "number"
      },
      "y_shift": {
         "title": "Y Shift",
         "default": 0.0,
         "descriptions": "Distance in meters to shift along the y-axis. Postive␣
↪values shift northward.",
         "type": "number"
      },
      "round_pixels": {
         "title": "Round Pixels",
         "default": true,
         "descriptions": "Whether to round shifted pixel values to integers.",
         "type": "boolean"
      }
   },
   "additionalProperties": false
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *round_pixels (bool)*
- *type_hint (Literal['shift_transformer'])*
- *x_shift (float)*
- *y_shift (float)*

field round_pixels: bool = True

field type_hint: Literal['shift_transformer'] = 'shift_transformer'

field x_shift: float = 0.0

field y_shift: float = 0.0

build(*class_config: Optional[*ClassConfig*] = None*) → *ShiftTransformer*

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

Parameters
class_config (*Optional[*ClassConfig*]*) –

Return type
ShiftTransformer

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*pipeline: Optional[*RVPipelineConfig*] = None*, *scene: Optional[*SceneConfig*] = None*) → None

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

Parameters

- **pipeline** (*Optional[*RVPipelineConfig*]*) –
- **scene** (*Optional[*SceneConfig*]*) –

Return type
None

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises
ConfigError – if field is invalid

vector_transformer

Classes

VectorTransformer	Transforms vector data.

VectorTransformer

class VectorTransformer

Bases: ABC

Transforms vector data.

__init__()

Methods

__init__()

transform(geojson[, crs_transformer])	Transform a GeoJSON mapping of vector data.

abstract transform(*geojson: dict*, *crs_transformer: Optional[CRSTransformer] = None*) → dict

Transform a GeoJSON mapping of vector data.

Parameters

- **geojson** (*dict*) – A GeoJSON-like mapping of a FeatureCollection.
- **crs_transformer** (*Optional[CRSTransformer]*) –

Returns
Transformed GeoJSON.

Return type
dict

vector_transformer_config

Configs

VectorTransformerConfig	Configure a *VectorTransformer*.

VectorTransformerConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model VectorTransformerConfig

Configure a *VectorTransformer*.


```
{
   "title": "VectorTransformerConfig",
   "description": "Configure a :class:`.VectorTransformer`.",
   "type": "object",
   "properties": {
      "type_hint": {
         "title": "Type Hint",
         "default": "vector_transformer",
         "enum": [
            "vector_transformer"
         ],
         "type": "string"
      }
   },
   "additionalProperties": false
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *type_hint (Literal['vector_transformer'])*

field type_hint: Literal['vector_transformer'] = 'vector_transformer'

abstract build(*class_config:* ClassConfig) → *VectorTransformer*

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

Parameters
class_config (ClassConfig) –

Return type
VectorTransformer

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*pipeline: Optional[RVPipelineConfig] = None*, *scene: Optional[SceneConfig] = None*) → None

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

Parameters

- **pipeline** (*Optional[RVPipelineConfig]*) –
- **scene** (*Optional[SceneConfig]*) –

Return type
None

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises
ConfigError – if field is invalid

9.2.6 data_sample

9.2.7 evaluation

Modules

chip_classification_evaluation	
chip_classification_evaluator	
chip_classification_evaluator_config	
class_evaluation_item	Defines `ClassEvaluationItem`.
classification_evaluation	Defines abstract base evaluation class for all tasks.
classification_evaluator	
classification_evaluator_config	
evaluation_item	
evaluator	
evaluator_config	
object_detection_evaluation	
object_detection_evaluator	
object_detection_evaluator_config	
semantic_segmentation_evaluation	
semantic_segmentation_evaluator	
semantic_segmentation_evaluator_config	

chip_classification_evaluation

Classes

ChipClassificationEvaluation	

ChipClassificationEvaluation

class ChipClassificationEvaluation

Bases: *ClassificationEvaluation*

__init__(*class_config:* ClassConfig)

Parameters
class_config (ClassConfig) –

Methods

__init__(class_config)	

compute(gt_labels, pred_labels)	Compute metrics for a single scene.
compute_avg()	Compute average metrics over all classes.
merge(other[, scene_id])	Merge Evaluation for another Scene into this one.
reset()	Reset the Evaluation.
save(output_uri)	Save this Evaluation to a file.
to_json()	Serialize to a dict or list.

__init__(*class_config: ClassConfig*)

Parameters
class_config (ClassConfig) –

compute(*gt_labels: ChipClassificationLabels*, *pred_labels: ChipClassificationLabels*) → None

Compute metrics for a single scene.

Parameters

- **ground_truth_labels** – Ground Truth labels to evaluate against.
- **prediction_labels** – The predicted labels to evaluate.
- **gt_labels** (ChipClassificationLabels) –
- **pred_labels** (ChipClassificationLabels) –

Return type
None

compute_avg() → None

Compute average metrics over all classes.

Return type
None

merge(*other: ClassificationEvaluation*, *scene_id: Optional[str] = None*) → None

Merge Evaluation for another Scene into this one.

This is useful for computing the average metrics of a set of scenes. The results of the averaging are stored in this Evaluation.

Parameters

- **other** (ClassificationEvaluation) – Evaluation to merge into this one
- **scene_id** (*Optional[str], optional*) – ID of scene. If specified, (a copy of) `other` will be saved and be availabel in `to_json()`'s output. Defaults to None.

Return type
None

reset()

Reset the Evaluation.

save(*output_uri: str*) → None

Save this Evaluation to a file.

Parameters
output_uri (*str*) – string URI for the file to write.

Return type
None

to_json() → Union[dict, list]

Serialize to a dict or list.

Returns

Class-wise and (if available) scene-wise
evaluations.

Return type
Union[dict, list]

chip_classification_evaluator

Classes

ChipClassificationEvaluator	Evaluates predictions for a set of scenes.

ChipClassificationEvaluator

class ChipClassificationEvaluator

Bases: *ClassificationEvaluator*

Evaluates predictions for a set of scenes.

__init__(*class_config*, *output_uri*)

Methods

__init__(class_config, output_uri)	
create_evaluation()	
evaluate_predictions(ground_truth, predictions)	Evaluate predictions against ground truth.
evaluate_scene(scene)	Evaluate predictions from a scene's labels store.
process(scenes[, tmp_dir])	Evaluate all given scenes and save the evaluations.

__init__(*class_config*, *output_uri*)

create_evaluation()

evaluate_predictions(*ground_truth:* Labels, *predictions:* Labels) → *ClassificationEvaluation*

Evaluate predictions against ground truth.

Parameters

- **ground_truth** (Labels) – Ground truth labels.

- **predictions** (Labels) – Predictions.

Returns
The evaluation.

Return type
Any

evaluate_scene(*scene:* Scene) → *ClassificationEvaluation*

Evaluate predictions from a scene's labels store.

The predictions are evalated against ground truth labels from the scene's label source.

Parameters
scene (Scene) – A scene with a label source and a label store.

Returns
The evaluation.

Return type
ClassificationEvaluation

process(*scenes: Iterable[*Scene*]*, *tmp_dir: Optional[str] = None*) → None

Evaluate all given scenes and save the evaluations.

Parameters

- **scenes** (*Iterable[*Scene*]*) – Scenes to evaluate.
- **tmp_dir** (*Optional[str]*) –

Return type
None

chip_classification_evaluator_config

Configs

ChipClassificationEvaluatorConfig	Configure a *ChipClassificationEvaluator*.

ChipClassificationEvaluatorConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model ChipClassificationEvaluatorConfig

Configure a *ChipClassificationEvaluator*.


```
{
   "title": "ChipClassificationEvaluatorConfig",
   "description": "Configure a :class:`.ChipClassificationEvaluator`.",
   "type": "object",
   "properties": {
      "output_uri": {
         "title": "Output Uri",
```


(continues on next page)

(continued from previous page)


```
        "description": "URI of directory where evaluator output will be saved.␣
↪Evaluations for each scene-group will be save in a JSON file at <output_uri>/
↪<scene-group-name>/eval.json. If None, and this Config is part of an RVPipeline,␣
↪this field will be auto-generated.",
        "type": "string"
      },
      "type_hint": {
        "title": "Type Hint",
        "default": "chip_classification_evaluator",
        "enum": [
          "chip_classification_evaluator"
        ],
        "type": "string"
      }
   },
   "additionalProperties": false
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *output_uri (Optional[str])*
- *type_hint (Literal['chip_classification_evaluator'])*

field output_uri: Optional[str] = None

URI of directory where evaluator output will be saved. Evaluations for each scene-group will be save in a JSON file at <output_uri>/<scene-group-name>/eval.json. If None, and this Config is part of an RVPipeline, this field will be auto-generated.

field type_hint: Literal['chip_classification_evaluator'] = 'chip_classification_evaluator'

build(*class_config:* ClassConfig, *scene_group: Optional[Tuple[str, Iterable[str]]] = None*) → *ChipClassificationEvaluator*

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

Parameters

- **class_config** (ClassConfig) –
- **scene_group** (*Optional[Tuple[str, Iterable[str]]]*) –

Return type
ChipClassificationEvaluator

get_output_uri(*scene_group_name: Optional[str] = None*) → str

Parameters
scene_group_name (*Optional[str]*) –

Return type
str

`recursive_validate_config()`

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

`revalidate()`

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

`update(pipeline: Optional[RVPipelineConfig] = None) → None`

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

Parameters
pipeline (*Optional[RVPipelineConfig]*) –

Return type
None

`validate_config()`

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

`validate_list(field: str, valid_options: List[str])`

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises
ConfigError – if field is invalid

class_evaluation_item

Defines `ClassEvaluationItem`.

Classes

ClassEvaluationItem	A wrapper around a binary (2x2) confusion matrix of the form

ClassEvaluationItem

class ClassEvaluationItem

Bases: *EvaluationItem*

A wrapper around a binary (2x2) confusion matrix of the form

[TN FP]
[FN TP]

where TN need not necessarily be available.

Exposes evaluation metrics computed from the confusion matrix as properties.

class_id

Class ID.

Type
int

class_name

Class name.

Type
str

conf_mat

Confusion matrix: `[[TN, FP], [FN, TP]]`.

Type
np.ndarray

extra_info

Arbitrary extra key-value pairs that will be included in the dict returned by `to_json()`.

Type
dict

Attributes

f1	F1 score = 2 * (precision * recall) / (precision + recall)
false_neg	False negative count.
false_pos	False positive count.
gt_count	Positive ground-truth count.
precision	TP / (TP + FP)
pred_count	Positive prediction count.
recall	TP / (TP + FN)
sensitivity	Equivalent to recall.
specificity	TN / (TN + FP)
true_neg	True negative count.
true_pos	True positive count.

__init__(*class_id: int*, *class_name: str*, *tp: int*, *fp: int*, *fn: int*, *tn: Optional[int] = None*, ***kwargs*)

Constructor.

Parameters

- **class_id** (*int*) – Class ID.
- **class_name** (*str*) – Class name.
- **tp** (*int*) – True positive count.
- **fp** (*int*) – False positive count.
- **fn** (*int*) – False negative count.
- **tn** (*Optional[int], optional*) – True negative count. Defaults to None.
- ****kwargs** – Additional data can be provided as keyword arguments. These will be included in the dict returned by `to_json()`.

Methods

__init__(class_id, class_name, tp, fp, fn[, tn])	Constructor.
from_multiclass_conf_mat(conf_mat, class_id, ...)	Construct from a multi-class confusion matrix and a target class ID.
merge(other)	Merge with another `ClassEvaluationItem`.
to_json()	Serialize to a dict.

__init__(*class_id: int*, *class_name: str*, *tp: int*, *fp: int*, *fn: int*, *tn: Optional[int] = None*, ***kwargs*)

Constructor.

Parameters

- **class_id** (*int*) – Class ID.
- **class_name** (*str*) – Class name.
- **tp** (*int*) – True positive count.
- **fp** (*int*) – False positive count.
- **fn** (*int*) – False negative count.
- **tn** (*Optional[int], optional*) – True negative count. Defaults to None.
- ****kwargs** – Additional data can be provided as keyword arguments. These will be included in the dict returned by `to_json()`.

classmethod from_multiclass_conf_mat(*conf_mat: ndarray*, *class_id: int*, *class_name: str*, ***kwargs*) → *ClassEvaluationItem*

Construct from a multi-class confusion matrix and a target class ID.

Parameters

- **conf_mat** (*np.ndarray*) – A multi-class confusion matrix.
- **class_id** (*int*) – The ID of the target class.
- **class_name** (*str*) – The name of the target class.

Returns

ClassEvaluationItem for target class.

Return type
ClassEvaluationItem

merge(*other:* ClassEvaluationItem) → None

Merge with another `ClassEvaluationItem`.

This is accomplished by summing the confusion matrices.

Parameters
other (ClassEvaluationItem) –

Return type
None

to_json() → dict

Serialize to a dict.

Return type
dict

property f1: float

F1 score = `2 * (precision * recall) / (precision + recall)`

property false_neg: int

False negative count.

property false_pos: int

False positive count.

property gt_count: int

Positive ground-truth count.

property precision: float

`TP / (TP + FP)`

property pred_count: int

Positive prediction count.

property recall: float

`TP / (TP + FN)`

property sensitivity: float

Equivalent to `recall`.

property specificity: Optional[float]

`TN / (TN + FP)`

property true_neg: Optional[int]

True negative count.

Returns
Count as int if available. Otherwise, None.

Return type
Optional[int]

property true_pos: int

True positive count.

classification_evaluation

Defines abstract base evaluation class for all tasks.

Classes

ClassificationEvaluation	Base class for evaluating predictions for pipelines that have classes.

ClassificationEvaluation

class ClassificationEvaluation

Bases: ABC

Base class for evaluating predictions for pipelines that have classes.

Evaluations can be keyed, for instance, if evaluations happen per class.

class_to_eval_item

Mapping from class IDs to ``ClassEvaluationItem``s.

Type
Dict[int, *ClassEvaluationItem*]

scene_to_eval

Mapping from scene IDs to ``ClassificationEvaluation``s.

Type
Dict[str, *ClassificationEvaluation*]

avg_item

Averaged evaluation over all classes.

Type
Optional[Dict[str, Any]]

conf_mat

Confusion matrix.

Type
Optional[np.ndarray]

__init__()

Methods

__init__()

compute(ground_truth_labels, prediction_labels)	Compute metrics for a single scene.
compute_avg()	Compute average metrics over all classes.
merge(other[, scene_id])	Merge Evaluation for another Scene into this one.
reset()	Reset the Evaluation.
save(output_uri)	Save this Evaluation to a file.
to_json()	Serialize to a dict or list.

__init__()

abstract compute(*ground_truth_labels*, *prediction_labels*)

Compute metrics for a single scene.

Parameters

- **ground_truth_labels** – Ground Truth labels to evaluate against.
- **prediction_labels** – The predicted labels to evaluate.

compute_avg() → None

Compute average metrics over all classes.

Return type
None

merge(*other:* ClassificationEvaluation, *scene_id: Optional[str] = None*) → None

Merge Evaluation for another Scene into this one.

This is useful for computing the average metrics of a set of scenes. The results of the averaging are stored in this Evaluation.

Parameters

- **other** (ClassificationEvaluation) – Evaluation to merge into this one
- **scene_id** (*Optional[str], optional*) – ID of scene. If specified, (a copy of) other will be saved and be availabel in to_json()'s output. Defaults to None.

Return type
None

reset()

Reset the Evaluation.

save(*output_uri: str*) → None

Save this Evaluation to a file.

Parameters
output_uri (*str*) – string URI for the file to write.

Return type
None

to_json() → Union[dict, list]

Serialize to a dict or list.

Returns

Class-wise and (if available) scene-wise
evaluations.

Return type
Union[dict, list]

Functions

ensure_json_serializable(obj)	Convert numpy types to JSON serializable equivalents.

ensure_json_serializable

ensure_json_serializable(*obj: Any*) → dict

Convert numpy types to JSON serializable equivalents.

Parameters
obj (*Any*) –

Return type
dict

classification_evaluator

Classes

ClassificationEvaluator	Evaluates predictions for a set of scenes.

ClassificationEvaluator

class ClassificationEvaluator

Bases: *Evaluator*

Evaluates predictions for a set of scenes.

__init__(*class_config: ClassConfig*, *output_uri: Optional[str] = None*)

Parameters

- **class_config** (*ClassConfig*) –
- **output_uri** (*Optional[str]*) –

Methods

__init__(class_config[, output_uri])	
create_evaluation()	
evaluate_predictions(ground_truth, predictions)	Evaluate predictions against ground truth.
evaluate_scene(scene)	Evaluate predictions from a scene's labels store.
process(scenes[, tmp_dir])	Evaluate all given scenes and save the evaluations.

__init__(*class_config:* ClassConfig, *output_uri:* *Optional[str]* = *None*)

Parameters

- **class_config** (ClassConfig) –
- **output_uri** (*Optional[str]*) –

abstract create_evaluation() → *ClassificationEvaluation*

Return type
ClassificationEvaluation

evaluate_predictions(*ground_truth:* Labels, *predictions:* Labels) → *ClassificationEvaluation*

Evaluate predictions against ground truth.

Parameters

- **ground_truth** (Labels) – Ground truth labels.
- **predictions** (Labels) – Predictions.

Returns
The evaluation.

Return type
Any

evaluate_scene(*scene:* Scene) → *ClassificationEvaluation*

Evaluate predictions from a scene's labels store.

The predictions are evalated against ground truth labels from the scene's label source.

Parameters
scene (Scene) – A scene with a label source and a label store.

Returns
The evaluation.

Return type
ClassificationEvaluation

process(*scenes:* *Iterable[*Scene*]*, *tmp_dir:* *Optional[str]* = *None*) → None

Evaluate all given scenes and save the evaluations.

Parameters

- **scenes** (*Iterable[*Scene*]*) – Scenes to evaluate.
- **tmp_dir** (*Optional[str]*) –

Return type
None

classification_evaluator_config

Configs

ClassificationEvaluatorConfig	Configure a *ClassificationEvaluator*.

ClassificationEvaluatorConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model ClassificationEvaluatorConfig

Configure a *ClassificationEvaluator*.


```
{
   "title": "ClassificationEvaluatorConfig",
   "description": "Configure a :class:`.ClassificationEvaluator`.",
   "type": "object",
   "properties": {
      "output_uri": {
         "title": "Output Uri",
         "description": "URI of directory where evaluator output will be saved.␣
↪Evaluations for each scene-group will be save in a JSON file at <output_uri>/
↪<scene-group-name>/eval.json. If None, and this Config is part of an RVPipeline,␣
↪this field will be auto-generated.",
         "type": "string"
      },
      "type_hint": {
         "title": "Type Hint",
         "default": "classification_evaluator",
         "enum": [
            "classification_evaluator"
         ],
         "type": "string"
      }
   },
   "additionalProperties": false
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *output_uri (Optional[str])*
- *type_hint (Literal['classification_evaluator'])*

field output_uri: Optional[str] = None

URI of directory where evaluator output will be saved. Evaluations for each scene-group will be save in a JSON file at <output_uri>/<scene-group-name>/eval.json. If None, and this Config is part of an RVPipeline, this field will be auto-generated.

field type_hint: Literal['classification_evaluator'] = 'classification_evaluator'

build(*class_config:* ClassConfig, *scene_group: Optional[Tuple[str, Iterable[str]]] = None*) → *Evaluator*

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

Parameters

- **class_config** (*ClassConfig*) –
- **scene_group** (*Optional[Tuple[str, Iterable[str]]]*) –

Return type
Evaluator

get_output_uri(*scene_group_name: Optional[str] = None*) → str

Parameters
scene_group_name (*Optional[str]*) –

Return type
str

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*pipeline: Optional[RVPipelineConfig] = None*) → None

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

Parameters
pipeline (*Optional[RVPipelineConfig]*) –

Return type
None

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

evaluation_item

Classes

EvaluationItem

EvaluationItem

class EvaluationItem

Bases: ABC

__init__()

Methods

__init__()	
merge(other)	Merges another item from a different scene into this one.
to_json()	

abstract merge(*other*)

Merges another item from a different scene into this one.

This is used to average metrics over scenes. Merges by taking a weighted average (by gt_count) of the metrics.

abstract to_json() → dict

Return type

dict

evaluator

Classes

Evaluator	Evaluates predictions for a set of scenes.

Evaluator

class Evaluator

Bases: ABC

Evaluates predictions for a set of scenes.

__init__()

Methods

__init__()

evaluate_predictions(ground_truth, predictions)	Evaluate predictions against ground truth.
evaluate_scene(scene)	Evaluate predictions from a scene's labels store.
process(scenes)	Evaluate all given scenes and save the evaluations.

abstract evaluate_predictions(*ground_truth:* Labels, *predictions:* Labels) → Any

Evaluate predictions against ground truth.

Parameters

- **ground_truth** (Labels) – Ground truth labels.
- **predictions** (Labels) – Predictions.

Returns
The evaluation.

Return type
Any

abstract evaluate_scene(*scene:* Scene) → Any

Evaluate predictions from a scene's labels store.

The predictions are evalated against ground truth labels from the scene's label source.

Parameters
scene (Scene) – A scene with a label source and a label store.

Returns
The evaluation.

Return type
ClassificationEvaluation

abstract process(*scenes: Iterable[Scene]*) → None

Evaluate all given scenes and save the evaluations.

Parameters
scenes (*Iterable[Scene]*) – Scenes to evaluate.

Return type
None

evaluator_config

Configs

EvaluatorConfig	Configure an *Evaluator*.

EvaluatorConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model EvaluatorConfig

Configure an *Evaluator*.


```
{
   "title": "EvaluatorConfig",
   "description": "Configure an :class:`.Evaluator`.",
   "type": "object",
   "properties": {
      "output_uri": {
         "title": "Output Uri",
         "description": "URI of directory where evaluator output will be saved.␣
↪Evaluations for each scene-group will be save in a JSON file at <output_uri>/
↪<scene-group-name>/eval.json. If None, and this Config is part of an RVPipeline,␣
↪this field will be auto-generated.",
         "type": "string"
      },
      "type_hint": {
         "title": "Type Hint",
         "default": "evaluator",
         "enum": [
            "evaluator"
         ],
         "type": "string"
      }
   },
   "additionalProperties": false
}
```


Config

- **extra**: *str = forbid*

- **validate_assignment**: *bool* = *True*

Fields

- *output_uri (Optional[str])*
- *type_hint (Literal['evaluator'])*

field output_uri: Optional[str] = None

URI of directory where evaluator output will be saved. Evaluations for each scene-group will be save in a JSON file at <output_uri>/<scene-group-name>/eval.json. If None, and this Config is part of an RVPipeline, this field will be auto-generated.

field type_hint: Literal['evaluator'] = 'evaluator'

build(*class_config:* ClassConfig, *scene_group: Optional[Tuple[str, Iterable[str]]] = None*) → *Evaluator*

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

Parameters

- **class_config** (*ClassConfig*) –
- **scene_group** (*Optional[Tuple[str, Iterable[str]]]*) –

Return type
Evaluator

get_output_uri(*scene_group_name: Optional[str] = None*) → str

Parameters
scene_group_name (*Optional[str]*) –

Return type
str

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*pipeline: Optional[RVPipelineConfig] = None*) → None

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

Parameters
pipeline (*Optional[RVPipelineConfig]*) –

Return type
None

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

object_detection_evaluation

Classes

ObjectDetectionEvaluation

ObjectDetectionEvaluation

class ObjectDetectionEvaluation

Bases: *ClassificationEvaluation*

__init__(*class_config: ClassConfig*, *iou_thresh: float = 0.5*)

Parameters

- **class_config** (*ClassConfig*) –
- **iou_thresh** (*float*) –

Methods

__init__(class_config[, iou_thresh])

compute(ground_truth_labels, prediction_labels)	Compute metrics for a single scene.
compute_avg()	Compute average metrics over all classes.
compute_eval_items(gt_labels, pred_labels, ...)	

merge(other[, scene_id])	Merge Evaluation for another Scene into this one.
reset()	Reset the Evaluation.
save(output_uri)	Save this Evaluation to a file.
to_json()	Serialize to a dict or list.

__init__(*class_config:* ClassConfig, *iou_thresh:* *float* = *0.5*)

Parameters

- **class_config** (ClassConfig) –
- **iou_thresh** (*float*) –

compute(*ground_truth_labels:* ObjectDetectionLabels, *prediction_labels:* ObjectDetectionLabels)

Compute metrics for a single scene.

Parameters

- **ground_truth_labels** (ObjectDetectionLabels) – Ground Truth labels to evaluate against.
- **prediction_labels** (ObjectDetectionLabels) – The predicted labels to evaluate.

compute_avg() → None

Compute average metrics over all classes.

Return type
None

static compute_eval_items(*gt_labels:* ObjectDetectionLabels, *pred_labels:* ObjectDetectionLabels, *class_config:* ClassConfig, *iou_thresh:* *float* = *0.5*) → Dict[int, *ClassEvaluationItem*]

Parameters

- **gt_labels** (ObjectDetectionLabels) –
- **pred_labels** (ObjectDetectionLabels) –
- **class_config** (ClassConfig) –
- **iou_thresh** (*float*) –

Return type
Dict[int, ClassEvaluationItem]

merge(*other:* ClassificationEvaluation, *scene_id:* *Optional[str]* = *None*) → None

Merge Evaluation for another Scene into this one.

This is useful for computing the average metrics of a set of scenes. The results of the averaging are stored in this Evaluation.

Parameters

- **other** (ClassificationEvaluation) – Evaluation to merge into this one
- **scene_id** (*Optional[str], optional*) – ID of scene. If specified, (a copy of) `other` will be saved and be availabel in `to_json()`'s output. Defaults to None.

Return type
None

reset()

Reset the Evaluation.

save(*output_uri:* *str*) → None

Save this Evaluation to a file.

Parameters
output_uri (*str*) – string URI for the file to write.

Return type
None

`to_json()` → Union[dict, list]

Serialize to a dict or list.

Returns

Class-wise and (if available) scene-wise evaluations.

Return type
Union[dict, list]

Functions

compute_metrics(gt_labels, pred_labels, ...)	

compute_metrics

compute_metrics(*gt_labels:* ObjectDetectionLabels, *pred_labels:* ObjectDetectionLabels, *num_classes: int*, *iou_thresh: float = 0.5*) → Tuple[ndarray, ndarray, ndarray]

Parameters

- **gt_labels** (*ObjectDetectionLabels*) –
- **pred_labels** (*ObjectDetectionLabels*) –
- **num_classes** (*int*) –
- **iou_thresh** (*float*) –

Return type
Tuple[*ndarray*, *ndarray*, *ndarray*]

object_detection_evaluator

Classes

ObjectDetectionEvaluator	Evaluates predictions for a set of scenes.

ObjectDetectionEvaluator

class ObjectDetectionEvaluator

Bases: *ClassificationEvaluator*

Evaluates predictions for a set of scenes.

__init__(*class_config*, *output_uri*)

Methods

__init__(class_config, output_uri)	
create_evaluation()	
evaluate_predictions(ground_truth, predictions)	Evaluate predictions against ground truth.
evaluate_scene(scene)	Evaluate predictions from a scene's labels store.
process(scenes[, tmp_dir])	Evaluate all given scenes and save the evaluations.

__init__(*class_config*, *output_uri*)

create_evaluation()

evaluate_predictions(*ground_truth:* Labels, *predictions:* Labels) → *ClassificationEvaluation*

Evaluate predictions against ground truth.

Parameters

- **ground_truth** (Labels) – Ground truth labels.
- **predictions** (Labels) – Predictions.

Returns
The evaluation.

Return type
Any

evaluate_scene(*scene:* Scene) → *ClassificationEvaluation*

Evaluate predictions from a scene's labels store.

The predictions are evalated against ground truth labels from the scene's label source.

Parameters
scene (Scene) – A scene with a label source and a label store.

Returns
The evaluation.

Return type
ClassificationEvaluation

process(*scenes:* *Iterable*[Scene], *tmp_dir:* *Optional*[*str*] = *None*) → None

Evaluate all given scenes and save the evaluations.

Parameters

- **scenes** (*Iterable*[Scene]) – Scenes to evaluate.
- **tmp_dir** (*Optional*[*str*]) –

Return type
None

object_detection_evaluator_config

Configs

ObjectDetectionEvaluatorConfig	Configure an *ObjectDetectionEvaluator*.

ObjectDetectionEvaluatorConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model ObjectDetectionEvaluatorConfig

Configure an *ObjectDetectionEvaluator*.


```
{
   "title": "ObjectDetectionEvaluatorConfig",
   "description": "Configure an :class:`.ObjectDetectionEvaluator`.",
   "type": "object",
   "properties": {
      "output_uri": {
         "title": "Output Uri",
         "description": "URI of directory where evaluator output will be saved.␣
↪Evaluations for each scene-group will be save in a JSON file at <output_uri>/
↪<scene-group-name>/eval.json. If None, and this Config is part of an RVPipeline,␣
↪this field will be auto-generated.",
         "type": "string"
      },
      "type_hint": {
         "title": "Type Hint",
         "default": "object_detection_evaluator",
         "enum": [
            "object_detection_evaluator"
         ],
         "type": "string"
      }
   },
   "additionalProperties": false
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *output_uri ()*
- *type_hint (Literal['object_detection_evaluator'])*

field output_uri: Optional[str] = None

URI of directory where evaluator output will be saved. Evaluations for each scene-group will be save in a JSON file at <output_uri>/<scene-group-name>/eval.json. If None, and this Config is part of an RVPipeline, this field will be auto-generated.

field type_hint: Literal['object_detection_evaluator'] = 'object_detection_evaluator'

build(*class_config:* ClassConfig, *scene_group: Optional[Tuple[str, Iterable[str]]] = None*) → *ObjectDetectionEvaluator*

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

Parameters

- **class_config** (*ClassConfig*) –
- **scene_group** (*Optional[Tuple[str, Iterable[str]]]*) –

Return type
ObjectDetectionEvaluator

get_output_uri(*scene_group_name: Optional[str] = None*) → str

Parameters
scene_group_name (*Optional[str]*) –

Return type
str

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*pipeline: Optional[RVPipelineConfig] = None*) → None

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

Parameters
pipeline (*Optional[RVPipelineConfig]*) –

Return type
None

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

semantic_segmentation_evaluation

Classes

SemanticSegmentationEvaluation	Evaluation for semantic segmentation.

SemanticSegmentationEvaluation

class SemanticSegmentationEvaluation

Bases: *ClassificationEvaluation*

Evaluation for semantic segmentation.

__init__(*class_config: ClassConfig*)

Parameters

class_config (*ClassConfig*) –

Methods

__init__(class_config)

compute(gt_labels, pred_labels)	Compute metrics for a single scene.
compute_avg()	Compute average metrics over all classes.
merge(other[, scene_id])	Merge Evaluation for another Scene into this one.
reset()	Reset the Evaluation.
save(output_uri)	Save this Evaluation to a file.
to_json()	Serialize to a dict or list.

__init__(*class_config: ClassConfig*)

Parameters

class_config (*ClassConfig*) –

compute(*gt_labels: SemanticSegmentationLabels*, *pred_labels: SemanticSegmentationLabels*) → None

Compute metrics for a single scene.

Parameters

- **ground_truth_labels** – Ground Truth labels to evaluate against.
- **prediction_labels** – The predicted labels to evaluate.

- **gt_labels** (SemanticSegmentationLabels) –
- **pred_labels** (SemanticSegmentationLabels) –

Return type
None

compute_avg() → None

Compute average metrics over all classes.

Return type
None

merge(*other:* ClassificationEvaluation, *scene_id: Optional[str] = None*) → None

Merge Evaluation for another Scene into this one.

This is useful for computing the average metrics of a set of scenes. The results of the averaging are stored in this Evaluation.

Parameters

- **other** (ClassificationEvaluation) – Evaluation to merge into this one
- **scene_id** (*Optional[str], optional*) – ID of scene. If specified, (a copy of) other will be saved and be availabel in to_json()'s output. Defaults to None.

Return type
None

reset()

Reset the Evaluation.

save(*output_uri: str*) → None

Save this Evaluation to a file.

Parameters
output_uri (*str*) – string URI for the file to write.

Return type
None

to_json() → Union[dict, list]

Serialize to a dict or list.

Returns

Class-wise and (if available) scene-wise
evaluations.

Return type
Union[dict, list]

semantic_segmentation_evaluator

Classes

SemanticSegmentationEvaluator	Evaluates predictions for a set of scenes.

SemanticSegmentationEvaluator

class SemanticSegmentationEvaluator

Bases: *ClassificationEvaluator*

Evaluates predictions for a set of scenes.

__init__(*class_config:* ClassConfig, *output_uri:* *Optional[str]* = *None*)

Parameters

- **class_config** (ClassConfig) –
- **output_uri** (*Optional[str]*) –

Methods

__init__(class_config[, output_uri])	
create_evaluation()	
evaluate_predictions(ground_truth, predictions)	Evaluate predictions against ground truth.
evaluate_scene(scene)	Override to pass null_class_id to filter_by_aoi().
process(scenes[, tmp_dir])	Evaluate all given scenes and save the evaluations.

__init__(*class_config:* ClassConfig, *output_uri:* *Optional[str]* = *None*)

Parameters

- **class_config** (ClassConfig) –
- **output_uri** (*Optional[str]*) –

create_evaluation() → *SemanticSegmentationEvaluation*

Return type
SemanticSegmentationEvaluation

evaluate_predictions(*ground_truth:* Labels, *predictions:* Labels) → *ClassificationEvaluation*

Evaluate predictions against ground truth.

Parameters

- **ground_truth** (Labels) – Ground truth labels.
- **predictions** (Labels) – Predictions.

Returns
The evaluation.

Return type
Any

evaluate_scene(*scene:* Scene) → *SemanticSegmentationEvaluation*

Override to pass null_class_id to filter_by_aoi().

Parameters
scene (Scene) –

Return type
SemanticSegmentationEvaluation

process(*scenes: Iterable[*Scene*]*, *tmp_dir: Optional[str] = None*) → None

Evaluate all given scenes and save the evaluations.

Parameters

- **scenes** (*Iterable[*Scene*]*) – Scenes to evaluate.
- **tmp_dir** (*Optional[str]*) –

Return type
None

semantic_segmentation_evaluator_config

Configs

SemanticSegmentationEvaluatorConfig	Configure a *SemanticSegmentationEvaluator*.

SemanticSegmentationEvaluatorConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model SemanticSegmentationEvaluatorConfig

Configure a *SemanticSegmentationEvaluator*.


```
{
   "title": "SemanticSegmentationEvaluatorConfig",
   "description": "Configure a :class:`.SemanticSegmentationEvaluator`.",
   "type": "object",
   "properties": {
      "output_uri": {
         "title": "Output Uri",
         "description": "URI of directory where evaluator output will be saved.␣
↪Evaluations for each scene-group will be save in a JSON file at <output_uri>/
↪<scene-group-name>/eval.json. If None, and this Config is part of an RVPipeline,␣
↪this field will be auto-generated.",
         "type": "string"
      },
      "type_hint": {
         "title": "Type Hint",
```


(continues on next page)

(continued from previous page)

```
            "default": "semantic_segmentation_evaluator",
            "enum": [
               "semantic_segmentation_evaluator"
            ],
            "type": "string"
         }
      },
      "additionalProperties": false
}
```

Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *output_uri ()*
- *type_hint (Literal['semantic_segmentation_evaluator'])*

field output_uri: Optional[str] = None

URI of directory where evaluator output will be saved. Evaluations for each scene-group will be save in a JSON file at <output_uri>/<scene-group-name>/eval.json. If None, and this Config is part of an RVPipeline, this field will be auto-generated.

field type_hint: Literal['semantic_segmentation_evaluator'] = 'semantic_segmentation_evaluator'

build(*class_config:* ClassConfig, *scene_group: Optional[Tuple[str, Iterable[str]]] = None*) → *SemanticSegmentationEvaluator*

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

Parameters

- **class_config** (ClassConfig) –
- **scene_group** (*Optional[Tuple[str, Iterable[str]]]*) –

Return type
SemanticSegmentationEvaluator

get_output_uri(*scene_group_name: Optional[str] = None*) → str

Parameters
scene_group_name (*Optional[str]*) –

Return type
str

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*pipeline: Optional[RVPipelineConfig] = None*) → None

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

Parameters
pipeline (*Optional[RVPipelineConfig]*) –

Return type
None

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises
ConfigError – if field is invalid

9.2.8 predictor

Classes

Predictor	Class for making predictions based off of a model bundle.

Predictor

class Predictor

Bases: object

Class for making predictions based off of a model bundle.

__init__(*model_bundle_uri: str*, *tmp_dir: str*, *update_stats: bool = False*, *channel_order: Optional[List[int]] = None*, *scene_group: Optional[str] = None*)

Creates a new Predictor.

Parameters

- **model_bundle_uri** (*str*) – URI of the model bundle to use. Can be any type of URI that Raster Vision can read.
- **tmp_dir** (*str*) – Temporary directory in which to store files that are used by the Predictor. This directory is not cleaned up by this class.
- **channel_order** (*Optional[List[int]]*) – Option for a new channel order to use for the imagery being predicted against. If not present, the channel_order from the original configuration in the predict package will be used.
- **update_stats** (*bool*) –
- **scene_group** (*Optional[str]*) –

Methods

__init__(model_bundle_uri, tmp_dir[, ...])	Creates a new Predictor.
predict(image_uris, label_uri)	Generate predictions for the given image.

__init__(*model_bundle_uri: str*, *tmp_dir: str*, *update_stats: bool = False*, *channel_order: Optional[List[int]] = None*, *scene_group: Optional[str] = None*)

Creates a new Predictor.

Parameters

- **model_bundle_uri** (*str*) – URI of the model bundle to use. Can be any type of URI that Raster Vision can read.
- **tmp_dir** (*str*) – Temporary directory in which to store files that are used by the Predictor. This directory is not cleaned up by this class.
- **channel_order** (*Optional[List[int]]*) – Option for a new channel order to use for the imagery being predicted against. If not present, the channel_order from the original configuration in the predict package will be used.
- **update_stats** (*bool*) –
- **scene_group** (*Optional[str]*) –

predict(*image_uris: List[str]*, *label_uri: str*) → None

Generate predictions for the given image.

Parameters

- **image_uris** (*List[str]*) – URIs of the images to make predictions against. This can be any type of URI readable by Raster Vision FileSystems.
- **label_uri** (*str*) – URI to save labels off into

Return type
None

9.2.9 raster_stats

Classes

RasterStats

RasterStats

class RasterStats

Bases: object

__init__()

Methods

__init__()	
compute(raster_sources[, sample_prob])	Compute the mean and stds over all the raster_sources.
load(stats_uri)	
save(stats_uri)	

__init__()

compute(*raster_sources: Sequence[RasterSource]*, *sample_prob: Optional[float] = None*) → None

Compute the mean and stds over all the raster_sources.

This ignores NODATA values.

If sample_prob is set, then a subset of each scene is used to compute stats which speeds up the computation. Roughly speaking, if sample_prob=0.5, then half the pixels in the scene will be used. More precisely, the number of chips is equal to sample_prob * (width * height / 300^2), or 1, whichever is greater. Each chip is uniformly sampled from the scene with replacement. Otherwise, it uses a sliding window over the entire scene to compute stats.

Parameters

- **raster_sources** (*Sequence[RasterSource]*) – list of RasterSource
- **sample_prob** (*Optional[float]*) – (float or None) between 0 and 1

Return type
None

static load(*stats_uri: str*) → None

Parameters
stats_uri (*str*) –

Return type
None

save(*stats_uri: str*) → None

Parameters
stats_uri (*str*) –

Return type
None

Functions

parallel_mean(mean_a, count_a, mean_b, count_b)	Compute the mean based on stats from two partitions of the data.
parallel_variance(mean_a, count_a, var_a, ...)	Compute the variance based on stats from two partitions of the data.

parallel_mean

parallel_mean(*mean_a*, *count_a*, *mean_b*, *count_b*)

Compute the mean based on stats from two partitions of the data.

See "Parallel Algorithm" in https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance

Parameters

- **mean_a** – the mean of partition a
- **count_a** – the number of elements in partition a
- **mean_b** – the mean of partition b
- **count_b** – the number of elements in partition b

Returns
the mean of the two partitions if they were combined

parallel_variance

parallel_variance(*mean_a*, *count_a*, *var_a*, *mean_b*, *count_b*, *var_b*)

Compute the variance based on stats from two partitions of the data.

See "Parallel Algorithm" in https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance

Parameters

- **mean_a** – the mean of partition a
- **count_a** – the number of elements in partition a
- **var_a** – the variance of partition a
- **mean_b** – the mean of partition b
- **count_b** – the number of elements in partition b
- **var_b** – the variance of partition b

Returns
the variance of the two partitions if they were combined

9.2.10 rv_pipeline

Modules

chip_classification
chip_classification_config
object_detection
object_detection_config
rv_pipeline
rv_pipeline_config
semantic_segmentation
semantic_segmentation_config
utils

chip_classification

Classes

ChipClassification

ChipClassification

class ChipClassification

Bases: *RVPipeline*

Attributes

commands	Built-in mutable sequence.
gpu_commands	Built-in mutable sequence.
split_commands	Built-in mutable sequence.

__init__(*config:* RVPipelineConfig, *tmp_dir:* *str*)

Constructor

Parameters

- **config** (*RVPipelineConfig*) – the configuration of this pipeline
- **tmp_dir** (*str*) – the root any temporary directories created by running this pipeline

Methods

__init__(config, tmp_dir)	Constructor
analyze()	Run each analyzer over training scenes.
bundle()	Save a model bundle with whatever is needed to make predictions.
chip([split_ind, num_splits])	Save training and validation chips.
eval()	Evaluate predictions against ground truth.
get_train_labels(window, scene)	Return the training labels in a window for a scene.
get_train_windows(scene)	Return the training windows for a Scene.
post_process_batch(windows, chips, labels)	Post-process a batch of predictions.
post_process_predictions(labels, scene)	Post-process all labels at end of prediction.
post_process_sample(sample)	Post-process sample in pipeline-specific way.
predict([split_ind, num_splits])	Make predictions over each validation and test scene.
predict_scene(scene, backend)	
test_cpu([split_ind, num_splits])	A command to test the ability to run split jobs on CPU.
test_gpu()	A command to test the ability to run on GPU.
train()	Train a model and save it.

__init__(*config:* RVPipelineConfig, *tmp_dir:* *str*)

Constructor

Parameters

- **config** (RVPipelineConfig) – the configuration of this pipeline
- **tmp_dir** (*str*) – the root any temporary directories created by running this pipeline

analyze()

Run each analyzer over training scenes.

bundle()

Save a model bundle with whatever is needed to make predictions.

The model bundle is a zip file and it is used by the Predictor and predict CLI subcommand.

chip(*split_ind:* *int* = *0*, *num_splits:* *int* = *1*)

Save training and validation chips.

Parameters

- **split_ind** (*int*) –
- **num_splits** (*int*) –

eval()

Evaluate predictions against ground truth.

get_train_labels(*window:* Box, *scene:* Scene)

Return the training labels in a window for a scene.

Returns

Labels that lie within window

Parameters

- **window** (Box) –

- **scene** (Scene) –

get_train_windows(*scene:* Scene) → List[*Box*]

Return the training windows for a Scene.

Each training window represents the spatial extent of a training chip to generate.

Parameters
scene (Scene) – Scene to generate windows for

Return type
List[Box]

post_process_batch(*windows: List[*Box*]*, *chips: ndarray*, *labels:* Labels) → *Labels*

Post-process a batch of predictions.

Parameters

- **windows** (*List[*Box*]*) –
- **chips** (*ndarray*) –
- **labels** (Labels) –

Return type
Labels

post_process_predictions(*labels:* Labels, *scene:* Scene) → *Labels*

Post-process all labels at end of prediction.

Parameters

- **labels** (Labels) –
- **scene** (Scene) –

Return type
Labels

post_process_sample(*sample: DataSample*) → DataSample

Post-process sample in pipeline-specific way.

This should be called before writing a sample during chipping.

Parameters
sample (*DataSample*) –

Return type
DataSample

predict(*split_ind=0*, *num_splits=1*)

Make predictions over each validation and test scene.

This uses a sliding window.

predict_scene(*scene:* Scene, *backend:* Backend) → *Labels*

Parameters

- **scene** (Scene) –
- **backend** (Backend) –

Return type
Labels

test_cpu(*split_ind: int = 0*, *num_splits: int = 1*)

A command to test the ability to run split jobs on CPU.

Parameters

- **split_ind** (*int*) –
- **num_splits** (*int*) –

test_gpu()

A command to test the ability to run on GPU.

train()

Train a model and save it.

property commands

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

property gpu_commands

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

property split_commands

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

Functions

get_train_windows(scene, chip_size[, ...])

get_train_windows

get_train_windows(*scene: Scene*, *chip_size: int*, *chip_nodata_threshold: float = 1.0*) → List[*Box*]

Parameters

- **scene** (Scene) –
- **chip_size** (*int*) –
- **chip_nodata_threshold** (*float*) –

Return type
List[Box]

chip_classification_config

Configs

ChipClassificationConfig	Configure a *ChipClassification* pipeline.

ChipClassificationConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model ChipClassificationConfig

Configure a *ChipClassification* pipeline.


```
{
   "title": "ChipClassificationConfig",
   "description": "Configure a :class:`.ChipClassification` pipeline.",
   "type": "object",
   "properties": {
      "root_uri": {
         "title": "Root Uri",
         "description": "The root URI for output generated by the pipeline",
         "type": "string"
      },
      "rv_config": {
         "title": "Rv Config",
         "description": "Used to store serialized RVConfig so pipeline can run in␣
↪remote environment with the local RVConfig. This should not be set explicitly by␣
↪users -- it is only used by the runner when running a remote pipeline.",
         "type": "object"
      },
      "plugin_versions": {
         "title": "Plugin Versions",
         "description": "Used to store a mapping of plugin module paths to the␣
↪latest version number. This should not be set explicitly by users -- it is set␣
↪automatically when serializing and saving the config to disk.",
         "type": "object",
         "additionalProperties": {
            "type": "integer"
         }
      },
      "type_hint": {
         "title": "Type Hint",
         "default": "chip_classification",
         "enum": [
            "chip_classification"
         ],
         "type": "string"
      },
      "dataset": {
         "title": "Dataset",
```


(continues on next page)

(continued from previous page)


```
        "description": "Dataset containing train, validation, and optional test␣
↪scenes.",
        "allOf": [
           {
              "$ref": "#/definitions/DatasetConfig"
           }
        ]
     },
     "backend": {
        "title": "Backend",
        "description": "Backend to use for interfacing with ML library.",
        "allOf": [
           {
              "$ref": "#/definitions/BackendConfig"
           }
        ]
     },
     "evaluators": {
        "title": "Evaluators",
        "description": "Evaluators to run during analyzer command. If list is␣
↪empty the default evaluator is added.",
        "default": [],
        "type": "array",
        "items": {
           "$ref": "#/definitions/EvaluatorConfig"
        }
     },
     "analyzers": {
        "title": "Analyzers",
        "description": "Analyzers to run during analyzer command. A StatsAnalyzer␣
↪will be added automatically if any scenes have a RasterTransformer.",
        "default": [],
        "type": "array",
        "items": {
           "$ref": "#/definitions/AnalyzerConfig"
        }
     },
     "train_chip_sz": {
        "title": "Train Chip Sz",
        "description": "Size of training chips in pixels.",
        "default": 300,
        "type": "integer"
     },
     "predict_chip_sz": {
        "title": "Predict Chip Sz",
        "description": "Size of predictions chips in pixels.",
        "default": 300,
        "type": "integer"
     },
     "predict_batch_sz": {
        "title": "Predict Batch Sz",
        "description": "Batch size to use during prediction.",
```


(continues on next page)

(continued from previous page)

```
        "default": 8,
        "type": "integer"
      },
      "chip_nodata_threshold": {
        "title": "Chip Nodata Threshold",
        "description": "Discard chips where the proportion of NODATA values is␣
↪greater than or equal to this value. Might result in false positives if there are␣
↪many legitimate black pixels in the chip. Use with caution.",
        "default": 1,
        "minimum": 0,
        "maximum": 1,
        "type": "number"
      },
      "analyze_uri": {
        "title": "Analyze Uri",
        "description": "URI for output of analyze. If None, will be auto-generated.
↪",
        "type": "string"
      },
      "chip_uri": {
        "title": "Chip Uri",
        "description": "URI for output of chip. If None, will be auto-generated.",
        "type": "string"
      },
      "train_uri": {
        "title": "Train Uri",
        "description": "URI for output of train. If None, will be auto-generated.",
        "type": "string"
      },
      "predict_uri": {
        "title": "Predict Uri",
        "description": "URI for output of predict. If None, will be auto-generated.
↪",
        "type": "string"
      },
      "eval_uri": {
        "title": "Eval Uri",
        "description": "URI for output of eval. If None, will be auto-generated.",
        "type": "string"
      },
      "bundle_uri": {
        "title": "Bundle Uri",
        "description": "URI for output of bundle. If None, will be auto-generated.
↪",
        "type": "string"
      },
      "source_bundle_uri": {
        "title": "Source Bundle Uri",
        "description": "If provided, the model will be loaded from this bundle for␣
↪the train stage. Useful for fine-tuning.",
        "type": "string"
      }
```

(continues on next page)

(continued from previous page)


```
    },
    "required": [
        "dataset",
        "backend"
    ],
    "additionalProperties": false,
    "definitions": {
        "ClassConfig": {
            "title": "ClassConfig",
            "description": "Configure class information for a machine learning task.",
            "type": "object",
            "properties": {
                "names": {
                    "title": "Names",
                    "description": "Names of classes. The i-th class in this list will␣
↪have class ID = i.",
                    "type": "array",
                    "items": {
                        "type": "string"
                    }
                },
                "colors": {
                    "title": "Colors",
                    "description": "Colors used to visualize classes. Can be color␣
↪strings accepted by matplotlib or RGB tuples. If None, a random color will be␣
↪auto-generated for each class.",
                    "type": "array",
                    "items": {
                        "anyOf": [
                            {
                                "type": "string"
                            },
                            {
                                "type": "array",
                                "items": {}
                            }
                        ]
                    }
                },
                "null_class": {
                    "title": "Null Class",
                    "description": "Optional name of class in `names` to use as the null␣
↪class. This is used in semantic segmentation to represent the label for imagery␣
↪pixels that are NODATA or that are missing a label. If None and the class names␣
↪include \"null\", it will automatically be used as the null class. If None, and␣
↪this Config is part of a SemanticSegmentationConfig, a null class will be added␣
↪automatically.",
                    "type": "string"
                },
                "type_hint": {
                    "title": "Type Hint",
                    "default": "class_config",
```


(continues on next page)

(continued from previous page)


```
                "enum": [
                    "class_config"
                ],
                "type": "string"
            }
        },
        "required": [
            "names"
        ],
        "additionalProperties": false
    },
    "RasterTransformerConfig": {
        "title": "RasterTransformerConfig",
        "description": "Configure a :class:`.RasterTransformer`.",
        "type": "object",
        "properties": {
            "type_hint": {
                "title": "Type Hint",
                "default": "raster_transformer",
                "enum": [
                    "raster_transformer"
                ],
                "type": "string"
            }
        },
        "additionalProperties": false
    },
    "RasterSourceConfig": {
        "title": "RasterSourceConfig",
        "description": "Configure a :class:`.RasterSource`.",
        "type": "object",
        "properties": {
            "channel_order": {
                "title": "Channel Order",
                "description": "The sequence of channel indices to use when reading␣
↪imagery.",
                "type": "array",
                "items": {
                    "type": "integer"
                }
            },
            "transformers": {
                "title": "Transformers",
                "default": [],
                "type": "array",
                "items": {
                    "$ref": "#/definitions/RasterTransformerConfig"
                }
            },
            "extent": {
                "title": "Extent",
                "description": "Use-specified extent in pixel coords in the form␣
```


(continues on next page)

(continued from previous page)

```
↪(ymin, xmin, ymax, xmax). Useful for cropping the raster source so that only part␣
↪of the raster is read from.",
                "type": "array",
                "minItems": 4,
                "maxItems": 4,
                "items": [
                   {
                      "type": "integer"
                   },
                   {
                      "type": "integer"
                   },
                   {
                      "type": "integer"
                   },
                   {
                      "type": "integer"
                   }
                ]
             },
             "type_hint": {
                "title": "Type Hint",
                "default": "raster_source",
                "enum": [
                   "raster_source"
                ],
                "type": "string"
             }
          },
          "additionalProperties": false
       },
       "LabelSourceConfig": {
          "title": "LabelSourceConfig",
          "description": "Configure a :class:`.LabelSource`.",
          "type": "object",
          "properties": {
             "type_hint": {
                "title": "Type Hint",
                "default": "label_source",
                "enum": [
                   "label_source"
                ],
                "type": "string"
             }
          },
          "additionalProperties": false
       },
       "LabelStoreConfig": {
          "title": "LabelStoreConfig",
          "description": "Configure a :class:`.LabelStore`.",
          "type": "object",
          "properties": {
```

(continues on next page)

(continued from previous page)


```
            "type_hint": {
               "title": "Type Hint",
               "default": "label_store",
               "enum": [
                  "label_store"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      },
      "SceneConfig": {
         "title": "SceneConfig",
         "description": "Configure a :class:`.Scene` comprising raster data &␣
↪labels for an AOI.\n    ",
         "type": "object",
         "properties": {
            "id": {
               "title": "Id",
               "type": "string"
            },
            "raster_source": {
               "$ref": "#/definitions/RasterSourceConfig"
            },
            "label_source": {
               "$ref": "#/definitions/LabelSourceConfig"
            },
            "label_store": {
               "$ref": "#/definitions/LabelStoreConfig"
            },
            "aoi_uris": {
               "title": "Aoi Uris",
               "description": "List of URIs of GeoJSON files that define the AOIs␣
↪for the scene. Each polygon defines an AOI which is a piece of the scene that is␣
↪assumed to be fully labeled and usable for training or validation. The AOIs are␣
↪assumed to be in EPSG:4326 coordinates.",
               "type": "array",
               "items": {
                  "type": "string"
               }
            },
            "type_hint": {
               "title": "Type Hint",
               "default": "scene",
               "enum": [
                  "scene"
               ],
               "type": "string"
            }
         },
         "required": [
            "id",
```


(continues on next page)

(continued from previous page)

```
            "raster_source"
         ],
         "additionalProperties": false
      },
      "DatasetConfig": {
         "title": "DatasetConfig",
         "description": "Configure train, validation, and test splits for a dataset.
↪",
         "type": "object",
         "properties": {
            "class_config": {
               "$ref": "#/definitions/ClassConfig"
            },
            "train_scenes": {
               "title": "Train Scenes",
               "type": "array",
               "items": {
                  "$ref": "#/definitions/SceneConfig"
               }
            },
            "validation_scenes": {
               "title": "Validation Scenes",
               "type": "array",
               "items": {
                  "$ref": "#/definitions/SceneConfig"
               }
            },
            "test_scenes": {
               "title": "Test Scenes",
               "default": [],
               "type": "array",
               "items": {
                  "$ref": "#/definitions/SceneConfig"
               }
            },
            "scene_groups": {
               "title": "Scene Groups",
               "description": "Groupings of scenes. Should be a dict of the form: {
↪<group-name>: Set(scene_id_1, scene_id_2, ...)}. Three groups are added by␣
↪default: \"train_scenes\", \"validation_scenes\", and \"test_scenes\"",
               "default": {},
               "type": "object",
               "additionalProperties": {
                  "type": "array",
                  "items": {
                     "type": "string"
                  },
                  "uniqueItems": true
               }
            },
            "type_hint": {
               "title": "Type Hint",
```

(continues on next page)

(continued from previous page)


```
                "default": "dataset",
                "enum": [
                   "dataset"
                ],
                "type": "string"
             }
          },
          "required": [
             "class_config",
             "train_scenes",
             "validation_scenes"
          ],
          "additionalProperties": false
       },
       "BackendConfig": {
          "title": "BackendConfig",
          "description": "Configure a :class:`.Backend`.",
          "type": "object",
          "properties": {
             "type_hint": {
                "title": "Type Hint",
                "default": "backend",
                "enum": [
                   "backend"
                ],
                "type": "string"
             }
          },
          "additionalProperties": false
       },
       "EvaluatorConfig": {
          "title": "EvaluatorConfig",
          "description": "Configure an :class:`.Evaluator`.",
          "type": "object",
          "properties": {
             "output_uri": {
                "title": "Output Uri",
                "description": "URI of directory where evaluator output will be␣
↪saved. Evaluations for each scene-group will be save in a JSON file at <output_
↪uri>/<scene-group-name>/eval.json. If None, and this Config is part of an␣
↪RVPipeline, this field will be auto-generated.",
                "type": "string"
             },
             "type_hint": {
                "title": "Type Hint",
                "default": "evaluator",
                "enum": [
                   "evaluator"
                ],
                "type": "string"
             }
          },
```


(continues on next page)

(continued from previous page)


```
            "additionalProperties": false
        },
        "AnalyzerConfig": {
            "title": "AnalyzerConfig",
            "description": "Configure an :class:`.Analyzer`.",
            "type": "object",
            "properties": {
                "type_hint": {
                    "title": "Type Hint",
                    "default": "analyzer",
                    "enum": [
                        "analyzer"
                    ],
                    "type": "string"
                }
            },
            "additionalProperties": false
        }
    }
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *analyze_uri (Optional[str])*
- *analyzers (List[rastervision.core.analyzer.analyzer_config.AnalyzerConfig])*
- *backend (rastervision.core.backend.backend_config.BackendConfig)*
- *bundle_uri (Optional[str])*
- *chip_nodata_threshold (rastervision.core.utils.misc.ConstrainedFloatValue)*
- *chip_uri (Optional[str])*
- *dataset (rastervision.core.data.dataset_config.DatasetConfig)*
- *eval_uri (Optional[str])*
- *evaluators (List[rastervision.core.evaluation.evaluator_config.EvaluatorConfig])*
- *plugin_versions (Optional[Dict[str, int]])*
- *predict_batch_sz (int)*
- *predict_chip_sz (int)*
- *predict_uri (Optional[str])*
- *root_uri (str)*
- *rv_config (dict)*

- *source_bundle_uri (Optional[str])*
- *train_chip_sz (int)*
- *train_uri (Optional[str])*
- *type_hint (Literal['chip_classification'])*

field analyze_uri: Optional[str] = None

URI for output of analyze. If None, will be auto-generated.

field analyzers: List[*AnalyzerConfig*] = []

Analyzers to run during analyzer command. A StatsAnalyzer will be added automatically if any scenes have a RasterTransformer.

field backend: *BackendConfig* [Required]

Backend to use for interfacing with ML library.

field bundle_uri: Optional[str] = None

URI for output of bundle. If None, will be auto-generated.

field chip_nodata_threshold: Proportion = 1

Discard chips where the proportion of NODATA values is greater than or equal to this value. Might result in false positives if there are many legitimate black pixels in the chip. Use with caution.

Constraints

- **minimum** = 0
- **maximum** = 1

field chip_uri: Optional[str] = None

URI for output of chip. If None, will be auto-generated.

field dataset: *DatasetConfig* [Required]

Dataset containing train, validation, and optional test scenes.

field eval_uri: Optional[str] = None

URI for output of eval. If None, will be auto-generated.

field evaluators: List[*EvaluatorConfig*] = []

Evaluators to run during analyzer command. If list is empty the default evaluator is added.

field plugin_versions: Optional[Dict[str, int]] = None

Used to store a mapping of plugin module paths to the latest version number. This should not be set explicitly by users – it is set automatically when serializing and saving the config to disk.

field predict_batch_sz: int = 8

Batch size to use during prediction.

field predict_chip_sz: int = 300

Size of predictions chips in pixels.

field predict_uri: Optional[str] = None

URI for output of predict. If None, will be auto-generated.

field root_uri: str = None

The root URI for output generated by the pipeline

field rv_config: dict = None

Used to store serialized RVConfig so pipeline can run in remote environment with the local RVConfig. This should not be set explicitly by users – it is only used by the runner when running a remote pipeline.

field source_bundle_uri: Optional[str] = None

If provided, the model will be loaded from this bundle for the train stage. Useful for fine-tuning.

field train_chip_sz: int = 300

Size of training chips in pixels.

field train_uri: Optional[str] = None

URI for output of train. If None, will be auto-generated.

field type_hint: Literal['chip_classification'] = 'chip_classification'

build(*tmp_dir*)

Return a pipeline based on this configuration.

Subclasses should override this to return an instance of the corresponding subclass of Pipeline.

Parameters
tmp_dir – root of any temporary directory to pass to pipeline

get_config_uri() → str

Get URI of serialized version of this PipelineConfig.

Return type
str

get_default_evaluator()

Returns a default EvaluatorConfig to use if one isn't set.

get_default_label_store(*scene*)

Returns a default LabelStoreConfig to fill in any missing ones.

get_model_bundle_uri()

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update()

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

object_detection

Classes

ObjectDetection

ObjectDetection

class ObjectDetection

Bases: *RVPipeline*

Attributes

commands	Built-in mutable sequence.
gpu_commands	Built-in mutable sequence.
split_commands	Built-in mutable sequence.

__init__(*config:* RVPipelineConfig, *tmp_dir: str*)

Constructor

Parameters

- **config** (RVPipelineConfig) – the configuration of this pipeline
- **tmp_dir** (*str*) – the root any temporary directories created by running this pipeline

Methods

__init__(config, tmp_dir)	Constructor
analyze()	Run each analyzer over training scenes.
bundle()	Save a model bundle with whatever is needed to make predictions.
chip([split_ind, num_splits])	Save training and validation chips.
eval()	Evaluate predictions against ground truth.
get_train_labels(window, scene)	Return the training labels in a window for a scene.
get_train_windows(scene)	Return the training windows for a Scene.
post_process_batch(windows, chips, labels)	Post-process a batch of predictions.
post_process_predictions(labels, scene)	Post-process all labels at end of prediction.
post_process_sample(sample)	Post-process sample in pipeline-specific way.
predict([split_ind, num_splits])	Make predictions over each validation and test scene.
predict_scene(scene, backend)	
test_cpu([split_ind, num_splits])	A command to test the ability to run split jobs on CPU.
test_gpu()	A command to test the ability to run on GPU.
train()	Train a model and save it.

__init__(*config:* RVPipelineConfig, *tmp_dir:* *str*)

Constructor

Parameters

- **config** (RVPipelineConfig) – the configuration of this pipeline
- **tmp_dir** (*str*) – the root any temporary directories created by running this pipeline

analyze()

Run each analyzer over training scenes.

bundle()

Save a model bundle with whatever is needed to make predictions.

The model bundle is a zip file and it is used by the Predictor and predict CLI subcommand.

chip(*split_ind:* *int* = *0*, *num_splits:* *int* = *1*)

Save training and validation chips.

Parameters

- **split_ind** (*int*) –
- **num_splits** (*int*) –

eval()

Evaluate predictions against ground truth.

get_train_labels(*window*, *scene*)

Return the training labels in a window for a scene.

Returns

Labels that lie within window

get_train_windows(*scene*)

Return the training windows for a Scene.

Each training window represents the spatial extent of a training chip to generate.

Parameters
scene – Scene to generate windows for

post_process_batch(*windows: List[Box]*, *chips: ndarray*, *labels:* Labels) → *Labels*

Post-process a batch of predictions.

Parameters

- **windows** (*List[Box]*) –
- **chips** (*ndarray*) –
- **labels** (Labels) –

Return type
Labels

post_process_predictions(*labels*, *scene*)

Post-process all labels at end of prediction.

post_process_sample(*sample: DataSample*) → DataSample

Post-process sample in pipeline-specific way.

This should be called before writing a sample during chipping.

Parameters
sample (*DataSample*) –

Return type
DataSample

predict(*split_ind=0*, *num_splits=1*)

Make predictions over each validation and test scene.

This uses a sliding window.

predict_scene(*scene:* Scene, *backend:* Backend) → *Labels*

Parameters

- **scene** (Scene) –
- **backend** (Backend) –

Return type
Labels

test_cpu(*split_ind: int = 0*, *num_splits: int = 1*)

A command to test the ability to run split jobs on CPU.

Parameters

- **split_ind** (*int*) –
- **num_splits** (*int*) –

test_gpu()

A command to test the ability to run on GPU.

train()

Train a model and save it.

property commands

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

property gpu_commands

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

property split_commands

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

Functions

get_train_windows(scene, chip_opts, chip_size)	
make_neg_windows(raster_source, label_store, ...)	
make_pos_windows(image_extent, label_store, ...)	

get_train_windows

get_train_windows(*scene*, *chip_opts*, *chip_size*, *chip_nodata_threshold=1.0*)

make_neg_windows

make_neg_windows(*raster_source*, *label_store*, *chip_size*, *nb_windows*, *max_attempts*, *filter_windows*, *chip_nodata_threshold=1.0*)

make_pos_windows

make_pos_windows(*image_extent*, *label_store*, *chip_size*, *window_method*, *label_buffer*)

object_detection_config

Classes

ObjectDetectionWindowMethod	Enum for window methods

ObjectDetectionWindowMethod

class ObjectDetectionWindowMethod

Bases: Enum

Enum for window methods

chip

the default method

Attributes

chip
label
image
sliding

__init__()

chip = 'chip'

image = 'image'

label = 'label'

sliding = 'sliding'

Configs

ObjectDetectionChipOptions	
ObjectDetectionConfig	Configure an *ObjectDetection* pipeline.
ObjectDetectionPredictOptions	

ObjectDetectionChipOptions

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model ObjectDetectionChipOptions


```
{
   "title": "ObjectDetectionChipOptions",
   "description": "Base class that can be extended to provide custom configurations.
↪\n\nThis adds some extra methods to Pydantic BaseModel.\nSee https://pydantic-
↪docs.helpmanual.io/\n\nThe general idea is that configuration schemas can be␣
↪defined by\nsubclassing this and adding class attributes with types and\ndefault␣
↪values for each field. Configs can be defined hierarchically,\nie. a Config can␣
↪have fields which are of type Config.\nValidation, serialization, deserialization,
↪ and IDE support is\nprovided automatically based on this schema.",
   "type": "object",
   "properties": {
      "neg_ratio": {
         "title": "Neg Ratio",
         "description": "The ratio of negative chips (those containing no bounding␣
↪boxes) to positive chips. This can be useful if the statistics of the background␣
↪is different in positive chips. For example, in car detection, the positive chips␣
↪will always contain roads, but no examples of rooftops since cars tend to not be␣
↪near rooftops.",
         "default": 1.0,
         "type": "number"
      },
      "ioa_thresh": {
         "title": "Ioa Thresh",
         "description": "When a box is partially outside of a training chip, it is␣
↪not clear if (a clipped version) of the box should be included in the chip. If␣
↪the IOA (intersection over area) of the box with the chip is greater than ioa_
↪thresh, it is included in the chip.",
         "default": 0.8,
         "type": "number"
      },
      "window_method": {
         "default": "chip",
         "allOf": [
            {
               "$ref": "#/definitions/ObjectDetectionWindowMethod"
            }
         ]
      },
      "label_buffer": {
         "title": "Label Buffer",
         "type": "integer"
      },
      "type_hint": {
         "title": "Type Hint",
         "default": "object_detection_chip_options",
```


(continues on next page)

(continued from previous page)


```
            "enum": [
               "object_detection_chip_options"
            ],
            "type": "string"
         }
      },
      "additionalProperties": false,
      "definitions": {
         "ObjectDetectionWindowMethod": {
            "title": "ObjectDetectionWindowMethod",
            "description": "Enum for window methods\n\n    Attributes:\n        chip:␣
↪the default method\n    ",
            "enum": [
               "chip",
               "label",
               "image",
               "sliding"
            ]
         }
      }
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *ioa_thresh (float)*
- *label_buffer (Optional[int])*
- *neg_ratio (float)*
- *type_hint (Literal['object_detection_chip_options'])*
- *window_method (rastervision.core.rv_pipeline.object_detection_config.ObjectDetectionWindowMethod)*

field ioa_thresh: float = 0.8

When a box is partially outside of a training chip, it is not clear if (a clipped version) of the box should be included in the chip. If the IOA (intersection over area) of the box with the chip is greater than ioa_thresh, it is included in the chip.

field label_buffer: Optional[int] = None

field neg_ratio: float = 1.0

The ratio of negative chips (those containing no bounding boxes) to positive chips. This can be useful if the statistics of the background is different in positive chips. For example, in car detection, the positive chips will always contain roads, but no examples of rooftops since cars tend to not be near rooftops.

field type_hint: Literal['object_detection_chip_options'] = 'object_detection_chip_options'

field window_method: *ObjectDetectionWindowMethod* = ObjectDetectionWindowMethod.chip

build()

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(**args*, ***kwargs*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

ObjectDetectionConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model ObjectDetectionConfig

Configure an *ObjectDetection* pipeline.


```
{
   "title": "ObjectDetectionConfig",
   "description": "Configure an :class:`.ObjectDetection` pipeline.",
   "type": "object",
   "properties": {
      "root_uri": {
```


(continues on next page)

(continued from previous page)

```
         "title": "Root Uri",
         "description": "The root URI for output generated by the pipeline",
         "type": "string"
      },
      "rv_config": {
         "title": "Rv Config",
         "description": "Used to store serialized RVConfig so pipeline can run in␣
↪remote environment with the local RVConfig. This should not be set explicitly by␣
↪users -- it is only used by the runner when running a remote pipeline.",
         "type": "object"
      },
      "plugin_versions": {
         "title": "Plugin Versions",
         "description": "Used to store a mapping of plugin module paths to the␣
↪latest version number. This should not be set explicitly by users -- it is set␣
↪automatically when serializing and saving the config to disk.",
         "type": "object",
         "additionalProperties": {
            "type": "integer"
         }
      },
      "type_hint": {
         "title": "Type Hint",
         "default": "object_detection",
         "enum": [
            "object_detection"
         ],
         "type": "string"
      },
      "dataset": {
         "title": "Dataset",
         "description": "Dataset containing train, validation, and optional test␣
↪scenes.",
         "allOf": [
            {
               "$ref": "#/definitions/DatasetConfig"
            }
         ]
      },
      "backend": {
         "title": "Backend",
         "description": "Backend to use for interfacing with ML library.",
         "allOf": [
            {
               "$ref": "#/definitions/BackendConfig"
            }
         ]
      },
      "evaluators": {
         "title": "Evaluators",
         "description": "Evaluators to run during analyzer command. If list is␣
↪empty the default evaluator is added.",
```

(continues on next page)

(continued from previous page)


```
         "default": [],
         "type": "array",
         "items": {
            "$ref": "#/definitions/EvaluatorConfig"
         }
      },
      "analyzers": {
         "title": "Analyzers",
         "description": "Analyzers to run during analyzer command. A StatsAnalyzer␣
↪will be added automatically if any scenes have a RasterTransformer.",
         "default": [],
         "type": "array",
         "items": {
            "$ref": "#/definitions/AnalyzerConfig"
         }
      },
      "train_chip_sz": {
         "title": "Train Chip Sz",
         "description": "Size of training chips in pixels.",
         "default": 300,
         "type": "integer"
      },
      "predict_chip_sz": {
         "title": "Predict Chip Sz",
         "description": "Size of predictions chips in pixels.",
         "default": 300,
         "type": "integer"
      },
      "predict_batch_sz": {
         "title": "Predict Batch Sz",
         "description": "Batch size to use during prediction.",
         "default": 8,
         "type": "integer"
      },
      "chip_nodata_threshold": {
         "title": "Chip Nodata Threshold",
         "description": "Discard chips where the proportion of NODATA values is␣
↪greater than or equal to this value. Might result in false positives if there are␣
↪many legitimate black pixels in the chip. Use with caution.",
         "default": 1,
         "minimum": 0,
         "maximum": 1,
         "type": "number"
      },
      "analyze_uri": {
         "title": "Analyze Uri",
         "description": "URI for output of analyze. If None, will be auto-generated.
↪",
         "type": "string"
      },
      "chip_uri": {
         "title": "Chip Uri",
```


(continues on next page)

(continued from previous page)


```
         "description": "URI for output of chip. If None, will be auto-generated.",
         "type": "string"
      },
      "train_uri": {
         "title": "Train Uri",
         "description": "URI for output of train. If None, will be auto-generated.",
         "type": "string"
      },
      "predict_uri": {
         "title": "Predict Uri",
         "description": "URI for output of predict. If None, will be auto-generated.
↪",
         "type": "string"
      },
      "eval_uri": {
         "title": "Eval Uri",
         "description": "URI for output of eval. If None, will be auto-generated.",
         "type": "string"
      },
      "bundle_uri": {
         "title": "Bundle Uri",
         "description": "URI for output of bundle. If None, will be auto-generated.
↪",
         "type": "string"
      },
      "source_bundle_uri": {
         "title": "Source Bundle Uri",
         "description": "If provided, the model will be loaded from this bundle for␣
↪the train stage. Useful for fine-tuning.",
         "type": "string"
      },
      "chip_options": {
         "title": "Chip Options",
         "default": {
            "neg_ratio": 1.0,
            "ioa_thresh": 0.8,
            "window_method": "ObjectDetectionWindowMethod.chip",
            "label_buffer": null,
            "type_hint": "object_detection_chip_options"
         },
         "allOf": [
            {
               "$ref": "#/definitions/ObjectDetectionChipOptions"
            }
         ]
      },
      "predict_options": {
         "title": "Predict Options",
         "default": {
            "type_hint": "object_detection_predict_options",
            "merge_thresh": 0.5,
            "score_thresh": 0.5
```


(continues on next page)

(continued from previous page)


```
        },
        "allOf": [
           {
              "$ref": "#/definitions/ObjectDetectionPredictOptions"
           }
        ]
     }
  },
  "required": [
     "dataset",
     "backend"
  ],
  "additionalProperties": false,
  "definitions": {
     "ClassConfig": {
        "title": "ClassConfig",
        "description": "Configure class information for a machine learning task.",
        "type": "object",
        "properties": {
           "names": {
              "title": "Names",
              "description": "Names of classes. The i-th class in this list will␣
↪have class ID = i.",
              "type": "array",
              "items": {
                 "type": "string"
              }
           },
           "colors": {
              "title": "Colors",
              "description": "Colors used to visualize classes. Can be color␣
↪strings accepted by matplotlib or RGB tuples. If None, a random color will be␣
↪auto-generated for each class.",
              "type": "array",
              "items": {
                 "anyOf": [
                    {
                       "type": "string"
                    },
                    {
                       "type": "array",
                       "items": {}
                    }
                 ]
              }
           },
           "null_class": {
              "title": "Null Class",
              "description": "Optional name of class in `names` to use as the null␣
↪class. This is used in semantic segmentation to represent the label for imagery␣
↪pixels that are NODATA or that are missing a label. If None and the class names␣
↪include \"null\", it will automatically be used as the null class. If None, and␣
```


(continues on next page)

(continued from previous page)


```
↪this Config is part of a SemanticSegmentationConfig, a null class will be added␣
↪automatically.",
                "type": "string"
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "class_config",
                "enum": [
                    "class_config"
                ],
                "type": "string"
            }
        },
        "required": [
            "names"
        ],
        "additionalProperties": false
    },
    "RasterTransformerConfig": {
        "title": "RasterTransformerConfig",
        "description": "Configure a :class:`.RasterTransformer`.",
        "type": "object",
        "properties": {
            "type_hint": {
                "title": "Type Hint",
                "default": "raster_transformer",
                "enum": [
                    "raster_transformer"
                ],
                "type": "string"
            }
        },
        "additionalProperties": false
    },
    "RasterSourceConfig": {
        "title": "RasterSourceConfig",
        "description": "Configure a :class:`.RasterSource`.",
        "type": "object",
        "properties": {
            "channel_order": {
                "title": "Channel Order",
                "description": "The sequence of channel indices to use when reading␣
↪imagery.",
                "type": "array",
                "items": {
                    "type": "integer"
                }
            },
            "transformers": {
                "title": "Transformers",
                "default": [],
                "type": "array",
```


(continues on next page)

(continued from previous page)


```
                "items": {
                    "$ref": "#/definitions/RasterTransformerConfig"
                }
            },
            "extent": {
                "title": "Extent",
                "description": "Use-specified extent in pixel coords in the form␣
↪(ymin, xmin, ymax, xmax). Useful for cropping the raster source so that only part␣
↪of the raster is read from.",
                "type": "array",
                "minItems": 4,
                "maxItems": 4,
                "items": [
                    {
                        "type": "integer"
                    },
                    {
                        "type": "integer"
                    },
                    {
                        "type": "integer"
                    },
                    {
                        "type": "integer"
                    }
                ]
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "raster_source",
                "enum": [
                    "raster_source"
                ],
                "type": "string"
            }
        },
        "additionalProperties": false
    },
    "LabelSourceConfig": {
        "title": "LabelSourceConfig",
        "description": "Configure a :class:`.LabelSource`.",
        "type": "object",
        "properties": {
            "type_hint": {
                "title": "Type Hint",
                "default": "label_source",
                "enum": [
                    "label_source"
                ],
                "type": "string"
            }
        },
```


(continues on next page)

(continued from previous page)


```
            "additionalProperties": false
         },
         "LabelStoreConfig": {
            "title": "LabelStoreConfig",
            "description": "Configure a :class:`.LabelStore`.",
            "type": "object",
            "properties": {
               "type_hint": {
                  "title": "Type Hint",
                  "default": "label_store",
                  "enum": [
                     "label_store"
                  ],
                  "type": "string"
               }
            },
            "additionalProperties": false
         },
         "SceneConfig": {
            "title": "SceneConfig",
            "description": "Configure a :class:`.Scene` comprising raster data &␣
↪labels for an AOI.\n    ",
            "type": "object",
            "properties": {
               "id": {
                  "title": "Id",
                  "type": "string"
               },
               "raster_source": {
                  "$ref": "#/definitions/RasterSourceConfig"
               },
               "label_source": {
                  "$ref": "#/definitions/LabelSourceConfig"
               },
               "label_store": {
                  "$ref": "#/definitions/LabelStoreConfig"
               },
               "aoi_uris": {
                  "title": "Aoi Uris",
                  "description": "List of URIs of GeoJSON files that define the AOIs␣
↪for the scene. Each polygon defines an AOI which is a piece of the scene that is␣
↪assumed to be fully labeled and usable for training or validation. The AOIs are␣
↪assumed to be in EPSG:4326 coordinates.",
                  "type": "array",
                  "items": {
                     "type": "string"
                  }
               },
               "type_hint": {
                  "title": "Type Hint",
                  "default": "scene",
                  "enum": [
```


(continues on next page)

(continued from previous page)


```
                    "scene"
                ],
                "type": "string"
            }
        },
        "required": [
            "id",
            "raster_source"
        ],
        "additionalProperties": false
    },
    "DatasetConfig": {
        "title": "DatasetConfig",
        "description": "Configure train, validation, and test splits for a dataset.
↪",
        "type": "object",
        "properties": {
            "class_config": {
                "$ref": "#/definitions/ClassConfig"
            },
            "train_scenes": {
                "title": "Train Scenes",
                "type": "array",
                "items": {
                    "$ref": "#/definitions/SceneConfig"
                }
            },
            "validation_scenes": {
                "title": "Validation Scenes",
                "type": "array",
                "items": {
                    "$ref": "#/definitions/SceneConfig"
                }
            },
            "test_scenes": {
                "title": "Test Scenes",
                "default": [],
                "type": "array",
                "items": {
                    "$ref": "#/definitions/SceneConfig"
                }
            },
            "scene_groups": {
                "title": "Scene Groups",
                "description": "Groupings of scenes. Should be a dict of the form: {
↪<group-name>: Set(scene_id_1, scene_id_2, ...)}. Three groups are added by␣
↪default: \"train_scenes\", \"validation_scenes\", and \"test_scenes\"",
                "default": {},
                "type": "object",
                "additionalProperties": {
                    "type": "array",
                    "items": {
```


(continues on next page)

(continued from previous page)


```
                    "type": "string"
                },
                "uniqueItems": true
            }
        },
        "type_hint": {
            "title": "Type Hint",
            "default": "dataset",
            "enum": [
                "dataset"
            ],
            "type": "string"
        }
    },
    "required": [
        "class_config",
        "train_scenes",
        "validation_scenes"
    ],
    "additionalProperties": false
},
"BackendConfig": {
    "title": "BackendConfig",
    "description": "Configure a :class:`.Backend`.",
    "type": "object",
    "properties": {
        "type_hint": {
            "title": "Type Hint",
            "default": "backend",
            "enum": [
                "backend"
            ],
            "type": "string"
        }
    },
    "additionalProperties": false
},
"EvaluatorConfig": {
    "title": "EvaluatorConfig",
    "description": "Configure an :class:`.Evaluator`.",
    "type": "object",
    "properties": {
        "output_uri": {
            "title": "Output Uri",
            "description": "URI of directory where evaluator output will be␣
↪saved. Evaluations for each scene-group will be save in a JSON file at <output_
↪uri>/<scene-group-name>/eval.json. If None, and this Config is part of an␣
↪RVPipeline, this field will be auto-generated.",
            "type": "string"
        },
        "type_hint": {
            "title": "Type Hint",
```


(continues on next page)

(continued from previous page)


```
                "default": "evaluator",
                "enum": [
                   "evaluator"
                ],
                "type": "string"
             }
          },
          "additionalProperties": false
       },
       "AnalyzerConfig": {
          "title": "AnalyzerConfig",
          "description": "Configure an :class:`.Analyzer`.",
          "type": "object",
          "properties": {
             "type_hint": {
                "title": "Type Hint",
                "default": "analyzer",
                "enum": [
                   "analyzer"
                ],
                "type": "string"
             }
          },
          "additionalProperties": false
       },
       "ObjectDetectionWindowMethod": {
          "title": "ObjectDetectionWindowMethod",
          "description": "Enum for window methods\n\n    Attributes:\n        chip:␣
↪the default method\n    ",
          "enum": [
             "chip",
             "label",
             "image",
             "sliding"
          ]
       },
       "ObjectDetectionChipOptions": {
          "title": "ObjectDetectionChipOptions",
          "description": "Base class that can be extended to provide custom␣
↪configurations.\n\nThis adds some extra methods to Pydantic BaseModel.\nSee https:
↪//pydantic-docs.helpmanual.io/\n\nThe general idea is that configuration schemas␣
↪can be defined by\nsubclassing this and adding class attributes with types and\
↪ndefault values for each field. Configs can be defined hierarchically,\nie. a␣
↪Config can have fields which are of type Config.\nValidation, serialization,␣
↪deserialization, and IDE support is\nprovided automatically based on this schema.
↪",
          "type": "object",
          "properties": {
             "neg_ratio": {
                "title": "Neg Ratio",
                "description": "The ratio of negative chips (those containing no␣
↪bounding boxes) to positive chips. This can be useful if the statistics of the␣
```


(continues on next page)

(continued from previous page)


```
↪background is different in positive chips. For example, in car detection, the␣
↪positive chips will always contain roads, but no examples of rooftops since cars␣
↪tend to not be near rooftops.",
                "default": 1.0,
                "type": "number"
            },
            "ioa_thresh": {
                "title": "Ioa Thresh",
                "description": "When a box is partially outside of a training chip,␣
↪it is not clear if (a clipped version) of the box should be included in the chip.␣
↪If the IOA (intersection over area) of the box with the chip is greater than ioa_
↪thresh, it is included in the chip.",
                "default": 0.8,
                "type": "number"
            },
            "window_method": {
                "default": "chip",
                "allOf": [
                    {
                        "$ref": "#/definitions/ObjectDetectionWindowMethod"
                    }
                ]
            },
            "label_buffer": {
                "title": "Label Buffer",
                "type": "integer"
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "object_detection_chip_options",
                "enum": [
                    "object_detection_chip_options"
                ],
                "type": "string"
            }
        },
        "additionalProperties": false
    },
    "ObjectDetectionPredictOptions": {
        "title": "ObjectDetectionPredictOptions",
        "description": "Base class that can be extended to provide custom␣
↪configurations.\n\nThis adds some extra methods to Pydantic BaseModel.\nSee https:
↪//pydantic-docs.helpmanual.io/\n\nThe general idea is that configuration schemas␣
↪can be defined by\nsubclassing this and adding class attributes with types and\
↪ndefault values for each field. Configs can be defined hierarchically,\nie. a␣
↪Config can have fields which are of type Config.\nValidation, serialization,␣
↪deserialization, and IDE support is\nprovided automatically based on this schema.
↪",
        "type": "object",
        "properties": {
            "type_hint": {
                "title": "Type Hint",
```


(continues on next page)

(continued from previous page)


```
                "default": "object_detection_predict_options",
                "enum": [
                    "object_detection_predict_options"
                ],
                "type": "string"
            },
            "merge_thresh": {
                "title": "Merge Thresh",
                "description": "If predicted boxes have an IOA (intersection over␣
↪area) greater than merge_thresh, then they are merged into a single box during␣
↪postprocessing. This is needed since the sliding window approach results in some␣
↪false duplicates.",
                "default": 0.5,
                "type": "number"
            },
            "score_thresh": {
                "title": "Score Thresh",
                "description": "Predicted boxes are only output if their score is␣
↪above score_thresh.",
                "default": 0.5,
                "type": "number"
            }
        },
        "additionalProperties": false
    }
  }
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *analyze_uri (Optional[str])*
- *analyzers (List[rastervision.core.analyzer.analyzer_config.AnalyzerConfig])*
- *backend (rastervision.core.backend.backend_config.BackendConfig)*
- *bundle_uri (Optional[str])*
- *chip_nodata_threshold (rastervision.core.utils.misc.ConstrainedFloatValue)*
- *chip_options (rastervision.core.rv_pipeline.object_detection_config.ObjectDetectionChipOptions)*
- *chip_uri (Optional[str])*
- *dataset (rastervision.core.data.dataset_config.DatasetConfig)*
- *eval_uri (Optional[str])*
- *evaluators (List[rastervision.core.evaluation.evaluator_config.EvaluatorConfig])*

- *plugin_versions (Optional[Dict[str, int]])*
- *predict_batch_sz (int)*
- *predict_chip_sz (int)*
- *predict_options (rastervision.core.rv_pipeline.object_detection_config.ObjectDetectionPredictOptions)*
- *predict_uri (Optional[str])*
- *root_uri (str)*
- *rv_config (dict)*
- *source_bundle_uri (Optional[str])*
- *train_chip_sz (int)*
- *train_uri (Optional[str])*
- *type_hint (Literal['object_detection'])*

field analyze_uri: Optional[str] = None

URI for output of analyze. If None, will be auto-generated.

field analyzers: List[*AnalyzerConfig*] = []

Analyzers to run during analyzer command. A StatsAnalyzer will be added automatically if any scenes have a RasterTransformer.

field backend: *BackendConfig* [Required]

Backend to use for interfacing with ML library.

field bundle_uri: Optional[str] = None

URI for output of bundle. If None, will be auto-generated.

field chip_nodata_threshold: Proportion = 1

Discard chips where the proportion of NODATA values is greater than or equal to this value. Might result in false positives if there are many legitimate black pixels in the chip. Use with caution.

Constraints

- **minimum** = 0
- **maximum** = 1

field chip_options: *ObjectDetectionChipOptions* = ObjectDetectionChipOptions(neg_ratio=1.0, ioa_thresh=0.8, window_method=<ObjectDetectionWindowMethod.chip: 'chip'>, label_buffer=None)

field chip_uri: Optional[str] = None

URI for output of chip. If None, will be auto-generated.

field dataset: *DatasetConfig* [Required]

Dataset containing train, validation, and optional test scenes.

field eval_uri: Optional[str] = None

URI for output of eval. If None, will be auto-generated.

field evaluators: List[*EvaluatorConfig*] = []

Evaluators to run during analyzer command. If list is empty the default evaluator is added.

field plugin_versions: Optional[Dict[str, int]] = None

Used to store a mapping of plugin module paths to the latest version number. This should not be set explicitly by users – it is set automatically when serializing and saving the config to disk.

field predict_batch_sz: int = 8

Batch size to use during prediction.

field predict_chip_sz: int = 300

Size of predictions chips in pixels.

field predict_options: *ObjectDetectionPredictOptions* = ObjectDetectionPredictOptions(merge_thresh=0.5, score_thresh=0.5)

field predict_uri: Optional[str] = None

URI for output of predict. If None, will be auto-generated.

field root_uri: str = None

The root URI for output generated by the pipeline

field rv_config: dict = None

Used to store serialized RVConfig so pipeline can run in remote environment with the local RVConfig. This should not be set explicitly by users – it is only used by the runner when running a remote pipeline.

field source_bundle_uri: Optional[str] = None

If provided, the model will be loaded from this bundle for the train stage. Useful for fine-tuning.

field train_chip_sz: int = 300

Size of training chips in pixels.

field train_uri: Optional[str] = None

URI for output of train. If None, will be auto-generated.

field type_hint: Literal['object_detection'] = 'object_detection'

build(*tmp_dir*)

Return a pipeline based on this configuration.

Subclasses should override this to return an instance of the corresponding subclass of Pipeline.

Parameters
tmp_dir – root of any temporary directory to pass to pipeline

get_config_uri() → str

Get URI of serialized version of this PipelineConfig.

Return type
str

get_default_evaluator()

Returns a default EvaluatorConfig to use if one isn't set.

get_default_label_store(*scene*)

Returns a default LabelStoreConfig to fill in any missing ones.

get_model_bundle_uri()

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update()

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

ObjectDetectionPredictOptions

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model ObjectDetectionPredictOptions


```
{
   "title": "ObjectDetectionPredictOptions",
   "description": "Base class that can be extended to provide custom configurations.
↪\n\nThis adds some extra methods to Pydantic BaseModel.\nSee https://pydantic-
↪docs.helpmanual.io/\n\nThe general idea is that configuration schemas can be␣
↪defined by\nsubclassing this and adding class attributes with types and\ndefault␣
↪values for each field. Configs can be defined hierarchically,\nie. a Config can␣
↪have fields which are of type Config.\nValidation, serialization, deserialization,
↪ and IDE support is\nprovided automatically based on this schema.",
   "type": "object",
   "properties": {
      "type_hint": {
         "title": "Type Hint",
         "default": "object_detection_predict_options",
         "enum": [
            "object_detection_predict_options"
         ],
         "type": "string"
```


(continues on next page)

(continued from previous page)

```
      },
      "merge_thresh": {
         "title": "Merge Thresh",
         "description": "If predicted boxes have an IOA (intersection over area)␣
↪greater than merge_thresh, then they are merged into a single box during␣
↪postprocessing. This is needed since the sliding window approach results in some␣
↪false duplicates.",
         "default": 0.5,
         "type": "number"
      },
      "score_thresh": {
         "title": "Score Thresh",
         "description": "Predicted boxes are only output if their score is above␣
↪score_thresh.",
         "default": 0.5,
         "type": "number"
      }
   },
   "additionalProperties": false
}
```

Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *merge_thresh (float)*
- *score_thresh (float)*
- *type_hint (Literal['object_detection_predict_options'])*

field merge_thresh: float = 0.5

If predicted boxes have an IOA (intersection over area) greater than merge_thresh, then they are merged into a single box during postprocessing. This is needed since the sliding window approach results in some false duplicates.

field score_thresh: float = 0.5

Predicted boxes are only output if their score is above score_thresh.

field type_hint: Literal['object_detection_predict_options'] = 'object_detection_predict_options'

build()

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(**args*, ***kwargs*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

rv_pipeline

Classes

RVPipeline	Base class of all Raster Vision Pipelines.

RVPipeline

class RVPipeline

Bases: *Pipeline*

Base class of all Raster Vision Pipelines.

This can be subclassed to implement Pipelines for different computer vision tasks over geospatial imagery. The commands and what they produce include:

- analyze: metrics on the imagery and labels
- chip: small training and validation images taken from larger scenes
- train: model trained on chips
- predict: predictions over entire validation and test scenes
- eval: evaluation metrics for predictions generated by model
- bundle: bundle containing model and any other files needed to make

predictions using the Predictor.

Attributes

commands	Built-in mutable sequence.
gpu_commands	Built-in mutable sequence.
split_commands	Built-in mutable sequence.

__init__(*config:* RVPipelineConfig, *tmp_dir:* *str*)

Constructor

Parameters

- **config** (RVPipelineConfig) – the configuration of this pipeline
- **tmp_dir** (*str*) – the root any temporary directories created by running this pipeline

Methods

__init__(config, tmp_dir)	Constructor
analyze()	Run each analyzer over training scenes.
bundle()	Save a model bundle with whatever is needed to make predictions.
chip([split_ind, num_splits])	Save training and validation chips.
eval()	Evaluate predictions against ground truth.
get_train_labels(window, scene)	Return the training labels in a window for a scene.
get_train_windows(scene)	Return the training windows for a Scene.
post_process_batch(windows, chips, labels)	Post-process a batch of predictions.
post_process_predictions(labels, scene)	Post-process all labels at end of prediction.
post_process_sample(sample)	Post-process sample in pipeline-specific way.
predict([split_ind, num_splits])	Make predictions over each validation and test scene.
predict_scene(scene, backend)	
test_cpu([split_ind, num_splits])	A command to test the ability to run split jobs on CPU.
test_gpu()	A command to test the ability to run on GPU.
train()	Train a model and save it.

__init__(*config:* RVPipelineConfig, *tmp_dir:* *str*)

Constructor

Parameters

- **config** (RVPipelineConfig) – the configuration of this pipeline
- **tmp_dir** (*str*) – the root any temporary directories created by running this pipeline

analyze()

Run each analyzer over training scenes.

bundle()

Save a model bundle with whatever is needed to make predictions.

The model bundle is a zip file and it is used by the Predictor and predict CLI subcommand.

chip(*split_ind: int = 0*, *num_splits: int = 1*)

Save training and validation chips.

Parameters

- **split_ind** (*int*) –
- **num_splits** (*int*) –

eval()

Evaluate predictions against ground truth.

get_train_labels(*window:* Box, *scene:* Scene) → *Labels*

Return the training labels in a window for a scene.

Returns
Labels that lie within window

Parameters

- **window** (Box) –
- **scene** (Scene) –

Return type
Labels

get_train_windows(*scene:* Scene) → List[*Box*]

Return the training windows for a Scene.

Each training window represents the spatial extent of a training chip to generate.

Parameters
scene (Scene) – Scene to generate windows for

Return type
List[Box]

post_process_batch(*windows: List[*Box*]*, *chips: ndarray*, *labels:* Labels) → *Labels*

Post-process a batch of predictions.

Parameters

- **windows** (*List[*Box*]*) –
- **chips** (*ndarray*) –
- **labels** (Labels) –

Return type
Labels

post_process_predictions(*labels:* Labels, *scene:* Scene) → *Labels*

Post-process all labels at end of prediction.

Parameters

- **labels** (Labels) –
- **scene** (Scene) –

Return type
Labels

post_process_sample(*sample: DataSample*) → DataSample

Post-process sample in pipeline-specific way.

This should be called before writing a sample during chipping.

Parameters
sample (*DataSample*) –

Return type
DataSample

predict(*split_ind=0*, *num_splits=1*)

Make predictions over each validation and test scene.

This uses a sliding window.

predict_scene(*scene:* Scene, *backend:* Backend) → *Labels*

Parameters

- **scene** (Scene) –
- **backend** (Backend) –

Return type
Labels

test_cpu(*split_ind: int = 0*, *num_splits: int = 1*)

A command to test the ability to run split jobs on CPU.

Parameters

- **split_ind** (*int*) –
- **num_splits** (*int*) –

test_gpu()

A command to test the ability to run on GPU.

train()

Train a model and save it.

property commands

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

property gpu_commands

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

property split_commands

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

rv_pipeline_config

Configs

PredictOptions	
RVPipelineConfig	Configure an *RVPipeline*.

PredictOptions

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model PredictOptions


```
{
   "title": "PredictOptions",
   "description": "Base class that can be extended to provide custom configurations.
↪\n\nThis adds some extra methods to Pydantic BaseModel.\nSee https://pydantic-
↪docs.helpmanual.io/\n\nThe general idea is that configuration schemas can be␣
↪defined by\nsubclassing this and adding class attributes with types and\ndefault␣
↪values for each field. Configs can be defined hierarchically,\nie. a Config can␣
↪have fields which are of type Config.\nValidation, serialization, deserialization,
↪ and IDE support is\nprovided automatically based on this schema.",
   "type": "object",
   "properties": {
      "type_hint": {
         "title": "Type Hint",
         "default": "predict_options",
         "enum": [
            "predict_options"
         ],
         "type": "string"
      }
   },
   "additionalProperties": false
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *type_hint (Literal['predict_options'])*

field type_hint: Literal['predict_options'] = 'predict_options'

build()

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(**args*, ***kwargs*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

RVPipelineConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model RVPipelineConfig

Configure an *RVPipeline*.


```
{
   "title": "RVPipelineConfig",
   "description": "Configure an :class:`.RVPipeline`.",
   "type": "object",
   "properties": {
      "root_uri": {
         "title": "Root Uri",
         "description": "The root URI for output generated by the pipeline",
```


(continues on next page)

(continued from previous page)


```
            "type": "string"
         },
         "rv_config": {
            "title": "Rv Config",
            "description": "Used to store serialized RVConfig so pipeline can run in␣
↪remote environment with the local RVConfig. This should not be set explicitly by␣
↪users -- it is only used by the runner when running a remote pipeline.",
            "type": "object"
         },
         "plugin_versions": {
            "title": "Plugin Versions",
            "description": "Used to store a mapping of plugin module paths to the␣
↪latest version number. This should not be set explicitly by users -- it is set␣
↪automatically when serializing and saving the config to disk.",
            "type": "object",
            "additionalProperties": {
               "type": "integer"
            }
         },
         "type_hint": {
            "title": "Type Hint",
            "default": "rv_pipeline",
            "enum": [
               "rv_pipeline"
            ],
            "type": "string"
         },
         "dataset": {
            "title": "Dataset",
            "description": "Dataset containing train, validation, and optional test␣
↪scenes.",
            "allOf": [
               {
                  "$ref": "#/definitions/DatasetConfig"
               }
            ]
         },
         "backend": {
            "title": "Backend",
            "description": "Backend to use for interfacing with ML library.",
            "allOf": [
               {
                  "$ref": "#/definitions/BackendConfig"
               }
            ]
         },
         "evaluators": {
            "title": "Evaluators",
            "description": "Evaluators to run during analyzer command. If list is␣
↪empty the default evaluator is added.",
            "default": [],
            "type": "array",
```


(continues on next page)

(continued from previous page)


```
            "items": {
               "$ref": "#/definitions/EvaluatorConfig"
            }
         },
         "analyzers": {
            "title": "Analyzers",
            "description": "Analyzers to run during analyzer command. A StatsAnalyzer␣
↪will be added automatically if any scenes have a RasterTransformer.",
            "default": [],
            "type": "array",
            "items": {
               "$ref": "#/definitions/AnalyzerConfig"
            }
         },
         "train_chip_sz": {
            "title": "Train Chip Sz",
            "description": "Size of training chips in pixels.",
            "default": 300,
            "type": "integer"
         },
         "predict_chip_sz": {
            "title": "Predict Chip Sz",
            "description": "Size of predictions chips in pixels.",
            "default": 300,
            "type": "integer"
         },
         "predict_batch_sz": {
            "title": "Predict Batch Sz",
            "description": "Batch size to use during prediction.",
            "default": 8,
            "type": "integer"
         },
         "chip_nodata_threshold": {
            "title": "Chip Nodata Threshold",
            "description": "Discard chips where the proportion of NODATA values is␣
↪greater than or equal to this value. Might result in false positives if there are␣
↪many legitimate black pixels in the chip. Use with caution.",
            "default": 1,
            "minimum": 0,
            "maximum": 1,
            "type": "number"
         },
         "analyze_uri": {
            "title": "Analyze Uri",
            "description": "URI for output of analyze. If None, will be auto-generated.
↪",
            "type": "string"
         },
         "chip_uri": {
            "title": "Chip Uri",
            "description": "URI for output of chip. If None, will be auto-generated.",
            "type": "string"
```


(continues on next page)

(continued from previous page)


```
      },
      "train_uri": {
         "title": "Train Uri",
         "description": "URI for output of train. If None, will be auto-generated.",
         "type": "string"
      },
      "predict_uri": {
         "title": "Predict Uri",
         "description": "URI for output of predict. If None, will be auto-generated.
↪",
         "type": "string"
      },
      "eval_uri": {
         "title": "Eval Uri",
         "description": "URI for output of eval. If None, will be auto-generated.",
         "type": "string"
      },
      "bundle_uri": {
         "title": "Bundle Uri",
         "description": "URI for output of bundle. If None, will be auto-generated.
↪",
         "type": "string"
      },
      "source_bundle_uri": {
         "title": "Source Bundle Uri",
         "description": "If provided, the model will be loaded from this bundle for␣
↪the train stage. Useful for fine-tuning.",
         "type": "string"
      }
   },
   "required": [
      "dataset",
      "backend"
   ],
   "additionalProperties": false,
   "definitions": {
      "ClassConfig": {
         "title": "ClassConfig",
         "description": "Configure class information for a machine learning task.",
         "type": "object",
         "properties": {
            "names": {
               "title": "Names",
               "description": "Names of classes. The i-th class in this list will␣
↪have class ID = i.",
               "type": "array",
               "items": {
                  "type": "string"
               }
            },
            "colors": {
               "title": "Colors",
```


(continues on next page)

(continued from previous page)


```
                "description": "Colors used to visualize classes. Can be color␣
↪strings accepted by matplotlib or RGB tuples. If None, a random color will be␣
↪auto-generated for each class.",
                "type": "array",
                "items": {
                   "anyOf": [
                      {
                         "type": "string"
                      },
                      {
                         "type": "array",
                         "items": {}
                      }
                   ]
                }
             },
             "null_class": {
                "title": "Null Class",
                "description": "Optional name of class in `names` to use as the null␣
↪class. This is used in semantic segmentation to represent the label for imagery␣
↪pixels that are NODATA or that are missing a label. If None and the class names␣
↪include \"null\", it will automatically be used as the null class. If None, and␣
↪this Config is part of a SemanticSegmentationConfig, a null class will be added␣
↪automatically.",
                "type": "string"
             },
             "type_hint": {
                "title": "Type Hint",
                "default": "class_config",
                "enum": [
                   "class_config"
                ],
                "type": "string"
             }
          },
          "required": [
             "names"
          ],
          "additionalProperties": false
       },
       "RasterTransformerConfig": {
          "title": "RasterTransformerConfig",
          "description": "Configure a :class:`.RasterTransformer`.",
          "type": "object",
          "properties": {
             "type_hint": {
                "title": "Type Hint",
                "default": "raster_transformer",
                "enum": [
                   "raster_transformer"
                ],
                "type": "string"
```


(continues on next page)

(continued from previous page)


```
            }
         },
         "additionalProperties": false
      },
      "RasterSourceConfig": {
         "title": "RasterSourceConfig",
         "description": "Configure a :class:`.RasterSource`.",
         "type": "object",
         "properties": {
            "channel_order": {
               "title": "Channel Order",
               "description": "The sequence of channel indices to use when reading␣
↪imagery.",
               "type": "array",
               "items": {
                  "type": "integer"
               }
            },
            "transformers": {
               "title": "Transformers",
               "default": [],
               "type": "array",
               "items": {
                  "$ref": "#/definitions/RasterTransformerConfig"
               }
            },
            "extent": {
               "title": "Extent",
               "description": "Use-specified extent in pixel coords in the form␣
↪(ymin, xmin, ymax, xmax). Useful for cropping the raster source so that only part␣
↪of the raster is read from.",
               "type": "array",
               "minItems": 4,
               "maxItems": 4,
               "items": [
                  {
                     "type": "integer"
                  },
                  {
                     "type": "integer"
                  },
                  {
                     "type": "integer"
                  },
                  {
                     "type": "integer"
                  }
               ]
            },
            "type_hint": {
               "title": "Type Hint",
               "default": "raster_source",
```


(continues on next page)

(continued from previous page)

```
                "enum": [
                    "raster_source"
                ],
                "type": "string"
            }
        },
        "additionalProperties": false
    },
    "LabelSourceConfig": {
        "title": "LabelSourceConfig",
        "description": "Configure a :class:`.LabelSource`.",
        "type": "object",
        "properties": {
            "type_hint": {
                "title": "Type Hint",
                "default": "label_source",
                "enum": [
                    "label_source"
                ],
                "type": "string"
            }
        },
        "additionalProperties": false
    },
    "LabelStoreConfig": {
        "title": "LabelStoreConfig",
        "description": "Configure a :class:`.LabelStore`.",
        "type": "object",
        "properties": {
            "type_hint": {
                "title": "Type Hint",
                "default": "label_store",
                "enum": [
                    "label_store"
                ],
                "type": "string"
            }
        },
        "additionalProperties": false
    },
    "SceneConfig": {
        "title": "SceneConfig",
        "description": "Configure a :class:`.Scene` comprising raster data &␣
↪labels for an AOI.\n    ",
        "type": "object",
        "properties": {
            "id": {
                "title": "Id",
                "type": "string"
            },
            "raster_source": {
                "$ref": "#/definitions/RasterSourceConfig"
```

(continues on next page)

(continued from previous page)


```
            },
            "label_source": {
               "$ref": "#/definitions/LabelSourceConfig"
            },
            "label_store": {
               "$ref": "#/definitions/LabelStoreConfig"
            },
            "aoi_uris": {
               "title": "Aoi Uris",
               "description": "List of URIs of GeoJSON files that define the AOIs␣
↪for the scene. Each polygon defines an AOI which is a piece of the scene that is␣
↪assumed to be fully labeled and usable for training or validation. The AOIs are␣
↪assumed to be in EPSG:4326 coordinates.",
               "type": "array",
               "items": {
                  "type": "string"
               }
            },
            "type_hint": {
               "title": "Type Hint",
               "default": "scene",
               "enum": [
                  "scene"
               ],
               "type": "string"
            }
         },
         "required": [
            "id",
            "raster_source"
         ],
         "additionalProperties": false
      },
      "DatasetConfig": {
         "title": "DatasetConfig",
         "description": "Configure train, validation, and test splits for a dataset.
↪",
         "type": "object",
         "properties": {
            "class_config": {
               "$ref": "#/definitions/ClassConfig"
            },
            "train_scenes": {
               "title": "Train Scenes",
               "type": "array",
               "items": {
                  "$ref": "#/definitions/SceneConfig"
               }
            },
            "validation_scenes": {
               "title": "Validation Scenes",
               "type": "array",
```


(continues on next page)

(continued from previous page)


```
                "items": {
                    "$ref": "#/definitions/SceneConfig"
                }
            },
            "test_scenes": {
                "title": "Test Scenes",
                "default": [],
                "type": "array",
                "items": {
                    "$ref": "#/definitions/SceneConfig"
                }
            },
            "scene_groups": {
                "title": "Scene Groups",
                "description": "Groupings of scenes. Should be a dict of the form: {
↪<group-name>: Set(scene_id_1, scene_id_2, ...)}. Three groups are added by␣
↪default: \"train_scenes\", \"validation_scenes\", and \"test_scenes\"",
                "default": {},
                "type": "object",
                "additionalProperties": {
                    "type": "array",
                    "items": {
                        "type": "string"
                    },
                    "uniqueItems": true
                }
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "dataset",
                "enum": [
                    "dataset"
                ],
                "type": "string"
            }
        },
        "required": [
            "class_config",
            "train_scenes",
            "validation_scenes"
        ],
        "additionalProperties": false
    },
    "BackendConfig": {
        "title": "BackendConfig",
        "description": "Configure a :class:`.Backend`.",
        "type": "object",
        "properties": {
            "type_hint": {
                "title": "Type Hint",
                "default": "backend",
                "enum": [
```


(continues on next page)

(continued from previous page)


```
                    "backend"
                ],
                "type": "string"
            }
        },
        "additionalProperties": false
    },
    "EvaluatorConfig": {
        "title": "EvaluatorConfig",
        "description": "Configure an :class:`.Evaluator`.",
        "type": "object",
        "properties": {
            "output_uri": {
                "title": "Output Uri",
                "description": "URI of directory where evaluator output will be␣
↪saved. Evaluations for each scene-group will be save in a JSON file at <output_
↪uri>/<scene-group-name>/eval.json. If None, and this Config is part of an␣
↪RVPipeline, this field will be auto-generated.",
                "type": "string"
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "evaluator",
                "enum": [
                    "evaluator"
                ],
                "type": "string"
            }
        },
        "additionalProperties": false
    },
    "AnalyzerConfig": {
        "title": "AnalyzerConfig",
        "description": "Configure an :class:`.Analyzer`.",
        "type": "object",
        "properties": {
            "type_hint": {
                "title": "Type Hint",
                "default": "analyzer",
                "enum": [
                    "analyzer"
                ],
                "type": "string"
            }
        },
        "additionalProperties": false
    }
  }
}
```


Config

- **extra**: *str = forbid*

- **validate_assignment**: *bool* = *True*

Fields

- *analyze_uri (Optional[str])*
- *analyzers (List[rastervision.core.analyzer.analyzer_config.AnalyzerConfig])*
- *backend (rastervision.core.backend.backend_config.BackendConfig)*
- *bundle_uri (Optional[str])*
- *chip_nodata_threshold (rastervision.core.utils.misc.ConstrainedFloatValue)*
- *chip_uri (Optional[str])*
- *dataset (rastervision.core.data.dataset_config.DatasetConfig)*
- *eval_uri (Optional[str])*
- *evaluators (List[rastervision.core.evaluation.evaluator_config.EvaluatorConfig])*
- *plugin_versions (Optional[Dict[str, int]])*
- *predict_batch_sz (int)*
- *predict_chip_sz (int)*
- *predict_uri (Optional[str])*
- *root_uri (str)*
- *rv_config (dict)*
- *source_bundle_uri (Optional[str])*
- *train_chip_sz (int)*
- *train_uri (Optional[str])*
- *type_hint (Literal['rv_pipeline'])*

field analyze_uri: Optional[str] = None

URI for output of analyze. If None, will be auto-generated.

field analyzers: List[*AnalyzerConfig*] = []

Analyzers to run during analyzer command. A StatsAnalyzer will be added automatically if any scenes have a RasterTransformer.

field backend: *BackendConfig* [Required]

Backend to use for interfacing with ML library.

field bundle_uri: Optional[str] = None

URI for output of bundle. If None, will be auto-generated.

field chip_nodata_threshold: Proportion = 1

Discard chips where the proportion of NODATA values is greater than or equal to this value. Might result in false positives if there are many legitimate black pixels in the chip. Use with caution.

Constraints

- **minimum** = 0

- **maximum** = 1

field chip_uri: Optional[str] = None

URI for output of chip. If None, will be auto-generated.

field dataset: *DatasetConfig* [Required]

Dataset containing train, validation, and optional test scenes.

field eval_uri: Optional[str] = None

URI for output of eval. If None, will be auto-generated.

field evaluators: List[*EvaluatorConfig*] = []

Evaluators to run during analyzer command. If list is empty the default evaluator is added.

field plugin_versions: Optional[Dict[str, int]] = None

Used to store a mapping of plugin module paths to the latest version number. This should not be set explicitly by users – it is set automatically when serializing and saving the config to disk.

field predict_batch_sz: int = 8

Batch size to use during prediction.

field predict_chip_sz: int = 300

Size of predictions chips in pixels.

field predict_uri: Optional[str] = None

URI for output of predict. If None, will be auto-generated.

field root_uri: str = None

The root URI for output generated by the pipeline

field rv_config: dict = None

Used to store serialized RVConfig so pipeline can run in remote environment with the local RVConfig. This should not be set explicitly by users – it is only used by the runner when running a remote pipeline.

field source_bundle_uri: Optional[str] = None

If provided, the model will be loaded from this bundle for the train stage. Useful for fine-tuning.

field train_chip_sz: int = 300

Size of training chips in pixels.

field train_uri: Optional[str] = None

URI for output of train. If None, will be auto-generated.

field type_hint: Literal['rv_pipeline'] = 'rv_pipeline'

build(*tmp_dir: str*) → *Pipeline*

Return a pipeline based on this configuration.

Subclasses should override this to return an instance of the corresponding subclass of Pipeline.

Parameters
tmp_dir (*str*) – root of any temporary directory to pass to pipeline

Return type
Pipeline

get_config_uri() → str

Get URI of serialized version of this PipelineConfig.

Return type
str

`get_default_evaluator`() → *EvaluatorConfig*

Returns a default EvaluatorConfig to use if one isn't set.

Return type
EvaluatorConfig

`get_default_label_store`(*scene:* SceneConfig) → *LabelStoreConfig*

Returns a default LabelStoreConfig to fill in any missing ones.

Parameters
scene (SceneConfig) –

Return type
LabelStoreConfig

`get_model_bundle_uri`()

`recursive_validate_config`()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

`revalidate`()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

`update`()

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

`validate_config`()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

`validate_list`(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises
ConfigError – if field is invalid

semantic_segmentation

Classes

SemanticSegmentation

SemanticSegmentation

class SemanticSegmentation

Bases: *RVPipeline*

Attributes

commands	Built-in mutable sequence.
gpu_commands	Built-in mutable sequence.
split_commands	Built-in mutable sequence.

__init__(*config:* RVPipelineConfig, *tmp_dir:* *str*)

Constructor

Parameters

- **config** (RVPipelineConfig) – the configuration of this pipeline
- **tmp_dir** (*str*) – the root any temporary directories created by running this pipeline

Methods

__init__(config, tmp_dir)	Constructor
analyze()	Run each analyzer over training scenes.
bundle()	Save a model bundle with whatever is needed to make predictions.
chip(*args, **kwargs)	Save training and validation chips.
eval()	Evaluate predictions against ground truth.
get_train_labels(window, scene)	Return the training labels in a window for a scene.
get_train_windows(scene)	Return the training windows for a Scene.
post_process_batch(windows, chips, labels)	Post-process a batch of predictions.
post_process_predictions(labels, scene)	Post-process all labels at end of prediction.
post_process_sample(sample)	Post-process sample in pipeline-specific way.
predict([split_ind, num_splits])	Make predictions over each validation and test scene.
predict_scene(scene, backend)	
test_cpu([split_ind, num_splits])	A command to test the ability to run split jobs on CPU.
test_gpu()	A command to test the ability to run on GPU.
train()	Train a model and save it.

__init__(*config:* RVPipelineConfig, *tmp_dir:* *str*)

Constructor

Parameters

- **config** (RVPipelineConfig) – the configuration of this pipeline
- **tmp_dir** (*str*) – the root any temporary directories created by running this pipeline

analyze()

Run each analyzer over training scenes.

bundle()

Save a model bundle with whatever is needed to make predictions.

The model bundle is a zip file and it is used by the Predictor and predict CLI subcommand.

chip(**args*, ***kwargs*)

Save training and validation chips.

eval()

Evaluate predictions against ground truth.

get_train_labels(*window*, *scene*)

Return the training labels in a window for a scene.

Returns
Labels that lie within window

get_train_windows(*scene*)

Return the training windows for a Scene.

Each training window represents the spatial extent of a training chip to generate.

Parameters
scene – Scene to generate windows for

post_process_batch(*windows*, *chips*, *labels*)

Post-process a batch of predictions.

post_process_predictions(*labels:* Labels, *scene:* Scene) → *Labels*

Post-process all labels at end of prediction.

Parameters

- **labels** (Labels) –
- **scene** (Scene) –

Return type
Labels

post_process_sample(*sample: DataSample*) → DataSample

Post-process sample in pipeline-specific way.

This should be called before writing a sample during chipping.

Parameters
sample (*DataSample*) –

Return type
DataSample

predict(*split_ind=0*, *num_splits=1*)

Make predictions over each validation and test scene.

This uses a sliding window.

predict_scene(*scene:* Scene, *backend:* Backend) → *Labels*

Parameters

- **scene** (Scene) –
- **backend** (Backend) –

Return type
Labels

test_cpu(*split_ind: int = 0*, *num_splits: int = 1*)

A command to test the ability to run split jobs on CPU.

Parameters

- **split_ind** (*int*) –
- **num_splits** (*int*) –

test_gpu()

A command to test the ability to run on GPU.

train()

Train a model and save it.

property commands

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

property gpu_commands

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

property split_commands

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

Functions

get_train_windows(scene, class_config, ...)	Get training windows covering a scene.

get_train_windows

get_train_windows(*scene: Scene*, *class_config: ClassConfig*, *chip_size: int*, *chip_options: SemanticSegmentationChipOptions*, *chip_nodata_threshold: float = 1.0*) → List[*Box*]

Get training windows covering a scene.

Parameters

- **scene** (Scene) – The scene over-which windows are to be generated.
- **class_config** (ClassConfig) –
- **chip_size** (*int*) –
- **chip_options** (SemanticSegmentationChipOptions) –
- **chip_nodata_threshold** (*float*) –

Returns
A list of windows, list(Box)

Return type
List[Box]

semantic_segmentation_config

Classes

SemanticSegmentationWindowMethod	Enum for window methods

SemanticSegmentationWindowMethod

class SemanticSegmentationWindowMethod

Bases: Enum

Enum for window methods

sliding
use a sliding window

random_sample
randomly sample windows

Attributes

sliding
random_sample

__init__()

random_sample = 'random_sample'

sliding = 'sliding'

Configs

SemanticSegmentationChipOptions	Chipping options for semantic segmentation.
SemanticSegmentationConfig	Configure a *SemanticSegmentation* pipeline.
SemanticSegmentationPredictOptions	

SemanticSegmentationChipOptions

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model SemanticSegmentationChipOptions

Chipping options for semantic segmentation.


```
{
   "title": "SemanticSegmentationChipOptions",
   "description": "Chipping options for semantic segmentation.",
   "type": "object",
   "properties": {
      "window_method": {
         "description": "Window method to use for chipping.",
         "default": "sliding",
         "allOf": [
            {
               "$ref": "#/definitions/SemanticSegmentationWindowMethod"
            }
         ]
      },
      "target_class_ids": {
         "title": "Target Class Ids",
         "description": "List of class ids considered as targets (ie. those to␣
↪prioritize when creating chips) which is only used in conjunction with the target_
↪count_threshold and negative_survival_probability options. Applies to the random_
↪sample window method.",
         "type": "array",
         "items": {
            "type": "integer"
         }
      },
      "negative_survival_prob": {
         "title": "Negative Survival Prob",
         "description": "List of class ids considered as targets (ie. those to␣
↪prioritize when creating chips) which is only used in conjunction with the target_
↪count_threshold and negative_survival_probability options. Applies to the random_
↪sample window method.",
         "default": 1.0,
         "type": "number"
```


(continues on next page)

(continued from previous page)


```
      },
      "chips_per_scene": {
         "title": "Chips Per Scene",
         "description": "Number of chips to generate per scene. Applies to the␣
↪random_sample window method.",
         "default": 1000,
         "type": "integer"
      },
      "target_count_threshold": {
         "title": "Target Count Threshold",
         "description": "Minimum number of pixels covering target_classes that a␣
↪chip must have. Applies to the random_sample window method.",
         "default": 1000,
         "type": "integer"
      },
      "stride": {
         "title": "Stride",
         "description": "Stride of windows across image. Defaults to half the chip␣
↪size. Applies to the sliding_window method.",
         "type": "integer"
      },
      "type_hint": {
         "title": "Type Hint",
         "default": "semantic_segmentation_chip_options",
         "enum": [
            "semantic_segmentation_chip_options"
         ],
         "type": "string"
      }
   },
   "additionalProperties": false,
   "definitions": {
      "SemanticSegmentationWindowMethod": {
         "title": "SemanticSegmentationWindowMethod",
         "description": "Enum for window methods\n\n    Attributes:\n       ␣
↪sliding: use a sliding window\n        random_sample: randomly sample windows\n  ␣
↪ ",
         "enum": [
            "sliding",
            "random_sample"
         ]
      }
   }
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *chips_per_scene (int)*

- *negative_survival_prob (float)*
- *stride (Optional[int])*
- *target_class_ids (Optional[List[int]])*
- *target_count_threshold (int)*
- *type_hint (Literal['semantic_segmentation_chip_options'])*
- *window_method (rastervision.core.rv_pipeline.semantic_segmentation_config.SemanticSegmentationWindowMethod)*

field chips_per_scene: int = 1000

Number of chips to generate per scene. Applies to the random_sample window method.

field negative_survival_prob: float = 1.0

List of class ids considered as targets (ie. those to prioritize when creating chips) which is only used in conjunction with the target_count_threshold and negative_survival_probability options. Applies to the random_sample window method.

field stride: Optional[int] = None

Stride of windows across image. Defaults to half the chip size. Applies to the sliding_window method.

field target_class_ids: Optional[List[int]] = None

List of class ids considered as targets (ie. those to prioritize when creating chips) which is only used in conjunction with the target_count_threshold and negative_survival_probability options. Applies to the random_sample window method.

field target_count_threshold: int = 1000

Minimum number of pixels covering target_classes that a chip must have. Applies to the random_sample window method.

field type_hint: Literal['semantic_segmentation_chip_options'] = 'semantic_segmentation_chip_options'

field window_method: *SemanticSegmentationWindowMethod* = SemanticSegmentationWindowMethod.sliding

Window method to use for chipping.

build()

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(**args*, ***kwargs*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

SemanticSegmentationConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model SemanticSegmentationConfig

Configure a *SemanticSegmentation* pipeline.


```
{
   "title": "SemanticSegmentationConfig",
   "description": "Configure a :class:`.SemanticSegmentation` pipeline.",
   "type": "object",
   "properties": {
      "root_uri": {
         "title": "Root Uri",
         "description": "The root URI for output generated by the pipeline",
         "type": "string"
      },
      "rv_config": {
         "title": "Rv Config",
         "description": "Used to store serialized RVConfig so pipeline can run in␣
↪remote environment with the local RVConfig. This should not be set explicitly by␣
↪users -- it is only used by the runner when running a remote pipeline.",
         "type": "object"
      },
      "plugin_versions": {
         "title": "Plugin Versions",
         "description": "Used to store a mapping of plugin module paths to the␣
↪latest version number. This should not be set explicitly by users -- it is set␣
↪automatically when serializing and saving the config to disk.",
         "type": "object",
```


(continues on next page)

(continued from previous page)


```
            "additionalProperties": {
               "type": "integer"
            }
         },
         "type_hint": {
            "title": "Type Hint",
            "default": "semantic_segmentation",
            "enum": [
               "semantic_segmentation"
            ],
            "type": "string"
         },
         "dataset": {
            "title": "Dataset",
            "description": "Dataset containing train, validation, and optional test␣
↪scenes.",
            "allOf": [
               {
                  "$ref": "#/definitions/DatasetConfig"
               }
            ]
         },
         "backend": {
            "title": "Backend",
            "description": "Backend to use for interfacing with ML library.",
            "allOf": [
               {
                  "$ref": "#/definitions/BackendConfig"
               }
            ]
         },
         "evaluators": {
            "title": "Evaluators",
            "description": "Evaluators to run during analyzer command. If list is␣
↪empty the default evaluator is added.",
            "default": [],
            "type": "array",
            "items": {
               "$ref": "#/definitions/EvaluatorConfig"
            }
         },
         "analyzers": {
            "title": "Analyzers",
            "description": "Analyzers to run during analyzer command. A StatsAnalyzer␣
↪will be added automatically if any scenes have a RasterTransformer.",
            "default": [],
            "type": "array",
            "items": {
               "$ref": "#/definitions/AnalyzerConfig"
            }
         },
         "train_chip_sz": {
```


(continues on next page)

(continued from previous page)

```
         "title": "Train Chip Sz",
         "description": "Size of training chips in pixels.",
         "default": 300,
         "type": "integer"
      },
      "predict_chip_sz": {
         "title": "Predict Chip Sz",
         "description": "Size of predictions chips in pixels.",
         "default": 300,
         "type": "integer"
      },
      "predict_batch_sz": {
         "title": "Predict Batch Sz",
         "description": "Batch size to use during prediction.",
         "default": 8,
         "type": "integer"
      },
      "chip_nodata_threshold": {
         "title": "Chip Nodata Threshold",
         "description": "Discard chips where the proportion of NODATA values is␣
↪greater than or equal to this value. Might result in false positives if there are␣
↪many legitimate black pixels in the chip. Use with caution.",
         "default": 1,
         "minimum": 0,
         "maximum": 1,
         "type": "number"
      },
      "analyze_uri": {
         "title": "Analyze Uri",
         "description": "URI for output of analyze. If None, will be auto-generated.
↪",
         "type": "string"
      },
      "chip_uri": {
         "title": "Chip Uri",
         "description": "URI for output of chip. If None, will be auto-generated.",
         "type": "string"
      },
      "train_uri": {
         "title": "Train Uri",
         "description": "URI for output of train. If None, will be auto-generated.",
         "type": "string"
      },
      "predict_uri": {
         "title": "Predict Uri",
         "description": "URI for output of predict. If None, will be auto-generated.
↪",
         "type": "string"
      },
      "eval_uri": {
         "title": "Eval Uri",
         "description": "URI for output of eval. If None, will be auto-generated.",
```

(continues on next page)

(continued from previous page)


```
            "type": "string"
         },
         "bundle_uri": {
            "title": "Bundle Uri",
            "description": "URI for output of bundle. If None, will be auto-generated.
↪",
            "type": "string"
         },
         "source_bundle_uri": {
            "title": "Source Bundle Uri",
            "description": "If provided, the model will be loaded from this bundle for␣
↪the train stage. Useful for fine-tuning.",
            "type": "string"
         },
         "chip_options": {
            "title": "Chip Options",
            "default": {
               "window_method": "SemanticSegmentationWindowMethod.sliding",
               "target_class_ids": null,
               "negative_survival_prob": 1.0,
               "chips_per_scene": 1000,
               "target_count_threshold": 1000,
               "stride": null,
               "type_hint": "semantic_segmentation_chip_options"
            },
            "allOf": [
               {
                  "$ref": "#/definitions/SemanticSegmentationChipOptions"
               }
            ]
         },
         "predict_options": {
            "title": "Predict Options",
            "default": {
               "type_hint": "semantic_segmentation_predict_options",
               "stride": null,
               "crop_sz": null
            },
            "allOf": [
               {
                  "$ref": "#/definitions/SemanticSegmentationPredictOptions"
               }
            ]
         }
      },
      "required": [
         "dataset",
         "backend"
      ],
      "additionalProperties": false,
      "definitions": {
         "ClassConfig": {
```


(continues on next page)

(continued from previous page)

```
        "title": "ClassConfig",
        "description": "Configure class information for a machine learning task.",
        "type": "object",
        "properties": {
           "names": {
              "title": "Names",
              "description": "Names of classes. The i-th class in this list will␣
↪have class ID = i.",
              "type": "array",
              "items": {
                 "type": "string"
              }
           },
           "colors": {
              "title": "Colors",
              "description": "Colors used to visualize classes. Can be color␣
↪strings accepted by matplotlib or RGB tuples. If None, a random color will be␣
↪auto-generated for each class.",
              "type": "array",
              "items": {
                 "anyOf": [
                    {
                       "type": "string"
                    },
                    {
                       "type": "array",
                       "items": {}
                    }
                 ]
              }
           },
           "null_class": {
              "title": "Null Class",
              "description": "Optional name of class in `names` to use as the null␣
↪class. This is used in semantic segmentation to represent the label for imagery␣
↪pixels that are NODATA or that are missing a label. If None and the class names␣
↪include \"null\", it will automatically be used as the null class. If None, and␣
↪this Config is part of a SemanticSegmentationConfig, a null class will be added␣
↪automatically.",
              "type": "string"
           },
           "type_hint": {
              "title": "Type Hint",
              "default": "class_config",
              "enum": [
                 "class_config"
              ],
              "type": "string"
           }
        },
        "required": [
           "names"
```

(continues on next page)

(continued from previous page)


```
        ],
        "additionalProperties": false
     },
     "RasterTransformerConfig": {
        "title": "RasterTransformerConfig",
        "description": "Configure a :class:`.RasterTransformer`.",
        "type": "object",
        "properties": {
           "type_hint": {
              "title": "Type Hint",
              "default": "raster_transformer",
              "enum": [
                 "raster_transformer"
              ],
              "type": "string"
           }
        },
        "additionalProperties": false
     },
     "RasterSourceConfig": {
        "title": "RasterSourceConfig",
        "description": "Configure a :class:`.RasterSource`.",
        "type": "object",
        "properties": {
           "channel_order": {
              "title": "Channel Order",
              "description": "The sequence of channel indices to use when reading␣
↪imagery.",
              "type": "array",
              "items": {
                 "type": "integer"
              }
           },
           "transformers": {
              "title": "Transformers",
              "default": [],
              "type": "array",
              "items": {
                 "$ref": "#/definitions/RasterTransformerConfig"
              }
           },
           "extent": {
              "title": "Extent",
              "description": "Use-specified extent in pixel coords in the form␣
↪(ymin, xmin, ymax, xmax). Useful for cropping the raster source so that only part␣
↪of the raster is read from.",
              "type": "array",
              "minItems": 4,
              "maxItems": 4,
              "items": [
                 {
                    "type": "integer"
```


(continues on next page)

(continued from previous page)


```
                },
                {
                    "type": "integer"
                },
                {
                    "type": "integer"
                },
                {
                    "type": "integer"
                }
            ]
        },
        "type_hint": {
            "title": "Type Hint",
            "default": "raster_source",
            "enum": [
                "raster_source"
            ],
            "type": "string"
        }
    },
    "additionalProperties": false
},
"LabelSourceConfig": {
    "title": "LabelSourceConfig",
    "description": "Configure a :class:`.LabelSource`.",
    "type": "object",
    "properties": {
        "type_hint": {
            "title": "Type Hint",
            "default": "label_source",
            "enum": [
                "label_source"
            ],
            "type": "string"
        }
    },
    "additionalProperties": false
},
"LabelStoreConfig": {
    "title": "LabelStoreConfig",
    "description": "Configure a :class:`.LabelStore`.",
    "type": "object",
    "properties": {
        "type_hint": {
            "title": "Type Hint",
            "default": "label_store",
            "enum": [
                "label_store"
            ],
            "type": "string"
        }
```


(continues on next page)

(continued from previous page)


```
        },
        "additionalProperties": false
     },
     "SceneConfig": {
        "title": "SceneConfig",
        "description": "Configure a :class:`.Scene` comprising raster data &␣
↪labels for an AOI.\n    ",
        "type": "object",
        "properties": {
           "id": {
              "title": "Id",
              "type": "string"
           },
           "raster_source": {
              "$ref": "#/definitions/RasterSourceConfig"
           },
           "label_source": {
              "$ref": "#/definitions/LabelSourceConfig"
           },
           "label_store": {
              "$ref": "#/definitions/LabelStoreConfig"
           },
           "aoi_uris": {
              "title": "Aoi Uris",
              "description": "List of URIs of GeoJSON files that define the AOIs␣
↪for the scene. Each polygon defines an AOI which is a piece of the scene that is␣
↪assumed to be fully labeled and usable for training or validation. The AOIs are␣
↪assumed to be in EPSG:4326 coordinates.",
              "type": "array",
              "items": {
                 "type": "string"
              }
           },
           "type_hint": {
              "title": "Type Hint",
              "default": "scene",
              "enum": [
                 "scene"
              ],
              "type": "string"
           }
        },
        "required": [
           "id",
           "raster_source"
        ],
        "additionalProperties": false
     },
     "DatasetConfig": {
        "title": "DatasetConfig",
        "description": "Configure train, validation, and test splits for a dataset.
↪",
```


(continues on next page)

(continued from previous page)


```
        "type": "object",
        "properties": {
           "class_config": {
              "$ref": "#/definitions/ClassConfig"
           },
           "train_scenes": {
              "title": "Train Scenes",
              "type": "array",
              "items": {
                 "$ref": "#/definitions/SceneConfig"
              }
           },
           "validation_scenes": {
              "title": "Validation Scenes",
              "type": "array",
              "items": {
                 "$ref": "#/definitions/SceneConfig"
              }
           },
           "test_scenes": {
              "title": "Test Scenes",
              "default": [],
              "type": "array",
              "items": {
                 "$ref": "#/definitions/SceneConfig"
              }
           },
           "scene_groups": {
              "title": "Scene Groups",
              "description": "Groupings of scenes. Should be a dict of the form: {
↪<group-name>: Set(scene_id_1, scene_id_2, ...)}. Three groups are added by␣
↪default: \"train_scenes\", \"validation_scenes\", and \"test_scenes\"",
              "default": {},
              "type": "object",
              "additionalProperties": {
                 "type": "array",
                 "items": {
                    "type": "string"
                 },
                 "uniqueItems": true
              }
           },
           "type_hint": {
              "title": "Type Hint",
              "default": "dataset",
              "enum": [
                 "dataset"
              ],
              "type": "string"
           }
        },
        "required": [
```


(continues on next page)

(continued from previous page)


```
            "class_config",
            "train_scenes",
            "validation_scenes"
         ],
         "additionalProperties": false
      },
      "BackendConfig": {
         "title": "BackendConfig",
         "description": "Configure a :class:`.Backend`.",
         "type": "object",
         "properties": {
            "type_hint": {
               "title": "Type Hint",
               "default": "backend",
               "enum": [
                  "backend"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      },
      "EvaluatorConfig": {
         "title": "EvaluatorConfig",
         "description": "Configure an :class:`.Evaluator`.",
         "type": "object",
         "properties": {
            "output_uri": {
               "title": "Output Uri",
               "description": "URI of directory where evaluator output will be␣
↪saved. Evaluations for each scene-group will be save in a JSON file at <output_
↪uri>/<scene-group-name>/eval.json. If None, and this Config is part of an␣
↪RVPipeline, this field will be auto-generated.",
               "type": "string"
            },
            "type_hint": {
               "title": "Type Hint",
               "default": "evaluator",
               "enum": [
                  "evaluator"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      },
      "AnalyzerConfig": {
         "title": "AnalyzerConfig",
         "description": "Configure an :class:`.Analyzer`.",
         "type": "object",
         "properties": {
            "type_hint": {
```


(continues on next page)

(continued from previous page)


```
                "title": "Type Hint",
                "default": "analyzer",
                "enum": [
                   "analyzer"
                ],
                "type": "string"
             }
          },
          "additionalProperties": false
       },
       "SemanticSegmentationWindowMethod": {
          "title": "SemanticSegmentationWindowMethod",
          "description": "Enum for window methods\n\n    Attributes:\n        ␣
↪sliding: use a sliding window\n        random_sample: randomly sample windows\n  ␣
↪ ",
          "enum": [
             "sliding",
             "random_sample"
          ]
       },
       "SemanticSegmentationChipOptions": {
          "title": "SemanticSegmentationChipOptions",
          "description": "Chipping options for semantic segmentation.",
          "type": "object",
          "properties": {
             "window_method": {
                "description": "Window method to use for chipping.",
                "default": "sliding",
                "allOf": [
                   {
                      "$ref": "#/definitions/SemanticSegmentationWindowMethod"
                   }
                ]
             },
             "target_class_ids": {
                "title": "Target Class Ids",
                "description": "List of class ids considered as targets (ie. those␣
↪to prioritize when creating chips) which is only used in conjunction with the␣
↪target_count_threshold and negative_survival_probability options. Applies to the␣
↪random_sample window method.",
                "type": "array",
                "items": {
                   "type": "integer"
                }
             },
             "negative_survival_prob": {
                "title": "Negative Survival Prob",
                "description": "List of class ids considered as targets (ie. those␣
↪to prioritize when creating chips) which is only used in conjunction with the␣
↪target_count_threshold and negative_survival_probability options. Applies to the␣
↪random_sample window method.",
                "default": 1.0,
```


(continues on next page)

(continued from previous page)


```
                "type": "number"
            },
            "chips_per_scene": {
                "title": "Chips Per Scene",
                "description": "Number of chips to generate per scene. Applies to␣
↪the random_sample window method.",
                "default": 1000,
                "type": "integer"
            },
            "target_count_threshold": {
                "title": "Target Count Threshold",
                "description": "Minimum number of pixels covering target_classes␣
↪that a chip must have. Applies to the random_sample window method.",
                "default": 1000,
                "type": "integer"
            },
            "stride": {
                "title": "Stride",
                "description": "Stride of windows across image. Defaults to half the␣
↪chip size. Applies to the sliding_window method.",
                "type": "integer"
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "semantic_segmentation_chip_options",
                "enum": [
                    "semantic_segmentation_chip_options"
                ],
                "type": "string"
            }
        },
        "additionalProperties": false
    },
    "SemanticSegmentationPredictOptions": {
        "title": "SemanticSegmentationPredictOptions",
        "description": "Base class that can be extended to provide custom␣
↪configurations.\n\nThis adds some extra methods to Pydantic BaseModel.\nSee https:
↪//pydantic-docs.helpmanual.io/\n\nThe general idea is that configuration schemas␣
↪can be defined by\nsubclassing this and adding class attributes with types and\
↪ndefault values for each field. Configs can be defined hierarchically,\nie. a␣
↪Config can have fields which are of type Config.\nValidation, serialization,␣
↪deserialization, and IDE support is\nprovided automatically based on this schema.
↪",
        "type": "object",
        "properties": {
            "type_hint": {
                "title": "Type Hint",
                "default": "semantic_segmentation_predict_options",
                "enum": [
                    "semantic_segmentation_predict_options"
                ],
                "type": "string"
```


(continues on next page)

(continued from previous page)


```
            },
            "stride": {
               "title": "Stride",
               "description": "Stride of windows across image. Allows aggregating␣
↪multiple predictions for each pixel if less than the chip size. Defaults to␣
↪predict_chip_sz.",
               "type": "integer"
            },
            "crop_sz": {
               "title": "Crop Sz",
               "description": "Number of rows/columns of pixels from the edge of␣
↪prediction windows to discard. This is useful because predictions near edges tend␣
↪to be lower quality and can result in very visible artifacts near the edges of␣
↪chips. If \"auto\", will be set to half the stride if stride is less than chip_sz.
↪ Defaults to None.",
               "anyOf": [
                  {
                     "type": "integer",
                     "exclusiveMinimum": 0
                  },
                  {
                     "enum": [
                        "auto"
                     ],
                     "type": "string"
                  }
               ]
            }
         },
         "additionalProperties": false
      }
   }
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *analyze_uri ()*
- *analyzers ()*
- *backend ()*
- *bundle_uri ()*
- *chip_nodata_threshold ()*
- *chip_options ()*
- *chip_uri ()*
- *dataset ()*

- *eval_uri ()*
- *evaluators ()*
- *plugin_versions ()*
- *predict_batch_sz ()*
- *predict_chip_sz ()*
- *predict_options ()*
- *predict_uri ()*
- *root_uri ()*
- *rv_config ()*
- *source_bundle_uri ()*
- *train_chip_sz ()*
- *train_uri ()*
- *type_hint (Literal['semantic_segmentation'])*

field analyze_uri: Optional[str] = None

URI for output of analyze. If None, will be auto-generated.

field analyzers: List[*AnalyzerConfig*] = []

Analyzers to run during analyzer command. A StatsAnalyzer will be added automatically if any scenes have a RasterTransformer.

field backend: *BackendConfig* [Required]

Backend to use for interfacing with ML library.

field bundle_uri: Optional[str] = None

URI for output of bundle. If None, will be auto-generated.

field chip_nodata_threshold: Proportion = 1

Discard chips where the proportion of NODATA values is greater than or equal to this value. Might result in false positives if there are many legitimate black pixels in the chip. Use with caution.

Constraints

- **minimum** = 0
- **maximum** = 1

field chip_options: *SemanticSegmentationChipOptions* = SemanticSegmentationChipOptions(window_method=<SemanticSegmentationWindowMethod.sliding: 'sliding'>, target_class_ids=None, negative_survival_prob=1.0, chips_per_scene=1000, target_count_threshold=1000, stride=None)

field chip_uri: Optional[str] = None

URI for output of chip. If None, will be auto-generated.

field dataset: *DatasetConfig* [Required]

Dataset containing train, validation, and optional test scenes.

field eval_uri: Optional[str] = None

URI for output of eval. If None, will be auto-generated.

field evaluators: List[*EvaluatorConfig*] = []

Evaluators to run during analyzer command. If list is empty the default evaluator is added.

field plugin_versions: Optional[Dict[str, int]] = None

Used to store a mapping of plugin module paths to the latest version number. This should not be set explicitly by users – it is set automatically when serializing and saving the config to disk.

field predict_batch_sz: int = 8

Batch size to use during prediction.

field predict_chip_sz: int = 300

Size of predictions chips in pixels.

field predict_options: *SemanticSegmentationPredictOptions* = SemanticSegmentationPredictOptions(stride=None, crop_sz=None)

field predict_uri: Optional[str] = None

URI for output of predict. If None, will be auto-generated.

field root_uri: str = None

The root URI for output generated by the pipeline

field rv_config: dict = None

Used to store serialized RVConfig so pipeline can run in remote environment with the local RVConfig. This should not be set explicitly by users – it is only used by the runner when running a remote pipeline.

field source_bundle_uri: Optional[str] = None

If provided, the model will be loaded from this bundle for the train stage. Useful for fine-tuning.

field train_chip_sz: int = 300

Size of training chips in pixels.

field train_uri: Optional[str] = None

URI for output of train. If None, will be auto-generated.

field type_hint: Literal['semantic_segmentation'] = 'semantic_segmentation'

build(*tmp_dir*)

Return a pipeline based on this configuration.

Subclasses should override this to return an instance of the corresponding subclass of Pipeline.

Parameters
tmp_dir – root of any temporary directory to pass to pipeline

get_config_uri() → str

Get URI of serialized version of this PipelineConfig.

Return type
str

get_default_evaluator()

Returns a default EvaluatorConfig to use if one isn't set.

get_default_label_store(*scene*)

Returns a default LabelStoreConfig to fill in any missing ones.

get_model_bundle_uri()

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update()

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

SemanticSegmentationPredictOptions

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model SemanticSegmentationPredictOptions


```
{
   "title": "SemanticSegmentationPredictOptions",
   "description": "Base class that can be extended to provide custom configurations.
↪\n\nThis adds some extra methods to Pydantic BaseModel.\nSee https://pydantic-
↪docs.helpmanual.io/\n\nThe general idea is that configuration schemas can be␣
↪defined by\nsubclassing this and adding class attributes with types and\ndefault␣
↪values for each field. Configs can be defined hierarchically,\nie. a Config can␣
↪have fields which are of type Config.\nValidation, serialization, deserialization,
↪ and IDE support is\nprovided automatically based on this schema.",
   "type": "object",
   "properties": {
      "type_hint": {
         "title": "Type Hint",
         "default": "semantic_segmentation_predict_options",
```


(continues on next page)

(continued from previous page)


```
            "enum": [
               "semantic_segmentation_predict_options"
            ],
            "type": "string"
         },
         "stride": {
            "title": "Stride",
            "description": "Stride of windows across image. Allows aggregating␣
↪multiple predictions for each pixel if less than the chip size. Defaults to␣
↪predict_chip_sz.",
            "type": "integer"
         },
         "crop_sz": {
            "title": "Crop Sz",
            "description": "Number of rows/columns of pixels from the edge of␣
↪prediction windows to discard. This is useful because predictions near edges tend␣
↪to be lower quality and can result in very visible artifacts near the edges of␣
↪chips. If \"auto\", will be set to half the stride if stride is less than chip_sz.
↪ Defaults to None.",
            "anyOf": [
               {
                  "type": "integer",
                  "exclusiveMinimum": 0
               },
               {
                  "enum": [
                     "auto"
                  ],
                  "type": "string"
               }
            ]
         }
      },
      "additionalProperties": false
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *crop_sz (Optional[Union[rastervision.core.rv_pipeline.semantic_segmentation_config.ConstrainedIntValue, Literal['auto']]])*
- *stride (Optional[int])*
- *type_hint (Literal['semantic_segmentation_predict_options'])*

Validators

- *validate_crop_sz* » *crop_sz*

field crop_sz: Optional[Union[ConstrainedIntValue, Literal['auto']]] = None

Number of rows/columns of pixels from the edge of prediction windows to discard. This is useful because predictions near edges tend to be lower quality and can result in very visible artifacts near the edges of chips. If "auto", will be set to half the stride if stride is less than chip_sz. Defaults to None.

Validated by

- *validate_crop_sz*

field stride: Optional[int] = None

Stride of windows across image. Allows aggregating multiple predictions for each pixel if less than the chip size. Defaults to predict_chip_sz.

field type_hint: Literal['semantic_segmentation_predict_options'] = 'semantic_segmentation_predict_options'

build()

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(**args*, ***kwargs*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validator validate_crop_sz » crop_sz

Parameters

- **v** (*Optional[Union[ConstrainedIntValue, Literal['auto']]]*) –
- **values** (*dict*) –

Return type
dict

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate

- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

utils

Functions

nodata_below_threshold(chip, threshold[, ...])	Check if proportion of nodata pixels is below the threshold.

nodata_below_threshold

nodata_below_threshold(*chip: ndarray*, *threshold: float*, *nodata_val: int = 0*) → bool

Check if proportion of nodata pixels is below the threshold.

Parameters

- **chip** (*np.ndarray*) – Raster as (H, W) or (H, W, C) numpy array.
- **threshold** (*float*) – Threshold to check the fraction of NODATA pixels against.
- **nodata_val** (*int, optional*) – Value that represents NODATA pixels. Defaults to 0.

Returns

Whether the fraction of NODATA pixels is below the given threshold.

Return type

bool

9.2.11 utils

Modules

cog
filter_geojson
misc
stac

cog

Functions

create_cog(source_uri, dest_uri, local_dir)	
gdal_cog_commands(input_path, tmp_dir[, ...])	GDAL commands to create a COG from an input file.
run_cmd(cmd)	

create_cog

create_cog(*source_uri*, *dest_uri*, *local_dir*, *block_size=512*, *resample_method='near'*, *compression='deflate'*, *overviews=None*)

gdal_cog_commands

gdal_cog_commands(*input_path*, *tmp_dir*, *block_size=512*, *resample_method='near'*, *compression='deflate'*, *overviews=None*)

GDAL commands to create a COG from an input file. Returns a tuple (commands, output_path)

run_cmd

run_cmd(*cmd*)

filter_geojson

misc

Classes

Proportion	alias of ConstrainedFloatValue

Proportion

Proportion

alias of ConstrainedFloatValue

Functions

numpy_to_png(array)	Get a PNG string from a Numpy array.
png_to_numpy(png[, dtype])	Get a Numpy array from a PNG string.
save_img(im_array, output_path)	

numpy_to_png

numpy_to_png(*array: ndarray*) → str

Get a PNG string from a Numpy array.

Parameters
array (*ndarray*) – A Numpy array of shape (w, h, 3) or (w, h), where the former is meant to become a three-channel image and the latter a one-channel image. The dtype of the array should be uint8.

Returns
str

Return type
str

png_to_numpy

png_to_numpy(*png: str*, *dtype=<class 'numpy.uint8'>*) → ndarray

Get a Numpy array from a PNG string.

Parameters
png (*str*) – A str containing a PNG-formatted image.

Returns
numpy.ndarray

Return type
ndarray

save_img

save_img(*im_array*, *output_path*)

stac

Functions

get_linked_image_item(label_item)	Find link in the item that has "rel" == "source" and return its "target" item.
is_label_item(item)	Resolve each extension schema into a dict, then check if it has the title of "Label Extension".
parse_stac(stac_uri[, item_limit])	Parse a STAC catalog JSON file to extract label URIs, images URIs, and AOIs.
read_stac(uri[, extract_dir])	Parse the contents of a STAC catalog (downloading it first, if remote).
setup_stac_io()	

get_linked_image_item

get_linked_image_item(*label_item: Item*) → Optional[Item]

Find link in the item that has “rel” == “source” and return its “target” item. If no such link, return None. If multiple such links, raise an exception.

Parameters
label_item (*Item*) –

Return type
Optional[*Item*]

is_label_item

is_label_item(*item: Item*) → bool

Resolve each extension schema into a dict, then check if it has the title of “Label Extension”.

Parameters
item (*Item*) –

Return type
bool

parse_stac

parse_stac(*stac_uri: str*, *item_limit: Optional[int] = None*) → List[dict]

Parse a STAC catalog JSON file to extract label URIs, images URIs, and AOIs.

Note: This has been tested to be compatible with STAC version 1.0.0 but not any other versions.

Parameters

- **stac_uri** (*str*) – Path to the STAC catalog JSON file.
- **item_limit** (*Optional[int]*) –

Returns

A lsit of dicts with keys: “label_uri”, “image_uris”,
”label_bbox”, “image_bbox”, “bboxes_intersect”, and “aoi_geometry”. Each dict corresponds to one label item and its associated image assets in the STAC catalog.

Return type
List[dict]

read_stac

read_stac(*uri: str*, *extract_dir: Optional[str] = None*, ***kwargs*) → List[dict]

Parse the contents of a STAC catalog (downloading it first, if remote). If the uri is a zip file, unzip it, find catalog.json inside it and parse that.

Parameters

- **uri** (*str*) – Either a URI to a STAC catalog JSON file or a URI to a zip file containing a STAC catalog JSON file.
- **extract_dir** (*Optional[str]*) –

Raises

- **FileNotFoundError** – If catalog.json is not found inside the zip file.
- **Exception** – If multiple catalog.json's are found inside the zip file.

Returns

A lsit of dicts with keys: "label_uri", "image_uris",
"label_bbox", "image_bbox", "bboxes_intersect", and "aoi_geometry". Each dict corresponds to one label item and its associated image assets in the STAC catalog.

Return type
List[dict]

setup_stac_io

setup_stac_io() → None

Return type
None

9.3 pytorch_learner

Modules

9.3.1 classification_learner

Classes

ClassificationLearner

class ClassificationLearner

Bases: *Learner*

__init__(*cfg:* LearnerConfig, *output_dir: Optional[str] = None*, *train_ds: Optional[Dataset] = None*, *valid_ds: Optional[Dataset] = None*, *test_ds: Optional[Dataset] = None*, *model: Optional[torch.nn.Module] = None*, *loss: Optional[Callable] = None*, *optimizer: Optional[Optimizer] = None*, *epoch_scheduler: Optional[_LRScheduler] = None*, *step_scheduler: Optional[_LRScheduler] = None*, *tmp_dir: Optional[str] = None*, *model_weights_path: Optional[str] = None*, *model_def_path: Optional[str] = None*, *loss_def_path: Optional[str] = None*, *training: bool = True*)

Constructor.

Parameters

- **cfg** (*LearnerConfig*) – LearnerConfig.
- **train_ds** (*Optional[Dataset], optional*) – The dataset to use for training. If None, will be generated from cfg.data. Defaults to None.
- **valid_ds** (*Optional[Dataset], optional*) – The dataset to use for validation. If None, will be generated from cfg.data. Defaults to None.
- **test_ds** (*Optional[Dataset], optional*) – The dataset to use for testing. If None, will be generated from cfg.data. Defaults to None.
- **model** (*Optional[nn.Module], optional*) – The model. If None, will be generated from cfg.model. Defaults to None.
- **loss** (*Optional[Callable], optional*) – The loss function. If None, will be generated from cfg.solver. Defaults to None.
- **optimizer** (*Optional[Optimizer], optional*) – The optimizer. If None, will be generated from cfg.solver. Defaults to None.
- **epoch_scheduler** (*Optional[_LRScheduler], optional*) – The scheduler that updates after each epoch. If None, will be generated from cfg.solver. Defaults to None.
- **step_scheduler** (*Optional[_LRScheduler], optional*) – The scheduler that updates after each optimizer-step. If None, will be generated from cfg.solver. Defaults to None.
- **tmp_dir** (*Optional[str], optional*) – A temporary directory to use for downloads etc. If None, will be auto-generated. Defaults to None.
- **model_weights_path** (*Optional[str], optional*) – URI of model weights to initialize the model with. Defaults to None.
- **model_def_path** (*Optional[str], optional*) – A local path to a directory with a hubconf.py. If provided, the model definition is imported from here. This is used when loading an external model from a model-bundle. Defaults to None.
- **loss_def_path** (*Optional[str], optional*) – A local path to a directory with a hubconf.py. If provided, the loss function definition is imported from here. This is used when loading an external loss function from a model-bundle. Defaults to None.
- **training** (*bool, optional*) – If False, the training apparatus (loss, optimizer, scheduler, logging, etc.) will not be set up and the model will be put into eval mode. If True, the training apparatus will be set up and the model will be put into training mode. Defaults to True.

- **output_dir** (*Optional[str]*) –

Methods

__init__(cfg[, output_dir, train_ds, ...])	Constructor.
build_dataloaders()	Set the DataLoaders for train, validation, and test sets.
build_datasets()	
build_epoch_scheduler([start_epoch])	Returns an LR scheduler that changes the LR each epoch.
build_loss([loss_def_path])	Build a loss Callable.
build_metric_names()	Returns names of metrics used to validate model at each epoch.
build_model([model_def_path])	Build a PyTorch model.
build_optimizer()	Returns optimizer.
build_step_scheduler([start_epoch])	Returns an LR scheduler that changes the LR each step.
eval_model(split)	Evaluate model using a particular dataset split.
from_model_bundle(model_bundle_uri[, ...])	Create a Learner from a model bundle.
get_collate_fn()	Returns a custom collate_fn to use in DataLoader.
get_dataloader(split)	Get the DataLoader for a split.
get_start_epoch()	Get start epoch.
get_train_sampler(train_ds)	Return a sampler to use for the training dataloader or None to not use any.
get_visualizer_class()	Returns a Visualizer class object for plotting data samples.
load_checkpoint()	Load last weights from previous run if available.
load_init_weights([model_weights_path])	Load the weights to initialize model.
load_weights(uri, **kwargs)	Load model weights from a file.
log_data_stats()	Log stats about each DataSet.
main()	Main training sequence.
normalize_input(x)	Normalize x to [0, 1].
numpy_predict(x[, raw_out])	Make a prediction using an image or batch of images in numpy format.
on_epoch_end(curr_epoch, metrics)	Hook that is called at end of epoch.
on_overfit_start()	Hook that is called at start of overfit routine.
on_train_start()	Hook that is called at start of train routine.
output_to_numpy(out)	Convert output of model to numpy format.
overfit()	Optimize model using the same batch repeatedly.
plot_dataloader(dl, output_path[, ...])	Plot images and ground truth labels for a DataLoader.
plot_dataloaders([batch_limit, show])	Plot images and ground truth labels for all DataLoaders.
plot_predictions(split[, batch_limit, show])	Plot predictions for a split.
post_forward(x)	Post process output of call to model().
predict(x[, raw_out])	Make prediction for an image or batch of images.
predict_dataloader(dl[, batched_output, ...])	Returns an iterator over predictions on the given dataloader.
predict_dataset(dataset[, return_format, ...])	Returns an iterator over predictions on the given dataset.

continues on next page

Table 2 – continued from previous page

prob_to_pred(x)	Convert a Tensor with prediction probabilities to class ids.
run_tensorboard()	Run TB server serving logged stats.
save_model_bundle()	Save a model bundle.
setup_data()	Set datasets and dataLoaders for train, validation, and test sets.
setup_loss([loss_def_path])	Setup self.loss.
setup_model([model_weights_path, model_def_path])	Setup self.model.
setup_tensorboard()	Setup for logging stats to TB.
setup_training([loss_def_path])	
stop_tensorboard()	Stop TB logging and server if it's running.
sync_from_cloud()	Sync any previous output in the cloud to output_dir.
sync_to_cloud()	Sync any output to the cloud at output_uri.
to_batch(x)	Ensure that image array has batch dimension.
to_device(x, device)	Load Tensors onto a device.
train([epochs])	Training loop that will attempt to resume training if appropriate.
train_end(outputs, num_samples)	Aggregate the ouput of train_step at the end of the epoch.
train_epoch(optimizer[, step_scheduler])	Train for a single epoch.
train_step(batch, batch_ind)	Compute loss for a single training batch.
validate_end(outputs, num_samples)	Aggregate the ouput of validate_step at the end of the epoch.
validate_epoch(dl)	Validate for a single epoch.
validate_step(batch, batch_ind)	Compute metrics on validation batch.

__init__(*cfg: LearnerConfig*, *output_dir: Optional[str] = None*, *train_ds: Optional[Dataset] = None*, *valid_ds: Optional[Dataset] = None*, *test_ds: Optional[Dataset] = None*, *model: Optional[torch.nn.Module] = None*, *loss: Optional[Callable] = None*, *optimizer: Optional[Optimizer] = None*, *epoch_scheduler: Optional[_LRScheduler] = None*, *step_scheduler: Optional[_LRScheduler] = None*, *tmp_dir: Optional[str] = None*, *model_weights_path: Optional[str] = None*, *model_def_path: Optional[str] = None*, *loss_def_path: Optional[str] = None*, *training: bool = True*)

Constructor.

Parameters

- **cfg** (LearnerConfig) – LearnerConfig.
- **train_ds** (*Optional[Dataset], optional*) – The dataset to use for training. If None, will be generated from cfg.data. Defaults to None.
- **valid_ds** (*Optional[Dataset], optional*) – The dataset to use for validation. If None, will be generated from cfg.data. Defaults to None.
- **test_ds** (*Optional[Dataset], optional*) – The dataset to use for testing. If None, will be generated from cfg.data. Defaults to None.
- **model** (*Optional[nn.Module], optional*) – The model. If None, will be generated from cfg.model. Defaults to None.
- **loss** (*Optional[Callable], optional*) – The loss function. If None, will be generated from cfg.solver. Defaults to None.

- **optimizer** (*Optional[Optimizer], optional*) – The optimizer. If None, will be generated from cfg.solver. Defaults to None.
- **epoch_scheduler** (*Optional[_LRScheduler], optional*) – The scheduler that updates after each epoch. If None, will be generated from cfg.solver. Defaults to None.
- **step_scheduler** (*Optional[_LRScheduler], optional*) – The scheduler that updates after each optimizer-step. If None, will be generated from cfg.solver. Defaults to None.
- **tmp_dir** (*Optional[str], optional*) – A temporary directory to use for downloads etc. If None, will be auto-generated. Defaults to None.
- **model_weights_path** (*Optional[str], optional*) – URI of model weights to initialize the model with. Defaults to None.
- **model_def_path** (*Optional[str], optional*) – A local path to a directory with a hubconf.py. If provided, the model definition is imported from here. This is used when loading an external model from a model-bundle. Defaults to None.
- **loss_def_path** (*Optional[str], optional*) – A local path to a directory with a hubconf.py. If provided, the loss function definition is imported from here. This is used when loading an external loss function from a model-bundle. Defaults to None.
- **training** (*bool, optional*) – If False, the training apparatus (loss, optimizer, scheduler, logging, etc.) will not be set up and the model will be put into eval mode. If True, the training apparatus will be set up and the model will be put into training mode. Defaults to True.
- **output_dir** (*Optional[str]*) –

build_dataloaders() → Tuple[torch.utils.data.DataLoader, torch.utils.data.DataLoader, torch.utils.data.DataLoader]

Set the DataLoaders for train, validation, and test sets.

Return type
Tuple[torch.utils.data.DataLoader, torch.utils.data.DataLoader, torch.utils.data.DataLoader]

build_datasets() → Tuple[Dataset, Dataset, Dataset]

Return type
Tuple[Dataset, Dataset, Dataset]

build_epoch_scheduler(*start_epoch: int = 0*) → _LRScheduler

Returns an LR scheduler that changes the LR each epoch.

Parameters
start_epoch (*int*) –

Return type
_LRScheduler

build_loss(*loss_def_path: Optional[str] = None*) → Callable

Build a loss Callable.

Parameters
loss_def_path (*Optional[str]*) –

Return type
Callable

build_metric_names() → List[str]

Returns names of metrics used to validate model at each epoch.

Return type
List[str]

build_model(*model_def_path: Optional[str] = None*) → torch.nn.Module

Build a PyTorch model.

Parameters
model_def_path (*Optional[str]*) –

Return type
torch.nn.Module

build_optimizer() → Optimizer

Returns optimizer.

Return type
Optimizer

build_step_scheduler(*start_epoch: int = 0*) → _LRScheduler

Returns an LR scheduler that changes the LR each step.

Parameters
start_epoch (*int*) –

Return type
_LRScheduler

eval_model(*split: str*)

Evaluate model using a particular dataset split.

Gets validation metrics and saves them along with prediction plots.

Parameters
split (*str*) – the dataset split to use: train, valid, or test.

classmethod from_model_bundle(*model_bundle_uri: str*, *tmp_dir: Optional[str] = None*, *cfg: Optional[LearnerConfig] = None*, *training: bool = False*, ***kwargs*) → *Learner*

Create a Learner from a model bundle.

Note: This is the bundle saved in `train/model-bundle.zip` and not `bundle/model-bundle.zip`.

Parameters

- **model_bundle_uri** (*str*) – URI of the model bundle.
- **tmp_dir** (*Optional[str], optional*) – Optional temporary directory. Will be used for unzipping bundle and also passed to the default constructor. If None, will be auto-generated. Defaults to None.
- **cfg** (*Optional[LearnerConfig], optional*) – If None, will be read from the bundle. Defaults to None.
- **training** (*bool, optional*) – If False, the training apparatus (loss, optimizer, scheduler, logging, etc.) will not be set up and the model will be put into eval mode. If True, the training apparatus will be set up and the model will be put into training mode. Defaults to True.

- ****kwargs** – See *Learner.__init__()*.

Raises
FileNotFoundError – If using custom Albumentations transforms and definition file is not found in bundle.

Returns
Object of the Learner subclass on which this was called.

Return type
Learner

get_collate_fn() → Optional[callable]

Returns a custom collate_fn to use in DataLoader.

None is returned if default collate_fn should be used.

See https://pytorch.org/docs/stable/data.html#working-with-collate-fn

Return type
Optional[callable]

get_dataloader(*split: str*) → torch.utils.data.DataLoader

Get the DataLoader for a split.

Parameters
split (*str*) – a split name which can be train, valid, or test

Return type
torch.utils.data.DataLoader

get_start_epoch() → int

Get start epoch.

If training was interrupted, this returns the last complete epoch + 1.

Return type
int

get_train_sampler(*train_ds: Dataset*) → Optional[Sampler]

Return a sampler to use for the training dataloader or None to not use any.

Parameters
train_ds (*Dataset*) –

Return type
Optional[Sampler]

get_visualizer_class()

Returns a Visualizer class object for plotting data samples.

load_checkpoint()

Load last weights from previous run if available.

load_init_weights(*model_weights_path: Optional[str] = None*) → None

Load the weights to initialize model.

Parameters
model_weights_path (*Optional[str]*) –

Return type
None

load_weights(*uri: str*, ***kwargs*) → None

Load model weights from a file.

Parameters
uri (*str*) –

Return type
None

log_data_stats()

Log stats about each DataSet.

main()

Main training sequence.

This plots the dataset, runs a training and validation loop (which will resume if interrupted), logs stats, plots predictions, and syncs results to the cloud.

normalize_input(*x: ndarray*) → ndarray

Normalize x to [0, 1].

If x.dtype is a subtype of np.unsignedinteger, normalize it to [0, 1] using the max possible value of that dtype. Otherwise, assume it is in [0, 1] already and do nothing.

Parameters
x (*np.ndarray*) – an image or batch of images

Returns
the same array scaled to [0, 1].

Return type
ndarray

numpy_predict(*x: ndarray*, *raw_out: bool = False*) → ndarray

Make a prediction using an image or batch of images in numpy format. If x.dtype is a subtype of np.unsignedinteger, it will be normalized to [0, 1] using the max possible value of that dtype. Otherwise, x will be assumed to be in [0, 1] already and will be cast to torch.float32 directly.

Parameters

- **x** (*ndarray*) – (ndarray) of shape [height, width, channels] or [batch_sz, height, width, channels]
- **raw_out** (*bool*) – if True, return prediction probabilities

Returns
predictions using numpy arrays

Return type
ndarray

on_epoch_end(*curr_epoch*, *metrics*)

Hook that is called at end of epoch.

Writes metrics to CSV and TB, and saves model.

on_overfit_start()

Hook that is called at start of overfit routine.

on_train_start()

Hook that is called at start of train routine.

output_to_numpy(*out: torch.Tensor*) → ndarray

Convert output of model to numpy format.

Parameters
out (*torch.Tensor*) – the output of the model in PyTorch format

Return type
ndarray

Returns: the output of the model in numpy format

overfit()

Optimize model using the same batch repeatedly.

plot_dataloader(*dl: torch.utils.data.DataLoader*, *output_path: str*, *batch_limit: Optional[int] = None*, *show: bool = False*)

Plot images and ground truth labels for a DataLoader.

Parameters

- **dl** (*torch.utils.data.DataLoader*) –
- **output_path** (*str*) –
- **batch_limit** (*Optional[int]*) –
- **show** (*bool*) –

plot_dataloaders(*batch_limit: Optional[int] = None*, *show: bool = False*)

Plot images and ground truth labels for all DataLoaders.

Parameters

- **batch_limit** (*Optional[int]*) –
- **show** (*bool*) –

plot_predictions(*split: str*, *batch_limit: Optional[int] = None*, *show: bool = False*)

Plot predictions for a split.

Uses the first batch for the corresponding DataLoader.

Parameters

- **split** (*str*) – dataset split. Can be train, valid, or test.
- **batch_limit** (*Optional[int]*) – optional limit on (rendered) batch size
- **show** (*bool*) –

post_forward(*x: Any*) → Any

Post process output of call to model().

Useful for when predictions are inside a structure returned by model().

Parameters
x (*Any*) –

Return type
Any

predict(*x: torch.Tensor*, *raw_out: bool = False*) → Any

Make prediction for an image or batch of images.

Parameters

- **x** (*Tensor*) – Image or batch of images as a float Tensor with pixel values normalized to [0, 1].
- **raw_out** (*bool*) – if True, return prediction probabilities

Returns

the predictions, in probability form if raw_out is True, in class_id form
otherwise

Return type
Any

predict_dataloader(*dl: torch.utils.data.DataLoader*, *batched_output: bool = True*, *return_format: Literal['xyz', 'yz', 'z'] = 'z'*, *raw_out: bool = True*, *predict_kw: dict = {}*) → Union[Iterator[Any], Iterator[Tuple[Any, ...]]]

Returns an iterator over predictions on the given dataloader.

Parameters

- **dl** (*DataLoader*) – The dataloader to make predictions on.
- **batched_output** (*bool, optional*) – If True, return batches of x, y, z as defined by the dataloader. If False, unroll the batches into individual items. Defaults to True.
- **return_format** (*Literal['xyz', 'yz', 'z'], optional*) – Format of the return elements of the returned iterator. Must be one of: 'xyz', 'yz', and 'z'. If 'xyz', elements are 3-tuples of x, y, and z. If 'yz', elements are 2-tuples of y and z. If 'z', elements are (non-tuple) values of z. Where x = input image, y = ground truth, and z = prediction. Defaults to 'z'.
- **raw_out** (*bool, optional*) – If true, return raw predicted scores. Defaults to True.
- **predict_kw** (*dict*) – Dict with keywords passed to Learner.predict(). Useful if a Learner subclass implements a custom predict() method.

Raises
ValueError – If return_format is not one of the allowed values.

Returns

If return_format
is 'z', the returned value is an iterator of whatever type the predictions are. Otherwise, the returned value is an iterator of tuples.

Return type
Union[Iterator[Any], Iterator[Tuple[Any, ...]]]

predict_dataset(*dataset: Dataset*, *return_format: Literal['xyz', 'yz', 'z'] = 'z'*, *raw_out: bool = True*, *numpy_out: bool = False*, *predict_kw: dict = {}*, *dataloader_kw: dict = {}*, *progress_bar: bool = True*, *progress_bar_kw: dict = {}*) → Union[Iterator[Any], Iterator[Tuple[Any, ...]]]

Returns an iterator over predictions on the given dataset.

Parameters

- **dataset** (*Dataset*) – The dataset to make predictions on.
- **return_format** (*Literal['xyz', 'yz', 'z'], optional*) – Format of the return elements of the returned iterator. Must be one of: 'xyz', 'yz', and 'z'. If 'xyz', elements are 3-tuples of x, y, and z. If 'yz', elements are 2-tuples of y and z. If 'z', elements are (non-tuple) values of z. Where x = input image, y = ground truth, and z = prediction. Defaults to 'z'.

- **raw_out** (*bool, optional*) – If true, return raw predicted scores. Defaults to True.
- **numpy_out** (*bool, optional*) – If True, convert predictions to numpy arrays before returning. Defaults to False.
- **predict_kw** (*dict*) – Dict with keywords passed to Learner.predict(). Useful if a Learner subclass implements a custom predict() method.
- **dataloader_kw** (*dict*) – Dict with keywords passed to the DataLoader constructor.
- **progress_bar** (*bool, optional*) – If True, display a progress bar. Since this function returns an iterator, the progress bar won't be visible until the iterator is consumed. Defaults to True.
- **progress_bar_kw** (*dict*) – Dict with keywords passed to tqdm.

Raises

ValueError – If return_format is not one of the allowed values.

Returns

If return_format
is 'z', the returned value is an iterator of whatever type the predictions are. Otherwise, the returned value is an iterator of tuples.

Return type
Union[Iterator[Any], Iterator[Tuple[Any, ...]]]

prob_to_pred(*x*)

Convert a Tensor with prediction probabilities to class ids.

The class ids should be the classes with the maximum probability.

run_tensorboard()

Run TB server serving logged stats.

save_model_bundle()

Save a model bundle.

This is a zip file with the model weights in .pth format and a serialized copy of the LearningConfig, which allows for making predictions in the future.

setup_data()

Set datasets and dataLoaders for train, validation, and test sets.

setup_loss(*loss_def_path: Optional[str] = None*) → None

Setup self.loss.

Parameters

- **loss_def_path** (*str, optional*) – Loss definition path. Will be
- **None.** (*available when loading from a bundle. Defaults to*) –

Return type
None

setup_model(*model_weights_path: Optional[str] = None*, *model_def_path: Optional[str] = None*) → None

Setup self.model.

Parameters

- **model_weights_path** (*Optional[str], optional*) – Path to model weights. Will be available when loading from a bundle. Defaults to None.

- **model_def_path** (*Optional[str], optional*) – Path to model definition. Will be available when loading from a bundle. Defaults to None.

Return type
None

setup_tensorboard()

Setup for logging stats to TB.

setup_training(*loss_def_path: Optional[str] = None*) → None

Parameters
loss_def_path (*Optional[str]*) –

Return type
None

stop_tensorboard()

Stop TB logging and server if it's running.

sync_from_cloud()

Sync any previous output in the cloud to output_dir.

sync_to_cloud()

Sync any output to the cloud at output_uri.

to_batch(*x: torch.Tensor*) → torch.Tensor

Ensure that image array has batch dimension.

Parameters
x (*torch.Tensor*) – assumed to be either image or batch of images

Returns
x with extra batch dimension of length 1 if needed

Return type
torch.Tensor

to_device(*x: Any*, *device: str*) → Any

Load Tensors onto a device.

Parameters

- **x** (*Any*) – some object with Tensors in it
- **device** (*str*) – 'cpu' or 'cuda'

Returns
x but with any Tensors in it on the device

Return type
Any

train(*epochs: Optional[int] = None*)

Training loop that will attempt to resume training if appropriate.

Parameters
epochs (*Optional[int]*) –

train_end(*outputs: List[Dict[str, float]]*, *num_samples: int*) → Dict[str, float]

Aggregate the ouput of train_step at the end of the epoch.

Parameters

- **outputs** (*List[Dict[str, float]]*) – a list of outputs of train_step
- **num_samples** (*int*) – total number of training samples processed in epoch

Return type
Dict[str, float]

train_epoch(*optimizer: Optimizer*, *step_scheduler: Optional[_LRScheduler] = None*) → Dict[str, float]

Train for a single epoch.

Parameters

- **optimizer** (*Optimizer*) –
- **step_scheduler** (*Optional[_LRScheduler]*) –

Return type
Dict[str, float]

train_step(*batch*, *batch_ind*)

Compute loss for a single training batch.

Parameters

- **batch** – batch data needed to compute loss
- **batch_ind** – index of batch within epoch

Returns
dict with 'train_loss' as key and possibly other losses

validate_end(*outputs*, *num_samples*)

Aggregate the ouput of validate_step at the end of the epoch.

Parameters

- **outputs** – a list of outputs of validate_step
- **num_samples** – total number of validation samples processed in epoch

validate_epoch(*dl: torch.utils.data.DataLoader*) → Dict[str, float]

Validate for a single epoch.

Parameters
dl (*torch.utils.data.DataLoader*) –

Return type
Dict[str, float]

validate_step(*batch*, *batch_ind*)

Compute metrics on validation batch.

Parameters

- **batch** – batch data needed to compute validation metrics
- **batch_ind** – index of batch within epoch

Returns
dict with metric names mapped to metric values

9.3.2 classification_learner_config

Classes

ClassificationDataFormat	An enumeration.

ClassificationDataFormat

class ClassificationDataFormat

Bases: Enum

An enumeration.

Attributes

image_folder

__init__()

image_folder = 'image_folder'

Configs

ClassificationDataConfig

ClassificationGeoDataConfig	Configure classification *GeoDatasets*.
ClassificationImageDataConfig	Configure *ClassificationImageDatasets*.
ClassificationLearnerConfig	Configure a *ClassificationLearner*.
ClassificationModelConfig	Configure a classification model.

ClassificationDataConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model ClassificationDataConfig

```
{
   "title": "ClassificationDataConfig",
   "description": "Base class that can be extended to provide custom configurations.
↪\n\nThis adds some extra methods to Pydantic BaseModel.\nSee https://pydantic-
↪docs.helpmanual.io/\n\nThe general idea is that configuration schemas can be␣
↪defined by\nsubclassing this and adding class attributes with types and\ndefault␣
↪values for each field. Configs can be defined hierarchically,\nie. a Config can␣
↪have fields which are of type Config.\nValidation, serialization, deserialization,
↪ and IDE support is\nprovided automatically based on this schema.",
```

(continues on next page)

(continued from previous page)


```
   "type": "object",
   "properties": {
      "type_hint": {
         "title": "Type Hint",
         "default": "classification_data",
         "enum": [
            "classification_data"
         ],
         "type": "string"
      }
   },
   "additionalProperties": false
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *type_hint (Literal['classification_data'])*

field type_hint: Literal['classification_data'] = 'classification_data'

build()

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(**args*, ***kwargs*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

ClassificationGeoDataConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model ClassificationGeoDataConfig

Configure classification *GeoDatasets*.

See *rastervision.pytorch_learner.dataset.classification_dataset*.


```
{
   "title": "ClassificationGeoDataConfig",
   "description": "Configure classification :class:`GeoDatasets <.GeoDataset>`.\n\
↪nSee :mod:`rastervision.pytorch_learner.dataset.classification_dataset`.",
   "type": "object",
   "properties": {
      "class_names": {
         "title": "Class Names",
         "description": "Names of classes.",
         "default": [],
         "type": "array",
         "items": {
            "type": "string"
         }
      },
      "class_colors": {
         "title": "Class Colors",
         "description": "Colors used to display classes. Can be color 3-tuples in␣
↪list form.",
         "type": "array",
         "items": {
            "anyOf": [
               {
                  "type": "string"
               },
               {
                  "type": "array",
                  "minItems": 3,
                  "maxItems": 3,
                  "items": [
                     {
                        "type": "integer"
```


(continues on next page)

(continued from previous page)

```
                    },
                    {
                        "type": "integer"
                    },
                    {
                        "type": "integer"
                    }
                ]
            }
        ]
    }
},
"img_channels": {
    "title": "Img Channels",
    "description": "The number of channels of the training images.",
    "exclusiveMinimum": 0,
    "type": "integer"
},
"img_sz": {
    "title": "Img Sz",
    "description": "Length of a side of each image in pixels. This is the size␣
↪to transform it to during training, not the size in the raw dataset.",
    "default": 256,
    "exclusiveMinimum": 0,
    "type": "integer"
},
"train_sz": {
    "title": "Train Sz",
    "description": "If set, the number of training images to use. If fewer␣
↪images exist, then an exception will be raised.",
    "type": "integer"
},
"train_sz_rel": {
    "title": "Train Sz Rel",
    "description": "If set, the proportion of training images to use.",
    "type": "number"
},
"num_workers": {
    "title": "Num Workers",
    "description": "Number of workers to use when DataLoader makes batches.",
    "default": 4,
    "type": "integer"
},
"augmentors": {
    "title": "Augmentors",
    "description": "Names of albumentations augmentors to use for training␣
↪batches. Choices include: ['Blur', 'RandomRotate90', 'HorizontalFlip',
↪'VerticalFlip', 'GaussianBlur', 'GaussNoise', 'RGBShift', 'ToGray'].␣
↪Alternatively, a custom transform can be provided via the aug_transform option.",
    "default": [
        "RandomRotate90",
        "HorizontalFlip",
```

(continues on next page)

(continued from previous page)


```
            "VerticalFlip"
         ],
         "type": "array",
         "items": {
            "type": "string"
         }
      },
      "base_transform": {
         "title": "Base Transform",
         "description": "An Albumentations transform serialized as a dict that will␣
↪be applied to all datasets: training, validation, and test. This transformation␣
↪is in addition to the resizing due to img_sz. This is useful for, for example,␣
↪applying the same normalization to all datasets.",
         "type": "object"
      },
      "aug_transform": {
         "title": "Aug Transform",
         "description": "An Albumentations transform serialized as a dict that will␣
↪be applied as data augmentation to the training dataset. This transform is␣
↪applied before base_transform. If provided, the augmentors option is ignored.",
         "type": "object"
      },
      "plot_options": {
         "title": "Plot Options",
         "description": "Options to control plotting.",
         "default": {
            "transform": {
               "__version__": "1.3.0",
               "transform": {
                  "__class_fullname__": "rastervision.pytorch_learner.utils.utils.
↪MinMaxNormalize",
                  "always_apply": false,
                  "p": 1.0,
                  "min_val": 0.0,
                  "max_val": 1.0,
                  "dtype": 5
               }
            },
            "channel_display_groups": null,
            "type_hint": "plot_options"
         },
         "allOf": [
            {
               "$ref": "#/definitions/PlotOptions"
            }
         ]
      },
      "preview_batch_limit": {
         "title": "Preview Batch Limit",
         "description": "Optional limit on the number of items in the preview plots␣
↪produced during training.",
         "type": "integer"
```


(continues on next page)

(continued from previous page)


```
      },
      "type_hint": {
         "title": "Type Hint",
         "default": "classification_geo_data",
         "enum": [
            "classification_geo_data"
         ],
         "type": "string"
      },
      "scene_dataset": {
         "$ref": "#/definitions/DatasetConfig"
      },
      "window_opts": {
         "title": "Window Opts",
         "default": {},
         "anyOf": [
            {
               "$ref": "#/definitions/GeoDataWindowConfig"
            },
            {
               "type": "object",
               "additionalProperties": {
                  "$ref": "#/definitions/GeoDataWindowConfig"
               }
            }
         ]
      }
   },
   "additionalProperties": false,
   "definitions": {
      "PlotOptions": {
         "title": "PlotOptions",
         "description": "Config related to plotting.",
         "type": "object",
         "properties": {
            "transform": {
               "title": "Transform",
               "description": "An Albumentations transform serialized as a dict␣
↪that will be applied to each image before it is plotted. Mainly useful for␣
↪undoing any data transformation that you do not want included in the plot, such␣
↪as normalization. The default value will shift and scale the image so the values␣
↪range from 0.0 to 1.0 which is the expected range for the plotting function. This␣
↪default is useful for cases where the values after normalization are close to␣
↪zero which makes the plot difficult to see.",
               "default": {
                  "__version__": "1.3.0",
                  "transform": {
                     "__class_fullname__": "rastervision.pytorch_learner.utils.
↪utils.MinMaxNormalize",
                     "always_apply": false,
                     "p": 1.0,
                     "min_val": 0.0,
```


(continues on next page)

(continued from previous page)


```
                    "max_val": 1.0,
                    "dtype": 5
                }
            },
            "type": "object"
        },
        "channel_display_groups": {
            "title": "Channel Display Groups",
            "description": "Groups of image channels to display together as a␣
↪subplot when plotting the data and predictions. Can be a list or tuple of groups␣
↪(e.g. [(0, 1, 2), (3,)]) or a dict containing title-to-group mappings (e.g. {\
↪"RGB\": [0, 1, 2], \"IR\": [3]}), where each group is a list or tuple of channel␣
↪indices and title is a string that will be used as the title of the subplot for␣
↪that group.",
            "anyOf": [
                {
                    "type": "object",
                    "additionalProperties": {
                        "type": "array",
                        "items": {
                            "type": "integer",
                            "minimum": 0
                        }
                    }
                },
                {
                    "type": "array",
                    "items": {
                        "type": "array",
                        "items": {
                            "type": "integer",
                            "minimum": 0
                        }
                    }
                }
            ]
        },
        "type_hint": {
            "title": "Type Hint",
            "default": "plot_options",
            "enum": [
                "plot_options"
            ],
            "type": "string"
        }
    },
    "additionalProperties": false
},
"ClassConfig": {
    "title": "ClassConfig",
    "description": "Configure class information for a machine learning task.",
    "type": "object",
```


(continues on next page)

(continued from previous page)

```
        "properties": {
           "names": {
              "title": "Names",
              "description": "Names of classes. The i-th class in this list will␣
↪have class ID = i.",
              "type": "array",
              "items": {
                 "type": "string"
              }
           },
           "colors": {
              "title": "Colors",
              "description": "Colors used to visualize classes. Can be color␣
↪strings accepted by matplotlib or RGB tuples. If None, a random color will be␣
↪auto-generated for each class.",
              "type": "array",
              "items": {
                 "anyOf": [
                    {
                       "type": "string"
                    },
                    {
                       "type": "array",
                       "items": {}
                    }
                 ]
              }
           },
           "null_class": {
              "title": "Null Class",
              "description": "Optional name of class in `names` to use as the null␣
↪class. This is used in semantic segmentation to represent the label for imagery␣
↪pixels that are NODATA or that are missing a label. If None and the class names␣
↪include \"null\", it will automatically be used as the null class. If None, and␣
↪this Config is part of a SemanticSegmentationConfig, a null class will be added␣
↪automatically.",
              "type": "string"
           },
           "type_hint": {
              "title": "Type Hint",
              "default": "class_config",
              "enum": [
                 "class_config"
              ],
              "type": "string"
           }
        },
        "required": [
           "names"
        ],
        "additionalProperties": false
     },
```

(continues on next page)

(continued from previous page)

```
      "RasterTransformerConfig": {
         "title": "RasterTransformerConfig",
         "description": "Configure a :class:`.RasterTransformer`.",
         "type": "object",
         "properties": {
            "type_hint": {
               "title": "Type Hint",
               "default": "raster_transformer",
               "enum": [
                  "raster_transformer"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      },
      "RasterSourceConfig": {
         "title": "RasterSourceConfig",
         "description": "Configure a :class:`.RasterSource`.",
         "type": "object",
         "properties": {
            "channel_order": {
               "title": "Channel Order",
               "description": "The sequence of channel indices to use when reading␣
↪imagery.",
               "type": "array",
               "items": {
                  "type": "integer"
               }
            },
            "transformers": {
               "title": "Transformers",
               "default": [],
               "type": "array",
               "items": {
                  "$ref": "#/definitions/RasterTransformerConfig"
               }
            },
            "extent": {
               "title": "Extent",
               "description": "Use-specified extent in pixel coords in the form␣
↪(ymin, xmin, ymax, xmax). Useful for cropping the raster source so that only part␣
↪of the raster is read from.",
               "type": "array",
               "minItems": 4,
               "maxItems": 4,
               "items": [
                  {
                     "type": "integer"
                  },
                  {
                     "type": "integer"
```

(continues on next page)

(continued from previous page)


```
                },
                {
                    "type": "integer"
                },
                {
                    "type": "integer"
                }
            ]
        },
        "type_hint": {
            "title": "Type Hint",
            "default": "raster_source",
            "enum": [
                "raster_source"
            ],
            "type": "string"
        }
    },
    "additionalProperties": false
},
"LabelSourceConfig": {
    "title": "LabelSourceConfig",
    "description": "Configure a :class:`.LabelSource`.",
    "type": "object",
    "properties": {
        "type_hint": {
            "title": "Type Hint",
            "default": "label_source",
            "enum": [
                "label_source"
            ],
            "type": "string"
        }
    },
    "additionalProperties": false
},
"LabelStoreConfig": {
    "title": "LabelStoreConfig",
    "description": "Configure a :class:`.LabelStore`.",
    "type": "object",
    "properties": {
        "type_hint": {
            "title": "Type Hint",
            "default": "label_store",
            "enum": [
                "label_store"
            ],
            "type": "string"
        }
    },
    "additionalProperties": false
},
```


(continues on next page)

(continued from previous page)

```
      "SceneConfig": {
         "title": "SceneConfig",
         "description": "Configure a :class:`.Scene` comprising raster data &␣
↪labels for an AOI.\n    ",
         "type": "object",
         "properties": {
            "id": {
               "title": "Id",
               "type": "string"
            },
            "raster_source": {
               "$ref": "#/definitions/RasterSourceConfig"
            },
            "label_source": {
               "$ref": "#/definitions/LabelSourceConfig"
            },
            "label_store": {
               "$ref": "#/definitions/LabelStoreConfig"
            },
            "aoi_uris": {
               "title": "Aoi Uris",
               "description": "List of URIs of GeoJSON files that define the AOIs␣
↪for the scene. Each polygon defines an AOI which is a piece of the scene that is␣
↪assumed to be fully labeled and usable for training or validation. The AOIs are␣
↪assumed to be in EPSG:4326 coordinates.",
               "type": "array",
               "items": {
                  "type": "string"
               }
            },
            "type_hint": {
               "title": "Type Hint",
               "default": "scene",
               "enum": [
                  "scene"
               ],
               "type": "string"
            }
         },
         "required": [
            "id",
            "raster_source"
         ],
         "additionalProperties": false
      },
      "DatasetConfig": {
         "title": "DatasetConfig",
         "description": "Configure train, validation, and test splits for a dataset.
↪",
         "type": "object",
         "properties": {
            "class_config": {
```

(continues on next page)

(continued from previous page)

```
                "$ref": "#/definitions/ClassConfig"
            },
            "train_scenes": {
                "title": "Train Scenes",
                "type": "array",
                "items": {
                    "$ref": "#/definitions/SceneConfig"
                }
            },
            "validation_scenes": {
                "title": "Validation Scenes",
                "type": "array",
                "items": {
                    "$ref": "#/definitions/SceneConfig"
                }
            },
            "test_scenes": {
                "title": "Test Scenes",
                "default": [],
                "type": "array",
                "items": {
                    "$ref": "#/definitions/SceneConfig"
                }
            },
            "scene_groups": {
                "title": "Scene Groups",
                "description": "Groupings of scenes. Should be a dict of the form: {
↪<group-name>: Set(scene_id_1, scene_id_2, ...)}. Three groups are added by␣
↪default: \"train_scenes\", \"validation_scenes\", and \"test_scenes\"",
                "default": {},
                "type": "object",
                "additionalProperties": {
                    "type": "array",
                    "items": {
                        "type": "string"
                    },
                    "uniqueItems": true
                }
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "dataset",
                "enum": [
                    "dataset"
                ],
                "type": "string"
            }
        },
        "required": [
            "class_config",
            "train_scenes",
            "validation_scenes"
```

(continues on next page)

(continued from previous page)


```
        ],
        "additionalProperties": false
     },
     "GeoDataWindowMethod": {
        "title": "GeoDataWindowMethod",
        "description": "An enumeration.",
        "enum": [
           "sliding",
           "random"
        ]
     },
     "GeoDataWindowConfig": {
        "title": "GeoDataWindowConfig",
        "description": "Configure a :class:`.GeoDataset`.\n\nSee :mod:
↪`rastervision.pytorch_learner.dataset.dataset`.",
        "type": "object",
        "properties": {
           "method": {
              "default": "sliding",
              "allOf": [
                 {
                    "$ref": "#/definitions/GeoDataWindowMethod"
                 }
              ]
           },
           "size": {
              "title": "Size",
              "description": "If method = sliding, this is the size of sliding␣
↪window. If method = random, this is the size that all the windows are resized to␣
↪before they are returned. If method = random and neither size_lims nor h_lims and␣
↪w_lims have been specified, then size_lims is set to (size, size + 1).",
              "anyOf": [
                 {
                    "type": "integer",
                    "exclusiveMinimum": 0
                 },
                 {
                    "type": "array",
                    "minItems": 2,
                    "maxItems": 2,
                    "items": [
                       {
                          "type": "integer",
                          "exclusiveMinimum": 0
                       },
                       {
                          "type": "integer",
                          "exclusiveMinimum": 0
                       }
                    ]
                 }
              ]
```


(continues on next page)

(continued from previous page)

```
            },
            "stride": {
               "title": "Stride",
               "description": "Stride of sliding window. Only used if method =␣
↪sliding.",
               "anyOf": [
                  {
                     "type": "integer",
                     "exclusiveMinimum": 0
                  },
                  {
                     "type": "array",
                     "minItems": 2,
                     "maxItems": 2,
                     "items": [
                        {
                           "type": "integer",
                           "exclusiveMinimum": 0
                        },
                        {
                           "type": "integer",
                           "exclusiveMinimum": 0
                        }
                     ]
                  }
               ]
            },
            "padding": {
               "title": "Padding",
               "description": "How many pixels are windows allowed to overflow the␣
↪edges of the raster source.",
               "anyOf": [
                  {
                     "type": "integer",
                     "minimum": 0
                  },
                  {
                     "type": "array",
                     "minItems": 2,
                     "maxItems": 2,
                     "items": [
                        {
                           "type": "integer",
                           "minimum": 0
                        },
                        {
                           "type": "integer",
                           "minimum": 0
                        }
                     ]
                  }
               ]
```

(continues on next page)

(continued from previous page)

```
            },
            "pad_direction": {
               "title": "Pad Direction",
               "description": "If \"end\", only pad ymax and xmax (bottom and␣
↪right). If \"start\", only pad ymin and xmin (top and left). If \"both\", pad all␣
↪sides. Has no effect if paddiong is zero. Defaults to \"end\".",
               "default": "end",
               "enum": [
                  "both",
                  "start",
                  "end"
               ],
               "type": "string"
            },
            "size_lims": {
               "title": "Size Lims",
               "description": "[min, max) interval from which window sizes will be␣
↪uniformly randomly sampled. The upper limit is exclusive. To fix the size to a␣
↪constant value, use size_lims = (sz, sz + 1). Only used if method = random.␣
↪Specify either size_lims or h_lims and w_lims, but not both. If neither size_lims␣
↪nor h_lims and w_lims have been specified, then this will be set to (size, size +␣
↪1).",
               "type": "array",
               "minItems": 2,
               "maxItems": 2,
               "items": [
                  {
                     "type": "integer",
                     "exclusiveMinimum": 0
                  },
                  {
                     "type": "integer",
                     "exclusiveMinimum": 0
                  }
               ]
            },
            "h_lims": {
               "title": "H Lims",
               "description": "[min, max] interval from which window heights will␣
↪be uniformly randomly sampled. Only used if method = random.",
               "type": "array",
               "minItems": 2,
               "maxItems": 2,
               "items": [
                  {
                     "type": "integer",
                     "exclusiveMinimum": 0
                  },
                  {
                     "type": "integer",
                     "exclusiveMinimum": 0
                  }
```

(continues on next page)

(continued from previous page)

```
            ]
         },
         "w_lims": {
            "title": "W Lims",
            "description": "[min, max] interval from which window widths will be␣
↪uniformly randomly sampled. Only used if method = random.",
            "type": "array",
            "minItems": 2,
            "maxItems": 2,
            "items": [
               {
                  "type": "integer",
                  "exclusiveMinimum": 0
               },
               {
                  "type": "integer",
                  "exclusiveMinimum": 0
               }
            ]
         },
         "max_windows": {
            "title": "Max Windows",
            "description": "Max allowed reads from a GeoDataset. Only used if␣
↪method = random.",
            "default": 10000,
            "minimum": 0,
            "type": "integer"
         },
         "max_sample_attempts": {
            "title": "Max Sample Attempts",
            "description": "Max attempts when trying to find a window within the␣
↪AOI of a scene. Only used if method = random and the scene has aoi_polygons␣
↪specified.",
            "default": 100,
            "exclusiveMinimum": 0,
            "type": "integer"
         },
         "efficient_aoi_sampling": {
            "title": "Efficient Aoi Sampling",
            "description": "If the scene has AOIs, sampling windows at random␣
↪anywhere in the extent and then checking if they fall within any of the AOIs can␣
↪be very inefficient. This flag enables the use of an alternate algorithm that␣
↪only samples window locations inside the AOIs. Only used if method = random and␣
↪the scene has aoi_polygons specified. Defaults to True",
            "default": true,
            "type": "boolean"
         },
         "type_hint": {
            "title": "Type Hint",
            "default": "geo_data_window",
            "enum": [
               "geo_data_window"
```

(continues on next page)

(continued from previous page)

```
                ],
                "type": "string"
            }
        },
        "required": [
            "size"
        ],
        "additionalProperties": false
    }
  }
}
```

Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *aug_transform (Optional[dict])*
- *augmentors (List[str])*
- *base_transform (Optional[dict])*
- *class_colors (Optional[List[Union[str, Tuple[int, int, int]]]])*
- *class_names (List[str])*
- *img_channels (Optional[pydantic.types.PositiveInt])*
- *img_sz (pydantic.types.PositiveInt)*
- *num_workers (int)*
- *plot_options (Optional[rastervision.pytorch_learner.learner_config.PlotOptions])*
- *preview_batch_limit (Optional[int])*
- *scene_dataset (Optional[rastervision.core.data.dataset_config.DatasetConfig])*
- *train_sz (Optional[int])*
- *train_sz_rel (Optional[float])*
- *type_hint (Literal['classification_geo_data'])*
- *window_opts (Union[rastervision.pytorch_learner.learner_config.GeoDataWindowConfig, Dict[str, rastervision.pytorch_learner.learner_config.GeoDataWindowConfig]])*

Validators

- `ensure_class_colors` » `all fields`
- `get_class_info_from_class_config_if_needed` » `all fields`
- *validate_albumentation_transform* » *aug_transform*
- *validate_albumentation_transform* » *base_transform*

- validate_augmentors » *augmentors*
- validate_plot_options » all fields
- validate_window_opts » *window_opts*

field aug_transform: Optional[dict] = None

An Albumentations transform serialized as a dict that will be applied as data augmentation to the training dataset. This transform is applied before base_transform. If provided, the augmentors option is ignored.

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- *validate_albumentation_transform*
- validate_plot_options

field augmentors: List[str] = ['RandomRotate90', 'HorizontalFlip', 'VerticalFlip']

Names of albumentations augmentors to use for training batches. Choices include: ['Blur', 'RandomRotate90', 'HorizontalFlip', 'VerticalFlip', 'GaussianBlur', 'GaussNoise', 'RGBShift', 'ToGray']. Alternatively, a custom transform can be provided via the aug_transform option.

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_augmentors
- validate_plot_options

field base_transform: Optional[dict] = None

An Albumentations transform serialized as a dict that will be applied to all datasets: training, validation, and test. This transformation is in addition to the resizing due to img_sz. This is useful for, for example, applying the same normalization to all datasets.

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- *validate_albumentation_transform*
- validate_plot_options

field class_colors: Optional[List[Union[str, RGBTuple]]] = None

Colors used to display classes. Can be color 3-tuples in list form.

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field class_names: List[str] = []

Names of classes.

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field img_channels: Optional[PosInt] = None

The number of channels of the training images.

Constraints

- **exclusiveMinimum** = 0

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field img_sz: PosInt = 256

Length of a side of each image in pixels. This is the size to transform it to during training, not the size in the raw dataset.

Constraints

- **exclusiveMinimum** = 0

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field num_workers: int = 4

Number of workers to use when DataLoader makes batches.

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field plot_options: Optional[*PlotOptions*] = PlotOptions(transform={'__version__': '1.3.0', 'transform': {'__class_fullname__': 'rastervision.pytorch_learner.utils.utils.MinMaxNormalize', 'always_apply': False, 'p': 1.0, 'min_val': 0.0, 'max_val': 1.0, 'dtype': 5}}, channel_display_groups=None)

Options to control plotting.

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field preview_batch_limit: Optional[int] = None

Optional limit on the number of items in the preview plots produced during training.

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field scene_dataset: Optional['SceneDatasetConfig'] = None

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field train_sz: Optional[int] = None

If set, the number of training images to use. If fewer images exist, then an exception will be raised.

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field train_sz_rel: Optional[float] = None

If set, the proportion of training images to use.

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field type_hint: Literal['classification_geo_data'] = 'classification_geo_data'

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field window_opts: Union[*GeoDataWindowConfig*, Dict[str, *GeoDataWindowConfig*]] = {}

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options
- validate_window_opts

build(*tmp_dir: str*, *overfit_mode: bool = False*, *test_mode: bool = False*) → Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

Parameters

- **tmp_dir** (*str*) –
- **overfit_mode** (*bool*) –
- **test_mode** (*bool*) –

Return type
Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

build_scenes(*tmp_dir: str*)

Build training, validation, and test scenes.

Parameters
tmp_dir (*str*) –

validator ensure_class_colors » *all fields*

Parameters
values (*dict*) –

Return type
dict

get_bbox_params() → Optional[BboxParams]

Returns BboxParams used by albumentations for data augmentation.

Return type
Optional[*BboxParams*]

validator get_class_info_from_class_config_if_needed » *all fields*

Parameters
values (*dict*) –

Return type
dict

get_custom_albumentations_transforms() → List[dict]

Returns all custom transforms found in this config.

This should return all serialized albumentations transforms with a 'lambda_transforms_path' field contained in this config or in any of its members no matter how deeply neseted.

The pupose is to make it easier to adjust their paths all at once while saving to or loading from a bundle.

Return type
List[dict]

get_data_transforms() → Tuple[BasicTransform, BasicTransform]

Get albumentations transform objects for data augmentation.

Returns
a transform that doesn't do any data augmentation 2nd tuple arg: a transform with data augmentation

Return type
1st tuple arg

make_datasets(*tmp_dir: str*, *train_tf: Optional[BasicTransform] = None*, *val_tf: Optional[BasicTransform] = None*, *test_tf: Optional[BasicTransform] = None*, ***kwargs*) → Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

Make training, validation, and test datasets.

Parameters

- **tmp_dir** (*str*) – Temporary directory to be used for building scenes.
- **train_tf** (*Optional[A.BasicTransform], optional*) – Transform for the training dataset. Defaults to None.
- **val_tf** (*Optional[A.BasicTransform], optional*) – Transform for the validation dataset. Defaults to None.
- **test_tf** (*Optional[A.BasicTransform], optional*) – Transform for the test dataset. Defaults to None.
- **kwargs** – Kwargs to pass to self.scene_to_dataset()

Returns

PyTorch-compatiable training,
validation, and test datasets.

Return type
Tuple[Dataset, Dataset, Dataset]

random_subset_dataset(*ds: torch.utils.data.Dataset*, *size: Optional[int] = None*, *fraction: Optional[ConstrainedFloatValue] = None*) → torch.utils.data.Subset

Parameters

- **ds** (*torch.utils.data.Dataset*) –
- **size** (*Optional[int]*) –
- **fraction** (*Optional[ConstrainedFloatValue]*) –

Return type
torch.utils.data.Subset

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

scene_to_dataset(*scene:* Scene, *transform: Optional[BasicTransform] = None*) → Union[*ClassificationSlidingWindowGeoDataset*, *ClassificationRandomWindowGeoDataset*]

Make a dataset from a single scene.

Parameters

- **scene** (Scene) –
- **transform** (*Optional[BasicTransform]*) –

Return type
Union[ClassificationSlidingWindowGeoDataset, ClassificationRandomWindowGeoDataset]

update(**args*, ***kwargs*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validator validate_augmentors » augmentors

Parameters
v (*str*) –

Return type
str

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises
ConfigError – if field is invalid

validator validate_plot_options » *all fields*

Parameters
values (*dict*) –

Return type
dict

validator validate_window_opts » window_opts

Parameters

- **v** (*Union[GeoDataWindowConfig, Dict[str, GeoDataWindowConfig]]*) –
- **values** (*dict*) –

Return type
Union[GeoDataWindowConfig, *Dict*[str, GeoDataWindowConfig]]

property num_classes

ClassificationImageDataConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model ClassificationImageDataConfig

Configure *ClassificationImageDatasets*.


```
{
   "title": "ClassificationImageDataConfig",
   "description": "Configure :class:`ClassificationImageDatasets <.
↪ClassificationImageDataset>`.",
   "type": "object",
   "properties": {
      "class_names": {
         "title": "Class Names",
         "description": "Names of classes.",
         "default": [],
         "type": "array",
         "items": {
            "type": "string"
         }
      },
      "class_colors": {
         "title": "Class Colors",
         "description": "Colors used to display classes. Can be color 3-tuples in␣
↪list form.",
         "type": "array",
         "items": {
            "anyOf": [
               {
                  "type": "string"
               },
               {
                  "type": "array",
                  "minItems": 3,
                  "maxItems": 3,
                  "items": [
                     {
                        "type": "integer"
                     },
                     {
                        "type": "integer"
                     },
                     {
                        "type": "integer"
                     }
                  ]
               }
            ]
         }
      },
      "img_channels": {
```


(continues on next page)

(continued from previous page)

```
         "title": "Img Channels",
         "description": "The number of channels of the training images.",
         "exclusiveMinimum": 0,
         "type": "integer"
      },
      "img_sz": {
         "title": "Img Sz",
         "description": "Length of a side of each image in pixels. This is the size␣
↪to transform it to during training, not the size in the raw dataset.",
         "default": 256,
         "exclusiveMinimum": 0,
         "type": "integer"
      },
      "train_sz": {
         "title": "Train Sz",
         "description": "If set, the number of training images to use. If fewer␣
↪images exist, then an exception will be raised.",
         "type": "integer"
      },
      "train_sz_rel": {
         "title": "Train Sz Rel",
         "description": "If set, the proportion of training images to use.",
         "type": "number"
      },
      "num_workers": {
         "title": "Num Workers",
         "description": "Number of workers to use when DataLoader makes batches.",
         "default": 4,
         "type": "integer"
      },
      "augmentors": {
         "title": "Augmentors",
         "description": "Names of albumentations augmentors to use for training␣
↪batches. Choices include: ['Blur', 'RandomRotate90', 'HorizontalFlip',
↪'VerticalFlip', 'GaussianBlur', 'GaussNoise', 'RGBShift', 'ToGray'].␣
↪Alternatively, a custom transform can be provided via the aug_transform option.",
         "default": [
            "RandomRotate90",
            "HorizontalFlip",
            "VerticalFlip"
         ],
         "type": "array",
         "items": {
            "type": "string"
         }
      },
      "base_transform": {
         "title": "Base Transform",
         "description": "An Albumentations transform serialized as a dict that will␣
↪be applied to all datasets: training, validation, and test. This transformation␣
↪is in addition to the resizing due to img_sz. This is useful for, for example,␣
↪applying the same normalization to all datasets.",
```

(continues on next page)

(continued from previous page)

```
         "type": "object"
      },
      "aug_transform": {
         "title": "Aug Transform",
         "description": "An Albumentations transform serialized as a dict that will␣
↪be applied as data augmentation to the training dataset. This transform is␣
↪applied before base_transform. If provided, the augmentors option is ignored.",
         "type": "object"
      },
      "plot_options": {
         "title": "Plot Options",
         "description": "Options to control plotting.",
         "default": {
            "transform": {
               "__version__": "1.3.0",
               "transform": {
                  "__class_fullname__": "rastervision.pytorch_learner.utils.utils.
↪MinMaxNormalize",
                  "always_apply": false,
                  "p": 1.0,
                  "min_val": 0.0,
                  "max_val": 1.0,
                  "dtype": 5
               }
            },
            "channel_display_groups": null,
            "type_hint": "plot_options"
         },
         "allOf": [
            {
               "$ref": "#/definitions/PlotOptions"
            }
         ]
      },
      "preview_batch_limit": {
         "title": "Preview Batch Limit",
         "description": "Optional limit on the number of items in the preview plots␣
↪produced during training.",
         "type": "integer"
      },
      "type_hint": {
         "title": "Type Hint",
         "default": "classification_image_data",
         "enum": [
            "classification_image_data"
         ],
         "type": "string"
      },
      "data_format": {
         "default": "image_folder",
         "allOf": [
            {
```

(continues on next page)

(continued from previous page)

```
                "$ref": "#/definitions/ClassificationDataFormat"
            }
         ]
      },
      "uri": {
         "title": "Uri",
         "description": "One of the following:\n(1) a URI of a directory containing␣
↪\"train\", \"valid\", and (optinally) \"test\" subdirectories;\n(2) a URI of a␣
↪zip file containing (1);\n(3) a list of (2);\n(4) a URI of a directory containing␣
↪zip files containing (1).",
         "anyOf": [
            {
               "type": "string"
            },
            {
               "type": "array",
               "items": {
                  "type": "string"
               }
            }
         ]
      },
      "group_uris": {
         "title": "Group Uris",
         "description": "This can be set instead of uri in order to specify groups␣
↪of chips. Each element in the list is expected to be an object of the same form␣
↪accepted by the uri field. The purpose of separating chips into groups is to be␣
↪able to use the group_train_sz field.",
         "type": "array",
         "items": {
            "anyOf": [
               {
                  "type": "string"
               },
               {
                  "type": "array",
                  "items": {
                     "type": "string"
                  }
               }
            ]
         }
      },
      "group_train_sz": {
         "title": "Group Train Sz",
         "description": "If group_uris is set, this can be used to specify the␣
↪number of chips to use per group. Only applies to training chips. This can either␣
↪be a single value that will be used for all groups or a list of values (one for␣
↪each group).",
         "anyOf": [
            {
               "type": "integer"
```

(continues on next page)

(continued from previous page)

```
            },
            {
               "type": "array",
               "items": {
                  "type": "integer"
               }
            }
         ]
      },
      "group_train_sz_rel": {
         "title": "Group Train Sz Rel",
         "description": "Relative version of group_train_sz. Must be a float in [0,␣
↪1]. If group_uris is set, this can be used to specify the proportion of the total␣
↪chips in each group to use per group. Only applies to training chips. This can␣
↪either be a single value that will be used for all groups or a list of values␣
↪(one for each group).",
         "anyOf": [
            {
               "type": "number",
               "minimum": 0,
               "maximum": 1
            },
            {
               "type": "array",
               "items": {
                  "type": "number",
                  "minimum": 0,
                  "maximum": 1
               }
            }
         ]
      }
   },
   "additionalProperties": false,
   "definitions": {
      "PlotOptions": {
         "title": "PlotOptions",
         "description": "Config related to plotting.",
         "type": "object",
         "properties": {
            "transform": {
               "title": "Transform",
               "description": "An Albumentations transform serialized as a dict␣
↪that will be applied to each image before it is plotted. Mainly useful for␣
↪undoing any data transformation that you do not want included in the plot, such␣
↪as normalization. The default value will shift and scale the image so the values␣
↪range from 0.0 to 1.0 which is the expected range for the plotting function. This␣
↪default is useful for cases where the values after normalization are close to␣
↪zero which makes the plot difficult to see.",
               "default": {
                  "__version__": "1.3.0",
                  "transform": {
```

(continues on next page)

(continued from previous page)


```
                    "__class_fullname__": "rastervision.pytorch_learner.utils.
↪utils.MinMaxNormalize",
                    "always_apply": false,
                    "p": 1.0,
                    "min_val": 0.0,
                    "max_val": 1.0,
                    "dtype": 5
                 }
              },
              "type": "object"
           },
           "channel_display_groups": {
              "title": "Channel Display Groups",
              "description": "Groups of image channels to display together as a␣
↪subplot when plotting the data and predictions. Can be a list or tuple of groups␣
↪(e.g. [(0, 1, 2), (3,)]) or a dict containing title-to-group mappings (e.g. {\
↪"RGB\": [0, 1, 2], \"IR\": [3]}), where each group is a list or tuple of channel␣
↪indices and title is a string that will be used as the title of the subplot for␣
↪that group.",
              "anyOf": [
                 {
                    "type": "object",
                    "additionalProperties": {
                       "type": "array",
                       "items": {
                          "type": "integer",
                          "minimum": 0
                       }
                    }
                 },
                 {
                    "type": "array",
                    "items": {
                       "type": "array",
                       "items": {
                          "type": "integer",
                          "minimum": 0
                       }
                    }
                 }
              ]
           },
           "type_hint": {
              "title": "Type Hint",
              "default": "plot_options",
              "enum": [
                 "plot_options"
              ],
              "type": "string"
           }
        },
        "additionalProperties": false
```


(continues on next page)

(continued from previous page)

```
        },
        "ClassificationDataFormat": {
            "title": "ClassificationDataFormat",
            "description": "An enumeration.",
            "enum": [
                "image_folder"
            ]
        }
    }
}
```

Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *aug_transform (Optional[dict])*
- *augmentors (List[str])*
- *base_transform (Optional[dict])*
- *class_colors (Optional[List[Union[str, Tuple[int, int, int]]]])*
- *class_names (List[str])*
- *data_format (rastervision.pytorch_learner.classification_learner_config.ClassificationDataFormat)*
- *group_train_sz (Optional[Union[int, List[int]]])*
- *group_train_sz_rel (Optional[Union[rastervision.pytorch_learner.learner_config.ConstrainedFloatValue, List[rastervision.pytorch_learner.learner_config.ConstrainedFloatValue]]])*
- *group_uris (Optional[List[Union[str, List[str]]]])*
- *img_channels (Optional[pydantic.types.PositiveInt])*
- *img_sz (pydantic.types.PositiveInt)*
- *num_workers (int)*
- *plot_options (Optional[rastervision.pytorch_learner.learner_config.PlotOptions])*
- *preview_batch_limit (Optional[int])*
- *train_sz (Optional[int])*
- *train_sz_rel (Optional[float])*
- *type_hint (Literal['classification_image_data'])*
- *uri (Optional[Union[str, List[str]]])*

Validators

- `ensure_class_colors` » `all fields`
- *validate_albumentation_transform* » *aug_transform*

- *validate_albumentation_transform* » *base_transform*
- validate_augmentors » *augmentors*
- validate_group_uris » all fields
- validate_plot_options » all fields

field aug_transform: Optional[dict] = None

An Albumentations transform serialized as a dict that will be applied as data augmentation to the training dataset. This transform is applied before base_transform. If provided, the augmentors option is ignored.

Validated by

- ensure_class_colors
- *validate_albumentation_transform*
- validate_group_uris
- validate_plot_options

field augmentors: List[str] = ['RandomRotate90', 'HorizontalFlip', 'VerticalFlip']

Names of albumentations augmentors to use for training batches. Choices include: ['Blur', 'RandomRotate90', 'HorizontalFlip', 'VerticalFlip', 'GaussianBlur', 'GaussNoise', 'RGBShift', 'ToGray']. Alternatively, a custom transform can be provided via the aug_transform option.

Validated by

- ensure_class_colors
- validate_augmentors
- validate_group_uris
- validate_plot_options

field base_transform: Optional[dict] = None

An Albumentations transform serialized as a dict that will be applied to all datasets: training, validation, and test. This transformation is in addition to the resizing due to img_sz. This is useful for, for example, applying the same normalization to all datasets.

Validated by

- ensure_class_colors
- *validate_albumentation_transform*
- validate_group_uris
- validate_plot_options

field class_colors: Optional[List[Union[str, RGBTuple]]] = None

Colors used to display classes. Can be color 3-tuples in list form.

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field class_names: List[str] = []

Names of classes.

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field data_format: *ClassificationDataFormat* = ClassificationDataFormat.image_folder

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field group_train_sz: Optional[Union[int, List[int]]] = None

If group_uris is set, this can be used to specify the number of chips to use per group. Only applies to training chips. This can either be a single value that will be used for all groups or a list of values (one for each group).

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field group_train_sz_rel: Optional[Union[Proportion, List[Proportion]]] = None

Relative version of group_train_sz. Must be a float in [0, 1]. If group_uris is set, this can be used to specify the proportion of the total chips in each group to use per group. Only applies to training chips. This can either be a single value that will be used for all groups or a list of values (one for each group).

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field group_uris: Optional[List[Union[str, List[str]]]] = None

This can be set instead of uri in order to specify groups of chips. Each element in the list is expected to be an object of the same form accepted by the uri field. The purpose of separating chips into groups is to be able to use the group_train_sz field.

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field img_channels: Optional[PosInt] = None

The number of channels of the training images.

Constraints

- **exclusiveMinimum** = 0

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field img_sz: PosInt = 256

Length of a side of each image in pixels. This is the size to transform it to during training, not the size in the raw dataset.

Constraints

- **exclusiveMinimum** = 0

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field num_workers: int = 4

Number of workers to use when DataLoader makes batches.

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field plot_options: Optional[*PlotOptions*] = PlotOptions(transform={'__version__': '1.3.0', 'transform': {'__class_fullname__': 'rastervision.pytorch_learner.utils.utils.MinMaxNormalize', 'always_apply': False, 'p': 1.0, 'min_val': 0.0, 'max_val': 1.0, 'dtype': 5}}, channel_display_groups=None)

Options to control plotting.

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field preview_batch_limit: Optional[int] = None

Optional limit on the number of items in the preview plots produced during training.

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field train_sz: Optional[int] = None

If set, the number of training images to use. If fewer images exist, then an exception will be raised.

Validated by

- ensure_class_colors

- validate_group_uris
- validate_plot_options

field train_sz_rel: Optional[float] = None

If set, the proportion of training images to use.

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field type_hint: Literal['classification_image_data'] = 'classification_image_data'

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field uri: Optional[Union[str, List[str]]] = None

One of the following: (1) a URI of a directory containing "train", "valid", and (optinally) "test" subdirectories; (2) a URI of a zip file containing (1); (3) a list of (2); (4) a URI of a directory containing zip files containing (1).

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

build(*tmp_dir: str*, *overfit_mode: bool = False*, *test_mode: bool = False*) → Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

Parameters

- **tmp_dir** (*str*) –
- **overfit_mode** (*bool*) –
- **test_mode** (*bool*) –

Return type

Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

dir_to_dataset(*data_dir: str*, *transform: BasicTransform*) → *ClassificationImageDataset*

Parameters

- **data_dir** (*str*) –
- **transform** (*BasicTransform*) –

Return type

ClassificationImageDataset

validator ensure_class_colors » *all fields*

Parameters
values (*dict*) –

Return type
dict

get_bbox_params() → Optional[BboxParams]

Returns BboxParams used by albumentations for data augmentation.

Return type
Optional[*BboxParams*]

get_custom_albumentations_transforms() → List[dict]

Returns all custom transforms found in this config.

This should return all serialized albumentations transforms with a 'lambda_transforms_path' field contained in this config or in any of its members no matter how deeply neseted.

The pupose is to make it easier to adjust their paths all at once while saving to or loading from a bundle.

Return type
List[dict]

get_data_dirs(*uri: Union[str, List[str]]*, *unzip_dir: str*) → List[str]

Extract data dirs from uri.

Data dirs are directories containing "train", "valid", and (optinally) "test" subdirectories.

Parameters

- **uri** (*Union[str, List[str]]*) – a URI or a list of URIs of one of the following:

(1) a URI of a directory containing "train", "valid", and (optinally) "test" subdirectories

(2) a URI of a zip file containing (1)

(3) a list of (2)

(4) a URI of a directory containing zip files containing (1)

- **unzip_dir** (*str*) –

Returns
paths to directories that each contain contents of one zip file

Return type
List[str]

get_data_transforms() → Tuple[BasicTransform, BasicTransform]

Get albumentations transform objects for data augmentation.

Returns
a transform that doesn't do any data augmentation 2nd tuple arg: a transform with data augmentation

Return type
1st tuple arg

get_datasets_from_group_uris(*uris: Union[str, List[str]]*, *tmp_dir: str*, *group_train_sz: Optional[int] = None*, *group_train_sz_rel: Optional[float] = None*, *overfit_mode: bool = False*, *test_mode: bool = False*) → Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

Parameters

- **uris** (*Union[str, List[str]]*) –
- **tmp_dir** (*str*) –
- **group_train_sz** (*Optional[int]*) –
- **group_train_sz_rel** (*Optional[float]*) –
- **overfit_mode** (*bool*) –
- **test_mode** (*bool*) –

Return type
Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

get_datasets_from_uri(*uri: Union[str, List[str]]*, *tmp_dir: str*, *overfit_mode: bool = False*, *test_mode: bool = False*) → Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

Get image train, validation, & test datasets from a single zip file.

Parameters

- **uri** (*Union[str, List[str]]*) – Uri of a zip file containing the images.
- **tmp_dir** (*str*) –
- **overfit_mode** (*bool*) –
- **test_mode** (*bool*) –

Returns

Training, validation, and test
dataSets.

Return type
Tuple[Dataset, Dataset, Dataset]

make_datasets(*train_dirs: Iterable[str]*, *val_dirs: Iterable[str]*, *test_dirs: Iterable[str]*, *train_tf: Optional[BasicTransform] = None*, *val_tf: Optional[BasicTransform] = None*, *test_tf: Optional[BasicTransform] = None*) → Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

Make training, validation, and test datasets.

Parameters

- **train_dirs** (*str*) – Directories where training data is located.
- **val_dirs** (*str*) – Directories where validation data is located.
- **test_dirs** (*str*) – Directories where test data is located.
- **train_tf** (*Optional[A.BasicTransform], optional*) – Transform for the training dataset. Defaults to None.
- **val_tf** (*Optional[A.BasicTransform], optional*) – Transform for the validation dataset. Defaults to None.
- **test_tf** (*Optional[A.BasicTransform], optional*) – Transform for the test dataset. Defaults to None.

Returns

PyTorch-compatiable training,
validation, and test datasets.

Return type
Tuple[Dataset, Dataset, Dataset]

random_subset_dataset(*ds: torch.utils.data.Dataset*, *size: Optional[int] = None*, *fraction: Optional[ConstrainedFloatValue] = None*) → torch.utils.data.Subset

Parameters

- **ds** (*torch.utils.data.Dataset*) –
- **size** (*Optional[int]*) –
- **fraction** (*Optional[ConstrainedFloatValue]*) –

Return type
torch.utils.data.Subset

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

unzip_data(*zip_uris: List[str]*, *unzip_dir: str*) → List[str]

Unzip dataset zip files.

Parameters

- **zip_uris** (*List[str]*) – a list of URIs of zip files:
- **unzip_dir** (*str*) – directory where zip files will be extrated to.

Returns
paths to directories that each contain contents of one zip file

Return type
List[str]

update(**args*, ***kwargs*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validator validate_augmentors » augmentors

Parameters
v (*str*) –

Return type
str

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validator validate_group_uris » *all fields*

Parameters
values (*dict*) –

Return type
dict

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises
ConfigError – if field is invalid

validator validate_plot_options » *all fields*

Parameters
values (*dict*) –

Return type
dict

property num_classes

ClassificationLearnerConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model ClassificationLearnerConfig

Configure a *ClassificationLearner*.


```
{
   "title": "ClassificationLearnerConfig",
   "description": "Configure a :class:`.ClassificationLearner`.",
   "type": "object",
   "properties": {
      "model": {
         "$ref": "#/definitions/ClassificationModelConfig"
      },
      "solver": {
         "$ref": "#/definitions/SolverConfig"
      },
      "data": {
         "title": "Data",
         "anyOf": [
```


(continues on next page)

(continued from previous page)

```
            {
               "$ref": "#/definitions/ClassificationImageDataConfig"
            },
            {
               "$ref": "#/definitions/ClassificationGeoDataConfig"
            }
         ]
      },
      "predict_mode": {
         "title": "Predict Mode",
         "description": "If True, skips training, loads model, and does final eval.
↪",
         "default": false,
         "type": "boolean"
      },
      "test_mode": {
         "title": "Test Mode",
         "description": "If True, uses test_num_epochs, test_batch_sz, truncated␣
↪datasets with only a single batch, image_sz that is cut in half, and num_workers␣
↪= 0. This is useful for testing that code runs correctly on CPU without␣
↪multithreading before running full job on GPU.",
         "default": false,
         "type": "boolean"
      },
      "overfit_mode": {
         "title": "Overfit Mode",
         "description": "If True, uses half image size, and instead of doing epoch-
↪based training, optimizes the model using a single batch repeatedly for overfit_
↪num_steps number of steps.",
         "default": false,
         "type": "boolean"
      },
      "eval_train": {
         "title": "Eval Train",
         "description": "If True, runs final evaluation on training set (in␣
↪addition to test set). Useful for debugging.",
         "default": false,
         "type": "boolean"
      },
      "save_model_bundle": {
         "title": "Save Model Bundle",
         "description": "If True, saves a model bundle at the end of training which␣
↪is zip file with model and this LearnerConfig which can be used to make␣
↪predictions on new images at a later time.",
         "default": true,
         "type": "boolean"
      },
      "log_tensorboard": {
         "title": "Log Tensorboard",
         "description": "Save Tensorboard log files at the end of each epoch.",
         "default": true,
         "type": "boolean"
```

(continues on next page)

(continued from previous page)


```
        },
        "run_tensorboard": {
            "title": "Run Tensorboard",
            "description": "run Tensorboard server during training",
            "default": false,
            "type": "boolean"
        },
        "output_uri": {
            "title": "Output Uri",
            "description": "URI of where to save output",
            "type": "string"
        },
        "type_hint": {
            "title": "Type Hint",
            "default": "classification_learner",
            "enum": [
                "classification_learner"
            ],
            "type": "string"
        }
    },
    "required": [
        "solver",
        "data"
    ],
    "additionalProperties": false,
    "definitions": {
        "Backbone": {
            "title": "Backbone",
            "description": "An enumeration.",
            "enum": [
                "alexnet",
                "densenet121",
                "densenet169",
                "densenet201",
                "densenet161",
                "googlenet",
                "inception_v3",
                "mnasnet0_5",
                "mnasnet0_75",
                "mnasnet1_0",
                "mnasnet1_3",
                "mobilenet_v2",
                "resnet18",
                "resnet34",
                "resnet50",
                "resnet101",
                "resnet152",
                "resnext50_32x4d",
                "resnext101_32x8d",
                "wide_resnet50_2",
                "wide_resnet101_2",
```


(continues on next page)

(continued from previous page)


```
            "shufflenet_v2_x0_5",
            "shufflenet_v2_x1_0",
            "shufflenet_v2_x1_5",
            "shufflenet_v2_x2_0",
            "squeezenet1_0",
            "squeezenet1_1",
            "vgg11",
            "vgg11_bn",
            "vgg13",
            "vgg13_bn",
            "vgg16",
            "vgg16_bn",
            "vgg19_bn",
            "vgg19"
         ]
      },
      "ExternalModuleConfig": {
         "title": "ExternalModuleConfig",
         "description": "Config describing an object to be loaded via Torch Hub.",
         "type": "object",
         "properties": {
            "uri": {
               "title": "Uri",
               "description": "Local uri of a zip file, or local uri of a directory,
↪or remote uri of zip file.",
               "minLength": 1,
               "type": "string"
            },
            "github_repo": {
               "title": "Github Repo",
               "description": "<repo-owner>/<repo-name>[:tag]",
               "pattern": ".+/.+",
               "type": "string"
            },
            "name": {
               "title": "Name",
               "description": "Name of the folder in which to extract/copy the␣
↪definition files.",
               "minLength": 1,
               "type": "string"
            },
            "entrypoint": {
               "title": "Entrypoint",
               "description": "Name of a callable present in hubconf.py. See docs␣
↪for torch.hub for details.",
               "minLength": 1,
               "type": "string"
            },
            "entrypoint_args": {
               "title": "Entrypoint Args",
               "description": "Args to pass to the entrypoint. Must be serializable.
↪",
```


(continues on next page)

(continued from previous page)


```
                "default": [],
                "type": "array",
                "items": {}
            },
            "entrypoint_kwargs": {
                "title": "Entrypoint Kwargs",
                "description": "Keyword args to pass to the entrypoint. Must be␣
↪serializable.",
                "default": {},
                "type": "object"
            },
            "force_reload": {
                "title": "Force Reload",
                "description": "Force reload of module definition.",
                "default": false,
                "type": "boolean"
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "external-module",
                "enum": [
                    "external-module"
                ],
                "type": "string"
            }
        },
        "required": [
            "entrypoint"
        ],
        "additionalProperties": false
    },
    "ClassificationModelConfig": {
        "title": "ClassificationModelConfig",
        "description": "Configure a classification model.",
        "type": "object",
        "properties": {
            "backbone": {
                "description": "The torchvision.models backbone to use.",
                "default": "resnet18",
                "allOf": [
                    {
                        "$ref": "#/definitions/Backbone"
                    }
                ]
            },
            "pretrained": {
                "title": "Pretrained",
                "description": "If True, use ImageNet weights. If False, use random␣
↪initialization.",
                "default": true,
                "type": "boolean"
            },
```


(continues on next page)

(continued from previous page)


```
            "init_weights": {
               "title": "Init Weights",
               "description": "URI of PyTorch model weights used to initialize␣
↪model. If set, this supercedes the pretrained option.",
               "type": "string"
            },
            "load_strict": {
               "title": "Load Strict",
               "description": "If True, the keys in the state dict referenced by␣
↪init_weights must match exactly. Setting this to False can be useful if you just␣
↪want to load the backbone of a model.",
               "default": true,
               "type": "boolean"
            },
            "external_def": {
               "title": "External Def",
               "description": "If specified, the model will be built from the␣
↪definition from this external source, using Torch Hub.",
               "allOf": [
                  {
                     "$ref": "#/definitions/ExternalModuleConfig"
                  }
               ]
            },
            "type_hint": {
               "title": "Type Hint",
               "default": "classification_model",
               "enum": [
                  "classification_model"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      },
      "SolverConfig": {
         "title": "SolverConfig",
         "description": "Config related to solver aka optimizer.",
         "type": "object",
         "properties": {
            "lr": {
               "title": "Lr",
               "description": "Learning rate.",
               "default": 0.0001,
               "exclusiveMinimum": 0,
               "type": "number"
            },
            "num_epochs": {
               "title": "Num Epochs",
               "description": "Number of epochs (ie. sweeps through the whole␣
↪training set).",
               "default": 10,
```


(continues on next page)

(continued from previous page)

```
            "exclusiveMinimum": 0,
            "type": "integer"
         },
         "test_num_epochs": {
            "title": "Test Num Epochs",
            "description": "Number of epochs to use in test mode.",
            "default": 2,
            "exclusiveMinimum": 0,
            "type": "integer"
         },
         "test_batch_sz": {
            "title": "Test Batch Sz",
            "description": "Batch size to use in test mode.",
            "default": 4,
            "exclusiveMinimum": 0,
            "type": "integer"
         },
         "overfit_num_steps": {
            "title": "Overfit Num Steps",
            "description": "Number of optimizer steps to use in overfit mode.",
            "default": 1,
            "exclusiveMinimum": 0,
            "type": "integer"
         },
         "sync_interval": {
            "title": "Sync Interval",
            "description": "The interval in epochs for each sync to the cloud.",
            "default": 1,
            "exclusiveMinimum": 0,
            "type": "integer"
         },
         "batch_sz": {
            "title": "Batch Sz",
            "description": "Batch size.",
            "default": 32,
            "exclusiveMinimum": 0,
            "type": "integer"
         },
         "one_cycle": {
            "title": "One Cycle",
            "description": "If True, use triangular LR scheduler with a single␣
↪cycle across all epochs with start and end LR being lr/10 and the peak being lr.",
            "default": true,
            "type": "boolean"
         },
         "multi_stage": {
            "title": "Multi Stage",
            "description": "List of epoch indices at which to divide LR by 10.",
            "default": [],
            "type": "array",
            "items": {}
         },
```

(continues on next page)

(continued from previous page)

```
            "class_loss_weights": {
               "title": "Class Loss Weights",
               "description": "Class weights for weighted loss.",
               "type": "array",
               "items": {
                  "type": "number"
               }
            },
            "ignore_class_index": {
               "title": "Ignore Class Index",
               "description": "If specified, this index is ignored when computing␣
↪the loss. See pytorch documentation for nn.CrossEntropyLoss for more details.␣
↪This can also be negative, in which case it is treated as a negative slice index␣
↪i.e. -1 = last index, -2 = second-last index, and so on.",
               "type": "integer"
            },
            "external_loss_def": {
               "title": "External Loss Def",
               "description": "If specified, the loss will be built from the␣
↪definition from this external source, using Torch Hub.",
               "allOf": [
                  {
                     "$ref": "#/definitions/ExternalModuleConfig"
                  }
               ]
            },
            "type_hint": {
               "title": "Type Hint",
               "default": "solver",
               "enum": [
                  "solver"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      },
      "PlotOptions": {
         "title": "PlotOptions",
         "description": "Config related to plotting.",
         "type": "object",
         "properties": {
            "transform": {
               "title": "Transform",
               "description": "An Albumentations transform serialized as a dict␣
↪that will be applied to each image before it is plotted. Mainly useful for␣
↪undoing any data transformation that you do not want included in the plot, such␣
↪as normalization. The default value will shift and scale the image so the values␣
↪range from 0.0 to 1.0 which is the expected range for the plotting function. This␣
↪default is useful for cases where the values after normalization are close to␣
↪zero which makes the plot difficult to see.",
               "default": {
```

(continues on next page)

(continued from previous page)


```
                "__version__": "1.3.0",
                "transform": {
                   "__class_fullname__": "rastervision.pytorch_learner.utils.
↪utils.MinMaxNormalize",
                   "always_apply": false,
                   "p": 1.0,
                   "min_val": 0.0,
                   "max_val": 1.0,
                   "dtype": 5
                }
             },
             "type": "object"
          },
          "channel_display_groups": {
             "title": "Channel Display Groups",
             "description": "Groups of image channels to display together as a␣
↪subplot when plotting the data and predictions. Can be a list or tuple of groups␣
↪(e.g. [(0, 1, 2), (3,)]) or a dict containing title-to-group mappings (e.g. {\
↪"RGB\": [0, 1, 2], \"IR\": [3]}), where each group is a list or tuple of channel␣
↪indices and title is a string that will be used as the title of the subplot for␣
↪that group.",
             "anyOf": [
                {
                   "type": "object",
                   "additionalProperties": {
                      "type": "array",
                      "items": {
                         "type": "integer",
                         "minimum": 0
                      }
                   }
                },
                {
                   "type": "array",
                   "items": {
                      "type": "array",
                      "items": {
                         "type": "integer",
                         "minimum": 0
                      }
                   }
                }
             ]
          },
          "type_hint": {
             "title": "Type Hint",
             "default": "plot_options",
             "enum": [
                "plot_options"
             ],
             "type": "string"
          }
```


(continues on next page)

(continued from previous page)


```
        },
        "additionalProperties": false
     },
     "ClassificationDataFormat": {
        "title": "ClassificationDataFormat",
        "description": "An enumeration.",
        "enum": [
           "image_folder"
        ]
     },
     "ClassificationImageDataConfig": {
        "title": "ClassificationImageDataConfig",
        "description": "Configure :class:`ClassificationImageDatasets <.
↪ClassificationImageDataset>`.",
        "type": "object",
        "properties": {
           "class_names": {
              "title": "Class Names",
              "description": "Names of classes.",
              "default": [],
              "type": "array",
              "items": {
                 "type": "string"
              }
           },
           "class_colors": {
              "title": "Class Colors",
              "description": "Colors used to display classes. Can be color 3-
↪tuples in list form.",
              "type": "array",
              "items": {
                 "anyOf": [
                    {
                       "type": "string"
                    },
                    {
                       "type": "array",
                       "minItems": 3,
                       "maxItems": 3,
                       "items": [
                          {
                             "type": "integer"
                          },
                          {
                             "type": "integer"
                          },
                          {
                             "type": "integer"
                          }
                       ]
                    }
                 ]
```


(continues on next page)

(continued from previous page)

```
            }
         },
         "img_channels": {
            "title": "Img Channels",
            "description": "The number of channels of the training images.",
            "exclusiveMinimum": 0,
            "type": "integer"
         },
         "img_sz": {
            "title": "Img Sz",
            "description": "Length of a side of each image in pixels. This is␣
↪the size to transform it to during training, not the size in the raw dataset.",
            "default": 256,
            "exclusiveMinimum": 0,
            "type": "integer"
         },
         "train_sz": {
            "title": "Train Sz",
            "description": "If set, the number of training images to use. If␣
↪fewer images exist, then an exception will be raised.",
            "type": "integer"
         },
         "train_sz_rel": {
            "title": "Train Sz Rel",
            "description": "If set, the proportion of training images to use.",
            "type": "number"
         },
         "num_workers": {
            "title": "Num Workers",
            "description": "Number of workers to use when DataLoader makes␣
↪batches.",
            "default": 4,
            "type": "integer"
         },
         "augmentors": {
            "title": "Augmentors",
            "description": "Names of albumentations augmentors to use for␣
↪training batches. Choices include: ['Blur', 'RandomRotate90', 'HorizontalFlip',
↪'VerticalFlip', 'GaussianBlur', 'GaussNoise', 'RGBShift', 'ToGray'].␣
↪Alternatively, a custom transform can be provided via the aug_transform option.",
            "default": [
               "RandomRotate90",
               "HorizontalFlip",
               "VerticalFlip"
            ],
            "type": "array",
            "items": {
               "type": "string"
            }
         },
         "base_transform": {
            "title": "Base Transform",
```

(continues on next page)

(continued from previous page)


```
                "description": "An Albumentations transform serialized as a dict␣
↪that will be applied to all datasets: training, validation, and test. This␣
↪transformation is in addition to the resizing due to img_sz. This is useful for,␣
↪for example, applying the same normalization to all datasets.",
                "type": "object"
            },
            "aug_transform": {
                "title": "Aug Transform",
                "description": "An Albumentations transform serialized as a dict␣
↪that will be applied as data augmentation to the training dataset. This transform␣
↪is applied before base_transform. If provided, the augmentors option is ignored.",
                "type": "object"
            },
            "plot_options": {
                "title": "Plot Options",
                "description": "Options to control plotting.",
                "default": {
                    "transform": {
                        "__version__": "1.3.0",
                        "transform": {
                            "__class_fullname__": "rastervision.pytorch_learner.utils.
↪utils.MinMaxNormalize",
                            "always_apply": false,
                            "p": 1.0,
                            "min_val": 0.0,
                            "max_val": 1.0,
                            "dtype": 5
                        }
                    },
                    "channel_display_groups": null,
                    "type_hint": "plot_options"
                },
                "allOf": [
                    {
                        "$ref": "#/definitions/PlotOptions"
                    }
                ]
            },
            "preview_batch_limit": {
                "title": "Preview Batch Limit",
                "description": "Optional limit on the number of items in the preview␣
↪plots produced during training.",
                "type": "integer"
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "classification_image_data",
                "enum": [
                    "classification_image_data"
                ],
                "type": "string"
            },
```


(continues on next page)

(continued from previous page)

```
            "data_format": {
               "default": "image_folder",
               "allOf": [
                  {
                     "$ref": "#/definitions/ClassificationDataFormat"
                  }
               ]
            },
            "uri": {
               "title": "Uri",
               "description": "One of the following:\n(1) a URI of a directory␣
↪containing \"train\", \"valid\", and (optinally) \"test\" subdirectories;\n(2) a␣
↪URI of a zip file containing (1);\n(3) a list of (2);\n(4) a URI of a directory␣
↪containing zip files containing (1).",
               "anyOf": [
                  {
                     "type": "string"
                  },
                  {
                     "type": "array",
                     "items": {
                        "type": "string"
                     }
                  }
               ]
            },
            "group_uris": {
               "title": "Group Uris",
               "description": "This can be set instead of uri in order to specify␣
↪groups of chips. Each element in the list is expected to be an object of the same␣
↪form accepted by the uri field. The purpose of separating chips into groups is to␣
↪be able to use the group_train_sz field.",
               "type": "array",
               "items": {
                  "anyOf": [
                     {
                        "type": "string"
                     },
                     {
                        "type": "array",
                        "items": {
                           "type": "string"
                        }
                     }
                  ]
               }
            },
            "group_train_sz": {
               "title": "Group Train Sz",
               "description": "If group_uris is set, this can be used to specify␣
↪the number of chips to use per group. Only applies to training chips. This can␣
↪either be a single value that will be used for all groups or a list of values␣
```

(continues on next page)

(continued from previous page)

```
↪(one for each group).",
                "anyOf": [
                   {
                      "type": "integer"
                   },
                   {
                      "type": "array",
                      "items": {
                         "type": "integer"
                      }
                   }
                ]
             },
             "group_train_sz_rel": {
                "title": "Group Train Sz Rel",
                "description": "Relative version of group_train_sz. Must be a float␣
↪in [0, 1]. If group_uris is set, this can be used to specify the proportion of␣
↪the total chips in each group to use per group. Only applies to training chips.␣
↪This can either be a single value that will be used for all groups or a list of␣
↪values (one for each group).",
                "anyOf": [
                   {
                      "type": "number",
                      "minimum": 0,
                      "maximum": 1
                   },
                   {
                      "type": "array",
                      "items": {
                         "type": "number",
                         "minimum": 0,
                         "maximum": 1
                      }
                   }
                ]
             }
          },
          "additionalProperties": false
       },
       "ClassConfig": {
          "title": "ClassConfig",
          "description": "Configure class information for a machine learning task.",
          "type": "object",
          "properties": {
             "names": {
                "title": "Names",
                "description": "Names of classes. The i-th class in this list will␣
↪have class ID = i.",
                "type": "array",
                "items": {
                   "type": "string"
                }
```

(continues on next page)

(continued from previous page)


```
            },
            "colors": {
               "title": "Colors",
               "description": "Colors used to visualize classes. Can be color␣
↪strings accepted by matplotlib or RGB tuples. If None, a random color will be␣
↪auto-generated for each class.",
               "type": "array",
               "items": {
                  "anyOf": [
                     {
                        "type": "string"
                     },
                     {
                        "type": "array",
                        "items": {}
                     }
                  ]
               }
            },
            "null_class": {
               "title": "Null Class",
               "description": "Optional name of class in `names` to use as the null␣
↪class. This is used in semantic segmentation to represent the label for imagery␣
↪pixels that are NODATA or that are missing a label. If None and the class names␣
↪include \"null\", it will automatically be used as the null class. If None, and␣
↪this Config is part of a SemanticSegmentationConfig, a null class will be added␣
↪automatically.",
               "type": "string"
            },
            "type_hint": {
               "title": "Type Hint",
               "default": "class_config",
               "enum": [
                  "class_config"
               ],
               "type": "string"
            }
         },
         "required": [
            "names"
         ],
         "additionalProperties": false
      },
      "RasterTransformerConfig": {
         "title": "RasterTransformerConfig",
         "description": "Configure a :class:`.RasterTransformer`.",
         "type": "object",
         "properties": {
            "type_hint": {
               "title": "Type Hint",
               "default": "raster_transformer",
               "enum": [
```


(continues on next page)

(continued from previous page)

```
                    "raster_transformer"
                ],
                "type": "string"
            }
        },
        "additionalProperties": false
    },
    "RasterSourceConfig": {
        "title": "RasterSourceConfig",
        "description": "Configure a :class:`.RasterSource`.",
        "type": "object",
        "properties": {
            "channel_order": {
                "title": "Channel Order",
                "description": "The sequence of channel indices to use when reading␣
↪imagery.",
                "type": "array",
                "items": {
                    "type": "integer"
                }
            },
            "transformers": {
                "title": "Transformers",
                "default": [],
                "type": "array",
                "items": {
                    "$ref": "#/definitions/RasterTransformerConfig"
                }
            },
            "extent": {
                "title": "Extent",
                "description": "Use-specified extent in pixel coords in the form␣
↪(ymin, xmin, ymax, xmax). Useful for cropping the raster source so that only part␣
↪of the raster is read from.",
                "type": "array",
                "minItems": 4,
                "maxItems": 4,
                "items": [
                    {
                        "type": "integer"
                    },
                    {
                        "type": "integer"
                    },
                    {
                        "type": "integer"
                    },
                    {
                        "type": "integer"
                    }
                ]
            },
```

(continues on next page)

(continued from previous page)


```
            "type_hint": {
               "title": "Type Hint",
               "default": "raster_source",
               "enum": [
                  "raster_source"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      },
      "LabelSourceConfig": {
         "title": "LabelSourceConfig",
         "description": "Configure a :class:`.LabelSource`.",
         "type": "object",
         "properties": {
            "type_hint": {
               "title": "Type Hint",
               "default": "label_source",
               "enum": [
                  "label_source"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      },
      "LabelStoreConfig": {
         "title": "LabelStoreConfig",
         "description": "Configure a :class:`.LabelStore`.",
         "type": "object",
         "properties": {
            "type_hint": {
               "title": "Type Hint",
               "default": "label_store",
               "enum": [
                  "label_store"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      },
      "SceneConfig": {
         "title": "SceneConfig",
         "description": "Configure a :class:`.Scene` comprising raster data &␣
↪labels for an AOI.\n    ",
         "type": "object",
         "properties": {
            "id": {
               "title": "Id",
               "type": "string"
```


(continues on next page)

(continued from previous page)

```
            },
            "raster_source": {
               "$ref": "#/definitions/RasterSourceConfig"
            },
            "label_source": {
               "$ref": "#/definitions/LabelSourceConfig"
            },
            "label_store": {
               "$ref": "#/definitions/LabelStoreConfig"
            },
            "aoi_uris": {
               "title": "Aoi Uris",
               "description": "List of URIs of GeoJSON files that define the AOIs␣
↪for the scene. Each polygon defines an AOI which is a piece of the scene that is␣
↪assumed to be fully labeled and usable for training or validation. The AOIs are␣
↪assumed to be in EPSG:4326 coordinates.",
               "type": "array",
               "items": {
                  "type": "string"
               }
            },
            "type_hint": {
               "title": "Type Hint",
               "default": "scene",
               "enum": [
                  "scene"
               ],
               "type": "string"
            }
         },
         "required": [
            "id",
            "raster_source"
         ],
         "additionalProperties": false
      },
      "DatasetConfig": {
         "title": "DatasetConfig",
         "description": "Configure train, validation, and test splits for a dataset.
↪",
         "type": "object",
         "properties": {
            "class_config": {
               "$ref": "#/definitions/ClassConfig"
            },
            "train_scenes": {
               "title": "Train Scenes",
               "type": "array",
               "items": {
                  "$ref": "#/definitions/SceneConfig"
               }
            },
```

(continues on next page)

(continued from previous page)

```
            "validation_scenes": {
               "title": "Validation Scenes",
               "type": "array",
               "items": {
                  "$ref": "#/definitions/SceneConfig"
               }
            },
            "test_scenes": {
               "title": "Test Scenes",
               "default": [],
               "type": "array",
               "items": {
                  "$ref": "#/definitions/SceneConfig"
               }
            },
            "scene_groups": {
               "title": "Scene Groups",
               "description": "Groupings of scenes. Should be a dict of the form: {
↪<group-name>: Set(scene_id_1, scene_id_2, ...)}. Three groups are added by␣
↪default: \"train_scenes\", \"validation_scenes\", and \"test_scenes\"",
               "default": {},
               "type": "object",
               "additionalProperties": {
                  "type": "array",
                  "items": {
                     "type": "string"
                  },
                  "uniqueItems": true
               }
            },
            "type_hint": {
               "title": "Type Hint",
               "default": "dataset",
               "enum": [
                  "dataset"
               ],
               "type": "string"
            }
         },
         "required": [
            "class_config",
            "train_scenes",
            "validation_scenes"
         ],
         "additionalProperties": false
      },
      "GeoDataWindowMethod": {
         "title": "GeoDataWindowMethod",
         "description": "An enumeration.",
         "enum": [
            "sliding",
            "random"
```

(continues on next page)

(continued from previous page)


```
        ]
      },
      "GeoDataWindowConfig": {
         "title": "GeoDataWindowConfig",
         "description": "Configure a :class:`.GeoDataset`.\n\nSee :mod:
↪`rastervision.pytorch_learner.dataset.dataset`.",
         "type": "object",
         "properties": {
            "method": {
               "default": "sliding",
               "allOf": [
                  {
                     "$ref": "#/definitions/GeoDataWindowMethod"
                  }
               ]
            },
            "size": {
               "title": "Size",
               "description": "If method = sliding, this is the size of sliding␣
↪window. If method = random, this is the size that all the windows are resized to␣
↪before they are returned. If method = random and neither size_lims nor h_lims and␣
↪w_lims have been specified, then size_lims is set to (size, size + 1).",
               "anyOf": [
                  {
                     "type": "integer",
                     "exclusiveMinimum": 0
                  },
                  {
                     "type": "array",
                     "minItems": 2,
                     "maxItems": 2,
                     "items": [
                        {
                           "type": "integer",
                           "exclusiveMinimum": 0
                        },
                        {
                           "type": "integer",
                           "exclusiveMinimum": 0
                        }
                     ]
                  }
               ]
            },
            "stride": {
               "title": "Stride",
               "description": "Stride of sliding window. Only used if method =␣
↪sliding.",
               "anyOf": [
                  {
                     "type": "integer",
                     "exclusiveMinimum": 0
```


(continues on next page)

(continued from previous page)

```
                },
                {
                    "type": "array",
                    "minItems": 2,
                    "maxItems": 2,
                    "items": [
                        {
                            "type": "integer",
                            "exclusiveMinimum": 0
                        },
                        {
                            "type": "integer",
                            "exclusiveMinimum": 0
                        }
                    ]
                }
            ]
        },
        "padding": {
            "title": "Padding",
            "description": "How many pixels are windows allowed to overflow the␣
↪edges of the raster source.",
            "anyOf": [
                {
                    "type": "integer",
                    "minimum": 0
                },
                {
                    "type": "array",
                    "minItems": 2,
                    "maxItems": 2,
                    "items": [
                        {
                            "type": "integer",
                            "minimum": 0
                        },
                        {
                            "type": "integer",
                            "minimum": 0
                        }
                    ]
                }
            ]
        },
        "pad_direction": {
            "title": "Pad Direction",
            "description": "If \"end\", only pad ymax and xmax (bottom and␣
↪right). If \"start\", only pad ymin and xmin (top and left). If \"both\", pad all␣
↪sides. Has no effect if paddiong is zero. Defaults to \"end\".",
            "default": "end",
            "enum": [
                "both",
```

(continues on next page)

(continued from previous page)


```
                "start",
                "end"
            ],
            "type": "string"
        },
        "size_lims": {
            "title": "Size Lims",
            "description": "[min, max) interval from which window sizes will be␣
↪uniformly randomly sampled. The upper limit is exclusive. To fix the size to a␣
↪constant value, use size_lims = (sz, sz + 1). Only used if method = random.␣
↪Specify either size_lims or h_lims and w_lims, but not both. If neither size_lims␣
↪nor h_lims and w_lims have been specified, then this will be set to (size, size +␣
↪1).",
            "type": "array",
            "minItems": 2,
            "maxItems": 2,
            "items": [
                {
                    "type": "integer",
                    "exclusiveMinimum": 0
                },
                {
                    "type": "integer",
                    "exclusiveMinimum": 0
                }
            ]
        },
        "h_lims": {
            "title": "H Lims",
            "description": "[min, max] interval from which window heights will␣
↪be uniformly randomly sampled. Only used if method = random.",
            "type": "array",
            "minItems": 2,
            "maxItems": 2,
            "items": [
                {
                    "type": "integer",
                    "exclusiveMinimum": 0
                },
                {
                    "type": "integer",
                    "exclusiveMinimum": 0
                }
            ]
        },
        "w_lims": {
            "title": "W Lims",
            "description": "[min, max] interval from which window widths will be␣
↪uniformly randomly sampled. Only used if method = random.",
            "type": "array",
            "minItems": 2,
            "maxItems": 2,
```


(continues on next page)

(continued from previous page)

```
                "items": [
                    {
                        "type": "integer",
                        "exclusiveMinimum": 0
                    },
                    {
                        "type": "integer",
                        "exclusiveMinimum": 0
                    }
                ]
            },
            "max_windows": {
                "title": "Max Windows",
                "description": "Max allowed reads from a GeoDataset. Only used if␣
↪method = random.",
                "default": 10000,
                "minimum": 0,
                "type": "integer"
            },
            "max_sample_attempts": {
                "title": "Max Sample Attempts",
                "description": "Max attempts when trying to find a window within the␣
↪AOI of a scene. Only used if method = random and the scene has aoi_polygons␣
↪specified.",
                "default": 100,
                "exclusiveMinimum": 0,
                "type": "integer"
            },
            "efficient_aoi_sampling": {
                "title": "Efficient Aoi Sampling",
                "description": "If the scene has AOIs, sampling windows at random␣
↪anywhere in the extent and then checking if they fall within any of the AOIs can␣
↪be very inefficient. This flag enables the use of an alternate algorithm that␣
↪only samples window locations inside the AOIs. Only used if method = random and␣
↪the scene has aoi_polygons specified. Defaults to True",
                "default": true,
                "type": "boolean"
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "geo_data_window",
                "enum": [
                    "geo_data_window"
                ],
                "type": "string"
            }
        },
        "required": [
            "size"
        ],
        "additionalProperties": false
    },
```

(continues on next page)

(continued from previous page)


```
    "ClassificationGeoDataConfig": {
      "title": "ClassificationGeoDataConfig",
      "description": "Configure classification :class:`GeoDatasets <.GeoDataset>
↪`.\n\nSee :mod:`rastervision.pytorch_learner.dataset.classification_dataset`.",
      "type": "object",
      "properties": {
        "class_names": {
          "title": "Class Names",
          "description": "Names of classes.",
          "default": [],
          "type": "array",
          "items": {
            "type": "string"
          }
        },
        "class_colors": {
          "title": "Class Colors",
          "description": "Colors used to display classes. Can be color 3-
↪tuples in list form.",
          "type": "array",
          "items": {
            "anyOf": [
              {
                "type": "string"
              },
              {
                "type": "array",
                "minItems": 3,
                "maxItems": 3,
                "items": [
                  {
                    "type": "integer"
                  },
                  {
                    "type": "integer"
                  },
                  {
                    "type": "integer"
                  }
                ]
              }
            ]
          }
        },
        "img_channels": {
          "title": "Img Channels",
          "description": "The number of channels of the training images.",
          "exclusiveMinimum": 0,
          "type": "integer"
        },
        "img_sz": {
          "title": "Img Sz",
```


(continues on next page)

(continued from previous page)

```
                "description": "Length of a side of each image in pixels. This is␣
↪the size to transform it to during training, not the size in the raw dataset.",
                "default": 256,
                "exclusiveMinimum": 0,
                "type": "integer"
            },
            "train_sz": {
                "title": "Train Sz",
                "description": "If set, the number of training images to use. If␣
↪fewer images exist, then an exception will be raised.",
                "type": "integer"
            },
            "train_sz_rel": {
                "title": "Train Sz Rel",
                "description": "If set, the proportion of training images to use.",
                "type": "number"
            },
            "num_workers": {
                "title": "Num Workers",
                "description": "Number of workers to use when DataLoader makes␣
↪batches.",
                "default": 4,
                "type": "integer"
            },
            "augmentors": {
                "title": "Augmentors",
                "description": "Names of albumentations augmentors to use for␣
↪training batches. Choices include: ['Blur', 'RandomRotate90', 'HorizontalFlip',
↪'VerticalFlip', 'GaussianBlur', 'GaussNoise', 'RGBShift', 'ToGray'].␣
↪Alternatively, a custom transform can be provided via the aug_transform option.",
                "default": [
                    "RandomRotate90",
                    "HorizontalFlip",
                    "VerticalFlip"
                ],
                "type": "array",
                "items": {
                    "type": "string"
                }
            },
            "base_transform": {
                "title": "Base Transform",
                "description": "An Albumentations transform serialized as a dict␣
↪that will be applied to all datasets: training, validation, and test. This␣
↪transformation is in addition to the resizing due to img_sz. This is useful for,␣
↪for example, applying the same normalization to all datasets.",
                "type": "object"
            },
            "aug_transform": {
                "title": "Aug Transform",
                "description": "An Albumentations transform serialized as a dict␣
↪that will be applied as data augmentation to the training dataset. This transform␣
```

(continues on next page)

(continued from previous page)


```
↪is applied before base_transform. If provided, the augmentors option is ignored.",
                "type": "object"
            },
            "plot_options": {
                "title": "Plot Options",
                "description": "Options to control plotting.",
                "default": {
                    "transform": {
                        "__version__": "1.3.0",
                        "transform": {
                            "__class_fullname__": "rastervision.pytorch_learner.utils.
↪utils.MinMaxNormalize",
                            "always_apply": false,
                            "p": 1.0,
                            "min_val": 0.0,
                            "max_val": 1.0,
                            "dtype": 5
                        }
                    },
                    "channel_display_groups": null,
                    "type_hint": "plot_options"
                },
                "allOf": [
                    {
                        "$ref": "#/definitions/PlotOptions"
                    }
                ]
            },
            "preview_batch_limit": {
                "title": "Preview Batch Limit",
                "description": "Optional limit on the number of items in the preview␣
↪plots produced during training.",
                "type": "integer"
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "classification_geo_data",
                "enum": [
                    "classification_geo_data"
                ],
                "type": "string"
            },
            "scene_dataset": {
                "$ref": "#/definitions/DatasetConfig"
            },
            "window_opts": {
                "title": "Window Opts",
                "default": {},
                "anyOf": [
                    {
                        "$ref": "#/definitions/GeoDataWindowConfig"
                    },
```


(continues on next page)

(continued from previous page)

```
                    {
                        "type": "object",
                        "additionalProperties": {
                            "$ref": "#/definitions/GeoDataWindowConfig"
                        }
                    }
                ]
            }
        },
        "additionalProperties": false
    }
}
}
```

Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *data (Union[rastervision.pytorch_learner.classification_learner_config.ClassificationImageDataConfig, rastervision.pytorch_learner.classification_learner_config.ClassificationGeoDataConfig])*
- *eval_train (bool)*
- *log_tensorboard (bool)*
- *model (Optional[rastervision.pytorch_learner.classification_learner_config.ClassificationModelConfig])*
- *output_uri (Optional[str])*
- *overfit_mode (bool)*
- *predict_mode (bool)*
- *run_tensorboard (bool)*
- *save_model_bundle (bool)*
- *solver (rastervision.pytorch_learner.learner_config.SolverConfig)*
- *test_mode (bool)*
- *type_hint (Literal['classification_learner'])*

Validators

- update_for_mode » all fields
- validate_class_loss_weights » all fields
- validate_run_tensorboard » *run_tensorboard*

field data: Union[*ClassificationImageDataConfig*, *ClassificationGeoDataConfig*] [Required]

Validated by

- update_for_mode
- validate_class_loss_weights

field eval_train: bool = False

If True, runs final evaluation on training set (in addition to test set). Useful for debugging.

Validated by

- update_for_mode
- validate_class_loss_weights

field log_tensorboard: bool = True

Save Tensorboard log files at the end of each epoch.

Validated by

- update_for_mode
- validate_class_loss_weights

field model: Optional[*ClassificationModelConfig*] = None

Validated by

- update_for_mode
- validate_class_loss_weights

field output_uri: Optional[str] = None

URI of where to save output

Validated by

- update_for_mode
- validate_class_loss_weights

field overfit_mode: bool = False

If True, uses half image size, and instead of doing epoch-based training, optimizes the model using a single batch repeatedly for overfit_num_steps number of steps.

Validated by

- update_for_mode
- validate_class_loss_weights

field predict_mode: bool = False

If True, skips training, loads model, and does final eval.

Validated by

- update_for_mode
- validate_class_loss_weights

field run_tensorboard: bool = False

run Tensorboard server during training

Validated by

- update_for_mode
- validate_class_loss_weights

- validate_run_tensorboard

field save_model_bundle: bool = True

If True, saves a model bundle at the end of training which is zip file with model and this LearnerConfig which can be used to make predictions on new images at a later time.

Validated by

- update_for_mode
- validate_class_loss_weights

field solver: *SolverConfig* [Required]

Validated by

- update_for_mode
- validate_class_loss_weights

field test_mode: bool = False

If True, uses test_num_epochs, test_batch_sz, truncated datasets with only a single batch, image_sz that is cut in half, and num_workers = 0. This is useful for testing that code runs correctly on CPU without multithreading before running full job on GPU.

Validated by

- update_for_mode
- validate_class_loss_weights

field type_hint: Literal['classification_learner'] = 'classification_learner'

Validated by

- update_for_mode
- validate_class_loss_weights

build(*tmp_dir=None*, *model_weights_path=None*, *model_def_path=None*, *loss_def_path=None*, *training=True*)

Returns a Learner instantiated using this Config.

Parameters

- **tmp_dir** (*str*) – Root of temp dirs.
- **model_weights_path** (*str, optional*) – A local path to model weights. Defaults to None.
- **model_def_path** (*str, optional*) – A local path to a directory with a hubconf.py. If provided, the model definition is imported from here. Defaults to None.
- **loss_def_path** (*str, optional*) – A local path to a directory with a hubconf.py. If provided, the loss function definition is imported from here. Defaults to None.
- **training** (*bool, optional*) – Whether the model is to be used for training or prediction. If False, the model is put in eval mode and the loss function, optimizer, etc. are not initialized. Defaults to True.

get_model_bundle_uri() → str

Returns the URI of where the model bundle is stored.

Return type

str

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(**args*, ***kwargs*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validator update_for_mode » *all fields*

Parameters
values (*dict*) –

Return type
dict

validator validate_class_loss_weights » *all fields*

Parameters
values (*dict*) –

Return type
dict

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises
ConfigError – if field is invalid

validator validate_run_tensorboard » run_tensorboard

Parameters

- **v** (*bool*) –
- **values** (*dict*) –

Return type
bool

ClassificationModelConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model ClassificationModelConfig

Configure a classification model.


```
{
   "title": "ClassificationModelConfig",
   "description": "Configure a classification model.",
   "type": "object",
   "properties": {
      "backbone": {
         "description": "The torchvision.models backbone to use.",
         "default": "resnet18",
         "allOf": [
            {
               "$ref": "#/definitions/Backbone"
            }
         ]
      },
      "pretrained": {
         "title": "Pretrained",
         "description": "If True, use ImageNet weights. If False, use random␣
↪initialization.",
         "default": true,
         "type": "boolean"
      },
      "init_weights": {
         "title": "Init Weights",
         "description": "URI of PyTorch model weights used to initialize model. If␣
↪set, this supercedes the pretrained option.",
         "type": "string"
      },
      "load_strict": {
         "title": "Load Strict",
         "description": "If True, the keys in the state dict referenced by init_
↪weights must match exactly. Setting this to False can be useful if you just want␣
↪to load the backbone of a model.",
         "default": true,
         "type": "boolean"
      },
      "external_def": {
         "title": "External Def",
         "description": "If specified, the model will be built from the definition␣
↪from this external source, using Torch Hub.",
         "allOf": [
            {
               "$ref": "#/definitions/ExternalModuleConfig"
            }
         ]
      },
```


(continues on next page)

(continued from previous page)


```
      "type_hint": {
         "title": "Type Hint",
         "default": "classification_model",
         "enum": [
            "classification_model"
         ],
         "type": "string"
      }
   },
   "additionalProperties": false,
   "definitions": {
      "Backbone": {
         "title": "Backbone",
         "description": "An enumeration.",
         "enum": [
            "alexnet",
            "densenet121",
            "densenet169",
            "densenet201",
            "densenet161",
            "googlenet",
            "inception_v3",
            "mnasnet0_5",
            "mnasnet0_75",
            "mnasnet1_0",
            "mnasnet1_3",
            "mobilenet_v2",
            "resnet18",
            "resnet34",
            "resnet50",
            "resnet101",
            "resnet152",
            "resnext50_32x4d",
            "resnext101_32x8d",
            "wide_resnet50_2",
            "wide_resnet101_2",
            "shufflenet_v2_x0_5",
            "shufflenet_v2_x1_0",
            "shufflenet_v2_x1_5",
            "shufflenet_v2_x2_0",
            "squeezenet1_0",
            "squeezenet1_1",
            "vgg11",
            "vgg11_bn",
            "vgg13",
            "vgg13_bn",
            "vgg16",
            "vgg16_bn",
            "vgg19_bn",
            "vgg19"
         ]
      },
```


(continues on next page)

(continued from previous page)


```
      "ExternalModuleConfig": {
         "title": "ExternalModuleConfig",
         "description": "Config describing an object to be loaded via Torch Hub.",
         "type": "object",
         "properties": {
            "uri": {
               "title": "Uri",
               "description": "Local uri of a zip file, or local uri of a directory,
↪or remote uri of zip file.",
               "minLength": 1,
               "type": "string"
            },
            "github_repo": {
               "title": "Github Repo",
               "description": "<repo-owner>/<repo-name>[:tag]",
               "pattern": ".+/.+",
               "type": "string"
            },
            "name": {
               "title": "Name",
               "description": "Name of the folder in which to extract/copy the␣
↪definition files.",
               "minLength": 1,
               "type": "string"
            },
            "entrypoint": {
               "title": "Entrypoint",
               "description": "Name of a callable present in hubconf.py. See docs␣
↪for torch.hub for details.",
               "minLength": 1,
               "type": "string"
            },
            "entrypoint_args": {
               "title": "Entrypoint Args",
               "description": "Args to pass to the entrypoint. Must be serializable.
↪",
               "default": [],
               "type": "array",
               "items": {}
            },
            "entrypoint_kwargs": {
               "title": "Entrypoint Kwargs",
               "description": "Keyword args to pass to the entrypoint. Must be␣
↪serializable.",
               "default": {},
               "type": "object"
            },
            "force_reload": {
               "title": "Force Reload",
               "description": "Force reload of module definition.",
               "default": false,
               "type": "boolean"
```


(continues on next page)

(continued from previous page)


```
            },
            "type_hint": {
               "title": "Type Hint",
               "default": "external-module",
               "enum": [
                  "external-module"
               ],
               "type": "string"
            }
         },
         "required": [
            "entrypoint"
         ],
         "additionalProperties": false
      }
   }
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *backbone (rastervision.pytorch_learner.learner_config.Backbone)*
- *external_def (Optional[rastervision.pytorch_learner.learner_config.ExternalModuleConfig])*
- *init_weights (Optional[str])*
- *load_strict (bool)*
- *pretrained (bool)*
- *type_hint (Literal['classification_model'])*

field backbone: *Backbone* = Backbone.resnet18

The torchvision.models backbone to use.

field external_def: Optional[*ExternalModuleConfig*] = None

If specified, the model will be built from the definition from this external source, using Torch Hub.

field init_weights: Optional[str] = None

URI of PyTorch model weights used to initialize model. If set, this supercedes the pretrained option.

field load_strict: bool = True

If True, the keys in the state dict referenced by init_weights must match exactly. Setting this to False can be useful if you just want to load the backbone of a model.

field pretrained: bool = True

If True, use ImageNet weights. If False, use random initialization.

field type_hint: Literal['classification_model'] = 'classification_model'

build(*num_classes: int*, *in_channels: int*, *save_dir: Optional[str] = None*, *hubconf_dir: Optional[str] = None*, ***kwargs*) → torch.nn.Module

Build and return a model based on the config.

Parameters

- **num_classes** (*int*) – Number of classes.
- **in_channels** (*int, optional*) – Number of channels in the images that will be fed into the model. Defaults to 3.
- **save_dir** (*Optional[str], optional*) – Used for building external_def if specified. Defaults to None.
- **hubconf_dir** (*Optional[str], optional*) – Used for building external_def if specified. Defaults to None.

Returns
a PyTorch nn.Module.

Return type
nn.Module

build_default_model(*num_classes: int*, *in_channels: int*) → torch.nn.Module

Build and return the default model.

Parameters

- **num_classes** (*int*) – Number of classes.
- **in_channels** (*int, optional*) – Number of channels in the images that will be fed into the model. Defaults to 3.

Returns
a PyTorch nn.Module.

Return type
nn.Module

build_external_model(*save_dir: str*, *hubconf_dir: Optional[str] = None*) → torch.nn.Module

Build and return an external model.

Parameters

- **save_dir** (*str*) – The module def will be saved here.
- **hubconf_dir** (*Optional[str], optional*) – Path to existing definition. Defaults to None.

Returns
a PyTorch nn.Module.

Return type
nn.Module

get_backbone_str()

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(**args*, ***kwargs*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

9.3.3 dataset

Modules

classification_dataset

dataset

object_detection_dataset

regression_dataset

semantic_segmentation_dataset

transform

utils

visualizer

classification_dataset

Classes

ClassificationImageDataset	Read images and class labels from images stored in class folders.
ClassificationRandomWindowGeoDataset	
ClassificationSlidingWindowGeoDataset	

ClassificationImageDataset

class ClassificationImageDataset

Bases: *ImageDataset*

Read images and class labels from images stored in class folders.

I.e., all images for a class "A" are stored in directory `A/`, all images for a class "B" are stored in directory `B/`, and so on. And all class directories are located in the same parent directory.

__init__(*data_dir: str*, *class_names: Optional[Iterable[str]]*, **args*, ***kwargs*)

Constructor.

Parameters

- **data_dir** (*str*) – Root directory containing class dirs.
- **class_names** (*Optional[Iterable[str]]*) – Class names. Should match class dir names.
- ***args** – See *ImageDataset.__init__()*.
- ****kwargs** – See *ImageDataset.__init__()*.

Methods

__init__(data_dir, class_names, *args, **kwargs)	Constructor.

__init__(*data_dir: str*, *class_names: Optional[Iterable[str]]*, **args*, ***kwargs*)

Constructor.

Parameters

- **data_dir** (*str*) – Root directory containing class dirs.
- **class_names** (*Optional[Iterable[str]]*) – Class names. Should match class dir names.
- ***args** – See *ImageDataset.__init__()*.
- ****kwargs** – See *ImageDataset.__init__()*.

static __new__(*cls*, **args: Any*, ***kwargs: Any*) → Any

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
Any

ClassificationRandomWindowGeoDataset

class ClassificationRandomWindowGeoDataset

Bases: *RandomWindowGeoDataset*

Attributes

max_size
min_size

__init__(**args*, ***kwargs*)

Constructor.

Will sample square windows if size_lims is specified. Otherwise, will sample rectangular windows with height and width sampled according to h_lims and w_lims.

Parameters

- **scene** (Scene) – A Scene object.
- **out_size** (*Optional[Union[PosInt, Tuple[PosInt, PosInt]]]*) – Resize windows to this size before returning. This is to aid in collating the windows into a batch. If None, windows are returned without being normalized or converted to pytorch, and will be of different sizes in successive reads.
- **size_lims** (*Optional[Tuple[PosInt, PosInt]]*) – Interval from which to sample window size.
- **h_lims** (*Optional[Tuple[PosInt, PosInt]]*) – Interval from which to sample window height.
- **w_lims** (*Optional[Tuple[PosInt, PosInt]]*) – Interval from which to sample window width.
- **padding** (*Optional[Union[NonNegInt, Tuple[NonNegInt, NonNegInt]]]*) – How many pixels the windows are allowed to overflow the sides of the raster source. If None, padding = size. Defaults to None.
- **max_windows** (*Optional[NonNegInt]*) – Max allowed reads. Will raise StopIteration on further read attempts. If None, will be set to np.inf. Defaults to None.
- **transform** (*Optional[A.BasicTransform], optional*) – Albumentations transform to apply to the windows. Defaults to None. Each transform in Albumentations takes images of type uint8, and sometimes other data types. The data type requirements can be

seen at https://albumentations.ai/docs/api_reference/augmentations/transforms/ If there is a mismatch between the data type of imagery and the transform requirements, a RasterTransformer should be set on the RasterSource that converts to uint8, such as MinMaxTransformer or StatsTransformer.

- **transform_type** (*Optional[TransformType], optional*) – Type of transform. Defaults to None.
- **max_sample_attempts** (*NonNegInt, optional*) – Max attempts when trying to find a window within the AOI of the scene. Only used if the scene has aoi_polygons specified. StopIteratioin is raised if this is exceeded. Defaults to 100.
- **return_window** (*bool, optional*) – Make __getitem__ return the window coordinates used to generate the image. Defaults to False.
- **efficient_aoi_sampling** (*bool, optional*) – If the scene has AOIs, sampling windows at random anywhere in the extent and then checking if they fall within any of the AOIs can be very inefficient. This flag enables the use of an alternate algorithm that only samples window locations inside the AOIs. Defaults to True.
- **transform** – Albumentations transform to apply to the windows. Defaults to None.
- **transform_type** – Type of transform. Defaults to None.
- **normalize** (*bool, optional*) – If True, x is normalized to [0, 1] based on its data type. Defaults to True.
- **to_pytorch** (*bool, optional*) – If True, x and y are converted to pytorch tensors. Defaults to True.

Methods

__init__(*args, **kwargs)	Constructor.
from_uris(image_uri[, label_vector_uri, ...])	Create an instance of this class from image and label URIs.
get_resize_transform(transform, out_size)	Get transform to use for resizing windows to out_size.
sample_window()	If scene has AOI polygons, try to find a random window that is within the AOI.
sample_window_loc(h, w)	Randomly sample coordinates of the top left corner of the window.
sample_window_size()	Randomly sample the window size.

__init__(**args*, ***kwargs*)

Constructor.

Will sample square windows if size_lims is specified. Otherwise, will sample rectangular windows with height and width sampled according to h_lims and w_lims.

Parameters

- **scene** (Scene) – A Scene object.
- **out_size** (*Optional[Union[PosInt, Tuple[PosInt, PosInt]]]*) – Resize windows to this size before returning. This is to aid in collating the windows into a batch. If None, windows are returned without being normalized or converted to pytorch, and will be of different sizes in successive reads.

- **size_lims** (*Optional[Tuple[PosInt, PosInt]]*) – Interval from which to sample window size.
- **h_lims** (*Optional[Tuple[PosInt, PosInt]]*) – Interval from which to sample window height.
- **w_lims** (*Optional[Tuple[PosInt, PosInt]]*) – Interval from which to sample window width.
- **padding** (*Optional[Union[NonNegInt, Tuple[NonNegInt, NonNegInt]]]*) – How many pixels the windows are allowed to overflow the sides of the raster source. If None, padding = size. Defaults to None.
- **max_windows** (*Optional[NonNegInt]*) – Max allowed reads. Will raise StopIteration on further read attempts. If None, will be set to np.inf. Defaults to None.
- **transform** (*Optional[A.BasicTransform], optional*) – Albumentations transform to apply to the windows. Defaults to None. Each transform in Albumentations takes images of type uint8, and sometimes other data types. The data type requirements can be seen at https://albumentations.ai/docs/api_reference/augmentations/transforms/ If there is a mismatch between the data type of imagery and the transform requirements, a RasterTransformer should be set on the RasterSource that converts to uint8, such as MinMaxTransformer or StatsTransformer.
- **transform_type** (*Optional[TransformType], optional*) – Type of transform. Defaults to None.
- **max_sample_attempts** (*NonNegInt, optional*) – Max attempts when trying to find a window within the AOI of the scene. Only used if the scene has aoi_polygons specified. StopIteratioin is raised if this is exceeded. Defaults to 100.
- **return_window** (*bool, optional*) – Make __getitem__ return the window coordinates used to generate the image. Defaults to False.
- **efficient_aoi_sampling** (*bool, optional*) – If the scene has AOIs, sampling windows at random anywhere in the extent and then checking if they fall within any of the AOIs can be very inefficient. This flag enables the use of an alternate algorithm that only samples window locations inside the AOIs. Defaults to True.
- **transform** – Albumentations transform to apply to the windows. Defaults to None.
- **transform_type** – Type of transform. Defaults to None.
- **normalize** (*bool, optional*) – If True, x is normalized to [0, 1] based on its data type. Defaults to True.
- **to_pytorch** (*bool, optional*) – If True, x and y are converted to pytorch tensors. Defaults to True.

static **__new__**(*cls*, **args: Any*, ***kwargs: Any*) → Any

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type

Any

classmethod from_uris(*image_uri: Union[str, List[str]]*, *label_vector_uri: Optional[str] = None*, *class_config: Optional[ClassConfig] = None*, *aoi_uri: Union[str, List[str]] = []*, *label_vector_default_class_id: Optional[int] = None*, *image_raster_source_kw: dict = {}*, *label_vector_source_kw: dict = {}*, *label_source_kw: dict = {}*, ***kwargs*)

Create an instance of this class from image and label URIs.

This is a convenience method. For more fine-grained control, it is recommended to use the default constructor.

Parameters

- **class_config** (*Optional['ClassConfig']*) – The ClassConfig.
- **image_uri** (*Union[str, List[str]]*) – URI or list of URIs of GeoTIFFs to use as the source of image data.
- **label_vector_uri** (*Optional[str], optional*) – URI of GeoJSON file to use as the source of segmentation label data. Defaults to None.
- **class_config** – The ClassConfig. Can be None if not using any labels.
- **aoi_uri** (*Union[str, List[str]], optional*) – URI or list of URIs of GeoJSONs that specify the area-of-interest. If provided, the dataset will only access data from this area. Defaults to [].
- **label_vector_default_class_id** (*Optional[int], optional*) – If using label_vector_uri and all polygons in that file belong to the same class and they do not contain a *class_id* property, then use this argument to map all of the polgons to the appropriate class ID. See docs for ClassInferenceTransformer for more details. Defaults to None.
- **image_raster_source_kw** (*dict, optional*) – Additional arguments to pass to the RasterioSource used for image data. See docs for RasterioSource for more details. Defaults to {}.
- **label_vector_source_kw** (*dict, optional*) – Additional arguments to pass to the GeoJSONVectorSourceConfig used for label data, if label_vector_uri is set. See docs for GeoJSONVectorSourceConfig for more details. Defaults to {}.
- **label_source_kw** (*dict, optional*) – Additional arguments to pass to the ChipClassificationLabelSourceConfig used for label data, if label_vector_uri is set. See docs for ChipClassificationLabelSourceConfig for more details. Defaults to {}.
- ****kwargs** – All other keyword args are passed to the default constructor for this class.

Returns
An instance of this GeoDataset subclass.

get_resize_transform(*transform: Optional[BasicTransform]*, *out_size: Tuple[PositiveInt, PositiveInt]*) → Union[Resize, Compose]

Get transform to use for resizing windows to out_size.

Parameters

- **transform** (*Optional[BasicTransform]*) –
- **out_size** (*Tuple[PositiveInt, PositiveInt]*) –

Return type
Union[*Resize*, *Compose*]

sample_window() → *Box*

If scene has AOI polygons, try to find a random window that is within the AOI. Otherwise, just return the first sampled window.

Raises
StopIteration – If unable to find a valid window within self.max_sample_attempts attempts.

Returns
The sampled window.

Return type
Box

sample_window_loc(*h: int*, *w: int*) → Tuple[int, int]

Randomly sample coordinates of the top left corner of the window.

Parameters

- **h** (*int*) –
- **w** (*int*) –

Return type
Tuple[int, int]

sample_window_size() → Tuple[int, int]

Randomly sample the window size.

Return type
Tuple[int, int]

property max_size

property min_size

ClassificationSlidingWindowGeoDataset

class ClassificationSlidingWindowGeoDataset

Bases: *SlidingWindowGeoDataset*

__init__(**args*, ***kwargs*)

Constructor.

Parameters

- **scene** (Scene) – A Scene object.
- **size** (*Union[PosInt, Tuple[PosInt, PosInt]]*) – Window size.
- **stride** (*Union[PosInt, Tuple[PosInt, PosInt]]*) – Step size between windows.
- **padding** (*Optional[Union[NonNegInt, Tuple[NonNegInt, NonNegInt]]]*) – How many pixels the windows are allowed to overflow the sides of the raster source. If None, padding is set to size // 2. Defaults to None.
- **pad_direction** (*Literal['both', 'start', 'end']*) – If 'end', only pad ymax and xmax (bottom and right). If 'start', only pad ymin and xmin (top and left). If 'both', pad all sides. Has no effect if paddiong is zero. Defaults to 'end'.

- **transform** (*Optional[A.BasicTransform], optional*) – Albumentations transform to apply to the windows. Defaults to None. Each transform in Albumentations takes images of type uint8, and sometimes other data types. The data type requirements can be seen at https://albumentations.ai/docs/api_reference/augmentations/transforms/ # noqa If there is a mismatch between the data type of imagery and the transform requirements, a RasterTransformer should be set on the RasterSource that converts to uint8, such as MinMaxTransformer or StatsTransformer.
- **transform_type** (*Optional[TransformType], optional*) – Type of transform. Defaults to None.
- **normalize** (*bool, optional*) – If True, x is normalized to [0, 1] based on its data type. Defaults to True.
- **to_pytorch** (*bool, optional*) – If True, x and y are converted to pytorch tensors. Defaults to True.

Methods

__init__(*args, **kwargs)	Constructor.
from_uris(image_uri[, label_vector_uri, ...])	Create an instance of this class from image and label URIs.
init_windows()	Pre-compute windows.

__init__(**args*, ***kwargs*)

Constructor.

Parameters

- **scene** (Scene) – A Scene object.
- **size** (*Union[PosInt, Tuple[PosInt, PosInt]]*) – Window size.
- **stride** (*Union[PosInt, Tuple[PosInt, PosInt]]*) – Step size between windows.
- **padding** (*Optional[Union[NonNegInt, Tuple[NonNegInt, NonNegInt]]]*) – How many pixels the windows are allowed to overflow the sides of the raster source. If None, padding is set to size // 2. Defaults to None.
- **pad_direction** (*Literal['both', 'start', 'end']*) – If 'end', only pad ymax and xmax (bottom and right). If 'start', only pad ymin and xmin (top and left). If 'both', pad all sides. Has no effect if paddiong is zero. Defaults to 'end'.
- **transform** (*Optional[A.BasicTransform], optional*) – Albumentations transform to apply to the windows. Defaults to None. Each transform in Albumentations takes images of type uint8, and sometimes other data types. The data type requirements can be seen at https://albumentations.ai/docs/api_reference/augmentations/transforms/ # noqa If there is a mismatch between the data type of imagery and the transform requirements, a RasterTransformer should be set on the RasterSource that converts to uint8, such as MinMaxTransformer or StatsTransformer.
- **transform_type** (*Optional[TransformType], optional*) – Type of transform. Defaults to None.
- **normalize** (*bool, optional*) – If True, x is normalized to [0, 1] based on its data type. Defaults to True.
- **to_pytorch** (*bool, optional*) – If True, x and y are converted to pytorch tensors. Defaults to True.

static __new__(*cls*, **args: Any*, ***kwargs: Any*) → Any

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
Any

classmethod from_uris(*image_uri: Union[str, List[str]]*, *label_vector_uri: Optional[str] = None*, *class_config: Optional[*ClassConfig*] = None*, *aoi_uri: Union[str, List[str]] = []*, *label_vector_default_class_id: Optional[int] = None*, *image_raster_source_kw: dict = {}*, *label_vector_source_kw: dict = {}*, *label_source_kw: dict = {}*, ***kwargs*)

Create an instance of this class from image and label URIs.

This is a convenience method. For more fine-grained control, it is recommended to use the default constructor.

Parameters

- **class_config** (*Optional['ClassConfig']*) – The ClassConfig.
- **image_uri** (*Union[str, List[str]]*) – URI or list of URIs of GeoTIFFs to use as the source of image data.
- **label_vector_uri** (*Optional[str], optional*) – URI of GeoJSON file to use as the source of segmentation label data. Defaults to None.
- **class_config** – The ClassConfig. Can be None if not using any labels.
- **aoi_uri** (*Union[str, List[str]], optional*) – URI or list of URIs of GeoJSONs that specify the area-of-interest. If provided, the dataset will only access data from this area. Defaults to [].
- **label_vector_default_class_id** (*Optional[int], optional*) – If using label_vector_uri and all polygons in that file belong to the same class and they do not contain a *class_id* property, then use this argument to map all of the polgons to the appropriate class ID. See docs for ClassInferenceTransformer for more details. Defaults to None.
- **image_raster_source_kw** (*dict, optional*) – Additional arguments to pass to the RasterioSource used for image data. See docs for RasterioSource for more details. Defaults to {}.
- **label_vector_source_kw** (*dict, optional*) – Additional arguments to pass to the GeoJSONVectorSourceConfig used for label data, if label_vector_uri is set. See docs for GeoJSONVectorSourceConfig for more details. Defaults to {}.
- **label_source_kw** (*dict, optional*) – Additional arguments to pass to the ChipClassificationLabelSourceConfig used for label data, if label_vector_uri is set. See docs for ChipClassificationLabelSourceConfig for more details. Defaults to {}.
- ****kwargs** – All other keyword args are passed to the default constructor for this class.

Returns
An instance of this GeoDataset subclass.

init_windows()

Pre-compute windows.

Functions

make_cc_geodataset(cls, image_uri[, ...])	Create an instance of this class from image and label URIs.

make_cc_geodataset

make_cc_geodataset(*cls*, *image_uri: Union[str, List[str]]*, *label_vector_uri: Optional[str] = None*, *class_config: Optional[*ClassConfig*] = None*, *aoi_uri: Union[str, List[str]] = []*, *label_vector_default_class_id: Optional[int] = None*, *image_raster_source_kw: dict = {}*, *label_vector_source_kw: dict = {}*, *label_source_kw: dict = {}*, ***kwargs*)

Create an instance of this class from image and label URIs.

This is a convenience method. For more fine-grained control, it is recommended to use the default constructor.

Parameters

- **class_config** (*Optional['ClassConfig']*) – The ClassConfig.
- **image_uri** (*Union[str, List[str]]*) – URI or list of URIs of GeoTIFFs to use as the source of image data.
- **label_vector_uri** (*Optional[str], optional*) – URI of GeoJSON file to use as the source of segmentation label data. Defaults to None.
- **class_config** – The ClassConfig. Can be None if not using any labels.
- **aoi_uri** (*Union[str, List[str]], optional*) – URI or list of URIs of GeoJSONs that specify the area-of-interest. If provided, the dataset will only access data from this area. Defaults to [].
- **label_vector_default_class_id** (*Optional[int], optional*) – If using label_vector_uri and all polygons in that file belong to the same class and they do not contain a *class_id* property, then use this argument to map all of the polgons to the appropriate class ID. See docs for ClassInferenceTransformer for more details. Defaults to None.
- **image_raster_source_kw** (*dict, optional*) – Additional arguments to pass to the RasterioSource used for image data. See docs for RasterioSource for more details. Defaults to {}.
- **label_vector_source_kw** (*dict, optional*) – Additional arguments to pass to the GeoJSONVectorSourceConfig used for label data, if label_vector_uri is set. See docs for GeoJSONVectorSourceConfig for more details. Defaults to {}.
- **label_source_kw** (*dict, optional*) – Additional arguments to pass to the ChipClassificationLabelSourceConfig used for label data, if label_vector_uri is set. See docs for ChipClassificationLabelSourceConfig for more details. Defaults to {}.
- ****kwargs** – All other keyword args are passed to the default constructor for this class.

Returns

An instance of this GeoDataset subclass.

dataset

Classes

AlbumentationsDataset	An adapter to use arbitrary datasets with albumentations transforms.
GeoDataset	Dataset that reads directly from a Scene (i.e.
ImageDataset	Dataset that reads from image files.
RandomWindowGeoDataset	Read the scene by sampling random window sizes and locations.
SlidingWindowGeoDataset	Read the scene left-to-right, top-to-bottom, using a sliding window.

AlbumentationsDataset

class AlbumentationsDataset

Bases: Dataset

An adapter to use arbitrary datasets with albumentations transforms.

__init__(*orig_dataset: Any*, *transform: Optional[BasicTransform] = None*, *transform_type:* TransformType *= TransformType.noop*, *normalize=True*, *to_pytorch=True*)

Constructor.

Parameters

- **orig_dataset** (*Any*) – An object with a __getitem__ and __len__.
- **transform** (*A.BasicTransform, optional*) – Albumentations transform to apply to the windows. Defaults to None. Each transform in Albumentations takes images of type uint8, and sometimes other data types. The data type requirements can be seen at https://albumentations.ai/docs/api_reference/augmentations/transforms/ # noqa If there is a mismatch between the data type of imagery and the transform requirements, a RasterTransformer should be set on the RasterSource that converts to uint8, such as MinMaxTransformer or StatsTransformer.
- **transform_type** (TransformType) – The type of transform so that its inputs and outputs can be handled correctly. Defaults to TransformType.noop.
- **normalize** (*bool, optional*) – If True, x is normalized to [0, 1] based on its data type. Defaults to True.
- **to_pytorch** (*bool, optional*) – If True, x and y are converted to pytorch tensors. Defaults to True.

Methods

__init__(orig_dataset[, transform, ...])	Constructor.

__init__(*orig_dataset: Any*, *transform: Optional[BasicTransform] = None*, *transform_type: TransformType = TransformType.noop*, *normalize=True*, *to_pytorch=True*)

Constructor.

Parameters

- **orig_dataset** (*Any*) – An object with a __getitem__ and __len__.
- **transform** (*A.BasicTransform, optional*) – Albumentations transform to apply to the windows. Defaults to None. Each transform in Albumentations takes images of type uint8, and sometimes other data types. The data type requirements can be seen at https://albumentations.ai/docs/api_reference/augmentations/transforms/ # noqa If there is a mismatch between the data type of imagery and the transform requirements, a RasterTransformer should be set on the RasterSource that converts to uint8, such as MinMaxTransformer or StatsTransformer.
- **transform_type** (*TransformType*) – The type of transform so that its inputs and outputs can be handled correctly. Defaults to TransformType.noop.
- **normalize** (*bool, optional*) – If True, x is normalized to [0, 1] based on its data type. Defaults to True.
- **to_pytorch** (*bool, optional*) – If True, x and y are converted to pytorch tensors. Defaults to True.

static __new__(*cls*, **args: Any*, ***kwargs: Any*) → Any

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
Any

GeoDataset

class GeoDataset

Bases: *AlbumentationsDataset*

Dataset that reads directly from a Scene (i.e. a raster source and a label source).

__init__(*scene: Scene*, *transform: Optional[BasicTransform] = None*, *transform_type: Optional[TransformType] = None*, *normalize: bool = True*, *to_pytorch: bool = True*)

Constructor.

Parameters

- **scene** (*Scene*) – A Scene object.
- **transform** (*Optional[A.BasicTransform], optional*) – Albumentations transform to apply to the windows. Defaults to None. Each transform in Albumentations takes images of type uint8, and sometimes other data types. The data type requirements can be seen at https://albumentations.ai/docs/api_reference/augmentations/transforms/ # noqa If

there is a mismatch between the data type of imagery and the transform requirements, a RasterTransformer should be set on the RasterSource that converts to uint8, such as MinMaxTransformer or StatsTransformer.

- **transform_type** (*Optional[TransformType], optional*) – Type of transform. Defaults to None.
- **normalize** (*bool, optional*) – If True, x is normalized to [0, 1] based on its data type. Defaults to True.
- **to_pytorch** (*bool, optional*) – If True, x and y are converted to pytorch tensors. Defaults to True.

Methods

__init__(scene[, transform, transform_type, ...])	Constructor.
from_uris(*args, **kwargs)	

__init__(*scene: Scene*, *transform: Optional[BasicTransform] = None*, *transform_type: Optional[TransformType] = None*, *normalize: bool = True*, *to_pytorch: bool = True*)

Constructor.

Parameters

- **scene** (*Scene*) – A Scene object.
- **transform** (*Optional[A.BasicTransform], optional*) – Albumentations transform to apply to the windows. Defaults to None. Each transform in Albumentations takes images of type uint8, and sometimes other data types. The data type requirements can be seen at https://albumentations.ai/docs/api_reference/augmentations/transforms/ # noqa If there is a mismatch between the data type of imagery and the transform requirements, a RasterTransformer should be set on the RasterSource that converts to uint8, such as MinMaxTransformer or StatsTransformer.
- **transform_type** (*Optional[TransformType], optional*) – Type of transform. Defaults to None.
- **normalize** (*bool, optional*) – If True, x is normalized to [0, 1] based on its data type. Defaults to True.
- **to_pytorch** (*bool, optional*) – If True, x and y are converted to pytorch tensors. Defaults to True.

static __new__(*cls*, **args: Any*, ***kwargs: Any*) → Any

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
Any

classmethod from_uris(**args*, ***kwargs*) → *GeoDataset*

Return type
GeoDataset

ImageDataset

class ImageDataset

Bases: *AlbumentationsDataset*

Dataset that reads from image files.

__init__(*orig_dataset: Any*, *transform: Optional[BasicTransform] = None*, *transform_type: TransformType = TransformType.noop*, *normalize=True*, *to_pytorch=True*)

Constructor.

Parameters

- **orig_dataset** (*Any*) – An object with a __getitem__ and __len__.
- **transform** (*A.BasicTransform, optional*) – Albumentations transform to apply to the windows. Defaults to None. Each transform in Albumentations takes images of type uint8, and sometimes other data types. The data type requirements can be seen at https://albumentations.ai/docs/api_reference/augmentations/transforms/ # noqa If there is a mismatch between the data type of imagery and the transform requirements, a RasterTransformer should be set on the RasterSource that converts to uint8, such as MinMaxTransformer or StatsTransformer.
- **transform_type** (*TransformType*) – The type of transform so that its inputs and outputs can be handled correctly. Defaults to TransformType.noop.
- **normalize** (*bool, optional*) – If True, x is normalized to [0, 1] based on its data type. Defaults to True.
- **to_pytorch** (*bool, optional*) – If True, x and y are converted to pytorch tensors. Defaults to True.

Methods

__init__(orig_dataset[, transform, ...])	Constructor.

__init__(*orig_dataset: Any*, *transform: Optional[BasicTransform] = None*, *transform_type: TransformType = TransformType.noop*, *normalize=True*, *to_pytorch=True*)

Constructor.

Parameters

- **orig_dataset** (*Any*) – An object with a __getitem__ and __len__.
- **transform** (*A.BasicTransform, optional*) – Albumentations transform to apply to the windows. Defaults to None. Each transform in Albumentations takes images of type uint8, and sometimes other data types. The data type requirements can be seen at https://albumentations.ai/docs/api_reference/augmentations/transforms/ # noqa If there is a mismatch between the data type of imagery and the transform requirements, a RasterTransformer should be set on the RasterSource that converts to uint8, such as MinMaxTransformer or StatsTransformer.
- **transform_type** (*TransformType*) – The type of transform so that its inputs and outputs can be handled correctly. Defaults to TransformType.noop.
- **normalize** (*bool, optional*) – If True, x is normalized to [0, 1] based on its data type. Defaults to True.

- **to_pytorch** (*bool, optional*) – If True, x and y are converted to pytorch tensors. Defaults to True.

static __new__(*cls*, **args: Any*, ***kwargs: Any*) → Any

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
Any

RandomWindowGeoDataset

class RandomWindowGeoDataset

Bases: *GeoDataset*

Read the scene by sampling random window sizes and locations.

Attributes

max_size

min_size

__init__(*scene: Scene*, *out_size: Optional[Union[PositiveInt, Tuple[PositiveInt, PositiveInt]]]*, *size_lims: Optional[Tuple[PositiveInt, PositiveInt]] = None*, *h_lims: Optional[Tuple[PositiveInt, PositiveInt]] = None*, *w_lims: Optional[Tuple[PositiveInt, PositiveInt]] = None*, *padding: Optional[Union[ConstrainedIntValue, Tuple[ConstrainedIntValue, ConstrainedIntValue]]] = None*, *max_windows: Optional[ConstrainedIntValue] = None*, *max_sample_attempts: PositiveInt = 100*, *return_window: bool = False*, *efficient_aoi_sampling: bool = True*, *transform: Optional[BasicTransform] = None*, *transform_type: Optional[TransformType] = None*, *normalize: bool = True*, *to_pytorch: bool = True*)

Constructor.

Will sample square windows if size_lims is specified. Otherwise, will sample rectangular windows with height and width sampled according to h_lims and w_lims.

Parameters

- **scene** (*Scene*) – A Scene object.
- **out_size** (*Optional[Union[PosInt, Tuple[PosInt, PosInt]]]*) – Resize windows to this size before returning. This is to aid in collating the windows into a batch. If None, windows are returned without being normalized or converted to pytorch, and will be of different sizes in successive reads.
- **size_lims** (*Optional[Tuple[PosInt, PosInt]]*) – Interval from which to sample window size.
- **h_lims** (*Optional[Tuple[PosInt, PosInt]]*) – Interval from which to sample window height.

- **w_lims** (*Optional[Tuple[PosInt, PosInt]]*) – Interval from which to sample window width.
- **padding** (*Optional[Union[NonNegInt, Tuple[NonNegInt, NonNegInt]]]*) – How many pixels the windows are allowed to overflow the sides of the raster source. If None, padding = size. Defaults to None.
- **max_windows** (*Optional[NonNegInt]*) – Max allowed reads. Will raise StopIteration on further read attempts. If None, will be set to np.inf. Defaults to None.
- **transform** (*Optional[A.BasicTransform], optional*) – Albumentations transform to apply to the windows. Defaults to None. Each transform in Albumentations takes images of type uint8, and sometimes other data types. The data type requirements can be seen at https://albumentations.ai/docs/api_reference/augmentations/transforms/ If there is a mismatch between the data type of imagery and the transform requirements, a RasterTransformer should be set on the RasterSource that converts to uint8, such as MinMaxTransformer or StatsTransformer.
- **transform_type** (*Optional[TransformType], optional*) – Type of transform. Defaults to None.
- **max_sample_attempts** (*NonNegInt, optional*) – Max attempts when trying to find a window within the AOI of the scene. Only used if the scene has aoi_polygons specified. StopIteratioin is raised if this is exceeded. Defaults to 100.
- **return_window** (*bool, optional*) – Make __getitem__ return the window coordinates used to generate the image. Defaults to False.
- **efficient_aoi_sampling** (*bool, optional*) – If the scene has AOIs, sampling windows at random anywhere in the extent and then checking if they fall within any of the AOIs can be very inefficient. This flag enables the use of an alternate algorithm that only samples window locations inside the AOIs. Defaults to True.
- **transform** – Albumentations transform to apply to the windows. Defaults to None.
- **transform_type** – Type of transform. Defaults to None.
- **normalize** (*bool, optional*) – If True, x is normalized to [0, 1] based on its data type. Defaults to True.
- **to_pytorch** (*bool, optional*) – If True, x and y are converted to pytorch tensors. Defaults to True.

Methods

__init__(scene, out_size[, size_lims, ...])	Constructor.
from_uris(*args, **kwargs)	
get_resize_transform(transform, out_size)	Get transform to use for resizing windows to out_size.
sample_window()	If scene has AOI polygons, try to find a random window that is within the AOI.
sample_window_loc(h, w)	Randomly sample coordinates of the top left corner of the window.
sample_window_size()	Randomly sample the window size.

__init__(*scene: Scene*, *out_size: Optional[Union[PositiveInt, Tuple[PositiveInt, PositiveInt]]]*, *size_lims: Optional[Tuple[PositiveInt, PositiveInt]] = None*, *h_lims: Optional[Tuple[PositiveInt, PositiveInt]] = None*, *w_lims: Optional[Tuple[PositiveInt, PositiveInt]] = None*, *padding: Optional[Union[ConstrainedIntValue, Tuple[ConstrainedIntValue, ConstrainedIntValue]]] = None*, *max_windows: Optional[ConstrainedIntValue] = None*, *max_sample_attempts: PositiveInt = 100*, *return_window: bool = False*, *efficient_aoi_sampling: bool = True*, *transform: Optional[BasicTransform] = None*, *transform_type: Optional[TransformType] = None*, *normalize: bool = True*, *to_pytorch: bool = True*)

Constructor.

Will sample square windows if size_lims is specified. Otherwise, will sample rectangular windows with height and width sampled according to h_lims and w_lims.

Parameters

- **scene** (Scene) – A Scene object.
- **out_size** (*Optional[Union[PosInt, Tuple[PosInt, PosInt]]]*) – Resize windows to this size before returning. This is to aid in collating the windows into a batch. If None, windows are returned without being normalized or converted to pytorch, and will be of different sizes in successive reads.
- **size_lims** (*Optional[Tuple[PosInt, PosInt]]*) – Interval from which to sample window size.
- **h_lims** (*Optional[Tuple[PosInt, PosInt]]*) – Interval from which to sample window height.
- **w_lims** (*Optional[Tuple[PosInt, PosInt]]*) – Interval from which to sample window width.
- **padding** (*Optional[Union[NonNegInt, Tuple[NonNegInt, NonNegInt]]]*) – How many pixels the windows are allowed to overflow the sides of the raster source. If None, padding = size. Defaults to None.
- **max_windows** (*Optional[NonNegInt]*) – Max allowed reads. Will raise StopIteration on further read attempts. If None, will be set to np.inf. Defaults to None.
- **transform** (*Optional[A.BasicTransform], optional*) – Albumentations transform to apply to the windows. Defaults to None. Each transform in Albumentations takes images of type uint8, and sometimes other data types. The data type requirements can be seen at https://albumentations.ai/docs/api_reference/augmentations/transforms/ If there is a mismatch between the data type of imagery and the transform requirements, a RasterTransformer should be set on the RasterSource that converts to uint8, such as MinMaxTransformer or StatsTransformer.
- **transform_type** (*Optional[TransformType], optional*) – Type of transform. Defaults to None.
- **max_sample_attempts** (*NonNegInt, optional*) – Max attempts when trying to find a window within the AOI of the scene. Only used if the scene has aoi_polygons specified. StopIteratioin is raised if this is exceeded. Defaults to 100.
- **return_window** (*bool, optional*) – Make __getitem__ return the window coordinates used to generate the image. Defaults to False.
- **efficient_aoi_sampling** (*bool, optional*) – If the scene has AOIs, sampling windows at random anywhere in the extent and then checking if they fall within any of the AOIs can be very inefficient. This flag enables the use of an alternate algorithm that only samples window locations inside the AOIs. Defaults to True.

- **transform** – Albumentations transform to apply to the windows. Defaults to None.
- **transform_type** – Type of transform. Defaults to None.
- **normalize** (*bool, optional*) – If True, x is normalized to [0, 1] based on its data type. Defaults to True.
- **to_pytorch** (*bool, optional*) – If True, x and y are converted to pytorch tensors. Defaults to True.

static __new__(*cls*, **args: Any*, ***kwargs: Any*) → Any

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
Any

classmethod from_uris(**args*, ***kwargs*) → *GeoDataset*

Return type
GeoDataset

get_resize_transform(*transform: Optional[BasicTransform]*, *out_size: Tuple[PositiveInt, PositiveInt]*) → Union[Resize, Compose]

Get transform to use for resizing windows to out_size.

Parameters

- **transform** (*Optional[BasicTransform]*) –
- **out_size** (*Tuple[PositiveInt, PositiveInt]*) –

Return type
Union[*Resize*, *Compose*]

sample_window() → *Box*

If scene has AOI polygons, try to find a random window that is within the AOI. Otherwise, just return the first sampled window.

Raises
StopIteration – If unable to find a valid window within self.max_sample_attempts attempts.

Returns
The sampled window.

Return type
Box

sample_window_loc(*h: int*, *w: int*) → Tuple[int, int]

Randomly sample coordinates of the top left corner of the window.

Parameters

- **h** (*int*) –
- **w** (*int*) –

Return type
Tuple[int, int]

sample_window_size() → Tuple[int, int]

Randomly sample the window size.

Return type
Tuple[int, int]

property max_size

property min_size

SlidingWindowGeoDataset

class SlidingWindowGeoDataset

Bases: *GeoDataset*

Read the scene left-to-right, top-to-bottom, using a sliding window.

__init__(*scene: Scene*, *size: Union[PositiveInt, Tuple[PositiveInt, PositiveInt]]*, *stride: Union[PositiveInt, Tuple[PositiveInt, PositiveInt]]*, *padding: Optional[Union[ConstrainedIntValue, Tuple[ConstrainedIntValue, ConstrainedIntValue]]] = None*, *pad_direction: Literal['both', 'start', 'end'] = 'end'*, *transform: Optional[BasicTransform] = None*, *transform_type: Optional[TransformType] = None*, *normalize: bool = True*, *to_pytorch: bool = True*)

Constructor.

Parameters

- **scene** (Scene) – A Scene object.
- **size** (*Union[PosInt, Tuple[PosInt, PosInt]]*) – Window size.
- **stride** (*Union[PosInt, Tuple[PosInt, PosInt]]*) – Step size between windows.
- **padding** (*Optional[Union[NonNegInt, Tuple[NonNegInt, NonNegInt]]]*) – How many pixels the windows are allowed to overflow the sides of the raster source. If None, padding is set to size // 2. Defaults to None.
- **pad_direction** (*Literal['both', 'start', 'end']*) – If 'end', only pad ymax and xmax (bottom and right). If 'start', only pad ymin and xmin (top and left). If 'both', pad all sides. Has no effect if paddiong is zero. Defaults to 'end'.
- **transform** (*Optional[A.BasicTransform], optional*) – Albumentations transform to apply to the windows. Defaults to None. Each transform in Albumentations takes images of type uint8, and sometimes other data types. The data type requirements can be seen at https://albumentations.ai/docs/api_reference/augmentations/transforms/ # noqa If there is a mismatch between the data type of imagery and the transform requirements, a RasterTransformer should be set on the RasterSource that converts to uint8, such as MinMaxTransformer or StatsTransformer.
- **transform_type** (*Optional[TransformType], optional*) – Type of transform. Defaults to None.
- **normalize** (*bool, optional*) – If True, x is normalized to [0, 1] based on its data type. Defaults to True.
- **to_pytorch** (*bool, optional*) – If True, x and y are converted to pytorch tensors. Defaults to True.

Methods

__init__(scene, size, stride[, padding, ...])	Constructor.
from_uris(*args, **kwargs)	
init_windows()	Pre-compute windows.

__init__(*scene: Scene*, *size: Union[PositiveInt, Tuple[PositiveInt, PositiveInt]]*, *stride: Union[PositiveInt, Tuple[PositiveInt, PositiveInt]]*, *padding: Optional[Union[ConstrainedIntValue, Tuple[ConstrainedIntValue, ConstrainedIntValue]]] = None*, *pad_direction: Literal['both', 'start', 'end'] = 'end'*, *transform: Optional[BasicTransform] = None*, *transform_type: Optional[TransformType] = None*, *normalize: bool = True*, *to_pytorch: bool = True*)

Constructor.

Parameters

- **scene** (*Scene*) – A Scene object.
- **size** (*Union[PosInt, Tuple[PosInt, PosInt]]*) – Window size.
- **stride** (*Union[PosInt, Tuple[PosInt, PosInt]]*) – Step size between windows.
- **padding** (*Optional[Union[NonNegInt, Tuple[NonNegInt, NonNegInt]]]*) – How many pixels the windows are allowed to overflow the sides of the raster source. If None, padding is set to size // 2. Defaults to None.
- **pad_direction** (*Literal['both', 'start', 'end']*) – If 'end', only pad ymax and xmax (bottom and right). If 'start', only pad ymin and xmin (top and left). If 'both', pad all sides. Has no effect if paddiong is zero. Defaults to 'end'.
- **transform** (*Optional[A.BasicTransform], optional*) – Albumentations transform to apply to the windows. Defaults to None. Each transform in Albumentations takes images of type uint8, and sometimes other data types. The data type requirements can be seen at https://albumentations.ai/docs/api_reference/augmentations/transforms/ # noqa If there is a mismatch between the data type of imagery and the transform requirements, a RasterTransformer should be set on the RasterSource that converts to uint8, such as MinMaxTransformer or StatsTransformer.
- **transform_type** (*Optional[TransformType], optional*) – Type of transform. Defaults to None.
- **normalize** (*bool, optional*) – If True, x is normalized to [0, 1] based on its data type. Defaults to True.
- **to_pytorch** (*bool, optional*) – If True, x and y are converted to pytorch tensors. Defaults to True.

static __new__(*cls*, **args: Any*, ***kwargs: Any*) → Any

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type

Any

classmethod from_uris(**args*, ***kwargs*) → *GeoDataset*

Return type
GeoDataset

init_windows() → None

Pre-compute windows.

Return type
None

object_detection_dataset

Classes

CocoDataset	Read Object Detection data in the COCO format.
ObjectDetectionImageDataset	Read Object Detection data in the COCO format.
ObjectDetectionRandomWindowGeoDataset	
ObjectDetectionSlidingWindowGeoDataset	

CocoDataset

class CocoDataset

Bases: Dataset

Read Object Detection data in the COCO format.

__init__(*img_dir: str*, *annotation_uri: str*)

Constructor.

Parameters

- **img_dir** (*str*) – Directory containing the images. Image filenames must match the image IDs in the annotations file.
- **annotation_uri** (*str*) – URI to a JSON file containing annotations in the COCO format.

Methods

__init__(img_dir, annotation_uri)	Constructor.

__init__(*img_dir: str*, *annotation_uri: str*)

Constructor.

Parameters

- **img_dir** (*str*) – Directory containing the images. Image filenames must match the image IDs in the annotations file.
- **annotation_uri** (*str*) – URI to a JSON file containing annotations in the COCO format.

static __new__(*cls*, **args: Any*, ***kwargs: Any*) → Any

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
Any

ObjectDetectionImageDataset

class ObjectDetectionImageDataset

Bases: *ImageDataset*

Read Object Detection data in the COCO format.

Uses *CocoDataset* to read the data.

__init__(*img_dir: str*, *annotation_uri: str*, **args*, ***kwargs*)

Constructor.

Parameters

- **img_dir** (*str*) – Directory containing the images. Image filenames must match the image IDs in the annotations file.
- **annotation_uri** (*str*) – URI to a JSON file containing annotations in the COCO format.
- ***args** – See *ImageDataset.__init__()*.
- ****kwargs** – See *ImageDataset.__init__()*.

Methods

__init__(img_dir, annotation_uri, *args, ...)	Constructor.

__init__(*img_dir: str*, *annotation_uri: str*, **args*, ***kwargs*)

Constructor.

Parameters

- **img_dir** (*str*) – Directory containing the images. Image filenames must match the image IDs in the annotations file.
- **annotation_uri** (*str*) – URI to a JSON file containing annotations in the COCO format.
- ***args** – See *ImageDataset.__init__()*.
- ****kwargs** – See *ImageDataset.__init__()*.

static __new__(*cls*, **args: Any*, ***kwargs: Any*) → Any

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
Any

ObjectDetectionRandomWindowGeoDataset

class ObjectDetectionRandomWindowGeoDataset

Bases: *RandomWindowGeoDataset*

Attributes

max_size
min_size

__init__(**args*, ***kwargs*)

Constructor.

Parameters
***args** – See *RandomWindowGeoDataset.__init__()*.

Keyword Arguments

- **bbox_params** (*Optional[A.BboxParams], optional*) – Optional bbox_params to use when resizing windows. Defaults to None.
- **ioa_thresh** (*float, optional*) – Minimum IoA of a bounding box with a given window for it to be included in the labels for that window. Defaults to 0.9.
- **clip** (*bool, optional*) – Clip bounding boxes to window limits when retrieving labels for a window. Defaults to False.
- **neg_ratio** (*Optional[float], optional*) – Ratio of sampling probabilities of negative windows (windows w/o bboxes) vs positive windows (windows w/ at least 1 bbox). E.g. neg_ratio=2 means 2/3 probability of sampling a negative window. If None, the default sampling behavior of RandomWindowGeoDataset is used, without taking bboxes into account. Defaults to None.
- **neg_ioa_thresh** (*float, optional*) – A window will be considered negative if its max IoA with any bounding box is less than this threshold. Defaults to 0.2.
- ****kwargs** – See *RandomWindowGeoDataset.__init__()*.

Methods

__init__(*args, **kwargs)	Constructor.
from_uris(image_uri[, label_vector_uri, ...])	Create an instance of this class from image and label URIs.
get_resize_transform(transform, out_size)	Get transform to use for resizing windows to out_size.
sample_window()	If scene has AOI polygons, try to find a random window that is within the AOI.
sample_window_loc(h, w)	Randomly sample coordinates of the top left corner of the window.
sample_window_size()	Randomly sample the window size.

__init__(*args, **kwargs)

Constructor.

Parameters

***args** – See RandomWindowGeoDataset.__init__().

Keyword Arguments

- **bbox_params** (Optional[A.BboxParams], optional) – Optional bbox_params to use when resizing windows. Defaults to None.
- **ioa_thresh** (float, optional) – Minimum IoA of a bounding box with a given window for it to be included in the labels for that window. Defaults to 0.9.
- **clip** (bool, optional) – Clip bounding boxes to window limits when retrieving labels for a window. Defaults to False.
- **neg_ratio** (Optional[float], optional) – Ratio of sampling probabilities of negative windows (windows w/o bboxes) vs positive windows (windows w/ at least 1 bbox). E.g. neg_ratio=2 means 2/3 probability of sampling a negative window. If None, the default sampling behavior of RandomWindowGeoDataset is used, without taking bboxes into account. Defaults to None.
- **neg_ioa_thresh** (float, optional) – A window will be considered negative if its max IoA with any bounding box is less than this threshold. Defaults to 0.2.
- ****kwargs** – See RandomWindowGeoDataset.__init__().

static __new__(*cls*, **args: Any*, ***kwargs: Any*) → Any

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type

Any

classmethod from_uris(*image_uri: Union[str, List[str]]*, *label_vector_uri: Optional[str] = None*, *class_config: Optional[ClassConfig] = None*, *aoi_uri: Union[str, List[str]] = []*, *label_vector_default_class_id: Optional[int] = None*, *image_raster_source_kw: dict = {}*, *label_vector_source_kw: dict = {}*, *label_source_kw: dict = {}*, ***kwargs*)

Create an instance of this class from image and label URIs.

This is a convenience method. For more fine-grained control, it is recommended to use the default constructor.

Parameters

- **image_uri** (Union[str, List[str]]) – URI or list of URIs of GeoTIFFs to use as the source of image data.
- **label_vector_uri** (Optional[str], optional) – URI of GeoJSON file to use as the source of segmentation label data. Defaults to None.
- **class_config** (Optional['ClassConfig']) – The ClassConfig. Can be None if not using any labels.
- **aoi_uri** (Union[str, List[str]], optional) – URI or list of URIs of GeoJSONs that specify the area-of-interest. If provided, the dataset will only access data from this area. Defaults to [].

- **label_vector_default_class_id** (*Optional[int], optional*) – If using label_vector_uri and all polygons in that file belong to the same class and they do not contain a *class_id* property, then use this argument to map all of the polgons to the appropriate class ID. See docs for ClassInferenceTransformer for more details. Defaults to None.
- **image_raster_source_kw** (*dict, optional*) – Additional arguments to pass to the RasterioSource used for image data. See docs for RasterioSource for more details. Defaults to {}.
- **label_vector_source_kw** (*dict, optional*) – Additional arguments to pass to the GeoJSONVectorSourceConfig used for label data, if label_vector_uri is set. See docs for GeoJSONVectorSourceConfig for more details. Defaults to {}.
- **label_source_kw** (*dict, optional*) – Additional arguments to pass to the ObjectDetectionLabelSourceConfig used for label data, if label_vector_uri is set. See docs for ObjectDetectionLabelSourceConfig for more details. Defaults to {}.
- ****kwargs** – All other keyword args are passed to the default constructor for this class.

Returns
An instance of this GeoDataset subclass.

get_resize_transform(*transform*, *out_size*)

Get transform to use for resizing windows to out_size.

sample_window() → *Box*

If scene has AOI polygons, try to find a random window that is within the AOI. Otherwise, just return the first sampled window.

Raises
StopIteration – If unable to find a valid window within self.max_sample_attempts attempts.

Returns
The sampled window.

Return type
Box

sample_window_loc(*h: int*, *w: int*) → Tuple[int, int]

Randomly sample coordinates of the top left corner of the window.

Parameters

- **h** (*int*) –
- **w** (*int*) –

Return type
Tuple[int, int]

sample_window_size() → Tuple[int, int]

Randomly sample the window size.

Return type
Tuple[int, int]

property max_size

property min_size

ObjectDetectionSlidingWindowGeoDataset

class ObjectDetectionSlidingWindowGeoDataset

Bases: *SlidingWindowGeoDataset*

__init__(**args*, ***kwargs*)

Constructor.

Parameters

- **scene** (Scene) – A Scene object.
- **size** (*Union[PosInt, Tuple[PosInt, PosInt]]*) – Window size.
- **stride** (*Union[PosInt, Tuple[PosInt, PosInt]]*) – Step size between windows.
- **padding** (*Optional[Union[NonNegInt, Tuple[NonNegInt, NonNegInt]]]*) – How many pixels the windows are allowed to overflow the sides of the raster source. If None, padding is set to size // 2. Defaults to None.
- **pad_direction** (*Literal['both', 'start', 'end']*) – If 'end', only pad ymax and xmax (bottom and right). If 'start', only pad ymin and xmin (top and left). If 'both', pad all sides. Has no effect if paddiong is zero. Defaults to 'end'.
- **transform** (*Optional[A.BasicTransform], optional*) – Albumentations transform to apply to the windows. Defaults to None. Each transform in Albumentations takes images of type uint8, and sometimes other data types. The data type requirements can be seen at https://albumentations.ai/docs/api_reference/augmentations/transforms/ # noqa If there is a mismatch between the data type of imagery and the transform requirements, a RasterTransformer should be set on the RasterSource that converts to uint8, such as MinMaxTransformer or StatsTransformer.
- **transform_type** (*Optional[TransformType], optional*) – Type of transform. Defaults to None.
- **normalize** (*bool, optional*) – If True, x is normalized to [0, 1] based on its data type. Defaults to True.
- **to_pytorch** (*bool, optional*) – If True, x and y are converted to pytorch tensors. Defaults to True.

Methods

__init__(*args, **kwargs)	Constructor.
from_uris(image_uri[, label_vector_uri, ...])	Create an instance of this class from image and label URIs.
init_windows()	Pre-compute windows.

__init__(**args*, ***kwargs*)

Constructor.

Parameters

- **scene** (Scene) – A Scene object.
- **size** (*Union[PosInt, Tuple[PosInt, PosInt]]*) – Window size.
- **stride** (*Union[PosInt, Tuple[PosInt, PosInt]]*) – Step size between windows.

- **padding** (*Optional[Union[NonNegInt, Tuple[NonNegInt, NonNegInt]]]*) – How many pixels the windows are allowed to overflow the sides of the raster source. If None, padding is set to size // 2. Defaults to None.
- **pad_direction** (*Literal['both', 'start', 'end']*) – If 'end', only pad ymax and xmax (bottom and right). If 'start', only pad ymin and xmin (top and left). If 'both', pad all sides. Has no effect if paddiong is zero. Defaults to 'end'.
- **transform** (*Optional[A.BasicTransform], optional*) – Albumentations transform to apply to the windows. Defaults to None. Each transform in Albumentations takes images of type uint8, and sometimes other data types. The data type requirements can be seen at https://albumentations.ai/docs/api_reference/augmentations/transforms/ # noqa If there is a mismatch between the data type of imagery and the transform requirements, a RasterTransformer should be set on the RasterSource that converts to uint8, such as MinMaxTransformer or StatsTransformer.
- **transform_type** (*Optional[TransformType], optional*) – Type of transform. Defaults to None.
- **normalize** (*bool, optional*) – If True, x is normalized to [0, 1] based on its data type. Defaults to True.
- **to_pytorch** (*bool, optional*) – If True, x and y are converted to pytorch tensors. Defaults to True.

static **__new__**(*cls*, **args: Any*, ***kwargs: Any*) → Any

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
Any

classmethod **from_uris**(*image_uri: Union[str, List[str]]*, *label_vector_uri: Optional[str] = None*, *class_config: Optional[ClassConfig] = None*, *aoi_uri: Union[str, List[str]] = []*, *label_vector_default_class_id: Optional[int] = None*, *image_raster_source_kw: dict = {}*, *label_vector_source_kw: dict = {}*, *label_source_kw: dict = {}*, ***kwargs*)

Create an instance of this class from image and label URIs.

This is a convenience method. For more fine-grained control, it is recommended to use the default constructor.

Parameters

- **image_uri** (*Union[str, List[str]]*) – URI or list of URIs of GeoTIFFs to use as the source of image data.
- **label_vector_uri** (*Optional[str], optional*) – URI of GeoJSON file to use as the source of segmentation label data. Defaults to None.
- **class_config** (*Optional['ClassConfig']*) – The ClassConfig. Can be None if not using any labels.
- **aoi_uri** (*Union[str, List[str]], optional*) – URI or list of URIs of GeoJSONs that specify the area-of-interest. If provided, the dataset will only access data from this area. Defaults to [].

- **label_vector_default_class_id** (*Optional[int], optional*) – If using label_vector_uri and all polygons in that file belong to the same class and they do not contain a *class_id* property, then use this argument to map all of the polgons to the appropriate class ID. See docs for ClassInferenceTransformer for more details. Defaults to None.
- **image_raster_source_kw** (*dict, optional*) – Additional arguments to pass to the RasterioSource used for image data. See docs for RasterioSource for more details. Defaults to {}.
- **label_vector_source_kw** (*dict, optional*) – Additional arguments to pass to the GeoJSONVectorSourceConfig used for label data, if label_vector_uri is set. See docs for GeoJSONVectorSourceConfig for more details. Defaults to {}.
- **label_source_kw** (*dict, optional*) – Additional arguments to pass to the ObjectDetectionLabelSourceConfig used for label data, if label_vector_uri is set. See docs for ObjectDetectionLabelSourceConfig for more details. Defaults to {}.
- ****kwargs** – All other keyword args are passed to the default constructor for this class.

Returns
An instance of this GeoDataset subclass.

init_windows() → None

Pre-compute windows.

Return type
None

Functions

make_od_geodataset(cls, image_uri[, ...])	Create an instance of this class from image and label URIs.

make_od_geodataset

make_od_geodataset(*cls*, *image_uri: Union[str, List[str]]*, *label_vector_uri: Optional[str] = None*, *class_config: Optional[*ClassConfig*] = None*, *aoi_uri: Union[str, List[str]] = []*, *label_vector_default_class_id: Optional[int] = None*, *image_raster_source_kw: dict = {}*, *label_vector_source_kw: dict = {}*, *label_source_kw: dict = {}*, ***kwargs*)

Create an instance of this class from image and label URIs.

This is a convenience method. For more fine-grained control, it is recommended to use the default constructor.

Parameters

- **image_uri** (*Union[str, List[str]]*) – URI or list of URIs of GeoTIFFs to use as the source of image data.
- **label_vector_uri** (*Optional[str], optional*) – URI of GeoJSON file to use as the source of segmentation label data. Defaults to None.
- **class_config** (*Optional['ClassConfig']*) – The ClassConfig. Can be None if not using any labels.
- **aoi_uri** (*Union[str, List[str]], optional*) – URI or list of URIs of GeoJSONs that specify the area-of-interest. If provided, the dataset will only access data from this area. Defaults to [].

- **label_vector_default_class_id** (*Optional[int], optional*) – If using label_vector_uri and all polygons in that file belong to the same class and they do not contain a *class_id* property, then use this argument to map all of the polgons to the appropriate class ID. See docs for ClassInferenceTransformer for more details. Defaults to None.
- **image_raster_source_kw** (*dict, optional*) – Additional arguments to pass to the RasterioSource used for image data. See docs for RasterioSource for more details. Defaults to {}.
- **label_vector_source_kw** (*dict, optional*) – Additional arguments to pass to the GeoJSONVectorSourceConfig used for label data, if label_vector_uri is set. See docs for GeoJSONVectorSourceConfig for more details. Defaults to {}.
- **label_source_kw** (*dict, optional*) – Additional arguments to pass to the ObjectDetectionLabelSourceConfig used for label data, if label_vector_uri is set. See docs for ObjectDetectionLabelSourceConfig for more details. Defaults to {}.
- ****kwargs** – All other keyword args are passed to the default constructor for this class.

Returns
An instance of this GeoDataset subclass.

regression_dataset

Classes

RegressionDataReader
RegressionImageDataset
RegressionRandomWindowGeoDataset
RegressionSlidingWindowGeoDataset

RegressionDataReader

class RegressionDataReader

Bases: Dataset

__init__(*data_dir: str*, *class_names: Iterable[str]*)

Parameters

- **data_dir** (*str*) –
- **class_names** (*Iterable[str]*) –

Methods

__init__(data_dir, class_names)

__init__(*data_dir: str*, *class_names: Iterable[str]*)

Parameters

- **data_dir** (*str*) –
- **class_names** (*Iterable[str]*) –

static __new__(*cls*, **args: Any*, ***kwargs: Any*) → Any

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
Any

RegressionImageDataset

class RegressionImageDataset

Bases: *ImageDataset*

__init__(*data_dir: str*, *class_names: Iterable[str]*, **args*, ***kwargs*)

Constructor.

Parameters

- **orig_dataset** (*Any*) – An object with a __getitem__ and __len__.
- **transform** (*A.BasicTransform, optional*) – Albumentations transform to apply to the windows. Defaults to None. Each transform in Albumentations takes images of type uint8, and sometimes other data types. The data type requirements can be seen at https://albumentations.ai/docs/api_reference/augmentations/transforms/ # noqa If there is a mismatch between the data type of imagery and the transform requirements, a RasterTransformer should be set on the RasterSource that converts to uint8, such as MinMaxTransformer or StatsTransformer.
- **transform_type** (*TransformType*) – The type of transform so that its inputs and outputs can be handled correctly. Defaults to TransformType.noop.
- **normalize** (*bool, optional*) – If True, x is normalized to [0, 1] based on its data type. Defaults to True.
- **to_pytorch** (*bool, optional*) – If True, x and y are converted to pytorch tensors. Defaults to True.
- **data_dir** (*str*) –
- **class_names** (*Iterable[str]*) –

Methods

__init__(data_dir, class_names, *args, **kwargs)	Constructor.

__init__(*data_dir: str*, *class_names: Iterable[str]*, **args*, ***kwargs*)

Constructor.

Parameters

- **orig_dataset** (*Any*) – An object with a __getitem__ and __len__.
- **transform** (*A.BasicTransform, optional*) – Albumentations transform to apply to the windows. Defaults to None. Each transform in Albumentations takes images of type uint8, and sometimes other data types. The data type requirements can be seen at https://albumentations.ai/docs/api_reference/augmentations/transforms/ # noqa If there is a mismatch between the data type of imagery and the transform requirements, a RasterTransformer should be set on the RasterSource that converts to uint8, such as MinMaxTransformer or StatsTransformer.
- **transform_type** (TransformType) – The type of transform so that its inputs and outputs can be handled correctly. Defaults to TransformType.noop.
- **normalize** (*bool, optional*) – If True, x is normalized to [0, 1] based on its data type. Defaults to True.
- **to_pytorch** (*bool, optional*) – If True, x and y are converted to pytorch tensors. Defaults to True.
- **data_dir** (*str*) –
- **class_names** (*Iterable[str]*) –

static __new__(*cls*, **args: Any*, ***kwargs: Any*) → Any

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type

Any

RegressionRandomWindowGeoDataset

class RegressionRandomWindowGeoDataset

Bases: *RandomWindowGeoDataset*

Attributes

max_size

min_size

__init__(**args*, ***kwargs*)

Constructor.

Will sample square windows if size_lims is specified. Otherwise, will sample rectangular windows with height and width sampled according to h_lims and w_lims.

Parameters

- **scene** (Scene) – A Scene object.
- **out_size** (*Optional[Union[PosInt, Tuple[PosInt, PosInt]]]*) – Resize windows to this size before returning. This is to aid in collating the windows into a batch. If None, windows are returned without being normalized or converted to pytorch, and will be of different sizes in successive reads.
- **size_lims** (*Optional[Tuple[PosInt, PosInt]]*) – Interval from which to sample window size.
- **h_lims** (*Optional[Tuple[PosInt, PosInt]]*) – Interval from which to sample window height.
- **w_lims** (*Optional[Tuple[PosInt, PosInt]]*) – Interval from which to sample window width.
- **padding** (*Optional[Union[NonNegInt, Tuple[NonNegInt, NonNegInt]]]*) – How many pixels the windows are allowed to overflow the sides of the raster source. If None, padding = size. Defaults to None.
- **max_windows** (*Optional[NonNegInt]*) – Max allowed reads. Will raise StopIteration on further read attempts. If None, will be set to np.inf. Defaults to None.
- **transform** (*Optional[A.BasicTransform], optional*) – Albumentations transform to apply to the windows. Defaults to None. Each transform in Albumentations takes images of type uint8, and sometimes other data types. The data type requirements can be seen at https://albumentations.ai/docs/api_reference/augmentations/transforms/ If there is a mismatch between the data type of imagery and the transform requirements, a RasterTransformer should be set on the RasterSource that converts to uint8, such as MinMaxTransformer or StatsTransformer.
- **transform_type** (*Optional[TransformType], optional*) – Type of transform. Defaults to None.
- **max_sample_attempts** (*NonNegInt, optional*) – Max attempts when trying to find a window within the AOI of the scene. Only used if the scene has aoi_polygons specified. StopIteratioin is raised if this is exceeded. Defaults to 100.
- **return_window** (*bool, optional*) – Make __getitem__ return the window coordinates used to generate the image. Defaults to False.
- **efficient_aoi_sampling** (*bool, optional*) – If the scene has AOIs, sampling windows at random anywhere in the extent and then checking if they fall within any of the AOIs can be very inefficient. This flag enables the use of an alternate algorithm that only samples window locations inside the AOIs. Defaults to True.

- **transform** – Albumentations transform to apply to the windows. Defaults to None.
- **transform_type** – Type of transform. Defaults to None.
- **normalize** (*bool, optional*) – If True, x is normalized to [0, 1] based on its data type. Defaults to True.
- **to_pytorch** (*bool, optional*) – If True, x and y are converted to pytorch tensors. Defaults to True.

Methods

__init__(*args, **kwargs)	Constructor.
from_uris(*args, **kwargs)	
get_resize_transform(transform, out_size)	Get transform to use for resizing windows to out_size.
sample_window()	If scene has AOI polygons, try to find a random window that is within the AOI.
sample_window_loc(h, w)	Randomly sample coordinates of the top left corner of the window.
sample_window_size()	Randomly sample the window size.

__init__(**args*, ***kwargs*)

Constructor.

Will sample square windows if size_lims is specified. Otherwise, will sample rectangular windows with height and width sampled according to h_lims and w_lims.

Parameters

- **scene** (Scene) – A Scene object.
- **out_size** (*Optional[Union[PosInt, Tuple[PosInt, PosInt]]]*) – Resize windows to this size before returning. This is to aid in collating the windows into a batch. If None, windows are returned without being normalized or converted to pytorch, and will be of different sizes in successive reads.
- **size_lims** (*Optional[Tuple[PosInt, PosInt]]*) – Interval from which to sample window size.
- **h_lims** (*Optional[Tuple[PosInt, PosInt]]*) – Interval from which to sample window height.
- **w_lims** (*Optional[Tuple[PosInt, PosInt]]*) – Interval from which to sample window width.
- **padding** (*Optional[Union[NonNegInt, Tuple[NonNegInt, NonNegInt]]]*) – How many pixels the windows are allowed to overflow the sides of the raster source. If None, padding = size. Defaults to None.
- **max_windows** (*Optional[NonNegInt]*) – Max allowed reads. Will raise StopIteration on further read attempts. If None, will be set to np.inf. Defaults to None.
- **transform** (*Optional[A.BasicTransform], optional*) – Albumentations transform to apply to the windows. Defaults to None. Each transform in Albumentations takes images of type uint8, and sometimes other data types. The data type requirements can be seen at https://albumentations.ai/docs/api_reference/augmentations/transforms/ If there is

a mismatch between the data type of imagery and the transform requirements, a RasterTransformer should be set on the RasterSource that converts to uint8, such as MinMaxTransformer or StatsTransformer.

- **transform_type** (*Optional[TransformType], optional*) – Type of transform. Defaults to None.
- **max_sample_attempts** (*NonNegInt, optional*) – Max attempts when trying to find a window within the AOI of the scene. Only used if the scene has aoi_polygons specified. StopIteratioin is raised if this is exceeded. Defaults to 100.
- **return_window** (*bool, optional*) – Make __getitem__ return the window coordinates used to generate the image. Defaults to False.
- **efficient_aoi_sampling** (*bool, optional*) – If the scene has AOIs, sampling windows at random anywhere in the extent and then checking if they fall within any of the AOIs can be very inefficient. This flag enables the use of an alternate algorithm that only samples window locations inside the AOIs. Defaults to True.
- **transform** – Albumentations transform to apply to the windows. Defaults to None.
- **transform_type** – Type of transform. Defaults to None.
- **normalize** (*bool, optional*) – If True, x is normalized to [0, 1] based on its data type. Defaults to True.
- **to_pytorch** (*bool, optional*) – If True, x and y are converted to pytorch tensors. Defaults to True.

static __new__(*cls*, **args: Any*, ***kwargs: Any*) → Any

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
Any

classmethod from_uris(**args*, ***kwargs*) → *GeoDataset*

Return type
GeoDataset

get_resize_transform(*transform: Optional[BasicTransform]*, *out_size: Tuple[PositiveInt, PositiveInt]*) → Union[Resize, Compose]

Get transform to use for resizing windows to out_size.

Parameters

- **transform** (*Optional[BasicTransform]*) –
- **out_size** (*Tuple[PositiveInt, PositiveInt]*) –

Return type
Union[*Resize*, *Compose*]

sample_window() → *Box*

If scene has AOI polygons, try to find a random window that is within the AOI. Otherwise, just return the first sampled window.

Raises
StopIteration – If unable to find a valid window within self.max_sample_attempts attempts.

Returns
The sampled window.

Return type
Box

`sample_window_loc`(*h: int*, *w: int*) → Tuple[int, int]

Randomly sample coordinates of the top left corner of the window.

Parameters

- **h** (*int*) –
- **w** (*int*) –

Return type
Tuple[int, int]

`sample_window_size`() → Tuple[int, int]

Randomly sample the window size.

Return type
Tuple[int, int]

`property max_size`

`property min_size`

RegressionSlidingWindowGeoDataset

`class RegressionSlidingWindowGeoDataset`

Bases: *SlidingWindowGeoDataset*

`__init__`(**args*, ***kwargs*)

Constructor.

Parameters

- **scene** (Scene) – A Scene object.
- **size** (*Union[PosInt, Tuple[PosInt, PosInt]]*) – Window size.
- **stride** (*Union[PosInt, Tuple[PosInt, PosInt]]*) – Step size between windows.
- **padding** (*Optional[Union[NonNegInt, Tuple[NonNegInt, NonNegInt]]]*) – How many pixels the windows are allowed to overflow the sides of the raster source. If None, padding is set to size // 2. Defaults to None.
- **pad_direction** (*Literal['both', 'start', 'end']*) – If 'end', only pad ymax and xmax (bottom and right). If 'start', only pad ymin and xmin (top and left). If 'both', pad all sides. Has no effect if paddiong is zero. Defaults to 'end'.
- **transform** (*Optional[A.BasicTransform], optional*) – Albumentations transform to apply to the windows. Defaults to None. Each transform in Albumentations takes images of type uint8, and sometimes other data types. The data type requirements can be seen at https://albumentations.ai/docs/api_reference/augmentations/transforms/ # noqa If there is a mismatch between the data type of imagery and the transform requirements, a

RasterTransformer should be set on the RasterSource that converts to uint8, such as MinMaxTransformer or StatsTransformer.

- **transform_type** (*Optional[TransformType], optional*) – Type of transform. Defaults to None.
- **normalize** (*bool, optional*) – If True, x is normalized to [0, 1] based on its data type. Defaults to True.
- **to_pytorch** (*bool, optional*) – If True, x and y are converted to pytorch tensors. Defaults to True.

Methods

__init__(*args, **kwargs)	Constructor.
from_uris(*args, **kwargs)	
init_windows()	Pre-compute windows.

__init__(**args*, ***kwargs*)

Constructor.

Parameters

- **scene** (Scene) – A Scene object.
- **size** (*Union[PosInt, Tuple[PosInt, PosInt]]*) – Window size.
- **stride** (*Union[PosInt, Tuple[PosInt, PosInt]]*) – Step size between windows.
- **padding** (*Optional[Union[NonNegInt, Tuple[NonNegInt, NonNegInt]]]*) – How many pixels the windows are allowed to overflow the sides of the raster source. If None, padding is set to size // 2. Defaults to None.
- **pad_direction** (*Literal['both', 'start', 'end']*) – If 'end', only pad ymax and xmax (bottom and right). If 'start', only pad ymin and xmin (top and left). If 'both', pad all sides. Has no effect if paddiong is zero. Defaults to 'end'.
- **transform** (*Optional[A.BasicTransform], optional*) – Albumentations transform to apply to the windows. Defaults to None. Each transform in Albumentations takes images of type uint8, and sometimes other data types. The data type requirements can be seen at https://albumentations.ai/docs/api_reference/augmentations/transforms/ # noqa If there is a mismatch between the data type of imagery and the transform requirements, a RasterTransformer should be set on the RasterSource that converts to uint8, such as MinMaxTransformer or StatsTransformer.
- **transform_type** (*Optional[TransformType], optional*) – Type of transform. Defaults to None.
- **normalize** (*bool, optional*) – If True, x is normalized to [0, 1] based on its data type. Defaults to True.
- **to_pytorch** (*bool, optional*) – If True, x and y are converted to pytorch tensors. Defaults to True.

static __new__(*cls*, **args: Any*, ***kwargs: Any*) → Any

Parameters

- **args** (*Any*) –

- **kwargs** (*Any*) –

Return type
Any

classmethod from_uris(**args*, ***kwargs*) → *GeoDataset*

Return type
GeoDataset

init_windows() → None

Pre-compute windows.

Return type
None

semantic_segmentation_dataset

Classes

SemanticSegmentationDataReader	Reads semantic segmentatioin images and labels from files.
SemanticSegmentationImageDataset	Reads semantic segmentatioin images and labels from files.
SemanticSegmentationRandomWindowGeoDataset	
SemanticSegmentationSlidingWindowGeoDataset	

SemanticSegmentationDataReader

class SemanticSegmentationDataReader

Bases: Dataset

Reads semantic segmentatioin images and labels from files.

__init__(*img_dir: str*, *label_dir: str*)

Constructor.

Parameters

- **img_dir** (*str*) – Directory containing images.
- **label_dir** (*str*) – Directory containing segmentation masks.

Methods

__init__(img_dir, label_dir)	Constructor.
validate_paths()	

__init__(*img_dir: str*, *label_dir: str*)

Constructor.

Parameters

- **img_dir** (*str*) – Directory containing images.
- **label_dir** (*str*) – Directory containing segmentation masks.

static __new__(*cls*, **args: Any*, ***kwargs: Any*) → Any

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
Any

validate_paths() → None

Return type
None

SemanticSegmentationImageDataset

class SemanticSegmentationImageDataset

Bases: *ImageDataset*

Reads semantic segmentatioin images and labels from files.

Uses *SemanticSegmentationDataReader* to read the data.

__init__(*img_dir: str*, *label_dir: str*, **args*, ***kwargs*)

Constructor.

Parameters

- **img_dir** (*str*) – Directory containing images.
- **label_dir** (*str*) – Directory containing segmentation masks.
- ***args** – See *ImageDataset.__init__()*.
- ****kwargs** – See *ImageDataset.__init__()*.

Methods

__init__(img_dir, label_dir, *args, **kwargs)	Constructor.

__init__(*img_dir: str*, *label_dir: str*, **args*, ***kwargs*)

Constructor.

Parameters

- **img_dir** (*str*) – Directory containing images.
- **label_dir** (*str*) – Directory containing segmentation masks.
- ***args** – See *ImageDataset.__init__()*.
- ****kwargs** – See *ImageDataset.__init__()*.

static __new__(*cls*, **args: Any*, ***kwargs: Any*) → Any

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
Any

SemanticSegmentationRandomWindowGeoDataset

class SemanticSegmentationRandomWindowGeoDataset

Bases: *RandomWindowGeoDataset*

Attributes

max_size
min_size

__init__(**args*, ***kwargs*)

Constructor.

Will sample square windows if size_lims is specified. Otherwise, will sample rectangular windows with height and width sampled according to h_lims and w_lims.

Parameters

- **scene** (Scene) – A Scene object.
- **out_size** (*Optional[Union[PosInt, Tuple[PosInt, PosInt]]]*) – Resize windows to this size before returning. This is to aid in collating the windows into a batch. If None, windows are returned without being normalized or converted to pytorch, and will be of different sizes in successive reads.
- **size_lims** (*Optional[Tuple[PosInt, PosInt]]*) – Interval from which to sample window size.

- **h_lims** (*Optional[Tuple[PosInt, PosInt]]*) – Interval from which to sample window height.
- **w_lims** (*Optional[Tuple[PosInt, PosInt]]*) – Interval from which to sample window width.
- **padding** (*Optional[Union[NonNegInt, Tuple[NonNegInt, NonNegInt]]]*) – How many pixels the windows are allowed to overflow the sides of the raster source. If None, padding = size. Defaults to None.
- **max_windows** (*Optional[NonNegInt]*) – Max allowed reads. Will raise StopIteration on further read attempts. If None, will be set to np.inf. Defaults to None.
- **transform** (*Optional[A.BasicTransform], optional*) – Albumentations transform to apply to the windows. Defaults to None. Each transform in Albumentations takes images of type uint8, and sometimes other data types. The data type requirements can be seen at https://albumentations.ai/docs/api_reference/augmentations/transforms/ If there is a mismatch between the data type of imagery and the transform requirements, a RasterTransformer should be set on the RasterSource that converts to uint8, such as MinMaxTransformer or StatsTransformer.
- **transform_type** (*Optional[TransformType], optional*) – Type of transform. Defaults to None.
- **max_sample_attempts** (*NonNegInt, optional*) – Max attempts when trying to find a window within the AOI of the scene. Only used if the scene has aoi_polygons specified. StopIteratioin is raised if this is exceeded. Defaults to 100.
- **return_window** (*bool, optional*) – Make __getitem__ return the window coordinates used to generate the image. Defaults to False.
- **efficient_aoi_sampling** (*bool, optional*) – If the scene has AOIs, sampling windows at random anywhere in the extent and then checking if they fall within any of the AOIs can be very inefficient. This flag enables the use of an alternate algorithm that only samples window locations inside the AOIs. Defaults to True.
- **transform** – Albumentations transform to apply to the windows. Defaults to None.
- **transform_type** – Type of transform. Defaults to None.
- **normalize** (*bool, optional*) – If True, x is normalized to [0, 1] based on its data type. Defaults to True.
- **to_pytorch** (*bool, optional*) – If True, x and y are converted to pytorch tensors. Defaults to True.

Methods

__init__(*args, **kwargs)	Constructor.
from_uris(image_uri[, label_raster_uri, ...])	Create an instance of this class from image and label URIs.
get_resize_transform(transform, out_size)	Get transform to use for resizing windows to out_size.
sample_window()	If scene has AOI polygons, try to find a random window that is within the AOI.
sample_window_loc(h, w)	Randomly sample coordinates of the top left corner of the window.
sample_window_size()	Randomly sample the window size.

__init__(**args*, ***kwargs*)

Constructor.

Will sample square windows if size_lims is specified. Otherwise, will sample rectangular windows with height and width sampled according to h_lims and w_lims.

Parameters

- **scene** (Scene) – A Scene object.
- **out_size** (*Optional[Union[PosInt, Tuple[PosInt, PosInt]]]*) – Resize windows to this size before returning. This is to aid in collating the windows into a batch. If None, windows are returned without being normalized or converted to pytorch, and will be of different sizes in successive reads.
- **size_lims** (*Optional[Tuple[PosInt, PosInt]]*) – Interval from which to sample window size.
- **h_lims** (*Optional[Tuple[PosInt, PosInt]]*) – Interval from which to sample window height.
- **w_lims** (*Optional[Tuple[PosInt, PosInt]]*) – Interval from which to sample window width.
- **padding** (*Optional[Union[NonNegInt, Tuple[NonNegInt, NonNegInt]]]*) – How many pixels the windows are allowed to overflow the sides of the raster source. If None, padding = size. Defaults to None.
- **max_windows** (*Optional[NonNegInt]*) – Max allowed reads. Will raise StopIteration on further read attempts. If None, will be set to np.inf. Defaults to None.
- **transform** (*Optional[A.BasicTransform], optional*) – Albumentations transform to apply to the windows. Defaults to None. Each transform in Albumentations takes images of type uint8, and sometimes other data types. The data type requirements can be seen at https://albumentations.ai/docs/api_reference/augmentations/transforms/ If there is a mismatch between the data type of imagery and the transform requirements, a RasterTransformer should be set on the RasterSource that converts to uint8, such as MinMaxTransformer or StatsTransformer.
- **transform_type** (*Optional[TransformType], optional*) – Type of transform. Defaults to None.
- **max_sample_attempts** (*NonNegInt, optional*) – Max attempts when trying to find a window within the AOI of the scene. Only used if the scene has aoi_polygons specified. StopIteratioin is raised if this is exceeded. Defaults to 100.
- **return_window** (*bool, optional*) – Make __getitem__ return the window coordinates used to generate the image. Defaults to False.
- **efficient_aoi_sampling** (*bool, optional*) – If the scene has AOIs, sampling windows at random anywhere in the extent and then checking if they fall within any of the AOIs can be very inefficient. This flag enables the use of an alternate algorithm that only samples window locations inside the AOIs. Defaults to True.
- **transform** – Albumentations transform to apply to the windows. Defaults to None.
- **transform_type** – Type of transform. Defaults to None.
- **normalize** (*bool, optional*) – If True, x is normalized to [0, 1] based on its data type. Defaults to True.
- **to_pytorch** (*bool, optional*) – If True, x and y are converted to pytorch tensors. Defaults to True.

static __new__(*cls*, **args: Any*, ***kwargs: Any*) → Any

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
Any

classmethod from_uris(*image_uri: Union[str, List[str]]*, *label_raster_uri: Optional[Union[str, List[str]]] = None*, *label_vector_uri: Optional[str] = None*, *class_config: Optional[ClassConfig] = None*, *aoi_uri: Union[str, List[str]] = []*, *label_vector_default_class_id: Optional[int] = None*, *image_raster_source_kw: dict = {}*, *label_raster_source_kw: dict = {}*, *label_vector_source_kw: dict = {}*, ***kwargs*)

Create an instance of this class from image and label URIs.

This is a convenience method. For more fine-grained control, it is recommended to use the default constructor.

Parameters

- **image_uri** (*Union[str, List[str]]*) – URI or list of URIs of GeoTIFFs to use as the source of image data.
- **label_raster_uri** (*Optional[Union[str, List[str]]], optional*) – URI or list of URIs of GeoTIFFs to use as the source of segmentation label data. If the labels are in the form of GeoJSONs, use label_vector_uri instead. Defaults to None.
- **label_vector_uri** (*Optional[str], optional*) – URI of GeoJSON file to use as the source of segmentation label data. If the labels are in the form of GeoTIFFs, use label_raster_uri instead. Defaults to None.
- **class_config** (*Optional['ClassConfig']*) – The ClassConfig. Can be None if not using any labels.
- **aoi_uri** (*Union[str, List[str]], optional*) – URI or list of URIs of GeoJSONs that specify the area-of-interest. If provided, the dataset will only access data from this area. Defaults to [].
- **label_vector_default_class_id** (*Optional[int], optional*) – If using label_vector_uri and all polygons in that file belong to the same class and they do not contain a *class_id* property, then use this argument to map all of the polgons to the appropriate class ID. See docs for ClassInferenceTransformer for more details. Defaults to None.
- **image_raster_source_kw** (*dict, optional*) – Additional arguments to pass to the RasterioSource used for image data. See docs for RasterioSource for more details. Defaults to {}.
- **label_raster_source_kw** (*dict, optional*) – Additional arguments to pass to the RasterioSource used for label data, if label_raster_uri is used. See docs for RasterioSource for more details. Defaults to {}.
- **label_vector_source_kw** (*dict, optional*) – Additional arguments to pass to the GeoJSONVectorSource used for label data, if label_vector_uri is used. See docs for GeoJSONVectorSource for more details. Defaults to {}.
- ****kwargs** – All other keyword args are passed to the default constructor for this class.

Raises
ValueError – If both label_raster_uri and label_vector_uri are specified.

Returns
An instance of this GeoDataset subclass.

get_resize_transform(*transform: Optional[BasicTransform]*, *out_size: Tuple[PositiveInt, PositiveInt]*) → Union[Resize, Compose]

Get transform to use for resizing windows to out_size.

Parameters

- **transform** (*Optional[BasicTransform]*) –
- **out_size** (*Tuple[PositiveInt, PositiveInt]*) –

Return type
Union[*Resize*, *Compose*]

sample_window() → *Box*

If scene has AOI polygons, try to find a random window that is within the AOI. Otherwise, just return the first sampled window.

Raises
StopIteration – If unable to find a valid window within self.max_sample_attempts attempts.

Returns
The sampled window.

Return type
Box

sample_window_loc(*h: int*, *w: int*) → Tuple[int, int]

Randomly sample coordinates of the top left corner of the window.

Parameters

- **h** (*int*) –
- **w** (*int*) –

Return type
Tuple[int, int]

sample_window_size() → Tuple[int, int]

Randomly sample the window size.

Return type
Tuple[int, int]

property max_size

property min_size

SemanticSegmentationSlidingWindowGeoDataset

class SemanticSegmentationSlidingWindowGeoDataset

Bases: *SlidingWindowGeoDataset*

__init__(**args*, ***kwargs*)

Constructor.

Parameters

- **scene** (Scene) – A Scene object.
- **size** (*Union[PosInt, Tuple[PosInt, PosInt]]*) – Window size.
- **stride** (*Union[PosInt, Tuple[PosInt, PosInt]]*) – Step size between windows.
- **padding** (*Optional[Union[NonNegInt, Tuple[NonNegInt, NonNegInt]]]*) – How many pixels the windows are allowed to overflow the sides of the raster source. If None, padding is set to size // 2. Defaults to None.
- **pad_direction** (*Literal['both', 'start', 'end']*) – If 'end', only pad ymax and xmax (bottom and right). If 'start', only pad ymin and xmin (top and left). If 'both', pad all sides. Has no effect if paddiong is zero. Defaults to 'end'.
- **transform** (*Optional[A.BasicTransform], optional*) – Albumentations transform to apply to the windows. Defaults to None. Each transform in Albumentations takes images of type uint8, and sometimes other data types. The data type requirements can be seen at https://albumentations.ai/docs/api_reference/augmentations/transforms/ # noqa If there is a mismatch between the data type of imagery and the transform requirements, a RasterTransformer should be set on the RasterSource that converts to uint8, such as MinMaxTransformer or StatsTransformer.
- **transform_type** (*Optional[TransformType], optional*) – Type of transform. Defaults to None.
- **normalize** (*bool, optional*) – If True, x is normalized to [0, 1] based on its data type. Defaults to True.
- **to_pytorch** (*bool, optional*) – If True, x and y are converted to pytorch tensors. Defaults to True.

Methods

__init__(*args, **kwargs)	Constructor.
from_uris(image_uri[, label_raster_uri, ...])	Create an instance of this class from image and label URIs.
init_windows()	Pre-compute windows.

__init__(**args*, ***kwargs*)

Constructor.

Parameters

- **scene** (Scene) – A Scene object.
- **size** (*Union[PosInt, Tuple[PosInt, PosInt]]*) – Window size.
- **stride** (*Union[PosInt, Tuple[PosInt, PosInt]]*) – Step size between windows.

- **padding** (*Optional[Union[NonNegInt, Tuple[NonNegInt, NonNegInt]]]*) – How many pixels the windows are allowed to overflow the sides of the raster source. If None, padding is set to size // 2. Defaults to None.
- **pad_direction** (*Literal['both', 'start', 'end']*) – If 'end', only pad ymax and xmax (bottom and right). If 'start', only pad ymin and xmin (top and left). If 'both', pad all sides. Has no effect if paddiong is zero. Defaults to 'end'.
- **transform** (*Optional[A.BasicTransform], optional*) – Albumentations transform to apply to the windows. Defaults to None. Each transform in Albumentations takes images of type uint8, and sometimes other data types. The data type requirements can be seen at https://albumentations.ai/docs/api_reference/augmentations/transforms/ # noqa If there is a mismatch between the data type of imagery and the transform requirements, a RasterTransformer should be set on the RasterSource that converts to uint8, such as MinMaxTransformer or StatsTransformer.
- **transform_type** (*Optional[TransformType], optional*) – Type of transform. Defaults to None.
- **normalize** (*bool, optional*) – If True, x is normalized to [0, 1] based on its data type. Defaults to True.
- **to_pytorch** (*bool, optional*) – If True, x and y are converted to pytorch tensors. Defaults to True.

static __new__(*cls*, **args: Any*, ***kwargs: Any*) → Any

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
Any

classmethod from_uris(*image_uri: Union[str, List[str]]*, *label_raster_uri: Optional[Union[str, List[str]]] = None*, *label_vector_uri: Optional[str] = None*, *class_config: Optional[ClassConfig] = None*, *aoi_uri: Union[str, List[str]] = []*, *label_vector_default_class_id: Optional[int] = None*, *image_raster_source_kw: dict = {}*, *label_raster_source_kw: dict = {}*, *label_vector_source_kw: dict = {}*, ***kwargs*)

Create an instance of this class from image and label URIs.

This is a convenience method. For more fine-grained control, it is recommended to use the default constructor.

Parameters

- **image_uri** (*Union[str, List[str]]*) – URI or list of URIs of GeoTIFFs to use as the source of image data.
- **label_raster_uri** (*Optional[Union[str, List[str]]], optional*) – URI or list of URIs of GeoTIFFs to use as the source of segmentation label data. If the labels are in the form of GeoJSONs, use label_vector_uri instead. Defaults to None.
- **label_vector_uri** (*Optional[str], optional*) – URI of GeoJSON file to use as the source of segmentation label data. If the labels are in the form of GeoTIFFs, use label_raster_uri instead. Defaults to None.
- **class_config** (*Optional['ClassConfig']*) – The ClassConfig. Can be None if not using any labels.

- **aoi_uri** (*Union[str, List[str]], optional*) – URI or list of URIs of GeoJSONs that specify the area-of-interest. If provided, the dataset will only access data from this area. Defaults to [].
- **label_vector_default_class_id** (*Optional[int], optional*) – If using label_vector_uri and all polygons in that file belong to the same class and they do not contain a *class_id* property, then use this argument to map all of the polgons to the appropriate class ID. See docs for ClassInferenceTransformer for more details. Defaults to None.
- **image_raster_source_kw** (*dict, optional*) – Additional arguments to pass to the RasterioSource used for image data. See docs for RasterioSource for more details. Defaults to {}.
- **label_raster_source_kw** (*dict, optional*) – Additional arguments to pass to the RasterioSource used for label data, if label_raster_uri is used. See docs for RasterioSource for more details. Defaults to {}.
- **label_vector_source_kw** (*dict, optional*) – Additional arguments to pass to the GeoJSONVectorSource used for label data, if label_vector_uri is used. See docs for GeoJSONVectorSource for more details. Defaults to {}.
- ****kwargs** – All other keyword args are passed to the default constructor for this class.

Raises

ValueError – If both label_raster_uri and label_vector_uri are specified.

Returns

An instance of this GeoDataset subclass.

init_windows() → None

Pre-compute windows.

Return type

None

Functions

make_ss_geodataset(cls, image_uri[, ...])	Create an instance of this class from image and label URIs.

make_ss_geodataset

make_ss_geodataset(*cls*, *image_uri: Union[str, List[str]]*, *label_raster_uri: Optional[Union[str, List[str]]] = None*, *label_vector_uri: Optional[str] = None*, *class_config: Optional[ClassConfig] = None*, *aoi_uri: Union[str, List[str]] = []*, *label_vector_default_class_id: Optional[int] = None*, *image_raster_source_kw: dict = {}*, *label_raster_source_kw: dict = {}*, *label_vector_source_kw: dict = {}*, ***kwargs*)

Create an instance of this class from image and label URIs.

This is a convenience method. For more fine-grained control, it is recommended to use the default constructor.

Parameters

- **image_uri** (*Union[str, List[str]]*) – URI or list of URIs of GeoTIFFs to use as the source of image data.

- **label_raster_uri** (*Optional[Union[str, List[str]]], optional*) – URI or list of URIs of GeoTIFFs to use as the source of segmentation label data. If the labels are in the form of GeoJSONs, use label_vector_uri instead. Defaults to None.
- **label_vector_uri** (*Optional[str], optional*) – URI of GeoJSON file to use as the source of segmentation label data. If the labels are in the form of GeoTIFFs, use label_raster_uri instead. Defaults to None.
- **class_config** (*Optional['ClassConfig']*) – The ClassConfig. Can be None if not using any labels.
- **aoi_uri** (*Union[str, List[str]], optional*) – URI or list of URIs of GeoJSONs that specify the area-of-interest. If provided, the dataset will only access data from this area. Defaults to [].
- **label_vector_default_class_id** (*Optional[int], optional*) – If using label_vector_uri and all polygons in that file belong to the same class and they do not contain a *class_id* property, then use this argument to map all of the polgons to the appropriate class ID. See docs for ClassInferenceTransformer for more details. Defaults to None.
- **image_raster_source_kw** (*dict, optional*) – Additional arguments to pass to the RasterioSource used for image data. See docs for RasterioSource for more details. Defaults to {}.
- **label_raster_source_kw** (*dict, optional*) – Additional arguments to pass to the RasterioSource used for label data, if label_raster_uri is used. See docs for RasterioSource for more details. Defaults to {}.
- **label_vector_source_kw** (*dict, optional*) – Additional arguments to pass to the GeoJSONVectorSource used for label data, if label_vector_uri is used. See docs for GeoJSONVectorSource for more details. Defaults to {}.
- ****kwargs** – All other keyword args are passed to the default constructor for this class.

Raises

ValueError – If both label_raster_uri and label_vector_uri are specified.

Returns

An instance of this GeoDataset subclass.

transform

Classes

TransformType	An enumeration.

TransformType

class TransformType

Bases: Enum

An enumeration.

Attributes

noop
classification
regression
object_detection
semantic_segmentation

__init__()

classification = 'classification'

noop = 'noop'

object_detection = 'object_detection'

regression = 'regression'

semantic_segmentation = 'semantic_segmentation'

Functions

albu_to_yxyx(xyxy, img_size)	Albumentations format (i.e.
classification_transformer(inp[, transform])	Apply transform to image only.
object_detection_transformer(inp[, transform])	Apply transform to image, bounding boxes, and labels.
regression_transformer(inp[, transform])	Apply transform to image only.
semantic_segmentation_transformer(inp[, ...])	Apply transform to image and mask.
xywh_to_albu(xywh, img_size)	Unnormalized [xmin, ymin, w, h] to Albumentations format i.e. normalized [ymin, xmin, ymax, xmax].
yxyx_to_albu(yxyx, img_size)	Unnormalized [ymin, xmin, ymax, xmax] to Albumentations format i.e. normalized [ymin, xmin, ymax, xmax].

albu_to_yxyx

albu_to_yxyx(*xyxy: ndarray*, *img_size: Tuple[PositiveInt, PositiveInt]*) → ndarray

Albumentations format (i.e. normalized [ymin, xmin, ymax, xmax]) to unnormalized [ymin, xmin, ymax, xmax].

Parameters

- **xyxy** (*ndarray*) –
- **img_size** (*Tuple[PositiveInt, PositiveInt]*) –

Return type
ndarray

classification_transformer

classification_transformer(*inp: Tuple[ndarray, Optional[int]]*, *transform=typing.Optional[albumentations.core.transforms_interface.BasicTransform]*) → Tuple[ndarray, Optional[ndarray]]

Apply transform to image only.

Parameters
inp (*Tuple[ndarray, Optional[int]]*) –

Return type
Tuple[*ndarray*, *Optional*[*ndarray*]]

object_detection_transformer

object_detection_transformer(*inp: Tuple[ndarray, Optional[Tuple[ndarray, ndarray, str]]]*, *transform: Optional[BasicTransform] = None*) → Tuple[torch.Tensor, Optional[*BoxList*]]

Apply transform to image, bounding boxes, and labels. Also perform normalization and conversion to pytorch tensors.

The transform's BBoxParams are expected to have the format 'albumentations' (i.e. normalized [ymin, xmin, ymax, xmax]).

Parameters

- **inp** (*Tuple[np.ndarray, Optional[Tuple[np.ndarray, np.ndarray, str]]]*) – Tuple of the form: (image, (boxes, class_ids, box_format)). box_format must be 'yxyx' or 'xywh'.
- **transform** (*Optional[A.BasicTransform], optional*) – A transform. Defaults to None.

Raises
NotImplementedError – If box_format is not 'yxyx' or 'xywh'.

Returns
Transformed image and boxes.

Return type
Tuple[torch.Tensor, *BoxList*]

regression_transformer

regression_transformer(*inp: Tuple[ndarray, Optional[Any]]*, *transform=typing.Optional[albumentations.core.transforms_interface.BasicTransform]*) → Tuple[ndarray, Optional[ndarray]]

Apply transform to image only.

Parameters
inp (*Tuple[ndarray, Optional[Any]]*) –

Return type
Tuple[*ndarray*, *Optional*[*ndarray*]]

semantic_segmentation_transformer

semantic_segmentation_transformer(*inp: Tuple[ndarray, Optional[ndarray]]*, *transform=typing.Optional[albumentations.core.transforms_interface.BasicTransform]*) → Tuple[ndarray, Optional[ndarray]]

Apply transform to image and mask.

Parameters
inp (*Tuple[ndarray, Optional[ndarray]]*) –

Return type
Tuple[*ndarray*, *Optional*[*ndarray*]]

xywh_to_albu

xywh_to_albu(*xywh: ndarray*, *img_size: Tuple[PositiveInt, PositiveInt]*) → ndarray

Unnormalized [xmin, ymin, w, h] to Albumentations format i.e. normalized [ymin, xmin, ymax, xmax].

Parameters

- **xywh** (*ndarray*) –
- **img_size** (*Tuple[PositiveInt, PositiveInt]*) –

Return type
ndarray

yxyx_to_albu

yxyx_to_albu(*yxyx: ndarray*, *img_size: Tuple[PositiveInt, PositiveInt]*) → ndarray

Unnormalized [ymin, xmin, ymax, xmax] to Albumentations format i.e. normalized [ymin, xmin, ymax, xmax].

Parameters

- **yxyx** (*ndarray*) –
- **img_size** (*Tuple[PositiveInt, PositiveInt]*) –

Return type
ndarray

utils

Modules

aoi_sampler

utils

aoi_sampler

Classes

AoiSampler	Given a set of polygons representing the AOI, allows efficiently sampling points inside the AOI uniformly at random.

AoiSampler

class AoiSampler

Bases: object

Given a set of polygons representing the AOI, allows efficiently sampling points inside the AOI uniformly at random.

To achieve this, each polygon is first partitioned into triangles (triangulation). Then, to sample a single point, we first sample a triangle at random with probability proportional to its area and then sample a point within that triangle uniformly at random.

__init__(*polygons: Sequence[Polygon]*) → None

Parameters
polygons (*Sequence[Polygon]*) –

Return type
None

Methods

__init__(polygons)	
polygon_to_graph(polygon)	Given the exterior of a polygon, return its graph representation.
sample([n])	Sample a random point within the AOI, using the following algorithm:
triangle_area(vertices, simplices)	Calculate area of each triangle specified by the simplices array using Heron's formula.
triangle_origin_and_basis(vertices, simplices)	For each triangle ABC, return point A, vector AB, and vector AC.
triangle_side_lengths(vertices, simplices)	Calculate lengths of all 3 sides of each triangle specified by the simplices array.
triangulate(polygons)	
triangulate_polygon(polygon)	Extract vertices and edges from the polygon (and its holes, if any) and pass them to the Triangle library for triangulation.

__init__(*polygons: Sequence[Polygon]*) → None

Parameters
polygons (*Sequence[Polygon]*) –

Return type
None

polygon_to_graph(*polygon: LinearRing*) → Tuple[ndarray, ndarray]

Given the exterior of a polygon, return its graph representation.

Parameters
polygon (*LinearRing*) – The exterior of a polygon.

Returns
An (N, 2) array of vertices and an (N, 2) array of indices to vertices representing edges.

Return type
Tuple[np.ndarray, np.ndarray]

sample(*n: int = 1*) → ndarray

Sample a random point within the AOI, using the following algorithm:

- Randomly sample one triangle (ABC) with probability proportional

to its area. - Starting at A, travel a random distance along vectors AB and AC. - Return the final position.

Parameters
n (*int, optional*) – Number of points to sample. Defaults to 1.

Returns
(n, 2) 2D coordinates of the sampled points.

Return type
np.ndarray

triangle_area(*vertices: ndarray*, *simplices: ndarray*) → ndarray

Calculate area of each triangle specified by the simplices array using Heron's formula.

Parameters

- **vertices** (*np.ndarray*) – (N, 2) array of vertex coords in 2D.
- **simplices** (*np.ndarray*) – (N, 3) array of indexes to entries in the vertices array. Each row represents one triangle.

Returns
(N,) array of areas

Return type
np.ndarray

triangle_origin_and_basis(*vertices: ndarray*, *simplices: ndarray*) → Tuple[ndarray, Tuple[ndarray, ndarray]]

For each triangle ABC, return point A, vector AB, and vector AC.

Parameters

- **vertices** (*np.ndarray*) – (N, 2) array of vertex coords in 2D.
- **simplices** (*np.ndarray*) – (N, 3) array of indexes to entries in the vertices array. Each row represents one triangle.

Returns

3 arrays of shape
(N, 2), organized into tuples like so: (point A, (vector AB, vector AC)).

Return type
Tuple[np.ndarray, Tuple[np.ndarray, np.ndarray]]

triangle_side_lengths(*vertices: ndarray*, *simplices: ndarray*) → Tuple[ndarray, ndarray, ndarray]

Calculate lengths of all 3 sides of each triangle specified by the simplices array.

Parameters

- **vertices** (*np.ndarray*) – (N, 2) array of vertex coords in 2D.
- **simplices** (*np.ndarray*) – (N, 3) array of indexes to entries in the vertices array. Each row represents one triangle.

Returns
||AB||, ||BC||, ||AC||

Return type
Tuple[np.ndarray, np.ndarray, np.ndarray]

triangulate(*polygons*) → dict

Return type
dict

triangulate_polygon(*polygon: Polygon*) → dict

Extract vertices and edges from the polygon (and its holes, if any) and pass them to the Triangle library for triangulation.

Parameters
polygon (*Polygon*) –

Return type
dict

utils

Functions

discover_images(dir[, extensions])	Find all images with the given `extensions` in `dir`.
load_image(path)	Read in image from path and return as a (H, W, C) numpy array.
make_image_folder_dataset(data_dir[, classes])	Initializes and returns an ImageFolder.

discover_images

discover_images(*dir: PathLike*, *extensions: Iterable[str] = ('.npy',)*) → List[Path]

Find all images with the given `extensions` in `dir`.

Parameters

- **dir** (*PathLike*) –
- **extensions** (*Iterable[str]*) –

Return type
List[*Path*]

load_image

load_image(*path: PathLike*) → ndarray

Read in image from path and return as a (H, W, C) numpy array.

Parameters
path (*PathLike*) –

Return type
ndarray

make_image_folder_dataset

make_image_folder_dataset(*data_dir: str*, *classes: Optional[Iterable[str]] = None*) → torchvision.datasets.folder.DatasetFolder

Initializes and returns an ImageFolder.

If classes is specified, ImageFolder's default class-to-index mapping behavior is overriden to use the indices of classes instead.

Parameters

- **data_dir** (*str*) –
- **classes** (*Optional[Iterable[str]]*) –

Return type
torchvision.datasets.folder.DatasetFolder

Exceptions

DatasetError
GeoDatasetError
ImageDatasetError

rastervision.pytorch_learner.dataset.utils.utils.DatasetError

exception DatasetError

__init__(**args*, ***kwargs*)

__new__(***kwargs*)

rastervision.pytorch_learner.dataset.utils.utils.GeoDatasetError

exception GeoDatasetError

__init__(**args*, ***kwargs*)

__new__(***kwargs*)

rastervision.pytorch_learner.dataset.utils.utils.ImageDatasetError

exception ImageDatasetError

__init__(**args*, ***kwargs*)

__new__(***kwargs*)

visualizer

Modules

classification_visualizer
object_detection_visualizer
regression_visualizer
semantic_segmentation_visualizer
visualizer

classification_visualizer

Classes

ClassificationVisualizer	Plots samples from image classification Datasets.

ClassificationVisualizer

class ClassificationVisualizer

Bases: *Visualizer*

Plots samples from image classification Datasets.

Attributes

scale

__init__(*class_names: List[str]*, *class_colors: Optional[List[Union[str, Tuple[int, int, int]]]] = None*, *transform: Optional[Dict] = {'__version__': '1.3.0', 'transform': {'__class_fullname__': 'rastervision.pytorch_learner.utils.utils.MinMaxNormalize', 'always_apply': False, 'dtype': 5, 'max_val': 1.0, 'min_val': 0.0, 'p': 1.0}}*, *channel_display_groups: Optional[Union[Dict[str, Sequence[ConstrainedIntValue]], Sequence[Sequence[ConstrainedIntValue]]] = None*)

Constructor.

Parameters

- **class_names** (*List[str]*) – names of classes
- **class_colors** (*Optional[List[Union[str, Tuple[int, int, int]]]]*) – Colors used to display classes. Can be color 3-tuples in list form.
- **transform** (*Optional[Dict]*) – An Albumentations transform serialized as a dict that will be applied to each image before it is plotted. Mainly useful for undoing any data transformation that you do not want included in the plot, such as normalization. The default value will shift and scale the image so the values range from 0.0 to 1.0 which is the expected range for the plotting function. This default is useful for cases where the values after normalization are close to zero which makes the plot difficult to see.
- **channel_display_groups** (*Optional[Union[Dict[str, Sequence[ConstrainedIntValue]], Sequence[Sequence[ConstrainedIntValue]]]]*) – Groups of image channels to display together as a subplot when plotting the data and predictions. Can be a list or tuple of groups (e.g. [(0, 1, 2), (3,)]) or a dict containing title-to-group mappings (e.g. {"RGB": [0, 1, 2], "IR": [3]}), where each group is a list or tuple of channel indices and title is a string that will be used as the title of the subplot for that group.

Methods

__init__(class_names[, class_colors, ...])	Constructor.
get_batch(dataset[, batch_sz])	Generate a batch from a dataset.
get_channel_display_groups(nb_img_channels)	
get_collate_fn()	Returns a custom collate_fn to use in DataLoader.
get_plot_ncols(**kwargs)	
get_plot_nrows(**kwargs)	
get_plot_params(**kwargs)	
plot_batch(x, y[, output_path, z, ...])	Plot a whole batch in a grid using plot_xyz.
plot_xyz(axs, x, y[, z])	Plot image, ground truth labels, and predicted labels.

__init__(*class_names: List[str]*, *class_colors: Optional[List[Union[str, Tuple[int, int, int]]]] = None*, *transform: Optional[Dict] = {'__version__': '1.3.0', 'transform': {'__class_fullname__': 'rastervision.pytorch_learner.utils.utils.MinMaxNormalize', 'always_apply': False, 'dtype': 5, 'max_val': 1.0, 'min_val': 0.0, 'p': 1.0}}*, *channel_display_groups: Optional[Union[Dict[str, Sequence[ConstrainedIntValue]], Sequence[Sequence[ConstrainedIntValue]]]] = None*)

Constructor.

Parameters

- **class_names** (*List[str]*) – names of classes
- **class_colors** (*Optional[List[Union[str, Tuple[int, int, int]]]]*) – Colors used to display classes. Can be color 3-tuples in list form.
- **transform** (*Optional[Dict]*) – An Albumentations transform serialized as a dict that will be applied to each image before it is plotted. Mainly useful for undoing any data transformation that you do not want included in the plot, such as normalization. The default value will shift and scale the image so the values range from 0.0 to 1.0 which is the expected range for the plotting function. This default is useful for cases where the values after normalization are close to zero which makes the plot difficult to see.
- **channel_display_groups** (*Optional[Union[Dict[str, Sequence[ConstrainedIntValue]], Sequence[Sequence[ConstrainedIntValue]]]]*) – Groups of image channels to display together as a subplot when plotting the data and predictions. Can be a list or tuple of groups (e.g. [(0, 1, 2), (3,)]) or a dict containing title-to-group mappings (e.g. {"RGB": [0, 1, 2], "IR": [3]}), where each group is a list or tuple of channel indices and title is a string that will be used as the title of the subplot for that group.

get_batch(*dataset: Dataset*, *batch_sz: int = 4*, ***kwargs*) → Tuple[torch.Tensor, Any]

Generate a batch from a dataset.

This is a convenience method for generating a batch of data to plot.

Returns (x, y) tuple where x is images and y is labels

Parameters

- **dataset** (*Dataset*) –
- **batch_sz** (*int*) –

Return type
Tuple[torch.Tensor, *Any*]

get_channel_display_groups(*nb_img_channels: int*) → Union[Dict[str, Sequence[ConstrainedIntValue]], Sequence[Sequence[ConstrainedIntValue]]]

Parameters
nb_img_channels (*int*) –

Return type
Union[*Dict*[str, *Sequence*[*ConstrainedIntValue*]], *Sequence*[*Sequence*[*ConstrainedIntValue*]]]

get_collate_fn() → Optional[callable]

Returns a custom collate_fn to use in DataLoader.

None is returned if default collate_fn should be used.

See https://pytorch.org/docs/stable/data.html#working-with-collate-fn

Return type
Optional[callable]

get_plot_ncols(***kwargs*) → int

Return type
int

get_plot_nrows(***kwargs*) → int

Return type
int

get_plot_params(***kwargs*) → dict

Return type
dict

plot_batch(*x: torch.Tensor*, *y: Sequence*, *output_path: Optional[str] = None*, *z: Optional[Sequence] = None*, *batch_limit: Optional[int] = None*, *show: bool = False*)

Plot a whole batch in a grid using plot_xyz.

Parameters

- **x** (*torch.Tensor*) – batch of images
- **y** (*Sequence*) – ground truth labels
- **output_path** (*Optional[str]*) – local path where to save plot image
- **z** (*Optional[Sequence]*) – optional predicted labels
- **batch_limit** (*Optional[int]*) – optional limit on (rendered) batch size
- **show** (*bool*) –

plot_xyz(*axs: Sequence*, *x: torch.Tensor*, *y: int*, *z: Optional[int] = None*) → None

Plot image, ground truth labels, and predicted labels.

Parameters

- **axs** (*Sequence*) – matplotlib axes on which to plot
- **x** (*torch.Tensor*) – image
- **y** (*int*) – ground truth labels
- **z** (*Optional[int]*) – optional predicted labels

Return type
None

scale: float = 3.0

object_detection_visualizer

Classes

ObjectDetectionVisualizer	Plots samples from object detection Datasets.

ObjectDetectionVisualizer

class ObjectDetectionVisualizer

Bases: *Visualizer*

Plots samples from object detection Datasets.

Attributes

scale

__init__(*class_names: List[str], class_colors: Optional[List[Union[str, Tuple[int, int, int]]]] = None, transform: Optional[Dict] = {'__version__': '1.3.0', 'transform': {'__class_fullname__': 'rastervision.pytorch_learner.utils.utils.MinMaxNormalize', 'always_apply': False, 'dtype': 5, 'max_val': 1.0, 'min_val': 0.0, 'p': 1.0}}, channel_display_groups: Optional[Union[Dict[str, Sequence[ConstrainedIntValue]], Sequence[Sequence[ConstrainedIntValue]]] = None*)

Constructor.

Parameters

- **class_names** (*List[str]*) – names of classes
- **class_colors** (*Optional[List[Union[str, Tuple[int, int, int]]]]*) – Colors used to display classes. Can be color 3-tuples in list form.
- **transform** (*Optional[Dict]*) – An Albumentations transform serialized as a dict that will be applied to each image before it is plotted. Mainly useful for undoing any data transformation that you do not want included in the plot, such as normalization. The default value will shift and scale the image so the values range from 0.0 to 1.0 which is the expected range for the plotting function. This default is useful for cases where the values after normalization are close to zero which makes the plot difficult to see.
- **channel_display_groups** (*Optional[Union[Dict[str, Sequence[ConstrainedIntValue]], Sequence[Sequence[ConstrainedIntValue]]]]*) – Groups of image channels to display together as a subplot when plotting the data and predictions. Can be a list or tuple of groups (e.g. [(0, 1, 2), (3,)]) or a dict containing title-to-group mappings (e.g. {"RGB": [0, 1, 2], "IR": [3]}), where each group is a list or tuple of channel indices and title is a string that will be used as the title of the subplot for that group.

Methods

__init__(class_names[, class_colors, ...])	Constructor.
get_batch(dataset[, batch_sz])	Generate a batch from a dataset.
get_channel_display_groups(nb_img_channels)	
get_collate_fn()	Returns a custom collate_fn to use in DataLoader.
get_plot_ncols(**kwargs)	
get_plot_nrows(**kwargs)	
get_plot_params(**kwargs)	
plot_batch(x, y[, output_path, z, ...])	Plot a whole batch in a grid using plot_xyz.
plot_xyz(axs, x, y[, z])	Plot image, ground truth labels, and predicted labels.

__init__(*class_names: List[str]*, *class_colors: Optional[List[Union[str, Tuple[int, int, int]]]] = None*, *transform: Optional[Dict] = {'__version__': '1.3.0', 'transform': {'__class_fullname__': 'rastervision.pytorch_learner.utils.utils.MinMaxNormalize', 'always_apply': False, 'dtype': 5, 'max_val': 1.0, 'min_val': 0.0, 'p': 1.0}}*, *channel_display_groups: Optional[Union[Dict[str, Sequence[ConstrainedIntValue]], Sequence[Sequence[ConstrainedIntValue]]] = None*)

Constructor.

Parameters

- **class_names** (*List[str]*) – names of classes
- **class_colors** (*Optional[List[Union[str, Tuple[int, int, int]]]]*) – Colors used to display classes. Can be color 3-tuples in list form.
- **transform** (*Optional[Dict]*) – An Albumentations transform serialized as a dict that will be applied to each image before it is plotted. Mainly useful for undoing any data transformation that you do not want included in the plot, such as normalization. The default value will shift and scale the image so the values range from 0.0 to 1.0 which is the expected range for the plotting function. This default is useful for cases where the values after normalization are close to zero which makes the plot difficult to see.
- **channel_display_groups** (*Optional[Union[Dict[str, Sequence[ConstrainedIntValue]], Sequence[Sequence[ConstrainedIntValue]]]]*) – Groups of image channels to display together as a subplot when plotting the data and predictions. Can be a list or tuple of groups (e.g. [(0, 1, 2), (3,)]) or a dict containing title-to-group mappings (e.g. {"RGB": [0, 1, 2], "IR": [3]}), where each group is a list or tuple of channel indices and title is a string that will be used as the title of the subplot for that group.

get_batch(*dataset: Dataset*, *batch_sz: int = 4*, ***kwargs*) → Tuple[torch.Tensor, Any]

Generate a batch from a dataset.

This is a convenience method for generating a batch of data to plot.

Returns (x, y) tuple where x is images and y is labels

Parameters

- **dataset** (*Dataset*) –
- **batch_sz** (*int*) –

Return type
Tuple[torch.Tensor, *Any*]

get_channel_display_groups(*nb_img_channels: int*) → Union[Dict[str, Sequence[ConstrainedIntValue]], Sequence[Sequence[ConstrainedIntValue]]]

Parameters
nb_img_channels (*int*) –

Return type
Union[*Dict*[str, *Sequence*[*ConstrainedIntValue*]], *Sequence*[*Sequence*[*ConstrainedIntValue*]]]

get_collate_fn()

Returns a custom collate_fn to use in DataLoader.

None is returned if default collate_fn should be used.

See https://pytorch.org/docs/stable/data.html#working-with-collate-fn

get_plot_ncols(***kwargs*) → int

Return type
int

get_plot_nrows(***kwargs*) → int

Return type
int

get_plot_params(***kwargs*) → dict

Return type
dict

plot_batch(*x: torch.Tensor*, *y: Sequence*, *output_path: Optional[str] = None*, *z: Optional[Sequence] = None*, *batch_limit: Optional[int] = None*, *show: bool = False*)

Plot a whole batch in a grid using plot_xyz.

Parameters

- **x** (*torch.Tensor*) – batch of images
- **y** (*Sequence*) – ground truth labels
- **output_path** (*Optional[str]*) – local path where to save plot image
- **z** (*Optional[Sequence]*) – optional predicted labels
- **batch_limit** (*Optional[int]*) – optional limit on (rendered) batch size
- **show** (*bool*) –

plot_xyz(*axs: Sequence*, *x: torch.Tensor*, *y:* BoxList, *z: Optional[*BoxList*] = None*) → None

Plot image, ground truth labels, and predicted labels.

Parameters

- **axs** (*Sequence*) – matplotlib axes on which to plot
- **x** (*torch.Tensor*) – image
- **y** (BoxList) – ground truth labels
- **z** (*Optional[*BoxList*]*) – optional predicted labels

Return type
None

scale: float = 3.0

regression_visualizer

Classes

RegressionVisualizer	Plots samples from image regression Datasets.

RegressionVisualizer

class RegressionVisualizer

Bases: *Visualizer*

Plots samples from image regression Datasets.

Attributes

scale

__init__(*class_names: List[str], class_colors: Optional[List[Union[str, Tuple[int, int, int]]]] = None, transform: Optional[Dict] = {'__version__': '1.3.0', 'transform': {'__class_fullname__': 'rastervision.pytorch_learner.utils.utils.MinMaxNormalize', 'always_apply': False, 'dtype': 5, 'max_val': 1.0, 'min_val': 0.0, 'p': 1.0}}, channel_display_groups: Optional[Union[Dict[str, Sequence[ConstrainedIntValue]], Sequence[Sequence[ConstrainedIntValue]]]] = None*)

Constructor.

Parameters

- **class_names** (*List[str]*) – names of classes
- **class_colors** (*Optional[List[Union[str, Tuple[int, int, int]]]]*) – Colors used to display classes. Can be color 3-tuples in list form.
- **transform** (*Optional[Dict]*) – An Albumentations transform serialized as a dict that will be applied to each image before it is plotted. Mainly useful for undoing any data transformation that you do not want included in the plot, such as normalization. The default value will shift and scale the image so the values range from 0.0 to 1.0 which is the expected range for the plotting function. This default is useful for cases where the values after normalization are close to zero which makes the plot difficult to see.
- **channel_display_groups** (*Optional[Union[Dict[str, Sequence[ConstrainedIntValue]], Sequence[Sequence[ConstrainedIntValue]]]]*) – Groups of image channels to display together as a subplot when plotting the data and predictions. Can be a list or tuple of groups (e.g. [(0, 1, 2), (3,)]) or a dict containing title-to-group mappings (e.g. {"RGB": [0, 1, 2], "IR": [3]}), where each group is a list or tuple of channel indices and title is a string that will be used as the title of the subplot for that group.

Methods

__init__(class_names[, class_colors, ...])	Constructor.
get_batch(dataset[, batch_sz])	Generate a batch from a dataset.
get_channel_display_groups(nb_img_channels)	
get_collate_fn()	Returns a custom collate_fn to use in DataLoader.
get_plot_ncols(**kwargs)	
get_plot_nrows(**kwargs)	
get_plot_params(**kwargs)	
plot_batch(x, y[, output_path, z, ...])	Plot a whole batch in a grid using plot_xyz.
plot_xyz(axs, x, y[, z])	Plot image, ground truth labels, and predicted labels.

__init__(*class_names: List[str]*, *class_colors: Optional[List[Union[str, Tuple[int, int, int]]]] = None*, *transform: Optional[Dict] = {'__version__': '1.3.0', 'transform': {'__class_fullname__': 'rastervision.pytorch_learner.utils.utils.MinMaxNormalize', 'always_apply': False, 'dtype': 5, 'max_val': 1.0, 'min_val': 0.0, 'p': 1.0}}*, *channel_display_groups: Optional[Union[Dict[str, Sequence[ConstrainedIntValue]], Sequence[Sequence[ConstrainedIntValue]]] = None*)

Constructor.

Parameters

- **class_names** (*List[str]*) – names of classes
- **class_colors** (*Optional[List[Union[str, Tuple[int, int, int]]]]*) – Colors used to display classes. Can be color 3-tuples in list form.
- **transform** (*Optional[Dict]*) – An Albumentations transform serialized as a dict that will be applied to each image before it is plotted. Mainly useful for undoing any data transformation that you do not want included in the plot, such as normalization. The default value will shift and scale the image so the values range from 0.0 to 1.0 which is the expected range for the plotting function. This default is useful for cases where the values after normalization are close to zero which makes the plot difficult to see.
- **channel_display_groups** (*Optional[Union[Dict[str, Sequence[ConstrainedIntValue]], Sequence[Sequence[ConstrainedIntValue]]]]*) – Groups of image channels to display together as a subplot when plotting the data and predictions. Can be a list or tuple of groups (e.g. [(0, 1, 2), (3,)]) or a dict containing title-to-group mappings (e.g. {"RGB": [0, 1, 2], "IR": [3]}), where each group is a list or tuple of channel indices and title is a string that will be used as the title of the subplot for that group.

get_batch(*dataset: Dataset*, *batch_sz: int = 4*, ***kwargs*) → Tuple[torch.Tensor, Any]

Generate a batch from a dataset.

This is a convenience method for generating a batch of data to plot.

Returns (x, y) tuple where x is images and y is labels

Parameters

- **dataset** (*Dataset*) –
- **batch_sz** (*int*) –

Return type
Tuple[torch.Tensor, *Any*]

get_channel_display_groups(*nb_img_channels: int*) → Union[Dict[str, Sequence[ConstrainedIntValue]], Sequence[Sequence[ConstrainedIntValue]]]

Parameters
nb_img_channels (*int*) –

Return type
Union[*Dict*[str, *Sequence*[*ConstrainedIntValue*]], *Sequence*[*Sequence*[*ConstrainedIntValue*]]]

get_collate_fn() → Optional[callable]

Returns a custom collate_fn to use in DataLoader.

None is returned if default collate_fn should be used.

See https://pytorch.org/docs/stable/data.html#working-with-collate-fn

Return type
Optional[callable]

get_plot_ncols(***kwargs*) → int

Return type
int

get_plot_nrows(***kwargs*) → int

Return type
int

get_plot_params(***kwargs*) → dict

Return type
dict

plot_batch(*x: torch.Tensor*, *y: Sequence*, *output_path: Optional[str] = None*, *z: Optional[Sequence] = None*, *batch_limit: Optional[int] = None*, *show: bool = False*)

Plot a whole batch in a grid using plot_xyz.

Parameters

- **x** (*torch.Tensor*) – batch of images
- **y** (*Sequence*) – ground truth labels
- **output_path** (*Optional[str]*) – local path where to save plot image
- **z** (*Optional[Sequence]*) – optional predicted labels
- **batch_limit** (*Optional[int]*) – optional limit on (rendered) batch size
- **show** (*bool*) –

plot_xyz(*axs: Sequence*, *x: torch.Tensor*, *y: int*, *z: Optional[int] = None*) → None

Plot image, ground truth labels, and predicted labels.

Parameters

- **axs** (*Sequence*) – matplotlib axes on which to plot
- **x** (*torch.Tensor*) – image

- **y** (*int*) – ground truth labels
- **z** (*Optional[int]*) – optional predicted labels

Return type
None

scale: float = 3.0

semantic_segmentation_visualizer

Classes

SemanticSegmentationVisualizer	Plots samples from semantic segmentation Datasets.

SemanticSegmentationVisualizer

class SemanticSegmentationVisualizer

Bases: *Visualizer*

Plots samples from semantic segmentation Datasets.

Attributes

scale

__init__(*class_names: List[str], class_colors: Optional[List[Union[str, Tuple[int, int, int]]]] = None, transform: Optional[Dict] = {'__version__': '1.3.0', 'transform': {'__class_fullname__': 'rastervision.pytorch_learner.utils.utils.MinMaxNormalize', 'always_apply': False, 'dtype': 5, 'max_val': 1.0, 'min_val': 0.0, 'p': 1.0}}, channel_display_groups: Optional[Union[Dict[str, Sequence[ConstrainedIntValue]], Sequence[Sequence[ConstrainedIntValue]]] = None*)

Constructor.

Parameters

- **class_names** (*List[str]*) – names of classes
- **class_colors** (*Optional[List[Union[str, Tuple[int, int, int]]]]*) – Colors used to display classes. Can be color 3-tuples in list form.
- **transform** (*Optional[Dict]*) – An Albumentations transform serialized as a dict that will be applied to each image before it is plotted. Mainly useful for undoing any data transformation that you do not want included in the plot, such as normalization. The default value will shift and scale the image so the values range from 0.0 to 1.0 which is the expected range for the plotting function. This default is useful for cases where the values after normalization are close to zero which makes the plot difficult to see.
- **channel_display_groups** (*Optional[Union[Dict[str, Sequence[ConstrainedIntValue]], Sequence[Sequence[ConstrainedIntValue]]]]*) – Groups of image channels to display together as a subplot when plotting the data and predictions. Can be a list or tuple of groups (e.g. [(0, 1, 2), (3,)]) or a dict containing title-to-group mappings (e.g. {"RGB": [0, 1, 2], "IR": [3]}), where each group is a list or

tuple of channel indices and title is a string that will be used as the title of the subplot for that group.

Methods

__init__(class_names[, class_colors, ...])	Constructor.
get_batch(dataset[, batch_sz])	Generate a batch from a dataset.
get_channel_display_groups(nb_img_channels)	
get_collate_fn()	Returns a custom collate_fn to use in DataLoader.
get_plot_ncols(**kwargs)	
get_plot_nrows(**kwargs)	
get_plot_params(**kwargs)	
plot_batch(x, y[, output_path, z, ...])	Plot a whole batch in a grid using plot_xyz.
plot_xyz(axs, x, y[, z])	Plot image, ground truth labels, and predicted labels.

__init__(*class_names: List[str]*, *class_colors: Optional[List[Union[str, Tuple[int, int, int]]]] = None*, *transform: Optional[Dict] = {'__version__': '1.3.0', 'transform': {'__class_fullname__': 'rastervision.pytorch_learner.utils.utils.MinMaxNormalize', 'always_apply': False, 'dtype': 5, 'max_val': 1.0, 'min_val': 0.0, 'p': 1.0}}*, *channel_display_groups: Optional[Union[Dict[str, Sequence[ConstrainedIntValue]], Sequence[Sequence[ConstrainedIntValue]]] = None*)

Constructor.

Parameters

- **class_names** (*List[str]*) – names of classes
- **class_colors** (*Optional[List[Union[str, Tuple[int, int, int]]]]*) – Colors used to display classes. Can be color 3-tuples in list form.
- **transform** (*Optional[Dict]*) – An Albumentations transform serialized as a dict that will be applied to each image before it is plotted. Mainly useful for undoing any data transformation that you do not want included in the plot, such as normalization. The default value will shift and scale the image so the values range from 0.0 to 1.0 which is the expected range for the plotting function. This default is useful for cases where the values after normalization are close to zero which makes the plot difficult to see.
- **channel_display_groups** (*Optional[Union[Dict[str, Sequence[ConstrainedIntValue]], Sequence[Sequence[ConstrainedIntValue]]]]*) – Groups of image channels to display together as a subplot when plotting the data and predictions. Can be a list or tuple of groups (e.g. [(0, 1, 2), (3,)]) or a dict containing title-to-group mappings (e.g. {"RGB": [0, 1, 2], "IR": [3]}), where each group is a list or tuple of channel indices and title is a string that will be used as the title of the subplot for that group.

get_batch(*dataset: Dataset*, *batch_sz: int = 4*, ***kwargs*) → Tuple[torch.Tensor, Any]

Generate a batch from a dataset.

This is a convenience method for generating a batch of data to plot.

Returns (x, y) tuple where x is images and y is labels

Parameters

- **dataset** (*Dataset*) –
- **batch_sz** (*int*) –

Return type
Tuple[torch.Tensor, *Any*]

get_channel_display_groups(*nb_img_channels: int*) → Union[Dict[str, Sequence[ConstrainedIntValue]], Sequence[Sequence[ConstrainedIntValue]]]

Parameters
nb_img_channels (*int*) –

Return type
Union[*Dict*[str, *Sequence*[*ConstrainedIntValue*]], *Sequence*[*Sequence*[*ConstrainedIntValue*]]]

get_collate_fn() → Optional[callable]

Returns a custom collate_fn to use in DataLoader.

None is returned if default collate_fn should be used.

See https://pytorch.org/docs/stable/data.html#working-with-collate-fn

Return type
Optional[callable]

get_plot_ncols(***kwargs*) → int

Return type
int

get_plot_nrows(***kwargs*) → int

Return type
int

get_plot_params(***kwargs*) → dict

Return type
dict

plot_batch(*x: torch.Tensor*, *y: Sequence*, *output_path: Optional[str] = None*, *z: Optional[Sequence] = None*, *batch_limit: Optional[int] = None*, *show: bool = False*)

Plot a whole batch in a grid using plot_xyz.

Parameters

- **x** (*torch.Tensor*) – batch of images
- **y** (*Sequence*) – ground truth labels
- **output_path** (*Optional[str]*) – local path where to save plot image
- **z** (*Optional[Sequence]*) – optional predicted labels
- **batch_limit** (*Optional[int]*) – optional limit on (rendered) batch size
- **show** (*bool*) –

plot_xyz(*axs: Sequence*, *x: torch.Tensor*, *y: Union[torch.Tensor, ndarray]*, *z: Optional[torch.Tensor] = None*) → None

Plot image, ground truth labels, and predicted labels.

Parameters

- **axs** (*Sequence*) – matplotlib axes on which to plot
- **x** (*torch.Tensor*) – image
- **y** (*Union[torch.Tensor, ndarray]*) – ground truth labels
- **z** (*Optional[torch.Tensor]*) – optional predicted labels

Return type
None

scale: float = 3.0

visualizer

Classes

Visualizer	Base class for plotting samples from computer vision PyTorch Datasets.

Visualizer

class Visualizer

Bases: ABC

Base class for plotting samples from computer vision PyTorch Datasets.

Attributes

scale

__init__(*class_names: List[str], class_colors: Optional[List[Union[str, Tuple[int, int, int]]]] = None, transform: Optional[Dict] = {'__version__': '1.3.0', 'transform': {'__class_fullname__': 'rastervision.pytorch_learner.utils.utils.MinMaxNormalize', 'always_apply': False, 'dtype': 5, 'max_val': 1.0, 'min_val': 0.0, 'p': 1.0}}, channel_display_groups: Optional[Union[Dict[str, Sequence[ConstrainedIntValue]], Sequence[Sequence[ConstrainedIntValue]]] = None*)

Constructor.

Parameters

- **class_names** (*List[str]*) – names of classes
- **class_colors** (*Optional[List[Union[str, Tuple[int, int, int]]]]*) – Colors used to display classes. Can be color 3-tuples in list form.
- **transform** (*Optional[Dict]*) – An Albumentations transform serialized as a dict that will be applied to each image before it is plotted. Mainly useful for undoing any data transformation that you do not want included in the plot, such as normalization. The default value will shift and scale the image so the values range from 0.0 to 1.0 which is the expected range for the plotting function. This default is useful for cases where the values after normalization are close to zero which makes the plot difficult to see.

- **channel_display_groups** (*Optional[Union[Dict[str, Sequence[ConstrainedIntValue]], Sequence[Sequence[ConstrainedIntValue]]]]*) – Groups of image channels to display together as a subplot when plotting the data and predictions. Can be a list or tuple of groups (e.g. [(0, 1, 2), (3,)]) or a dict containing title-to-group mappings (e.g. {"RGB": [0, 1, 2], "IR": [3]}), where each group is a list or tuple of channel indices and title is a string that will be used as the title of the subplot for that group.

Methods

__init__(class_names[, class_colors, ...])	Constructor.
get_batch(dataset[, batch_sz])	Generate a batch from a dataset.
get_channel_display_groups(nb_img_channels)	
get_collate_fn()	Returns a custom collate_fn to use in DataLoader.
get_plot_ncols(**kwargs)	
get_plot_nrows(**kwargs)	
get_plot_params(**kwargs)	
plot_batch(x, y[, output_path, z, ...])	Plot a whole batch in a grid using plot_xyz.
plot_xyz(axs, x, y[, z])	Plot image, ground truth labels, and predicted labels.

__init__(*class_names: List[str]*, *class_colors: Optional[List[Union[str, Tuple[int, int, int]]]] = None*, *transform: Optional[Dict] = {'__version__': '1.3.0', 'transform': {'__class_fullname__': 'rastervision.pytorch_learner.utils.utils.MinMaxNormalize', 'always_apply': False, 'dtype': 5, 'max_val': 1.0, 'min_val': 0.0, 'p': 1.0}}*, *channel_display_groups: Optional[Union[Dict[str, Sequence[ConstrainedIntValue]], Sequence[Sequence[ConstrainedIntValue]]]] = None*)

Constructor.

Parameters

- **class_names** (*List[str]*) – names of classes
- **class_colors** (*Optional[List[Union[str, Tuple[int, int, int]]]]*) – Colors used to display classes. Can be color 3-tuples in list form.
- **transform** (*Optional[Dict]*) – An Albumentations transform serialized as a dict that will be applied to each image before it is plotted. Mainly useful for undoing any data transformation that you do not want included in the plot, such as normalization. The default value will shift and scale the image so the values range from 0.0 to 1.0 which is the expected range for the plotting function. This default is useful for cases where the values after normalization are close to zero which makes the plot difficult to see.
- **channel_display_groups** (*Optional[Union[Dict[str, Sequence[ConstrainedIntValue]], Sequence[Sequence[ConstrainedIntValue]]]]*) – Groups of image channels to display together as a subplot when plotting the data and predictions. Can be a list or tuple of groups (e.g. [(0, 1, 2), (3,)]) or a dict containing title-to-group mappings (e.g. {"RGB": [0, 1, 2], "IR": [3]}), where each group is a list or tuple of channel indices and title is a string that will be used as the title of the subplot for that group.

get_batch(*dataset: Dataset*, *batch_sz: int = 4*, ***kwargs*) → Tuple[torch.Tensor, Any]

Generate a batch from a dataset.

This is a convenience method for generating a batch of data to plot.

Returns (x, y) tuple where x is images and y is labels

Parameters

- **dataset** (*Dataset*) –
- **batch_sz** (*int*) –

Return type
Tuple[torch.Tensor, *Any*]

get_channel_display_groups(*nb_img_channels: int*) → Union[Dict[str, Sequence[ConstrainedIntValue]], Sequence[Sequence[ConstrainedIntValue]]]

Parameters
nb_img_channels (*int*) –

Return type
Union[*Dict*[str, *Sequence*[*ConstrainedIntValue*]], *Sequence*[*Sequence*[*ConstrainedIntValue*]]]

get_collate_fn() → Optional[callable]

Returns a custom collate_fn to use in DataLoader.

None is returned if default collate_fn should be used.

See https://pytorch.org/docs/stable/data.html#working-with-collate-fn

Return type
Optional[callable]

get_plot_ncols(***kwargs*) → int

Return type
int

get_plot_nrows(***kwargs*) → int

Return type
int

get_plot_params(***kwargs*) → dict

Return type
dict

plot_batch(*x: torch.Tensor*, *y: Sequence*, *output_path: Optional[str] = None*, *z: Optional[Sequence] = None*, *batch_limit: Optional[int] = None*, *show: bool = False*)

Plot a whole batch in a grid using plot_xyz.

Parameters

- **x** (*torch.Tensor*) – batch of images
- **y** (*Sequence*) – ground truth labels
- **output_path** (*Optional[str]*) – local path where to save plot image
- **z** (*Optional[Sequence]*) – optional predicted labels

- **batch_limit** (*Optional[int]*) – optional limit on (rendered) batch size
- **show** (*bool*) –

abstract plot_xyz(*axs*, *x: torch.Tensor*, *y*, *z=None*)

Plot image, ground truth labels, and predicted labels.

Parameters

- **axs** – matplotlib axes on which to plot
- **x** (*torch.Tensor*) – image
- **y** – ground truth labels
- **z** – optional predicted labels

scale: float = 3.0

9.3.4 learner

Classes

Learner	Abstract training and prediction routines for a model.

Learner

class Learner

Bases: ABC

Abstract training and prediction routines for a model.

This can be subclassed to handle different computer vision tasks.

The datasets, model, optimizer, and schedulers will be generated from the cfg if not specified in the constructor.

If instantiated with training=False, the training apparatus (loss, optimizer, scheduler, logging, etc.) will not be set up and the model will be put into eval mode.

Note that various training and prediction methods have the side effect of putting Learner.model into training or eval mode. No attempt is made to put the model back into the mode it was previously in.

__init__(*cfg: LearnerConfig*, *output_dir: Optional[str] = None*, *train_ds: Optional[Dataset] = None*, *valid_ds: Optional[Dataset] = None*, *test_ds: Optional[Dataset] = None*, *model: Optional[torch.nn.Module] = None*, *loss: Optional[Callable] = None*, *optimizer: Optional[Optimizer] = None*, *epoch_scheduler: Optional[_LRScheduler] = None*, *step_scheduler: Optional[_LRScheduler] = None*, *tmp_dir: Optional[str] = None*, *model_weights_path: Optional[str] = None*, *model_def_path: Optional[str] = None*, *loss_def_path: Optional[str] = None*, *training: bool = True*)

Constructor.

Parameters

- **cfg** (*LearnerConfig*) – LearnerConfig.
- **train_ds** (*Optional[Dataset], optional*) – The dataset to use for training. If None, will be generated from cfg.data. Defaults to None.

- **valid_ds** (*Optional[Dataset], optional*) – The dataset to use for validation. If None, will be generated from cfg.data. Defaults to None.
- **test_ds** (*Optional[Dataset], optional*) – The dataset to use for testing. If None, will be generated from cfg.data. Defaults to None.
- **model** (*Optional[nn.Module], optional*) – The model. If None, will be generated from cfg.model. Defaults to None.
- **loss** (*Optional[Callable], optional*) – The loss function. If None, will be generated from cfg.solver. Defaults to None.
- **optimizer** (*Optional[Optimizer], optional*) – The optimizer. If None, will be generated from cfg.solver. Defaults to None.
- **epoch_scheduler** (*Optional[_LRScheduler], optional*) – The scheduler that updates after each epoch. If None, will be generated from cfg.solver. Defaults to None.
- **step_scheduler** (*Optional[_LRScheduler], optional*) – The scheduler that updates after each optimizer-step. If None, will be generated from cfg.solver. Defaults to None.
- **tmp_dir** (*Optional[str], optional*) – A temporary directory to use for downloads etc. If None, will be auto-generated. Defaults to None.
- **model_weights_path** (*Optional[str], optional*) – URI of model weights to initialize the model with. Defaults to None.
- **model_def_path** (*Optional[str], optional*) – A local path to a directory with a hubconf.py. If provided, the model definition is imported from here. This is used when loading an external model from a model-bundle. Defaults to None.
- **loss_def_path** (*Optional[str], optional*) – A local path to a directory with a hubconf.py. If provided, the loss function definition is imported from here. This is used when loading an external loss function from a model-bundle. Defaults to None.
- **training** (*bool, optional*) – If False, the training apparatus (loss, optimizer, scheduler, logging, etc.) will not be set up and the model will be put into eval mode. If True, the training apparatus will be set up and the model will be put into training mode. Defaults to True.
- **output_dir** (*Optional[str]*) –

Methods

__init__(cfg[, output_dir, train_ds, ...])	Constructor.
build_dataloaders()	Set the DataLoaders for train, validation, and test sets.
build_datasets()	
build_epoch_scheduler([start_epoch])	Returns an LR scheduler that changes the LR each epoch.
build_loss([loss_def_path])	Build a loss Callable.
build_metric_names()	Returns names of metrics used to validate model at each epoch.
build_model([model_def_path])	Build a PyTorch model.
build_optimizer()	Returns optimizer.

continues on next page

Table 3 – continued from previous page

build_step_scheduler([start_epoch])	Returns an LR scheduler that changes the LR each step.
eval_model(split)	Evaluate model using a particular dataset split.
from_model_bundle(model_bundle_uri[, ...])	Create a Learner from a model bundle.
get_collate_fn()	Returns a custom collate_fn to use in DataLoader.
get_dataloader(split)	Get the DataLoader for a split.
get_start_epoch()	Get start epoch.
get_train_sampler(train_ds)	Return a sampler to use for the training dataloader or None to not use any.
get_visualizer_class()	Returns a Visualizer class object for plotting data samples.
load_checkpoint()	Load last weights from previous run if available.
load_init_weights([model_weights_path])	Load the weights to initialize model.
load_weights(uri, **kwargs)	Load model weights from a file.
log_data_stats()	Log stats about each DataSet.
main()	Main training sequence.
normalize_input(x)	Normalize x to [0, 1].
numpy_predict(x[, raw_out])	Make a prediction using an image or batch of images in numpy format.
on_epoch_end(curr_epoch, metrics)	Hook that is called at end of epoch.
on_overfit_start()	Hook that is called at start of overfit routine.
on_train_start()	Hook that is called at start of train routine.
output_to_numpy(out)	Convert output of model to numpy format.
overfit()	Optimize model using the same batch repeatedly.
plot_dataloader(dl, output_path[, ...])	Plot images and ground truth labels for a DataLoader.
plot_dataloaders([batch_limit, show])	Plot images and ground truth labels for all DataLoaders.
plot_predictions(split[, batch_limit, show])	Plot predictions for a split.
post_forward(x)	Post process output of call to model().
predict(x[, raw_out])	Make prediction for an image or batch of images.
predict_dataloader(dl[, batched_output, ...])	Returns an iterator over predictions on the given dataloader.
predict_dataset(dataset[, return_format, ...])	Returns an iterator over predictions on the given dataset.
prob_to_pred(x)	Convert a Tensor with prediction probabilities to class ids.
run_tensorboard()	Run TB server serving logged stats.
save_model_bundle()	Save a model bundle.
setup_data()	Set datasets and dataLoaders for train, validation, and test sets.
setup_loss([loss_def_path])	Setup self.loss.
setup_model([model_weights_path, model_def_path])	Setup self.model.
setup_tensorboard()	Setup for logging stats to TB.
setup_training([loss_def_path])	
stop_tensorboard()	Stop TB logging and server if it's running.
sync_from_cloud()	Sync any previous output in the cloud to output_dir.
sync_to_cloud()	Sync any output to the cloud at output_uri.
to_batch(x)	Ensure that image array has batch dimension.
to_device(x, device)	Load Tensors onto a device.

continues on next page

Table 3 – continued from previous page

train([epochs])	Training loop that will attempt to resume training if appropriate.
train_end(outputs, num_samples)	Aggregate the ouput of train_step at the end of the epoch.
train_epoch(optimizer[, step_scheduler])	Train for a single epoch.
train_step(batch, batch_ind)	Compute loss for a single training batch.
validate_end(outputs, num_samples)	Aggregate the ouput of validate_step at the end of the epoch.
validate_epoch(dl)	Validate for a single epoch.
validate_step(batch, batch_ind)	Compute metrics on validation batch.

__init__(*cfg: LearnerConfig*, *output_dir: Optional[str] = None*, *train_ds: Optional[Dataset] = None*, *valid_ds: Optional[Dataset] = None*, *test_ds: Optional[Dataset] = None*, *model: Optional[torch.nn.Module] = None*, *loss: Optional[Callable] = None*, *optimizer: Optional[Optimizer] = None*, *epoch_scheduler: Optional[_LRScheduler] = None*, *step_scheduler: Optional[_LRScheduler] = None*, *tmp_dir: Optional[str] = None*, *model_weights_path: Optional[str] = None*, *model_def_path: Optional[str] = None*, *loss_def_path: Optional[str] = None*, *training: bool = True*)

Constructor.

Parameters

- **cfg** (*LearnerConfig*) – LearnerConfig.
- **train_ds** (*Optional[Dataset], optional*) – The dataset to use for training. If None, will be generated from cfg.data. Defaults to None.
- **valid_ds** (*Optional[Dataset], optional*) – The dataset to use for validation. If None, will be generated from cfg.data. Defaults to None.
- **test_ds** (*Optional[Dataset], optional*) – The dataset to use for testing. If None, will be generated from cfg.data. Defaults to None.
- **model** (*Optional[nn.Module], optional*) – The model. If None, will be generated from cfg.model. Defaults to None.
- **loss** (*Optional[Callable], optional*) – The loss function. If None, will be generated from cfg.solver. Defaults to None.
- **optimizer** (*Optional[Optimizer], optional*) – The optimizer. If None, will be generated from cfg.solver. Defaults to None.
- **epoch_scheduler** (*Optional[_LRScheduler], optional*) – The scheduler that updates after each epoch. If None, will be generated from cfg.solver. Defaults to None.
- **step_scheduler** (*Optional[_LRScheduler], optional*) – The scheduler that updates after each optimizer-step. If None, will be generated from cfg.solver. Defaults to None.
- **tmp_dir** (*Optional[str], optional*) – A temporary directory to use for downloads etc. If None, will be auto-generated. Defaults to None.
- **model_weights_path** (*Optional[str], optional*) – URI of model weights to initialize the model with. Defaults to None.
- **model_def_path** (*Optional[str], optional*) – A local path to a directory with a hubconf.py. If provided, the model definition is imported from here. This is used when loading an external model from a model-bundle. Defaults to None.

- **loss_def_path** (*Optional[str], optional*) – A local path to a directory with a hubconf.py. If provided, the loss function definition is imported from here. This is used when loading an external loss function from a model-bundle. Defaults to None.
- **training** (*bool, optional*) – If False, the training apparatus (loss, optimizer, scheduler, logging, etc.) will not be set up and the model will be put into eval mode. If True, the training apparatus will be set up and the model will be put into training mode. Defaults to True.
- **output_dir** (*Optional[str]*) –

build_dataloaders() → Tuple[torch.utils.data.DataLoader, torch.utils.data.DataLoader, torch.utils.data.DataLoader]

Set the DataLoaders for train, validation, and test sets.

Return type
Tuple[torch.utils.data.DataLoader, torch.utils.data.DataLoader, torch.utils.data.DataLoader]

build_datasets() → Tuple[Dataset, Dataset, Dataset]

Return type
Tuple[Dataset, Dataset, Dataset]

build_epoch_scheduler(*start_epoch: int = 0*) → _LRScheduler

Returns an LR scheduler that changes the LR each epoch.

Parameters
start_epoch (*int*) –

Return type
_LRScheduler

build_loss(*loss_def_path: Optional[str] = None*) → Callable

Build a loss Callable.

Parameters
loss_def_path (*Optional[str]*) –

Return type
Callable

build_metric_names() → List[str]

Returns names of metrics used to validate model at each epoch.

Return type
List[str]

build_model(*model_def_path: Optional[str] = None*) → torch.nn.Module

Build a PyTorch model.

Parameters
model_def_path (*Optional[str]*) –

Return type
torch.nn.Module

build_optimizer() → Optimizer

Returns optimizer.

Return type
Optimizer

build_step_scheduler(*start_epoch: int = 0*) → _LRScheduler

Returns an LR scheduler that changes the LR each step.

Parameters
start_epoch (*int*) –

Return type
_LRScheduler

eval_model(*split: str*)

Evaluate model using a particular dataset split.

Gets validation metrics and saves them along with prediction plots.

Parameters
split (*str*) – the dataset split to use: train, valid, or test.

classmethod from_model_bundle(*model_bundle_uri: str*, *tmp_dir: Optional[str] = None*, *cfg: Optional[LearnerConfig] = None*, *training: bool = False*, ***kwargs*) → *Learner*

Create a Learner from a model bundle.

Note: This is the bundle saved in `train/model-bundle.zip` and not `bundle/model-bundle.zip`.

Parameters

- **model_bundle_uri** (*str*) – URI of the model bundle.
- **tmp_dir** (*Optional[str], optional*) – Optional temporary directory. Will be used for unzipping bundle and also passed to the default constructor. If None, will be auto-generated. Defaults to None.
- **cfg** (*Optional[LearnerConfig], optional*) – If None, will be read from the bundle. Defaults to None.
- **training** (*bool, optional*) – If False, the training apparatus (loss, optimizer, scheduler, logging, etc.) will not be set up and the model will be put into eval mode. If True, the training apparatus will be set up and the model will be put into training mode. Defaults to True.
- ****kwargs** – See *Learner.__init__()*.

Raises
FileNotFoundError – If using custom Albumentations transforms and definition file is not found in bundle.

Returns
Object of the Learner subclass on which this was called.

Return type
Learner

get_collate_fn() → Optional[callable]

Returns a custom collate_fn to use in DataLoader.

None is returned if default collate_fn should be used.

See https://pytorch.org/docs/stable/data.html#working-with-collate-fn

Return type
Optional[callable]

get_dataloader(*split: str*) → torch.utils.data.DataLoader

Get the DataLoader for a split.

Parameters
split (*str*) – a split name which can be train, valid, or test

Return type
torch.utils.data.DataLoader

get_start_epoch() → int

Get start epoch.

If training was interrupted, this returns the last complete epoch + 1.

Return type
int

get_train_sampler(*train_ds: Dataset*) → Optional[Sampler]

Return a sampler to use for the training dataloader or None to not use any.

Parameters
train_ds (*Dataset*) –

Return type
Optional[Sampler]

abstract get_visualizer_class() → Type[*Visualizer*]

Returns a Visualizer class object for plotting data samples.

Return type
Type[Visualizer]

load_checkpoint()

Load last weights from previous run if available.

load_init_weights(*model_weights_path: Optional[str] = None*) → None

Load the weights to initialize model.

Parameters
model_weights_path (*Optional[str]*) –

Return type
None

load_weights(*uri: str*, ***kwargs*) → None

Load model weights from a file.

Parameters
uri (*str*) –

Return type
None

log_data_stats()

Log stats about each DataSet.

main()

Main training sequence.

This plots the dataset, runs a training and validation loop (which will resume if interrupted), logs stats, plots predictions, and syncs results to the cloud.

normalize_input(*x: ndarray*) → ndarray

Normalize x to [0, 1].

If x.dtype is a subtype of np.unsignedinteger, normalize it to [0, 1] using the max possible value of that dtype. Otherwise, assume it is in [0, 1] already and do nothing.

Parameters
x (*np.ndarray*) – an image or batch of images

Returns
the same array scaled to [0, 1].

Return type
ndarray

numpy_predict(*x: ndarray*, *raw_out: bool = False*) → ndarray

Make a prediction using an image or batch of images in numpy format. If x.dtype is a subtype of np.unsignedinteger, it will be normalized to [0, 1] using the max possible value of that dtype. Otherwise, x will be assumed to be in [0, 1] already and will be cast to torch.float32 directly.

Parameters

- **x** (*ndarray*) – (ndarray) of shape [height, width, channels] or [batch_sz, height, width, channels]
- **raw_out** (*bool*) – if True, return prediction probabilities

Returns
predictions using numpy arrays

Return type
ndarray

on_epoch_end(*curr_epoch*, *metrics*)

Hook that is called at end of epoch.

Writes metrics to CSV and TB, and saves model.

on_overfit_start()

Hook that is called at start of overfit routine.

on_train_start()

Hook that is called at start of train routine.

output_to_numpy(*out: torch.Tensor*) → ndarray

Convert output of model to numpy format.

Parameters
out (*torch.Tensor*) – the output of the model in PyTorch format

Return type
ndarray

Returns: the output of the model in numpy format

overfit()

Optimize model using the same batch repeatedly.

plot_dataloader(*dl: torch.utils.data.DataLoader*, *output_path: str*, *batch_limit: Optional[int] = None*, *show: bool = False*)

Plot images and ground truth labels for a DataLoader.

Parameters

- **dl** (*torch.utils.data.DataLoader*) –
- **output_path** (*str*) –
- **batch_limit** (*Optional[int]*) –
- **show** (*bool*) –

plot_dataloaders(*batch_limit: Optional[int] = None*, *show: bool = False*)

Plot images and ground truth labels for all DataLoaders.

Parameters

- **batch_limit** (*Optional[int]*) –
- **show** (*bool*) –

plot_predictions(*split: str*, *batch_limit: Optional[int] = None*, *show: bool = False*)

Plot predictions for a split.

Uses the first batch for the corresponding DataLoader.

Parameters

- **split** (*str*) – dataset split. Can be train, valid, or test.
- **batch_limit** (*Optional[int]*) – optional limit on (rendered) batch size
- **show** (*bool*) –

post_forward(*x: Any*) → Any

Post process output of call to model().

Useful for when predictions are inside a structure returned by model().

Parameters
x (*Any*) –

Return type
Any

predict(*x: torch.Tensor*, *raw_out: bool = False*) → Any

Make prediction for an image or batch of images.

Parameters

- **x** (*Tensor*) – Image or batch of images as a float Tensor with pixel values normalized to [0, 1].
- **raw_out** (*bool*) – if True, return prediction probabilities

Returns

the predictions, in probability form if raw_out is True, in class_id form
otherwise

Return type
Any

predict_dataloader(*dl: torch.utils.data.DataLoader*, *batched_output: bool = True*, *return_format: Literal['xyz', 'yz', 'z'] = 'z'*, *raw_out: bool = True*, *predict_kw: dict = {}*) → Union[Iterator[Any], Iterator[Tuple[Any, ...]]]

Returns an iterator over predictions on the given dataloader.

Parameters

- **dl** (*DataLoader*) – The dataloader to make predictions on.
- **batched_output** (*bool, optional*) – If True, return batches of x, y, z as defined by the dataloader. If False, unroll the batches into individual items. Defaults to True.
- **return_format** (*Literal['xyz', 'yz', 'z'], optional*) – Format of the return elements of the returned iterator. Must be one of: 'xyz', 'yz', and 'z'. If 'xyz', elements are 3-tuples of x, y, and z. If 'yz', elements are 2-tuples of y and z. If 'z', elements are (non-tuple) values of z. Where x = input image, y = ground truth, and z = prediction. Defaults to 'z'.
- **raw_out** (*bool, optional*) – If true, return raw predicted scores. Defaults to True.
- **predict_kw** (*dict*) – Dict with keywords passed to Learner.predict(). Useful if a Learner subclass implements a custom predict() method.

Raises
ValueError – If return_format is not one of the allowed values.

Returns

If return_format
is 'z', the returned value is an iterator of whatever type the predictions are. Otherwise, the returned value is an iterator of tuples.

Return type
Union[Iterator[Any], Iterator[Tuple[Any, ...]]]

predict_dataset(*dataset: Dataset*, *return_format: Literal['xyz', 'yz', 'z'] = 'z'*, *raw_out: bool = True*, *numpy_out: bool = False*, *predict_kw: dict = {}*, *dataloader_kw: dict = {}*, *progress_bar: bool = True*, *progress_bar_kw: dict = {}*) → Union[Iterator[Any], Iterator[Tuple[Any, ...]]]

Returns an iterator over predictions on the given dataset.

Parameters

- **dataset** (*Dataset*) – The dataset to make predictions on.
- **return_format** (*Literal['xyz', 'yz', 'z'], optional*) – Format of the return elements of the returned iterator. Must be one of: 'xyz', 'yz', and 'z'. If 'xyz', elements are 3-tuples of x, y, and z. If 'yz', elements are 2-tuples of y and z. If 'z', elements are (non-tuple) values of z. Where x = input image, y = ground truth, and z = prediction. Defaults to 'z'.
- **raw_out** (*bool, optional*) – If true, return raw predicted scores. Defaults to True.
- **numpy_out** (*bool, optional*) – If True, convert predictions to numpy arrays before returning. Defaults to False.
- **predict_kw** (*dict*) – Dict with keywords passed to Learner.predict(). Useful if a Learner subclass implements a custom predict() method.
- **dataloader_kw** (*dict*) – Dict with keywords passed to the DataLoader constructor.

- **progress_bar** (*bool, optional*) – If True, display a progress bar. Since this function returns an iterator, the progress bar won't be visible until the iterator is consumed. Defaults to True.
- **progress_bar_kw** (*dict*) – Dict with keywords passed to tqdm.

Raises
ValueError – If return_format is not one of the allowed values.

Returns

If return_format
is 'z', the returned value is an iterator of whatever type the predictions are. Otherwise, the returned value is an iterator of tuples.

Return type
Union[Iterator[Any], Iterator[Tuple[Any, ...]]]

prob_to_pred(*x: torch.Tensor*) → torch.Tensor

Convert a Tensor with prediction probabilities to class ids.

The class ids should be the classes with the maximum probability.

Parameters
x (*torch.Tensor*) –

Return type
torch.Tensor

run_tensorboard()

Run TB server serving logged stats.

save_model_bundle()

Save a model bundle.

This is a zip file with the model weights in .pth format and a serialized copy of the LearningConfig, which allows for making predictions in the future.

setup_data()

Set datasets and dataLoaders for train, validation, and test sets.

setup_loss(*loss_def_path: Optional[str] = None*) → None

Setup self.loss.

Parameters

- **loss_def_path** (*str, optional*) – Loss definition path. Will be
- **None.** (*available when loading from a bundle. Defaults to*) –

Return type
None

setup_model(*model_weights_path: Optional[str] = None*, *model_def_path: Optional[str] = None*) → None

Setup self.model.

Parameters

- **model_weights_path** (*Optional[str], optional*) – Path to model weights. Will be available when loading from a bundle. Defaults to None.
- **model_def_path** (*Optional[str], optional*) – Path to model definition. Will be available when loading from a bundle. Defaults to None.

Return type
None

setup_tensorboard()

Setup for logging stats to TB.

setup_training(*loss_def_path: Optional[str] = None*) → None

Parameters
loss_def_path (*Optional[str]*) –

Return type
None

stop_tensorboard()

Stop TB logging and server if it's running.

sync_from_cloud()

Sync any previous output in the cloud to output_dir.

sync_to_cloud()

Sync any output to the cloud at output_uri.

to_batch(*x: torch.Tensor*) → torch.Tensor

Ensure that image array has batch dimension.

Parameters
x (*torch.Tensor*) – assumed to be either image or batch of images

Returns
x with extra batch dimension of length 1 if needed

Return type
torch.Tensor

to_device(*x: Any*, *device: str*) → Any

Load Tensors onto a device.

Parameters

- **x** (*Any*) – some object with Tensors in it
- **device** (*str*) – 'cpu' or 'cuda'

Returns
x but with any Tensors in it on the device

Return type
Any

train(*epochs: Optional[int] = None*)

Training loop that will attempt to resume training if appropriate.

Parameters
epochs (*Optional[int]*) –

train_end(*outputs: List[Dict[str, float]]*, *num_samples: int*) → Dict[str, float]

Aggregate the ouput of train_step at the end of the epoch.

Parameters

- **outputs** (*List[Dict[str, float]]*) – a list of outputs of train_step

- **num_samples** (*int*) – total number of training samples processed in epoch

Return type
Dict[str, float]

train_epoch(*optimizer: Optimizer*, *step_scheduler: Optional[_LRScheduler] = None*) → Dict[str, float]

Train for a single epoch.

Parameters

- **optimizer** (*Optimizer*) –
- **step_scheduler** (*Optional[_LRScheduler]*) –

Return type
Dict[str, float]

abstract train_step(*batch: Any*, *batch_ind: int*) → Dict[str, float]

Compute loss for a single training batch.

Parameters

- **batch** (*Any*) – batch data needed to compute loss
- **batch_ind** (*int*) – index of batch within epoch

Returns
dict with 'train_loss' as key and possibly other losses

Return type
Dict[str, float]

validate_end(*outputs: List[Dict[str, float]]*, *num_samples: int*) → Dict[str, float]

Aggregate the ouput of validate_step at the end of the epoch.

Parameters

- **outputs** (*List[Dict[str, float]]*) – a list of outputs of validate_step
- **num_samples** (*int*) – total number of validation samples processed in epoch

Return type
Dict[str, float]

validate_epoch(*dl: torch.utils.data.DataLoader*) → Dict[str, float]

Validate for a single epoch.

Parameters
dl (*torch.utils.data.DataLoader*) –

Return type
Dict[str, float]

abstract validate_step(*batch: Any*, *batch_ind: int*) → Dict[str, float]

Compute metrics on validation batch.

Parameters

- **batch** (*Any*) – batch data needed to compute validation metrics
- **batch_ind** (*int*) – index of batch within epoch

Returns
dict with metric names mapped to metric values

Return type
Dict[str, float]

Functions

log_system_details()	Log some system details.

log_system_details

`log_system_details()`

Log some system details.

9.3.5 learner_config

Classes

Backbone	An enumeration.
GeoDataWindowMethod	An enumeration.
NonEmptyStr	alias of `ConstrainedStrValue`
NonNegInt	alias of `ConstrainedIntValue`
Proportion	alias of `ConstrainedFloatValue`

Backbone

`class Backbone`

Bases: `Enum`

An enumeration.

Attributes

alexnet
densenet121
densenet169
densenet201
densenet161
googlenet
inception_v3

continues on next page

Table 4 – continued from previous page

mnasnet0_5
mnasnet0_75
mnasnet1_0
mnasnet1_3
mobilenet_v2
resnet18
resnet34
resnet50
resnet101
resnet152
resnext50_32x4d
resnext101_32x8d
wide_resnet50_2
wide_resnet101_2
shufflenet_v2_x0_5
shufflenet_v2_x1_0
shufflenet_v2_x1_5
shufflenet_v2_x2_0
squeezenet1_0
squeezenet1_1
vgg11
vgg11_bn
vgg13
vgg13_bn
vgg16

continues on next page

Table 4 – continued from previous page

vgg16_bn
vgg19_bn
vgg19

__init__()

Methods

int_to_str(x)

static int_to_str(*x*)

alexnet = 'alexnet'

densenet121 = 'densenet121'

densenet161 = 'densenet161'

densenet169 = 'densenet169'

densenet201 = 'densenet201'

googlenet = 'googlenet'

inception_v3 = 'inception_v3'

mnasnet0_5 = 'mnasnet0_5'

mnasnet0_75 = 'mnasnet0_75'

mnasnet1_0 = 'mnasnet1_0'

mnasnet1_3 = 'mnasnet1_3'

mobilenet_v2 = 'mobilenet_v2'

resnet101 = 'resnet101'

resnet152 = 'resnet152'

resnet18 = 'resnet18'

resnet34 = 'resnet34'

resnet50 = 'resnet50'

resnext101_32x8d = 'resnext101_32x8d'

resnext50_32x4d = 'resnext50_32x4d'

shufflenet_v2_x0_5 = 'shufflenet_v2_x0_5'

shufflenet_v2_x1_0 = 'shufflenet_v2_x1_0'

shufflenet_v2_x1_5 = 'shufflenet_v2_x1_5'

shufflenet_v2_x2_0 = 'shufflenet_v2_x2_0'

squeezenet1_0 = 'squeezenet1_0'

squeezenet1_1 = 'squeezenet1_1'

vgg11 = 'vgg11'

vgg11_bn = 'vgg11_bn'

vgg13 = 'vgg13'

vgg13_bn = 'vgg13_bn'

vgg16 = 'vgg16'

vgg16_bn = 'vgg16_bn'

vgg19 = 'vgg19'

vgg19_bn = 'vgg19_bn'

wide_resnet101_2 = 'wide_resnet101_2'

wide_resnet50_2 = 'wide_resnet50_2'

GeoDataWindowMethod

class GeoDataWindowMethod

Bases: Enum

An enumeration.

Attributes

sliding

random

__init__()

random = 'random'

sliding = 'sliding'

NonEmptyStr

NonEmptyStr

alias of ConstrainedStrValue

NonNegInt

NonNegInt

alias of ConstrainedIntValue

Proportion

Proportion

alias of ConstrainedFloatValue

Configs

DataConfig	Config related to dataset for training and testing.
ExternalModuleConfig	Config describing an object to be loaded via Torch Hub.
GeoDataConfig	Configure *GeoDatasets*.
GeoDataWindowConfig	Configure a *GeoDataset*.
ImageDataConfig	Config related to dataset for training and testing.
LearnerConfig	Config for Learner.
ModelConfig	Config related to models.
PlotOptions	Config related to plotting.
SolverConfig	Config related to solver aka optimizer.

DataConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model DataConfig

Config related to dataset for training and testing.


```
{
   "title": "DataConfig",
   "description": "Config related to dataset for training and testing.",
   "type": "object",
   "properties": {
      "class_names": {
         "title": "Class Names",
         "description": "Names of classes.",
         "default": [],
         "type": "array",
         "items": {
            "type": "string"
         }
```


(continues on next page)

(continued from previous page)

```
      },
      "class_colors": {
         "title": "Class Colors",
         "description": "Colors used to display classes. Can be color 3-tuples in␣
↪list form.",
         "type": "array",
         "items": {
            "anyOf": [
               {
                  "type": "string"
               },
               {
                  "type": "array",
                  "minItems": 3,
                  "maxItems": 3,
                  "items": [
                     {
                        "type": "integer"
                     },
                     {
                        "type": "integer"
                     },
                     {
                        "type": "integer"
                     }
                  ]
               }
            ]
         }
      },
      "img_channels": {
         "title": "Img Channels",
         "description": "The number of channels of the training images.",
         "exclusiveMinimum": 0,
         "type": "integer"
      },
      "img_sz": {
         "title": "Img Sz",
         "description": "Length of a side of each image in pixels. This is the size␣
↪to transform it to during training, not the size in the raw dataset.",
         "default": 256,
         "exclusiveMinimum": 0,
         "type": "integer"
      },
      "train_sz": {
         "title": "Train Sz",
         "description": "If set, the number of training images to use. If fewer␣
↪images exist, then an exception will be raised.",
         "type": "integer"
      },
      "train_sz_rel": {
         "title": "Train Sz Rel",
```

(continues on next page)

(continued from previous page)

```
         "description": "If set, the proportion of training images to use.",
         "type": "number"
      },
      "num_workers": {
         "title": "Num Workers",
         "description": "Number of workers to use when DataLoader makes batches.",
         "default": 4,
         "type": "integer"
      },
      "augmentors": {
         "title": "Augmentors",
         "description": "Names of albumentations augmentors to use for training␣
↪batches. Choices include: ['Blur', 'RandomRotate90', 'HorizontalFlip',
↪'VerticalFlip', 'GaussianBlur', 'GaussNoise', 'RGBShift', 'ToGray'].␣
↪Alternatively, a custom transform can be provided via the aug_transform option.",
         "default": [
            "RandomRotate90",
            "HorizontalFlip",
            "VerticalFlip"
         ],
         "type": "array",
         "items": {
            "type": "string"
         }
      },
      "base_transform": {
         "title": "Base Transform",
         "description": "An Albumentations transform serialized as a dict that will␣
↪be applied to all datasets: training, validation, and test. This transformation␣
↪is in addition to the resizing due to img_sz. This is useful for, for example,␣
↪applying the same normalization to all datasets.",
         "type": "object"
      },
      "aug_transform": {
         "title": "Aug Transform",
         "description": "An Albumentations transform serialized as a dict that will␣
↪be applied as data augmentation to the training dataset. This transform is␣
↪applied before base_transform. If provided, the augmentors option is ignored.",
         "type": "object"
      },
      "plot_options": {
         "title": "Plot Options",
         "description": "Options to control plotting.",
         "default": {
            "transform": {
               "__version__": "1.3.0",
               "transform": {
                  "__class_fullname__": "rastervision.pytorch_learner.utils.utils.
↪MinMaxNormalize",
                  "always_apply": false,
                  "p": 1.0,
                  "min_val": 0.0,
```

(continues on next page)

(continued from previous page)


```
                    "max_val": 1.0,
                    "dtype": 5
                }
            },
            "channel_display_groups": null,
            "type_hint": "plot_options"
        },
        "allOf": [
            {
                "$ref": "#/definitions/PlotOptions"
            }
        ]
    },
    "preview_batch_limit": {
        "title": "Preview Batch Limit",
        "description": "Optional limit on the number of items in the preview plots␣
↪produced during training.",
        "type": "integer"
    },
    "type_hint": {
        "title": "Type Hint",
        "default": "data",
        "enum": [
            "data"
        ],
        "type": "string"
    }
  },
  "additionalProperties": false,
  "definitions": {
    "PlotOptions": {
        "title": "PlotOptions",
        "description": "Config related to plotting.",
        "type": "object",
        "properties": {
            "transform": {
                "title": "Transform",
                "description": "An Albumentations transform serialized as a dict␣
↪that will be applied to each image before it is plotted. Mainly useful for␣
↪undoing any data transformation that you do not want included in the plot, such␣
↪as normalization. The default value will shift and scale the image so the values␣
↪range from 0.0 to 1.0 which is the expected range for the plotting function. This␣
↪default is useful for cases where the values after normalization are close to␣
↪zero which makes the plot difficult to see.",
                "default": {
                    "__version__": "1.3.0",
                    "transform": {
                        "__class_fullname__": "rastervision.pytorch_learner.utils.
↪utils.MinMaxNormalize",
                        "always_apply": false,
                        "p": 1.0,
                        "min_val": 0.0,
```


(continues on next page)

(continued from previous page)


```
                    "max_val": 1.0,
                    "dtype": 5
                }
            },
            "type": "object"
        },
        "channel_display_groups": {
            "title": "Channel Display Groups",
            "description": "Groups of image channels to display together as a␣
↪subplot when plotting the data and predictions. Can be a list or tuple of groups␣
↪(e.g. [(0, 1, 2), (3,)]) or a dict containing title-to-group mappings (e.g. {\
↪"RGB\": [0, 1, 2], \"IR\": [3]}), where each group is a list or tuple of channel␣
↪indices and title is a string that will be used as the title of the subplot for␣
↪that group.",
            "anyOf": [
                {
                    "type": "object",
                    "additionalProperties": {
                        "type": "array",
                        "items": {
                            "type": "integer",
                            "minimum": 0
                        }
                    }
                },
                {
                    "type": "array",
                    "items": {
                        "type": "array",
                        "items": {
                            "type": "integer",
                            "minimum": 0
                        }
                    }
                }
            ]
        },
        "type_hint": {
            "title": "Type Hint",
            "default": "plot_options",
            "enum": [
                "plot_options"
            ],
            "type": "string"
        }
    },
    "additionalProperties": false
  }
 }
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *aug_transform (Optional[dict])*
- *augmentors (List[str])*
- *base_transform (Optional[dict])*
- *class_colors (Optional[List[Union[str, Tuple[int, int, int]]]])*
- *class_names (List[str])*
- *img_channels (Optional[pydantic.types.PositiveInt])*
- *img_sz (pydantic.types.PositiveInt)*
- *num_workers (int)*
- *plot_options (Optional[rastervision.pytorch_learner.learner_config.PlotOptions])*
- *preview_batch_limit (Optional[int])*
- *train_sz (Optional[int])*
- *train_sz_rel (Optional[float])*
- *type_hint (Literal['data'])*

Validators

- *ensure_class_colors* » `all fields`
- *validate_albumentation_transform* » *aug_transform*
- *validate_albumentation_transform* » *base_transform*
- *validate_augmentors* » *augmentors*
- *validate_plot_options* » `all fields`

field aug_transform: Optional[dict] = None

An Albumentations transform serialized as a dict that will be applied as data augmentation to the training dataset. This transform is applied before base_transform. If provided, the augmentors option is ignored.

Validated by

- *ensure_class_colors*
- *validate_albumentation_transform*
- *validate_plot_options*

field augmentors: List[str] = ['RandomRotate90', 'HorizontalFlip', 'VerticalFlip']

Names of albumentations augmentors to use for training batches. Choices include: [‘Blur’, ‘RandomRotate90’, ‘HorizontalFlip’, ‘VerticalFlip’, ‘GaussianBlur’, ‘GaussNoise’, ‘RGBShift’, ‘ToGray’]. Alternatively, a custom transform can be provided via the aug_transform option.

Validated by

- *ensure_class_colors*
- *validate_augmentors*

- *validate_plot_options*

field base_transform: Optional[dict] = None

An Albumentations transform serialized as a dict that will be applied to all datasets: training, validation, and test. This transformation is in addition to the resizing due to img_sz. This is useful for, for example, applying the same normalization to all datasets.

Validated by

- *ensure_class_colors*
- *validate_albumentation_transform*
- *validate_plot_options*

field class_colors: Optional[List[Union[str, Tuple[int, int, int]]]] = None

Colors used to display classes. Can be color 3-tuples in list form.

Validated by

- *ensure_class_colors*
- *validate_plot_options*

field class_names: List[str] = []

Names of classes.

Validated by

- *ensure_class_colors*
- *validate_plot_options*

field img_channels: Optional[PositiveInt] = None

The number of channels of the training images.

Constraints

- **exclusiveMinimum** = 0

Validated by

- *ensure_class_colors*
- *validate_plot_options*

field img_sz: PositiveInt = 256

Length of a side of each image in pixels. This is the size to transform it to during training, not the size in the raw dataset.

Constraints

- **exclusiveMinimum** = 0

Validated by

- *ensure_class_colors*
- *validate_plot_options*

field num_workers: int = 4

Number of workers to use when DataLoader makes batches.

Validated by

- *ensure_class_colors*

- *validate_plot_options*

field plot_options: Optional[*PlotOptions*] = PlotOptions(transform={'__version__': '1.3.0', 'transform': {'__class_fullname__': 'rastervision.pytorch_learner.utils.utils.MinMaxNormalize', 'always_apply': False, 'p': 1.0, 'min_val': 0.0, 'max_val': 1.0, 'dtype': 5}}, channel_display_groups=None)

Options to control plotting.

Validated by

- *ensure_class_colors*
- *validate_plot_options*

field preview_batch_limit: Optional[int] = None

Optional limit on the number of items in the preview plots produced during training.

Validated by

- *ensure_class_colors*
- *validate_plot_options*

field train_sz: Optional[int] = None

If set, the number of training images to use. If fewer images exist, then an exception will be raised.

Validated by

- *ensure_class_colors*
- *validate_plot_options*

field train_sz_rel: Optional[float] = None

If set, the proportion of training images to use.

Validated by

- *ensure_class_colors*
- *validate_plot_options*

field type_hint: Literal['data'] = 'data'

Validated by

- *ensure_class_colors*
- *validate_plot_options*

build(*tmp_dir: str*, *overfit_mode: bool = False*, *test_mode: bool = False*) → Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

Build and return train, val, and test datasets.

Parameters

- **tmp_dir** (*str*) –
- **overfit_mode** (*bool*) –
- **test_mode** (*bool*) –

Return type

Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

validator ensure_class_colors » *all fields*

Parameters
values (*dict*) –

Return type
dict

get_bbox_params() → Optional[BboxParams]

Returns BboxParams used by albumentations for data augmentation.

Return type
Optional[*BboxParams*]

get_custom_albumentations_transforms() → List[dict]

Returns all custom transforms found in this config.

This should return all serialized albumentations transforms with a 'lambda_transforms_path' field contained in this config or in any of its members no matter how deeply neseted.

The pupose is to make it easier to adjust their paths all at once while saving to or loading from a bundle.

Return type
List[dict]

get_data_transforms() → Tuple[BasicTransform, BasicTransform]

Get albumentations transform objects for data augmentation.

Returns
a transform that doesn't do any data augmentation 2nd tuple arg: a transform with data augmentation

Return type
1st tuple arg

make_datasets() → Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

Return type
Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

random_subset_dataset(*ds: torch.utils.data.Dataset*, *size: Optional[int] = None*, *fraction: Optional[ConstrainedFloatValue] = None*) → torch.utils.data.Subset

Parameters

- **ds** (*torch.utils.data.Dataset*) –
- **size** (*Optional[int]*) –
- **fraction** (*Optional[ConstrainedFloatValue]*) –

Return type
torch.utils.data.Subset

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(**args*, ***kwargs*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validator validate_augmentors » augmentors

Parameters
v (*str*) –

Return type
str

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises
ConfigError – if field is invalid

validator validate_plot_options » *all fields*

Parameters
values (*dict*) –

Return type
dict

property num_classes

ExternalModuleConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model ExternalModuleConfig

Config describing an object to be loaded via Torch Hub.


```
{
   "title": "ExternalModuleConfig",
   "description": "Config describing an object to be loaded via Torch Hub.",
   "type": "object",
   "properties": {
      "uri": {
```


(continues on next page)

(continued from previous page)

```
            "title": "Uri",
            "description": "Local uri of a zip file, or local uri of a directory,or␣
↪remote uri of zip file.",
            "minLength": 1,
            "type": "string"
         },
         "github_repo": {
            "title": "Github Repo",
            "description": "<repo-owner>/<repo-name>[:tag]",
            "pattern": ".+/.+",
            "type": "string"
         },
         "name": {
            "title": "Name",
            "description": "Name of the folder in which to extract/copy the definition␣
↪files.",
            "minLength": 1,
            "type": "string"
         },
         "entrypoint": {
            "title": "Entrypoint",
            "description": "Name of a callable present in hubconf.py. See docs for␣
↪torch.hub for details.",
            "minLength": 1,
            "type": "string"
         },
         "entrypoint_args": {
            "title": "Entrypoint Args",
            "description": "Args to pass to the entrypoint. Must be serializable.",
            "default": [],
            "type": "array",
            "items": {}
         },
         "entrypoint_kwargs": {
            "title": "Entrypoint Kwargs",
            "description": "Keyword args to pass to the entrypoint. Must be␣
↪serializable.",
            "default": {},
            "type": "object"
         },
         "force_reload": {
            "title": "Force Reload",
            "description": "Force reload of module definition.",
            "default": false,
            "type": "boolean"
         },
         "type_hint": {
            "title": "Type Hint",
            "default": "external-module",
            "enum": [
               "external-module"
            ],
```

(continues on next page)

(continued from previous page)

```
            "type": "string"
        }
    },
    "required": [
        "entrypoint"
    ],
    "additionalProperties": false
}
```

Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *entrypoint (rastervision.pytorch_learner.learner_config.ConstrainedStrValue)*
- *entrypoint_args (list)*
- *entrypoint_kwargs (dict)*
- *force_reload (bool)*
- *github_repo (Optional[rastervision.pytorch_learner.learner_config.ConstrainedStrValue])*
- *name (Optional[rastervision.pytorch_learner.learner_config.ConstrainedStrValue])*
- *type_hint (Literal['external-module'])*
- *uri (Optional[rastervision.pytorch_learner.learner_config.ConstrainedStrValue])*

field entrypoint: ConstrainedStrValue [Required]

Name of a callable present in hubconf.py. See docs for torch.hub for details.

Constraints

- **minLength** = 1

Validated by

- *check_either_uri_or_repo*

field entrypoint_args: list = []

Args to pass to the entrypoint. Must be serializable.

Validated by

- *check_either_uri_or_repo*

field entrypoint_kwargs: dict = {}

Keyword args to pass to the entrypoint. Must be serializable.

Validated by

- *check_either_uri_or_repo*

field force_reload: bool = False

Force reload of module definition.

Validated by

- *check_either_uri_or_repo*

field github_repo: Optional[ConstrainedStrValue] = None

<repo-owner>/<repo-name>[:tag]

Constraints

- **pattern** = .+/.+

Validated by

- *check_either_uri_or_repo*

field name: Optional[ConstrainedStrValue] = None

Name of the folder in which to extract/copy the definition files.

Constraints

- **minLength** = 1

Validated by

- *check_either_uri_or_repo*

field type_hint: Literal['external-module'] = 'external-module'

Validated by

- *check_either_uri_or_repo*

field uri: Optional[ConstrainedStrValue] = None

Local uri of a zip file, or local uri of a directory,or remote uri of zip file.

Constraints

- **minLength** = 1

Validated by

- *check_either_uri_or_repo*

build(*save_dir: str*, *hubconf_dir: Optional[str] = None*) → Any

Load an external module via torch.hub.

Note: Loading a PyTorch module is the typical use case, but there are no type restrictions on the object loaded through torch.hub.

Parameters

- **save_dir** (*str, optional*) – The module def will be saved here.
- **hubconf_dir** (*str, optional*) – Path to existing definition. If provided, the definition will not be fetched from the external source but instead from this dir. Defaults to None.

Returns

The module loaded via torch.hub.

Return type

Any

validator check_either_uri_or_repo » *all fields*

Parameters
values (*dict*) –

Return type
dict

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(**args*, ***kwargs*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises
ConfigError – if field is invalid

GeoDataConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model GeoDataConfig

Configure *GeoDatasets*.

See *rastervision.pytorch_learner.dataset.dataset*.


```
{
   "title": "GeoDataConfig",
   "description": "Configure :class:`GeoDatasets <.GeoDataset>`.\n\nSee :mod:
↪`rastervision.pytorch_learner.dataset.dataset`.",
   "type": "object",
```


(continues on next page)

(continued from previous page)

```
   "properties": {
      "class_names": {
         "title": "Class Names",
         "description": "Names of classes.",
         "default": [],
         "type": "array",
         "items": {
            "type": "string"
         }
      },
      "class_colors": {
         "title": "Class Colors",
         "description": "Colors used to display classes. Can be color 3-tuples in␣
↪list form.",
         "type": "array",
         "items": {
            "anyOf": [
               {
                  "type": "string"
               },
               {
                  "type": "array",
                  "minItems": 3,
                  "maxItems": 3,
                  "items": [
                     {
                        "type": "integer"
                     },
                     {
                        "type": "integer"
                     },
                     {
                        "type": "integer"
                     }
                  ]
               }
            ]
         }
      },
      "img_channels": {
         "title": "Img Channels",
         "description": "The number of channels of the training images.",
         "exclusiveMinimum": 0,
         "type": "integer"
      },
      "img_sz": {
         "title": "Img Sz",
         "description": "Length of a side of each image in pixels. This is the size␣
↪to transform it to during training, not the size in the raw dataset.",
         "default": 256,
         "exclusiveMinimum": 0,
         "type": "integer"
```

(continues on next page)

(continued from previous page)

```
      },
      "train_sz": {
         "title": "Train Sz",
         "description": "If set, the number of training images to use. If fewer␣
↪images exist, then an exception will be raised.",
         "type": "integer"
      },
      "train_sz_rel": {
         "title": "Train Sz Rel",
         "description": "If set, the proportion of training images to use.",
         "type": "number"
      },
      "num_workers": {
         "title": "Num Workers",
         "description": "Number of workers to use when DataLoader makes batches.",
         "default": 4,
         "type": "integer"
      },
      "augmentors": {
         "title": "Augmentors",
         "description": "Names of albumentations augmentors to use for training␣
↪batches. Choices include: ['Blur', 'RandomRotate90', 'HorizontalFlip',
↪'VerticalFlip', 'GaussianBlur', 'GaussNoise', 'RGBShift', 'ToGray'].␣
↪Alternatively, a custom transform can be provided via the aug_transform option.",
         "default": [
            "RandomRotate90",
            "HorizontalFlip",
            "VerticalFlip"
         ],
         "type": "array",
         "items": {
            "type": "string"
         }
      },
      "base_transform": {
         "title": "Base Transform",
         "description": "An Albumentations transform serialized as a dict that will␣
↪be applied to all datasets: training, validation, and test. This transformation␣
↪is in addition to the resizing due to img_sz. This is useful for, for example,␣
↪applying the same normalization to all datasets.",
         "type": "object"
      },
      "aug_transform": {
         "title": "Aug Transform",
         "description": "An Albumentations transform serialized as a dict that will␣
↪be applied as data augmentation to the training dataset. This transform is␣
↪applied before base_transform. If provided, the augmentors option is ignored.",
         "type": "object"
      },
      "plot_options": {
         "title": "Plot Options",
         "description": "Options to control plotting.",
```

(continues on next page)

(continued from previous page)


```
         "default": {
            "transform": {
               "__version__": "1.3.0",
               "transform": {
                  "__class_fullname__": "rastervision.pytorch_learner.utils.utils.
↪MinMaxNormalize",
                  "always_apply": false,
                  "p": 1.0,
                  "min_val": 0.0,
                  "max_val": 1.0,
                  "dtype": 5
               }
            },
            "channel_display_groups": null,
            "type_hint": "plot_options"
         },
         "allOf": [
            {
               "$ref": "#/definitions/PlotOptions"
            }
         ]
      },
      "preview_batch_limit": {
         "title": "Preview Batch Limit",
         "description": "Optional limit on the number of items in the preview plots␣
↪produced during training.",
         "type": "integer"
      },
      "type_hint": {
         "title": "Type Hint",
         "default": "geo_data",
         "enum": [
            "geo_data"
         ],
         "type": "string"
      },
      "scene_dataset": {
         "$ref": "#/definitions/DatasetConfig"
      },
      "window_opts": {
         "title": "Window Opts",
         "default": {},
         "anyOf": [
            {
               "$ref": "#/definitions/GeoDataWindowConfig"
            },
            {
               "type": "object",
               "additionalProperties": {
                  "$ref": "#/definitions/GeoDataWindowConfig"
               }
            }
```


(continues on next page)

(continued from previous page)


```
        ]
      }
   },
   "additionalProperties": false,
   "definitions": {
      "PlotOptions": {
         "title": "PlotOptions",
         "description": "Config related to plotting.",
         "type": "object",
         "properties": {
            "transform": {
               "title": "Transform",
               "description": "An Albumentations transform serialized as a dict␣
↪that will be applied to each image before it is plotted. Mainly useful for␣
↪undoing any data transformation that you do not want included in the plot, such␣
↪as normalization. The default value will shift and scale the image so the values␣
↪range from 0.0 to 1.0 which is the expected range for the plotting function. This␣
↪default is useful for cases where the values after normalization are close to␣
↪zero which makes the plot difficult to see.",
               "default": {
                  "__version__": "1.3.0",
                  "transform": {
                     "__class_fullname__": "rastervision.pytorch_learner.utils.
↪utils.MinMaxNormalize",
                     "always_apply": false,
                     "p": 1.0,
                     "min_val": 0.0,
                     "max_val": 1.0,
                     "dtype": 5
                  }
               },
               "type": "object"
            },
            "channel_display_groups": {
               "title": "Channel Display Groups",
               "description": "Groups of image channels to display together as a␣
↪subplot when plotting the data and predictions. Can be a list or tuple of groups␣
↪(e.g. [(0, 1, 2), (3,)]) or a dict containing title-to-group mappings (e.g. {\
↪"RGB\": [0, 1, 2], \"IR\": [3]}), where each group is a list or tuple of channel␣
↪indices and title is a string that will be used as the title of the subplot for␣
↪that group.",
               "anyOf": [
                  {
                     "type": "object",
                     "additionalProperties": {
                        "type": "array",
                        "items": {
                           "type": "integer",
                           "minimum": 0
                        }
                     }
                  },
```


(continues on next page)

(continued from previous page)

```
                {
                    "type": "array",
                    "items": {
                        "type": "array",
                        "items": {
                            "type": "integer",
                            "minimum": 0
                        }
                    }
                }
            ]
        },
        "type_hint": {
            "title": "Type Hint",
            "default": "plot_options",
            "enum": [
                "plot_options"
            ],
            "type": "string"
        }
    },
    "additionalProperties": false
},
"ClassConfig": {
    "title": "ClassConfig",
    "description": "Configure class information for a machine learning task.",
    "type": "object",
    "properties": {
        "names": {
            "title": "Names",
            "description": "Names of classes. The i-th class in this list will␣
↪have class ID = i.",
            "type": "array",
            "items": {
                "type": "string"
            }
        },
        "colors": {
            "title": "Colors",
            "description": "Colors used to visualize classes. Can be color␣
↪strings accepted by matplotlib or RGB tuples. If None, a random color will be␣
↪auto-generated for each class.",
            "type": "array",
            "items": {
                "anyOf": [
                    {
                        "type": "string"
                    },
                    {
                        "type": "array",
                        "items": {}
                    }
```

(continues on next page)

(continued from previous page)


```
                ]
             }
          },
          "null_class": {
             "title": "Null Class",
             "description": "Optional name of class in `names` to use as the null␣
↪class. This is used in semantic segmentation to represent the label for imagery␣
↪pixels that are NODATA or that are missing a label. If None and the class names␣
↪include \"null\", it will automatically be used as the null class. If None, and␣
↪this Config is part of a SemanticSegmentationConfig, a null class will be added␣
↪automatically.",
             "type": "string"
          },
          "type_hint": {
             "title": "Type Hint",
             "default": "class_config",
             "enum": [
                "class_config"
             ],
             "type": "string"
          }
       },
       "required": [
          "names"
       ],
       "additionalProperties": false
    },
    "RasterTransformerConfig": {
       "title": "RasterTransformerConfig",
       "description": "Configure a :class:`.RasterTransformer`.",
       "type": "object",
       "properties": {
          "type_hint": {
             "title": "Type Hint",
             "default": "raster_transformer",
             "enum": [
                "raster_transformer"
             ],
             "type": "string"
          }
       },
       "additionalProperties": false
    },
    "RasterSourceConfig": {
       "title": "RasterSourceConfig",
       "description": "Configure a :class:`.RasterSource`.",
       "type": "object",
       "properties": {
          "channel_order": {
             "title": "Channel Order",
             "description": "The sequence of channel indices to use when reading␣
↪imagery.",
```


(continues on next page)

(continued from previous page)


```
                "type": "array",
                "items": {
                    "type": "integer"
                }
            },
            "transformers": {
                "title": "Transformers",
                "default": [],
                "type": "array",
                "items": {
                    "$ref": "#/definitions/RasterTransformerConfig"
                }
            },
            "extent": {
                "title": "Extent",
                "description": "Use-specified extent in pixel coords in the form␣
↪(ymin, xmin, ymax, xmax). Useful for cropping the raster source so that only part␣
↪of the raster is read from.",
                "type": "array",
                "minItems": 4,
                "maxItems": 4,
                "items": [
                    {
                        "type": "integer"
                    },
                    {
                        "type": "integer"
                    },
                    {
                        "type": "integer"
                    },
                    {
                        "type": "integer"
                    }
                ]
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "raster_source",
                "enum": [
                    "raster_source"
                ],
                "type": "string"
            }
        },
        "additionalProperties": false
    },
    "LabelSourceConfig": {
        "title": "LabelSourceConfig",
        "description": "Configure a :class:`.LabelSource`.",
        "type": "object",
        "properties": {
```


(continues on next page)

(continued from previous page)


```
            "type_hint": {
               "title": "Type Hint",
               "default": "label_source",
               "enum": [
                  "label_source"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      },
      "LabelStoreConfig": {
         "title": "LabelStoreConfig",
         "description": "Configure a :class:`.LabelStore`.",
         "type": "object",
         "properties": {
            "type_hint": {
               "title": "Type Hint",
               "default": "label_store",
               "enum": [
                  "label_store"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      },
      "SceneConfig": {
         "title": "SceneConfig",
         "description": "Configure a :class:`.Scene` comprising raster data &␣
↪labels for an AOI.\n    ",
         "type": "object",
         "properties": {
            "id": {
               "title": "Id",
               "type": "string"
            },
            "raster_source": {
               "$ref": "#/definitions/RasterSourceConfig"
            },
            "label_source": {
               "$ref": "#/definitions/LabelSourceConfig"
            },
            "label_store": {
               "$ref": "#/definitions/LabelStoreConfig"
            },
            "aoi_uris": {
               "title": "Aoi Uris",
               "description": "List of URIs of GeoJSON files that define the AOIs␣
↪for the scene. Each polygon defines an AOI which is a piece of the scene that is␣
↪assumed to be fully labeled and usable for training or validation. The AOIs are␣
↪assumed to be in EPSG:4326 coordinates.",
```


(continues on next page)

(continued from previous page)


```
                "type": "array",
                "items": {
                   "type": "string"
                }
             },
             "type_hint": {
                "title": "Type Hint",
                "default": "scene",
                "enum": [
                   "scene"
                ],
                "type": "string"
             }
          },
          "required": [
             "id",
             "raster_source"
          ],
          "additionalProperties": false
       },
       "DatasetConfig": {
          "title": "DatasetConfig",
          "description": "Configure train, validation, and test splits for a dataset.
↪",
          "type": "object",
          "properties": {
             "class_config": {
                "$ref": "#/definitions/ClassConfig"
             },
             "train_scenes": {
                "title": "Train Scenes",
                "type": "array",
                "items": {
                   "$ref": "#/definitions/SceneConfig"
                }
             },
             "validation_scenes": {
                "title": "Validation Scenes",
                "type": "array",
                "items": {
                   "$ref": "#/definitions/SceneConfig"
                }
             },
             "test_scenes": {
                "title": "Test Scenes",
                "default": [],
                "type": "array",
                "items": {
                   "$ref": "#/definitions/SceneConfig"
                }
             },
             "scene_groups": {
```


(continues on next page)

(continued from previous page)


```
                "title": "Scene Groups",
                "description": "Groupings of scenes. Should be a dict of the form: {
↪<group-name>: Set(scene_id_1, scene_id_2, ...)}. Three groups are added by␣
↪default: \"train_scenes\", \"validation_scenes\", and \"test_scenes\"",
                "default": {},
                "type": "object",
                "additionalProperties": {
                   "type": "array",
                   "items": {
                      "type": "string"
                   },
                   "uniqueItems": true
                }
             },
             "type_hint": {
                "title": "Type Hint",
                "default": "dataset",
                "enum": [
                   "dataset"
                ],
                "type": "string"
             }
          },
          "required": [
             "class_config",
             "train_scenes",
             "validation_scenes"
          ],
          "additionalProperties": false
       },
       "GeoDataWindowMethod": {
          "title": "GeoDataWindowMethod",
          "description": "An enumeration.",
          "enum": [
             "sliding",
             "random"
          ]
       },
       "GeoDataWindowConfig": {
          "title": "GeoDataWindowConfig",
          "description": "Configure a :class:`.GeoDataset`.\n\nSee :mod:
↪`rastervision.pytorch_learner.dataset.dataset`.",
          "type": "object",
          "properties": {
             "method": {
                "default": "sliding",
                "allOf": [
                   {
                      "$ref": "#/definitions/GeoDataWindowMethod"
                   }
                ]
             },
```


(continues on next page)

(continued from previous page)

```
            "size": {
               "title": "Size",
               "description": "If method = sliding, this is the size of sliding␣
↪window. If method = random, this is the size that all the windows are resized to␣
↪before they are returned. If method = random and neither size_lims nor h_lims and␣
↪w_lims have been specified, then size_lims is set to (size, size + 1).",
               "anyOf": [
                  {
                     "type": "integer",
                     "exclusiveMinimum": 0
                  },
                  {
                     "type": "array",
                     "minItems": 2,
                     "maxItems": 2,
                     "items": [
                        {
                           "type": "integer",
                           "exclusiveMinimum": 0
                        },
                        {
                           "type": "integer",
                           "exclusiveMinimum": 0
                        }
                     ]
                  }
               ]
            },
            "stride": {
               "title": "Stride",
               "description": "Stride of sliding window. Only used if method =␣
↪sliding.",
               "anyOf": [
                  {
                     "type": "integer",
                     "exclusiveMinimum": 0
                  },
                  {
                     "type": "array",
                     "minItems": 2,
                     "maxItems": 2,
                     "items": [
                        {
                           "type": "integer",
                           "exclusiveMinimum": 0
                        },
                        {
                           "type": "integer",
                           "exclusiveMinimum": 0
                        }
                     ]
                  }
```

(continues on next page)

(continued from previous page)

```
                ]
            },
            "padding": {
                "title": "Padding",
                "description": "How many pixels are windows allowed to overflow the␣
↪edges of the raster source.",
                "anyOf": [
                    {
                        "type": "integer",
                        "minimum": 0
                    },
                    {
                        "type": "array",
                        "minItems": 2,
                        "maxItems": 2,
                        "items": [
                            {
                                "type": "integer",
                                "minimum": 0
                            },
                            {
                                "type": "integer",
                                "minimum": 0
                            }
                        ]
                    }
                ]
            },
            "pad_direction": {
                "title": "Pad Direction",
                "description": "If \"end\", only pad ymax and xmax (bottom and␣
↪right). If \"start\", only pad ymin and xmin (top and left). If \"both\", pad all␣
↪sides. Has no effect if paddiong is zero. Defaults to \"end\".",
                "default": "end",
                "enum": [
                    "both",
                    "start",
                    "end"
                ],
                "type": "string"
            },
            "size_lims": {
                "title": "Size Lims",
                "description": "[min, max) interval from which window sizes will be␣
↪uniformly randomly sampled. The upper limit is exclusive. To fix the size to a␣
↪constant value, use size_lims = (sz, sz + 1). Only used if method = random.␣
↪Specify either size_lims or h_lims and w_lims, but not both. If neither size_lims␣
↪nor h_lims and w_lims have been specified, then this will be set to (size, size +␣
↪1).",
                "type": "array",
                "minItems": 2,
                "maxItems": 2,
```

(continues on next page)

(continued from previous page)

```
                "items": [
                    {
                        "type": "integer",
                        "exclusiveMinimum": 0
                    },
                    {
                        "type": "integer",
                        "exclusiveMinimum": 0
                    }
                ]
            },
            "h_lims": {
                "title": "H Lims",
                "description": "[min, max] interval from which window heights will␣
↪be uniformly randomly sampled. Only used if method = random.",
                "type": "array",
                "minItems": 2,
                "maxItems": 2,
                "items": [
                    {
                        "type": "integer",
                        "exclusiveMinimum": 0
                    },
                    {
                        "type": "integer",
                        "exclusiveMinimum": 0
                    }
                ]
            },
            "w_lims": {
                "title": "W Lims",
                "description": "[min, max] interval from which window widths will be␣
↪uniformly randomly sampled. Only used if method = random.",
                "type": "array",
                "minItems": 2,
                "maxItems": 2,
                "items": [
                    {
                        "type": "integer",
                        "exclusiveMinimum": 0
                    },
                    {
                        "type": "integer",
                        "exclusiveMinimum": 0
                    }
                ]
            },
            "max_windows": {
                "title": "Max Windows",
                "description": "Max allowed reads from a GeoDataset. Only used if␣
↪method = random.",
                "default": 10000,
```

(continues on next page)

(continued from previous page)


```
                "minimum": 0,
                "type": "integer"
            },
            "max_sample_attempts": {
                "title": "Max Sample Attempts",
                "description": "Max attempts when trying to find a window within the␣
↪AOI of a scene. Only used if method = random and the scene has aoi_polygons␣
↪specified.",
                "default": 100,
                "exclusiveMinimum": 0,
                "type": "integer"
            },
            "efficient_aoi_sampling": {
                "title": "Efficient Aoi Sampling",
                "description": "If the scene has AOIs, sampling windows at random␣
↪anywhere in the extent and then checking if they fall within any of the AOIs can␣
↪be very inefficient. This flag enables the use of an alternate algorithm that␣
↪only samples window locations inside the AOIs. Only used if method = random and␣
↪the scene has aoi_polygons specified. Defaults to True",
                "default": true,
                "type": "boolean"
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "geo_data_window",
                "enum": [
                    "geo_data_window"
                ],
                "type": "string"
            }
        },
        "required": [
            "size"
        ],
        "additionalProperties": false
    }
  }
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *aug_transform (Optional[dict])*
- *augmentors (List[str])*
- *base_transform (Optional[dict])*
- *class_colors (Optional[List[Union[str, Tuple[int, int, int]]]])*
- *class_names (List[str])*

- *img_channels (Optional[pydantic.types.PositiveInt])*
- *img_sz (pydantic.types.PositiveInt)*
- *num_workers (int)*
- *plot_options (Optional[rastervision.pytorch_learner.learner_config.PlotOptions])*
- *preview_batch_limit (Optional[int])*
- *scene_dataset (Optional[rastervision.core.data.dataset_config.DatasetConfig])*
- *train_sz (Optional[int])*
- *train_sz_rel (Optional[float])*
- *type_hint (Literal['geo_data'])*
- *window_opts (Union[rastervision.pytorch_learner.learner_config.GeoDataWindowConfig, Dict[str, rastervision.pytorch_learner.learner_config.GeoDataWindowConfig]])*

Validators

- *ensure_class_colors* » `all fields`
- *get_class_info_from_class_config_if_needed* » `all fields`
- *validate_albumentation_transform* » *aug_transform*
- *validate_albumentation_transform* » *base_transform*
- *validate_augmentors* » *augmentors*
- *validate_plot_options* » `all fields`
- *validate_window_opts* » *window_opts*

field aug_transform: Optional[dict] = None

An Albumentations transform serialized as a dict that will be applied as data augmentation to the training dataset. This transform is applied before base_transform. If provided, the augmentors option is ignored.

Validated by

- *ensure_class_colors*
- *get_class_info_from_class_config_if_needed*
- *validate_albumentation_transform*
- *validate_plot_options*

field augmentors: List[str] = ['RandomRotate90', 'HorizontalFlip', 'VerticalFlip']

Names of albumentations augmentors to use for training batches. Choices include: [‘Blur’, ‘RandomRotate90’, ‘HorizontalFlip’, ‘VerticalFlip’, ‘GaussianBlur’, ‘GaussNoise’, ‘RGBShift’, ‘ToGray’]. Alternatively, a custom transform can be provided via the aug_transform option.

Validated by

- *ensure_class_colors*
- *get_class_info_from_class_config_if_needed*
- *validate_augmentors*

- *validate_plot_options*

field base_transform: Optional[dict] = None

An Albumentations transform serialized as a dict that will be applied to all datasets: training, validation, and test. This transformation is in addition to the resizing due to img_sz. This is useful for, for example, applying the same normalization to all datasets.

Validated by

- *ensure_class_colors*
- *get_class_info_from_class_config_if_needed*
- *validate_albumentation_transform*
- *validate_plot_options*

field class_colors: Optional[List[Union[str, Tuple[int, int, int]]]] = None

Colors used to display classes. Can be color 3-tuples in list form.

Validated by

- *ensure_class_colors*
- *get_class_info_from_class_config_if_needed*
- *validate_plot_options*

field class_names: List[str] = []

Names of classes.

Validated by

- *ensure_class_colors*
- *get_class_info_from_class_config_if_needed*
- *validate_plot_options*

field img_channels: Optional[PositiveInt] = None

The number of channels of the training images.

Constraints

- **exclusiveMinimum** = 0

Validated by

- *ensure_class_colors*
- *get_class_info_from_class_config_if_needed*
- *validate_plot_options*

field img_sz: PositiveInt = 256

Length of a side of each image in pixels. This is the size to transform it to during training, not the size in the raw dataset.

Constraints

- **exclusiveMinimum** = 0

Validated by

- *ensure_class_colors*
- *get_class_info_from_class_config_if_needed*

- *validate_plot_options*

field num_workers: int = 4

Number of workers to use when DataLoader makes batches.

Validated by

- *ensure_class_colors*
- *get_class_info_from_class_config_if_needed*
- *validate_plot_options*

field plot_options: Optional[*PlotOptions*] = PlotOptions(transform={'__version__': '1.3.0', 'transform': {'__class_fullname__': 'rastervision.pytorch_learner.utils.utils.MinMaxNormalize', 'always_apply': False, 'p': 1.0, 'min_val': 0.0, 'max_val': 1.0, 'dtype': 5}}, channel_display_groups=None)

Options to control plotting.

Validated by

- *ensure_class_colors*
- *get_class_info_from_class_config_if_needed*
- *validate_plot_options*

field preview_batch_limit: Optional[int] = None

Optional limit on the number of items in the preview plots produced during training.

Validated by

- *ensure_class_colors*
- *get_class_info_from_class_config_if_needed*
- *validate_plot_options*

field scene_dataset: Optional[*DatasetConfig*] = None

Validated by

- *ensure_class_colors*
- *get_class_info_from_class_config_if_needed*
- *validate_plot_options*

field train_sz: Optional[int] = None

If set, the number of training images to use. If fewer images exist, then an exception will be raised.

Validated by

- *ensure_class_colors*
- *get_class_info_from_class_config_if_needed*
- *validate_plot_options*

field train_sz_rel: Optional[float] = None

If set, the proportion of training images to use.

Validated by

- *ensure_class_colors*
- *get_class_info_from_class_config_if_needed*

- *validate_plot_options*

field type_hint: Literal['geo_data'] = 'geo_data'

Validated by

- *ensure_class_colors*
- *get_class_info_from_class_config_if_needed*
- *validate_plot_options*

field window_opts: Union[*GeoDataWindowConfig*, Dict[str, *GeoDataWindowConfig*]] = {}

Validated by

- *ensure_class_colors*
- *get_class_info_from_class_config_if_needed*
- *validate_plot_options*
- *validate_window_opts*

build(*tmp_dir: str*, *overfit_mode: bool = False*, *test_mode: bool = False*) → Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

Build and return train, val, and test datasets.

Parameters

- **tmp_dir** (*str*) –
- **overfit_mode** (*bool*) –
- **test_mode** (*bool*) –

Return type

Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

build_scenes(*tmp_dir: str*) → Tuple[List[*Scene*], List[*Scene*], List[*Scene*]]

Build training, validation, and test scenes.

Parameters

tmp_dir (*str*) –

Return type

Tuple[*List*[Scene], *List*[Scene], *List*[Scene]]

validator ensure_class_colors » *all fields*

Parameters

values (*dict*) –

Return type

dict

get_bbox_params() → Optional[BboxParams]

Returns BboxParams used by albumentations for data augmentation.

Return type

Optional[*BboxParams*]

validator get_class_info_from_class_config_if_needed » *all fields*

Parameters
values (*dict*) –

Return type
dict

get_custom_albumentations_transforms() → List[dict]

Returns all custom transforms found in this config.

This should return all serialized albumentations transforms with a 'lambda_transforms_path' field contained in this config or in any of its members no matter how deeply neseted.

The pupose is to make it easier to adjust their paths all at once while saving to or loading from a bundle.

Return type
List[dict]

get_data_transforms() → Tuple[BasicTransform, BasicTransform]

Get albumentations transform objects for data augmentation.

Returns
a transform that doesn't do any data augmentation 2nd tuple arg: a transform with data augmentation

Return type
1st tuple arg

make_datasets(*tmp_dir: str*, *train_tf: Optional[BasicTransform] = None*, *val_tf: Optional[BasicTransform] = None*, *test_tf: Optional[BasicTransform] = None*, ***kwargs*) → Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

Make training, validation, and test datasets.

Parameters

- **tmp_dir** (*str*) – Temporary directory to be used for building scenes.
- **train_tf** (*Optional[A.BasicTransform], optional*) – Transform for the training dataset. Defaults to None.
- **val_tf** (*Optional[A.BasicTransform], optional*) – Transform for the validation dataset. Defaults to None.
- **test_tf** (*Optional[A.BasicTransform], optional*) – Transform for the test dataset. Defaults to None.
- **kwargs** – Kwargs to pass to self.scene_to_dataset()

Returns

PyTorch-compatiable training,
validation, and test datasets.

Return type
Tuple[Dataset, Dataset, Dataset]

random_subset_dataset(*ds: torch.utils.data.Dataset*, *size: Optional[int] = None*, *fraction: Optional[ConstrainedFloatValue] = None*) → torch.utils.data.Subset

Parameters

- **ds** (*torch.utils.data.Dataset*) –

- **size** (*Optional[int]*) –
- **fraction** (*Optional[ConstrainedFloatValue]*) –

Return type
torch.utils.data.Subset

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

scene_to_dataset(*scene:* Scene, *transform: Optional[BasicTransform] = None*) → torch.utils.data.Dataset

Make a dataset from a single scene.

Parameters

- **scene** (Scene) –
- **transform** (*Optional[BasicTransform]*) –

Return type
torch.utils.data.Dataset

update(**args*, ***kwargs*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validator validate_augmentors » augmentors

Parameters
v (*str*) –

Return type
str

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises
ConfigError – if field is invalid

validator validate_plot_options » *all fields*

Parameters
values (*dict*) –

Return type
dict

validator validate_window_opts » window_opts

Parameters

- **v** (*Union[GeoDataWindowConfig, Dict[str, GeoDataWindowConfig]]*) –
- **values** (*dict*) –

Return type
Union[GeoDataWindowConfig, *Dict*[str, GeoDataWindowConfig]]

property num_classes

GeoDataWindowConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model GeoDataWindowConfig

Configure a *GeoDataset*.

See *rastervision.pytorch_learner.dataset.dataset*.


```
{
   "title": "GeoDataWindowConfig",
   "description": "Configure a :class:`.GeoDataset`.\n\nSee :mod:`rastervision.
↪pytorch_learner.dataset.dataset`.",
   "type": "object",
   "properties": {
      "method": {
         "default": "sliding",
         "allOf": [
            {
               "$ref": "#/definitions/GeoDataWindowMethod"
            }
         ]
      },
      "size": {
         "title": "Size",
         "description": "If method = sliding, this is the size of sliding window.␣
↪If method = random, this is the size that all the windows are resized to before␣
↪they are returned. If method = random and neither size_lims nor h_lims and w_lims␣
↪have been specified, then size_lims is set to (size, size + 1).",
         "anyOf": [
            {
               "type": "integer",
               "exclusiveMinimum": 0
            },
```


(continues on next page)

(continued from previous page)

```
            {
               "type": "array",
               "minItems": 2,
               "maxItems": 2,
               "items": [
                  {
                     "type": "integer",
                     "exclusiveMinimum": 0
                  },
                  {
                     "type": "integer",
                     "exclusiveMinimum": 0
                  }
               ]
            }
         ]
      },
      "stride": {
         "title": "Stride",
         "description": "Stride of sliding window. Only used if method = sliding.",
         "anyOf": [
            {
               "type": "integer",
               "exclusiveMinimum": 0
            },
            {
               "type": "array",
               "minItems": 2,
               "maxItems": 2,
               "items": [
                  {
                     "type": "integer",
                     "exclusiveMinimum": 0
                  },
                  {
                     "type": "integer",
                     "exclusiveMinimum": 0
                  }
               ]
            }
         ]
      },
      "padding": {
         "title": "Padding",
         "description": "How many pixels are windows allowed to overflow the edges␣
↪of the raster source.",
         "anyOf": [
            {
               "type": "integer",
               "minimum": 0
            },
            {
```

(continues on next page)

(continued from previous page)


```
                "type": "array",
                "minItems": 2,
                "maxItems": 2,
                "items": [
                   {
                      "type": "integer",
                      "minimum": 0
                   },
                   {
                      "type": "integer",
                      "minimum": 0
                   }
                ]
             }
          ]
       },
       "pad_direction": {
          "title": "Pad Direction",
          "description": "If \"end\", only pad ymax and xmax (bottom and right). If \
↪"start\", only pad ymin and xmin (top and left). If \"both\", pad all sides. Has␣
↪no effect if paddiong is zero. Defaults to \"end\".",
          "default": "end",
          "enum": [
             "both",
             "start",
             "end"
          ],
          "type": "string"
       },
       "size_lims": {
          "title": "Size Lims",
          "description": "[min, max) interval from which window sizes will be␣
↪uniformly randomly sampled. The upper limit is exclusive. To fix the size to a␣
↪constant value, use size_lims = (sz, sz + 1). Only used if method = random.␣
↪Specify either size_lims or h_lims and w_lims, but not both. If neither size_lims␣
↪nor h_lims and w_lims have been specified, then this will be set to (size, size +␣
↪1).",
          "type": "array",
          "minItems": 2,
          "maxItems": 2,
          "items": [
             {
                "type": "integer",
                "exclusiveMinimum": 0
             },
             {
                "type": "integer",
                "exclusiveMinimum": 0
             }
          ]
       },
       "h_lims": {
```


(continues on next page)

(continued from previous page)

```
         "title": "H Lims",
         "description": "[min, max] interval from which window heights will be␣
↪uniformly randomly sampled. Only used if method = random.",
         "type": "array",
         "minItems": 2,
         "maxItems": 2,
         "items": [
            {
               "type": "integer",
               "exclusiveMinimum": 0
            },
            {
               "type": "integer",
               "exclusiveMinimum": 0
            }
         ]
      },
      "w_lims": {
         "title": "W Lims",
         "description": "[min, max] interval from which window widths will be␣
↪uniformly randomly sampled. Only used if method = random.",
         "type": "array",
         "minItems": 2,
         "maxItems": 2,
         "items": [
            {
               "type": "integer",
               "exclusiveMinimum": 0
            },
            {
               "type": "integer",
               "exclusiveMinimum": 0
            }
         ]
      },
      "max_windows": {
         "title": "Max Windows",
         "description": "Max allowed reads from a GeoDataset. Only used if method =␣
↪random.",
         "default": 10000,
         "minimum": 0,
         "type": "integer"
      },
      "max_sample_attempts": {
         "title": "Max Sample Attempts",
         "description": "Max attempts when trying to find a window within the AOI␣
↪of a scene. Only used if method = random and the scene has aoi_polygons specified.
↪",
         "default": 100,
         "exclusiveMinimum": 0,
         "type": "integer"
      },
```

(continues on next page)

(continued from previous page)


```
      "efficient_aoi_sampling": {
         "title": "Efficient Aoi Sampling",
         "description": "If the scene has AOIs, sampling windows at random anywhere␣
↪in the extent and then checking if they fall within any of the AOIs can be very␣
↪inefficient. This flag enables the use of an alternate algorithm that only␣
↪samples window locations inside the AOIs. Only used if method = random and the␣
↪scene has aoi_polygons specified. Defaults to True",
         "default": true,
         "type": "boolean"
      },
      "type_hint": {
         "title": "Type Hint",
         "default": "geo_data_window",
         "enum": [
            "geo_data_window"
         ],
         "type": "string"
      }
   },
   "required": [
      "size"
   ],
   "additionalProperties": false,
   "definitions": {
      "GeoDataWindowMethod": {
         "title": "GeoDataWindowMethod",
         "description": "An enumeration.",
         "enum": [
            "sliding",
            "random"
         ]
      }
   }
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *efficient_aoi_sampling (bool)*
- *h_lims (Optional[Tuple[pydantic.types.PositiveInt, pydantic.types.PositiveInt]])*
- *max_sample_attempts (pydantic.types.PositiveInt)*
- *max_windows (rastervision.pytorch_learner.learner_config.ConstrainedIntValue)*
- *method (rastervision.pytorch_learner.learner_config.GeoDataWindowMethod)*
- *pad_direction (Literal['both', 'start', 'end'])*

- *padding (Optional[Union[rastervision.pytorch_learner.learner_config.ConstrainedIntValue, Tuple[rastervision.pytorch_learner.learner_config.ConstrainedIntValue, rastervision.pytorch_learner.learner_config.ConstrainedIntValue]]])*
- *size (Union[pydantic.types.PositiveInt, Tuple[pydantic.types.PositiveInt, pydantic.types.PositiveInt]])*
- *size_lims (Optional[Tuple[pydantic.types.PositiveInt, pydantic.types.PositiveInt]])*
- *stride (Optional[Union[pydantic.types.PositiveInt, Tuple[pydantic.types.PositiveInt, pydantic.types.PositiveInt]]])*
- *type_hint (Literal['geo_data_window'])*
- *w_lims (Optional[Tuple[pydantic.types.PositiveInt, pydantic.types.PositiveInt]])*

field efficient_aoi_sampling: bool = True

If the scene has AOIs, sampling windows at random anywhere in the extent and then checking if they fall within any of the AOIs can be very inefficient. This flag enables the use of an alternate algorithm that only samples window locations inside the AOIs. Only used if method = random and the scene has aoi_polygons specified. Defaults to True

Validated by

- *validate_options*

field h_lims: Optional[Tuple[PositiveInt, PositiveInt]] = None

[min, max] interval from which window heights will be uniformly randomly sampled. Only used if method = random.

Validated by

- *validate_options*

field max_sample_attempts: PositiveInt = 100

Max attempts when trying to find a window within the AOI of a scene. Only used if method = random and the scene has aoi_polygons specified.

Constraints

- **exclusiveMinimum** = 0

Validated by

- *validate_options*

field max_windows: ConstrainedIntValue = 10000

Max allowed reads from a GeoDataset. Only used if method = random.

Constraints

- **minimum** = 0

Validated by

- *validate_options*

field method: *GeoDataWindowMethod* = GeoDataWindowMethod.sliding

Validated by

- *validate_options*

field pad_direction: Literal['both', 'start', 'end'] = 'end'

If "end", only pad ymax and xmax (bottom and right). If "start", only pad ymin and xmin (top and left). If "both", pad all sides. Has no effect if paddiong is zero. Defaults to "end".

Validated by

- *validate_options*

field padding: Optional[Union[ConstrainedIntValue, Tuple[ConstrainedIntValue, ConstrainedIntValue]]] = None

How many pixels are windows allowed to overflow the edges of the raster source.

Validated by

- *validate_options*

field size: Union[PositiveInt, Tuple[PositiveInt, PositiveInt]] [Required]

If method = sliding, this is the size of sliding window. If method = random, this is the size that all the windows are resized to before they are returned. If method = random and neither size_lims nor h_lims and w_lims have been specified, then size_lims is set to (size, size + 1).

Validated by

- *validate_options*

field size_lims: Optional[Tuple[PositiveInt, PositiveInt]] = None

[min, max) interval from which window sizes will be uniformly randomly sampled. The upper limit is exclusive. To fix the size to a constant value, use size_lims = (sz, sz + 1). Only used if method = random. Specify either size_lims or h_lims and w_lims, but not both. If neither size_lims nor h_lims and w_lims have been specified, then this will be set to (size, size + 1).

Validated by

- *validate_options*

field stride: Optional[Union[PositiveInt, Tuple[PositiveInt, PositiveInt]]] = None

Stride of sliding window. Only used if method = sliding.

Validated by

- *validate_options*

field type_hint: Literal['geo_data_window'] = 'geo_data_window'

Validated by

- *validate_options*

field w_lims: Optional[Tuple[PositiveInt, PositiveInt]] = None

[min, max] interval from which window widths will be uniformly randomly sampled. Only used if method = random.

Validated by

- *validate_options*

build()

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(**args*, ***kwargs*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

validator validate_options » *all fields*

Parameters

values (*dict*) –

Return type

dict

ImageDataConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model ImageDataConfig

Config related to dataset for training and testing.


```
{
   "title": "ImageDataConfig",
   "description": "Config related to dataset for training and testing.",
   "type": "object",
   "properties": {
      "class_names": {
```


(continues on next page)

(continued from previous page)

```
         "title": "Class Names",
         "description": "Names of classes.",
         "default": [],
         "type": "array",
         "items": {
            "type": "string"
         }
      },
      "class_colors": {
         "title": "Class Colors",
         "description": "Colors used to display classes. Can be color 3-tuples in␣
↪list form.",
         "type": "array",
         "items": {
            "anyOf": [
               {
                  "type": "string"
               },
               {
                  "type": "array",
                  "minItems": 3,
                  "maxItems": 3,
                  "items": [
                     {
                        "type": "integer"
                     },
                     {
                        "type": "integer"
                     },
                     {
                        "type": "integer"
                     }
                  ]
               }
            ]
         }
      },
      "img_channels": {
         "title": "Img Channels",
         "description": "The number of channels of the training images.",
         "exclusiveMinimum": 0,
         "type": "integer"
      },
      "img_sz": {
         "title": "Img Sz",
         "description": "Length of a side of each image in pixels. This is the size␣
↪to transform it to during training, not the size in the raw dataset.",
         "default": 256,
         "exclusiveMinimum": 0,
         "type": "integer"
      },
      "train_sz": {
```

(continues on next page)

(continued from previous page)

```
        "title": "Train Sz",
        "description": "If set, the number of training images to use. If fewer␣
↪images exist, then an exception will be raised.",
        "type": "integer"
     },
     "train_sz_rel": {
        "title": "Train Sz Rel",
        "description": "If set, the proportion of training images to use.",
        "type": "number"
     },
     "num_workers": {
        "title": "Num Workers",
        "description": "Number of workers to use when DataLoader makes batches.",
        "default": 4,
        "type": "integer"
     },
     "augmentors": {
        "title": "Augmentors",
        "description": "Names of albumentations augmentors to use for training␣
↪batches. Choices include: ['Blur', 'RandomRotate90', 'HorizontalFlip',
↪'VerticalFlip', 'GaussianBlur', 'GaussNoise', 'RGBShift', 'ToGray'].␣
↪Alternatively, a custom transform can be provided via the aug_transform option.",
        "default": [
           "RandomRotate90",
           "HorizontalFlip",
           "VerticalFlip"
        ],
        "type": "array",
        "items": {
           "type": "string"
        }
     },
     "base_transform": {
        "title": "Base Transform",
        "description": "An Albumentations transform serialized as a dict that will␣
↪be applied to all datasets: training, validation, and test. This transformation␣
↪is in addition to the resizing due to img_sz. This is useful for, for example,␣
↪applying the same normalization to all datasets.",
        "type": "object"
     },
     "aug_transform": {
        "title": "Aug Transform",
        "description": "An Albumentations transform serialized as a dict that will␣
↪be applied as data augmentation to the training dataset. This transform is␣
↪applied before base_transform. If provided, the augmentors option is ignored.",
        "type": "object"
     },
     "plot_options": {
        "title": "Plot Options",
        "description": "Options to control plotting.",
        "default": {
           "transform": {
```

(continues on next page)

(continued from previous page)


```
                "__version__": "1.3.0",
                "transform": {
                   "__class_fullname__": "rastervision.pytorch_learner.utils.utils.
↪MinMaxNormalize",
                   "always_apply": false,
                   "p": 1.0,
                   "min_val": 0.0,
                   "max_val": 1.0,
                   "dtype": 5
                }
             },
             "channel_display_groups": null,
             "type_hint": "plot_options"
          },
          "allOf": [
             {
                "$ref": "#/definitions/PlotOptions"
             }
          ]
       },
       "preview_batch_limit": {
          "title": "Preview Batch Limit",
          "description": "Optional limit on the number of items in the preview plots␣
↪produced during training.",
          "type": "integer"
       },
       "type_hint": {
          "title": "Type Hint",
          "default": "image_data",
          "enum": [
             "image_data"
          ],
          "type": "string"
       },
       "data_format": {
          "title": "Data Format",
          "description": "Name of dataset format.",
          "type": "string"
       },
       "uri": {
          "title": "Uri",
          "description": "One of the following:\n(1) a URI of a directory containing␣
↪\"train\", \"valid\", and (optinally) \"test\" subdirectories;\n(2) a URI of a␣
↪zip file containing (1);\n(3) a list of (2);\n(4) a URI of a directory containing␣
↪zip files containing (1).",
          "anyOf": [
             {
                "type": "string"
             },
             {
                "type": "array",
                "items": {
```


(continues on next page)

(continued from previous page)

```
                    "type": "string"
                }
            }
        ]
    },
    "group_uris": {
        "title": "Group Uris",
        "description": "This can be set instead of uri in order to specify groups␣
↪of chips. Each element in the list is expected to be an object of the same form␣
↪accepted by the uri field. The purpose of separating chips into groups is to be␣
↪able to use the group_train_sz field.",
        "type": "array",
        "items": {
            "anyOf": [
                {
                    "type": "string"
                },
                {
                    "type": "array",
                    "items": {
                        "type": "string"
                    }
                }
            ]
        }
    },
    "group_train_sz": {
        "title": "Group Train Sz",
        "description": "If group_uris is set, this can be used to specify the␣
↪number of chips to use per group. Only applies to training chips. This can either␣
↪be a single value that will be used for all groups or a list of values (one for␣
↪each group).",
        "anyOf": [
            {
                "type": "integer"
            },
            {
                "type": "array",
                "items": {
                    "type": "integer"
                }
            }
        ]
    },
    "group_train_sz_rel": {
        "title": "Group Train Sz Rel",
        "description": "Relative version of group_train_sz. Must be a float in [0,␣
↪1]. If group_uris is set, this can be used to specify the proportion of the total␣
↪chips in each group to use per group. Only applies to training chips. This can␣
↪either be a single value that will be used for all groups or a list of values␣
↪(one for each group).",
        "anyOf": [
```

(continues on next page)

(continued from previous page)


```
            {
               "type": "number",
               "minimum": 0,
               "maximum": 1
            },
            {
               "type": "array",
               "items": {
                  "type": "number",
                  "minimum": 0,
                  "maximum": 1
               }
            }
         ]
      }
   },
   "additionalProperties": false,
   "definitions": {
      "PlotOptions": {
         "title": "PlotOptions",
         "description": "Config related to plotting.",
         "type": "object",
         "properties": {
            "transform": {
               "title": "Transform",
               "description": "An Albumentations transform serialized as a dict␣
↪that will be applied to each image before it is plotted. Mainly useful for␣
↪undoing any data transformation that you do not want included in the plot, such␣
↪as normalization. The default value will shift and scale the image so the values␣
↪range from 0.0 to 1.0 which is the expected range for the plotting function. This␣
↪default is useful for cases where the values after normalization are close to␣
↪zero which makes the plot difficult to see.",
               "default": {
                  "__version__": "1.3.0",
                  "transform": {
                     "__class_fullname__": "rastervision.pytorch_learner.utils.
↪utils.MinMaxNormalize",
                     "always_apply": false,
                     "p": 1.0,
                     "min_val": 0.0,
                     "max_val": 1.0,
                     "dtype": 5
                  }
               },
               "type": "object"
            },
            "channel_display_groups": {
               "title": "Channel Display Groups",
               "description": "Groups of image channels to display together as a␣
↪subplot when plotting the data and predictions. Can be a list or tuple of groups␣
↪(e.g. [(0, 1, 2), (3,)]) or a dict containing title-to-group mappings (e.g. {\
↪"RGB\": [0, 1, 2], \"IR\": [3]}), where each group is a list or tuple of channel␣
```


(continues on next page)

(continued from previous page)

```
↪indices and title is a string that will be used as the title of the subplot for␣
↪that group.",
                "anyOf": [
                    {
                        "type": "object",
                        "additionalProperties": {
                            "type": "array",
                            "items": {
                                "type": "integer",
                                "minimum": 0
                            }
                        }
                    },
                    {
                        "type": "array",
                        "items": {
                            "type": "array",
                            "items": {
                                "type": "integer",
                                "minimum": 0
                            }
                        }
                    }
                ]
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "plot_options",
                "enum": [
                    "plot_options"
                ],
                "type": "string"
            }
        },
        "additionalProperties": false
    }
}
```

Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *aug_transform (Optional[dict])*
- *augmentors (List[str])*
- *base_transform (Optional[dict])*
- *class_colors (Optional[List[Union[str, Tuple[int, int, int]]]])*
- *class_names (List[str])*

- *data_format (Optional[str])*
- *group_train_sz (Optional[Union[int, List[int]]])*
- *group_train_sz_rel (Optional[Union[rastervision.pytorch_learner.learner_config.ConstrainedFloatValue, List[rastervision.pytorch_learner.learner_config.ConstrainedFloatValue]]])*
- *group_uris (Optional[List[Union[str, List[str]]]])*
- *img_channels (Optional[pydantic.types.PositiveInt])*
- *img_sz (pydantic.types.PositiveInt)*
- *num_workers (int)*
- *plot_options (Optional[rastervision.pytorch_learner.learner_config.PlotOptions])*
- *preview_batch_limit (Optional[int])*
- *train_sz (Optional[int])*
- *train_sz_rel (Optional[float])*
- *type_hint (Literal['image_data'])*
- *uri (Optional[Union[str, List[str]]])*

Validators

- *ensure_class_colors* » all fields
- *validate_albumentation_transform* » *aug_transform*
- *validate_albumentation_transform* » *base_transform*
- *validate_augmentors* » *augmentors*
- *validate_group_uris* » all fields
- *validate_plot_options* » all fields

field aug_transform: Optional[dict] = None

An Albumentations transform serialized as a dict that will be applied as data augmentation to the training dataset. This transform is applied before base_transform. If provided, the augmentors option is ignored.

Validated by

- *ensure_class_colors*
- *validate_albumentation_transform*
- *validate_group_uris*
- *validate_plot_options*

field augmentors: List[str] = ['RandomRotate90', 'HorizontalFlip', 'VerticalFlip']

Names of albumentations augmentors to use for training batches. Choices include: [‘Blur’, ‘RandomRotate90’, ‘HorizontalFlip’, ‘VerticalFlip’, ‘GaussianBlur’, ‘GaussNoise’, ‘RGBShift’, ‘ToGray’]. Alternatively, a custom transform can be provided via the aug_transform option.

Validated by

- *ensure_class_colors*
- *validate_augmentors*

- *validate_group_uris*
- *validate_plot_options*

field base_transform: Optional[dict] = None

An Albumentations transform serialized as a dict that will be applied to all datasets: training, validation, and test. This transformation is in addition to the resizing due to img_sz. This is useful for, for example, applying the same normalization to all datasets.

Validated by

- *ensure_class_colors*
- *validate_albumentation_transform*
- *validate_group_uris*
- *validate_plot_options*

field class_colors: Optional[List[Union[str, Tuple[int, int, int]]]] = None

Colors used to display classes. Can be color 3-tuples in list form.

Validated by

- *ensure_class_colors*
- *validate_group_uris*
- *validate_plot_options*

field class_names: List[str] = []

Names of classes.

Validated by

- *ensure_class_colors*
- *validate_group_uris*
- *validate_plot_options*

field data_format: Optional[str] = None

Name of dataset format.

Validated by

- *ensure_class_colors*
- *validate_group_uris*
- *validate_plot_options*

field group_train_sz: Optional[Union[int, List[int]]] = None

If group_uris is set, this can be used to specify the number of chips to use per group. Only applies to training chips. This can either be a single value that will be used for all groups or a list of values (one for each group).

Validated by

- *ensure_class_colors*
- *validate_group_uris*
- *validate_plot_options*

field group_train_sz_rel: Optional[Union[ConstrainedFloatValue, List[ConstrainedFloatValue]]] = None

Relative version of group_train_sz. Must be a float in [0, 1]. If group_uris is set, this can be used to specify the proportion of the total chips in each group to use per group. Only applies to training chips. This can either be a single value that will be used for all groups or a list of values (one for each group).

Validated by

- *ensure_class_colors*
- *validate_group_uris*
- *validate_plot_options*

field group_uris: Optional[List[Union[str, List[str]]]] = None

This can be set instead of uri in order to specify groups of chips. Each element in the list is expected to be an object of the same form accepted by the uri field. The purpose of separating chips into groups is to be able to use the group_train_sz field.

Validated by

- *ensure_class_colors*
- *validate_group_uris*
- *validate_plot_options*

field img_channels: Optional[PositiveInt] = None

The number of channels of the training images.

Constraints

- **exclusiveMinimum** = 0

Validated by

- *ensure_class_colors*
- *validate_group_uris*
- *validate_plot_options*

field img_sz: PositiveInt = 256

Length of a side of each image in pixels. This is the size to transform it to during training, not the size in the raw dataset.

Constraints

- **exclusiveMinimum** = 0

Validated by

- *ensure_class_colors*
- *validate_group_uris*
- *validate_plot_options*

field num_workers: int = 4

Number of workers to use when DataLoader makes batches.

Validated by

- *ensure_class_colors*
- *validate_group_uris*

- *validate_plot_options*

field plot_options: Optional[*PlotOptions*] = PlotOptions(transform={'__version__': '1.3.0', 'transform': {'__class_fullname__': 'rastervision.pytorch_learner.utils.utils.MinMaxNormalize', 'always_apply': False, 'p': 1.0, 'min_val': 0.0, 'max_val': 1.0, 'dtype': 5}}, channel_display_groups=None)

Options to control plotting.

Validated by

- *ensure_class_colors*
- *validate_group_uris*
- *validate_plot_options*

field preview_batch_limit: Optional[int] = None

Optional limit on the number of items in the preview plots produced during training.

Validated by

- *ensure_class_colors*
- *validate_group_uris*
- *validate_plot_options*

field train_sz: Optional[int] = None

If set, the number of training images to use. If fewer images exist, then an exception will be raised.

Validated by

- *ensure_class_colors*
- *validate_group_uris*
- *validate_plot_options*

field train_sz_rel: Optional[float] = None

If set, the proportion of training images to use.

Validated by

- *ensure_class_colors*
- *validate_group_uris*
- *validate_plot_options*

field type_hint: Literal['image_data'] = 'image_data'

Validated by

- *ensure_class_colors*
- *validate_group_uris*
- *validate_plot_options*

field uri: Optional[Union[str, List[str]]] = None

One of the following: (1) a URI of a directory containing "train", "valid", and (optinally) "test" subdirectories; (2) a URI of a zip file containing (1); (3) a list of (2); (4) a URI of a directory containing zip files containing (1).

Validated by

- *ensure_class_colors*
- *validate_group_uris*
- *validate_plot_options*

build(*tmp_dir: str*, *overfit_mode: bool = False*, *test_mode: bool = False*) → Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

Build and return train, val, and test datasets.

Parameters

- **tmp_dir** (*str*) –
- **overfit_mode** (*bool*) –
- **test_mode** (*bool*) –

Return type
Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

dir_to_dataset(*data_dir: str*, *transform: BasicTransform*) → torch.utils.data.Dataset

Parameters

- **data_dir** (*str*) –
- **transform** (*BasicTransform*) –

Return type
torch.utils.data.Dataset

validator ensure_class_colors » *all fields*

Parameters
values (*dict*) –

Return type
dict

get_bbox_params() → Optional[BboxParams]

Returns BboxParams used by albumentations for data augmentation.

Return type
Optional[*BboxParams*]

get_custom_albumentations_transforms() → List[dict]

Returns all custom transforms found in this config.

This should return all serialized albumentations transforms with a 'lambda_transforms_path' field contained in this config or in any of its members no matter how deeply neseted.

The pupose is to make it easier to adjust their paths all at once while saving to or loading from a bundle.

Return type
List[dict]

get_data_dirs(*uri: Union[str, List[str]]*, *unzip_dir: str*) → List[str]

Extract data dirs from uri.

Data dirs are directories containing "train", "valid", and (optinally) "test" subdirectories.

Parameters

- **uri** (*Union[str, List[str]]*) – a URI or a list of URIs of one of the following:

(1) a URI of a directory containing "train", "valid", and (optinally) "test" subdirectories

(2) a URI of a zip file containing (1)

(3) a list of (2)

(4) a URI of a directory containing zip files containing (1)

- **unzip_dir** (*str*) –

Returns
paths to directories that each contain contents of one zip file

Return type
List[str]

get_data_transforms() → Tuple[BasicTransform, BasicTransform]

Get albumentations transform objects for data augmentation.

Returns
a transform that doesn't do any data augmentation 2nd tuple arg: a transform with data augmentation

Return type
1st tuple arg

get_datasets_from_group_uris(*uris: Union[str, List[str]]*, *tmp_dir: str*, *group_train_sz: Optional[int] = None*, *group_train_sz_rel: Optional[float] = None*, *overfit_mode: bool = False*, *test_mode: bool = False*) → Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

Parameters

- **uris** (*Union[str, List[str]]*) –
- **tmp_dir** (*str*) –
- **group_train_sz** (*Optional[int]*) –
- **group_train_sz_rel** (*Optional[float]*) –
- **overfit_mode** (*bool*) –
- **test_mode** (*bool*) –

Return type
Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

get_datasets_from_uri(*uri: Union[str, List[str]]*, *tmp_dir: str*, *overfit_mode: bool = False*, *test_mode: bool = False*) → Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

Get image train, validation, & test datasets from a single zip file.

Parameters

- **uri** (*Union[str, List[str]]*) – Uri of a zip file containing the images.
- **tmp_dir** (*str*) –
- **overfit_mode** (*bool*) –
- **test_mode** (*bool*) –

Returns

Training, validation, and test
dataSets.

Return type
Tuple[Dataset, Dataset, Dataset]

make_datasets(*train_dirs: Iterable[str]*, *val_dirs: Iterable[str]*, *test_dirs: Iterable[str]*, *train_tf: Optional[BasicTransform] = None*, *val_tf: Optional[BasicTransform] = None*, *test_tf: Optional[BasicTransform] = None*) → Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

Make training, validation, and test datasets.

Parameters

- **train_dirs** (*str*) – Directories where training data is located.
- **val_dirs** (*str*) – Directories where validation data is located.
- **test_dirs** (*str*) – Directories where test data is located.
- **train_tf** (*Optional[A.BasicTransform], optional*) – Transform for the training dataset. Defaults to None.
- **val_tf** (*Optional[A.BasicTransform], optional*) – Transform for the validation dataset. Defaults to None.
- **test_tf** (*Optional[A.BasicTransform], optional*) – Transform for the test dataset. Defaults to None.

Returns

PyTorch-compatiable training,
validation, and test datasets.

Return type
Tuple[Dataset, Dataset, Dataset]

random_subset_dataset(*ds: torch.utils.data.Dataset*, *size: Optional[int] = None*, *fraction: Optional[ConstrainedFloatValue] = None*) → torch.utils.data.Subset

Parameters

- **ds** (*torch.utils.data.Dataset*) –
- **size** (*Optional[int]*) –
- **fraction** (*Optional[ConstrainedFloatValue]*) –

Return type
torch.utils.data.Subset

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

unzip_data(*zip_uris: List[str]*, *unzip_dir: str*) → List[str]

Unzip dataset zip files.

Parameters

- **zip_uris** (*List[str]*) – a list of URIs of zip files:
- **unzip_dir** (*str*) – directory where zip files will be extrated to.

Returns
paths to directories that each contain contents of one zip file

Return type
List[str]

update(**args*, ***kwargs*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validator validate_augmentors » augmentors

Parameters
v (*str*) –

Return type
str

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validator validate_group_uris » *all fields*

Parameters
values (*dict*) –

Return type
dict

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises
ConfigError – if field is invalid

validator validate_plot_options » *all fields*

Parameters
values (*dict*) –

Return type
dict

property num_classes

LearnerConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model LearnerConfig

Config for Learner.


```
{
   "title": "LearnerConfig",
   "description": "Config for Learner.",
   "type": "object",
   "properties": {
      "model": {
         "$ref": "#/definitions/ModelConfig"
      },
      "solver": {
         "$ref": "#/definitions/SolverConfig"
      },
      "data": {
         "$ref": "#/definitions/DataConfig"
      },
      "predict_mode": {
         "title": "Predict Mode",
         "description": "If True, skips training, loads model, and does final eval.
↪",
         "default": false,
         "type": "boolean"
      },
      "test_mode": {
         "title": "Test Mode",
         "description": "If True, uses test_num_epochs, test_batch_sz, truncated␣
↪datasets with only a single batch, image_sz that is cut in half, and num_workers␣
↪= 0. This is useful for testing that code runs correctly on CPU without␣
↪multithreading before running full job on GPU.",
         "default": false,
         "type": "boolean"
      },
      "overfit_mode": {
         "title": "Overfit Mode",
         "description": "If True, uses half image size, and instead of doing epoch-
↪based training, optimizes the model using a single batch repeatedly for overfit_
↪num_steps number of steps.",
         "default": false,
         "type": "boolean"
      },
      "eval_train": {
         "title": "Eval Train",
         "description": "If True, runs final evaluation on training set (in␣
↪addition to test set). Useful for debugging.",
         "default": false,
         "type": "boolean"
      },
```


(continues on next page)

(continued from previous page)

```
      "save_model_bundle": {
         "title": "Save Model Bundle",
         "description": "If True, saves a model bundle at the end of training which␣
↪is zip file with model and this LearnerConfig which can be used to make␣
↪predictions on new images at a later time.",
         "default": true,
         "type": "boolean"
      },
      "log_tensorboard": {
         "title": "Log Tensorboard",
         "description": "Save Tensorboard log files at the end of each epoch.",
         "default": true,
         "type": "boolean"
      },
      "run_tensorboard": {
         "title": "Run Tensorboard",
         "description": "run Tensorboard server during training",
         "default": false,
         "type": "boolean"
      },
      "output_uri": {
         "title": "Output Uri",
         "description": "URI of where to save output",
         "type": "string"
      },
      "type_hint": {
         "title": "Type Hint",
         "default": "learner",
         "enum": [
            "learner"
         ],
         "type": "string"
      }
   },
   "required": [
      "solver",
      "data"
   ],
   "additionalProperties": false,
   "definitions": {
      "Backbone": {
         "title": "Backbone",
         "description": "An enumeration.",
         "enum": [
            "alexnet",
            "densenet121",
            "densenet169",
            "densenet201",
            "densenet161",
            "googlenet",
            "inception_v3",
            "mnasnet0_5",
```

(continues on next page)

(continued from previous page)

```
            "mnasnet0_75",
            "mnasnet1_0",
            "mnasnet1_3",
            "mobilenet_v2",
            "resnet18",
            "resnet34",
            "resnet50",
            "resnet101",
            "resnet152",
            "resnext50_32x4d",
            "resnext101_32x8d",
            "wide_resnet50_2",
            "wide_resnet101_2",
            "shufflenet_v2_x0_5",
            "shufflenet_v2_x1_0",
            "shufflenet_v2_x1_5",
            "shufflenet_v2_x2_0",
            "squeezenet1_0",
            "squeezenet1_1",
            "vgg11",
            "vgg11_bn",
            "vgg13",
            "vgg13_bn",
            "vgg16",
            "vgg16_bn",
            "vgg19_bn",
            "vgg19"
         ]
      },
      "ExternalModuleConfig": {
         "title": "ExternalModuleConfig",
         "description": "Config describing an object to be loaded via Torch Hub.",
         "type": "object",
         "properties": {
            "uri": {
               "title": "Uri",
               "description": "Local uri of a zip file, or local uri of a directory,
↪or remote uri of zip file.",
               "minLength": 1,
               "type": "string"
            },
            "github_repo": {
               "title": "Github Repo",
               "description": "<repo-owner>/<repo-name>[:tag]",
               "pattern": ".+/.+",
               "type": "string"
            },
            "name": {
               "title": "Name",
               "description": "Name of the folder in which to extract/copy the␣
↪definition files.",
               "minLength": 1,
```

(continues on next page)

(continued from previous page)

```
                "type": "string"
            },
            "entrypoint": {
                "title": "Entrypoint",
                "description": "Name of a callable present in hubconf.py. See docs␣
↪for torch.hub for details.",
                "minLength": 1,
                "type": "string"
            },
            "entrypoint_args": {
                "title": "Entrypoint Args",
                "description": "Args to pass to the entrypoint. Must be serializable.
↪",
                "default": [],
                "type": "array",
                "items": {}
            },
            "entrypoint_kwargs": {
                "title": "Entrypoint Kwargs",
                "description": "Keyword args to pass to the entrypoint. Must be␣
↪serializable.",
                "default": {},
                "type": "object"
            },
            "force_reload": {
                "title": "Force Reload",
                "description": "Force reload of module definition.",
                "default": false,
                "type": "boolean"
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "external-module",
                "enum": [
                    "external-module"
                ],
                "type": "string"
            }
        },
        "required": [
            "entrypoint"
        ],
        "additionalProperties": false
    },
    "ModelConfig": {
        "title": "ModelConfig",
        "description": "Config related to models.",
        "type": "object",
        "properties": {
            "backbone": {
                "description": "The torchvision.models backbone to use.",
                "default": "resnet18",
```

(continues on next page)

(continued from previous page)


```
                "allOf": [
                    {
                        "$ref": "#/definitions/Backbone"
                    }
                ]
            },
            "pretrained": {
                "title": "Pretrained",
                "description": "If True, use ImageNet weights. If False, use random␣
↪initialization.",
                "default": true,
                "type": "boolean"
            },
            "init_weights": {
                "title": "Init Weights",
                "description": "URI of PyTorch model weights used to initialize␣
↪model. If set, this supercedes the pretrained option.",
                "type": "string"
            },
            "load_strict": {
                "title": "Load Strict",
                "description": "If True, the keys in the state dict referenced by␣
↪init_weights must match exactly. Setting this to False can be useful if you just␣
↪want to load the backbone of a model.",
                "default": true,
                "type": "boolean"
            },
            "external_def": {
                "title": "External Def",
                "description": "If specified, the model will be built from the␣
↪definition from this external source, using Torch Hub.",
                "allOf": [
                    {
                        "$ref": "#/definitions/ExternalModuleConfig"
                    }
                ]
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "model",
                "enum": [
                    "model"
                ],
                "type": "string"
            }
        },
        "additionalProperties": false
    },
    "SolverConfig": {
        "title": "SolverConfig",
        "description": "Config related to solver aka optimizer.",
        "type": "object",
```


(continues on next page)

(continued from previous page)

```
        "properties": {
           "lr": {
              "title": "Lr",
              "description": "Learning rate.",
              "default": 0.0001,
              "exclusiveMinimum": 0,
              "type": "number"
           },
           "num_epochs": {
              "title": "Num Epochs",
              "description": "Number of epochs (ie. sweeps through the whole␣
↪training set).",
              "default": 10,
              "exclusiveMinimum": 0,
              "type": "integer"
           },
           "test_num_epochs": {
              "title": "Test Num Epochs",
              "description": "Number of epochs to use in test mode.",
              "default": 2,
              "exclusiveMinimum": 0,
              "type": "integer"
           },
           "test_batch_sz": {
              "title": "Test Batch Sz",
              "description": "Batch size to use in test mode.",
              "default": 4,
              "exclusiveMinimum": 0,
              "type": "integer"
           },
           "overfit_num_steps": {
              "title": "Overfit Num Steps",
              "description": "Number of optimizer steps to use in overfit mode.",
              "default": 1,
              "exclusiveMinimum": 0,
              "type": "integer"
           },
           "sync_interval": {
              "title": "Sync Interval",
              "description": "The interval in epochs for each sync to the cloud.",
              "default": 1,
              "exclusiveMinimum": 0,
              "type": "integer"
           },
           "batch_sz": {
              "title": "Batch Sz",
              "description": "Batch size.",
              "default": 32,
              "exclusiveMinimum": 0,
              "type": "integer"
           },
           "one_cycle": {
```

(continues on next page)

(continued from previous page)


```
                "title": "One Cycle",
                "description": "If True, use triangular LR scheduler with a single␣
↪cycle across all epochs with start and end LR being lr/10 and the peak being lr.",
                "default": true,
                "type": "boolean"
            },
            "multi_stage": {
                "title": "Multi Stage",
                "description": "List of epoch indices at which to divide LR by 10.",
                "default": [],
                "type": "array",
                "items": {}
            },
            "class_loss_weights": {
                "title": "Class Loss Weights",
                "description": "Class weights for weighted loss.",
                "type": "array",
                "items": {
                    "type": "number"
                }
            },
            "ignore_class_index": {
                "title": "Ignore Class Index",
                "description": "If specified, this index is ignored when computing␣
↪the loss. See pytorch documentation for nn.CrossEntropyLoss for more details.␣
↪This can also be negative, in which case it is treated as a negative slice index␣
↪i.e. -1 = last index, -2 = second-last index, and so on.",
                "type": "integer"
            },
            "external_loss_def": {
                "title": "External Loss Def",
                "description": "If specified, the loss will be built from the␣
↪definition from this external source, using Torch Hub.",
                "allOf": [
                    {
                        "$ref": "#/definitions/ExternalModuleConfig"
                    }
                ]
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "solver",
                "enum": [
                    "solver"
                ],
                "type": "string"
            }
        },
        "additionalProperties": false
    },
    "PlotOptions": {
        "title": "PlotOptions",
```


(continues on next page)

(continued from previous page)


```
        "description": "Config related to plotting.",
        "type": "object",
        "properties": {
           "transform": {
              "title": "Transform",
              "description": "An Albumentations transform serialized as a dict␣
↪that will be applied to each image before it is plotted. Mainly useful for␣
↪undoing any data transformation that you do not want included in the plot, such␣
↪as normalization. The default value will shift and scale the image so the values␣
↪range from 0.0 to 1.0 which is the expected range for the plotting function. This␣
↪default is useful for cases where the values after normalization are close to␣
↪zero which makes the plot difficult to see.",
              "default": {
                 "__version__": "1.3.0",
                 "transform": {
                    "__class_fullname__": "rastervision.pytorch_learner.utils.
↪utils.MinMaxNormalize",
                    "always_apply": false,
                    "p": 1.0,
                    "min_val": 0.0,
                    "max_val": 1.0,
                    "dtype": 5
                 }
              },
              "type": "object"
           },
           "channel_display_groups": {
              "title": "Channel Display Groups",
              "description": "Groups of image channels to display together as a␣
↪subplot when plotting the data and predictions. Can be a list or tuple of groups␣
↪(e.g. [(0, 1, 2), (3,)]) or a dict containing title-to-group mappings (e.g. {\
↪"RGB\": [0, 1, 2], \"IR\": [3]}), where each group is a list or tuple of channel␣
↪indices and title is a string that will be used as the title of the subplot for␣
↪that group.",
              "anyOf": [
                 {
                    "type": "object",
                    "additionalProperties": {
                       "type": "array",
                       "items": {
                          "type": "integer",
                          "minimum": 0
                       }
                    }
                 },
                 {
                    "type": "array",
                    "items": {
                       "type": "array",
                       "items": {
                          "type": "integer",
                          "minimum": 0
```


(continues on next page)

(continued from previous page)

```
                        }
                     }
                  }
               ]
            },
            "type_hint": {
               "title": "Type Hint",
               "default": "plot_options",
               "enum": [
                  "plot_options"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      },
      "DataConfig": {
         "title": "DataConfig",
         "description": "Config related to dataset for training and testing.",
         "type": "object",
         "properties": {
            "class_names": {
               "title": "Class Names",
               "description": "Names of classes.",
               "default": [],
               "type": "array",
               "items": {
                  "type": "string"
               }
            },
            "class_colors": {
               "title": "Class Colors",
               "description": "Colors used to display classes. Can be color 3-
↪tuples in list form.",
               "type": "array",
               "items": {
                  "anyOf": [
                     {
                        "type": "string"
                     },
                     {
                        "type": "array",
                        "minItems": 3,
                        "maxItems": 3,
                        "items": [
                           {
                              "type": "integer"
                           },
                           {
                              "type": "integer"
                           },
                           {
```

(continues on next page)

(continued from previous page)

```
                                "type": "integer"
                            }
                        ]
                    }
                ]
            }
        },
        "img_channels": {
            "title": "Img Channels",
            "description": "The number of channels of the training images.",
            "exclusiveMinimum": 0,
            "type": "integer"
        },
        "img_sz": {
            "title": "Img Sz",
            "description": "Length of a side of each image in pixels. This is␣
↪the size to transform it to during training, not the size in the raw dataset.",
            "default": 256,
            "exclusiveMinimum": 0,
            "type": "integer"
        },
        "train_sz": {
            "title": "Train Sz",
            "description": "If set, the number of training images to use. If␣
↪fewer images exist, then an exception will be raised.",
            "type": "integer"
        },
        "train_sz_rel": {
            "title": "Train Sz Rel",
            "description": "If set, the proportion of training images to use.",
            "type": "number"
        },
        "num_workers": {
            "title": "Num Workers",
            "description": "Number of workers to use when DataLoader makes␣
↪batches.",
            "default": 4,
            "type": "integer"
        },
        "augmentors": {
            "title": "Augmentors",
            "description": "Names of albumentations augmentors to use for␣
↪training batches. Choices include: ['Blur', 'RandomRotate90', 'HorizontalFlip',
↪'VerticalFlip', 'GaussianBlur', 'GaussNoise', 'RGBShift', 'ToGray'].␣
↪Alternatively, a custom transform can be provided via the aug_transform option.",
            "default": [
                "RandomRotate90",
                "HorizontalFlip",
                "VerticalFlip"
            ],
            "type": "array",
            "items": {
```

(continues on next page)

(continued from previous page)


```
                "type": "string"
            }
        },
        "base_transform": {
            "title": "Base Transform",
            "description": "An Albumentations transform serialized as a dict␣
↪that will be applied to all datasets: training, validation, and test. This␣
↪transformation is in addition to the resizing due to img_sz. This is useful for,␣
↪for example, applying the same normalization to all datasets.",
            "type": "object"
        },
        "aug_transform": {
            "title": "Aug Transform",
            "description": "An Albumentations transform serialized as a dict␣
↪that will be applied as data augmentation to the training dataset. This transform␣
↪is applied before base_transform. If provided, the augmentors option is ignored.",
            "type": "object"
        },
        "plot_options": {
            "title": "Plot Options",
            "description": "Options to control plotting.",
            "default": {
                "transform": {
                    "__version__": "1.3.0",
                    "transform": {
                        "__class_fullname__": "rastervision.pytorch_learner.utils.
↪utils.MinMaxNormalize",
                        "always_apply": false,
                        "p": 1.0,
                        "min_val": 0.0,
                        "max_val": 1.0,
                        "dtype": 5
                    }
                },
                "channel_display_groups": null,
                "type_hint": "plot_options"
            },
            "allOf": [
                {
                    "$ref": "#/definitions/PlotOptions"
                }
            ]
        },
        "preview_batch_limit": {
            "title": "Preview Batch Limit",
            "description": "Optional limit on the number of items in the preview␣
↪plots produced during training.",
            "type": "integer"
        },
        "type_hint": {
            "title": "Type Hint",
            "default": "data",
```


(continues on next page)

(continued from previous page)

```
                "enum": [
                    "data"
                ],
                "type": "string"
            }
        },
        "additionalProperties": false
    }
}
}
```

Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *data (rastervision.pytorch_learner.learner_config.DataConfig)*
- *eval_train (bool)*
- *log_tensorboard (bool)*
- *model (Optional[rastervision.pytorch_learner.learner_config.ModelConfig])*
- *output_uri (Optional[str])*
- *overfit_mode (bool)*
- *predict_mode (bool)*
- *run_tensorboard (bool)*
- *save_model_bundle (bool)*
- *solver (rastervision.pytorch_learner.learner_config.SolverConfig)*
- *test_mode (bool)*
- *type_hint (Literal['learner'])*

Validators

- *update_for_mode* » `all fields`
- *validate_class_loss_weights* » `all fields`
- *validate_run_tensorboard* » *run_tensorboard*

field data: *DataConfig* [Required]

Validated by

- *update_for_mode*
- *validate_class_loss_weights*

field eval_train: bool = False

If True, runs final evaluation on training set (in addition to test set). Useful for debugging.

Validated by

- *update_for_mode*
- *validate_class_loss_weights*

field log_tensorboard: bool = True

Save Tensorboard log files at the end of each epoch.

Validated by

- *update_for_mode*
- *validate_class_loss_weights*

field model: Optional[*ModelConfig*] = None

Validated by

- *update_for_mode*
- *validate_class_loss_weights*

field output_uri: Optional[str] = None

URI of where to save output

Validated by

- *update_for_mode*
- *validate_class_loss_weights*

field overfit_mode: bool = False

If True, uses half image size, and instead of doing epoch-based training, optimizes the model using a single batch repeatedly for overfit_num_steps number of steps.

Validated by

- *update_for_mode*
- *validate_class_loss_weights*

field predict_mode: bool = False

If True, skips training, loads model, and does final eval.

Validated by

- *update_for_mode*
- *validate_class_loss_weights*

field run_tensorboard: bool = False

run Tensorboard server during training

Validated by

- *update_for_mode*
- *validate_class_loss_weights*
- *validate_run_tensorboard*

field save_model_bundle: bool = True

If True, saves a model bundle at the end of training which is zip file with model and this LearnerConfig which can be used to make predictions on new images at a later time.

Validated by

- *update_for_mode*

- *validate_class_loss_weights*

field solver: *SolverConfig* [Required]

Validated by

- *update_for_mode*
- *validate_class_loss_weights*

field test_mode: bool = False

If True, uses test_num_epochs, test_batch_sz, truncated datasets with only a single batch, image_sz that is cut in half, and num_workers = 0. This is useful for testing that code runs correctly on CPU without multithreading before running full job on GPU.

Validated by

- *update_for_mode*
- *validate_class_loss_weights*

field type_hint: Literal['learner'] = 'learner'

Validated by

- *update_for_mode*
- *validate_class_loss_weights*

build(*tmp_dir: Optional[str] = None*, *model_weights_path: Optional[str] = None*, *model_def_path: Optional[str] = None*, *loss_def_path: Optional[str] = None*, *training=True*) → *Learner*

Returns a Learner instantiated using this Config.

Parameters

- **tmp_dir** (*str*) – Root of temp dirs.
- **model_weights_path** (*str, optional*) – A local path to model weights. Defaults to None.
- **model_def_path** (*str, optional*) – A local path to a directory with a hubconf.py. If provided, the model definition is imported from here. Defaults to None.
- **loss_def_path** (*str, optional*) – A local path to a directory with a hubconf.py. If provided, the loss function definition is imported from here. Defaults to None.
- **training** (*bool, optional*) – Whether the model is to be used for training or prediction. If False, the model is put in eval mode and the loss function, optimizer, etc. are not initialized. Defaults to True.

Return type
Learner

get_model_bundle_uri() → str

Returns the URI of where the model bundle is stored.

Return type
str

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(**args*, ***kwargs*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validator update_for_mode » *all fields*

Parameters
values (*dict*) –

Return type
dict

validator validate_class_loss_weights » *all fields*

Parameters
values (*dict*) –

Return type
dict

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises
ConfigError – if field is invalid

validator validate_run_tensorboard » run_tensorboard

Parameters

- **v** (*bool*) –
- **values** (*dict*) –

Return type
bool

ModelConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model ModelConfig

Config related to models.


```
{
   "title": "ModelConfig",
   "description": "Config related to models.",
   "type": "object",
   "properties": {
      "backbone": {
         "description": "The torchvision.models backbone to use.",
         "default": "resnet18",
         "allOf": [
            {
               "$ref": "#/definitions/Backbone"
            }
         ]
      },
      "pretrained": {
         "title": "Pretrained",
         "description": "If True, use ImageNet weights. If False, use random␣
↪initialization.",
         "default": true,
         "type": "boolean"
      },
      "init_weights": {
         "title": "Init Weights",
         "description": "URI of PyTorch model weights used to initialize model. If␣
↪set, this supercedes the pretrained option.",
         "type": "string"
      },
      "load_strict": {
         "title": "Load Strict",
         "description": "If True, the keys in the state dict referenced by init_
↪weights must match exactly. Setting this to False can be useful if you just want␣
↪to load the backbone of a model.",
         "default": true,
         "type": "boolean"
      },
      "external_def": {
         "title": "External Def",
         "description": "If specified, the model will be built from the definition␣
↪from this external source, using Torch Hub.",
         "allOf": [
            {
               "$ref": "#/definitions/ExternalModuleConfig"
            }
         ]
      },
```


(continues on next page)

(continued from previous page)


```
      "type_hint": {
         "title": "Type Hint",
         "default": "model",
         "enum": [
            "model"
         ],
         "type": "string"
      }
   },
   "additionalProperties": false,
   "definitions": {
      "Backbone": {
         "title": "Backbone",
         "description": "An enumeration.",
         "enum": [
            "alexnet",
            "densenet121",
            "densenet169",
            "densenet201",
            "densenet161",
            "googlenet",
            "inception_v3",
            "mnasnet0_5",
            "mnasnet0_75",
            "mnasnet1_0",
            "mnasnet1_3",
            "mobilenet_v2",
            "resnet18",
            "resnet34",
            "resnet50",
            "resnet101",
            "resnet152",
            "resnext50_32x4d",
            "resnext101_32x8d",
            "wide_resnet50_2",
            "wide_resnet101_2",
            "shufflenet_v2_x0_5",
            "shufflenet_v2_x1_0",
            "shufflenet_v2_x1_5",
            "shufflenet_v2_x2_0",
            "squeezenet1_0",
            "squeezenet1_1",
            "vgg11",
            "vgg11_bn",
            "vgg13",
            "vgg13_bn",
            "vgg16",
            "vgg16_bn",
            "vgg19_bn",
            "vgg19"
         ]
      },
```


(continues on next page)

(continued from previous page)

```
    "ExternalModuleConfig": {
       "title": "ExternalModuleConfig",
       "description": "Config describing an object to be loaded via Torch Hub.",
       "type": "object",
       "properties": {
          "uri": {
             "title": "Uri",
             "description": "Local uri of a zip file, or local uri of a directory,
↪or remote uri of zip file.",
             "minLength": 1,
             "type": "string"
          },
          "github_repo": {
             "title": "Github Repo",
             "description": "<repo-owner>/<repo-name>[:tag]",
             "pattern": ".+/.+",
             "type": "string"
          },
          "name": {
             "title": "Name",
             "description": "Name of the folder in which to extract/copy the␣
↪definition files.",
             "minLength": 1,
             "type": "string"
          },
          "entrypoint": {
             "title": "Entrypoint",
             "description": "Name of a callable present in hubconf.py. See docs␣
↪for torch.hub for details.",
             "minLength": 1,
             "type": "string"
          },
          "entrypoint_args": {
             "title": "Entrypoint Args",
             "description": "Args to pass to the entrypoint. Must be serializable.
↪",
             "default": [],
             "type": "array",
             "items": {}
          },
          "entrypoint_kwargs": {
             "title": "Entrypoint Kwargs",
             "description": "Keyword args to pass to the entrypoint. Must be␣
↪serializable.",
             "default": {},
             "type": "object"
          },
          "force_reload": {
             "title": "Force Reload",
             "description": "Force reload of module definition.",
             "default": false,
             "type": "boolean"
```

(continues on next page)

(continued from previous page)


```
            },
            "type_hint": {
               "title": "Type Hint",
               "default": "external-module",
               "enum": [
                  "external-module"
               ],
               "type": "string"
            }
         },
         "required": [
            "entrypoint"
         ],
         "additionalProperties": false
      }
   }
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *backbone (rastervision.pytorch_learner.learner_config.Backbone)*
- *external_def (Optional[rastervision.pytorch_learner.learner_config.ExternalModuleConfig])*
- *init_weights (Optional[str])*
- *load_strict (bool)*
- *pretrained (bool)*
- *type_hint (Literal['model'])*

field backbone: *Backbone* = Backbone.resnet18

The torchvision.models backbone to use.

field external_def: Optional[*ExternalModuleConfig*] = None

If specified, the model will be built from the definition from this external source, using Torch Hub.

field init_weights: Optional[str] = None

URI of PyTorch model weights used to initialize model. If set, this supercedes the pretrained option.

field load_strict: bool = True

If True, the keys in the state dict referenced by init_weights must match exactly. Setting this to False can be useful if you just want to load the backbone of a model.

field pretrained: bool = True

If True, use ImageNet weights. If False, use random initialization.

field type_hint: Literal['model'] = 'model'

build(*num_classes: int*, *in_channels: int*, *save_dir: Optional[str] = None*, *hubconf_dir: Optional[str] = None*, ***kwargs*) → torch.nn.Module

Build and return a model based on the config.

Parameters

- **num_classes** (*int*) – Number of classes.
- **in_channels** (*int, optional*) – Number of channels in the images that will be fed into the model. Defaults to 3.
- **save_dir** (*Optional[str], optional*) – Used for building external_def if specified. Defaults to None.
- **hubconf_dir** (*Optional[str], optional*) – Used for building external_def if specified. Defaults to None.

Returns
a PyTorch nn.Module.

Return type
nn.Module

build_default_model(*num_classes: int*, *in_channels: int*, ***kwargs*) → torch.nn.Module

Build and return the default model.

Parameters

- **num_classes** (*int*) – Number of classes.
- **in_channels** (*int, optional*) – Number of channels in the images that will be fed into the model. Defaults to 3.

Returns
a PyTorch nn.Module.

Return type
nn.Module

build_external_model(*save_dir: str*, *hubconf_dir: Optional[str] = None*) → torch.nn.Module

Build and return an external model.

Parameters

- **save_dir** (*str*) – The module def will be saved here.
- **hubconf_dir** (*Optional[str], optional*) – Path to existing definition. Defaults to None.

Returns
a PyTorch nn.Module.

Return type
nn.Module

get_backbone_str()

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(**args*, ***kwargs*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

PlotOptions

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model PlotOptions

Config related to plotting.


```
{
   "title": "PlotOptions",
   "description": "Config related to plotting.",
   "type": "object",
   "properties": {
      "transform": {
         "title": "Transform",
         "description": "An Albumentations transform serialized as a dict that will␣
↪be applied to each image before it is plotted. Mainly useful for undoing any data␣
↪transformation that you do not want included in the plot, such as normalization.␣
↪The default value will shift and scale the image so the values range from 0.0 to␣
↪1.0 which is the expected range for the plotting function. This default is useful␣
↪for cases where the values after normalization are close to zero which makes the␣
↪plot difficult to see.",
         "default": {
            "__version__": "1.3.0",
            "transform": {
```


(continues on next page)

(continued from previous page)

```
                "__class_fullname__": "rastervision.pytorch_learner.utils.utils.
↪MinMaxNormalize",
                "always_apply": false,
                "p": 1.0,
                "min_val": 0.0,
                "max_val": 1.0,
                "dtype": 5
             }
          },
          "type": "object"
       },
       "channel_display_groups": {
          "title": "Channel Display Groups",
          "description": "Groups of image channels to display together as a subplot␣
↪when plotting the data and predictions. Can be a list or tuple of groups (e.g.␣
↪[(0, 1, 2), (3,)]) or a dict containing title-to-group mappings (e.g. {\"RGB\":␣
↪[0, 1, 2], \"IR\": [3]}), where each group is a list or tuple of channel indices␣
↪and title is a string that will be used as the title of the subplot for that␣
↪group.",
          "anyOf": [
             {
                "type": "object",
                "additionalProperties": {
                   "type": "array",
                   "items": {
                      "type": "integer",
                      "minimum": 0
                   }
                }
             },
             {
                "type": "array",
                "items": {
                   "type": "array",
                   "items": {
                      "type": "integer",
                      "minimum": 0
                   }
                }
             }
          ]
       },
       "type_hint": {
          "title": "Type Hint",
          "default": "plot_options",
          "enum": [
             "plot_options"
          ],
          "type": "string"
       }
    },
    "additionalProperties": false
```

(continues on next page)

(continued from previous page)

```
}
```

Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *channel_display_groups (Optional[Union[Dict[str, Sequence[rastervision.pytorch_learner.learner_config.ConstrainedIntValue]], Sequence[Sequence[rastervision.pytorch_learner.learner_config.ConstrainedIntValue]]]])*
- *transform (Optional[dict])*
- *type_hint (Literal['plot_options'])*

Validators

- *validate_albumentation_transform* » *transform*
- *validate_channel_display_groups* » *channel_display_groups*

field channel_display_groups: Optional[Union[Dict[str, Sequence[ConstrainedIntValue]], Sequence[Sequence[ConstrainedIntValue]]]] = None

Groups of image channels to display together as a subplot when plotting the data and predictions. Can be a list or tuple of groups (e.g. [(0, 1, 2), (3,)]) or a dict containing title-to-group mappings (e.g. {"RGB": [0, 1, 2], "IR": [3]}), where each group is a list or tuple of channel indices and title is a string that will be used as the title of the subplot for that group.

Validated by

- *validate_channel_display_groups*

field transform: Optional[dict] = {'__version__': '1.3.0', 'transform': {'__class_fullname__': 'rastervision.pytorch_learner.utils.utils.MinMaxNormalize', 'always_apply': False, 'dtype': 5, 'max_val': 1.0, 'min_val': 0.0, 'p': 1.0}}

An Albumentations transform serialized as a dict that will be applied to each image before it is plotted. Mainly useful for undoing any data transformation that you do not want included in the plot, such as normalization. The default value will shift and scale the image so the values range from 0.0 to 1.0 which is the expected range for the plotting function. This default is useful for cases where the values after normalization are close to zero which makes the plot difficult to see.

Validated by

- *validate_albumentation_transform*

field type_hint: Literal['plot_options'] = 'plot_options'

build()

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(***kwargs*) → None

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

Return type
None

validator validate_channel_display_groups » channel_display_groups

Parameters
v (*Optional[Union[Dict[str, Sequence[ConstrainedIntValue]], Sequence[Sequence[ConstrainedIntValue]]]]*) –

Return type
Optional[*Dict*[str, *List*[*ConstrainedIntValue*]]]

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises
ConfigError – if field is invalid

SolverConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model SolverConfig

Config related to solver aka optimizer.


```
{
   "title": "SolverConfig",
   "description": "Config related to solver aka optimizer.",
```


(continues on next page)

(continued from previous page)


```
   "type": "object",
   "properties": {
      "lr": {
         "title": "Lr",
         "description": "Learning rate.",
         "default": 0.0001,
         "exclusiveMinimum": 0,
         "type": "number"
      },
      "num_epochs": {
         "title": "Num Epochs",
         "description": "Number of epochs (ie. sweeps through the whole training␣
↪set).",
         "default": 10,
         "exclusiveMinimum": 0,
         "type": "integer"
      },
      "test_num_epochs": {
         "title": "Test Num Epochs",
         "description": "Number of epochs to use in test mode.",
         "default": 2,
         "exclusiveMinimum": 0,
         "type": "integer"
      },
      "test_batch_sz": {
         "title": "Test Batch Sz",
         "description": "Batch size to use in test mode.",
         "default": 4,
         "exclusiveMinimum": 0,
         "type": "integer"
      },
      "overfit_num_steps": {
         "title": "Overfit Num Steps",
         "description": "Number of optimizer steps to use in overfit mode.",
         "default": 1,
         "exclusiveMinimum": 0,
         "type": "integer"
      },
      "sync_interval": {
         "title": "Sync Interval",
         "description": "The interval in epochs for each sync to the cloud.",
         "default": 1,
         "exclusiveMinimum": 0,
         "type": "integer"
      },
      "batch_sz": {
         "title": "Batch Sz",
         "description": "Batch size.",
         "default": 32,
         "exclusiveMinimum": 0,
         "type": "integer"
      },
```


(continues on next page)

(continued from previous page)

```
      "one_cycle": {
         "title": "One Cycle",
         "description": "If True, use triangular LR scheduler with a single cycle␣
↪across all epochs with start and end LR being lr/10 and the peak being lr.",
         "default": true,
         "type": "boolean"
      },
      "multi_stage": {
         "title": "Multi Stage",
         "description": "List of epoch indices at which to divide LR by 10.",
         "default": [],
         "type": "array",
         "items": {}
      },
      "class_loss_weights": {
         "title": "Class Loss Weights",
         "description": "Class weights for weighted loss.",
         "type": "array",
         "items": {
            "type": "number"
         }
      },
      "ignore_class_index": {
         "title": "Ignore Class Index",
         "description": "If specified, this index is ignored when computing the␣
↪loss. See pytorch documentation for nn.CrossEntropyLoss for more details. This␣
↪can also be negative, in which case it is treated as a negative slice index i.e. -
↪1 = last index, -2 = second-last index, and so on.",
         "type": "integer"
      },
      "external_loss_def": {
         "title": "External Loss Def",
         "description": "If specified, the loss will be built from the definition␣
↪from this external source, using Torch Hub.",
         "allOf": [
            {
               "$ref": "#/definitions/ExternalModuleConfig"
            }
         ]
      },
      "type_hint": {
         "title": "Type Hint",
         "default": "solver",
         "enum": [
            "solver"
         ],
         "type": "string"
      }
   },
   "additionalProperties": false,
   "definitions": {
      "ExternalModuleConfig": {
```

(continues on next page)

(continued from previous page)


```
        "title": "ExternalModuleConfig",
        "description": "Config describing an object to be loaded via Torch Hub.",
        "type": "object",
        "properties": {
           "uri": {
              "title": "Uri",
              "description": "Local uri of a zip file, or local uri of a directory,
↪or remote uri of zip file.",
              "minLength": 1,
              "type": "string"
           },
           "github_repo": {
              "title": "Github Repo",
              "description": "<repo-owner>/<repo-name>[:tag]",
              "pattern": ".+/.+",
              "type": "string"
           },
           "name": {
              "title": "Name",
              "description": "Name of the folder in which to extract/copy the␣
↪definition files.",
              "minLength": 1,
              "type": "string"
           },
           "entrypoint": {
              "title": "Entrypoint",
              "description": "Name of a callable present in hubconf.py. See docs␣
↪for torch.hub for details.",
              "minLength": 1,
              "type": "string"
           },
           "entrypoint_args": {
              "title": "Entrypoint Args",
              "description": "Args to pass to the entrypoint. Must be serializable.
↪",
              "default": [],
              "type": "array",
              "items": {}
           },
           "entrypoint_kwargs": {
              "title": "Entrypoint Kwargs",
              "description": "Keyword args to pass to the entrypoint. Must be␣
↪serializable.",
              "default": {},
              "type": "object"
           },
           "force_reload": {
              "title": "Force Reload",
              "description": "Force reload of module definition.",
              "default": false,
              "type": "boolean"
           },
```


(continues on next page)

(continued from previous page)

```
            "type_hint": {
               "title": "Type Hint",
               "default": "external-module",
               "enum": [
                  "external-module"
               ],
               "type": "string"
            }
         },
         "required": [
            "entrypoint"
         ],
         "additionalProperties": false
      }
   }
}
```

Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *batch_sz (pydantic.types.PositiveInt)*
- *class_loss_weights (Optional[Sequence[float]])*
- *external_loss_def (Optional[rastervision.pytorch_learner.learner_config.ExternalModuleConfig])*
- *ignore_class_index (Optional[int])*
- *lr (pydantic.types.PositiveFloat)*
- *multi_stage (List)*
- *num_epochs (pydantic.types.PositiveInt)*
- *one_cycle (bool)*
- *overfit_num_steps (pydantic.types.PositiveInt)*
- *sync_interval (pydantic.types.PositiveInt)*
- *test_batch_sz (pydantic.types.PositiveInt)*
- *test_num_epochs (pydantic.types.PositiveInt)*
- *type_hint (Literal['solver'])*

field batch_sz: PositiveInt = 32

Batch size.

Constraints

- **exclusiveMinimum** = 0

Validated by

- *check_no_loss_opts_if_external*

field class_loss_weights: Optional[Sequence[float]] = None

Class weights for weighted loss.

Validated by

- *check_no_loss_opts_if_external*

field external_loss_def: Optional[*ExternalModuleConfig*] = None

If specified, the loss will be built from the definition from this external source, using Torch Hub.

Validated by

- *check_no_loss_opts_if_external*

field ignore_class_index: Optional[int] = None

If specified, this index is ignored when computing the loss. See pytorch documentation for nn.CrossEntropyLoss for more details. This can also be negative, in which case it is treated as a negative slice index i.e. -1 = last index, -2 = second-last index, and so on.

Validated by

- *check_no_loss_opts_if_external*

field lr: PositiveFloat = 0.0001

Learning rate.

Constraints

- **exclusiveMinimum** = 0

Validated by

- *check_no_loss_opts_if_external*

field multi_stage: List = []

List of epoch indices at which to divide LR by 10.

Validated by

- *check_no_loss_opts_if_external*

field num_epochs: PositiveInt = 10

Number of epochs (ie. sweeps through the whole training set).

Constraints

- **exclusiveMinimum** = 0

Validated by

- *check_no_loss_opts_if_external*

field one_cycle: bool = True

If True, use triangular LR scheduler with a single cycle across all epochs with start and end LR being lr/10 and the peak being lr.

Validated by

- *check_no_loss_opts_if_external*

field overfit_num_steps: PositiveInt = 1

Number of optimizer steps to use in overfit mode.

Constraints

- **exclusiveMinimum** = 0

Validated by

- *check_no_loss_opts_if_external*

`field sync_interval: PositiveInt = 1`

The interval in epochs for each sync to the cloud.

Constraints

- **exclusiveMinimum** = 0

Validated by

- *check_no_loss_opts_if_external*

`field test_batch_sz: PositiveInt = 4`

Batch size to use in test mode.

Constraints

- **exclusiveMinimum** = 0

Validated by

- *check_no_loss_opts_if_external*

`field test_num_epochs: PositiveInt = 2`

Number of epochs to use in test mode.

Constraints

- **exclusiveMinimum** = 0

Validated by

- *check_no_loss_opts_if_external*

`field type_hint: Literal['solver'] = 'solver'`

Validated by

- *check_no_loss_opts_if_external*

`build()`

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

`build_epoch_scheduler(`*optimizer: torch.optim.Optimizer*, *last_epoch: int = -1*, ***kwargs*`)` → Optional[torch.optim.lr_scheduler._LRScheduler]

Returns an LR scheduler tha changes the LR each epoch.

This is used to divide the LR by 10 at certain epochs.

Parameters

- **optimizer** (*torch.optim.Optimizer*) –
- **last_epoch** (*int*) –

Return type

Optional[torch.optim.lr_scheduler._LRScheduler]

build_loss(*num_classes: int*, *save_dir: Optional[str] = None*, *hubconf_dir: Optional[str] = None*) → Callable

Build and return a loss function based on the config.

Parameters

- **num_classes** (*int*) – Number of classes.
- **save_dir** (*Optional[str], optional*) – Used for building external_loss_def if specified. Defaults to None.
- **hubconf_dir** (*Optional[str], optional*) – Used for building external_loss_def if specified. Defaults to None.

Returns
Loss function.

Return type
Callable

build_optimizer(*model: torch.nn.Module*, ***kwargs*) → torch.optim.Optimizer

Parameters
model (*torch.nn.Module*) –

Return type
torch.optim.Optimizer

build_step_scheduler(*optimizer: torch.optim.Optimizer*, *train_ds_sz: int*, *last_epoch: int = -1*, ***kwargs*) → Optional[torch.optim.lr_scheduler._LRScheduler]

Returns an LR scheduler that changes the LR each step.

This is used to implement the "one cycle" schedule popularized by fastai.

Parameters

- **optimizer** (*torch.optim.Optimizer*) –
- **train_ds_sz** (*int*) –
- **last_epoch** (*int*) –

Return type
Optional[torch.optim.lr_scheduler._LRScheduler]

validator check_no_loss_opts_if_external » *all fields*

Parameters
values (*dict*) –

Return type
dict

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(**args*, ***kwargs*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

Functions

ensure_class_colors(class_names[, class_colors])	Ensure that class_colors is valid.
get_default_channel_display_groups(...)	Returns the default channel_display_groups object.
validate_channel_display_groups(groups)	Validate channel display groups object.

ensure_class_colors

ensure_class_colors(*class_names: List[str]*, *class_colors: Optional[List[Union[str, Tuple[int, int, int]]]] = None*)

Ensure that class_colors is valid.

If class_names is empty, fill with random colors.

Parameters

- **class_names** (*List[str]*) – see DataConfig.class_names
- **class_colors** (*Optional[List[Union[str, Tuple[int, int, int]]]]*) – see DataConfig.class_colors

get_default_channel_display_groups

get_default_channel_display_groups(*nb_img_channels: int*) → Dict[str, Sequence[ConstrainedIntValue]]

Returns the default channel_display_groups object.

See PlotOptions.channel_display_groups. Displays at most the first 3 channels as RGB.

Parameters

nb_img_channels (*int*) – number of channels in the image that this is for

Return type

Dict[str, *Sequence*[*ConstrainedIntValue*]]

validate_channel_display_groups

validate_channel_display_groups(*groups: Optional[Union[Dict[str, Sequence[ConstrainedIntValue]], Sequence[Sequence[ConstrainedIntValue]]]]*)

Validate channel display groups object.

See PlotOptions.channel_display_groups.

Parameters

groups (*Optional[Union[Dict[str, Sequence[ConstrainedIntValue]], Sequence[Sequence[ConstrainedIntValue]]]]*) –

9.3.6 learner_pipeline

Classes

LearnerPipeline	Simple Pipeline that is a wrapper around Learner.main()

LearnerPipeline

class LearnerPipeline

Bases: *Pipeline*

Simple Pipeline that is a wrapper around Learner.main()

This supports the ability to use the pytorch_learner package to train models using the RV pipeline package and its runner functionality without the rest of RV.

Attributes

commands
gpu_commands
split_commands

__init__(*config:* PipelineConfig, *tmp_dir: str*)

Constructor

Parameters

- **config** (PipelineConfig) – the configuration of this pipeline
- **tmp_dir** (*str*) – the root any temporary directories created by running this pipeline

Methods

__init__(config, tmp_dir)	Constructor
test_cpu([split_ind, num_splits])	A command to test the ability to run split jobs on CPU.
test_gpu()	A command to test the ability to run on GPU.
train()	

__init__(*config:* PipelineConfig, *tmp_dir:* *str*)

Constructor

Parameters

- **config** (PipelineConfig) – the configuration of this pipeline
- **tmp_dir** (*str*) – the root any temporary directories created by running this pipeline

test_cpu(*split_ind:* *int* = *0*, *num_splits:* *int* = *1*)

A command to test the ability to run split jobs on CPU.

Parameters

- **split_ind** (*int*) –
- **num_splits** (*int*) –

test_gpu()

A command to test the ability to run on GPU.

train()

commands: List[str] = ['train']

gpu_commands: List[str] = ['train']

split_commands: List[str] = ['test_cpu']

9.3.7 learner_pipeline_config

Configs

LearnerPipelineConfig	Configure a *LearnerPipeline*.

LearnerPipelineConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model LearnerPipelineConfig

Configure a *LearnerPipeline*.

```
{
   "title": "LearnerPipelineConfig",
   "description": "Configure a :class:`.LearnerPipeline`.",
   "type": "object",
   "properties": {
      "root_uri": {
         "title": "Root Uri",
         "description": "The root URI for output generated by the pipeline",
         "type": "string"
      },
      "rv_config": {
         "title": "Rv Config",
         "description": "Used to store serialized RVConfig so pipeline can run in␣
↪remote environment with the local RVConfig. This should not be set explicitly by␣
↪users -- it is only used by the runner when running a remote pipeline.",
         "type": "object"
      },
      "plugin_versions": {
         "title": "Plugin Versions",
         "description": "Used to store a mapping of plugin module paths to the␣
↪latest version number. This should not be set explicitly by users -- it is set␣
↪automatically when serializing and saving the config to disk.",
         "type": "object",
         "additionalProperties": {
            "type": "integer"
         }
      },
      "type_hint": {
         "title": "Type Hint",
         "default": "learner_pipeline",
         "enum": [
            "learner_pipeline"
         ],
         "type": "string"
      },
      "learner": {
         "$ref": "#/definitions/LearnerConfig"
      }
   },
   "required": [
      "learner"
   ],
   "additionalProperties": false,
   "definitions": {
      "Backbone": {
         "title": "Backbone",
         "description": "An enumeration.",
         "enum": [
            "alexnet",
            "densenet121",
            "densenet169",
            "densenet201",
            "densenet161",
```


(continues on next page)

(continued from previous page)


```
            "googlenet",
            "inception_v3",
            "mnasnet0_5",
            "mnasnet0_75",
            "mnasnet1_0",
            "mnasnet1_3",
            "mobilenet_v2",
            "resnet18",
            "resnet34",
            "resnet50",
            "resnet101",
            "resnet152",
            "resnext50_32x4d",
            "resnext101_32x8d",
            "wide_resnet50_2",
            "wide_resnet101_2",
            "shufflenet_v2_x0_5",
            "shufflenet_v2_x1_0",
            "shufflenet_v2_x1_5",
            "shufflenet_v2_x2_0",
            "squeezenet1_0",
            "squeezenet1_1",
            "vgg11",
            "vgg11_bn",
            "vgg13",
            "vgg13_bn",
            "vgg16",
            "vgg16_bn",
            "vgg19_bn",
            "vgg19"
         ]
      },
      "ExternalModuleConfig": {
         "title": "ExternalModuleConfig",
         "description": "Config describing an object to be loaded via Torch Hub.",
         "type": "object",
         "properties": {
            "uri": {
               "title": "Uri",
               "description": "Local uri of a zip file, or local uri of a directory,
↪or remote uri of zip file.",
               "minLength": 1,
               "type": "string"
            },
            "github_repo": {
               "title": "Github Repo",
               "description": "<repo-owner>/<repo-name>[:tag]",
               "pattern": ".+/.+",
               "type": "string"
            },
            "name": {
               "title": "Name",
```


(continues on next page)

(continued from previous page)

```
                "description": "Name of the folder in which to extract/copy the␣
↪definition files.",
                "minLength": 1,
                "type": "string"
            },
            "entrypoint": {
                "title": "Entrypoint",
                "description": "Name of a callable present in hubconf.py. See docs␣
↪for torch.hub for details.",
                "minLength": 1,
                "type": "string"
            },
            "entrypoint_args": {
                "title": "Entrypoint Args",
                "description": "Args to pass to the entrypoint. Must be serializable.
↪",
                "default": [],
                "type": "array",
                "items": {}
            },
            "entrypoint_kwargs": {
                "title": "Entrypoint Kwargs",
                "description": "Keyword args to pass to the entrypoint. Must be␣
↪serializable.",
                "default": {},
                "type": "object"
            },
            "force_reload": {
                "title": "Force Reload",
                "description": "Force reload of module definition.",
                "default": false,
                "type": "boolean"
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "external-module",
                "enum": [
                    "external-module"
                ],
                "type": "string"
            }
        },
        "required": [
            "entrypoint"
        ],
        "additionalProperties": false
    },
    "ModelConfig": {
        "title": "ModelConfig",
        "description": "Config related to models.",
        "type": "object",
        "properties": {
```

(continues on next page)

(continued from previous page)

```
            "backbone": {
               "description": "The torchvision.models backbone to use.",
               "default": "resnet18",
               "allOf": [
                  {
                     "$ref": "#/definitions/Backbone"
                  }
               ]
            },
            "pretrained": {
               "title": "Pretrained",
               "description": "If True, use ImageNet weights. If False, use random␣
↪initialization.",
               "default": true,
               "type": "boolean"
            },
            "init_weights": {
               "title": "Init Weights",
               "description": "URI of PyTorch model weights used to initialize␣
↪model. If set, this supercedes the pretrained option.",
               "type": "string"
            },
            "load_strict": {
               "title": "Load Strict",
               "description": "If True, the keys in the state dict referenced by␣
↪init_weights must match exactly. Setting this to False can be useful if you just␣
↪want to load the backbone of a model.",
               "default": true,
               "type": "boolean"
            },
            "external_def": {
               "title": "External Def",
               "description": "If specified, the model will be built from the␣
↪definition from this external source, using Torch Hub.",
               "allOf": [
                  {
                     "$ref": "#/definitions/ExternalModuleConfig"
                  }
               ]
            },
            "type_hint": {
               "title": "Type Hint",
               "default": "model",
               "enum": [
                  "model"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      },
      "SolverConfig": {
```

(continues on next page)

(continued from previous page)


```
        "title": "SolverConfig",
        "description": "Config related to solver aka optimizer.",
        "type": "object",
        "properties": {
           "lr": {
              "title": "Lr",
              "description": "Learning rate.",
              "default": 0.0001,
              "exclusiveMinimum": 0,
              "type": "number"
           },
           "num_epochs": {
              "title": "Num Epochs",
              "description": "Number of epochs (ie. sweeps through the whole␣
↪training set).",
              "default": 10,
              "exclusiveMinimum": 0,
              "type": "integer"
           },
           "test_num_epochs": {
              "title": "Test Num Epochs",
              "description": "Number of epochs to use in test mode.",
              "default": 2,
              "exclusiveMinimum": 0,
              "type": "integer"
           },
           "test_batch_sz": {
              "title": "Test Batch Sz",
              "description": "Batch size to use in test mode.",
              "default": 4,
              "exclusiveMinimum": 0,
              "type": "integer"
           },
           "overfit_num_steps": {
              "title": "Overfit Num Steps",
              "description": "Number of optimizer steps to use in overfit mode.",
              "default": 1,
              "exclusiveMinimum": 0,
              "type": "integer"
           },
           "sync_interval": {
              "title": "Sync Interval",
              "description": "The interval in epochs for each sync to the cloud.",
              "default": 1,
              "exclusiveMinimum": 0,
              "type": "integer"
           },
           "batch_sz": {
              "title": "Batch Sz",
              "description": "Batch size.",
              "default": 32,
              "exclusiveMinimum": 0,
```


(continues on next page)

(continued from previous page)


```
                "type": "integer"
            },
            "one_cycle": {
                "title": "One Cycle",
                "description": "If True, use triangular LR scheduler with a single␣
↪cycle across all epochs with start and end LR being lr/10 and the peak being lr.",
                "default": true,
                "type": "boolean"
            },
            "multi_stage": {
                "title": "Multi Stage",
                "description": "List of epoch indices at which to divide LR by 10.",
                "default": [],
                "type": "array",
                "items": {}
            },
            "class_loss_weights": {
                "title": "Class Loss Weights",
                "description": "Class weights for weighted loss.",
                "type": "array",
                "items": {
                    "type": "number"
                }
            },
            "ignore_class_index": {
                "title": "Ignore Class Index",
                "description": "If specified, this index is ignored when computing␣
↪the loss. See pytorch documentation for nn.CrossEntropyLoss for more details.␣
↪This can also be negative, in which case it is treated as a negative slice index␣
↪i.e. -1 = last index, -2 = second-last index, and so on.",
                "type": "integer"
            },
            "external_loss_def": {
                "title": "External Loss Def",
                "description": "If specified, the loss will be built from the␣
↪definition from this external source, using Torch Hub.",
                "allOf": [
                    {
                        "$ref": "#/definitions/ExternalModuleConfig"
                    }
                ]
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "solver",
                "enum": [
                    "solver"
                ],
                "type": "string"
            }
        },
        "additionalProperties": false
```


(continues on next page)

(continued from previous page)


```
      },
      "PlotOptions": {
         "title": "PlotOptions",
         "description": "Config related to plotting.",
         "type": "object",
         "properties": {
            "transform": {
               "title": "Transform",
               "description": "An Albumentations transform serialized as a dict␣
↪that will be applied to each image before it is plotted. Mainly useful for␣
↪undoing any data transformation that you do not want included in the plot, such␣
↪as normalization. The default value will shift and scale the image so the values␣
↪range from 0.0 to 1.0 which is the expected range for the plotting function. This␣
↪default is useful for cases where the values after normalization are close to␣
↪zero which makes the plot difficult to see.",
               "default": {
                  "__version__": "1.3.0",
                  "transform": {
                     "__class_fullname__": "rastervision.pytorch_learner.utils.
↪utils.MinMaxNormalize",
                     "always_apply": false,
                     "p": 1.0,
                     "min_val": 0.0,
                     "max_val": 1.0,
                     "dtype": 5
                  }
               },
               "type": "object"
            },
            "channel_display_groups": {
               "title": "Channel Display Groups",
               "description": "Groups of image channels to display together as a␣
↪subplot when plotting the data and predictions. Can be a list or tuple of groups␣
↪(e.g. [(0, 1, 2), (3,)]) or a dict containing title-to-group mappings (e.g. {\
↪"RGB\": [0, 1, 2], \"IR\": [3]}), where each group is a list or tuple of channel␣
↪indices and title is a string that will be used as the title of the subplot for␣
↪that group.",
               "anyOf": [
                  {
                     "type": "object",
                     "additionalProperties": {
                        "type": "array",
                        "items": {
                           "type": "integer",
                           "minimum": 0
                        }
                     }
                  },
                  {
                     "type": "array",
                     "items": {
                        "type": "array",
```


(continues on next page)

(continued from previous page)

```
                        "items": {
                            "type": "integer",
                            "minimum": 0
                        }
                    }
                }
            ]
        },
        "type_hint": {
            "title": "Type Hint",
            "default": "plot_options",
            "enum": [
                "plot_options"
            ],
            "type": "string"
        }
    },
    "additionalProperties": false
},
"DataConfig": {
    "title": "DataConfig",
    "description": "Config related to dataset for training and testing.",
    "type": "object",
    "properties": {
        "class_names": {
            "title": "Class Names",
            "description": "Names of classes.",
            "default": [],
            "type": "array",
            "items": {
                "type": "string"
            }
        },
        "class_colors": {
            "title": "Class Colors",
            "description": "Colors used to display classes. Can be color 3-
↪tuples in list form.",
            "type": "array",
            "items": {
                "anyOf": [
                    {
                        "type": "string"
                    },
                    {
                        "type": "array",
                        "minItems": 3,
                        "maxItems": 3,
                        "items": [
                            {
                                "type": "integer"
                            },
                            {
```

(continues on next page)

(continued from previous page)

```
                            "type": "integer"
                        },
                        {
                            "type": "integer"
                        }
                    ]
                }
            ]
        }
    },
    "img_channels": {
        "title": "Img Channels",
        "description": "The number of channels of the training images.",
        "exclusiveMinimum": 0,
        "type": "integer"
    },
    "img_sz": {
        "title": "Img Sz",
        "description": "Length of a side of each image in pixels. This is␣
↪the size to transform it to during training, not the size in the raw dataset.",
        "default": 256,
        "exclusiveMinimum": 0,
        "type": "integer"
    },
    "train_sz": {
        "title": "Train Sz",
        "description": "If set, the number of training images to use. If␣
↪fewer images exist, then an exception will be raised.",
        "type": "integer"
    },
    "train_sz_rel": {
        "title": "Train Sz Rel",
        "description": "If set, the proportion of training images to use.",
        "type": "number"
    },
    "num_workers": {
        "title": "Num Workers",
        "description": "Number of workers to use when DataLoader makes␣
↪batches.",
        "default": 4,
        "type": "integer"
    },
    "augmentors": {
        "title": "Augmentors",
        "description": "Names of albumentations augmentors to use for␣
↪training batches. Choices include: ['Blur', 'RandomRotate90', 'HorizontalFlip',
↪'VerticalFlip', 'GaussianBlur', 'GaussNoise', 'RGBShift', 'ToGray'].␣
↪Alternatively, a custom transform can be provided via the aug_transform option.",
        "default": [
            "RandomRotate90",
            "HorizontalFlip",
            "VerticalFlip"
```

(continues on next page)

(continued from previous page)


```
                ],
                "type": "array",
                "items": {
                    "type": "string"
                }
            },
            "base_transform": {
                "title": "Base Transform",
                "description": "An Albumentations transform serialized as a dict␣
↪that will be applied to all datasets: training, validation, and test. This␣
↪transformation is in addition to the resizing due to img_sz. This is useful for,␣
↪for example, applying the same normalization to all datasets.",
                "type": "object"
            },
            "aug_transform": {
                "title": "Aug Transform",
                "description": "An Albumentations transform serialized as a dict␣
↪that will be applied as data augmentation to the training dataset. This transform␣
↪is applied before base_transform. If provided, the augmentors option is ignored.",
                "type": "object"
            },
            "plot_options": {
                "title": "Plot Options",
                "description": "Options to control plotting.",
                "default": {
                    "transform": {
                        "__version__": "1.3.0",
                        "transform": {
                            "__class_fullname__": "rastervision.pytorch_learner.utils.
↪utils.MinMaxNormalize",
                            "always_apply": false,
                            "p": 1.0,
                            "min_val": 0.0,
                            "max_val": 1.0,
                            "dtype": 5
                        }
                    },
                    "channel_display_groups": null,
                    "type_hint": "plot_options"
                },
                "allOf": [
                    {
                        "$ref": "#/definitions/PlotOptions"
                    }
                ]
            },
            "preview_batch_limit": {
                "title": "Preview Batch Limit",
                "description": "Optional limit on the number of items in the preview␣
↪plots produced during training.",
                "type": "integer"
            },
```


(continues on next page)

(continued from previous page)


```
            "type_hint": {
               "title": "Type Hint",
               "default": "data",
               "enum": [
                  "data"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      },
      "LearnerConfig": {
         "title": "LearnerConfig",
         "description": "Config for Learner.",
         "type": "object",
         "properties": {
            "model": {
               "$ref": "#/definitions/ModelConfig"
            },
            "solver": {
               "$ref": "#/definitions/SolverConfig"
            },
            "data": {
               "$ref": "#/definitions/DataConfig"
            },
            "predict_mode": {
               "title": "Predict Mode",
               "description": "If True, skips training, loads model, and does final␣
↪eval.",
               "default": false,
               "type": "boolean"
            },
            "test_mode": {
               "title": "Test Mode",
               "description": "If True, uses test_num_epochs, test_batch_sz,␣
↪truncated datasets with only a single batch, image_sz that is cut in half, and␣
↪num_workers = 0. This is useful for testing that code runs correctly on CPU␣
↪without multithreading before running full job on GPU.",
               "default": false,
               "type": "boolean"
            },
            "overfit_mode": {
               "title": "Overfit Mode",
               "description": "If True, uses half image size, and instead of doing␣
↪epoch-based training, optimizes the model using a single batch repeatedly for␣
↪overfit_num_steps number of steps.",
               "default": false,
               "type": "boolean"
            },
            "eval_train": {
               "title": "Eval Train",
               "description": "If True, runs final evaluation on training set (in␣
```


(continues on next page)

(continued from previous page)

```
↪addition to test set). Useful for debugging.",
                "default": false,
                "type": "boolean"
            },
            "save_model_bundle": {
                "title": "Save Model Bundle",
                "description": "If True, saves a model bundle at the end of training␣
↪which is zip file with model and this LearnerConfig which can be used to make␣
↪predictions on new images at a later time.",
                "default": true,
                "type": "boolean"
            },
            "log_tensorboard": {
                "title": "Log Tensorboard",
                "description": "Save Tensorboard log files at the end of each epoch.
↪",
                "default": true,
                "type": "boolean"
            },
            "run_tensorboard": {
                "title": "Run Tensorboard",
                "description": "run Tensorboard server during training",
                "default": false,
                "type": "boolean"
            },
            "output_uri": {
                "title": "Output Uri",
                "description": "URI of where to save output",
                "type": "string"
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "learner",
                "enum": [
                    "learner"
                ],
                "type": "string"
            }
        },
        "required": [
            "solver",
            "data"
        ],
        "additionalProperties": false
    }
  }
}
```

Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *learner (rastervision.pytorch_learner.learner_config.LearnerConfig)*
- *plugin_versions (Optional[Dict[str, int]])*
- *root_uri (str)*
- *rv_config (dict)*
- *type_hint (Literal['learner_pipeline'])*

field learner: *LearnerConfig* [Required]

field plugin_versions: Optional[Dict[str, int]] = None

Used to store a mapping of plugin module paths to the latest version number. This should not be set explicitly by users – it is set automatically when serializing and saving the config to disk.

field root_uri: str = None

The root URI for output generated by the pipeline

field rv_config: dict = None

Used to store serialized RVConfig so pipeline can run in remote environment with the local RVConfig. This should not be set explicitly by users – it is only used by the runner when running a remote pipeline.

field type_hint: Literal['learner_pipeline'] = 'learner_pipeline'

build(*tmp_dir*)

Return a pipeline based on this configuration.

Subclasses should override this to return an instance of the corresponding subclass of Pipeline.

Parameters
tmp_dir – root of any temporary directory to pass to pipeline

get_config_uri() → str

Get URI of serialized version of this PipelineConfig.

Return type
str

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update()

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

9.3.8 object_detection_learner

Classes

ObjectDetectionLearner

ObjectDetectionLearner

class ObjectDetectionLearner

Bases: *Learner*

__init__(*cfg: LearnerConfig*, *output_dir: Optional[str] = None*, *train_ds: Optional[Dataset] = None*, *valid_ds: Optional[Dataset] = None*, *test_ds: Optional[Dataset] = None*, *model: Optional[torch.nn.Module] = None*, *loss: Optional[Callable] = None*, *optimizer: Optional[Optimizer] = None*, *epoch_scheduler: Optional[_LRScheduler] = None*, *step_scheduler: Optional[_LRScheduler] = None*, *tmp_dir: Optional[str] = None*, *model_weights_path: Optional[str] = None*, *model_def_path: Optional[str] = None*, *loss_def_path: Optional[str] = None*, *training: bool = True*)

Constructor.

Parameters

- **cfg** (*LearnerConfig*) – LearnerConfig.
- **train_ds** (*Optional[Dataset], optional*) – The dataset to use for training. If None, will be generated from cfg.data. Defaults to None.
- **valid_ds** (*Optional[Dataset], optional*) – The dataset to use for validation. If None, will be generated from cfg.data. Defaults to None.
- **test_ds** (*Optional[Dataset], optional*) – The dataset to use for testing. If None, will be generated from cfg.data. Defaults to None.
- **model** (*Optional[nn.Module], optional*) – The model. If None, will be generated from cfg.model. Defaults to None.
- **loss** (*Optional[Callable], optional*) – The loss function. If None, will be generated from cfg.solver. Defaults to None.

- **optimizer** (*Optional[Optimizer], optional*) – The optimizer. If None, will be generated from cfg.solver. Defaults to None.
- **epoch_scheduler** (*Optional[_LRScheduler], optional*) – The scheduler that updates after each epoch. If None, will be generated from cfg.solver. Defaults to None.
- **step_scheduler** (*Optional[_LRScheduler], optional*) – The scheduler that updates after each optimizer-step. If None, will be generated from cfg.solver. Defaults to None.
- **tmp_dir** (*Optional[str], optional*) – A temporary directory to use for downloads etc. If None, will be auto-generated. Defaults to None.
- **model_weights_path** (*Optional[str], optional*) – URI of model weights to initialize the model with. Defaults to None.
- **model_def_path** (*Optional[str], optional*) – A local path to a directory with a hubconf.py. If provided, the model definition is imported from here. This is used when loading an external model from a model-bundle. Defaults to None.
- **loss_def_path** (*Optional[str], optional*) – A local path to a directory with a hubconf.py. If provided, the loss function definition is imported from here. This is used when loading an external loss function from a model-bundle. Defaults to None.
- **training** (*bool, optional*) – If False, the training apparatus (loss, optimizer, scheduler, logging, etc.) will not be set up and the model will be put into eval mode. If True, the training apparatus will be set up and the model will be put into training mode. Defaults to True.
- **output_dir** (*Optional[str]*) –

Methods

__init__(cfg[, output_dir, train_ds, ...])	Constructor.
build_dataloaders()	Set the DataLoaders for train, validation, and test sets.
build_datasets()	
build_epoch_scheduler([start_epoch])	Returns an LR scheduler that changes the LR each epoch.
build_loss([loss_def_path])	Build a loss Callable.
build_metric_names()	Returns names of metrics used to validate model at each epoch.
build_model([model_def_path])	Override to pass img_sz.
build_optimizer()	Returns optimizer.
build_step_scheduler([start_epoch])	Returns an LR scheduler that changes the LR each step.
eval_model(split)	Evaluate model using a particular dataset split.
from_model_bundle(model_bundle_uri[, ...])	Create a Learner from a model bundle.
get_collate_fn()	Returns a custom collate_fn to use in DataLoader.
get_dataloader(split)	Get the DataLoader for a split.
get_start_epoch()	Get start epoch.
get_train_sampler(train_ds)	Return a sampler to use for the training dataloader or None to not use any.
get_visualizer_class()	Returns a Visualizer class object for plotting data samples.

continues on next page

Table 5 – continued from previous page

load_checkpoint()	Load last weights from previous run if available.
load_init_weights([model_weights_path])	Load the weights to initialize model.
load_weights(uri, **kwargs)	Load model weights from a file.
log_data_stats()	Log stats about each DataSet.
main()	Main training sequence.
normalize_input(x)	Normalize x to [0, 1].
numpy_predict(x[, raw_out])	Make a prediction using an image or batch of images in numpy format.
on_epoch_end(curr_epoch, metrics)	Hook that is called at end of epoch.
on_overfit_start()	Hook that is called at start of overfit routine.
on_train_start()	Hook that is called at start of train routine.
output_to_numpy(out)	Convert output of model to numpy format.
overfit()	Optimize model using the same batch repeatedly.
plot_dataloader(dl, output_path[, ...])	Plot images and ground truth labels for a DataLoader.
plot_dataloaders([batch_limit, show])	Plot images and ground truth labels for all DataLoaders.
plot_predictions(split[, batch_limit, show])	Plot predictions for a split.
post_forward(x)	Post process output of call to model().
predict(x[, raw_out])	Make prediction for an image or batch of images.
predict_dataloader(dl[, batched_output, ...])	Returns an iterator over predictions on the given dataloader.
predict_dataset(dataset[, return_format, ...])	Returns an iterator over predictions on the given dataset.
prob_to_pred(x)	Convert a Tensor with prediction probabilities to class ids.
run_tensorboard()	Run TB server serving logged stats.
save_model_bundle()	Save a model bundle.
setup_data()	Set datasets and dataLoaders for train, validation, and test sets.
setup_loss([loss_def_path])	Setup self.loss.
setup_model([model_weights_path, model_def_path])	Override to apply the TorchVisionODAdapter wrapper.
setup_tensorboard()	Setup for logging stats to TB.
setup_training([loss_def_path])	
stop_tensorboard()	Stop TB logging and server if it's running.
sync_from_cloud()	Sync any previous output in the cloud to output_dir.
sync_to_cloud()	Sync any output to the cloud at output_uri.
to_batch(x)	Ensure that image array has batch dimension.
to_device(x, device)	Load Tensors onto a device.
train([epochs])	Training loop that will attempt to resume training if appropriate.
train_end(outputs, num_samples)	Aggregate the ouput of train_step at the end of the epoch.
train_epoch(optimizer[, step_scheduler])	Train for a single epoch.
train_step(batch, batch_ind)	Compute loss for a single training batch.
validate_end(outputs, num_samples)	Aggregate the ouput of validate_step at the end of the epoch.
validate_epoch(dl)	Validate for a single epoch.
validate_step(batch, batch_ind)	Compute metrics on validation batch.

__init__(*cfg: LearnerConfig*, *output_dir: Optional[str] = None*, *train_ds: Optional[Dataset] = None*, *valid_ds: Optional[Dataset] = None*, *test_ds: Optional[Dataset] = None*, *model: Optional[torch.nn.Module] = None*, *loss: Optional[Callable] = None*, *optimizer: Optional[Optimizer] = None*, *epoch_scheduler: Optional[_LRScheduler] = None*, *step_scheduler: Optional[_LRScheduler] = None*, *tmp_dir: Optional[str] = None*, *model_weights_path: Optional[str] = None*, *model_def_path: Optional[str] = None*, *loss_def_path: Optional[str] = None*, *training: bool = True*)

Constructor.

Parameters

- **cfg** (LearnerConfig) – LearnerConfig.
- **train_ds** (*Optional[Dataset], optional*) – The dataset to use for training. If None, will be generated from cfg.data. Defaults to None.
- **valid_ds** (*Optional[Dataset], optional*) – The dataset to use for validation. If None, will be generated from cfg.data. Defaults to None.
- **test_ds** (*Optional[Dataset], optional*) – The dataset to use for testing. If None, will be generated from cfg.data. Defaults to None.
- **model** (*Optional[nn.Module], optional*) – The model. If None, will be generated from cfg.model. Defaults to None.
- **loss** (*Optional[Callable], optional*) – The loss function. If None, will be generated from cfg.solver. Defaults to None.
- **optimizer** (*Optional[Optimizer], optional*) – The optimizer. If None, will be generated from cfg.solver. Defaults to None.
- **epoch_scheduler** (*Optional[_LRScheduler], optional*) – The scheduler that updates after each epoch. If None, will be generated from cfg.solver. Defaults to None.
- **step_scheduler** (*Optional[_LRScheduler], optional*) – The scheduler that updates after each optimizer-step. If None, will be generated from cfg.solver. Defaults to None.
- **tmp_dir** (*Optional[str], optional*) – A temporary directory to use for downloads etc. If None, will be auto-generated. Defaults to None.
- **model_weights_path** (*Optional[str], optional*) – URI of model weights to initialize the model with. Defaults to None.
- **model_def_path** (*Optional[str], optional*) – A local path to a directory with a hubconf.py. If provided, the model definition is imported from here. This is used when loading an external model from a model-bundle. Defaults to None.
- **loss_def_path** (*Optional[str], optional*) – A local path to a directory with a hubconf.py. If provided, the loss function definition is imported from here. This is used when loading an external loss function from a model-bundle. Defaults to None.
- **training** (*bool, optional*) – If False, the training apparatus (loss, optimizer, scheduler, logging, etc.) will not be set up and the model will be put into eval mode. If True, the training apparatus will be set up and the model will be put into training mode. Defaults to True.
- **output_dir** (*Optional[str]*) –

build_dataloaders() → Tuple[torch.utils.data.DataLoader, torch.utils.data.DataLoader, torch.utils.data.DataLoader]

Set the DataLoaders for train, validation, and test sets.

Return type
Tuple[torch.utils.data.DataLoader, torch.utils.data.DataLoader, torch.utils.data.DataLoader]

build_datasets() → Tuple[Dataset, Dataset, Dataset]

Return type
Tuple[Dataset, Dataset, Dataset]

build_epoch_scheduler(*start_epoch: int = 0*) → _LRScheduler

Returns an LR scheduler that changes the LR each epoch.

Parameters
start_epoch (*int*) –

Return type
_LRScheduler

build_loss(*loss_def_path: Optional[str] = None*) → Callable

Build a loss Callable.

Parameters
loss_def_path (*Optional[str]*) –

Return type
Callable

build_metric_names()

Returns names of metrics used to validate model at each epoch.

build_model(*model_def_path: Optional[str] = None*) → nn.Module

Override to pass img_sz.

Parameters
model_def_path (*Optional[str]*) –

Return type
nn.Module

build_optimizer() → Optimizer

Returns optimizer.

Return type
Optimizer

build_step_scheduler(*start_epoch: int = 0*) → _LRScheduler

Returns an LR scheduler that changes the LR each step.

Parameters
start_epoch (*int*) –

Return type
_LRScheduler

eval_model(*split: str*)

Evaluate model using a particular dataset split.

Gets validation metrics and saves them along with prediction plots.

Parameters
split (*str*) – the dataset split to use: train, valid, or test.

classmethod from_model_bundle(*model_bundle_uri: str*, *tmp_dir: Optional[str] = None*, *cfg: Optional[LearnerConfig] = None*, *training: bool = False*, ***kwargs*) → *Learner*

Create a Learner from a model bundle.

Note: This is the bundle saved in `train/model-bundle.zip` and not `bundle/model-bundle.zip`.

Parameters

- **model_bundle_uri** (*str*) – URI of the model bundle.
- **tmp_dir** (*Optional[str], optional*) – Optional temporary directory. Will be used for unzipping bundle and also passed to the default constructor. If None, will be auto-generated. Defaults to None.
- **cfg** (*Optional[LearnerConfig], optional*) – If None, will be read from the bundle. Defaults to None.
- **training** (*bool, optional*) – If False, the training apparatus (loss, optimizer, scheduler, logging, etc.) will not be set up and the model will be put into eval mode. If True, the training apparatus will be set up and the model will be put into training mode. Defaults to True.
- ****kwargs** – See *Learner.__init__()*.

Raises

FileNotFoundError – If using custom Albumentations transforms and definition file is not found in bundle.

Returns

Object of the Learner subclass on which this was called.

Return type

Learner

get_collate_fn()

Returns a custom collate_fn to use in DataLoader.

None is returned if default collate_fn should be used.

See https://pytorch.org/docs/stable/data.html#working-with-collate-fn

get_dataloader(*split: str*) → torch.utils.data.DataLoader

Get the DataLoader for a split.

Parameters

split (*str*) – a split name which can be train, valid, or test

Return type

torch.utils.data.DataLoader

get_start_epoch() → int

Get start epoch.

If training was interrupted, this returns the last complete epoch + 1.

Return type

int

get_train_sampler(*train_ds: Dataset*) → Optional[Sampler]

Return a sampler to use for the training dataloader or None to not use any.

Parameters
train_ds (*Dataset*) –

Return type
Optional[Sampler]

get_visualizer_class()

Returns a Visualizer class object for plotting data samples.

load_checkpoint()

Load last weights from previous run if available.

load_init_weights(*model_weights_path: Optional[str] = None*) → None

Load the weights to initialize model.

Parameters
model_weights_path (*Optional[str]*) –

Return type
None

load_weights(*uri: str*, ***kwargs*) → None

Load model weights from a file.

Parameters
uri (*str*) –

Return type
None

log_data_stats()

Log stats about each DataSet.

main()

Main training sequence.

This plots the dataset, runs a training and validation loop (which will resume if interrupted), logs stats, plots predictions, and syncs results to the cloud.

normalize_input(*x: ndarray*) → ndarray

Normalize x to [0, 1].

If x.dtype is a subtype of np.unsignedinteger, normalize it to [0, 1] using the max possible value of that dtype. Otherwise, assume it is in [0, 1] already and do nothing.

Parameters
x (*np.ndarray*) – an image or batch of images

Returns
the same array scaled to [0, 1].

Return type
ndarray

numpy_predict(*x: ndarray*, *raw_out: bool = False*) → ndarray

Make a prediction using an image or batch of images in numpy format. If x.dtype is a subtype of np.unsignedinteger, it will be normalized to [0, 1] using the max possible value of that dtype. Otherwise, x will be assumed to be in [0, 1] already and will be cast to torch.float32 directly.

Parameters

- **x** (*ndarray*) – (ndarray) of shape [height, width, channels] or [batch_sz, height, width, channels]
- **raw_out** (*bool*) – if True, return prediction probabilities

Returns
predictions using numpy arrays

Return type
ndarray

on_epoch_end(*curr_epoch*, *metrics*)

Hook that is called at end of epoch.

Writes metrics to CSV and TB, and saves model.

on_overfit_start()

Hook that is called at start of overfit routine.

on_train_start()

Hook that is called at start of train routine.

output_to_numpy(*out: Iterable[BoxList]*) → Union[Dict[str, ndarray], List[Dict[str, ndarray]]]

Convert output of model to numpy format.

Parameters
out (*Iterable[BoxList]*) – the output of the model in PyTorch format

Return type
Union[*Dict*[str, *ndarray*], *List*[*Dict*[str, *ndarray*]]]

Returns: the output of the model in numpy format

overfit()

Optimize model using the same batch repeatedly.

plot_dataloader(*dl: torch.utils.data.DataLoader*, *output_path: str*, *batch_limit: Optional[int] = None*, *show: bool = False*)

Plot images and ground truth labels for a DataLoader.

Parameters

- **dl** (*torch.utils.data.DataLoader*) –
- **output_path** (*str*) –
- **batch_limit** (*Optional[int]*) –
- **show** (*bool*) –

plot_dataloaders(*batch_limit: Optional[int] = None*, *show: bool = False*)

Plot images and ground truth labels for all DataLoaders.

Parameters

- **batch_limit** (*Optional[int]*) –
- **show** (*bool*) –

plot_predictions(*split: str*, *batch_limit: Optional[int] = None*, *show: bool = False*)

Plot predictions for a split.

Uses the first batch for the corresponding DataLoader.

Parameters

- **split** (*str*) – dataset split. Can be train, valid, or test.
- **batch_limit** (*Optional[int]*) – optional limit on (rendered) batch size
- **show** (*bool*) –

post_forward(*x: Any*) → Any

Post process output of call to model().

Useful for when predictions are inside a structure returned by model().

Parameters
x (*Any*) –

Return type
Any

predict(*x: torch.Tensor*, *raw_out: bool = False*) → Any

Make prediction for an image or batch of images.

Parameters

- **x** (*Tensor*) – Image or batch of images as a float Tensor with pixel values normalized to [0, 1].
- **raw_out** (*bool*) – if True, return prediction probabilities

Returns

the predictions, in probability form if raw_out is True, in class_id form
otherwise

Return type
Any

predict_dataloader(*dl: torch.utils.data.DataLoader*, *batched_output: bool = True*, *return_format: Literal['xyz', 'yz', 'z'] = 'z'*, *raw_out: bool = True*, *predict_kw: dict = {}*) → Union[Iterator[Any], Iterator[Tuple[Any, ...]]]

Returns an iterator over predictions on the given dataloader.

Parameters

- **dl** (*DataLoader*) – The dataloader to make predictions on.
- **batched_output** (*bool, optional*) – If True, return batches of x, y, z as defined by the dataloader. If False, unroll the batches into individual items. Defaults to True.
- **return_format** (*Literal['xyz', 'yz', 'z'], optional*) – Format of the return elements of the returned iterator. Must be one of: 'xyz', 'yz', and 'z'. If 'xyz', elements are 3-tuples of x, y, and z. If 'yz', elements are 2-tuples of y and z. If 'z', elements are (non-tuple) values of z. Where x = input image, y = ground truth, and z = prediction. Defaults to 'z'.
- **raw_out** (*bool, optional*) – If true, return raw predicted scores. Defaults to True.
- **predict_kw** (*dict*) – Dict with keywords passed to Learner.predict(). Useful if a Learner subclass implements a custom predict() method.

Raises
ValueError – If return_format is not one of the allowed values.

Returns

If return_format
is 'z', the returned value is an iterator of whatever type the predictions are. Otherwise, the returned value is an iterator of tuples.

Return type
Union[Iterator[Any], Iterator[Tuple[Any, ...]]]

predict_dataset(*dataset: Dataset*, *return_format: Literal['xyz', 'yz', 'z'] = 'z'*, *raw_out: bool = True*, *numpy_out: bool = False*, *predict_kw: dict = {}*, *dataloader_kw: dict = {}*, *progress_bar: bool = True*, *progress_bar_kw: dict = {}*) → Union[Iterator[Any], Iterator[Tuple[Any, ...]]]

Returns an iterator over predictions on the given dataset.

Parameters

- **dataset** (*Dataset*) – The dataset to make predictions on.
- **return_format** (*Literal['xyz', 'yz', 'z'], optional*) – Format of the return elements of the returned iterator. Must be one of: 'xyz', 'yz', and 'z'. If 'xyz', elements are 3-tuples of x, y, and z. If 'yz', elements are 2-tuples of y and z. If 'z', elements are (non-tuple) values of z. Where x = input image, y = ground truth, and z = prediction. Defaults to 'z'.
- **raw_out** (*bool, optional*) – If true, return raw predicted scores. Defaults to True.
- **numpy_out** (*bool, optional*) – If True, convert predictions to numpy arrays before returning. Defaults to False.
- **predict_kw** (*dict*) – Dict with keywords passed to Learner.predict(). Useful if a Learner subclass implements a custom predict() method.
- **dataloader_kw** (*dict*) – Dict with keywords passed to the DataLoader constructor.
- **progress_bar** (*bool, optional*) – If True, display a progress bar. Since this function returns an iterator, the progress bar won't be visible until the iterator is consumed. Defaults to True.
- **progress_bar_kw** (*dict*) – Dict with keywords passed to tqdm.

Raises
ValueError – If return_format is not one of the allowed values.

Returns

If return_format
is 'z', the returned value is an iterator of whatever type the predictions are. Otherwise, the returned value is an iterator of tuples.

Return type
Union[Iterator[Any], Iterator[Tuple[Any, ...]]]

prob_to_pred(*x*)

Convert a Tensor with prediction probabilities to class ids.

The class ids should be the classes with the maximum probability.

run_tensorboard()

Run TB server serving logged stats.

save_model_bundle()

Save a model bundle.

This is a zip file with the model weights in .pth format and a serialized copy of the LearningConfig, which allows for making predictions in the future.

setup_data()

Set datasets and dataLoaders for train, validation, and test sets.

setup_loss(*loss_def_path: Optional[str] = None*) → None

Setup self.loss.

Parameters

- **loss_def_path** (*str, optional*) – Loss definition path. Will be
- **None.** (*available when loading from a bundle. Defaults to*) –

Return type
None

setup_model(*model_weights_path: Optional[str] = None*, *model_def_path: Optional[str] = None*) → None

Override to apply the TorchVisionODAdapter wrapper.

Parameters

- **model_weights_path** (*Optional[str]*) –
- **model_def_path** (*Optional[str]*) –

Return type
None

setup_tensorboard()

Setup for logging stats to TB.

setup_training(*loss_def_path: Optional[str] = None*) → None

Parameters
loss_def_path (*Optional[str]*) –

Return type
None

stop_tensorboard()

Stop TB logging and server if it's running.

sync_from_cloud()

Sync any previous output in the cloud to output_dir.

sync_to_cloud()

Sync any output to the cloud at output_uri.

to_batch(*x: torch.Tensor*) → torch.Tensor

Ensure that image array has batch dimension.

Parameters
x (*torch.Tensor*) – assumed to be either image or batch of images

Returns
x with extra batch dimension of length 1 if needed

Return type
torch.Tensor

to_device(*x: Any*, *device: str*) → Any

Load Tensors onto a device.

Parameters

- **x** (*Any*) – some object with Tensors in it
- **device** (*str*) – 'cpu' or 'cuda'

Returns
x but with any Tensors in it on the device

Return type
Any

train(*epochs: Optional[int] = None*)

Training loop that will attempt to resume training if appropriate.

Parameters
epochs (*Optional[int]*) –

train_end(*outputs: List[Dict[str, float]]*, *num_samples: int*) → Dict[str, float]

Aggregate the ouput of train_step at the end of the epoch.

Parameters

- **outputs** (*List[Dict[str, float]]*) – a list of outputs of train_step
- **num_samples** (*int*) – total number of training samples processed in epoch

Return type
Dict[str, float]

train_epoch(*optimizer: Optimizer*, *step_scheduler: Optional[_LRScheduler] = None*) → Dict[str, float]

Train for a single epoch.

Parameters

- **optimizer** (*Optimizer*) –
- **step_scheduler** (*Optional[_LRScheduler]*) –

Return type
Dict[str, float]

train_step(*batch*, *batch_ind*)

Compute loss for a single training batch.

Parameters

- **batch** – batch data needed to compute loss
- **batch_ind** – index of batch within epoch

Returns
dict with 'train_loss' as key and possibly other losses

validate_end(*outputs*, *num_samples*)

Aggregate the ouput of validate_step at the end of the epoch.

Parameters

- **outputs** – a list of outputs of validate_step

- **num_samples** – total number of validation samples processed in epoch

validate_epoch(*dl: torch.utils.data.DataLoader*) → Dict[str, float]

Validate for a single epoch.

Parameters
dl (*torch.utils.data.DataLoader*) –

Return type
Dict[str, float]

validate_step(*batch*, *batch_ind*)

Compute metrics on validation batch.

Parameters

- **batch** – batch data needed to compute validation metrics
- **batch_ind** – index of batch within epoch

Returns
dict with metric names mapped to metric values

9.3.9 object_detection_learner_config

Classes

ObjectDetectionDataFormat	An enumeration.

ObjectDetectionDataFormat

class ObjectDetectionDataFormat

Bases: Enum

An enumeration.

Attributes

coco

__init__()

coco = 'coco'

Configs

ObjectDetectionDataConfig

ObjectDetectionGeoDataConfig	Configure object detection *GeoDatasets*.
ObjectDetectionGeoDataWindowConfig	Configure an object detection *GeoDataset*.
ObjectDetectionImageDataConfig	Configure *ObjectDetectionImageDatasets*.
ObjectDetectionLearnerConfig	Configure an *ObjectDetectionLearner*.
ObjectDetectionModelConfig	Configure an object detection model.

ObjectDetectionDataConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model ObjectDetectionDataConfig


```
{
   "title": "ObjectDetectionDataConfig",
   "description": "Base class that can be extended to provide custom configurations.
↪\n\nThis adds some extra methods to Pydantic BaseModel.\nSee https://pydantic-
↪docs.helpmanual.io/\n\nThe general idea is that configuration schemas can be␣
↪defined by\nsubclassing this and adding class attributes with types and\ndefault␣
↪values for each field. Configs can be defined hierarchically,\nie. a Config can␣
↪have fields which are of type Config.\nValidation, serialization, deserialization,
↪ and IDE support is\nprovided automatically based on this schema.",
   "type": "object",
   "properties": {
      "type_hint": {
         "title": "Type Hint",
         "default": "object_detection_data",
         "enum": [
            "object_detection_data"
         ],
         "type": "string"
      }
   },
   "additionalProperties": false
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *type_hint (Literal['object_detection_data'])*

field type_hint: Literal['object_detection_data'] = 'object_detection_data'

build()

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

get_bbox_params()

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(**args*, ***kwargs*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

ObjectDetectionGeoDataConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model ObjectDetectionGeoDataConfig

Configure object detection *GeoDatasets*.

See *rastervision.pytorch_learner.dataset.object_detection_dataset*.


```
{
   "title": "ObjectDetectionGeoDataConfig",
   "description": "Configure object detection :class:`GeoDatasets <.GeoDataset>`.\n\
↪nSee :mod:`rastervision.pytorch_learner.dataset.object_detection_dataset`.",
   "type": "object",
```


(continues on next page)

(continued from previous page)

```
   "properties": {
      "class_names": {
         "title": "Class Names",
         "description": "Names of classes.",
         "default": [],
         "type": "array",
         "items": {
            "type": "string"
         }
      },
      "class_colors": {
         "title": "Class Colors",
         "description": "Colors used to display classes. Can be color 3-tuples in␣
↪list form.",
         "type": "array",
         "items": {
            "anyOf": [
               {
                  "type": "string"
               },
               {
                  "type": "array",
                  "minItems": 3,
                  "maxItems": 3,
                  "items": [
                     {
                        "type": "integer"
                     },
                     {
                        "type": "integer"
                     },
                     {
                        "type": "integer"
                     }
                  ]
               }
            ]
         }
      },
      "img_channels": {
         "title": "Img Channels",
         "description": "The number of channels of the training images.",
         "exclusiveMinimum": 0,
         "type": "integer"
      },
      "img_sz": {
         "title": "Img Sz",
         "description": "Length of a side of each image in pixels. This is the size␣
↪to transform it to during training, not the size in the raw dataset.",
         "default": 256,
         "exclusiveMinimum": 0,
         "type": "integer"
```

(continues on next page)

(continued from previous page)

```
      },
      "train_sz": {
         "title": "Train Sz",
         "description": "If set, the number of training images to use. If fewer␣
↪images exist, then an exception will be raised.",
         "type": "integer"
      },
      "train_sz_rel": {
         "title": "Train Sz Rel",
         "description": "If set, the proportion of training images to use.",
         "type": "number"
      },
      "num_workers": {
         "title": "Num Workers",
         "description": "Number of workers to use when DataLoader makes batches.",
         "default": 4,
         "type": "integer"
      },
      "augmentors": {
         "title": "Augmentors",
         "description": "Names of albumentations augmentors to use for training␣
↪batches. Choices include: ['Blur', 'RandomRotate90', 'HorizontalFlip',
↪'VerticalFlip', 'GaussianBlur', 'GaussNoise', 'RGBShift', 'ToGray'].␣
↪Alternatively, a custom transform can be provided via the aug_transform option.",
         "default": [
            "RandomRotate90",
            "HorizontalFlip",
            "VerticalFlip"
         ],
         "type": "array",
         "items": {
            "type": "string"
         }
      },
      "base_transform": {
         "title": "Base Transform",
         "description": "An Albumentations transform serialized as a dict that will␣
↪be applied to all datasets: training, validation, and test. This transformation␣
↪is in addition to the resizing due to img_sz. This is useful for, for example,␣
↪applying the same normalization to all datasets.",
         "type": "object"
      },
      "aug_transform": {
         "title": "Aug Transform",
         "description": "An Albumentations transform serialized as a dict that will␣
↪be applied as data augmentation to the training dataset. This transform is␣
↪applied before base_transform. If provided, the augmentors option is ignored.",
         "type": "object"
      },
      "plot_options": {
         "title": "Plot Options",
         "description": "Options to control plotting.",
```

(continues on next page)

(continued from previous page)


```
         "default": {
            "transform": {
               "__version__": "1.3.0",
               "transform": {
                  "__class_fullname__": "rastervision.pytorch_learner.utils.utils.
↪MinMaxNormalize",
                  "always_apply": false,
                  "p": 1.0,
                  "min_val": 0.0,
                  "max_val": 1.0,
                  "dtype": 5
               }
            },
            "channel_display_groups": null,
            "type_hint": "plot_options"
         },
         "allOf": [
            {
               "$ref": "#/definitions/PlotOptions"
            }
         ]
      },
      "preview_batch_limit": {
         "title": "Preview Batch Limit",
         "description": "Optional limit on the number of items in the preview plots␣
↪produced during training.",
         "type": "integer"
      },
      "type_hint": {
         "title": "Type Hint",
         "default": "object_detection_geo_data",
         "enum": [
            "object_detection_geo_data"
         ],
         "type": "string"
      },
      "scene_dataset": {
         "$ref": "#/definitions/DatasetConfig"
      },
      "window_opts": {
         "title": "Window Opts",
         "default": {},
         "anyOf": [
            {
               "$ref": "#/definitions/GeoDataWindowConfig"
            },
            {
               "type": "object",
               "additionalProperties": {
                  "$ref": "#/definitions/GeoDataWindowConfig"
               }
            }
```


(continues on next page)

(continued from previous page)


```
        ]
      }
   },
   "additionalProperties": false,
   "definitions": {
      "PlotOptions": {
         "title": "PlotOptions",
         "description": "Config related to plotting.",
         "type": "object",
         "properties": {
            "transform": {
               "title": "Transform",
               "description": "An Albumentations transform serialized as a dict␣
↪that will be applied to each image before it is plotted. Mainly useful for␣
↪undoing any data transformation that you do not want included in the plot, such␣
↪as normalization. The default value will shift and scale the image so the values␣
↪range from 0.0 to 1.0 which is the expected range for the plotting function. This␣
↪default is useful for cases where the values after normalization are close to␣
↪zero which makes the plot difficult to see.",
               "default": {
                  "__version__": "1.3.0",
                  "transform": {
                     "__class_fullname__": "rastervision.pytorch_learner.utils.
↪utils.MinMaxNormalize",
                     "always_apply": false,
                     "p": 1.0,
                     "min_val": 0.0,
                     "max_val": 1.0,
                     "dtype": 5
                  }
               },
               "type": "object"
            },
            "channel_display_groups": {
               "title": "Channel Display Groups",
               "description": "Groups of image channels to display together as a␣
↪subplot when plotting the data and predictions. Can be a list or tuple of groups␣
↪(e.g. [(0, 1, 2), (3,)]) or a dict containing title-to-group mappings (e.g. {\
↪"RGB\": [0, 1, 2], \"IR\": [3]}), where each group is a list or tuple of channel␣
↪indices and title is a string that will be used as the title of the subplot for␣
↪that group.",
               "anyOf": [
                  {
                     "type": "object",
                     "additionalProperties": {
                        "type": "array",
                        "items": {
                           "type": "integer",
                           "minimum": 0
                        }
                     }
                  },
```


(continues on next page)

(continued from previous page)

```
                {
                   "type": "array",
                   "items": {
                      "type": "array",
                      "items": {
                         "type": "integer",
                         "minimum": 0
                      }
                   }
                }
             ]
          },
          "type_hint": {
             "title": "Type Hint",
             "default": "plot_options",
             "enum": [
                "plot_options"
             ],
             "type": "string"
          }
       },
       "additionalProperties": false
    },
    "ClassConfig": {
       "title": "ClassConfig",
       "description": "Configure class information for a machine learning task.",
       "type": "object",
       "properties": {
          "names": {
             "title": "Names",
             "description": "Names of classes. The i-th class in this list will␣
↪have class ID = i.",
             "type": "array",
             "items": {
                "type": "string"
             }
          },
          "colors": {
             "title": "Colors",
             "description": "Colors used to visualize classes. Can be color␣
↪strings accepted by matplotlib or RGB tuples. If None, a random color will be␣
↪auto-generated for each class.",
             "type": "array",
             "items": {
                "anyOf": [
                   {
                      "type": "string"
                   },
                   {
                      "type": "array",
                      "items": {}
                   }
```

(continues on next page)

(continued from previous page)


```
                    ]
                }
            },
            "null_class": {
                "title": "Null Class",
                "description": "Optional name of class in `names` to use as the null␣
↪class. This is used in semantic segmentation to represent the label for imagery␣
↪pixels that are NODATA or that are missing a label. If None and the class names␣
↪include \"null\", it will automatically be used as the null class. If None, and␣
↪this Config is part of a SemanticSegmentationConfig, a null class will be added␣
↪automatically.",
                "type": "string"
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "class_config",
                "enum": [
                    "class_config"
                ],
                "type": "string"
            }
        },
        "required": [
            "names"
        ],
        "additionalProperties": false
    },
    "RasterTransformerConfig": {
        "title": "RasterTransformerConfig",
        "description": "Configure a :class:`.RasterTransformer`.",
        "type": "object",
        "properties": {
            "type_hint": {
                "title": "Type Hint",
                "default": "raster_transformer",
                "enum": [
                    "raster_transformer"
                ],
                "type": "string"
            }
        },
        "additionalProperties": false
    },
    "RasterSourceConfig": {
        "title": "RasterSourceConfig",
        "description": "Configure a :class:`.RasterSource`.",
        "type": "object",
        "properties": {
            "channel_order": {
                "title": "Channel Order",
                "description": "The sequence of channel indices to use when reading␣
↪imagery.",
```


(continues on next page)

(continued from previous page)


```
                "type": "array",
                "items": {
                   "type": "integer"
                }
             },
             "transformers": {
                "title": "Transformers",
                "default": [],
                "type": "array",
                "items": {
                   "$ref": "#/definitions/RasterTransformerConfig"
                }
             },
             "extent": {
                "title": "Extent",
                "description": "Use-specified extent in pixel coords in the form␣
↪(ymin, xmin, ymax, xmax). Useful for cropping the raster source so that only part␣
↪of the raster is read from.",
                "type": "array",
                "minItems": 4,
                "maxItems": 4,
                "items": [
                   {
                      "type": "integer"
                   },
                   {
                      "type": "integer"
                   },
                   {
                      "type": "integer"
                   },
                   {
                      "type": "integer"
                   }
                ]
             },
             "type_hint": {
                "title": "Type Hint",
                "default": "raster_source",
                "enum": [
                   "raster_source"
                ],
                "type": "string"
             }
          },
          "additionalProperties": false
       },
       "LabelSourceConfig": {
          "title": "LabelSourceConfig",
          "description": "Configure a :class:`.LabelSource`.",
          "type": "object",
          "properties": {
```


(continues on next page)

(continued from previous page)


```
            "type_hint": {
               "title": "Type Hint",
               "default": "label_source",
               "enum": [
                  "label_source"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      },
      "LabelStoreConfig": {
         "title": "LabelStoreConfig",
         "description": "Configure a :class:`.LabelStore`.",
         "type": "object",
         "properties": {
            "type_hint": {
               "title": "Type Hint",
               "default": "label_store",
               "enum": [
                  "label_store"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      },
      "SceneConfig": {
         "title": "SceneConfig",
         "description": "Configure a :class:`.Scene` comprising raster data &␣
↪labels for an AOI.\n    ",
         "type": "object",
         "properties": {
            "id": {
               "title": "Id",
               "type": "string"
            },
            "raster_source": {
               "$ref": "#/definitions/RasterSourceConfig"
            },
            "label_source": {
               "$ref": "#/definitions/LabelSourceConfig"
            },
            "label_store": {
               "$ref": "#/definitions/LabelStoreConfig"
            },
            "aoi_uris": {
               "title": "Aoi Uris",
               "description": "List of URIs of GeoJSON files that define the AOIs␣
↪for the scene. Each polygon defines an AOI which is a piece of the scene that is␣
↪assumed to be fully labeled and usable for training or validation. The AOIs are␣
↪assumed to be in EPSG:4326 coordinates.",
```


(continues on next page)

(continued from previous page)


```
                "type": "array",
                "items": {
                   "type": "string"
                }
             },
             "type_hint": {
                "title": "Type Hint",
                "default": "scene",
                "enum": [
                   "scene"
                ],
                "type": "string"
             }
          },
          "required": [
             "id",
             "raster_source"
          ],
          "additionalProperties": false
       },
       "DatasetConfig": {
          "title": "DatasetConfig",
          "description": "Configure train, validation, and test splits for a dataset.
↪",
          "type": "object",
          "properties": {
             "class_config": {
                "$ref": "#/definitions/ClassConfig"
             },
             "train_scenes": {
                "title": "Train Scenes",
                "type": "array",
                "items": {
                   "$ref": "#/definitions/SceneConfig"
                }
             },
             "validation_scenes": {
                "title": "Validation Scenes",
                "type": "array",
                "items": {
                   "$ref": "#/definitions/SceneConfig"
                }
             },
             "test_scenes": {
                "title": "Test Scenes",
                "default": [],
                "type": "array",
                "items": {
                   "$ref": "#/definitions/SceneConfig"
                }
             },
             "scene_groups": {
```


(continues on next page)

(continued from previous page)


```
                "title": "Scene Groups",
                "description": "Groupings of scenes. Should be a dict of the form: {
↪<group-name>: Set(scene_id_1, scene_id_2, ...)}. Three groups are added by␣
↪default: \"train_scenes\", \"validation_scenes\", and \"test_scenes\"",
                "default": {},
                "type": "object",
                "additionalProperties": {
                   "type": "array",
                   "items": {
                      "type": "string"
                   },
                   "uniqueItems": true
                }
             },
             "type_hint": {
                "title": "Type Hint",
                "default": "dataset",
                "enum": [
                   "dataset"
                ],
                "type": "string"
             }
          },
          "required": [
             "class_config",
             "train_scenes",
             "validation_scenes"
          ],
          "additionalProperties": false
       },
       "GeoDataWindowMethod": {
          "title": "GeoDataWindowMethod",
          "description": "An enumeration.",
          "enum": [
             "sliding",
             "random"
          ]
       },
       "GeoDataWindowConfig": {
          "title": "GeoDataWindowConfig",
          "description": "Configure a :class:`.GeoDataset`.\n\nSee :mod:
↪`rastervision.pytorch_learner.dataset.dataset`.",
          "type": "object",
          "properties": {
             "method": {
                "default": "sliding",
                "allOf": [
                   {
                      "$ref": "#/definitions/GeoDataWindowMethod"
                   }
                ]
             },
```


(continues on next page)

(continued from previous page)

```
            "size": {
               "title": "Size",
               "description": "If method = sliding, this is the size of sliding␣
↪window. If method = random, this is the size that all the windows are resized to␣
↪before they are returned. If method = random and neither size_lims nor h_lims and␣
↪w_lims have been specified, then size_lims is set to (size, size + 1).",
               "anyOf": [
                  {
                     "type": "integer",
                     "exclusiveMinimum": 0
                  },
                  {
                     "type": "array",
                     "minItems": 2,
                     "maxItems": 2,
                     "items": [
                        {
                           "type": "integer",
                           "exclusiveMinimum": 0
                        },
                        {
                           "type": "integer",
                           "exclusiveMinimum": 0
                        }
                     ]
                  }
               ]
            },
            "stride": {
               "title": "Stride",
               "description": "Stride of sliding window. Only used if method =␣
↪sliding.",
               "anyOf": [
                  {
                     "type": "integer",
                     "exclusiveMinimum": 0
                  },
                  {
                     "type": "array",
                     "minItems": 2,
                     "maxItems": 2,
                     "items": [
                        {
                           "type": "integer",
                           "exclusiveMinimum": 0
                        },
                        {
                           "type": "integer",
                           "exclusiveMinimum": 0
                        }
                     ]
                  }
```

(continues on next page)

(continued from previous page)

```
            ]
         },
         "padding": {
            "title": "Padding",
            "description": "How many pixels are windows allowed to overflow the␣
↪edges of the raster source.",
            "anyOf": [
               {
                  "type": "integer",
                  "minimum": 0
               },
               {
                  "type": "array",
                  "minItems": 2,
                  "maxItems": 2,
                  "items": [
                     {
                        "type": "integer",
                        "minimum": 0
                     },
                     {
                        "type": "integer",
                        "minimum": 0
                     }
                  ]
               }
            ]
         },
         "pad_direction": {
            "title": "Pad Direction",
            "description": "If \"end\", only pad ymax and xmax (bottom and␣
↪right). If \"start\", only pad ymin and xmin (top and left). If \"both\", pad all␣
↪sides. Has no effect if paddiong is zero. Defaults to \"end\".",
            "default": "end",
            "enum": [
               "both",
               "start",
               "end"
            ],
            "type": "string"
         },
         "size_lims": {
            "title": "Size Lims",
            "description": "[min, max) interval from which window sizes will be␣
↪uniformly randomly sampled. The upper limit is exclusive. To fix the size to a␣
↪constant value, use size_lims = (sz, sz + 1). Only used if method = random.␣
↪Specify either size_lims or h_lims and w_lims, but not both. If neither size_lims␣
↪nor h_lims and w_lims have been specified, then this will be set to (size, size +␣
↪1).",
            "type": "array",
            "minItems": 2,
            "maxItems": 2,
```

(continues on next page)

(continued from previous page)

```
                "items": [
                    {
                        "type": "integer",
                        "exclusiveMinimum": 0
                    },
                    {
                        "type": "integer",
                        "exclusiveMinimum": 0
                    }
                ]
            },
            "h_lims": {
                "title": "H Lims",
                "description": "[min, max] interval from which window heights will␣
↪be uniformly randomly sampled. Only used if method = random.",
                "type": "array",
                "minItems": 2,
                "maxItems": 2,
                "items": [
                    {
                        "type": "integer",
                        "exclusiveMinimum": 0
                    },
                    {
                        "type": "integer",
                        "exclusiveMinimum": 0
                    }
                ]
            },
            "w_lims": {
                "title": "W Lims",
                "description": "[min, max] interval from which window widths will be␣
↪uniformly randomly sampled. Only used if method = random.",
                "type": "array",
                "minItems": 2,
                "maxItems": 2,
                "items": [
                    {
                        "type": "integer",
                        "exclusiveMinimum": 0
                    },
                    {
                        "type": "integer",
                        "exclusiveMinimum": 0
                    }
                ]
            },
            "max_windows": {
                "title": "Max Windows",
                "description": "Max allowed reads from a GeoDataset. Only used if␣
↪method = random.",
                "default": 10000,
```

(continues on next page)

(continued from previous page)


```
                "minimum": 0,
                "type": "integer"
            },
            "max_sample_attempts": {
                "title": "Max Sample Attempts",
                "description": "Max attempts when trying to find a window within the␣
↪AOI of a scene. Only used if method = random and the scene has aoi_polygons␣
↪specified.",
                "default": 100,
                "exclusiveMinimum": 0,
                "type": "integer"
            },
            "efficient_aoi_sampling": {
                "title": "Efficient Aoi Sampling",
                "description": "If the scene has AOIs, sampling windows at random␣
↪anywhere in the extent and then checking if they fall within any of the AOIs can␣
↪be very inefficient. This flag enables the use of an alternate algorithm that␣
↪only samples window locations inside the AOIs. Only used if method = random and␣
↪the scene has aoi_polygons specified. Defaults to True",
                "default": true,
                "type": "boolean"
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "geo_data_window",
                "enum": [
                    "geo_data_window"
                ],
                "type": "string"
            }
        },
        "required": [
            "size"
        ],
        "additionalProperties": false
    }
  }
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *aug_transform (Optional[dict])*
- *augmentors (List[str])*
- *base_transform (Optional[dict])*
- *class_colors (Optional[List[Union[str, Tuple[int, int, int]]]])*
- *class_names (List[str])*

- *img_channels (Optional[pydantic.types.PositiveInt])*
- *img_sz (pydantic.types.PositiveInt)*
- *num_workers (int)*
- *plot_options (Optional[rastervision.pytorch_learner.learner_config.PlotOptions])*
- *preview_batch_limit (Optional[int])*
- *scene_dataset (Optional[rastervision.core.data.dataset_config.DatasetConfig])*
- *train_sz (Optional[int])*
- *train_sz_rel (Optional[float])*
- *type_hint (Literal['object_detection_geo_data'])*
- *window_opts (Union[rastervision.pytorch_learner.learner_config.GeoDataWindowConfig, Dict[str, rastervision.pytorch_learner.learner_config.GeoDataWindowConfig]])*

Validators

- ensure_class_colors » all fields
- get_class_info_from_class_config_if_needed » all fields
- *validate_albumentation_transform* » *aug_transform*
- *validate_albumentation_transform* » *base_transform*
- validate_augmentors » *augmentors*
- validate_plot_options » all fields
- validate_window_opts » *window_opts*

field aug_transform: Optional[dict] = None

An Albumentations transform serialized as a dict that will be applied as data augmentation to the training dataset. This transform is applied before base_transform. If provided, the augmentors option is ignored.

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- *validate_albumentation_transform*
- validate_plot_options

field augmentors: List[str] = ['RandomRotate90', 'HorizontalFlip', 'VerticalFlip']

Names of albumentations augmentors to use for training batches. Choices include: [‘Blur’, ‘RandomRotate90’, ‘HorizontalFlip’, ‘VerticalFlip’, ‘GaussianBlur’, ‘GaussNoise’, ‘RGBShift’, ‘ToGray’]. Alternatively, a custom transform can be provided via the aug_transform option.

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_augmentors

- validate_plot_options

field base_transform: Optional[dict] = None

An Albumentations transform serialized as a dict that will be applied to all datasets: training, validation, and test. This transformation is in addition to the resizing due to img_sz. This is useful for, for example, applying the same normalization to all datasets.

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- *validate_albumentation_transform*
- validate_plot_options

field class_colors: Optional[List[Union[str, RGBTuple]]] = None

Colors used to display classes. Can be color 3-tuples in list form.

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field class_names: List[str] = []

Names of classes.

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field img_channels: Optional[PosInt] = None

The number of channels of the training images.

Constraints

- **exclusiveMinimum** = 0

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field img_sz: PosInt = 256

Length of a side of each image in pixels. This is the size to transform it to during training, not the size in the raw dataset.

Constraints

- **exclusiveMinimum** = 0

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed

- validate_plot_options

field num_workers: int = 4

Number of workers to use when DataLoader makes batches.

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field plot_options: Optional[*PlotOptions*] = PlotOptions(transform={'__version__': '1.3.0', 'transform': {'__class_fullname__': 'rastervision.pytorch_learner.utils.utils.MinMaxNormalize', 'always_apply': False, 'p': 1.0, 'min_val': 0.0, 'max_val': 1.0, 'dtype': 5}}, channel_display_groups=None)

Options to control plotting.

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field preview_batch_limit: Optional[int] = None

Optional limit on the number of items in the preview plots produced during training.

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field scene_dataset: Optional['SceneDatasetConfig'] = None

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field train_sz: Optional[int] = None

If set, the number of training images to use. If fewer images exist, then an exception will be raised.

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field train_sz_rel: Optional[float] = None

If set, the proportion of training images to use.

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed

- validate_plot_options

field type_hint: Literal['object_detection_geo_data'] = 'object_detection_geo_data'

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field window_opts: Union[GeoDataWindowConfig, Dict[str, GeoDataWindowConfig]] = {}

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options
- validate_window_opts

build(*tmp_dir: str*, *overfit_mode: bool = False*, *test_mode: bool = False*) → Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

Parameters

- **tmp_dir** (*str*) –
- **overfit_mode** (*bool*) –
- **test_mode** (*bool*) –

Return type
Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

build_scenes(*tmp_dir: str*) → Tuple[List[*Scene*], List[*Scene*], List[*Scene*]]

Build training, validation, and test scenes.

Parameters
tmp_dir (*str*) –

Return type
Tuple[*List*[Scene], *List*[Scene], *List*[Scene]]

validator ensure_class_colors » *all fields*

Parameters
values (*dict*) –

Return type
dict

get_bbox_params()

Returns BboxParams used by albumentations for data augmentation.

validator get_class_info_from_class_config_if_needed » *all fields*

Parameters
values (*dict*) –

Return type
dict

get_custom_albumentations_transforms() → List[dict]

Returns all custom transforms found in this config.

This should return all serialized albumentations transforms with a 'lambda_transforms_path' field contained in this config or in any of its members no matter how deeply neseted.

The pupose is to make it easier to adjust their paths all at once while saving to or loading from a bundle.

Return type
List[dict]

get_data_transforms() → Tuple[BasicTransform, BasicTransform]

Get albumentations transform objects for data augmentation.

Returns
a transform that doesn't do any data augmentation 2nd tuple arg: a transform with data augmentation

Return type
1st tuple arg

make_datasets(*tmp_dir: str*, *train_tf: Optional[BasicTransform] = None*, *val_tf: Optional[BasicTransform] = None*, *test_tf: Optional[BasicTransform] = None*, ***kwargs*) → Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

Make training, validation, and test datasets.

Parameters

- **tmp_dir** (*str*) – Temporary directory to be used for building scenes.
- **train_tf** (*Optional[A.BasicTransform], optional*) – Transform for the training dataset. Defaults to None.
- **val_tf** (*Optional[A.BasicTransform], optional*) – Transform for the validation dataset. Defaults to None.
- **test_tf** (*Optional[A.BasicTransform], optional*) – Transform for the test dataset. Defaults to None.
- **kwargs** – Kwargs to pass to self.scene_to_dataset()

Returns

PyTorch-compatiable training,
validation, and test datasets.

Return type
Tuple[Dataset, Dataset, Dataset]

random_subset_dataset(*ds: torch.utils.data.Dataset*, *size: Optional[int] = None*, *fraction: Optional[ConstrainedFloatValue] = None*) → torch.utils.data.Subset

Parameters

- **ds** (*torch.utils.data.Dataset*) –

- **size** (*Optional[int]*) –
- **fraction** (*Optional[ConstrainedFloatValue]*) –

Return type
torch.utils.data.Subset

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

scene_to_dataset(*scene: ~rastervision.core.data.scene.Scene*, *transform: ~typing.Optional[~albumentations.core.transforms_interface.BasicTransform] = None*, *bbox_params: ~typing.Optional[~albumentations.core.bbox_utils.BboxParams] = <albumentations.core.bbox_utils.BboxParams object>*) → Union[*ObjectDetectionSlidingWindowGeoDataset*, *ObjectDetectionRandomWindowGeoDataset*]

Make a dataset from a single scene.

Parameters

- **scene** (Scene) –
- **transform** (*Optional[BasicTransform]*) –
- **bbox_params** (*Optional[BboxParams]*) –

Return type
Union[ObjectDetectionSlidingWindowGeoDataset, ObjectDetectionRandomWindowGeoDataset]

update(**args*, ***kwargs*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validator validate_augmentors » augmentors

Parameters
v (*str*) –

Return type
str

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises
ConfigError – if field is invalid

validator validate_plot_options » *all fields*

Parameters
values (*dict*) –

Return type
dict

validator validate_window_opts » window_opts

Parameters

- **v** (*Union[GeoDataWindowConfig, Dict[str, GeoDataWindowConfig]]*) –
- **values** (*dict*) –

Return type
Union[GeoDataWindowConfig, *Dict*[str, GeoDataWindowConfig]]

property num_classes

ObjectDetectionGeoDataWindowConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model ObjectDetectionGeoDataWindowConfig

Configure an object detection *GeoDataset*.

See *rastervision.pytorch_learner.dataset.object_detection_dataset*.


```
{
   "title": "ObjectDetectionGeoDataWindowConfig",
   "description": "Configure an object detection :class:`.GeoDataset`.\n\nSee :mod:
↪`rastervision.pytorch_learner.dataset.object_detection_dataset`.",
   "type": "object",
   "properties": {
      "method": {
         "default": "sliding",
         "allOf": [
            {
               "$ref": "#/definitions/GeoDataWindowMethod"
            }
         ]
      },
      "size": {
         "title": "Size",
         "description": "If method = sliding, this is the size of sliding window.␣
↪If method = random, this is the size that all the windows are resized to before␣
↪they are returned. If method = random and neither size_lims nor h_lims and w_lims␣
```


(continues on next page)

(continued from previous page)

```
↪have been specified, then size_lims is set to (size, size + 1).",
        "anyOf": [
           {
              "type": "integer",
              "exclusiveMinimum": 0
           },
           {
              "type": "array",
              "minItems": 2,
              "maxItems": 2,
              "items": [
                 {
                    "type": "integer",
                    "exclusiveMinimum": 0
                 },
                 {
                    "type": "integer",
                    "exclusiveMinimum": 0
                 }
              ]
           }
        ]
     },
     "stride": {
        "title": "Stride",
        "description": "Stride of sliding window. Only used if method = sliding.",
        "anyOf": [
           {
              "type": "integer",
              "exclusiveMinimum": 0
           },
           {
              "type": "array",
              "minItems": 2,
              "maxItems": 2,
              "items": [
                 {
                    "type": "integer",
                    "exclusiveMinimum": 0
                 },
                 {
                    "type": "integer",
                    "exclusiveMinimum": 0
                 }
              ]
           }
        ]
     },
     "padding": {
        "title": "Padding",
        "description": "How many pixels are windows allowed to overflow the edges␣
↪of the raster source.",
```

(continues on next page)

(continued from previous page)

```
        "anyOf": [
           {
              "type": "integer",
              "minimum": 0
           },
           {
              "type": "array",
              "minItems": 2,
              "maxItems": 2,
              "items": [
                 {
                    "type": "integer",
                    "minimum": 0
                 },
                 {
                    "type": "integer",
                    "minimum": 0
                 }
              ]
           }
        ]
     },
     "pad_direction": {
        "title": "Pad Direction",
        "description": "If \"end\", only pad ymax and xmax (bottom and right). If \
↪"start\", only pad ymin and xmin (top and left). If \"both\", pad all sides. Has␣
↪no effect if paddiong is zero. Defaults to \"end\".",
        "default": "end",
        "enum": [
           "both",
           "start",
           "end"
        ],
        "type": "string"
     },
     "size_lims": {
        "title": "Size Lims",
        "description": "[min, max) interval from which window sizes will be␣
↪uniformly randomly sampled. The upper limit is exclusive. To fix the size to a␣
↪constant value, use size_lims = (sz, sz + 1). Only used if method = random.␣
↪Specify either size_lims or h_lims and w_lims, but not both. If neither size_lims␣
↪nor h_lims and w_lims have been specified, then this will be set to (size, size +␣
↪1).",
        "type": "array",
        "minItems": 2,
        "maxItems": 2,
        "items": [
           {
              "type": "integer",
              "exclusiveMinimum": 0
           },
           {
```

(continues on next page)

(continued from previous page)

```
                "type": "integer",
                "exclusiveMinimum": 0
            }
         ]
      },
      "h_lims": {
         "title": "H Lims",
         "description": "[min, max] interval from which window heights will be␣
↪uniformly randomly sampled. Only used if method = random.",
         "type": "array",
         "minItems": 2,
         "maxItems": 2,
         "items": [
            {
               "type": "integer",
               "exclusiveMinimum": 0
            },
            {
               "type": "integer",
               "exclusiveMinimum": 0
            }
         ]
      },
      "w_lims": {
         "title": "W Lims",
         "description": "[min, max] interval from which window widths will be␣
↪uniformly randomly sampled. Only used if method = random.",
         "type": "array",
         "minItems": 2,
         "maxItems": 2,
         "items": [
            {
               "type": "integer",
               "exclusiveMinimum": 0
            },
            {
               "type": "integer",
               "exclusiveMinimum": 0
            }
         ]
      },
      "max_windows": {
         "title": "Max Windows",
         "description": "Max allowed reads from a GeoDataset. Only used if method =␣
↪random.",
         "default": 10000,
         "minimum": 0,
         "type": "integer"
      },
      "max_sample_attempts": {
         "title": "Max Sample Attempts",
         "description": "Max attempts when trying to find a window within the AOI␣
```

(continues on next page)

(continued from previous page)

```
↪of a scene. Only used if method = random and the scene has aoi_polygons specified.
↪",
         "default": 100,
         "exclusiveMinimum": 0,
         "type": "integer"
      },
      "efficient_aoi_sampling": {
         "title": "Efficient Aoi Sampling",
         "description": "If the scene has AOIs, sampling windows at random anywhere␣
↪in the extent and then checking if they fall within any of the AOIs can be very␣
↪inefficient. This flag enables the use of an alternate algorithm that only␣
↪samples window locations inside the AOIs. Only used if method = random and the␣
↪scene has aoi_polygons specified. Defaults to True",
         "default": true,
         "type": "boolean"
      },
      "type_hint": {
         "title": "Type Hint",
         "default": "object_detection_geo_data_window",
         "enum": [
            "object_detection_geo_data_window"
         ],
         "type": "string"
      },
      "ioa_thresh": {
         "title": "Ioa Thresh",
         "description": "When a box is partially outside of a training chip, it is␣
↪not clear if (a clipped version) of the box should be included in the chip. If␣
↪the IOA (intersection over area) of the box with the chip is greater than ioa_
↪thresh, it is included in the chip. Defaults to 0.8.",
         "default": 0.8,
         "type": "number"
      },
      "clip": {
         "title": "Clip",
         "description": "Clip bounding boxes to window limits when retrieving␣
↪labels for a window.",
         "default": false,
         "type": "boolean"
      },
      "neg_ratio": {
         "title": "Neg Ratio",
         "description": "The ratio of negative chips (those containing no bounding␣
↪boxes) to positive chips. This can be useful if the statistics of the background␣
↪is different in positive chips. For example, in car detection, the positive chips␣
↪will always contain roads, but no examples of rooftops since cars tend to not be␣
↪near rooftops. Defaults to None.",
         "type": "number"
      },
      "neg_ioa_thresh": {
         "title": "Neg Ioa Thresh",
         "description": "A window will be considered negative if its max IoA with␣
```

(continues on next page)

(continued from previous page)

```
↪any bounding box is less than this threshold. Defaults to 0.2.",
            "default": 0.2,
            "type": "number"
         }
      },
      "required": [
         "size"
      ],
      "additionalProperties": false,
      "definitions": {
         "GeoDataWindowMethod": {
            "title": "GeoDataWindowMethod",
            "description": "An enumeration.",
            "enum": [
               "sliding",
               "random"
            ]
         }
      }
}
```

Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *clip (bool)*
- *efficient_aoi_sampling (bool)*
- *h_lims (Optional[Tuple[pydantic.types.PositiveInt, pydantic.types.PositiveInt]])*
- *ioa_thresh (float)*
- *max_sample_attempts (pydantic.types.PositiveInt)*
- *max_windows (rastervision.pytorch_learner.learner_config.ConstrainedIntValue)*
- *method (rastervision.pytorch_learner.learner_config.GeoDataWindowMethod)*
- *neg_ioa_thresh (float)*
- *neg_ratio (Optional[float])*
- *pad_direction (Literal['both', 'start', 'end'])*
- *padding (Optional[Union[rastervision.pytorch_learner.learner_config.ConstrainedIntValue, Tuple[rastervision.pytorch_learner.learner_config.ConstrainedIntValue, rastervision.pytorch_learner.learner_config.ConstrainedIntValue]]])*
- *size (Union[pydantic.types.PositiveInt, Tuple[pydantic.types.PositiveInt, pydantic.types.PositiveInt]])*

- *size_lims (Optional[Tuple[pydantic.types.PositiveInt, pydantic.types.PositiveInt]])*
- *stride (Optional[Union[pydantic.types.PositiveInt, Tuple[pydantic.types.PositiveInt, pydantic.types.PositiveInt]]])*
- *type_hint (Literal['object_detection_geo_data_window'])*
- *w_lims (Optional[Tuple[pydantic.types.PositiveInt, pydantic.types.PositiveInt]])*

field clip: bool = False

Clip bounding boxes to window limits when retrieving labels for a window.

Validated by

- validate_options

field efficient_aoi_sampling: bool = True

If the scene has AOIs, sampling windows at random anywhere in the extent and then checking if they fall within any of the AOIs can be very inefficient. This flag enables the use of an alternate algorithm that only samples window locations inside the AOIs. Only used if method = random and the scene has aoi_polygons specified. Defaults to True

Validated by

- validate_options

field h_lims: Optional[Tuple[PosInt, PosInt]] = None

[min, max] interval from which window heights will be uniformly randomly sampled. Only used if method = random.

Validated by

- validate_options

field ioa_thresh: float = 0.8

When a box is partially outside of a training chip, it is not clear if (a clipped version) of the box should be included in the chip. If the IOA (intersection over area) of the box with the chip is greater than ioa_thresh, it is included in the chip. Defaults to 0.8.

Validated by

- validate_options

field max_sample_attempts: PosInt = 100

Max attempts when trying to find a window within the AOI of a scene. Only used if method = random and the scene has aoi_polygons specified.

Constraints

- **exclusiveMinimum** = 0

Validated by

- validate_options

field max_windows: NonNegInt = 10000

Max allowed reads from a GeoDataset. Only used if method = random.

Constraints

- **minimum** = 0

Validated by

- validate_options

field method: *GeoDataWindowMethod* = GeoDataWindowMethod.sliding

Validated by

- validate_options

field neg_ioa_thresh: float = 0.2

A window will be considered negative if its max IoA with any bounding box is less than this threshold. Defaults to 0.2.

Validated by

- validate_options

field neg_ratio: Optional[float] = None

The ratio of negative chips (those containing no bounding boxes) to positive chips. This can be useful if the statistics of the background is different in positive chips. For example, in car detection, the positive chips will always contain roads, but no examples of rooftops since cars tend to not be near rooftops. Defaults to None.

Validated by

- validate_options

field pad_direction: Literal['both', 'start', 'end'] = 'end'

If "end", only pad ymax and xmax (bottom and right). If "start", only pad ymin and xmin (top and left). If "both", pad all sides. Has no effect if paddiong is zero. Defaults to "end".

Validated by

- validate_options

field padding: Optional[Union[NonNegInt, Tuple[NonNegInt, NonNegInt]]] = None

How many pixels are windows allowed to overflow the edges of the raster source.

Validated by

- validate_options

field size: Union[PosInt, Tuple[PosInt, PosInt]] [Required]

If method = sliding, this is the size of sliding window. If method = random, this is the size that all the windows are resized to before they are returned. If method = random and neither size_lims nor h_lims and w_lims have been specified, then size_lims is set to (size, size + 1).

Validated by

- validate_options

field size_lims: Optional[Tuple[PosInt, PosInt]] = None

[min, max) interval from which window sizes will be uniformly randomly sampled. The upper limit is exclusive. To fix the size to a constant value, use size_lims = (sz, sz + 1). Only used if method = random. Specify either size_lims or h_lims and w_lims, but not both. If neither size_lims nor h_lims and w_lims have been specified, then this will be set to (size, size + 1).

Validated by

- validate_options

field stride: Optional[Union[PosInt, Tuple[PosInt, PosInt]]] = None

Stride of sliding window. Only used if method = sliding.

Validated by

- validate_options

field type_hint: Literal['object_detection_geo_data_window'] = 'object_detection_geo_data_window'

Validated by

- validate_options

field w_lims: Optional[Tuple[PosInt, PosInt]] = None

[min, max] interval from which window widths will be uniformly randomly sampled. Only used if method = random.

Validated by

- validate_options

build()

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(**args*, ***kwargs*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

validator validate_options » *all fields*

Parameters
values (*dict*) –

Return type
dict

ObjectDetectionImageDataConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model ObjectDetectionImageDataConfig

Configure *ObjectDetectionImageDatasets*.


```
{
   "title": "ObjectDetectionImageDataConfig",
   "description": "Configure :class:`ObjectDetectionImageDatasets <.
↪ObjectDetectionImageDataset>`.",
   "type": "object",
   "properties": {
      "class_names": {
         "title": "Class Names",
         "description": "Names of classes.",
         "default": [],
         "type": "array",
         "items": {
            "type": "string"
         }
      },
      "class_colors": {
         "title": "Class Colors",
         "description": "Colors used to display classes. Can be color 3-tuples in␣
↪list form.",
         "type": "array",
         "items": {
            "anyOf": [
               {
                  "type": "string"
               },
               {
                  "type": "array",
                  "minItems": 3,
                  "maxItems": 3,
                  "items": [
                     {
                        "type": "integer"
                     },
                     {
                        "type": "integer"
                     },
                     {
```


(continues on next page)

(continued from previous page)

```
                        "type": "integer"
                     }
                  ]
               }
            ]
         }
      },
      "img_channels": {
         "title": "Img Channels",
         "description": "The number of channels of the training images.",
         "exclusiveMinimum": 0,
         "type": "integer"
      },
      "img_sz": {
         "title": "Img Sz",
         "description": "Length of a side of each image in pixels. This is the size␣
↪to transform it to during training, not the size in the raw dataset.",
         "default": 256,
         "exclusiveMinimum": 0,
         "type": "integer"
      },
      "train_sz": {
         "title": "Train Sz",
         "description": "If set, the number of training images to use. If fewer␣
↪images exist, then an exception will be raised.",
         "type": "integer"
      },
      "train_sz_rel": {
         "title": "Train Sz Rel",
         "description": "If set, the proportion of training images to use.",
         "type": "number"
      },
      "num_workers": {
         "title": "Num Workers",
         "description": "Number of workers to use when DataLoader makes batches.",
         "default": 4,
         "type": "integer"
      },
      "augmentors": {
         "title": "Augmentors",
         "description": "Names of albumentations augmentors to use for training␣
↪batches. Choices include: ['Blur', 'RandomRotate90', 'HorizontalFlip',
↪'VerticalFlip', 'GaussianBlur', 'GaussNoise', 'RGBShift', 'ToGray'].␣
↪Alternatively, a custom transform can be provided via the aug_transform option.",
         "default": [
            "RandomRotate90",
            "HorizontalFlip",
            "VerticalFlip"
         ],
         "type": "array",
         "items": {
            "type": "string"
```

(continues on next page)

(continued from previous page)

```
        }
     },
     "base_transform": {
        "title": "Base Transform",
        "description": "An Albumentations transform serialized as a dict that will␣
↪be applied to all datasets: training, validation, and test. This transformation␣
↪is in addition to the resizing due to img_sz. This is useful for, for example,␣
↪applying the same normalization to all datasets.",
        "type": "object"
     },
     "aug_transform": {
        "title": "Aug Transform",
        "description": "An Albumentations transform serialized as a dict that will␣
↪be applied as data augmentation to the training dataset. This transform is␣
↪applied before base_transform. If provided, the augmentors option is ignored.",
        "type": "object"
     },
     "plot_options": {
        "title": "Plot Options",
        "description": "Options to control plotting.",
        "default": {
           "transform": {
              "__version__": "1.3.0",
              "transform": {
                 "__class_fullname__": "rastervision.pytorch_learner.utils.utils.
↪MinMaxNormalize",
                 "always_apply": false,
                 "p": 1.0,
                 "min_val": 0.0,
                 "max_val": 1.0,
                 "dtype": 5
              }
           },
           "channel_display_groups": null,
           "type_hint": "plot_options"
        },
        "allOf": [
           {
              "$ref": "#/definitions/PlotOptions"
           }
        ]
     },
     "preview_batch_limit": {
        "title": "Preview Batch Limit",
        "description": "Optional limit on the number of items in the preview plots␣
↪produced during training.",
        "type": "integer"
     },
     "type_hint": {
        "title": "Type Hint",
        "default": "object_detection_image_data",
        "enum": [
```

(continues on next page)

(continued from previous page)


```
            "object_detection_image_data"
         ],
         "type": "string"
      },
      "data_format": {
         "default": "coco",
         "allOf": [
            {
               "$ref": "#/definitions/ObjectDetectionDataFormat"
            }
         ]
      },
      "uri": {
         "title": "Uri",
         "description": "One of the following:\n(1) a URI of a directory containing␣
↪\"train\", \"valid\", and (optinally) \"test\" subdirectories;\n(2) a URI of a␣
↪zip file containing (1);\n(3) a list of (2);\n(4) a URI of a directory containing␣
↪zip files containing (1).",
         "anyOf": [
            {
               "type": "string"
            },
            {
               "type": "array",
               "items": {
                  "type": "string"
               }
            }
         ]
      },
      "group_uris": {
         "title": "Group Uris",
         "description": "This can be set instead of uri in order to specify groups␣
↪of chips. Each element in the list is expected to be an object of the same form␣
↪accepted by the uri field. The purpose of separating chips into groups is to be␣
↪able to use the group_train_sz field.",
         "type": "array",
         "items": {
            "anyOf": [
               {
                  "type": "string"
               },
               {
                  "type": "array",
                  "items": {
                     "type": "string"
                  }
               }
            ]
         }
      },
      "group_train_sz": {
```


(continues on next page)

(continued from previous page)

```
        "title": "Group Train Sz",
        "description": "If group_uris is set, this can be used to specify the␣
↪number of chips to use per group. Only applies to training chips. This can either␣
↪be a single value that will be used for all groups or a list of values (one for␣
↪each group).",
        "anyOf": [
           {
              "type": "integer"
           },
           {
              "type": "array",
              "items": {
                 "type": "integer"
              }
           }
        ]
     },
     "group_train_sz_rel": {
        "title": "Group Train Sz Rel",
        "description": "Relative version of group_train_sz. Must be a float in [0,␣
↪1]. If group_uris is set, this can be used to specify the proportion of the total␣
↪chips in each group to use per group. Only applies to training chips. This can␣
↪either be a single value that will be used for all groups or a list of values␣
↪(one for each group).",
        "anyOf": [
           {
              "type": "number",
              "minimum": 0,
              "maximum": 1
           },
           {
              "type": "array",
              "items": {
                 "type": "number",
                 "minimum": 0,
                 "maximum": 1
              }
           }
        ]
     }
  },
  "additionalProperties": false,
  "definitions": {
     "PlotOptions": {
        "title": "PlotOptions",
        "description": "Config related to plotting.",
        "type": "object",
        "properties": {
           "transform": {
              "title": "Transform",
              "description": "An Albumentations transform serialized as a dict␣
↪that will be applied to each image before it is plotted. Mainly useful for␣
```

(continues on next page)

(continued from previous page)


```
↪undoing any data transformation that you do not want included in the plot, such␣
↪as normalization. The default value will shift and scale the image so the values␣
↪range from 0.0 to 1.0 which is the expected range for the plotting function. This␣
↪default is useful for cases where the values after normalization are close to␣
↪zero which makes the plot difficult to see.",
                "default": {
                   "__version__": "1.3.0",
                   "transform": {
                      "__class_fullname__": "rastervision.pytorch_learner.utils.
↪utils.MinMaxNormalize",
                      "always_apply": false,
                      "p": 1.0,
                      "min_val": 0.0,
                      "max_val": 1.0,
                      "dtype": 5
                   }
                },
                "type": "object"
             },
             "channel_display_groups": {
                "title": "Channel Display Groups",
                "description": "Groups of image channels to display together as a␣
↪subplot when plotting the data and predictions. Can be a list or tuple of groups␣
↪(e.g. [(0, 1, 2), (3,)]) or a dict containing title-to-group mappings (e.g. {\
↪"RGB\": [0, 1, 2], \"IR\": [3]}), where each group is a list or tuple of channel␣
↪indices and title is a string that will be used as the title of the subplot for␣
↪that group.",
                "anyOf": [
                   {
                      "type": "object",
                      "additionalProperties": {
                         "type": "array",
                         "items": {
                            "type": "integer",
                            "minimum": 0
                         }
                      }
                   },
                   {
                      "type": "array",
                      "items": {
                         "type": "array",
                         "items": {
                            "type": "integer",
                            "minimum": 0
                         }
                      }
                   }
                ]
             },
             "type_hint": {
                "title": "Type Hint",
```


(continues on next page)

(continued from previous page)


```
                "default": "plot_options",
                "enum": [
                    "plot_options"
                ],
                "type": "string"
            }
        },
        "additionalProperties": false
    },
    "ObjectDetectionDataFormat": {
        "title": "ObjectDetectionDataFormat",
        "description": "An enumeration.",
        "enum": [
            "coco"
        ]
    }
  }
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *aug_transform (Optional[dict])*
- *augmentors (List[str])*
- *base_transform (Optional[dict])*
- *class_colors (Optional[List[Union[str, Tuple[int, int, int]]]])*
- *class_names (List[str])*
- *data_format (rastervision.pytorch_learner.object_detection_learner_config.ObjectDetectionDataFormat)*
- *group_train_sz (Optional[Union[int, List[int]]])*
- *group_train_sz_rel (Optional[Union[rastervision.pytorch_learner.learner_config.ConstrainedFloatValue, List[rastervision.pytorch_learner.learner_config.ConstrainedFloatValue]]])*
- *group_uris (Optional[List[Union[str, List[str]]]])*
- *img_channels (Optional[pydantic.types.PositiveInt])*
- *img_sz (pydantic.types.PositiveInt)*
- *num_workers (int)*
- *plot_options (Optional[rastervision.pytorch_learner.learner_config.PlotOptions])*
- *preview_batch_limit (Optional[int])*
- *train_sz (Optional[int])*
- *train_sz_rel (Optional[float])*

- *type_hint (Literal['object_detection_image_data'])*
- *uri (Optional[Union[str, List[str]]])*

Validators

- ensure_class_colors » all fields
- *validate_albumentation_transform* » *aug_transform*
- *validate_albumentation_transform* » *base_transform*
- validate_augmentors » *augmentors*
- validate_group_uris » all fields
- validate_plot_options » all fields

field aug_transform: Optional[dict] = None

An Albumentations transform serialized as a dict that will be applied as data augmentation to the training dataset. This transform is applied before base_transform. If provided, the augmentors option is ignored.

Validated by

- ensure_class_colors
- *validate_albumentation_transform*
- validate_group_uris
- validate_plot_options

field augmentors: List[str] = ['RandomRotate90', 'HorizontalFlip', 'VerticalFlip']

Names of albumentations augmentors to use for training batches. Choices include: [‘Blur’, ‘RandomRotate90’, ‘HorizontalFlip’, ‘VerticalFlip’, ‘GaussianBlur’, ‘GaussNoise’, ‘RGBShift’, ‘ToGray’]. Alternatively, a custom transform can be provided via the aug_transform option.

Validated by

- ensure_class_colors
- validate_augmentors
- validate_group_uris
- validate_plot_options

field base_transform: Optional[dict] = None

An Albumentations transform serialized as a dict that will be applied to all datasets: training, validation, and test. This transformation is in addition to the resizing due to img_sz. This is useful for, for example, applying the same normalization to all datasets.

Validated by

- ensure_class_colors
- *validate_albumentation_transform*
- validate_group_uris
- validate_plot_options

field class_colors: Optional[List[Union[str, RGBTuple]]] = None

Colors used to display classes. Can be color 3-tuples in list form.

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field class_names: List[str] = []

Names of classes.

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field data_format: *ObjectDetectionDataFormat* = ObjectDetectionDataFormat.coco

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field group_train_sz: Optional[Union[int, List[int]]] = None

If group_uris is set, this can be used to specify the number of chips to use per group. Only applies to training chips. This can either be a single value that will be used for all groups or a list of values (one for each group).

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field group_train_sz_rel: Optional[Union[Proportion, List[Proportion]]] = None

Relative version of group_train_sz. Must be a float in [0, 1]. If group_uris is set, this can be used to specify the proportion of the total chips in each group to use per group. Only applies to training chips. This can either be a single value that will be used for all groups or a list of values (one for each group).

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field group_uris: Optional[List[Union[str, List[str]]]] = None

This can be set instead of uri in order to specify groups of chips. Each element in the list is expected to be an object of the same form accepted by the uri field. The purpose of separating chips into groups is to be able to use the group_train_sz field.

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field img_channels: Optional[PosInt] = None

The number of channels of the training images.

Constraints

- **exclusiveMinimum** = 0

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field img_sz: PosInt = 256

Length of a side of each image in pixels. This is the size to transform it to during training, not the size in the raw dataset.

Constraints

- **exclusiveMinimum** = 0

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field num_workers: int = 4

Number of workers to use when DataLoader makes batches.

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field plot_options: Optional[*PlotOptions*] = PlotOptions(transform={'__version__': '1.3.0', 'transform': {'__class_fullname__': 'rastervision.pytorch_learner.utils.utils.MinMaxNormalize', 'always_apply': False, 'p': 1.0, 'min_val': 0.0, 'max_val': 1.0, 'dtype': 5}}, channel_display_groups=None)

Options to control plotting.

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field preview_batch_limit: Optional[int] = None

Optional limit on the number of items in the preview plots produced during training.

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field train_sz: Optional[int] = None

If set, the number of training images to use. If fewer images exist, then an exception will be raised.

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field train_sz_rel: Optional[float] = None

If set, the proportion of training images to use.

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field type_hint: Literal['object_detection_image_data'] = 'object_detection_image_data'

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field uri: Optional[Union[str, List[str]]] = None

One of the following: (1) a URI of a directory containing "train", "valid", and (optinally) "test" subdirectories; (2) a URI of a zip file containing (1); (3) a list of (2); (4) a URI of a directory containing zip files containing (1).

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

build(*tmp_dir: str*, *overfit_mode: bool = False*, *test_mode: bool = False*) → Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

Parameters

- **tmp_dir** (*str*) –
- **overfit_mode** (*bool*) –
- **test_mode** (*bool*) –

Return type

Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

dir_to_dataset(*data_dir: str*, *transform: BasicTransform*) → *ObjectDetectionImageDataset*

Parameters

- **data_dir** (*str*) –
- **transform** (*BasicTransform*) –

Return type
ObjectDetectionImageDataset

validator ensure_class_colors » *all fields*

Parameters
values (*dict*) –

Return type
dict

get_bbox_params()

Returns BboxParams used by albumentations for data augmentation.

get_custom_albumentations_transforms() → List[dict]

Returns all custom transforms found in this config.

This should return all serialized albumentations transforms with a 'lambda_transforms_path' field contained in this config or in any of its members no matter how deeply neseted.

The pupose is to make it easier to adjust their paths all at once while saving to or loading from a bundle.

Return type
List[dict]

get_data_dirs(*uri: Union[str, List[str]]*, *unzip_dir: str*) → List[str]

Extract data dirs from uri.

Data dirs are directories containing "train", "valid", and (optinally) "test" subdirectories.

Parameters

- **uri** (*Union[str, List[str]]*) – a URI or a list of URIs of one of the following:

(1) a URI of a directory containing "train", "valid", and (optinally) "test" subdirectories

(2) a URI of a zip file containing (1)

(3) a list of (2)

(4) a URI of a directory containing zip files containing (1)

- **unzip_dir** (*str*) –

Returns
paths to directories that each contain contents of one zip file

Return type
List[str]

get_data_transforms() → Tuple[BasicTransform, BasicTransform]

Get albumentations transform objects for data augmentation.

Returns
a transform that doesn't do any data augmentation 2nd tuple arg: a transform with data augmentation

Return type
1st tuple arg

get_datasets_from_group_uris(*uris: Union[str, List[str]]*, *tmp_dir: str*, *group_train_sz: Optional[int] = None*, *group_train_sz_rel: Optional[float] = None*, *overfit_mode: bool = False*, *test_mode: bool = False*) → Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

Parameters

- **uris** (*Union[str, List[str]]*) –
- **tmp_dir** (*str*) –
- **group_train_sz** (*Optional[int]*) –
- **group_train_sz_rel** (*Optional[float]*) –
- **overfit_mode** (*bool*) –
- **test_mode** (*bool*) –

Return type
Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

get_datasets_from_uri(*uri: Union[str, List[str]]*, *tmp_dir: str*, *overfit_mode: bool = False*, *test_mode: bool = False*) → Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

Get image train, validation, & test datasets from a single zip file.

Parameters

- **uri** (*Union[str, List[str]]*) – Uri of a zip file containing the images.
- **tmp_dir** (*str*) –
- **overfit_mode** (*bool*) –
- **test_mode** (*bool*) –

Returns

Training, validation, and test
dataSets.

Return type
Tuple[Dataset, Dataset, Dataset]

make_datasets(*train_dirs: Iterable[str]*, *val_dirs: Iterable[str]*, *test_dirs: Iterable[str]*, *train_tf: Optional[BasicTransform] = None*, *val_tf: Optional[BasicTransform] = None*, *test_tf: Optional[BasicTransform] = None*) → Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

Make training, validation, and test datasets.

Parameters

- **train_dirs** (*str*) – Directories where training data is located.
- **val_dirs** (*str*) – Directories where validation data is located.
- **test_dirs** (*str*) – Directories where test data is located.
- **train_tf** (*Optional[A.BasicTransform], optional*) – Transform for the training dataset. Defaults to None.

- **val_tf** (*Optional[A.BasicTransform], optional*) – Transform for the validation dataset. Defaults to None.
- **test_tf** (*Optional[A.BasicTransform], optional*) – Transform for the test dataset. Defaults to None.

Returns

PyTorch-compatiable training,
validation, and test datasets.

Return type
Tuple[Dataset, Dataset, Dataset]

random_subset_dataset(*ds: torch.utils.data.Dataset*, *size: Optional[int] = None*, *fraction: Optional[ConstrainedFloatValue] = None*) → torch.utils.data.Subset

Parameters

- **ds** (*torch.utils.data.Dataset*) –
- **size** (*Optional[int]*) –
- **fraction** (*Optional[ConstrainedFloatValue]*) –

Return type
torch.utils.data.Subset

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

unzip_data(*zip_uris: List[str]*, *unzip_dir: str*) → List[str]

Unzip dataset zip files.

Parameters

- **zip_uris** (*List[str]*) – a list of URIs of zip files:
- **unzip_dir** (*str*) – directory where zip files will be extrated to.

Returns
paths to directories that each contain contents of one zip file

Return type
List[str]

update(**args*, ***kwargs*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validator validate_augmentors » augmentors

Parameters
v (*str*) –

Return type
str

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validator validate_group_uris » *all fields*

Parameters
values (*dict*) –

Return type
dict

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises
ConfigError – if field is invalid

validator validate_plot_options » *all fields*

Parameters
values (*dict*) –

Return type
dict

property num_classes

ObjectDetectionLearnerConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model ObjectDetectionLearnerConfig

Configure an *ObjectDetectionLearner*.


```
{
   "title": "ObjectDetectionLearnerConfig",
   "description": "Configure an :class:`.ObjectDetectionLearner`.",
   "type": "object",
   "properties": {
      "model": {
         "$ref": "#/definitions/ObjectDetectionModelConfig"
```


(continues on next page)

(continued from previous page)


```
      },
      "solver": {
         "$ref": "#/definitions/SolverConfig"
      },
      "data": {
         "title": "Data",
         "anyOf": [
            {
               "$ref": "#/definitions/ObjectDetectionImageDataConfig"
            },
            {
               "$ref": "#/definitions/ObjectDetectionGeoDataConfig"
            }
         ]
      },
      "predict_mode": {
         "title": "Predict Mode",
         "description": "If True, skips training, loads model, and does final eval.
↪",
         "default": false,
         "type": "boolean"
      },
      "test_mode": {
         "title": "Test Mode",
         "description": "If True, uses test_num_epochs, test_batch_sz, truncated␣
↪datasets with only a single batch, image_sz that is cut in half, and num_workers␣
↪= 0. This is useful for testing that code runs correctly on CPU without␣
↪multithreading before running full job on GPU.",
         "default": false,
         "type": "boolean"
      },
      "overfit_mode": {
         "title": "Overfit Mode",
         "description": "If True, uses half image size, and instead of doing epoch-
↪based training, optimizes the model using a single batch repeatedly for overfit_
↪num_steps number of steps.",
         "default": false,
         "type": "boolean"
      },
      "eval_train": {
         "title": "Eval Train",
         "description": "If True, runs final evaluation on training set (in␣
↪addition to test set). Useful for debugging.",
         "default": false,
         "type": "boolean"
      },
      "save_model_bundle": {
         "title": "Save Model Bundle",
         "description": "If True, saves a model bundle at the end of training which␣
↪is zip file with model and this LearnerConfig which can be used to make␣
↪predictions on new images at a later time.",
         "default": true,
```


(continues on next page)

(continued from previous page)


```
            "type": "boolean"
        },
        "log_tensorboard": {
            "title": "Log Tensorboard",
            "description": "Save Tensorboard log files at the end of each epoch.",
            "default": true,
            "type": "boolean"
        },
        "run_tensorboard": {
            "title": "Run Tensorboard",
            "description": "run Tensorboard server during training",
            "default": false,
            "type": "boolean"
        },
        "output_uri": {
            "title": "Output Uri",
            "description": "URI of where to save output",
            "type": "string"
        },
        "type_hint": {
            "title": "Type Hint",
            "default": "object_detection_learner",
            "enum": [
                "object_detection_learner"
            ],
            "type": "string"
        }
    },
    "required": [
        "solver",
        "data"
    ],
    "additionalProperties": false,
    "definitions": {
        "Backbone": {
            "title": "Backbone",
            "description": "An enumeration.",
            "enum": [
                "alexnet",
                "densenet121",
                "densenet169",
                "densenet201",
                "densenet161",
                "googlenet",
                "inception_v3",
                "mnasnet0_5",
                "mnasnet0_75",
                "mnasnet1_0",
                "mnasnet1_3",
                "mobilenet_v2",
                "resnet18",
                "resnet34",
```


(continues on next page)

(continued from previous page)

```
            "resnet50",
            "resnet101",
            "resnet152",
            "resnext50_32x4d",
            "resnext101_32x8d",
            "wide_resnet50_2",
            "wide_resnet101_2",
            "shufflenet_v2_x0_5",
            "shufflenet_v2_x1_0",
            "shufflenet_v2_x1_5",
            "shufflenet_v2_x2_0",
            "squeezenet1_0",
            "squeezenet1_1",
            "vgg11",
            "vgg11_bn",
            "vgg13",
            "vgg13_bn",
            "vgg16",
            "vgg16_bn",
            "vgg19_bn",
            "vgg19"
         ]
      },
      "ExternalModuleConfig": {
         "title": "ExternalModuleConfig",
         "description": "Config describing an object to be loaded via Torch Hub.",
         "type": "object",
         "properties": {
            "uri": {
               "title": "Uri",
               "description": "Local uri of a zip file, or local uri of a directory,
↪or remote uri of zip file.",
               "minLength": 1,
               "type": "string"
            },
            "github_repo": {
               "title": "Github Repo",
               "description": "<repo-owner>/<repo-name>[:tag]",
               "pattern": ".+/.+",
               "type": "string"
            },
            "name": {
               "title": "Name",
               "description": "Name of the folder in which to extract/copy the␣
↪definition files.",
               "minLength": 1,
               "type": "string"
            },
            "entrypoint": {
               "title": "Entrypoint",
               "description": "Name of a callable present in hubconf.py. See docs␣
↪for torch.hub for details.",
```

(continues on next page)

(continued from previous page)


```
                "minLength": 1,
                "type": "string"
            },
            "entrypoint_args": {
                "title": "Entrypoint Args",
                "description": "Args to pass to the entrypoint. Must be serializable.
↪",
                "default": [],
                "type": "array",
                "items": {}
            },
            "entrypoint_kwargs": {
                "title": "Entrypoint Kwargs",
                "description": "Keyword args to pass to the entrypoint. Must be␣
↪serializable.",
                "default": {},
                "type": "object"
            },
            "force_reload": {
                "title": "Force Reload",
                "description": "Force reload of module definition.",
                "default": false,
                "type": "boolean"
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "external-module",
                "enum": [
                    "external-module"
                ],
                "type": "string"
            }
        },
        "required": [
            "entrypoint"
        ],
        "additionalProperties": false
    },
    "ObjectDetectionModelConfig": {
        "title": "ObjectDetectionModelConfig",
        "description": "Configure an object detection model.",
        "type": "object",
        "properties": {
            "backbone": {
                "description": "The torchvision.models backbone to use, which must␣
↪be in the resnet* family.",
                "default": "resnet50",
                "allOf": [
                    {
                        "$ref": "#/definitions/Backbone"
                    }
                ]
```


(continues on next page)

(continued from previous page)


```
            },
            "pretrained": {
               "title": "Pretrained",
               "description": "If True, use ImageNet weights. If False, use random␣
↪initialization.",
               "default": true,
               "type": "boolean"
            },
            "init_weights": {
               "title": "Init Weights",
               "description": "URI of PyTorch model weights used to initialize␣
↪model. If set, this supercedes the pretrained option.",
               "type": "string"
            },
            "load_strict": {
               "title": "Load Strict",
               "description": "If True, the keys in the state dict referenced by␣
↪init_weights must match exactly. Setting this to False can be useful if you just␣
↪want to load the backbone of a model.",
               "default": true,
               "type": "boolean"
            },
            "external_def": {
               "title": "External Def",
               "description": "If specified, the model will be built from the␣
↪definition from this external source, using Torch Hub.",
               "allOf": [
                  {
                     "$ref": "#/definitions/ExternalModuleConfig"
                  }
               ]
            },
            "type_hint": {
               "title": "Type Hint",
               "default": "object_detection_model",
               "enum": [
                  "object_detection_model"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      },
      "SolverConfig": {
         "title": "SolverConfig",
         "description": "Config related to solver aka optimizer.",
         "type": "object",
         "properties": {
            "lr": {
               "title": "Lr",
               "description": "Learning rate.",
               "default": 0.0001,
```


(continues on next page)

(continued from previous page)


```
                "exclusiveMinimum": 0,
                "type": "number"
            },
            "num_epochs": {
                "title": "Num Epochs",
                "description": "Number of epochs (ie. sweeps through the whole␣
↪training set).",
                "default": 10,
                "exclusiveMinimum": 0,
                "type": "integer"
            },
            "test_num_epochs": {
                "title": "Test Num Epochs",
                "description": "Number of epochs to use in test mode.",
                "default": 2,
                "exclusiveMinimum": 0,
                "type": "integer"
            },
            "test_batch_sz": {
                "title": "Test Batch Sz",
                "description": "Batch size to use in test mode.",
                "default": 4,
                "exclusiveMinimum": 0,
                "type": "integer"
            },
            "overfit_num_steps": {
                "title": "Overfit Num Steps",
                "description": "Number of optimizer steps to use in overfit mode.",
                "default": 1,
                "exclusiveMinimum": 0,
                "type": "integer"
            },
            "sync_interval": {
                "title": "Sync Interval",
                "description": "The interval in epochs for each sync to the cloud.",
                "default": 1,
                "exclusiveMinimum": 0,
                "type": "integer"
            },
            "batch_sz": {
                "title": "Batch Sz",
                "description": "Batch size.",
                "default": 32,
                "exclusiveMinimum": 0,
                "type": "integer"
            },
            "one_cycle": {
                "title": "One Cycle",
                "description": "If True, use triangular LR scheduler with a single␣
↪cycle across all epochs with start and end LR being lr/10 and the peak being lr.",
                "default": true,
                "type": "boolean"
```


(continues on next page)

(continued from previous page)

```
            },
            "multi_stage": {
               "title": "Multi Stage",
               "description": "List of epoch indices at which to divide LR by 10.",
               "default": [],
               "type": "array",
               "items": {}
            },
            "class_loss_weights": {
               "title": "Class Loss Weights",
               "description": "Class weights for weighted loss.",
               "type": "array",
               "items": {
                  "type": "number"
               }
            },
            "ignore_class_index": {
               "title": "Ignore Class Index",
               "description": "If specified, this index is ignored when computing␣
↪the loss. See pytorch documentation for nn.CrossEntropyLoss for more details.␣
↪This can also be negative, in which case it is treated as a negative slice index␣
↪i.e. -1 = last index, -2 = second-last index, and so on.",
               "type": "integer"
            },
            "external_loss_def": {
               "title": "External Loss Def",
               "description": "If specified, the loss will be built from the␣
↪definition from this external source, using Torch Hub.",
               "allOf": [
                  {
                     "$ref": "#/definitions/ExternalModuleConfig"
                  }
               ]
            },
            "type_hint": {
               "title": "Type Hint",
               "default": "solver",
               "enum": [
                  "solver"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      },
      "PlotOptions": {
         "title": "PlotOptions",
         "description": "Config related to plotting.",
         "type": "object",
         "properties": {
            "transform": {
               "title": "Transform",
```

(continues on next page)

(continued from previous page)

```
                "description": "An Albumentations transform serialized as a dict␣
↪that will be applied to each image before it is plotted. Mainly useful for␣
↪undoing any data transformation that you do not want included in the plot, such␣
↪as normalization. The default value will shift and scale the image so the values␣
↪range from 0.0 to 1.0 which is the expected range for the plotting function. This␣
↪default is useful for cases where the values after normalization are close to␣
↪zero which makes the plot difficult to see.",
                "default": {
                   "__version__": "1.3.0",
                   "transform": {
                      "__class_fullname__": "rastervision.pytorch_learner.utils.
↪utils.MinMaxNormalize",
                      "always_apply": false,
                      "p": 1.0,
                      "min_val": 0.0,
                      "max_val": 1.0,
                      "dtype": 5
                   }
                },
                "type": "object"
             },
             "channel_display_groups": {
                "title": "Channel Display Groups",
                "description": "Groups of image channels to display together as a␣
↪subplot when plotting the data and predictions. Can be a list or tuple of groups␣
↪(e.g. [(0, 1, 2), (3,)]) or a dict containing title-to-group mappings (e.g. {\
↪"RGB\": [0, 1, 2], \"IR\": [3]}), where each group is a list or tuple of channel␣
↪indices and title is a string that will be used as the title of the subplot for␣
↪that group.",
                "anyOf": [
                   {
                      "type": "object",
                      "additionalProperties": {
                         "type": "array",
                         "items": {
                            "type": "integer",
                            "minimum": 0
                         }
                      }
                   },
                   {
                      "type": "array",
                      "items": {
                         "type": "array",
                         "items": {
                            "type": "integer",
                            "minimum": 0
                         }
                      }
                   }
                ]
             },
```

(continues on next page)

(continued from previous page)


```
            "type_hint": {
               "title": "Type Hint",
               "default": "plot_options",
               "enum": [
                  "plot_options"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      },
      "ObjectDetectionDataFormat": {
         "title": "ObjectDetectionDataFormat",
         "description": "An enumeration.",
         "enum": [
            "coco"
         ]
      },
      "ObjectDetectionImageDataConfig": {
         "title": "ObjectDetectionImageDataConfig",
         "description": "Configure :class:`ObjectDetectionImageDatasets <.
↪ObjectDetectionImageDataset>`.",
         "type": "object",
         "properties": {
            "class_names": {
               "title": "Class Names",
               "description": "Names of classes.",
               "default": [],
               "type": "array",
               "items": {
                  "type": "string"
               }
            },
            "class_colors": {
               "title": "Class Colors",
               "description": "Colors used to display classes. Can be color 3-
↪tuples in list form.",
               "type": "array",
               "items": {
                  "anyOf": [
                     {
                        "type": "string"
                     },
                     {
                        "type": "array",
                        "minItems": 3,
                        "maxItems": 3,
                        "items": [
                           {
                              "type": "integer"
                           },
                           {
```


(continues on next page)

(continued from previous page)

```
                                "type": "integer"
                            },
                            {
                                "type": "integer"
                            }
                        ]
                    }
                ]
            }
        },
        "img_channels": {
            "title": "Img Channels",
            "description": "The number of channels of the training images.",
            "exclusiveMinimum": 0,
            "type": "integer"
        },
        "img_sz": {
            "title": "Img Sz",
            "description": "Length of a side of each image in pixels. This is␣
↪the size to transform it to during training, not the size in the raw dataset.",
            "default": 256,
            "exclusiveMinimum": 0,
            "type": "integer"
        },
        "train_sz": {
            "title": "Train Sz",
            "description": "If set, the number of training images to use. If␣
↪fewer images exist, then an exception will be raised.",
            "type": "integer"
        },
        "train_sz_rel": {
            "title": "Train Sz Rel",
            "description": "If set, the proportion of training images to use.",
            "type": "number"
        },
        "num_workers": {
            "title": "Num Workers",
            "description": "Number of workers to use when DataLoader makes␣
↪batches.",
            "default": 4,
            "type": "integer"
        },
        "augmentors": {
            "title": "Augmentors",
            "description": "Names of albumentations augmentors to use for␣
↪training batches. Choices include: ['Blur', 'RandomRotate90', 'HorizontalFlip',
↪'VerticalFlip', 'GaussianBlur', 'GaussNoise', 'RGBShift', 'ToGray'].␣
↪Alternatively, a custom transform can be provided via the aug_transform option.",
            "default": [
                "RandomRotate90",
                "HorizontalFlip",
                "VerticalFlip"
```

(continues on next page)

(continued from previous page)


```
                ],
                "type": "array",
                "items": {
                    "type": "string"
                }
            },
            "base_transform": {
                "title": "Base Transform",
                "description": "An Albumentations transform serialized as a dict␣
↪that will be applied to all datasets: training, validation, and test. This␣
↪transformation is in addition to the resizing due to img_sz. This is useful for,␣
↪for example, applying the same normalization to all datasets.",
                "type": "object"
            },
            "aug_transform": {
                "title": "Aug Transform",
                "description": "An Albumentations transform serialized as a dict␣
↪that will be applied as data augmentation to the training dataset. This transform␣
↪is applied before base_transform. If provided, the augmentors option is ignored.",
                "type": "object"
            },
            "plot_options": {
                "title": "Plot Options",
                "description": "Options to control plotting.",
                "default": {
                    "transform": {
                        "__version__": "1.3.0",
                        "transform": {
                            "__class_fullname__": "rastervision.pytorch_learner.utils.
↪utils.MinMaxNormalize",
                            "always_apply": false,
                            "p": 1.0,
                            "min_val": 0.0,
                            "max_val": 1.0,
                            "dtype": 5
                        }
                    },
                    "channel_display_groups": null,
                    "type_hint": "plot_options"
                },
                "allOf": [
                    {
                        "$ref": "#/definitions/PlotOptions"
                    }
                ]
            },
            "preview_batch_limit": {
                "title": "Preview Batch Limit",
                "description": "Optional limit on the number of items in the preview␣
↪plots produced during training.",
                "type": "integer"
            },
```


(continues on next page)

(continued from previous page)

```
            "type_hint": {
               "title": "Type Hint",
               "default": "object_detection_image_data",
               "enum": [
                  "object_detection_image_data"
               ],
               "type": "string"
            },
            "data_format": {
               "default": "coco",
               "allOf": [
                  {
                     "$ref": "#/definitions/ObjectDetectionDataFormat"
                  }
               ]
            },
            "uri": {
               "title": "Uri",
               "description": "One of the following:\n(1) a URI of a directory␣
↪containing \"train\", \"valid\", and (optinally) \"test\" subdirectories;\n(2) a␣
↪URI of a zip file containing (1);\n(3) a list of (2);\n(4) a URI of a directory␣
↪containing zip files containing (1).",
               "anyOf": [
                  {
                     "type": "string"
                  },
                  {
                     "type": "array",
                     "items": {
                        "type": "string"
                     }
                  }
               ]
            },
            "group_uris": {
               "title": "Group Uris",
               "description": "This can be set instead of uri in order to specify␣
↪groups of chips. Each element in the list is expected to be an object of the same␣
↪form accepted by the uri field. The purpose of separating chips into groups is to␣
↪be able to use the group_train_sz field.",
               "type": "array",
               "items": {
                  "anyOf": [
                     {
                        "type": "string"
                     },
                     {
                        "type": "array",
                        "items": {
                           "type": "string"
                        }
                     }
```

(continues on next page)

(continued from previous page)


```
                ]
             }
          },
          "group_train_sz": {
             "title": "Group Train Sz",
             "description": "If group_uris is set, this can be used to specify␣
↪the number of chips to use per group. Only applies to training chips. This can␣
↪either be a single value that will be used for all groups or a list of values␣
↪(one for each group).",
             "anyOf": [
                {
                   "type": "integer"
                },
                {
                   "type": "array",
                   "items": {
                      "type": "integer"
                   }
                }
             ]
          },
          "group_train_sz_rel": {
             "title": "Group Train Sz Rel",
             "description": "Relative version of group_train_sz. Must be a float␣
↪in [0, 1]. If group_uris is set, this can be used to specify the proportion of␣
↪the total chips in each group to use per group. Only applies to training chips.␣
↪This can either be a single value that will be used for all groups or a list of␣
↪values (one for each group).",
             "anyOf": [
                {
                   "type": "number",
                   "minimum": 0,
                   "maximum": 1
                },
                {
                   "type": "array",
                   "items": {
                      "type": "number",
                      "minimum": 0,
                      "maximum": 1
                   }
                }
             ]
          }
       },
       "additionalProperties": false
    },
    "ClassConfig": {
       "title": "ClassConfig",
       "description": "Configure class information for a machine learning task.",
       "type": "object",
       "properties": {
```


(continues on next page)

(continued from previous page)

```
            "names": {
               "title": "Names",
               "description": "Names of classes. The i-th class in this list will␣
↪have class ID = i.",
               "type": "array",
               "items": {
                  "type": "string"
               }
            },
            "colors": {
               "title": "Colors",
               "description": "Colors used to visualize classes. Can be color␣
↪strings accepted by matplotlib or RGB tuples. If None, a random color will be␣
↪auto-generated for each class.",
               "type": "array",
               "items": {
                  "anyOf": [
                     {
                        "type": "string"
                     },
                     {
                        "type": "array",
                        "items": {}
                     }
                  ]
               }
            },
            "null_class": {
               "title": "Null Class",
               "description": "Optional name of class in `names` to use as the null␣
↪class. This is used in semantic segmentation to represent the label for imagery␣
↪pixels that are NODATA or that are missing a label. If None and the class names␣
↪include \"null\", it will automatically be used as the null class. If None, and␣
↪this Config is part of a SemanticSegmentationConfig, a null class will be added␣
↪automatically.",
               "type": "string"
            },
            "type_hint": {
               "title": "Type Hint",
               "default": "class_config",
               "enum": [
                  "class_config"
               ],
               "type": "string"
            }
         },
         "required": [
            "names"
         ],
         "additionalProperties": false
      },
      "RasterTransformerConfig": {
```

(continues on next page)

(continued from previous page)


```
        "title": "RasterTransformerConfig",
        "description": "Configure a :class:`.RasterTransformer`.",
        "type": "object",
        "properties": {
           "type_hint": {
              "title": "Type Hint",
              "default": "raster_transformer",
              "enum": [
                 "raster_transformer"
              ],
              "type": "string"
           }
        },
        "additionalProperties": false
     },
     "RasterSourceConfig": {
        "title": "RasterSourceConfig",
        "description": "Configure a :class:`.RasterSource`.",
        "type": "object",
        "properties": {
           "channel_order": {
              "title": "Channel Order",
              "description": "The sequence of channel indices to use when reading␣
↪imagery.",
              "type": "array",
              "items": {
                 "type": "integer"
              }
           },
           "transformers": {
              "title": "Transformers",
              "default": [],
              "type": "array",
              "items": {
                 "$ref": "#/definitions/RasterTransformerConfig"
              }
           },
           "extent": {
              "title": "Extent",
              "description": "Use-specified extent in pixel coords in the form␣
↪(ymin, xmin, ymax, xmax). Useful for cropping the raster source so that only part␣
↪of the raster is read from.",
              "type": "array",
              "minItems": 4,
              "maxItems": 4,
              "items": [
                 {
                    "type": "integer"
                 },
                 {
                    "type": "integer"
                 },
```


(continues on next page)

(continued from previous page)


```
                {
                    "type": "integer"
                },
                {
                    "type": "integer"
                }
            ]
        },
        "type_hint": {
            "title": "Type Hint",
            "default": "raster_source",
            "enum": [
                "raster_source"
            ],
            "type": "string"
        }
    },
    "additionalProperties": false
},
"LabelSourceConfig": {
    "title": "LabelSourceConfig",
    "description": "Configure a :class:`.LabelSource`.",
    "type": "object",
    "properties": {
        "type_hint": {
            "title": "Type Hint",
            "default": "label_source",
            "enum": [
                "label_source"
            ],
            "type": "string"
        }
    },
    "additionalProperties": false
},
"LabelStoreConfig": {
    "title": "LabelStoreConfig",
    "description": "Configure a :class:`.LabelStore`.",
    "type": "object",
    "properties": {
        "type_hint": {
            "title": "Type Hint",
            "default": "label_store",
            "enum": [
                "label_store"
            ],
            "type": "string"
        }
    },
    "additionalProperties": false
},
"SceneConfig": {
```


(continues on next page)

(continued from previous page)


```
         "title": "SceneConfig",
         "description": "Configure a :class:`.Scene` comprising raster data &␣
↪labels for an AOI.\n    ",
         "type": "object",
         "properties": {
            "id": {
               "title": "Id",
               "type": "string"
            },
            "raster_source": {
               "$ref": "#/definitions/RasterSourceConfig"
            },
            "label_source": {
               "$ref": "#/definitions/LabelSourceConfig"
            },
            "label_store": {
               "$ref": "#/definitions/LabelStoreConfig"
            },
            "aoi_uris": {
               "title": "Aoi Uris",
               "description": "List of URIs of GeoJSON files that define the AOIs␣
↪for the scene. Each polygon defines an AOI which is a piece of the scene that is␣
↪assumed to be fully labeled and usable for training or validation. The AOIs are␣
↪assumed to be in EPSG:4326 coordinates.",
               "type": "array",
               "items": {
                  "type": "string"
               }
            },
            "type_hint": {
               "title": "Type Hint",
               "default": "scene",
               "enum": [
                  "scene"
               ],
               "type": "string"
            }
         },
         "required": [
            "id",
            "raster_source"
         ],
         "additionalProperties": false
      },
      "DatasetConfig": {
         "title": "DatasetConfig",
         "description": "Configure train, validation, and test splits for a dataset.
↪",
         "type": "object",
         "properties": {
            "class_config": {
               "$ref": "#/definitions/ClassConfig"
```


(continues on next page)

(continued from previous page)

```
            },
            "train_scenes": {
               "title": "Train Scenes",
               "type": "array",
               "items": {
                  "$ref": "#/definitions/SceneConfig"
               }
            },
            "validation_scenes": {
               "title": "Validation Scenes",
               "type": "array",
               "items": {
                  "$ref": "#/definitions/SceneConfig"
               }
            },
            "test_scenes": {
               "title": "Test Scenes",
               "default": [],
               "type": "array",
               "items": {
                  "$ref": "#/definitions/SceneConfig"
               }
            },
            "scene_groups": {
               "title": "Scene Groups",
               "description": "Groupings of scenes. Should be a dict of the form: {
↪<group-name>: Set(scene_id_1, scene_id_2, ...)}. Three groups are added by␣
↪default: \"train_scenes\", \"validation_scenes\", and \"test_scenes\"",
               "default": {},
               "type": "object",
               "additionalProperties": {
                  "type": "array",
                  "items": {
                     "type": "string"
                  },
                  "uniqueItems": true
               }
            },
            "type_hint": {
               "title": "Type Hint",
               "default": "dataset",
               "enum": [
                  "dataset"
               ],
               "type": "string"
            }
         },
         "required": [
            "class_config",
            "train_scenes",
            "validation_scenes"
         ],
```

(continues on next page)

(continued from previous page)


```
         "additionalProperties": false
      },
      "GeoDataWindowMethod": {
         "title": "GeoDataWindowMethod",
         "description": "An enumeration.",
         "enum": [
            "sliding",
            "random"
         ]
      },
      "GeoDataWindowConfig": {
         "title": "GeoDataWindowConfig",
         "description": "Configure a :class:`.GeoDataset`.\n\nSee :mod:
↪`rastervision.pytorch_learner.dataset.dataset`.",
         "type": "object",
         "properties": {
            "method": {
               "default": "sliding",
               "allOf": [
                  {
                     "$ref": "#/definitions/GeoDataWindowMethod"
                  }
               ]
            },
            "size": {
               "title": "Size",
               "description": "If method = sliding, this is the size of sliding␣
↪window. If method = random, this is the size that all the windows are resized to␣
↪before they are returned. If method = random and neither size_lims nor h_lims and␣
↪w_lims have been specified, then size_lims is set to (size, size + 1).",
               "anyOf": [
                  {
                     "type": "integer",
                     "exclusiveMinimum": 0
                  },
                  {
                     "type": "array",
                     "minItems": 2,
                     "maxItems": 2,
                     "items": [
                        {
                           "type": "integer",
                           "exclusiveMinimum": 0
                        },
                        {
                           "type": "integer",
                           "exclusiveMinimum": 0
                        }
                     ]
                  }
               ]
            },
```


(continues on next page)

(continued from previous page)

```
            "stride": {
               "title": "Stride",
               "description": "Stride of sliding window. Only used if method =␣
↪sliding.",
               "anyOf": [
                  {
                     "type": "integer",
                     "exclusiveMinimum": 0
                  },
                  {
                     "type": "array",
                     "minItems": 2,
                     "maxItems": 2,
                     "items": [
                        {
                           "type": "integer",
                           "exclusiveMinimum": 0
                        },
                        {
                           "type": "integer",
                           "exclusiveMinimum": 0
                        }
                     ]
                  }
               ]
            },
            "padding": {
               "title": "Padding",
               "description": "How many pixels are windows allowed to overflow the␣
↪edges of the raster source.",
               "anyOf": [
                  {
                     "type": "integer",
                     "minimum": 0
                  },
                  {
                     "type": "array",
                     "minItems": 2,
                     "maxItems": 2,
                     "items": [
                        {
                           "type": "integer",
                           "minimum": 0
                        },
                        {
                           "type": "integer",
                           "minimum": 0
                        }
                     ]
                  }
               ]
            },
```

(continues on next page)

(continued from previous page)


```
            "pad_direction": {
               "title": "Pad Direction",
               "description": "If \"end\", only pad ymax and xmax (bottom and␣
↪right). If \"start\", only pad ymin and xmin (top and left). If \"both\", pad all␣
↪sides. Has no effect if paddiong is zero. Defaults to \"end\".",
               "default": "end",
               "enum": [
                  "both",
                  "start",
                  "end"
               ],
               "type": "string"
            },
            "size_lims": {
               "title": "Size Lims",
               "description": "[min, max) interval from which window sizes will be␣
↪uniformly randomly sampled. The upper limit is exclusive. To fix the size to a␣
↪constant value, use size_lims = (sz, sz + 1). Only used if method = random.␣
↪Specify either size_lims or h_lims and w_lims, but not both. If neither size_lims␣
↪nor h_lims and w_lims have been specified, then this will be set to (size, size +␣
↪1).",
               "type": "array",
               "minItems": 2,
               "maxItems": 2,
               "items": [
                  {
                     "type": "integer",
                     "exclusiveMinimum": 0
                  },
                  {
                     "type": "integer",
                     "exclusiveMinimum": 0
                  }
               ]
            },
            "h_lims": {
               "title": "H Lims",
               "description": "[min, max] interval from which window heights will␣
↪be uniformly randomly sampled. Only used if method = random.",
               "type": "array",
               "minItems": 2,
               "maxItems": 2,
               "items": [
                  {
                     "type": "integer",
                     "exclusiveMinimum": 0
                  },
                  {
                     "type": "integer",
                     "exclusiveMinimum": 0
                  }
               ]
```


(continues on next page)

(continued from previous page)

```
            },
            "w_lims": {
               "title": "W Lims",
               "description": "[min, max] interval from which window widths will be␣
↪uniformly randomly sampled. Only used if method = random.",
               "type": "array",
               "minItems": 2,
               "maxItems": 2,
               "items": [
                  {
                     "type": "integer",
                     "exclusiveMinimum": 0
                  },
                  {
                     "type": "integer",
                     "exclusiveMinimum": 0
                  }
               ]
            },
            "max_windows": {
               "title": "Max Windows",
               "description": "Max allowed reads from a GeoDataset. Only used if␣
↪method = random.",
               "default": 10000,
               "minimum": 0,
               "type": "integer"
            },
            "max_sample_attempts": {
               "title": "Max Sample Attempts",
               "description": "Max attempts when trying to find a window within the␣
↪AOI of a scene. Only used if method = random and the scene has aoi_polygons␣
↪specified.",
               "default": 100,
               "exclusiveMinimum": 0,
               "type": "integer"
            },
            "efficient_aoi_sampling": {
               "title": "Efficient Aoi Sampling",
               "description": "If the scene has AOIs, sampling windows at random␣
↪anywhere in the extent and then checking if they fall within any of the AOIs can␣
↪be very inefficient. This flag enables the use of an alternate algorithm that␣
↪only samples window locations inside the AOIs. Only used if method = random and␣
↪the scene has aoi_polygons specified. Defaults to True",
               "default": true,
               "type": "boolean"
            },
            "type_hint": {
               "title": "Type Hint",
               "default": "geo_data_window",
               "enum": [
                  "geo_data_window"
               ],
```

(continues on next page)

(continued from previous page)


```
                "type": "string"
            }
         },
         "required": [
            "size"
         ],
         "additionalProperties": false
      },
      "ObjectDetectionGeoDataConfig": {
         "title": "ObjectDetectionGeoDataConfig",
         "description": "Configure object detection :class:`GeoDatasets <.
↪GeoDataset>`.\n\nSee :mod:`rastervision.pytorch_learner.dataset.object_detection_
↪dataset`.",
         "type": "object",
         "properties": {
            "class_names": {
               "title": "Class Names",
               "description": "Names of classes.",
               "default": [],
               "type": "array",
               "items": {
                  "type": "string"
               }
            },
            "class_colors": {
               "title": "Class Colors",
               "description": "Colors used to display classes. Can be color 3-
↪tuples in list form.",
               "type": "array",
               "items": {
                  "anyOf": [
                     {
                        "type": "string"
                     },
                     {
                        "type": "array",
                        "minItems": 3,
                        "maxItems": 3,
                        "items": [
                           {
                              "type": "integer"
                           },
                           {
                              "type": "integer"
                           },
                           {
                              "type": "integer"
                           }
                        ]
                     }
                  ]
               }
```


(continues on next page)

(continued from previous page)

```
            },
            "img_channels": {
               "title": "Img Channels",
               "description": "The number of channels of the training images.",
               "exclusiveMinimum": 0,
               "type": "integer"
            },
            "img_sz": {
               "title": "Img Sz",
               "description": "Length of a side of each image in pixels. This is␣
↪the size to transform it to during training, not the size in the raw dataset.",
               "default": 256,
               "exclusiveMinimum": 0,
               "type": "integer"
            },
            "train_sz": {
               "title": "Train Sz",
               "description": "If set, the number of training images to use. If␣
↪fewer images exist, then an exception will be raised.",
               "type": "integer"
            },
            "train_sz_rel": {
               "title": "Train Sz Rel",
               "description": "If set, the proportion of training images to use.",
               "type": "number"
            },
            "num_workers": {
               "title": "Num Workers",
               "description": "Number of workers to use when DataLoader makes␣
↪batches.",
               "default": 4,
               "type": "integer"
            },
            "augmentors": {
               "title": "Augmentors",
               "description": "Names of albumentations augmentors to use for␣
↪training batches. Choices include: ['Blur', 'RandomRotate90', 'HorizontalFlip',
↪'VerticalFlip', 'GaussianBlur', 'GaussNoise', 'RGBShift', 'ToGray'].␣
↪Alternatively, a custom transform can be provided via the aug_transform option.",
               "default": [
                  "RandomRotate90",
                  "HorizontalFlip",
                  "VerticalFlip"
               ],
               "type": "array",
               "items": {
                  "type": "string"
               }
            },
            "base_transform": {
               "title": "Base Transform",
               "description": "An Albumentations transform serialized as a dict␣
```

(continues on next page)

(continued from previous page)


```
↪that will be applied to all datasets: training, validation, and test. This␣
↪transformation is in addition to the resizing due to img_sz. This is useful for,␣
↪for example, applying the same normalization to all datasets.",
                "type": "object"
            },
            "aug_transform": {
                "title": "Aug Transform",
                "description": "An Albumentations transform serialized as a dict␣
↪that will be applied as data augmentation to the training dataset. This transform␣
↪is applied before base_transform. If provided, the augmentors option is ignored.",
                "type": "object"
            },
            "plot_options": {
                "title": "Plot Options",
                "description": "Options to control plotting.",
                "default": {
                    "transform": {
                        "__version__": "1.3.0",
                        "transform": {
                            "__class_fullname__": "rastervision.pytorch_learner.utils.
↪utils.MinMaxNormalize",
                            "always_apply": false,
                            "p": 1.0,
                            "min_val": 0.0,
                            "max_val": 1.0,
                            "dtype": 5
                        }
                    },
                    "channel_display_groups": null,
                    "type_hint": "plot_options"
                },
                "allOf": [
                    {
                        "$ref": "#/definitions/PlotOptions"
                    }
                ]
            },
            "preview_batch_limit": {
                "title": "Preview Batch Limit",
                "description": "Optional limit on the number of items in the preview␣
↪plots produced during training.",
                "type": "integer"
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "object_detection_geo_data",
                "enum": [
                    "object_detection_geo_data"
                ],
                "type": "string"
            },
            "scene_dataset": {
```


(continues on next page)

(continued from previous page)


```
                "$ref": "#/definitions/DatasetConfig"
            },
            "window_opts": {
                "title": "Window Opts",
                "default": {},
                "anyOf": [
                    {
                        "$ref": "#/definitions/GeoDataWindowConfig"
                    },
                    {
                        "type": "object",
                        "additionalProperties": {
                            "$ref": "#/definitions/GeoDataWindowConfig"
                        }
                    }
                ]
            }
        },
        "additionalProperties": false
    }
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *data (Union[rastervision.pytorch_learner.object_detection_learner_config.ObjectDetectionImageDataConfig, rastervision.pytorch_learner.object_detection_learner_config.ObjectDetectionGeoDataConfig])*
- *eval_train (bool)*
- *log_tensorboard (bool)*
- *model (Optional[rastervision.pytorch_learner.object_detection_learner_config.ObjectDetectionModelConfig])*
- *output_uri (Optional[str])*
- *overfit_mode (bool)*
- *predict_mode (bool)*
- *run_tensorboard (bool)*
- *save_model_bundle (bool)*
- *solver (rastervision.pytorch_learner.learner_config.SolverConfig)*
- *test_mode (bool)*
- *type_hint (Literal['object_detection_learner'])*

Validators

- `update_for_mode » all fields`

- validate_class_loss_weights » all fields
- validate_run_tensorboard » *run_tensorboard*
- *validate_solver_config* » *solver*

field data: Union[*ObjectDetectionImageDataConfig*, *ObjectDetectionGeoDataConfig*] [Required]

Validated by

- update_for_mode
- validate_class_loss_weights

field eval_train: bool = False

If True, runs final evaluation on training set (in addition to test set). Useful for debugging.

Validated by

- update_for_mode
- validate_class_loss_weights

field log_tensorboard: bool = True

Save Tensorboard log files at the end of each epoch.

Validated by

- update_for_mode
- validate_class_loss_weights

field model: Optional[*ObjectDetectionModelConfig*] = None

Validated by

- update_for_mode
- validate_class_loss_weights

field output_uri: Optional[str] = None

URI of where to save output

Validated by

- update_for_mode
- validate_class_loss_weights

field overfit_mode: bool = False

If True, uses half image size, and instead of doing epoch-based training, optimizes the model using a single batch repeatedly for overfit_num_steps number of steps.

Validated by

- update_for_mode
- validate_class_loss_weights

field predict_mode: bool = False

If True, skips training, loads model, and does final eval.

Validated by

- update_for_mode

- validate_class_loss_weights

field run_tensorboard: bool = False

run Tensorboard server during training

Validated by

- update_for_mode
- validate_class_loss_weights
- validate_run_tensorboard

field save_model_bundle: bool = True

If True, saves a model bundle at the end of training which is zip file with model and this LearnerConfig which can be used to make predictions on new images at a later time.

Validated by

- update_for_mode
- validate_class_loss_weights

field solver: *SolverConfig* [Required]

Validated by

- update_for_mode
- validate_class_loss_weights
- *validate_solver_config*

field test_mode: bool = False

If True, uses test_num_epochs, test_batch_sz, truncated datasets with only a single batch, image_sz that is cut in half, and num_workers = 0. This is useful for testing that code runs correctly on CPU without multithreading before running full job on GPU.

Validated by

- update_for_mode
- validate_class_loss_weights

field type_hint: Literal['object_detection_learner'] = 'object_detection_learner'

Validated by

- update_for_mode
- validate_class_loss_weights

build(*tmp_dir=None*, *model_weights_path=None*, *model_def_path=None*, *loss_def_path=None*, *training=True*)

Returns a Learner instantiated using this Config.

Parameters

- **tmp_dir** (*str*) – Root of temp dirs.
- **model_weights_path** (*str, optional*) – A local path to model weights. Defaults to None.
- **model_def_path** (*str, optional*) – A local path to a directory with a hubconf.py. If provided, the model definition is imported from here. Defaults to None.

- **loss_def_path** (*str, optional*) – A local path to a directory with a hubconf.py. If provided, the loss function definition is imported from here. Defaults to None.
- **training** (*bool, optional*) – Whether the model is to be used for training or prediction. If False, the model is put in eval mode and the loss function, optimizer, etc. are not initialized. Defaults to True.

get_model_bundle_uri() → str

Returns the URI of where the model bundle is stored.

Return type
str

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(**args*, ***kwargs*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validator update_for_mode » *all fields*

Parameters
values (*dict*) –

Return type
dict

validator validate_class_loss_weights » *all fields*

Parameters
values (*dict*) –

Return type
dict

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises
ConfigError – if field is invalid

validator validate_run_tensorboard » run_tensorboard

Parameters

- **v** (*bool*) –
- **values** (*dict*) –

Return type
bool

validator validate_solver_config » solver

Parameters
v (SolverConfig) –

Return type
SolverConfig

ObjectDetectionModelConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model ObjectDetectionModelConfig

Configure an object detection model.


```
{
   "title": "ObjectDetectionModelConfig",
   "description": "Configure an object detection model.",
   "type": "object",
   "properties": {
      "backbone": {
         "description": "The torchvision.models backbone to use, which must be in␣
↪the resnet* family.",
         "default": "resnet50",
         "allOf": [
            {
               "$ref": "#/definitions/Backbone"
            }
         ]
      },
      "pretrained": {
         "title": "Pretrained",
         "description": "If True, use ImageNet weights. If False, use random␣
↪initialization.",
         "default": true,
         "type": "boolean"
      },
      "init_weights": {
         "title": "Init Weights",
         "description": "URI of PyTorch model weights used to initialize model. If␣
↪set, this supercedes the pretrained option.",
         "type": "string"
      },
```


(continues on next page)

(continued from previous page)


```
      "load_strict": {
         "title": "Load Strict",
         "description": "If True, the keys in the state dict referenced by init_
↪weights must match exactly. Setting this to False can be useful if you just want␣
↪to load the backbone of a model.",
         "default": true,
         "type": "boolean"
      },
      "external_def": {
         "title": "External Def",
         "description": "If specified, the model will be built from the definition␣
↪from this external source, using Torch Hub.",
         "allOf": [
            {
               "$ref": "#/definitions/ExternalModuleConfig"
            }
         ]
      },
      "type_hint": {
         "title": "Type Hint",
         "default": "object_detection_model",
         "enum": [
            "object_detection_model"
         ],
         "type": "string"
      }
   },
   "additionalProperties": false,
   "definitions": {
      "Backbone": {
         "title": "Backbone",
         "description": "An enumeration.",
         "enum": [
            "alexnet",
            "densenet121",
            "densenet169",
            "densenet201",
            "densenet161",
            "googlenet",
            "inception_v3",
            "mnasnet0_5",
            "mnasnet0_75",
            "mnasnet1_0",
            "mnasnet1_3",
            "mobilenet_v2",
            "resnet18",
            "resnet34",
            "resnet50",
            "resnet101",
            "resnet152",
            "resnext50_32x4d",
            "resnext101_32x8d",
```


(continues on next page)

(continued from previous page)


```
            "wide_resnet50_2",
            "wide_resnet101_2",
            "shufflenet_v2_x0_5",
            "shufflenet_v2_x1_0",
            "shufflenet_v2_x1_5",
            "shufflenet_v2_x2_0",
            "squeezenet1_0",
            "squeezenet1_1",
            "vgg11",
            "vgg11_bn",
            "vgg13",
            "vgg13_bn",
            "vgg16",
            "vgg16_bn",
            "vgg19_bn",
            "vgg19"
         ]
      },
      "ExternalModuleConfig": {
         "title": "ExternalModuleConfig",
         "description": "Config describing an object to be loaded via Torch Hub.",
         "type": "object",
         "properties": {
            "uri": {
               "title": "Uri",
               "description": "Local uri of a zip file, or local uri of a directory,
↪or remote uri of zip file.",
               "minLength": 1,
               "type": "string"
            },
            "github_repo": {
               "title": "Github Repo",
               "description": "<repo-owner>/<repo-name>[:tag]",
               "pattern": ".+/.+",
               "type": "string"
            },
            "name": {
               "title": "Name",
               "description": "Name of the folder in which to extract/copy the␣
↪definition files.",
               "minLength": 1,
               "type": "string"
            },
            "entrypoint": {
               "title": "Entrypoint",
               "description": "Name of a callable present in hubconf.py. See docs␣
↪for torch.hub for details.",
               "minLength": 1,
               "type": "string"
            },
            "entrypoint_args": {
               "title": "Entrypoint Args",
```


(continues on next page)

(continued from previous page)


```
                "description": "Args to pass to the entrypoint. Must be serializable.
↪",
                "default": [],
                "type": "array",
                "items": {}
            },
            "entrypoint_kwargs": {
                "title": "Entrypoint Kwargs",
                "description": "Keyword args to pass to the entrypoint. Must be␣
↪serializable.",
                "default": {},
                "type": "object"
            },
            "force_reload": {
                "title": "Force Reload",
                "description": "Force reload of module definition.",
                "default": false,
                "type": "boolean"
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "external-module",
                "enum": [
                    "external-module"
                ],
                "type": "string"
            }
        },
        "required": [
            "entrypoint"
        ],
        "additionalProperties": false
    }
  }
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *backbone (rastervision.pytorch_learner.learner_config.Backbone)*
- *external_def (Optional[rastervision.pytorch_learner.learner_config.ExternalModuleConfig])*
- *init_weights (Optional[str])*
- *load_strict (bool)*
- *pretrained (bool)*
- *type_hint (Literal['object_detection_model'])*

Validators

- *only_valid_backbones* » *backbone*

field backbone: *Backbone* = Backbone.resnet50

The torchvision.models backbone to use, which must be in the resnet* family.

Validated by

- *only_valid_backbones*

field external_def: Optional[*ExternalModuleConfig*] = None

If specified, the model will be built from the definition from this external source, using Torch Hub.

field init_weights: Optional[str] = None

URI of PyTorch model weights used to initialize model. If set, this supercedes the pretrained option.

field load_strict: bool = True

If True, the keys in the state dict referenced by init_weights must match exactly. Setting this to False can be useful if you just want to load the backbone of a model.

field pretrained: bool = True

If True, use ImageNet weights. If False, use random initialization.

field type_hint: Literal['object_detection_model'] = 'object_detection_model'

build(*num_classes: int*, *in_channels: int*, *save_dir: Optional[str] = None*, *hubconf_dir: Optional[str] = None*, ***kwargs*) → torch.nn.Module

Build and return a model based on the config.

Parameters

- **num_classes** (*int*) – Number of classes.
- **in_channels** (*int, optional*) – Number of channels in the images that will be fed into the model. Defaults to 3.
- **save_dir** (*Optional[str], optional*) – Used for building external_def if specified. Defaults to None.
- **hubconf_dir** (*Optional[str], optional*) – Used for building external_def if specified. Defaults to None.

Returns
a PyTorch nn.Module.

Return type
nn.Module

build_default_model(*num_classes: int*, *in_channels: int*, *img_sz: int*) → torchvision.models.detection.faster_rcnn.FasterRCNN

Returns a FasterRCNN model.

Note that the model returned will have (num_classes + 2) output classes. +1 for the null class (zeroth index), and another +1 (last index) for backward compatibility with earlier Raster Vision versions.

Returns
a FasterRCNN model.

Return type
FasterRCNN

Parameters

- **num_classes** (*int*) –
- **in_channels** (*int*) –
- **img_sz** (*int*) –

build_external_model(*save_dir: str*, *hubconf_dir: Optional[str] = None*) → torch.nn.Module

Build and return an external model.

Parameters

- **save_dir** (*str*) – The module def will be saved here.
- **hubconf_dir** (*Optional[str], optional*) – Path to existing definition. Defaults to None.

Returns
a PyTorch nn.Module.

Return type
nn.Module

get_backbone_str()

validator only_valid_backbones » backbone

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(**args*, ***kwargs*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises
ConfigError – if field is invalid

9.3.10 object_detection_utils

Classes

BoxList	
TorchVisionODAdapter	Adapter for interfacing with TorchVision's object detection models.

BoxList

class BoxList

Bases: object

__init__(*boxes: torch.Tensor*, *format: str = 'xyxy'*, ***extras*) → None

Representation of a list of bounding boxes and associated data.

Internally, boxes are always stored in the xyxy format.

Parameters

- **boxes** (*torch.Tensor*) – tensor<n, 4>
- **format** (*str*) – format of input boxes.
- **extras** – dict with values that are tensors with first dimension corresponding to boxes first dimension

Return type
None

Methods

__init__(boxes[, format])	Representation of a list of bounding boxes and associated data.
cat(box_lists)	
clip_boxes(img_height, img_width)	
convert_boxes(out_fmt)	
copy()	
equal(other)	
get_field(name)	
ind_filter(inds)	
nms([iou_thresh])	
pin_memory()	
scale(yscale, xscale)	
score_filter([score_thresh])	
to(*args, **kwargs)	

__init__(*boxes: torch.Tensor*, *format: str = 'xyxy'*, ***extras*) → None

Representation of a list of bounding boxes and associated data.

Internally, boxes are always stored in the xyxy format.

Parameters

- **boxes** (*torch.Tensor*) – tensor<n, 4>
- **format** (*str*) – format of input boxes.
- **extras** – dict with values that are tensors with first dimension corresponding to boxes first dimension

Return type
None

static cat(*box_lists: Iterable[*BoxList*]*) → *BoxList*

Parameters
box_lists (*Iterable[*BoxList*]*) –

Return type
BoxList

clip_boxes(*img_height: int*, *img_width: int*) → *BoxList*

Parameters

- **img_height** (*int*) –
- **img_width** (*int*) –

Return type
BoxList

convert_boxes(*out_fmt: str*) → torch.Tensor

Parameters
out_fmt (*str*) –

Return type
torch.Tensor

copy() → *BoxList*

Return type
BoxList

equal(*other:* BoxList) → bool

Parameters
other (BoxList) –

Return type
bool

get_field(*name: str*) → Any

Parameters
name (*str*) –

Return type
Any

ind_filter(*inds: Sequence[int]*) → *BoxList*

Parameters
inds (*Sequence[int]*) –

Return type
BoxList

nms(*iou_thresh: float = 0.5*) → torch.Tensor

Parameters
iou_thresh (*float*) –

Return type
torch.Tensor

pin_memory() → *BoxList*

Return type
BoxList

scale(*yscale: float*, *xscale: float*) → *BoxList*

Parameters

- **yscale** (*float*) –
- **xscale** (*float*) –

Return type
BoxList

score_filter(*score_thresh: float = 0.25*) → *BoxList*

Parameters
score_thresh (*float*) –

Return type
BoxList

to(**args*, ***kwargs*) → *BoxList*

Return type
BoxList

TorchVisionODAdapter

class TorchVisionODAdapter

Bases: Module

Adapter for interfacing with TorchVision's object detection models.

The purpose of this adapter is: 1) to convert input BoxLists to dicts before feeding them into the model 2) to convert detections output by the model as dicts into BoxLists

Additionally, it automatically converts to/from 1-indexed class labels (which is what the TorchVision models expect).

__init__(*model: torch.nn.Module*, *ignored_output_inds: Sequence[int] = [0]*) → None

Constructor.

Parameters

- **model** (*nn.Module*) – A torchvision object detection model.
- **ignored_output_inds** (*Iterable[int], optional*) – Class labels to exclude. Defaults to [0].

Return type
None

Methods

__init__(model[, ignored_output_inds])	Constructor.
boxlist_to_model_input_dict(boxlist)	Convert BoxList to a dict compatible with torchvision detection models.
forward(input[, targets])	Forward pass.
model_output_dict_to_boxlist(out)	Convert torchvision detection dict to BoxList.

__init__(*model: torch.nn.Module*, *ignored_output_inds: Sequence[int] = [0]*) → None

Constructor.

Parameters

- **model** (*nn.Module*) – A torchvision object detection model.

- **ignored_output_inds** (*Iterable[int], optional*) – Class labels to exclude. Defaults to [0].

Return type
None

static __new__(*cls*, **args: Any*, ***kwargs: Any*) → Any

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
Any

boxlist_to_model_input_dict(*boxlist:* BoxList) → dict

Convert BoxList to a dict compatible with torchvision detection models. Also, make class labels 1-indexed.

Parameters
boxlist (BoxList) – A BoxList with a "class_ids" field.

Returns
A dict with keys: "boxes" and "labels".

Return type
dict

forward(*input: torch.Tensor*, *targets: Optional[Iterable[*BoxList*]] = None*) → Union[dict, List[*BoxList*]]

Forward pass.

Parameters

- **input** (*Tensor[batch_size, in_channels, in_height, in_width]*) – batch of images.
- **targets** (*Optional[Iterable[*BoxList*]], optional*) – In training mode, should be Iterable[BoxList]], with each BoxList having a 'class_ids' field. In eval mode, should be None. Defaults to None.

Returns

In training mode,
returns a dict of losses. In eval mode, returns a list of BoxLists containing predicted boxes, class_ids, and scores. Further filtering based on score should be done before considering the prediction "final".

Return type
Union[dict, List[*BoxList*]]

model_output_dict_to_boxlist(*out: dict*) → *BoxList*

Convert torchvision detection dict to BoxList. Also, exclude any null classes and make class labels 0-indexed.

Parameters
out (*dict*) – A dict output by a torchvision detection model in eval mode.

Returns
A BoxList with "class_ids" and "scores" fields.

Return type
BoxList

Functions

collate_fn(data)	
compute_coco_eval(outputs, targets, ...)	Return mAP averaged over 0.5-0.95 using pycocotools eval.
draw_boxes(x, y, class_names, class_colors)	Given an image and a BoxList, draw the boxes in the BoxList on the image.
get_coco_gt(targets, num_class_ids)	
get_coco_preds(outputs)	

collate_fn

collate_fn(*data: Iterable[Sequence]*) → Tuple[torch.Tensor, List[*BoxList*]]

Parameters
data (*Iterable[Sequence]*) –

Return type
Tuple[torch.Tensor, *List*[BoxList]]

compute_coco_eval

compute_coco_eval(*outputs*, *targets*, *num_class_ids*)

Return mAP averaged over 0.5-0.95 using pycocotools eval.

Note: boxes are in (ymin, xmin, ymax, xmax) format with values ranging
from 0 to h or w.

Parameters

- **outputs** – (list) of length m containing dicts of form {'boxes': <tensor with shape (n, 4)>, 'class_ids': <tensor with shape (n,)>, 'scores': <tensor with shape (n,)>}
- **targets** – (list) of length m containing dicts of form {'boxes': <tensor with shape (n, 4)>, 'class_ids': <tensor with shape (n,)>}

draw_boxes

draw_boxes(*x: torch.Tensor*, *y:* BoxList, *class_names: Sequence[str]*, *class_colors: Sequence[str]*) → torch.Tensor

Given an image and a BoxList, draw the boxes in the BoxList on the image.

Parameters

- **x** (*torch.Tensor*) –
- **y** (BoxList) –
- **class_names** (*Sequence[str]*) –
- **class_colors** (*Sequence[str]*) –

Return type
torch.Tensor

get_coco_gt

get_coco_gt(*targets: Iterable[BoxList]*, *num_class_ids: int*) → Dict[str, List[dict]]

Parameters

- **targets** (*Iterable[BoxList]*) –
- **num_class_ids** (*int*) –

Return type
Dict[str, *List*[dict]]

get_coco_preds

get_coco_preds(*outputs: Iterable[BoxList]*) → List[dict]

Parameters
outputs (*Iterable[BoxList]*) –

Return type
List[dict]

9.3.11 regression_learner

Classes

RegressionLearner

RegressionLearner

class RegressionLearner

Bases: *Learner*

__init__(*cfg: LearnerConfig*, *output_dir: Optional[str] = None*, *train_ds: Optional[Dataset] = None*, *valid_ds: Optional[Dataset] = None*, *test_ds: Optional[Dataset] = None*, *model: Optional[torch.nn.Module] = None*, *loss: Optional[Callable] = None*, *optimizer: Optional[Optimizer] = None*, *epoch_scheduler: Optional[_LRScheduler] = None*, *step_scheduler: Optional[_LRScheduler] = None*, *tmp_dir: Optional[str] = None*, *model_weights_path: Optional[str] = None*, *model_def_path: Optional[str] = None*, *loss_def_path: Optional[str] = None*, *training: bool = True*)

Constructor.

Parameters

- **cfg** (*LearnerConfig*) – LearnerConfig.
- **train_ds** (*Optional[Dataset], optional*) – The dataset to use for training. If None, will be generated from cfg.data. Defaults to None.

- **valid_ds** (*Optional[Dataset], optional*) – The dataset to use for validation. If None, will be generated from cfg.data. Defaults to None.
- **test_ds** (*Optional[Dataset], optional*) – The dataset to use for testing. If None, will be generated from cfg.data. Defaults to None.
- **model** (*Optional[nn.Module], optional*) – The model. If None, will be generated from cfg.model. Defaults to None.
- **loss** (*Optional[Callable], optional*) – The loss function. If None, will be generated from cfg.solver. Defaults to None.
- **optimizer** (*Optional[Optimizer], optional*) – The optimizer. If None, will be generated from cfg.solver. Defaults to None.
- **epoch_scheduler** (*Optional[_LRScheduler], optional*) – The scheduler that updates after each epoch. If None, will be generated from cfg.solver. Defaults to None.
- **step_scheduler** (*Optional[_LRScheduler], optional*) – The scheduler that updates after each optimizer-step. If None, will be generated from cfg.solver. Defaults to None.
- **tmp_dir** (*Optional[str], optional*) – A temporary directory to use for downloads etc. If None, will be auto-generated. Defaults to None.
- **model_weights_path** (*Optional[str], optional*) – URI of model weights to initialize the model with. Defaults to None.
- **model_def_path** (*Optional[str], optional*) – A local path to a directory with a hubconf.py. If provided, the model definition is imported from here. This is used when loading an external model from a model-bundle. Defaults to None.
- **loss_def_path** (*Optional[str], optional*) – A local path to a directory with a hubconf.py. If provided, the loss function definition is imported from here. This is used when loading an external loss function from a model-bundle. Defaults to None.
- **training** (*bool, optional*) – If False, the training apparatus (loss, optimizer, scheduler, logging, etc.) will not be set up and the model will be put into eval mode. If True, the training apparatus will be set up and the model will be put into training mode. Defaults to True.
- **output_dir** (*Optional[str]*) –

Methods

__init__(cfg[, output_dir, train_ds, ...])	Constructor.
build_dataloaders()	Set the DataLoaders for train, validation, and test sets.
build_datasets()	
build_epoch_scheduler([start_epoch])	Returns an LR scheduler that changes the LR each epoch.
build_loss([loss_def_path])	Build a loss Callable.
build_metric_names()	Returns names of metrics used to validate model at each epoch.
build_model([model_def_path])	Override to pass class_names, pos_class_names, and prob_class_names.
build_optimizer()	Returns optimizer.

continues on next page

Table 6 – continued from previous page

build_step_scheduler([start_epoch])	Returns an LR scheduler that changes the LR each step.
eval_model(split)	Evaluate model using a particular dataset split.
from_model_bundle(model_bundle_uri[, ...])	Create a Learner from a model bundle.
get_collate_fn()	Returns a custom collate_fn to use in DataLoader.
get_dataloader(split)	Get the DataLoader for a split.
get_start_epoch()	Get start epoch.
get_train_sampler(train_ds)	Return a sampler to use for the training dataloader or None to not use any.
get_visualizer_class()	Returns a Visualizer class object for plotting data samples.
load_checkpoint()	Load last weights from previous run if available.
load_init_weights([model_weights_path])	Load the weights to initialize model.
load_weights(uri, **kwargs)	Load model weights from a file.
log_data_stats()	Log stats about each DataSet.
main()	Main training sequence.
normalize_input(x)	Normalize x to [0, 1].
numpy_predict(x[, raw_out])	Make a prediction using an image or batch of images in numpy format.
on_epoch_end(curr_epoch, metrics)	Hook that is called at end of epoch.
on_overfit_start()	Hook that is called at start of overfit routine.
on_train_start()	Hook that is called at start of train routine.
output_to_numpy(out)	Convert output of model to numpy format.
overfit()	Optimize model using the same batch repeatedly.
plot_dataloader(dl, output_path[, ...])	Plot images and ground truth labels for a DataLoader.
plot_dataloaders([batch_limit, show])	Plot images and ground truth labels for all DataLoaders.
plot_predictions(split[, batch_limit, show])	Plot predictions for a split.
post_forward(x)	Post process output of call to model().
predict(x[, raw_out])	Make prediction for an image or batch of images.
predict_dataloader(dl[, batched_output, ...])	Returns an iterator over predictions on the given dataloader.
predict_dataset(dataset[, return_format, ...])	Returns an iterator over predictions on the given dataset.
prob_to_pred(x)	Convert a Tensor with prediction probabilities to class ids.
run_tensorboard()	Run TB server serving logged stats.
save_model_bundle()	Save a model bundle.
setup_data()	Set datasets and dataLoaders for train, validation, and test sets.
setup_loss([loss_def_path])	Setup self.loss.
setup_model([model_weights_path, model_def_path])	Setup self.model.
setup_tensorboard()	Setup for logging stats to TB.
setup_training([loss_def_path])	
stop_tensorboard()	Stop TB logging and server if it's running.
sync_from_cloud()	Sync any previous output in the cloud to output_dir.
sync_to_cloud()	Sync any output to the cloud at output_uri.
to_batch(x)	Ensure that image array has batch dimension.
to_device(x, device)	Load Tensors onto a device.

continues on next page

Table 6 – continued from previous page

train([epochs])	Training loop that will attempt to resume training if appropriate.
train_end(outputs, num_samples)	Aggregate the ouput of train_step at the end of the epoch.
train_epoch(optimizer[, step_scheduler])	Train for a single epoch.
train_step(batch, batch_ind)	Compute loss for a single training batch.
validate_end(outputs, num_samples)	Aggregate the ouput of validate_step at the end of the epoch.
validate_epoch(dl)	Validate for a single epoch.
validate_step(batch, batch_nb)	Compute metrics on validation batch.

__init__(*cfg: LearnerConfig*, *output_dir: Optional[str] = None*, *train_ds: Optional[Dataset] = None*, *valid_ds: Optional[Dataset] = None*, *test_ds: Optional[Dataset] = None*, *model: Optional[torch.nn.Module] = None*, *loss: Optional[Callable] = None*, *optimizer: Optional[Optimizer] = None*, *epoch_scheduler: Optional[_LRScheduler] = None*, *step_scheduler: Optional[_LRScheduler] = None*, *tmp_dir: Optional[str] = None*, *model_weights_path: Optional[str] = None*, *model_def_path: Optional[str] = None*, *loss_def_path: Optional[str] = None*, *training: bool = True*)

Constructor.

Parameters

- **cfg** (*LearnerConfig*) – LearnerConfig.
- **train_ds** (*Optional[Dataset], optional*) – The dataset to use for training. If None, will be generated from cfg.data. Defaults to None.
- **valid_ds** (*Optional[Dataset], optional*) – The dataset to use for validation. If None, will be generated from cfg.data. Defaults to None.
- **test_ds** (*Optional[Dataset], optional*) – The dataset to use for testing. If None, will be generated from cfg.data. Defaults to None.
- **model** (*Optional[nn.Module], optional*) – The model. If None, will be generated from cfg.model. Defaults to None.
- **loss** (*Optional[Callable], optional*) – The loss function. If None, will be generated from cfg.solver. Defaults to None.
- **optimizer** (*Optional[Optimizer], optional*) – The optimizer. If None, will be generated from cfg.solver. Defaults to None.
- **epoch_scheduler** (*Optional[_LRScheduler], optional*) – The scheduler that updates after each epoch. If None, will be generated from cfg.solver. Defaults to None.
- **step_scheduler** (*Optional[_LRScheduler], optional*) – The scheduler that updates after each optimizer-step. If None, will be generated from cfg.solver. Defaults to None.
- **tmp_dir** (*Optional[str], optional*) – A temporary directory to use for downloads etc. If None, will be auto-generated. Defaults to None.
- **model_weights_path** (*Optional[str], optional*) – URI of model weights to initialize the model with. Defaults to None.
- **model_def_path** (*Optional[str], optional*) – A local path to a directory with a hubconf.py. If provided, the model definition is imported from here. This is used when loading an external model from a model-bundle. Defaults to None.

- **loss_def_path** (*Optional[str], optional*) – A local path to a directory with a hubconf.py. If provided, the loss function definition is imported from here. This is used when loading an external loss function from a model-bundle. Defaults to None.
- **training** (*bool, optional*) – If False, the training apparatus (loss, optimizer, scheduler, logging, etc.) will not be set up and the model will be put into eval mode. If True, the training apparatus will be set up and the model will be put into training mode. Defaults to True.
- **output_dir** (*Optional[str]*) –

build_dataloaders() → Tuple[torch.utils.data.DataLoader, torch.utils.data.DataLoader, torch.utils.data.DataLoader]

Set the DataLoaders for train, validation, and test sets.

Return type
Tuple[torch.utils.data.DataLoader, torch.utils.data.DataLoader, torch.utils.data.DataLoader]

build_datasets() → Tuple[Dataset, Dataset, Dataset]

Return type
Tuple[Dataset, Dataset, Dataset]

build_epoch_scheduler(*start_epoch: int = 0*) → _LRScheduler

Returns an LR scheduler that changes the LR each epoch.

Parameters
start_epoch (*int*) –

Return type
_LRScheduler

build_loss(*loss_def_path: Optional[str] = None*) → Callable

Build a loss Callable.

Parameters
loss_def_path (*Optional[str]*) –

Return type
Callable

build_metric_names()

Returns names of metrics used to validate model at each epoch.

build_model(*model_def_path: Optional[str] = None*) → nn.Module

Override to pass class_names, pos_class_names, and prob_class_names.

Parameters
model_def_path (*Optional[str]*) –

Return type
nn.Module

build_optimizer() → Optimizer

Returns optimizer.

Return type
Optimizer

build_step_scheduler(*start_epoch: int = 0*) → _LRScheduler

Returns an LR scheduler that changes the LR each step.

Parameters
start_epoch (*int*) –

Return type
_LRScheduler

eval_model(*split*)

Evaluate model using a particular dataset split.

Gets validation metrics and saves them along with prediction plots.

Parameters
split – the dataset split to use: train, valid, or test.

classmethod from_model_bundle(*model_bundle_uri: str*, *tmp_dir: Optional[str] = None*, *cfg: Optional[LearnerConfig] = None*, *training: bool = False*, ***kwargs*) → *Learner*

Create a Learner from a model bundle.

Note: This is the bundle saved in `train/model-bundle.zip` and not `bundle/model-bundle.zip`.

Parameters

- **model_bundle_uri** (*str*) – URI of the model bundle.
- **tmp_dir** (*Optional[str], optional*) – Optional temporary directory. Will be used for unzipping bundle and also passed to the default constructor. If None, will be auto-generated. Defaults to None.
- **cfg** (*Optional[LearnerConfig], optional*) – If None, will be read from the bundle. Defaults to None.
- **training** (*bool, optional*) – If False, the training apparatus (loss, optimizer, scheduler, logging, etc.) will not be set up and the model will be put into eval mode. If True, the training apparatus will be set up and the model will be put into training mode. Defaults to True.
- ****kwargs** – See *Learner.__init__()*.

Raises
FileNotFoundError – If using custom Albumentations transforms and definition file is not found in bundle.

Returns
Object of the Learner subclass on which this was called.

Return type
Learner

get_collate_fn() → Optional[callable]

Returns a custom collate_fn to use in DataLoader.

None is returned if default collate_fn should be used.

See https://pytorch.org/docs/stable/data.html#working-with-collate-fn

Return type
Optional[callable]

get_dataloader(*split: str*) → torch.utils.data.DataLoader

Get the DataLoader for a split.

Parameters
split (*str*) – a split name which can be train, valid, or test

Return type
torch.utils.data.DataLoader

get_start_epoch() → int

Get start epoch.

If training was interrupted, this returns the last complete epoch + 1.

Return type
int

get_train_sampler(*train_ds: Dataset*) → Optional[Sampler]

Return a sampler to use for the training dataloader or None to not use any.

Parameters
train_ds (*Dataset*) –

Return type
Optional[Sampler]

get_visualizer_class()

Returns a Visualizer class object for plotting data samples.

load_checkpoint()

Load last weights from previous run if available.

load_init_weights(*model_weights_path: Optional[str] = None*) → None

Load the weights to initialize model.

Parameters
model_weights_path (*Optional[str]*) –

Return type
None

load_weights(*uri: str*, ***kwargs*) → None

Load model weights from a file.

Parameters
uri (*str*) –

Return type
None

log_data_stats()

Log stats about each DataSet.

main()

Main training sequence.

This plots the dataset, runs a training and validation loop (which will resume if interrupted), logs stats, plots predictions, and syncs results to the cloud.

normalize_input(*x: ndarray*) → ndarray

Normalize x to [0, 1].

If x.dtype is a subtype of np.unsignedinteger, normalize it to [0, 1] using the max possible value of that dtype. Otherwise, assume it is in [0, 1] already and do nothing.

Parameters
x (*np.ndarray*) – an image or batch of images

Returns
the same array scaled to [0, 1].

Return type
ndarray

numpy_predict(*x: ndarray*, *raw_out: bool = False*) → ndarray

Make a prediction using an image or batch of images in numpy format. If x.dtype is a subtype of np.unsignedinteger, it will be normalized to [0, 1] using the max possible value of that dtype. Otherwise, x will be assumed to be in [0, 1] already and will be cast to torch.float32 directly.

Parameters

- **x** (*ndarray*) – (ndarray) of shape [height, width, channels] or [batch_sz, height, width, channels]
- **raw_out** (*bool*) – if True, return prediction probabilities

Returns
predictions using numpy arrays

Return type
ndarray

on_epoch_end(*curr_epoch*, *metrics*)

Hook that is called at end of epoch.

Writes metrics to CSV and TB, and saves model.

on_overfit_start()

Hook that is called at start of overfit routine.

on_train_start()

Hook that is called at start of train routine.

output_to_numpy(*out: torch.Tensor*) → ndarray

Convert output of model to numpy format.

Parameters
out (*torch.Tensor*) – the output of the model in PyTorch format

Return type
ndarray

Returns: the output of the model in numpy format

overfit()

Optimize model using the same batch repeatedly.

plot_dataloader(*dl: torch.utils.data.DataLoader*, *output_path: str*, *batch_limit: Optional[int] = None*, *show: bool = False*)

Plot images and ground truth labels for a DataLoader.

Parameters

- **dl** (*torch.utils.data.DataLoader*) –
- **output_path** (*str*) –
- **batch_limit** (*Optional[int]*) –
- **show** (*bool*) –

plot_dataloaders(*batch_limit: Optional[int] = None*, *show: bool = False*)

Plot images and ground truth labels for all DataLoaders.

Parameters

- **batch_limit** (*Optional[int]*) –
- **show** (*bool*) –

plot_predictions(*split: str*, *batch_limit: Optional[int] = None*, *show: bool = False*)

Plot predictions for a split.

Uses the first batch for the corresponding DataLoader.

Parameters

- **split** (*str*) – dataset split. Can be train, valid, or test.
- **batch_limit** (*Optional[int]*) – optional limit on (rendered) batch size
- **show** (*bool*) –

post_forward(*x: Any*) → Any

Post process output of call to model().

Useful for when predictions are inside a structure returned by model().

Parameters
x (*Any*) –

Return type
Any

predict(*x: torch.Tensor*, *raw_out: bool = False*) → Any

Make prediction for an image or batch of images.

Parameters

- **x** (*Tensor*) – Image or batch of images as a float Tensor with pixel values normalized to [0, 1].
- **raw_out** (*bool*) – if True, return prediction probabilities

Returns

the predictions, in probability form if raw_out is True, in class_id form
otherwise

Return type
Any

predict_dataloader(*dl: torch.utils.data.DataLoader*, *batched_output: bool = True*, *return_format: Literal['xyz', 'yz', 'z'] = 'z'*, *raw_out: bool = True*, *predict_kw: dict = {}*) → Union[Iterator[Any], Iterator[Tuple[Any, ...]]]

Returns an iterator over predictions on the given dataloader.

Parameters

- **dl** (*DataLoader*) – The dataloader to make predictions on.
- **batched_output** (*bool, optional*) – If True, return batches of x, y, z as defined by the dataloader. If False, unroll the batches into individual items. Defaults to True.
- **return_format** (*Literal['xyz', 'yz', 'z'], optional*) – Format of the return elements of the returned iterator. Must be one of: 'xyz', 'yz', and 'z'. If 'xyz', elements are 3-tuples of x, y, and z. If 'yz', elements are 2-tuples of y and z. If 'z', elements are (non-tuple) values of z. Where x = input image, y = ground truth, and z = prediction. Defaults to 'z'.
- **raw_out** (*bool, optional*) – If true, return raw predicted scores. Defaults to True.
- **predict_kw** (*dict*) – Dict with keywords passed to Learner.predict(). Useful if a Learner subclass implements a custom predict() method.

Raises

ValueError – If return_format is not one of the allowed values.

Returns

If return_format
is 'z', the returned value is an iterator of whatever type the predictions are. Otherwise, the returned value is an iterator of tuples.

Return type

Union[Iterator[Any], Iterator[Tuple[Any, ...]]]

predict_dataset(*dataset: Dataset*, *return_format: Literal['xyz', 'yz', 'z'] = 'z'*, *raw_out: bool = True*, *numpy_out: bool = False*, *predict_kw: dict = {}*, *dataloader_kw: dict = {}*, *progress_bar: bool = True*, *progress_bar_kw: dict = {}*) → Union[Iterator[Any], Iterator[Tuple[Any, ...]]]

Returns an iterator over predictions on the given dataset.

Parameters

- **dataset** (*Dataset*) – The dataset to make predictions on.
- **return_format** (*Literal['xyz', 'yz', 'z'], optional*) – Format of the return elements of the returned iterator. Must be one of: 'xyz', 'yz', and 'z'. If 'xyz', elements are 3-tuples of x, y, and z. If 'yz', elements are 2-tuples of y and z. If 'z', elements are (non-tuple) values of z. Where x = input image, y = ground truth, and z = prediction. Defaults to 'z'.
- **raw_out** (*bool, optional*) – If true, return raw predicted scores. Defaults to True.
- **numpy_out** (*bool, optional*) – If True, convert predictions to numpy arrays before returning. Defaults to False.
- **predict_kw** (*dict*) – Dict with keywords passed to Learner.predict(). Useful if a Learner subclass implements a custom predict() method.
- **dataloader_kw** (*dict*) – Dict with keywords passed to the DataLoader constructor.
- **progress_bar** (*bool, optional*) – If True, display a progress bar. Since this function returns an iterator, the progress bar won't be visible until the iterator is consumed. Defaults to True.
- **progress_bar_kw** (*dict*) – Dict with keywords passed to tqdm.

Raises

ValueError – If return_format is not one of the allowed values.

Returns

If return_format
is 'z', the returned value is an iterator of whatever type the predictions are. Otherwise, the returned value is an iterator of tuples.

Return type
Union[Iterator[Any], Iterator[Tuple[Any, ...]]]

`prob_to_pred`(*x*)

Convert a Tensor with prediction probabilities to class ids.

The class ids should be the classes with the maximum probability.

`run_tensorboard`()

Run TB server serving logged stats.

`save_model_bundle`()

Save a model bundle.

This is a zip file with the model weights in .pth format and a serialized copy of the LearningConfig, which allows for making predictions in the future.

`setup_data`()

Set datasets and dataLoaders for train, validation, and test sets.

`setup_loss`(*loss_def_path: Optional[str] = None*) → None

Setup self.loss.

Parameters

- **loss_def_path** (*str, optional*) – Loss definition path. Will be
- **None.** (*available when loading from a bundle. Defaults to*) –

Return type
None

`setup_model`(*model_weights_path: Optional[str] = None*, *model_def_path: Optional[str] = None*) → None

Setup self.model.

Parameters

- **model_weights_path** (*Optional[str], optional*) – Path to model weights. Will be available when loading from a bundle. Defaults to None.
- **model_def_path** (*Optional[str], optional*) – Path to model definition. Will be available when loading from a bundle. Defaults to None.

Return type
None

`setup_tensorboard`()

Setup for logging stats to TB.

`setup_training`(*loss_def_path: Optional[str] = None*) → None

Parameters
loss_def_path (*Optional[str]*) –

Return type
None

stop_tensorboard()

Stop TB logging and server if it's running.

sync_from_cloud()

Sync any previous output in the cloud to output_dir.

sync_to_cloud()

Sync any output to the cloud at output_uri.

to_batch(*x: torch.Tensor*) → torch.Tensor

Ensure that image array has batch dimension.

Parameters
x (*torch.Tensor*) – assumed to be either image or batch of images

Returns
x with extra batch dimension of length 1 if needed

Return type
torch.Tensor

to_device(*x: Any*, *device: str*) → Any

Load Tensors onto a device.

Parameters

- **x** (*Any*) – some object with Tensors in it
- **device** (*str*) – 'cpu' or 'cuda'

Returns
x but with any Tensors in it on the device

Return type
Any

train(*epochs: Optional[int] = None*)

Training loop that will attempt to resume training if appropriate.

Parameters
epochs (*Optional[int]*) –

train_end(*outputs: List[Dict[str, float]]*, *num_samples: int*) → Dict[str, float]

Aggregate the ouput of train_step at the end of the epoch.

Parameters

- **outputs** (*List[Dict[str, float]]*) – a list of outputs of train_step
- **num_samples** (*int*) – total number of training samples processed in epoch

Return type
Dict[str, float]

train_epoch(*optimizer: Optimizer*, *step_scheduler: Optional[_LRScheduler] = None*) → Dict[str, float]

Train for a single epoch.

Parameters

- **optimizer** (*Optimizer*) –
- **step_scheduler** (*Optional[_LRScheduler]*) –

Return type
Dict[str, float]

train_step(*batch*, *batch_ind*)

Compute loss for a single training batch.

Parameters

- **batch** – batch data needed to compute loss
- **batch_ind** – index of batch within epoch

Returns
dict with 'train_loss' as key and possibly other losses

validate_end(*outputs: List[Dict[str, float]]*, *num_samples: int*) → Dict[str, float]

Aggregate the ouput of validate_step at the end of the epoch.

Parameters

- **outputs** (*List[Dict[str, float]]*) – a list of outputs of validate_step
- **num_samples** (*int*) – total number of validation samples processed in epoch

Return type
Dict[str, float]

validate_epoch(*dl: torch.utils.data.DataLoader*) → Dict[str, float]

Validate for a single epoch.

Parameters
dl (*torch.utils.data.DataLoader*) –

Return type
Dict[str, float]

validate_step(*batch*, *batch_nb*)

Compute metrics on validation batch.

Parameters

- **batch** – batch data needed to compute validation metrics
- **batch_ind** – index of batch within epoch

Returns
dict with metric names mapped to metric values

9.3.12 regression_learner_config

Classes

RegressionDataFormat	An enumeration.
RegressionModel	

RegressionDataFormat

class RegressionDataFormat

Bases: Enum

An enumeration.

Attributes

csv

__init__()

csv = 'csv'

RegressionModel

class RegressionModel

Bases: Module

__init__(*backbone_arch*, *out_features*, *pretrained=True*, *pos_out_inds=None*, *prob_out_inds=None*)

Methods

__init__(backbone_arch, out_features[, ...])

forward(x)

__init__(*backbone_arch*, *out_features*, *pretrained=True*, *pos_out_inds=None*, *prob_out_inds=None*)

static __new__(*cls*, **args: Any*, ***kwargs: Any*) → Any

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
Any

forward(*x*)

Configs

RegressionDataConfig	
RegressionGeoDataConfig	Configure regression *GeoDatasets*.
RegressionImageDataConfig	Configure *RegressionImageDatasets*.
RegressionLearnerConfig	Configure a *RegressionLearner*.
RegressionModelConfig	Configure a regression model.
RegressionPlotOptions	

RegressionDataConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model RegressionDataConfig


```
{
   "title": "RegressionDataConfig",
   "description": "Base class that can be extended to provide custom configurations.
↪\n\nThis adds some extra methods to Pydantic BaseModel.\nSee https://pydantic-
↪docs.helpmanual.io/\n\nThe general idea is that configuration schemas can be␣
↪defined by\nsubclassing this and adding class attributes with types and\ndefault␣
↪values for each field. Configs can be defined hierarchically,\nie. a Config can␣
↪have fields which are of type Config.\nValidation, serialization, deserialization,
↪ and IDE support is\nprovided automatically based on this schema.",
   "type": "object",
   "properties": {
      "pos_class_names": {
         "title": "Pos Class Names",
         "default": [],
         "type": "array",
         "items": {
            "type": "string"
         }
      },
      "prob_class_names": {
         "title": "Prob Class Names",
         "default": [],
         "type": "array",
         "items": {
            "type": "string"
         }
      },
      "type_hint": {
         "title": "Type Hint",
         "default": "regression_data",
         "enum": [
            "regression_data"
         ],
```


(continues on next page)

(continued from previous page)

```
            "type": "string"
        }
    },
    "additionalProperties": false
}
```

Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *pos_class_names (List[str])*
- *prob_class_names (List[str])*
- *type_hint (Literal['regression_data'])*

field pos_class_names: List[str] = []

field prob_class_names: List[str] = []

field type_hint: Literal['regression_data'] = 'regression_data'

build()

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(**args*, ***kwargs*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

RegressionGeoDataConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model RegressionGeoDataConfig

Configure regression *GeoDatasets*.

See *rastervision.pytorch_learner.dataset.regression_dataset*.


```
{
   "title": "RegressionGeoDataConfig",
   "description": "Configure regression :class:`GeoDatasets <.GeoDataset>`.\n\nSee :
↪mod:`rastervision.pytorch_learner.dataset.regression_dataset`.",
   "type": "object",
   "properties": {
      "class_names": {
         "title": "Class Names",
         "description": "Names of classes.",
         "default": [],
         "type": "array",
         "items": {
            "type": "string"
         }
      },
      "class_colors": {
         "title": "Class Colors",
         "description": "Colors used to display classes. Can be color 3-tuples in␣
↪list form.",
         "type": "array",
         "items": {
            "anyOf": [
               {
                  "type": "string"
               },
               {
                  "type": "array",
                  "minItems": 3,
                  "maxItems": 3,
                  "items": [
                     {
                        "type": "integer"
                     },
                     {
                        "type": "integer"
                     },
```


(continues on next page)

(continued from previous page)

```
                    {
                        "type": "integer"
                    }
                ]
            }
        ]
    }
},
"img_channels": {
    "title": "Img Channels",
    "description": "The number of channels of the training images.",
    "exclusiveMinimum": 0,
    "type": "integer"
},
"img_sz": {
    "title": "Img Sz",
    "description": "Length of a side of each image in pixels. This is the size␣
↪to transform it to during training, not the size in the raw dataset.",
    "default": 256,
    "exclusiveMinimum": 0,
    "type": "integer"
},
"train_sz": {
    "title": "Train Sz",
    "description": "If set, the number of training images to use. If fewer␣
↪images exist, then an exception will be raised.",
    "type": "integer"
},
"train_sz_rel": {
    "title": "Train Sz Rel",
    "description": "If set, the proportion of training images to use.",
    "type": "number"
},
"num_workers": {
    "title": "Num Workers",
    "description": "Number of workers to use when DataLoader makes batches.",
    "default": 4,
    "type": "integer"
},
"augmentors": {
    "title": "Augmentors",
    "description": "Names of albumentations augmentors to use for training␣
↪batches. Choices include: ['Blur', 'RandomRotate90', 'HorizontalFlip',
↪'VerticalFlip', 'GaussianBlur', 'GaussNoise', 'RGBShift', 'ToGray'].␣
↪Alternatively, a custom transform can be provided via the aug_transform option.",
    "default": [
        "RandomRotate90",
        "HorizontalFlip",
        "VerticalFlip"
    ],
    "type": "array",
    "items": {
```

(continues on next page)

(continued from previous page)


```
            "type": "string"
         }
      },
      "base_transform": {
         "title": "Base Transform",
         "description": "An Albumentations transform serialized as a dict that will␣
↪be applied to all datasets: training, validation, and test. This transformation␣
↪is in addition to the resizing due to img_sz. This is useful for, for example,␣
↪applying the same normalization to all datasets.",
         "type": "object"
      },
      "aug_transform": {
         "title": "Aug Transform",
         "description": "An Albumentations transform serialized as a dict that will␣
↪be applied as data augmentation to the training dataset. This transform is␣
↪applied before base_transform. If provided, the augmentors option is ignored.",
         "type": "object"
      },
      "plot_options": {
         "title": "Plot Options",
         "description": "Options to control plotting.",
         "default": {
            "transform": {
               "__version__": "1.3.0",
               "transform": {
                  "__class_fullname__": "rastervision.pytorch_learner.utils.utils.
↪MinMaxNormalize",
                  "always_apply": false,
                  "p": 1.0,
                  "min_val": 0.0,
                  "max_val": 1.0,
                  "dtype": 5
               }
            },
            "channel_display_groups": null,
            "type_hint": "regression_plot_options",
            "max_scatter_points": 5000,
            "hist_bins": 30
         },
         "allOf": [
            {
               "$ref": "#/definitions/RegressionPlotOptions"
            }
         ]
      },
      "preview_batch_limit": {
         "title": "Preview Batch Limit",
         "description": "Optional limit on the number of items in the preview plots␣
↪produced during training.",
         "type": "integer"
      },
      "type_hint": {
```


(continues on next page)

(continued from previous page)


```
        "title": "Type Hint",
        "default": "regression_geo_data",
        "enum": [
          "regression_geo_data"
        ],
        "type": "string"
      },
      "scene_dataset": {
        "$ref": "#/definitions/DatasetConfig"
      },
      "window_opts": {
        "title": "Window Opts",
        "default": {},
        "anyOf": [
          {
            "$ref": "#/definitions/GeoDataWindowConfig"
          },
          {
            "type": "object",
            "additionalProperties": {
              "$ref": "#/definitions/GeoDataWindowConfig"
            }
          }
        ]
      },
      "pos_class_names": {
        "title": "Pos Class Names",
        "default": [],
        "type": "array",
        "items": {
          "type": "string"
        }
      },
      "prob_class_names": {
        "title": "Prob Class Names",
        "default": [],
        "type": "array",
        "items": {
          "type": "string"
        }
      }
    },
    "additionalProperties": false,
    "definitions": {
      "RegressionPlotOptions": {
        "title": "RegressionPlotOptions",
        "description": "Config related to plotting.",
        "type": "object",
        "properties": {
          "transform": {
            "title": "Transform",
            "description": "An Albumentations transform serialized as a dict␣
```


(continues on next page)

(continued from previous page)

```
↪that will be applied to each image before it is plotted. Mainly useful for␣
↪undoing any data transformation that you do not want included in the plot, such␣
↪as normalization. The default value will shift and scale the image so the values␣
↪range from 0.0 to 1.0 which is the expected range for the plotting function. This␣
↪default is useful for cases where the values after normalization are close to␣
↪zero which makes the plot difficult to see.",
                "default": {
                    "__version__": "1.3.0",
                    "transform": {
                        "__class_fullname__": "rastervision.pytorch_learner.utils.
↪utils.MinMaxNormalize",
                        "always_apply": false,
                        "p": 1.0,
                        "min_val": 0.0,
                        "max_val": 1.0,
                        "dtype": 5
                    }
                },
                "type": "object"
            },
            "channel_display_groups": {
                "title": "Channel Display Groups",
                "description": "Groups of image channels to display together as a␣
↪subplot when plotting the data and predictions. Can be a list or tuple of groups␣
↪(e.g. [(0, 1, 2), (3,)]) or a dict containing title-to-group mappings (e.g. {\
↪"RGB\": [0, 1, 2], \"IR\": [3]}), where each group is a list or tuple of channel␣
↪indices and title is a string that will be used as the title of the subplot for␣
↪that group.",
                "anyOf": [
                    {
                        "type": "object",
                        "additionalProperties": {
                            "type": "array",
                            "items": {
                                "type": "integer",
                                "minimum": 0
                            }
                        }
                    },
                    {
                        "type": "array",
                        "items": {
                            "type": "array",
                            "items": {
                                "type": "integer",
                                "minimum": 0
                            }
                        }
                    }
                ]
            },
            "type_hint": {
```

(continues on next page)

(continued from previous page)

```
                "title": "Type Hint",
                "default": "regression_plot_options",
                "enum": [
                   "regression_plot_options"
                ],
                "type": "string"
             },
             "max_scatter_points": {
                "title": "Max Scatter Points",
                "description": "Maximum number of datapoints to use in scatter plot.␣
↪Useful to avoid running out of memory and cluttering.",
                "default": 5000,
                "type": "integer"
             },
             "hist_bins": {
                "title": "Hist Bins",
                "description": "Number of bins to use for histogram.",
                "default": 30,
                "type": "integer"
             }
          },
          "additionalProperties": false
       },
       "ClassConfig": {
          "title": "ClassConfig",
          "description": "Configure class information for a machine learning task.",
          "type": "object",
          "properties": {
             "names": {
                "title": "Names",
                "description": "Names of classes. The i-th class in this list will␣
↪have class ID = i.",
                "type": "array",
                "items": {
                   "type": "string"
                }
             },
             "colors": {
                "title": "Colors",
                "description": "Colors used to visualize classes. Can be color␣
↪strings accepted by matplotlib or RGB tuples. If None, a random color will be␣
↪auto-generated for each class.",
                "type": "array",
                "items": {
                   "anyOf": [
                      {
                         "type": "string"
                      },
                      {
                         "type": "array",
                         "items": {}
                      }
```

(continues on next page)

(continued from previous page)


```
                    ]
                }
            },
            "null_class": {
                "title": "Null Class",
                "description": "Optional name of class in `names` to use as the null␣
↪class. This is used in semantic segmentation to represent the label for imagery␣
↪pixels that are NODATA or that are missing a label. If None and the class names␣
↪include \"null\", it will automatically be used as the null class. If None, and␣
↪this Config is part of a SemanticSegmentationConfig, a null class will be added␣
↪automatically.",
                "type": "string"
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "class_config",
                "enum": [
                    "class_config"
                ],
                "type": "string"
            }
        },
        "required": [
            "names"
        ],
        "additionalProperties": false
    },
    "RasterTransformerConfig": {
        "title": "RasterTransformerConfig",
        "description": "Configure a :class:`.RasterTransformer`.",
        "type": "object",
        "properties": {
            "type_hint": {
                "title": "Type Hint",
                "default": "raster_transformer",
                "enum": [
                    "raster_transformer"
                ],
                "type": "string"
            }
        },
        "additionalProperties": false
    },
    "RasterSourceConfig": {
        "title": "RasterSourceConfig",
        "description": "Configure a :class:`.RasterSource`.",
        "type": "object",
        "properties": {
            "channel_order": {
                "title": "Channel Order",
                "description": "The sequence of channel indices to use when reading␣
↪imagery.",
```


(continues on next page)

(continued from previous page)


```
                "type": "array",
                "items": {
                    "type": "integer"
                }
            },
            "transformers": {
                "title": "Transformers",
                "default": [],
                "type": "array",
                "items": {
                    "$ref": "#/definitions/RasterTransformerConfig"
                }
            },
            "extent": {
                "title": "Extent",
                "description": "Use-specified extent in pixel coords in the form␣
↪(ymin, xmin, ymax, xmax). Useful for cropping the raster source so that only part␣
↪of the raster is read from.",
                "type": "array",
                "minItems": 4,
                "maxItems": 4,
                "items": [
                    {
                        "type": "integer"
                    },
                    {
                        "type": "integer"
                    },
                    {
                        "type": "integer"
                    },
                    {
                        "type": "integer"
                    }
                ]
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "raster_source",
                "enum": [
                    "raster_source"
                ],
                "type": "string"
            }
        },
        "additionalProperties": false
    },
    "LabelSourceConfig": {
        "title": "LabelSourceConfig",
        "description": "Configure a :class:`.LabelSource`.",
        "type": "object",
        "properties": {
```


(continues on next page)

(continued from previous page)


```
            "type_hint": {
               "title": "Type Hint",
               "default": "label_source",
               "enum": [
                  "label_source"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      },
      "LabelStoreConfig": {
         "title": "LabelStoreConfig",
         "description": "Configure a :class:`.LabelStore`.",
         "type": "object",
         "properties": {
            "type_hint": {
               "title": "Type Hint",
               "default": "label_store",
               "enum": [
                  "label_store"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      },
      "SceneConfig": {
         "title": "SceneConfig",
         "description": "Configure a :class:`.Scene` comprising raster data &␣
↪labels for an AOI.\n    ",
         "type": "object",
         "properties": {
            "id": {
               "title": "Id",
               "type": "string"
            },
            "raster_source": {
               "$ref": "#/definitions/RasterSourceConfig"
            },
            "label_source": {
               "$ref": "#/definitions/LabelSourceConfig"
            },
            "label_store": {
               "$ref": "#/definitions/LabelStoreConfig"
            },
            "aoi_uris": {
               "title": "Aoi Uris",
               "description": "List of URIs of GeoJSON files that define the AOIs␣
↪for the scene. Each polygon defines an AOI which is a piece of the scene that is␣
↪assumed to be fully labeled and usable for training or validation. The AOIs are␣
↪assumed to be in EPSG:4326 coordinates.",
```


(continues on next page)

(continued from previous page)


```
                "type": "array",
                "items": {
                   "type": "string"
                }
             },
             "type_hint": {
                "title": "Type Hint",
                "default": "scene",
                "enum": [
                   "scene"
                ],
                "type": "string"
             }
          },
          "required": [
             "id",
             "raster_source"
          ],
          "additionalProperties": false
       },
       "DatasetConfig": {
          "title": "DatasetConfig",
          "description": "Configure train, validation, and test splits for a dataset.
↪",
          "type": "object",
          "properties": {
             "class_config": {
                "$ref": "#/definitions/ClassConfig"
             },
             "train_scenes": {
                "title": "Train Scenes",
                "type": "array",
                "items": {
                   "$ref": "#/definitions/SceneConfig"
                }
             },
             "validation_scenes": {
                "title": "Validation Scenes",
                "type": "array",
                "items": {
                   "$ref": "#/definitions/SceneConfig"
                }
             },
             "test_scenes": {
                "title": "Test Scenes",
                "default": [],
                "type": "array",
                "items": {
                   "$ref": "#/definitions/SceneConfig"
                }
             },
             "scene_groups": {
```


(continues on next page)

(continued from previous page)

```
                "title": "Scene Groups",
                "description": "Groupings of scenes. Should be a dict of the form: {
↪<group-name>: Set(scene_id_1, scene_id_2, ...)}. Three groups are added by␣
↪default: \"train_scenes\", \"validation_scenes\", and \"test_scenes\"",
                "default": {},
                "type": "object",
                "additionalProperties": {
                   "type": "array",
                   "items": {
                      "type": "string"
                   },
                   "uniqueItems": true
                }
             },
             "type_hint": {
                "title": "Type Hint",
                "default": "dataset",
                "enum": [
                   "dataset"
                ],
                "type": "string"
             }
          },
          "required": [
             "class_config",
             "train_scenes",
             "validation_scenes"
          ],
          "additionalProperties": false
       },
       "GeoDataWindowMethod": {
          "title": "GeoDataWindowMethod",
          "description": "An enumeration.",
          "enum": [
             "sliding",
             "random"
          ]
       },
       "GeoDataWindowConfig": {
          "title": "GeoDataWindowConfig",
          "description": "Configure a :class:`.GeoDataset`.\n\nSee :mod:
↪`rastervision.pytorch_learner.dataset.dataset`.",
          "type": "object",
          "properties": {
             "method": {
                "default": "sliding",
                "allOf": [
                   {
                      "$ref": "#/definitions/GeoDataWindowMethod"
                   }
                ]
             },
```

(continues on next page)

(continued from previous page)

```
            "size": {
               "title": "Size",
               "description": "If method = sliding, this is the size of sliding␣
↪window. If method = random, this is the size that all the windows are resized to␣
↪before they are returned. If method = random and neither size_lims nor h_lims and␣
↪w_lims have been specified, then size_lims is set to (size, size + 1).",
               "anyOf": [
                  {
                     "type": "integer",
                     "exclusiveMinimum": 0
                  },
                  {
                     "type": "array",
                     "minItems": 2,
                     "maxItems": 2,
                     "items": [
                        {
                           "type": "integer",
                           "exclusiveMinimum": 0
                        },
                        {
                           "type": "integer",
                           "exclusiveMinimum": 0
                        }
                     ]
                  }
               ]
            },
            "stride": {
               "title": "Stride",
               "description": "Stride of sliding window. Only used if method =␣
↪sliding.",
               "anyOf": [
                  {
                     "type": "integer",
                     "exclusiveMinimum": 0
                  },
                  {
                     "type": "array",
                     "minItems": 2,
                     "maxItems": 2,
                     "items": [
                        {
                           "type": "integer",
                           "exclusiveMinimum": 0
                        },
                        {
                           "type": "integer",
                           "exclusiveMinimum": 0
                        }
                     ]
                  }
```

(continues on next page)

(continued from previous page)

```
            ]
         },
         "padding": {
            "title": "Padding",
            "description": "How many pixels are windows allowed to overflow the␣
↪edges of the raster source.",
            "anyOf": [
               {
                  "type": "integer",
                  "minimum": 0
               },
               {
                  "type": "array",
                  "minItems": 2,
                  "maxItems": 2,
                  "items": [
                     {
                        "type": "integer",
                        "minimum": 0
                     },
                     {
                        "type": "integer",
                        "minimum": 0
                     }
                  ]
               }
            ]
         },
         "pad_direction": {
            "title": "Pad Direction",
            "description": "If \"end\", only pad ymax and xmax (bottom and␣
↪right). If \"start\", only pad ymin and xmin (top and left). If \"both\", pad all␣
↪sides. Has no effect if paddiong is zero. Defaults to \"end\".",
            "default": "end",
            "enum": [
               "both",
               "start",
               "end"
            ],
            "type": "string"
         },
         "size_lims": {
            "title": "Size Lims",
            "description": "[min, max) interval from which window sizes will be␣
↪uniformly randomly sampled. The upper limit is exclusive. To fix the size to a␣
↪constant value, use size_lims = (sz, sz + 1). Only used if method = random.␣
↪Specify either size_lims or h_lims and w_lims, but not both. If neither size_lims␣
↪nor h_lims and w_lims have been specified, then this will be set to (size, size +␣
↪1).",
            "type": "array",
            "minItems": 2,
            "maxItems": 2,
```

(continues on next page)

(continued from previous page)

```
                "items": [
                    {
                        "type": "integer",
                        "exclusiveMinimum": 0
                    },
                    {
                        "type": "integer",
                        "exclusiveMinimum": 0
                    }
                ]
            },
            "h_lims": {
                "title": "H Lims",
                "description": "[min, max] interval from which window heights will␣
↪be uniformly randomly sampled. Only used if method = random.",
                "type": "array",
                "minItems": 2,
                "maxItems": 2,
                "items": [
                    {
                        "type": "integer",
                        "exclusiveMinimum": 0
                    },
                    {
                        "type": "integer",
                        "exclusiveMinimum": 0
                    }
                ]
            },
            "w_lims": {
                "title": "W Lims",
                "description": "[min, max] interval from which window widths will be␣
↪uniformly randomly sampled. Only used if method = random.",
                "type": "array",
                "minItems": 2,
                "maxItems": 2,
                "items": [
                    {
                        "type": "integer",
                        "exclusiveMinimum": 0
                    },
                    {
                        "type": "integer",
                        "exclusiveMinimum": 0
                    }
                ]
            },
            "max_windows": {
                "title": "Max Windows",
                "description": "Max allowed reads from a GeoDataset. Only used if␣
↪method = random.",
                "default": 10000,
```

(continues on next page)

(continued from previous page)


```
                "minimum": 0,
                "type": "integer"
            },
            "max_sample_attempts": {
                "title": "Max Sample Attempts",
                "description": "Max attempts when trying to find a window within the␣
↪AOI of a scene. Only used if method = random and the scene has aoi_polygons␣
↪specified.",
                "default": 100,
                "exclusiveMinimum": 0,
                "type": "integer"
            },
            "efficient_aoi_sampling": {
                "title": "Efficient Aoi Sampling",
                "description": "If the scene has AOIs, sampling windows at random␣
↪anywhere in the extent and then checking if they fall within any of the AOIs can␣
↪be very inefficient. This flag enables the use of an alternate algorithm that␣
↪only samples window locations inside the AOIs. Only used if method = random and␣
↪the scene has aoi_polygons specified. Defaults to True",
                "default": true,
                "type": "boolean"
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "geo_data_window",
                "enum": [
                    "geo_data_window"
                ],
                "type": "string"
            }
        },
        "required": [
            "size"
        ],
        "additionalProperties": false
    }
  }
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *aug_transform (Optional[dict])*
- *augmentors (List[str])*
- *base_transform (Optional[dict])*
- *class_colors (Optional[List[Union[str, Tuple[int, int, int]]]])*
- *class_names (List[str])*

- *img_channels (Optional[pydantic.types.PositiveInt])*
- *img_sz (pydantic.types.PositiveInt)*
- *num_workers (int)*
- *plot_options (Optional[rastervision.pytorch_learner.regression_learner_config.RegressionPlotOptions])*
- *pos_class_names (List[str])*
- *preview_batch_limit (Optional[int])*
- *prob_class_names (List[str])*
- *scene_dataset (Optional[rastervision.core.data.dataset_config.DatasetConfig])*
- *train_sz (Optional[int])*
- *train_sz_rel (Optional[float])*
- *type_hint (Literal['regression_geo_data'])*
- *window_opts (Union[rastervision.pytorch_learner.learner_config.GeoDataWindowConfig, Dict[str, rastervision.pytorch_learner.learner_config.GeoDataWindowConfig]])*

Validators

- ensure_class_colors » all fields
- get_class_info_from_class_config_if_needed » all fields
- *validate_albumentation_transform* » *aug_transform*
- *validate_albumentation_transform* » *base_transform*
- validate_augmentors » *augmentors*
- validate_plot_options » all fields
- validate_window_opts » *window_opts*

field aug_transform: Optional[dict] = None

An Albumentations transform serialized as a dict that will be applied as data augmentation to the training dataset. This transform is applied before base_transform. If provided, the augmentors option is ignored.

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- *validate_albumentation_transform*
- validate_plot_options

field augmentors: List[str] = ['RandomRotate90', 'HorizontalFlip', 'VerticalFlip']

Names of albumentations augmentors to use for training batches. Choices include: [‘Blur’, ‘RandomRotate90’, ‘HorizontalFlip’, ‘VerticalFlip’, ‘GaussianBlur’, ‘GaussNoise’, ‘RGBShift’, ‘ToGray’]. Alternatively, a custom transform can be provided via the aug_transform option.

Validated by

- ensure_class_colors

- get_class_info_from_class_config_if_needed
- validate_augmentors
- validate_plot_options

field base_transform: Optional[dict] = None

An Albumentations transform serialized as a dict that will be applied to all datasets: training, validation, and test. This transformation is in addition to the resizing due to img_sz. This is useful for, for example, applying the same normalization to all datasets.

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- *validate_albumentation_transform*
- validate_plot_options

field class_colors: Optional[List[Union[str, RGBTuple]]] = None

Colors used to display classes. Can be color 3-tuples in list form.

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field class_names: List[str] = []

Names of classes.

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field img_channels: Optional[PosInt] = None

The number of channels of the training images.

Constraints

- **exclusiveMinimum** = 0

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field img_sz: PosInt = 256

Length of a side of each image in pixels. This is the size to transform it to during training, not the size in the raw dataset.

Constraints

- **exclusiveMinimum** = 0

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field num_workers: int = 4

Number of workers to use when DataLoader makes batches.

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field plot_options: Optional[*RegressionPlotOptions*] = RegressionPlotOptions(transform={'__version__': '1.3.0', 'transform': {'__class_fullname__': 'rastervision.pytorch_learner.utils.utils.MinMaxNormalize', 'always_apply': False, 'p': 1.0, 'min_val': 0.0, 'max_val': 1.0, 'dtype': 5}}, channel_display_groups=None, max_scatter_points=5000, hist_bins=30)

Options to control plotting.

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field pos_class_names: List[str] = []

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field preview_batch_limit: Optional[int] = None

Optional limit on the number of items in the preview plots produced during training.

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field prob_class_names: List[str] = []

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field scene_dataset: Optional['SceneDatasetConfig'] = None

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field train_sz: Optional[int] = None

If set, the number of training images to use. If fewer images exist, then an exception will be raised.

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field train_sz_rel: Optional[float] = None

If set, the proportion of training images to use.

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field type_hint: Literal['regression_geo_data'] = 'regression_geo_data'

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field window_opts: Union[*GeoDataWindowConfig*, Dict[str, *GeoDataWindowConfig*]] = {}

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options
- validate_window_opts

build(*tmp_dir: str*, *overfit_mode: bool = False*, *test_mode: bool = False*) → Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

Parameters

- **tmp_dir** (*str*) –
- **overfit_mode** (*bool*) –
- **test_mode** (*bool*) –

Return type
Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

build_scenes(*tmp_dir: str*) → Tuple[List[*Scene*], List[*Scene*], List[*Scene*]]

Build training, validation, and test scenes.

Parameters
tmp_dir (*str*) –

Return type
Tuple[*List*[Scene], *List*[Scene], *List*[Scene]]

validator ensure_class_colors » *all fields*

Parameters
values (*dict*) –

Return type
dict

get_bbox_params() → Optional[BboxParams]

Returns BboxParams used by albumentations for data augmentation.

Return type
Optional[*BboxParams*]

validator get_class_info_from_class_config_if_needed » *all fields*

Parameters
values (*dict*) –

Return type
dict

get_custom_albumentations_transforms() → List[dict]

Returns all custom transforms found in this config.

This should return all serialized albumentations transforms with a 'lambda_transforms_path' field contained in this config or in any of its members no matter how deeply neseted.

The pupose is to make it easier to adjust their paths all at once while saving to or loading from a bundle.

Return type
List[dict]

get_data_transforms() → Tuple[BasicTransform, BasicTransform]

Get albumentations transform objects for data augmentation.

Returns
a transform that doesn't do any data augmentation 2nd tuple arg: a transform with data augmentation

Return type
1st tuple arg

make_datasets(*tmp_dir: str*, *train_tf: Optional[BasicTransform] = None*, *val_tf: Optional[BasicTransform] = None*, *test_tf: Optional[BasicTransform] = None*, ***kwargs*) → Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

Make training, validation, and test datasets.

Parameters

- **tmp_dir** (*str*) – Temporary directory to be used for building scenes.

- **train_tf** (*Optional[A.BasicTransform], optional*) – Transform for the training dataset. Defaults to None.
- **val_tf** (*Optional[A.BasicTransform], optional*) – Transform for the validation dataset. Defaults to None.
- **test_tf** (*Optional[A.BasicTransform], optional*) – Transform for the test dataset. Defaults to None.
- **kwargs** – Kwargs to pass to self.scene_to_dataset()

Returns

PyTorch-compatiable training,
validation, and test datasets.

Return type
Tuple[Dataset, Dataset, Dataset]

random_subset_dataset(*ds: torch.utils.data.Dataset*, *size: Optional[int] = None*, *fraction: Optional[ConstrainedFloatValue] = None*) → torch.utils.data.Subset

Parameters

- **ds** (*torch.utils.data.Dataset*) –
- **size** (*Optional[int]*) –
- **fraction** (*Optional[ConstrainedFloatValue]*) –

Return type
torch.utils.data.Subset

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

scene_to_dataset(*scene: Scene*, *transform: Optional[BasicTransform] = None*) → Union[*RegressionSlidingWindowGeoDataset*, *RegressionRandomWindowGeoDataset*]

Make a dataset from a single scene.

Parameters

- **scene** (Scene) –
- **transform** (*Optional[BasicTransform]*) –

Return type
Union[RegressionSlidingWindowGeoDataset, RegressionRandomWindowGeoDataset]

update(**args*, ***kwargs*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validator validate_augmentors » augmentors

Parameters
v (*str*) –

Return type
str

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises
ConfigError – if field is invalid

validator validate_plot_options » *all fields*

Parameters
values (*dict*) –

Return type
dict

validator validate_window_opts » window_opts

Parameters

- **v** (*Union[GeoDataWindowConfig, Dict[str, GeoDataWindowConfig]]*) –
- **values** (*dict*) –

Return type
Union[GeoDataWindowConfig, *Dict*[str, GeoDataWindowConfig]]

property num_classes

RegressionImageDataConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model RegressionImageDataConfig

Configure *RegressionImageDatasets*.

```
{
   "title": "RegressionImageDataConfig",
   "description": "Configure :class:`RegressionImageDatasets <.
↪RegressionImageDataset>`.",
   "type": "object",
```

(continues on next page)

(continued from previous page)

```
   "properties": {
      "class_names": {
         "title": "Class Names",
         "description": "Names of classes.",
         "default": [],
         "type": "array",
         "items": {
            "type": "string"
         }
      },
      "class_colors": {
         "title": "Class Colors",
         "description": "Colors used to display classes. Can be color 3-tuples in␣
↪list form.",
         "type": "array",
         "items": {
            "anyOf": [
               {
                  "type": "string"
               },
               {
                  "type": "array",
                  "minItems": 3,
                  "maxItems": 3,
                  "items": [
                     {
                        "type": "integer"
                     },
                     {
                        "type": "integer"
                     },
                     {
                        "type": "integer"
                     }
                  ]
               }
            ]
         }
      },
      "img_channels": {
         "title": "Img Channels",
         "description": "The number of channels of the training images.",
         "exclusiveMinimum": 0,
         "type": "integer"
      },
      "img_sz": {
         "title": "Img Sz",
         "description": "Length of a side of each image in pixels. This is the size␣
↪to transform it to during training, not the size in the raw dataset.",
         "default": 256,
         "exclusiveMinimum": 0,
         "type": "integer"
```

(continues on next page)

(continued from previous page)


```
      },
      "train_sz": {
         "title": "Train Sz",
         "description": "If set, the number of training images to use. If fewer␣
↪images exist, then an exception will be raised.",
         "type": "integer"
      },
      "train_sz_rel": {
         "title": "Train Sz Rel",
         "description": "If set, the proportion of training images to use.",
         "type": "number"
      },
      "num_workers": {
         "title": "Num Workers",
         "description": "Number of workers to use when DataLoader makes batches.",
         "default": 4,
         "type": "integer"
      },
      "augmentors": {
         "title": "Augmentors",
         "description": "Names of albumentations augmentors to use for training␣
↪batches. Choices include: ['Blur', 'RandomRotate90', 'HorizontalFlip',
↪'VerticalFlip', 'GaussianBlur', 'GaussNoise', 'RGBShift', 'ToGray'].␣
↪Alternatively, a custom transform can be provided via the aug_transform option.",
         "default": [
            "RandomRotate90",
            "HorizontalFlip",
            "VerticalFlip"
         ],
         "type": "array",
         "items": {
            "type": "string"
         }
      },
      "base_transform": {
         "title": "Base Transform",
         "description": "An Albumentations transform serialized as a dict that will␣
↪be applied to all datasets: training, validation, and test. This transformation␣
↪is in addition to the resizing due to img_sz. This is useful for, for example,␣
↪applying the same normalization to all datasets.",
         "type": "object"
      },
      "aug_transform": {
         "title": "Aug Transform",
         "description": "An Albumentations transform serialized as a dict that will␣
↪be applied as data augmentation to the training dataset. This transform is␣
↪applied before base_transform. If provided, the augmentors option is ignored.",
         "type": "object"
      },
      "plot_options": {
         "title": "Plot Options",
         "description": "Options to control plotting.",
```


(continues on next page)

(continued from previous page)


```
         "default": {
            "transform": {
               "__version__": "1.3.0",
               "transform": {
                  "__class_fullname__": "rastervision.pytorch_learner.utils.utils.
↪MinMaxNormalize",
                  "always_apply": false,
                  "p": 1.0,
                  "min_val": 0.0,
                  "max_val": 1.0,
                  "dtype": 5
               }
            },
            "channel_display_groups": null,
            "type_hint": "regression_plot_options",
            "max_scatter_points": 5000,
            "hist_bins": 30
         },
         "allOf": [
            {
               "$ref": "#/definitions/RegressionPlotOptions"
            }
         ]
      },
      "preview_batch_limit": {
         "title": "Preview Batch Limit",
         "description": "Optional limit on the number of items in the preview plots␣
↪produced during training.",
         "type": "integer"
      },
      "type_hint": {
         "title": "Type Hint",
         "default": "regression_image_data",
         "enum": [
            "regression_image_data"
         ],
         "type": "string"
      },
      "data_format": {
         "default": "csv",
         "allOf": [
            {
               "$ref": "#/definitions/RegressionDataFormat"
            }
         ]
      },
      "uri": {
         "title": "Uri",
         "description": "One of the following:\n(1) a URI of a directory containing␣
↪\"train\", \"valid\", and (optinally) \"test\" subdirectories;\n(2) a URI of a␣
↪zip file containing (1);\n(3) a list of (2);\n(4) a URI of a directory containing␣
↪zip files containing (1).",
```


(continues on next page)

(continued from previous page)

```
         "anyOf": [
            {
               "type": "string"
            },
            {
               "type": "array",
               "items": {
                  "type": "string"
               }
            }
         ]
      },
      "group_uris": {
         "title": "Group Uris",
         "description": "This can be set instead of uri in order to specify groups␣
↪of chips. Each element in the list is expected to be an object of the same form␣
↪accepted by the uri field. The purpose of separating chips into groups is to be␣
↪able to use the group_train_sz field.",
         "type": "array",
         "items": {
            "anyOf": [
               {
                  "type": "string"
               },
               {
                  "type": "array",
                  "items": {
                     "type": "string"
                  }
               }
            ]
         }
      },
      "group_train_sz": {
         "title": "Group Train Sz",
         "description": "If group_uris is set, this can be used to specify the␣
↪number of chips to use per group. Only applies to training chips. This can either␣
↪be a single value that will be used for all groups or a list of values (one for␣
↪each group).",
         "anyOf": [
            {
               "type": "integer"
            },
            {
               "type": "array",
               "items": {
                  "type": "integer"
               }
            }
         ]
      },
      "group_train_sz_rel": {
```

(continues on next page)

(continued from previous page)

```
        "title": "Group Train Sz Rel",
        "description": "Relative version of group_train_sz. Must be a float in [0,␣
↪1]. If group_uris is set, this can be used to specify the proportion of the total␣
↪chips in each group to use per group. Only applies to training chips. This can␣
↪either be a single value that will be used for all groups or a list of values␣
↪(one for each group).",
        "anyOf": [
           {
              "type": "number",
              "minimum": 0,
              "maximum": 1
           },
           {
              "type": "array",
              "items": {
                 "type": "number",
                 "minimum": 0,
                 "maximum": 1
              }
           }
        ]
     },
     "pos_class_names": {
        "title": "Pos Class Names",
        "default": [],
        "type": "array",
        "items": {
           "type": "string"
        }
     },
     "prob_class_names": {
        "title": "Prob Class Names",
        "default": [],
        "type": "array",
        "items": {
           "type": "string"
        }
     }
  },
  "additionalProperties": false,
  "definitions": {
     "RegressionPlotOptions": {
        "title": "RegressionPlotOptions",
        "description": "Config related to plotting.",
        "type": "object",
        "properties": {
           "transform": {
              "title": "Transform",
              "description": "An Albumentations transform serialized as a dict␣
↪that will be applied to each image before it is plotted. Mainly useful for␣
↪undoing any data transformation that you do not want included in the plot, such␣
↪as normalization. The default value will shift and scale the image so the values␣
```

(continues on next page)

(continued from previous page)

```
↪range from 0.0 to 1.0 which is the expected range for the plotting function. This␣
↪default is useful for cases where the values after normalization are close to␣
↪zero which makes the plot difficult to see.",
               "default": {
                  "__version__": "1.3.0",
                  "transform": {
                     "__class_fullname__": "rastervision.pytorch_learner.utils.
↪utils.MinMaxNormalize",
                     "always_apply": false,
                     "p": 1.0,
                     "min_val": 0.0,
                     "max_val": 1.0,
                     "dtype": 5
                  }
               },
               "type": "object"
            },
            "channel_display_groups": {
               "title": "Channel Display Groups",
               "description": "Groups of image channels to display together as a␣
↪subplot when plotting the data and predictions. Can be a list or tuple of groups␣
↪(e.g. [(0, 1, 2), (3,)]) or a dict containing title-to-group mappings (e.g. {\
↪"RGB\": [0, 1, 2], \"IR\": [3]}), where each group is a list or tuple of channel␣
↪indices and title is a string that will be used as the title of the subplot for␣
↪that group.",
               "anyOf": [
                  {
                     "type": "object",
                     "additionalProperties": {
                        "type": "array",
                        "items": {
                           "type": "integer",
                           "minimum": 0
                        }
                     }
                  },
                  {
                     "type": "array",
                     "items": {
                        "type": "array",
                        "items": {
                           "type": "integer",
                           "minimum": 0
                        }
                     }
                  }
               ]
            },
            "type_hint": {
               "title": "Type Hint",
               "default": "regression_plot_options",
               "enum": [
```

(continues on next page)

(continued from previous page)


```
                    "regression_plot_options"
                ],
                "type": "string"
            },
            "max_scatter_points": {
                "title": "Max Scatter Points",
                "description": "Maximum number of datapoints to use in scatter plot.␣
↪Useful to avoid running out of memory and cluttering.",
                "default": 5000,
                "type": "integer"
            },
            "hist_bins": {
                "title": "Hist Bins",
                "description": "Number of bins to use for histogram.",
                "default": 30,
                "type": "integer"
            }
        },
        "additionalProperties": false
    },
    "RegressionDataFormat": {
        "title": "RegressionDataFormat",
        "description": "An enumeration.",
        "enum": [
            "csv"
        ]
    }
  }
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *aug_transform (Optional[dict])*
- *augmentors (List[str])*
- *base_transform (Optional[dict])*
- *class_colors (Optional[List[Union[str, Tuple[int, int, int]]]])*
- *class_names (List[str])*
- *data_format (rastervision.pytorch_learner.regression_learner_config.RegressionDataFormat)*
- *group_train_sz (Optional[Union[int, List[int]]])*
- *group_train_sz_rel (Optional[Union[rastervision.pytorch_learner.learner_config.ConstrainedFloatValue, List[rastervision.pytorch_learner.learner_config.ConstrainedFloatValue]]])*
- *group_uris (Optional[List[Union[str, List[str]]]])*

- *img_channels (Optional[pydantic.types.PositiveInt])*
- *img_sz (pydantic.types.PositiveInt)*
- *num_workers (int)*
- *plot_options (Optional[rastervision.pytorch_learner.regression_learner_config.RegressionPlotOptions])*
- *pos_class_names (List[str])*
- *preview_batch_limit (Optional[int])*
- *prob_class_names (List[str])*
- *train_sz (Optional[int])*
- *train_sz_rel (Optional[float])*
- *type_hint (Literal['regression_image_data'])*
- *uri (Optional[Union[str, List[str]]])*

Validators

- ensure_class_colors » all fields
- *validate_albumentation_transform* » *aug_transform*
- *validate_albumentation_transform* » *base_transform*
- validate_augmentors » *augmentors*
- validate_group_uris » all fields
- validate_plot_options » all fields

field aug_transform: Optional[dict] = None

An Albumentations transform serialized as a dict that will be applied as data augmentation to the training dataset. This transform is applied before base_transform. If provided, the augmentors option is ignored.

Validated by

- ensure_class_colors
- *validate_albumentation_transform*
- validate_group_uris
- validate_plot_options

field augmentors: List[str] = ['RandomRotate90', 'HorizontalFlip', 'VerticalFlip']

Names of albumentations augmentors to use for training batches. Choices include: [‘Blur’, ‘RandomRotate90’, ‘HorizontalFlip’, ‘VerticalFlip’, ‘GaussianBlur’, ‘GaussNoise’, ‘RGBShift’, ‘ToGray’]. Alternatively, a custom transform can be provided via the aug_transform option.

Validated by

- ensure_class_colors
- validate_augmentors
- validate_group_uris
- validate_plot_options

field base_transform: Optional[dict] = None

An Albumentations transform serialized as a dict that will be applied to all datasets: training, validation, and test. This transformation is in addition to the resizing due to img_sz. This is useful for, for example, applying the same normalization to all datasets.

Validated by

- ensure_class_colors
- *validate_albumentation_transform*
- validate_group_uris
- validate_plot_options

field class_colors: Optional[List[Union[str, RGBTuple]]] = None

Colors used to display classes. Can be color 3-tuples in list form.

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field class_names: List[str] = []

Names of classes.

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field data_format: *RegressionDataFormat* = RegressionDataFormat.csv

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field group_train_sz: Optional[Union[int, List[int]]] = None

If group_uris is set, this can be used to specify the number of chips to use per group. Only applies to training chips. This can either be a single value that will be used for all groups or a list of values (one for each group).

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field group_train_sz_rel: Optional[Union[Proportion, List[Proportion]]] = None

Relative version of group_train_sz. Must be a float in [0, 1]. If group_uris is set, this can be used to specify the proportion of the total chips in each group to use per group. Only applies to training chips. This can either be a single value that will be used for all groups or a list of values (one for each group).

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field group_uris: Optional[List[Union[str, List[str]]]] = None

This can be set instead of uri in order to specify groups of chips. Each element in the list is expected to be an object of the same form accepted by the uri field. The purpose of separating chips into groups is to be able to use the group_train_sz field.

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field img_channels: Optional[PosInt] = None

The number of channels of the training images.

Constraints

- **exclusiveMinimum** = 0

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field img_sz: PosInt = 256

Length of a side of each image in pixels. This is the size to transform it to during training, not the size in the raw dataset.

Constraints

- **exclusiveMinimum** = 0

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field num_workers: int = 4

Number of workers to use when DataLoader makes batches.

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field plot_options: Optional[*RegressionPlotOptions*] = RegressionPlotOptions(transform={'__version__': '1.3.0', 'transform': {'__class_fullname__': 'rastervision.pytorch_learner.utils.utils.MinMaxNormalize', 'always_apply': False, 'p': 1.0, 'min_val': 0.0, 'max_val': 1.0, 'dtype': 5}}, channel_display_groups=None, max_scatter_points=5000, hist_bins=30)

Options to control plotting.

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field pos_class_names: List[str] = []

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field preview_batch_limit: Optional[int] = None

Optional limit on the number of items in the preview plots produced during training.

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field prob_class_names: List[str] = []

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field train_sz: Optional[int] = None

If set, the number of training images to use. If fewer images exist, then an exception will be raised.

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field train_sz_rel: Optional[float] = None

If set, the proportion of training images to use.

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field type_hint: Literal['regression_image_data'] = 'regression_image_data'

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field uri: Optional[Union[str, List[str]]] = None

One of the following: (1) a URI of a directory containing "train", "valid", and (optinally) "test" subdirectories; (2) a URI of a zip file containing (1); (3) a list of (2); (4) a URI of a directory containing zip files containing (1).

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

build(*tmp_dir: str*, *overfit_mode: bool = False*, *test_mode: bool = False*) → Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

Parameters

- **tmp_dir** (*str*) –
- **overfit_mode** (*bool*) –
- **test_mode** (*bool*) –

Return type
Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

dir_to_dataset(*data_dir: str*, *transform: BasicTransform*) → *RegressionImageDataset*

Parameters

- **data_dir** (*str*) –
- **transform** (*BasicTransform*) –

Return type
RegressionImageDataset

validator ensure_class_colors » *all fields*

Parameters
values (*dict*) –

Return type
dict

get_bbox_params() → Optional[BboxParams]

Returns BboxParams used by albumentations for data augmentation.

Return type
Optional[*BboxParams*]

get_custom_albumentations_transforms() → List[dict]

Returns all custom transforms found in this config.

This should return all serialized albumentations transforms with a 'lambda_transforms_path' field contained in this config or in any of its members no matter how deeply neseted.

The pupose is to make it easier to adjust their paths all at once while saving to or loading from a bundle.

Return type
List[dict]

get_data_dirs(*uri: Union[str, List[str]]*, *unzip_dir: str*) → List[str]

Extract data dirs from uri.

Data dirs are directories containing "train", "valid", and (optinally) "test" subdirectories.

Parameters

- **uri** (*Union[str, List[str]]*) – a URI or a list of URIs of one of the following:

(1) a URI of a directory containing "train", "valid", and (optinally) "test" subdirectories

(2) a URI of a zip file containing (1)

(3) a list of (2)

(4) a URI of a directory containing zip files containing (1)

- **unzip_dir** (*str*) –

Returns
paths to directories that each contain contents of one zip file

Return type
List[str]

get_data_transforms() → Tuple[BasicTransform, BasicTransform]

Get albumentations transform objects for data augmentation.

Returns
a transform that doesn't do any data augmentation 2nd tuple arg: a transform with data augmentation

Return type
1st tuple arg

get_datasets_from_group_uris(*uris: Union[str, List[str]]*, *tmp_dir: str*, *group_train_sz: Optional[int] = None*, *group_train_sz_rel: Optional[float] = None*, *overfit_mode: bool = False*, *test_mode: bool = False*) → Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

Parameters

- **uris** (*Union[str, List[str]]*) –
- **tmp_dir** (*str*) –
- **group_train_sz** (*Optional[int]*) –
- **group_train_sz_rel** (*Optional[float]*) –
- **overfit_mode** (*bool*) –
- **test_mode** (*bool*) –

Return type
Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

get_datasets_from_uri(*uri: Union[str, List[str]]*, *tmp_dir: str*, *overfit_mode: bool = False*, *test_mode: bool = False*) → Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

Get image train, validation, & test datasets from a single zip file.

Parameters

- **uri** (*Union[str, List[str]]*) – Uri of a zip file containing the images.
- **tmp_dir** (*str*) –
- **overfit_mode** (*bool*) –
- **test_mode** (*bool*) –

Returns

Training, validation, and test
dataSets.

Return type
Tuple[Dataset, Dataset, Dataset]

make_datasets(*train_dirs: Iterable[str]*, *val_dirs: Iterable[str]*, *test_dirs: Iterable[str]*, *train_tf: Optional[BasicTransform] = None*, *val_tf: Optional[BasicTransform] = None*, *test_tf: Optional[BasicTransform] = None*) → Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

Make training, validation, and test datasets.

Parameters

- **train_dirs** (*str*) – Directories where training data is located.
- **val_dirs** (*str*) – Directories where validation data is located.
- **test_dirs** (*str*) – Directories where test data is located.
- **train_tf** (*Optional[A.BasicTransform], optional*) – Transform for the training dataset. Defaults to None.
- **val_tf** (*Optional[A.BasicTransform], optional*) – Transform for the validation dataset. Defaults to None.
- **test_tf** (*Optional[A.BasicTransform], optional*) – Transform for the test dataset. Defaults to None.

Returns

PyTorch-compatiable training,
validation, and test datasets.

Return type
Tuple[Dataset, Dataset, Dataset]

random_subset_dataset(*ds: torch.utils.data.Dataset*, *size: Optional[int] = None*, *fraction: Optional[ConstrainedFloatValue] = None*) → torch.utils.data.Subset

Parameters

- **ds** (*torch.utils.data.Dataset*) –
- **size** (*Optional[int]*) –

- **fraction** (*Optional[ConstrainedFloatValue]*) –

Return type
torch.utils.data.Subset

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

unzip_data(*zip_uris: List[str]*, *unzip_dir: str*) → List[str]

Unzip dataset zip files.

Parameters

- **zip_uris** (*List[str]*) – a list of URIs of zip files:
- **unzip_dir** (*str*) – directory where zip files will be extrated to.

Returns
paths to directories that each contain contents of one zip file

Return type
List[str]

update(**args*, ***kwargs*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validator validate_augmentors » augmentors

Parameters
v (*str*) –

Return type
str

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validator validate_group_uris » *all fields*

Parameters
values (*dict*) –

Return type
dict

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises
ConfigError – if field is invalid

validator validate_plot_options » *all fields*

Parameters
values (*dict*) –

Return type
dict

property num_classes

RegressionLearnerConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model RegressionLearnerConfig

Configure a *RegressionLearner*.


```
{
   "title": "RegressionLearnerConfig",
   "description": "Configure a :class:`.RegressionLearner`.",
   "type": "object",
   "properties": {
      "model": {
         "$ref": "#/definitions/RegressionModelConfig"
      },
      "solver": {
         "$ref": "#/definitions/SolverConfig"
      },
      "data": {
         "title": "Data",
         "anyOf": [
            {
               "$ref": "#/definitions/RegressionImageDataConfig"
            },
            {
               "$ref": "#/definitions/RegressionGeoDataConfig"
            }
         ]
      },
      "predict_mode": {
         "title": "Predict Mode",
         "description": "If True, skips training, loads model, and does final eval.
↪",
         "default": false,
         "type": "boolean"
      },
```


(continues on next page)

(continued from previous page)

```
      "test_mode": {
         "title": "Test Mode",
         "description": "If True, uses test_num_epochs, test_batch_sz, truncated␣
↪datasets with only a single batch, image_sz that is cut in half, and num_workers␣
↪= 0. This is useful for testing that code runs correctly on CPU without␣
↪multithreading before running full job on GPU.",
         "default": false,
         "type": "boolean"
      },
      "overfit_mode": {
         "title": "Overfit Mode",
         "description": "If True, uses half image size, and instead of doing epoch-
↪based training, optimizes the model using a single batch repeatedly for overfit_
↪num_steps number of steps.",
         "default": false,
         "type": "boolean"
      },
      "eval_train": {
         "title": "Eval Train",
         "description": "If True, runs final evaluation on training set (in␣
↪addition to test set). Useful for debugging.",
         "default": false,
         "type": "boolean"
      },
      "save_model_bundle": {
         "title": "Save Model Bundle",
         "description": "If True, saves a model bundle at the end of training which␣
↪is zip file with model and this LearnerConfig which can be used to make␣
↪predictions on new images at a later time.",
         "default": true,
         "type": "boolean"
      },
      "log_tensorboard": {
         "title": "Log Tensorboard",
         "description": "Save Tensorboard log files at the end of each epoch.",
         "default": true,
         "type": "boolean"
      },
      "run_tensorboard": {
         "title": "Run Tensorboard",
         "description": "run Tensorboard server during training",
         "default": false,
         "type": "boolean"
      },
      "output_uri": {
         "title": "Output Uri",
         "description": "URI of where to save output",
         "type": "string"
      },
      "type_hint": {
         "title": "Type Hint",
         "default": "regression_learner",
```

(continues on next page)

(continued from previous page)

```
        "enum": [
           "regression_learner"
        ],
        "type": "string"
     }
  },
  "required": [
     "solver",
     "data"
  ],
  "additionalProperties": false,
  "definitions": {
     "Backbone": {
        "title": "Backbone",
        "description": "An enumeration.",
        "enum": [
           "alexnet",
           "densenet121",
           "densenet169",
           "densenet201",
           "densenet161",
           "googlenet",
           "inception_v3",
           "mnasnet0_5",
           "mnasnet0_75",
           "mnasnet1_0",
           "mnasnet1_3",
           "mobilenet_v2",
           "resnet18",
           "resnet34",
           "resnet50",
           "resnet101",
           "resnet152",
           "resnext50_32x4d",
           "resnext101_32x8d",
           "wide_resnet50_2",
           "wide_resnet101_2",
           "shufflenet_v2_x0_5",
           "shufflenet_v2_x1_0",
           "shufflenet_v2_x1_5",
           "shufflenet_v2_x2_0",
           "squeezenet1_0",
           "squeezenet1_1",
           "vgg11",
           "vgg11_bn",
           "vgg13",
           "vgg13_bn",
           "vgg16",
           "vgg16_bn",
           "vgg19_bn",
           "vgg19"
        ]
```

(continues on next page)

(continued from previous page)


```
      },
      "ExternalModuleConfig": {
         "title": "ExternalModuleConfig",
         "description": "Config describing an object to be loaded via Torch Hub.",
         "type": "object",
         "properties": {
            "uri": {
               "title": "Uri",
               "description": "Local uri of a zip file, or local uri of a directory,
↪or remote uri of zip file.",
               "minLength": 1,
               "type": "string"
            },
            "github_repo": {
               "title": "Github Repo",
               "description": "<repo-owner>/<repo-name>[:tag]",
               "pattern": ".+/.+",
               "type": "string"
            },
            "name": {
               "title": "Name",
               "description": "Name of the folder in which to extract/copy the␣
↪definition files.",
               "minLength": 1,
               "type": "string"
            },
            "entrypoint": {
               "title": "Entrypoint",
               "description": "Name of a callable present in hubconf.py. See docs␣
↪for torch.hub for details.",
               "minLength": 1,
               "type": "string"
            },
            "entrypoint_args": {
               "title": "Entrypoint Args",
               "description": "Args to pass to the entrypoint. Must be serializable.
↪",
               "default": [],
               "type": "array",
               "items": {}
            },
            "entrypoint_kwargs": {
               "title": "Entrypoint Kwargs",
               "description": "Keyword args to pass to the entrypoint. Must be␣
↪serializable.",
               "default": {},
               "type": "object"
            },
            "force_reload": {
               "title": "Force Reload",
               "description": "Force reload of module definition.",
               "default": false,
```


(continues on next page)

(continued from previous page)


```
                "type": "boolean"
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "external-module",
                "enum": [
                    "external-module"
                ],
                "type": "string"
            }
        },
        "required": [
            "entrypoint"
        ],
        "additionalProperties": false
    },
    "RegressionModelConfig": {
        "title": "RegressionModelConfig",
        "description": "Configure a regression model.",
        "type": "object",
        "properties": {
            "backbone": {
                "description": "The torchvision.models backbone to use.",
                "default": "resnet18",
                "allOf": [
                    {
                        "$ref": "#/definitions/Backbone"
                    }
                ]
            },
            "pretrained": {
                "title": "Pretrained",
                "description": "If True, use ImageNet weights. If False, use random␣
↪initialization.",
                "default": true,
                "type": "boolean"
            },
            "init_weights": {
                "title": "Init Weights",
                "description": "URI of PyTorch model weights used to initialize␣
↪model. If set, this supercedes the pretrained option.",
                "type": "string"
            },
            "load_strict": {
                "title": "Load Strict",
                "description": "If True, the keys in the state dict referenced by␣
↪init_weights must match exactly. Setting this to False can be useful if you just␣
↪want to load the backbone of a model.",
                "default": true,
                "type": "boolean"
            },
            "external_def": {
```


(continues on next page)

(continued from previous page)

```
                "title": "External Def",
                "description": "If specified, the model will be built from the␣
↪definition from this external source, using Torch Hub.",
                "allOf": [
                   {
                      "$ref": "#/definitions/ExternalModuleConfig"
                   }
                ]
             },
             "type_hint": {
                "title": "Type Hint",
                "default": "regression_model",
                "enum": [
                   "regression_model"
                ],
                "type": "string"
             },
             "output_multiplier": {
                "title": "Output Multiplier",
                "type": "array",
                "items": {
                   "type": "number"
                }
             }
          },
          "additionalProperties": false
       },
       "SolverConfig": {
          "title": "SolverConfig",
          "description": "Config related to solver aka optimizer.",
          "type": "object",
          "properties": {
             "lr": {
                "title": "Lr",
                "description": "Learning rate.",
                "default": 0.0001,
                "exclusiveMinimum": 0,
                "type": "number"
             },
             "num_epochs": {
                "title": "Num Epochs",
                "description": "Number of epochs (ie. sweeps through the whole␣
↪training set).",
                "default": 10,
                "exclusiveMinimum": 0,
                "type": "integer"
             },
             "test_num_epochs": {
                "title": "Test Num Epochs",
                "description": "Number of epochs to use in test mode.",
                "default": 2,
                "exclusiveMinimum": 0,
```

(continues on next page)

(continued from previous page)

```
            "type": "integer"
         },
         "test_batch_sz": {
            "title": "Test Batch Sz",
            "description": "Batch size to use in test mode.",
            "default": 4,
            "exclusiveMinimum": 0,
            "type": "integer"
         },
         "overfit_num_steps": {
            "title": "Overfit Num Steps",
            "description": "Number of optimizer steps to use in overfit mode.",
            "default": 1,
            "exclusiveMinimum": 0,
            "type": "integer"
         },
         "sync_interval": {
            "title": "Sync Interval",
            "description": "The interval in epochs for each sync to the cloud.",
            "default": 1,
            "exclusiveMinimum": 0,
            "type": "integer"
         },
         "batch_sz": {
            "title": "Batch Sz",
            "description": "Batch size.",
            "default": 32,
            "exclusiveMinimum": 0,
            "type": "integer"
         },
         "one_cycle": {
            "title": "One Cycle",
            "description": "If True, use triangular LR scheduler with a single␣
↪cycle across all epochs with start and end LR being lr/10 and the peak being lr.",
            "default": true,
            "type": "boolean"
         },
         "multi_stage": {
            "title": "Multi Stage",
            "description": "List of epoch indices at which to divide LR by 10.",
            "default": [],
            "type": "array",
            "items": {}
         },
         "class_loss_weights": {
            "title": "Class Loss Weights",
            "description": "Class weights for weighted loss.",
            "type": "array",
            "items": {
               "type": "number"
            }
         },
```

(continues on next page)

(continued from previous page)


```
            "ignore_class_index": {
               "title": "Ignore Class Index",
               "description": "If specified, this index is ignored when computing␣
↪the loss. See pytorch documentation for nn.CrossEntropyLoss for more details.␣
↪This can also be negative, in which case it is treated as a negative slice index␣
↪i.e. -1 = last index, -2 = second-last index, and so on.",
               "type": "integer"
            },
            "external_loss_def": {
               "title": "External Loss Def",
               "description": "If specified, the loss will be built from the␣
↪definition from this external source, using Torch Hub.",
               "allOf": [
                  {
                     "$ref": "#/definitions/ExternalModuleConfig"
                  }
               ]
            },
            "type_hint": {
               "title": "Type Hint",
               "default": "solver",
               "enum": [
                  "solver"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      },
      "RegressionPlotOptions": {
         "title": "RegressionPlotOptions",
         "description": "Config related to plotting.",
         "type": "object",
         "properties": {
            "transform": {
               "title": "Transform",
               "description": "An Albumentations transform serialized as a dict␣
↪that will be applied to each image before it is plotted. Mainly useful for␣
↪undoing any data transformation that you do not want included in the plot, such␣
↪as normalization. The default value will shift and scale the image so the values␣
↪range from 0.0 to 1.0 which is the expected range for the plotting function. This␣
↪default is useful for cases where the values after normalization are close to␣
↪zero which makes the plot difficult to see.",
               "default": {
                  "__version__": "1.3.0",
                  "transform": {
                     "__class_fullname__": "rastervision.pytorch_learner.utils.
↪utils.MinMaxNormalize",
                     "always_apply": false,
                     "p": 1.0,
                     "min_val": 0.0,
                     "max_val": 1.0,
```


(continues on next page)

(continued from previous page)

```
                    "dtype": 5
                }
            },
            "type": "object"
        },
        "channel_display_groups": {
            "title": "Channel Display Groups",
            "description": "Groups of image channels to display together as a␣
↪subplot when plotting the data and predictions. Can be a list or tuple of groups␣
↪(e.g. [(0, 1, 2), (3,)]) or a dict containing title-to-group mappings (e.g. {\
↪"RGB\": [0, 1, 2], \"IR\": [3]}), where each group is a list or tuple of channel␣
↪indices and title is a string that will be used as the title of the subplot for␣
↪that group.",
            "anyOf": [
                {
                    "type": "object",
                    "additionalProperties": {
                        "type": "array",
                        "items": {
                            "type": "integer",
                            "minimum": 0
                        }
                    }
                },
                {
                    "type": "array",
                    "items": {
                        "type": "array",
                        "items": {
                            "type": "integer",
                            "minimum": 0
                        }
                    }
                }
            ]
        },
        "type_hint": {
            "title": "Type Hint",
            "default": "regression_plot_options",
            "enum": [
                "regression_plot_options"
            ],
            "type": "string"
        },
        "max_scatter_points": {
            "title": "Max Scatter Points",
            "description": "Maximum number of datapoints to use in scatter plot.␣
↪Useful to avoid running out of memory and cluttering.",
            "default": 5000,
            "type": "integer"
        },
        "hist_bins": {
```

(continues on next page)

(continued from previous page)


```
                "title": "Hist Bins",
                "description": "Number of bins to use for histogram.",
                "default": 30,
                "type": "integer"
             }
          },
          "additionalProperties": false
       },
       "RegressionDataFormat": {
          "title": "RegressionDataFormat",
          "description": "An enumeration.",
          "enum": [
             "csv"
          ]
       },
       "RegressionImageDataConfig": {
          "title": "RegressionImageDataConfig",
          "description": "Configure :class:`RegressionImageDatasets <.
↪RegressionImageDataset>`.",
          "type": "object",
          "properties": {
             "class_names": {
                "title": "Class Names",
                "description": "Names of classes.",
                "default": [],
                "type": "array",
                "items": {
                   "type": "string"
                }
             },
             "class_colors": {
                "title": "Class Colors",
                "description": "Colors used to display classes. Can be color 3-
↪tuples in list form.",
                "type": "array",
                "items": {
                   "anyOf": [
                      {
                         "type": "string"
                      },
                      {
                         "type": "array",
                         "minItems": 3,
                         "maxItems": 3,
                         "items": [
                            {
                               "type": "integer"
                            },
                            {
                               "type": "integer"
                            },
                            {
```


(continues on next page)

(continued from previous page)

```
                            "type": "integer"
                        }
                    ]
                }
            ]
        }
    },
    "img_channels": {
        "title": "Img Channels",
        "description": "The number of channels of the training images.",
        "exclusiveMinimum": 0,
        "type": "integer"
    },
    "img_sz": {
        "title": "Img Sz",
        "description": "Length of a side of each image in pixels. This is␣
↪the size to transform it to during training, not the size in the raw dataset.",
        "default": 256,
        "exclusiveMinimum": 0,
        "type": "integer"
    },
    "train_sz": {
        "title": "Train Sz",
        "description": "If set, the number of training images to use. If␣
↪fewer images exist, then an exception will be raised.",
        "type": "integer"
    },
    "train_sz_rel": {
        "title": "Train Sz Rel",
        "description": "If set, the proportion of training images to use.",
        "type": "number"
    },
    "num_workers": {
        "title": "Num Workers",
        "description": "Number of workers to use when DataLoader makes␣
↪batches.",
        "default": 4,
        "type": "integer"
    },
    "augmentors": {
        "title": "Augmentors",
        "description": "Names of albumentations augmentors to use for␣
↪training batches. Choices include: ['Blur', 'RandomRotate90', 'HorizontalFlip',
↪'VerticalFlip', 'GaussianBlur', 'GaussNoise', 'RGBShift', 'ToGray'].␣
↪Alternatively, a custom transform can be provided via the aug_transform option.",
        "default": [
            "RandomRotate90",
            "HorizontalFlip",
            "VerticalFlip"
        ],
        "type": "array",
        "items": {
```

(continues on next page)

(continued from previous page)


```
                "type": "string"
            }
        },
        "base_transform": {
            "title": "Base Transform",
            "description": "An Albumentations transform serialized as a dict␣
↪that will be applied to all datasets: training, validation, and test. This␣
↪transformation is in addition to the resizing due to img_sz. This is useful for,␣
↪for example, applying the same normalization to all datasets.",
            "type": "object"
        },
        "aug_transform": {
            "title": "Aug Transform",
            "description": "An Albumentations transform serialized as a dict␣
↪that will be applied as data augmentation to the training dataset. This transform␣
↪is applied before base_transform. If provided, the augmentors option is ignored.",
            "type": "object"
        },
        "plot_options": {
            "title": "Plot Options",
            "description": "Options to control plotting.",
            "default": {
                "transform": {
                    "__version__": "1.3.0",
                    "transform": {
                        "__class_fullname__": "rastervision.pytorch_learner.utils.
↪utils.MinMaxNormalize",
                        "always_apply": false,
                        "p": 1.0,
                        "min_val": 0.0,
                        "max_val": 1.0,
                        "dtype": 5
                    }
                },
                "channel_display_groups": null,
                "type_hint": "regression_plot_options",
                "max_scatter_points": 5000,
                "hist_bins": 30
            },
            "allOf": [
                {
                    "$ref": "#/definitions/RegressionPlotOptions"
                }
            ]
        },
        "preview_batch_limit": {
            "title": "Preview Batch Limit",
            "description": "Optional limit on the number of items in the preview␣
↪plots produced during training.",
            "type": "integer"
        },
        "type_hint": {
```


(continues on next page)

(continued from previous page)


```
                "title": "Type Hint",
                "default": "regression_image_data",
                "enum": [
                   "regression_image_data"
                ],
                "type": "string"
             },
             "data_format": {
                "default": "csv",
                "allOf": [
                   {
                      "$ref": "#/definitions/RegressionDataFormat"
                   }
                ]
             },
             "uri": {
                "title": "Uri",
                "description": "One of the following:\n(1) a URI of a directory␣
↪containing \"train\", \"valid\", and (optinally) \"test\" subdirectories;\n(2) a␣
↪URI of a zip file containing (1);\n(3) a list of (2);\n(4) a URI of a directory␣
↪containing zip files containing (1).",
                "anyOf": [
                   {
                      "type": "string"
                   },
                   {
                      "type": "array",
                      "items": {
                         "type": "string"
                      }
                   }
                ]
             },
             "group_uris": {
                "title": "Group Uris",
                "description": "This can be set instead of uri in order to specify␣
↪groups of chips. Each element in the list is expected to be an object of the same␣
↪form accepted by the uri field. The purpose of separating chips into groups is to␣
↪be able to use the group_train_sz field.",
                "type": "array",
                "items": {
                   "anyOf": [
                      {
                         "type": "string"
                      },
                      {
                         "type": "array",
                         "items": {
                            "type": "string"
                         }
                      }
                   ]
```


(continues on next page)

(continued from previous page)

```
            }
         },
         "group_train_sz": {
            "title": "Group Train Sz",
            "description": "If group_uris is set, this can be used to specify␣
↪the number of chips to use per group. Only applies to training chips. This can␣
↪either be a single value that will be used for all groups or a list of values␣
↪(one for each group).",
            "anyOf": [
               {
                  "type": "integer"
               },
               {
                  "type": "array",
                  "items": {
                     "type": "integer"
                  }
               }
            ]
         },
         "group_train_sz_rel": {
            "title": "Group Train Sz Rel",
            "description": "Relative version of group_train_sz. Must be a float␣
↪in [0, 1]. If group_uris is set, this can be used to specify the proportion of␣
↪the total chips in each group to use per group. Only applies to training chips.␣
↪This can either be a single value that will be used for all groups or a list of␣
↪values (one for each group).",
            "anyOf": [
               {
                  "type": "number",
                  "minimum": 0,
                  "maximum": 1
               },
               {
                  "type": "array",
                  "items": {
                     "type": "number",
                     "minimum": 0,
                     "maximum": 1
                  }
               }
            ]
         },
         "pos_class_names": {
            "title": "Pos Class Names",
            "default": [],
            "type": "array",
            "items": {
               "type": "string"
            }
         },
         "prob_class_names": {
```

(continues on next page)

(continued from previous page)

```
                "title": "Prob Class Names",
                "default": [],
                "type": "array",
                "items": {
                   "type": "string"
                }
             }
          },
          "additionalProperties": false
       },
       "ClassConfig": {
          "title": "ClassConfig",
          "description": "Configure class information for a machine learning task.",
          "type": "object",
          "properties": {
             "names": {
                "title": "Names",
                "description": "Names of classes. The i-th class in this list will␣
↪have class ID = i.",
                "type": "array",
                "items": {
                   "type": "string"
                }
             },
             "colors": {
                "title": "Colors",
                "description": "Colors used to visualize classes. Can be color␣
↪strings accepted by matplotlib or RGB tuples. If None, a random color will be␣
↪auto-generated for each class.",
                "type": "array",
                "items": {
                   "anyOf": [
                      {
                         "type": "string"
                      },
                      {
                         "type": "array",
                         "items": {}
                      }
                   ]
                }
             },
             "null_class": {
                "title": "Null Class",
                "description": "Optional name of class in `names` to use as the null␣
↪class. This is used in semantic segmentation to represent the label for imagery␣
↪pixels that are NODATA or that are missing a label. If None and the class names␣
↪include \"null\", it will automatically be used as the null class. If None, and␣
↪this Config is part of a SemanticSegmentationConfig, a null class will be added␣
↪automatically.",
                "type": "string"
             },
```

(continues on next page)

(continued from previous page)


```
            "type_hint": {
               "title": "Type Hint",
               "default": "class_config",
               "enum": [
                  "class_config"
               ],
               "type": "string"
            }
         },
         "required": [
            "names"
         ],
         "additionalProperties": false
      },
      "RasterTransformerConfig": {
         "title": "RasterTransformerConfig",
         "description": "Configure a :class:`.RasterTransformer`.",
         "type": "object",
         "properties": {
            "type_hint": {
               "title": "Type Hint",
               "default": "raster_transformer",
               "enum": [
                  "raster_transformer"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      },
      "RasterSourceConfig": {
         "title": "RasterSourceConfig",
         "description": "Configure a :class:`.RasterSource`.",
         "type": "object",
         "properties": {
            "channel_order": {
               "title": "Channel Order",
               "description": "The sequence of channel indices to use when reading␣
↪imagery.",
               "type": "array",
               "items": {
                  "type": "integer"
               }
            },
            "transformers": {
               "title": "Transformers",
               "default": [],
               "type": "array",
               "items": {
                  "$ref": "#/definitions/RasterTransformerConfig"
               }
            },
```


(continues on next page)

(continued from previous page)


```
            "extent": {
               "title": "Extent",
               "description": "Use-specified extent in pixel coords in the form␣
↪(ymin, xmin, ymax, xmax). Useful for cropping the raster source so that only part␣
↪of the raster is read from.",
               "type": "array",
               "minItems": 4,
               "maxItems": 4,
               "items": [
                  {
                     "type": "integer"
                  },
                  {
                     "type": "integer"
                  },
                  {
                     "type": "integer"
                  },
                  {
                     "type": "integer"
                  }
               ]
            },
            "type_hint": {
               "title": "Type Hint",
               "default": "raster_source",
               "enum": [
                  "raster_source"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      },
      "LabelSourceConfig": {
         "title": "LabelSourceConfig",
         "description": "Configure a :class:`.LabelSource`.",
         "type": "object",
         "properties": {
            "type_hint": {
               "title": "Type Hint",
               "default": "label_source",
               "enum": [
                  "label_source"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      },
      "LabelStoreConfig": {
         "title": "LabelStoreConfig",
```


(continues on next page)

(continued from previous page)


```
        "description": "Configure a :class:`.LabelStore`.",
        "type": "object",
        "properties": {
           "type_hint": {
              "title": "Type Hint",
              "default": "label_store",
              "enum": [
                 "label_store"
              ],
              "type": "string"
           }
        },
        "additionalProperties": false
     },
     "SceneConfig": {
        "title": "SceneConfig",
        "description": "Configure a :class:`.Scene` comprising raster data &␣
↪labels for an AOI.\n    ",
        "type": "object",
        "properties": {
           "id": {
              "title": "Id",
              "type": "string"
           },
           "raster_source": {
              "$ref": "#/definitions/RasterSourceConfig"
           },
           "label_source": {
              "$ref": "#/definitions/LabelSourceConfig"
           },
           "label_store": {
              "$ref": "#/definitions/LabelStoreConfig"
           },
           "aoi_uris": {
              "title": "Aoi Uris",
              "description": "List of URIs of GeoJSON files that define the AOIs␣
↪for the scene. Each polygon defines an AOI which is a piece of the scene that is␣
↪assumed to be fully labeled and usable for training or validation. The AOIs are␣
↪assumed to be in EPSG:4326 coordinates.",
              "type": "array",
              "items": {
                 "type": "string"
              }
           },
           "type_hint": {
              "title": "Type Hint",
              "default": "scene",
              "enum": [
                 "scene"
              ],
              "type": "string"
           }
```


(continues on next page)

(continued from previous page)


```
        },
        "required": [
           "id",
           "raster_source"
        ],
        "additionalProperties": false
     },
     "DatasetConfig": {
        "title": "DatasetConfig",
        "description": "Configure train, validation, and test splits for a dataset.
↪",
        "type": "object",
        "properties": {
           "class_config": {
              "$ref": "#/definitions/ClassConfig"
           },
           "train_scenes": {
              "title": "Train Scenes",
              "type": "array",
              "items": {
                 "$ref": "#/definitions/SceneConfig"
              }
           },
           "validation_scenes": {
              "title": "Validation Scenes",
              "type": "array",
              "items": {
                 "$ref": "#/definitions/SceneConfig"
              }
           },
           "test_scenes": {
              "title": "Test Scenes",
              "default": [],
              "type": "array",
              "items": {
                 "$ref": "#/definitions/SceneConfig"
              }
           },
           "scene_groups": {
              "title": "Scene Groups",
              "description": "Groupings of scenes. Should be a dict of the form: {
↪<group-name>: Set(scene_id_1, scene_id_2, ...)}. Three groups are added by␣
↪default: \"train_scenes\", \"validation_scenes\", and \"test_scenes\"",
              "default": {},
              "type": "object",
              "additionalProperties": {
                 "type": "array",
                 "items": {
                    "type": "string"
                 },
                 "uniqueItems": true
              }
```


(continues on next page)

(continued from previous page)


```
            },
            "type_hint": {
               "title": "Type Hint",
               "default": "dataset",
               "enum": [
                  "dataset"
               ],
               "type": "string"
            }
         },
         "required": [
            "class_config",
            "train_scenes",
            "validation_scenes"
         ],
         "additionalProperties": false
      },
      "GeoDataWindowMethod": {
         "title": "GeoDataWindowMethod",
         "description": "An enumeration.",
         "enum": [
            "sliding",
            "random"
         ]
      },
      "GeoDataWindowConfig": {
         "title": "GeoDataWindowConfig",
         "description": "Configure a :class:`.GeoDataset`.\n\nSee :mod:
↪`rastervision.pytorch_learner.dataset.dataset`.",
         "type": "object",
         "properties": {
            "method": {
               "default": "sliding",
               "allOf": [
                  {
                     "$ref": "#/definitions/GeoDataWindowMethod"
                  }
               ]
            },
            "size": {
               "title": "Size",
               "description": "If method = sliding, this is the size of sliding␣
↪window. If method = random, this is the size that all the windows are resized to␣
↪before they are returned. If method = random and neither size_lims nor h_lims and␣
↪w_lims have been specified, then size_lims is set to (size, size + 1).",
               "anyOf": [
                  {
                     "type": "integer",
                     "exclusiveMinimum": 0
                  },
                  {
                     "type": "array",
```


(continues on next page)

(continued from previous page)

```
                    "minItems": 2,
                    "maxItems": 2,
                    "items": [
                       {
                          "type": "integer",
                          "exclusiveMinimum": 0
                       },
                       {
                          "type": "integer",
                          "exclusiveMinimum": 0
                       }
                    ]
                 }
              ]
           },
           "stride": {
              "title": "Stride",
              "description": "Stride of sliding window. Only used if method =␣
↪sliding.",
              "anyOf": [
                 {
                    "type": "integer",
                    "exclusiveMinimum": 0
                 },
                 {
                    "type": "array",
                    "minItems": 2,
                    "maxItems": 2,
                    "items": [
                       {
                          "type": "integer",
                          "exclusiveMinimum": 0
                       },
                       {
                          "type": "integer",
                          "exclusiveMinimum": 0
                       }
                    ]
                 }
              ]
           },
           "padding": {
              "title": "Padding",
              "description": "How many pixels are windows allowed to overflow the␣
↪edges of the raster source.",
              "anyOf": [
                 {
                    "type": "integer",
                    "minimum": 0
                 },
                 {
                    "type": "array",
```

(continues on next page)

(continued from previous page)

```
                    "minItems": 2,
                    "maxItems": 2,
                    "items": [
                       {
                          "type": "integer",
                          "minimum": 0
                       },
                       {
                          "type": "integer",
                          "minimum": 0
                       }
                    ]
                 }
              ]
           },
           "pad_direction": {
              "title": "Pad Direction",
              "description": "If \"end\", only pad ymax and xmax (bottom and␣
↪right). If \"start\", only pad ymin and xmin (top and left). If \"both\", pad all␣
↪sides. Has no effect if paddiong is zero. Defaults to \"end\".",
              "default": "end",
              "enum": [
                 "both",
                 "start",
                 "end"
              ],
              "type": "string"
           },
           "size_lims": {
              "title": "Size Lims",
              "description": "[min, max) interval from which window sizes will be␣
↪uniformly randomly sampled. The upper limit is exclusive. To fix the size to a␣
↪constant value, use size_lims = (sz, sz + 1). Only used if method = random.␣
↪Specify either size_lims or h_lims and w_lims, but not both. If neither size_lims␣
↪nor h_lims and w_lims have been specified, then this will be set to (size, size +␣
↪1).",
              "type": "array",
              "minItems": 2,
              "maxItems": 2,
              "items": [
                 {
                    "type": "integer",
                    "exclusiveMinimum": 0
                 },
                 {
                    "type": "integer",
                    "exclusiveMinimum": 0
                 }
              ]
           },
           "h_lims": {
              "title": "H Lims",
```

(continues on next page)

(continued from previous page)

```
                "description": "[min, max] interval from which window heights will␣
↪be uniformly randomly sampled. Only used if method = random.",
                "type": "array",
                "minItems": 2,
                "maxItems": 2,
                "items": [
                   {
                      "type": "integer",
                      "exclusiveMinimum": 0
                   },
                   {
                      "type": "integer",
                      "exclusiveMinimum": 0
                   }
                ]
             },
             "w_lims": {
                "title": "W Lims",
                "description": "[min, max] interval from which window widths will be␣
↪uniformly randomly sampled. Only used if method = random.",
                "type": "array",
                "minItems": 2,
                "maxItems": 2,
                "items": [
                   {
                      "type": "integer",
                      "exclusiveMinimum": 0
                   },
                   {
                      "type": "integer",
                      "exclusiveMinimum": 0
                   }
                ]
             },
             "max_windows": {
                "title": "Max Windows",
                "description": "Max allowed reads from a GeoDataset. Only used if␣
↪method = random.",
                "default": 10000,
                "minimum": 0,
                "type": "integer"
             },
             "max_sample_attempts": {
                "title": "Max Sample Attempts",
                "description": "Max attempts when trying to find a window within the␣
↪AOI of a scene. Only used if method = random and the scene has aoi_polygons␣
↪specified.",
                "default": 100,
                "exclusiveMinimum": 0,
                "type": "integer"
             },
             "efficient_aoi_sampling": {
```

(continues on next page)

(continued from previous page)

```
                "title": "Efficient Aoi Sampling",
                "description": "If the scene has AOIs, sampling windows at random␣
↪anywhere in the extent and then checking if they fall within any of the AOIs can␣
↪be very inefficient. This flag enables the use of an alternate algorithm that␣
↪only samples window locations inside the AOIs. Only used if method = random and␣
↪the scene has aoi_polygons specified. Defaults to True",
                "default": true,
                "type": "boolean"
             },
             "type_hint": {
                "title": "Type Hint",
                "default": "geo_data_window",
                "enum": [
                   "geo_data_window"
                ],
                "type": "string"
             }
          },
          "required": [
             "size"
          ],
          "additionalProperties": false
       },
       "RegressionGeoDataConfig": {
          "title": "RegressionGeoDataConfig",
          "description": "Configure regression :class:`GeoDatasets <.GeoDataset>`.\n\
↪nSee :mod:`rastervision.pytorch_learner.dataset.regression_dataset`.",
          "type": "object",
          "properties": {
             "class_names": {
                "title": "Class Names",
                "description": "Names of classes.",
                "default": [],
                "type": "array",
                "items": {
                   "type": "string"
                }
             },
             "class_colors": {
                "title": "Class Colors",
                "description": "Colors used to display classes. Can be color 3-
↪tuples in list form.",
                "type": "array",
                "items": {
                   "anyOf": [
                      {
                         "type": "string"
                      },
                      {
                         "type": "array",
                         "minItems": 3,
                         "maxItems": 3,
```

(continues on next page)

(continued from previous page)

```
                        "items": [
                           {
                              "type": "integer"
                           },
                           {
                              "type": "integer"
                           },
                           {
                              "type": "integer"
                           }
                        ]
                     }
                  ]
               }
            },
            "img_channels": {
               "title": "Img Channels",
               "description": "The number of channels of the training images.",
               "exclusiveMinimum": 0,
               "type": "integer"
            },
            "img_sz": {
               "title": "Img Sz",
               "description": "Length of a side of each image in pixels. This is␣
↪the size to transform it to during training, not the size in the raw dataset.",
               "default": 256,
               "exclusiveMinimum": 0,
               "type": "integer"
            },
            "train_sz": {
               "title": "Train Sz",
               "description": "If set, the number of training images to use. If␣
↪fewer images exist, then an exception will be raised.",
               "type": "integer"
            },
            "train_sz_rel": {
               "title": "Train Sz Rel",
               "description": "If set, the proportion of training images to use.",
               "type": "number"
            },
            "num_workers": {
               "title": "Num Workers",
               "description": "Number of workers to use when DataLoader makes␣
↪batches.",
               "default": 4,
               "type": "integer"
            },
            "augmentors": {
               "title": "Augmentors",
               "description": "Names of albumentations augmentors to use for␣
↪training batches. Choices include: ['Blur', 'RandomRotate90', 'HorizontalFlip',
↪'VerticalFlip', 'GaussianBlur', 'GaussNoise', 'RGBShift', 'ToGray'].␣
```

(continues on next page)

(continued from previous page)

```
↪Alternatively, a custom transform can be provided via the aug_transform option.",
                "default": [
                    "RandomRotate90",
                    "HorizontalFlip",
                    "VerticalFlip"
                ],
                "type": "array",
                "items": {
                    "type": "string"
                }
            },
            "base_transform": {
                "title": "Base Transform",
                "description": "An Albumentations transform serialized as a dict␣
↪that will be applied to all datasets: training, validation, and test. This␣
↪transformation is in addition to the resizing due to img_sz. This is useful for,␣
↪for example, applying the same normalization to all datasets.",
                "type": "object"
            },
            "aug_transform": {
                "title": "Aug Transform",
                "description": "An Albumentations transform serialized as a dict␣
↪that will be applied as data augmentation to the training dataset. This transform␣
↪is applied before base_transform. If provided, the augmentors option is ignored.",
                "type": "object"
            },
            "plot_options": {
                "title": "Plot Options",
                "description": "Options to control plotting.",
                "default": {
                    "transform": {
                        "__version__": "1.3.0",
                        "transform": {
                            "__class_fullname__": "rastervision.pytorch_learner.utils.
↪utils.MinMaxNormalize",
                            "always_apply": false,
                            "p": 1.0,
                            "min_val": 0.0,
                            "max_val": 1.0,
                            "dtype": 5
                        }
                    },
                    "channel_display_groups": null,
                    "type_hint": "regression_plot_options",
                    "max_scatter_points": 5000,
                    "hist_bins": 30
                },
                "allOf": [
                    {
                        "$ref": "#/definitions/RegressionPlotOptions"
                    }
                ]
```

(continues on next page)

(continued from previous page)

```
            },
            "preview_batch_limit": {
               "title": "Preview Batch Limit",
               "description": "Optional limit on the number of items in the preview␣
↪plots produced during training.",
               "type": "integer"
            },
            "type_hint": {
               "title": "Type Hint",
               "default": "regression_geo_data",
               "enum": [
                  "regression_geo_data"
               ],
               "type": "string"
            },
            "scene_dataset": {
               "$ref": "#/definitions/DatasetConfig"
            },
            "window_opts": {
               "title": "Window Opts",
               "default": {},
               "anyOf": [
                  {
                     "$ref": "#/definitions/GeoDataWindowConfig"
                  },
                  {
                     "type": "object",
                     "additionalProperties": {
                        "$ref": "#/definitions/GeoDataWindowConfig"
                     }
                  }
               ]
            },
            "pos_class_names": {
               "title": "Pos Class Names",
               "default": [],
               "type": "array",
               "items": {
                  "type": "string"
               }
            },
            "prob_class_names": {
               "title": "Prob Class Names",
               "default": [],
               "type": "array",
               "items": {
                  "type": "string"
               }
            }
         },
         "additionalProperties": false
      }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *data (Union[rastervision.pytorch_learner.regression_learner_config.RegressionImageDataConfig, rastervision.pytorch_learner.regression_learner_config.RegressionGeoDataConfig])*
- *eval_train (bool)*
- *log_tensorboard (bool)*
- *model (Optional[rastervision.pytorch_learner.regression_learner_config.RegressionModelConfig])*
- *output_uri (Optional[str])*
- *overfit_mode (bool)*
- *predict_mode (bool)*
- *run_tensorboard (bool)*
- *save_model_bundle (bool)*
- *solver (rastervision.pytorch_learner.learner_config.SolverConfig)*
- *test_mode (bool)*
- *type_hint (Literal['regression_learner'])*

Validators

- update_for_mode » all fields
- validate_class_loss_weights » all fields
- validate_run_tensorboard » *run_tensorboard*

field data: Union[*RegressionImageDataConfig*, *RegressionGeoDataConfig*] [Required]

Validated by

- update_for_mode
- validate_class_loss_weights

field eval_train: bool = False

If True, runs final evaluation on training set (in addition to test set). Useful for debugging.

Validated by

- update_for_mode
- validate_class_loss_weights

field log_tensorboard: bool = True

Save Tensorboard log files at the end of each epoch.

Validated by

- update_for_mode
- validate_class_loss_weights

field model: Optional[*RegressionModelConfig*] = None

Validated by

- update_for_mode
- validate_class_loss_weights

field output_uri: Optional[str] = None

URI of where to save output

Validated by

- update_for_mode
- validate_class_loss_weights

field overfit_mode: bool = False

If True, uses half image size, and instead of doing epoch-based training, optimizes the model using a single batch repeatedly for overfit_num_steps number of steps.

Validated by

- update_for_mode
- validate_class_loss_weights

field predict_mode: bool = False

If True, skips training, loads model, and does final eval.

Validated by

- update_for_mode
- validate_class_loss_weights

field run_tensorboard: bool = False

run Tensorboard server during training

Validated by

- update_for_mode
- validate_class_loss_weights
- validate_run_tensorboard

field save_model_bundle: bool = True

If True, saves a model bundle at the end of training which is zip file with model and this LearnerConfig which can be used to make predictions on new images at a later time.

Validated by

- update_for_mode
- validate_class_loss_weights

field solver: *SolverConfig* [Required]

Validated by

- update_for_mode
- validate_class_loss_weights

field test_mode: bool = False

If True, uses test_num_epochs, test_batch_sz, truncated datasets with only a single batch, image_sz that is cut in half, and num_workers = 0. This is useful for testing that code runs correctly on CPU without multithreading before running full job on GPU.

Validated by

- update_for_mode
- validate_class_loss_weights

field type_hint: Literal['regression_learner'] = 'regression_learner'

Validated by

- update_for_mode
- validate_class_loss_weights

build(*tmp_dir*, *model_weights_path=None*, *model_def_path=None*, *loss_def_path=None*, *training=True*)

Returns a Learner instantiated using this Config.

Parameters

- **tmp_dir** (*str*) – Root of temp dirs.
- **model_weights_path** (*str, optional*) – A local path to model weights. Defaults to None.
- **model_def_path** (*str, optional*) – A local path to a directory with a hubconf.py. If provided, the model definition is imported from here. Defaults to None.
- **loss_def_path** (*str, optional*) – A local path to a directory with a hubconf.py. If provided, the loss function definition is imported from here. Defaults to None.
- **training** (*bool, optional*) – Whether the model is to be used for training or prediction. If False, the model is put in eval mode and the loss function, optimizer, etc. are not initialized. Defaults to True.

get_model_bundle_uri() → str

Returns the URI of where the model bundle is stored.

Return type
str

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(**args*, ***kwargs*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validator update_for_mode » *all fields*

Parameters
values (*dict*) –

Return type
dict

validator validate_class_loss_weights » *all fields*

Parameters
values (*dict*) –

Return type
dict

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises
ConfigError – if field is invalid

validator validate_run_tensorboard » run_tensorboard

Parameters

- **v** (*bool*) –
- **values** (*dict*) –

Return type
bool

RegressionModelConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model RegressionModelConfig

Configure a regression model.

```
{
   "title": "RegressionModelConfig",
   "description": "Configure a regression model.",
   "type": "object",
   "properties": {
      "backbone": {
         "description": "The torchvision.models backbone to use.",
         "default": "resnet18",
         "allOf": [
            {
               "$ref": "#/definitions/Backbone"
            }
         ]
      },
      "pretrained": {
         "title": "Pretrained",
         "description": "If True, use ImageNet weights. If False, use random␣
↪initialization.",
         "default": true,
         "type": "boolean"
      },
      "init_weights": {
         "title": "Init Weights",
         "description": "URI of PyTorch model weights used to initialize model. If␣
↪set, this supercedes the pretrained option.",
         "type": "string"
      },
      "load_strict": {
         "title": "Load Strict",
         "description": "If True, the keys in the state dict referenced by init_
↪weights must match exactly. Setting this to False can be useful if you just want␣
↪to load the backbone of a model.",
         "default": true,
         "type": "boolean"
      },
      "external_def": {
         "title": "External Def",
         "description": "If specified, the model will be built from the definition␣
↪from this external source, using Torch Hub.",
         "allOf": [
            {
               "$ref": "#/definitions/ExternalModuleConfig"
            }
         ]
      },
      "type_hint": {
         "title": "Type Hint",
         "default": "regression_model",
         "enum": [
            "regression_model"
         ],
         "type": "string"
      },
```


(continues on next page)

(continued from previous page)

```
        "output_multiplier": {
            "title": "Output Multiplier",
            "type": "array",
            "items": {
                "type": "number"
            }
        }
    },
    "additionalProperties": false,
    "definitions": {
        "Backbone": {
            "title": "Backbone",
            "description": "An enumeration.",
            "enum": [
                "alexnet",
                "densenet121",
                "densenet169",
                "densenet201",
                "densenet161",
                "googlenet",
                "inception_v3",
                "mnasnet0_5",
                "mnasnet0_75",
                "mnasnet1_0",
                "mnasnet1_3",
                "mobilenet_v2",
                "resnet18",
                "resnet34",
                "resnet50",
                "resnet101",
                "resnet152",
                "resnext50_32x4d",
                "resnext101_32x8d",
                "wide_resnet50_2",
                "wide_resnet101_2",
                "shufflenet_v2_x0_5",
                "shufflenet_v2_x1_0",
                "shufflenet_v2_x1_5",
                "shufflenet_v2_x2_0",
                "squeezenet1_0",
                "squeezenet1_1",
                "vgg11",
                "vgg11_bn",
                "vgg13",
                "vgg13_bn",
                "vgg16",
                "vgg16_bn",
                "vgg19_bn",
                "vgg19"
            ]
        },
        "ExternalModuleConfig": {
```

(continues on next page)

(continued from previous page)


```
        "title": "ExternalModuleConfig",
        "description": "Config describing an object to be loaded via Torch Hub.",
        "type": "object",
        "properties": {
           "uri": {
              "title": "Uri",
              "description": "Local uri of a zip file, or local uri of a directory,
↪or remote uri of zip file.",
              "minLength": 1,
              "type": "string"
           },
           "github_repo": {
              "title": "Github Repo",
              "description": "<repo-owner>/<repo-name>[:tag]",
              "pattern": ".+/.+",
              "type": "string"
           },
           "name": {
              "title": "Name",
              "description": "Name of the folder in which to extract/copy the␣
↪definition files.",
              "minLength": 1,
              "type": "string"
           },
           "entrypoint": {
              "title": "Entrypoint",
              "description": "Name of a callable present in hubconf.py. See docs␣
↪for torch.hub for details.",
              "minLength": 1,
              "type": "string"
           },
           "entrypoint_args": {
              "title": "Entrypoint Args",
              "description": "Args to pass to the entrypoint. Must be serializable.
↪",
              "default": [],
              "type": "array",
              "items": {}
           },
           "entrypoint_kwargs": {
              "title": "Entrypoint Kwargs",
              "description": "Keyword args to pass to the entrypoint. Must be␣
↪serializable.",
              "default": {},
              "type": "object"
           },
           "force_reload": {
              "title": "Force Reload",
              "description": "Force reload of module definition.",
              "default": false,
              "type": "boolean"
           },
```


(continues on next page)

(continued from previous page)


```
            "type_hint": {
               "title": "Type Hint",
               "default": "external-module",
               "enum": [
                  "external-module"
               ],
               "type": "string"
            }
         },
         "required": [
            "entrypoint"
         ],
         "additionalProperties": false
      }
   }
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *backbone (rastervision.pytorch_learner.learner_config.Backbone)*
- *external_def (Optional[rastervision.pytorch_learner.learner_config.ExternalModuleConfig])*
- *init_weights (Optional[str])*
- *load_strict (bool)*
- *output_multiplier (List[float])*
- *pretrained (bool)*
- *type_hint (Literal['regression_model'])*

field backbone: *Backbone* = Backbone.resnet18

The torchvision.models backbone to use.

field external_def: Optional[*ExternalModuleConfig*] = None

If specified, the model will be built from the definition from this external source, using Torch Hub.

field init_weights: Optional[str] = None

URI of PyTorch model weights used to initialize model. If set, this supercedes the pretrained option.

field load_strict: bool = True

If True, the keys in the state dict referenced by init_weights must match exactly. Setting this to False can be useful if you just want to load the backbone of a model.

field output_multiplier: List[float] = None

field pretrained: bool = True

If True, use ImageNet weights. If False, use random initialization.

field type_hint: Literal['regression_model'] = 'regression_model'

build(*num_classes: int*, *in_channels: int*, *save_dir: Optional[str] = None*, *hubconf_dir: Optional[str] = None*, ***kwargs*) → torch.nn.Module

Build and return a model based on the config.

Parameters

- **num_classes** (*int*) – Number of classes.
- **in_channels** (*int, optional*) – Number of channels in the images that will be fed into the model. Defaults to 3.
- **save_dir** (*Optional[str], optional*) – Used for building external_def if specified. Defaults to None.
- **hubconf_dir** (*Optional[str], optional*) – Used for building external_def if specified. Defaults to None.

Returns
a PyTorch nn.Module.

Return type
nn.Module

build_default_model(*num_classes: int*, *in_channels: int*, *class_names: Optional[Sequence[str]] = None*, *pos_class_names: Optional[Iterable[str]] = None*, *prob_class_names: Optional[Iterable[str]] = None*) → torch.nn.Module

Build and return the default model.

Parameters

- **num_classes** (*int*) – Number of classes.
- **in_channels** (*int, optional*) – Number of channels in the images that will be fed into the model. Defaults to 3.
- **class_names** (*Optional[Sequence[str]]*) –
- **pos_class_names** (*Optional[Iterable[str]]*) –
- **prob_class_names** (*Optional[Iterable[str]]*) –

Returns
a PyTorch nn.Module.

Return type
nn.Module

build_external_model(*save_dir: str*, *hubconf_dir: Optional[str] = None*) → torch.nn.Module

Build and return an external model.

Parameters

- **save_dir** (*str*) – The module def will be saved here.
- **hubconf_dir** (*Optional[str], optional*) – Path to existing definition. Defaults to None.

Returns
a PyTorch nn.Module.

Return type
nn.Module

get_backbone_str()

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*learner=None*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

RegressionPlotOptions

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model RegressionPlotOptions

```
{
   "title": "RegressionPlotOptions",
   "description": "Config related to plotting.",
   "type": "object",
   "properties": {
      "transform": {
         "title": "Transform",
         "description": "An Albumentations transform serialized as a dict that will␣
↪be applied to each image before it is plotted. Mainly useful for undoing any data␣
↪transformation that you do not want included in the plot, such as normalization.␣
↪The default value will shift and scale the image so the values range from 0.0 to␣
↪1.0 which is the expected range for the plotting function. This default is useful␣
```

(continues on next page)

(continued from previous page)


```
↪for cases where the values after normalization are close to zero which makes the␣
↪plot difficult to see.",
        "default": {
           "__version__": "1.3.0",
           "transform": {
              "__class_fullname__": "rastervision.pytorch_learner.utils.utils.
↪MinMaxNormalize",
              "always_apply": false,
              "p": 1.0,
              "min_val": 0.0,
              "max_val": 1.0,
              "dtype": 5
           }
        },
        "type": "object"
     },
     "channel_display_groups": {
        "title": "Channel Display Groups",
        "description": "Groups of image channels to display together as a subplot␣
↪when plotting the data and predictions. Can be a list or tuple of groups (e.g.␣
↪[(0, 1, 2), (3,)]) or a dict containing title-to-group mappings (e.g. {\"RGB\":␣
↪[0, 1, 2], \"IR\": [3]}), where each group is a list or tuple of channel indices␣
↪and title is a string that will be used as the title of the subplot for that␣
↪group.",
        "anyOf": [
           {
              "type": "object",
              "additionalProperties": {
                 "type": "array",
                 "items": {
                    "type": "integer",
                    "minimum": 0
                 }
              }
           },
           {
              "type": "array",
              "items": {
                 "type": "array",
                 "items": {
                    "type": "integer",
                    "minimum": 0
                 }
              }
           }
        ]
     },
     "type_hint": {
        "title": "Type Hint",
        "default": "regression_plot_options",
        "enum": [
           "regression_plot_options"
```


(continues on next page)

(continued from previous page)

```
         ],
         "type": "string"
      },
      "max_scatter_points": {
         "title": "Max Scatter Points",
         "description": "Maximum number of datapoints to use in scatter plot.␣
↪Useful to avoid running out of memory and cluttering.",
         "default": 5000,
         "type": "integer"
      },
      "hist_bins": {
         "title": "Hist Bins",
         "description": "Number of bins to use for histogram.",
         "default": 30,
         "type": "integer"
      }
   },
   "additionalProperties": false
}
```

Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *channel_display_groups (Optional[Union[Dict[str, Sequence[rastervision.pytorch_learner.learner_config.ConstrainedIntValue]], Sequence[Sequence[rastervision.pytorch_learner.learner_config.ConstrainedIntValue]]])*
- *hist_bins (int)*
- *max_scatter_points (int)*
- *transform (Optional[dict])*
- *type_hint (Literal['regression_plot_options'])*

Validators

- *validate_albumentation_transform* » *transform*
- validate_channel_display_groups » *channel_display_groups*

field channel_display_groups: Optional[Union[Dict[str, ChannelInds], Sequence[ChannelInds]]] = None

Groups of image channels to display together as a subplot when plotting the data and predictions. Can be a list or tuple of groups (e.g. [(0, 1, 2), (3,)]) or a dict containing title-to-group mappings (e.g. {"RGB": [0, 1, 2], "IR": [3]}), where each group is a list or tuple of channel indices and title is a string that will be used as the title of the subplot for that group.

Validated by

- validate_channel_display_groups

field hist_bins: int = 30

Number of bins to use for histogram.

field max_scatter_points: int = 5000

Maximum number of datapoints to use in scatter plot. Useful to avoid running out of memory and cluttering.

field transform: Optional[dict] = {'__version__': '1.3.0', 'transform': {'__class_fullname__': 'rastervision.pytorch_learner.utils.utils.MinMaxNormalize', 'always_apply': False, 'dtype': 5, 'max_val': 1.0, 'min_val': 0.0, 'p': 1.0}}

An Albumentations transform serialized as a dict that will be applied to each image before it is plotted. Mainly useful for undoing any data transformation that you do not want included in the plot, such as normalization. The default value will shift and scale the image so the values range from 0.0 to 1.0 which is the expected range for the plotting function. This default is useful for cases where the values after normalization are close to zero which makes the plot difficult to see.

Validated by

- *validate_albumentation_transform*

field type_hint: Literal['regression_plot_options'] = 'regression_plot_options'

build()

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(***kwargs*) → None

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

Return type
None

validator validate_channel_display_groups » channel_display_groups

Parameters
v (*Optional[Union[Dict[str, Sequence[ConstrainedIntValue]], Sequence[Sequence[ConstrainedIntValue]]]]*) –

Return type
Optional[*Dict*[str, *List*[*ConstrainedIntValue*]]]

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

9.3.13 semantic_segmentation_learner

Classes

SemanticSegmentationLearner

SemanticSegmentationLearner

class SemanticSegmentationLearner

Bases: *Learner*

__init__(*cfg: LearnerConfig*, *output_dir: Optional[str] = None*, *train_ds: Optional[Dataset] = None*, *valid_ds: Optional[Dataset] = None*, *test_ds: Optional[Dataset] = None*, *model: Optional[torch.nn.Module] = None*, *loss: Optional[Callable] = None*, *optimizer: Optional[Optimizer] = None*, *epoch_scheduler: Optional[_LRScheduler] = None*, *step_scheduler: Optional[_LRScheduler] = None*, *tmp_dir: Optional[str] = None*, *model_weights_path: Optional[str] = None*, *model_def_path: Optional[str] = None*, *loss_def_path: Optional[str] = None*, *training: bool = True*)

Constructor.

Parameters

- **cfg** (*LearnerConfig*) – LearnerConfig.
- **train_ds** (*Optional[Dataset], optional*) – The dataset to use for training. If None, will be generated from cfg.data. Defaults to None.
- **valid_ds** (*Optional[Dataset], optional*) – The dataset to use for validation. If None, will be generated from cfg.data. Defaults to None.
- **test_ds** (*Optional[Dataset], optional*) – The dataset to use for testing. If None, will be generated from cfg.data. Defaults to None.
- **model** (*Optional[nn.Module], optional*) – The model. If None, will be generated from cfg.model. Defaults to None.
- **loss** (*Optional[Callable], optional*) – The loss function. If None, will be generated from cfg.solver. Defaults to None.
- **optimizer** (*Optional[Optimizer], optional*) – The optimizer. If None, will be generated from cfg.solver. Defaults to None.
- **epoch_scheduler** (*Optional[_LRScheduler], optional*) – The scheduler that updates after each epoch. If None, will be generated from cfg.solver. Defaults to None.

- **step_scheduler** (*Optional[_LRScheduler], optional*) – The scheduler that updates after each optimizer-step. If None, will be generated from cfg.solver. Defaults to None.
- **tmp_dir** (*Optional[str], optional*) – A temporary directory to use for downloads etc. If None, will be auto-generated. Defaults to None.
- **model_weights_path** (*Optional[str], optional*) – URI of model weights to initialize the model with. Defaults to None.
- **model_def_path** (*Optional[str], optional*) – A local path to a directory with a hubconf.py. If provided, the model definition is imported from here. This is used when loading an external model from a model-bundle. Defaults to None.
- **loss_def_path** (*Optional[str], optional*) – A local path to a directory with a hubconf.py. If provided, the loss function definition is imported from here. This is used when loading an external loss function from a model-bundle. Defaults to None.
- **training** (*bool, optional*) – If False, the training apparatus (loss, optimizer, scheduler, logging, etc.) will not be set up and the model will be put into eval mode. If True, the training apparatus will be set up and the model will be put into training mode. Defaults to True.
- **output_dir** (*Optional[str]*) –

Methods

__init__(cfg[, output_dir, train_ds, ...])	Constructor.
build_dataloaders()	Set the DataLoaders for train, validation, and test sets.
build_datasets()	
build_epoch_scheduler([start_epoch])	Returns an LR scheduler that changes the LR each epoch.
build_loss([loss_def_path])	Build a loss Callable.
build_metric_names()	Returns names of metrics used to validate model at each epoch.
build_model([model_def_path])	Build a PyTorch model.
build_optimizer()	Returns optimizer.
build_step_scheduler([start_epoch])	Returns an LR scheduler that changes the LR each step.
eval_model(split)	Evaluate model using a particular dataset split.
from_model_bundle(model_bundle_uri[, ...])	Create a Learner from a model bundle.
get_collate_fn()	Returns a custom collate_fn to use in DataLoader.
get_dataloader(split)	Get the DataLoader for a split.
get_start_epoch()	Get start epoch.
get_train_sampler(train_ds)	Return a sampler to use for the training dataloader or None to not use any.
get_visualizer_class()	Returns a Visualizer class object for plotting data samples.
load_checkpoint()	Load last weights from previous run if available.
load_init_weights([model_weights_path])	Load the weights to initialize model.
load_weights(uri, **kwargs)	Load model weights from a file.
log_data_stats()	Log stats about each DataSet.
main()	Main training sequence.

continues on next page

Table 7 – continued from previous page

normalize_input(x)	Normalize x to [0, 1].
numpy_predict(x[, raw_out])	Make a prediction using an image or batch of images in numpy format.
on_epoch_end(curr_epoch, metrics)	Hook that is called at end of epoch.
on_overfit_start()	Hook that is called at start of overfit routine.
on_train_start()	Hook that is called at start of train routine.
output_to_numpy(out)	Convert output of model to numpy format.
overfit()	Optimize model using the same batch repeatedly.
plot_dataloader(dl, output_path[, ...])	Plot images and ground truth labels for a DataLoader.
plot_dataloaders([batch_limit, show])	Plot images and ground truth labels for all DataLoaders.
plot_predictions(split[, batch_limit, show])	Plot predictions for a split.
post_forward(x)	Post process output of call to model().
predict(x[, raw_out, out_shape])	Make prediction for an image or batch of images.
predict_dataloader(dl[, batched_output, ...])	Returns an iterator over predictions on the given dataloader.
predict_dataset(dataset[, return_format, ...])	Returns an iterator over predictions on the given dataset.
prob_to_pred(x)	Convert a Tensor with prediction probabilities to class ids.
run_tensorboard()	Run TB server serving logged stats.
save_model_bundle()	Save a model bundle.
setup_data()	Set datasets and dataLoaders for train, validation, and test sets.
setup_loss([loss_def_path])	Setup self.loss.
setup_model([model_weights_path, model_def_path])	Setup self.model.
setup_tensorboard()	Setup for logging stats to TB.
setup_training([loss_def_path])	
stop_tensorboard()	Stop TB logging and server if it's running.
sync_from_cloud()	Sync any previous output in the cloud to output_dir.
sync_to_cloud()	Sync any output to the cloud at output_uri.
to_batch(x)	Ensure that image array has batch dimension.
to_device(x, device)	Load Tensors onto a device.
train([epochs])	Training loop that will attempt to resume training if appropriate.
train_end(outputs, num_samples)	Aggregate the ouput of train_step at the end of the epoch.
train_epoch(optimizer[, step_scheduler])	Train for a single epoch.
train_step(batch, batch_ind)	Compute loss for a single training batch.
validate_end(outputs, num_samples)	Aggregate the ouput of validate_step at the end of the epoch.
validate_epoch(dl)	Validate for a single epoch.
validate_step(batch, batch_ind)	Compute metrics on validation batch.

__init__(*cfg: LearnerConfig*, *output_dir: Optional[str] = None*, *train_ds: Optional[Dataset] = None*, *valid_ds: Optional[Dataset] = None*, *test_ds: Optional[Dataset] = None*, *model: Optional[torch.nn.Module] = None*, *loss: Optional[Callable] = None*, *optimizer: Optional[Optimizer] = None*, *epoch_scheduler: Optional[_LRScheduler] = None*, *step_scheduler: Optional[_LRScheduler] = None*, *tmp_dir: Optional[str] = None*, *model_weights_path: Optional[str] = None*, *model_def_path: Optional[str] = None*, *loss_def_path: Optional[str] = None*, *training: bool = True*)

Constructor.

Parameters

- **cfg** (LearnerConfig) – LearnerConfig.
- **train_ds** (*Optional[Dataset], optional*) – The dataset to use for training. If None, will be generated from cfg.data. Defaults to None.
- **valid_ds** (*Optional[Dataset], optional*) – The dataset to use for validation. If None, will be generated from cfg.data. Defaults to None.
- **test_ds** (*Optional[Dataset], optional*) – The dataset to use for testing. If None, will be generated from cfg.data. Defaults to None.
- **model** (*Optional[nn.Module], optional*) – The model. If None, will be generated from cfg.model. Defaults to None.
- **loss** (*Optional[Callable], optional*) – The loss function. If None, will be generated from cfg.solver. Defaults to None.
- **optimizer** (*Optional[Optimizer], optional*) – The optimizer. If None, will be generated from cfg.solver. Defaults to None.
- **epoch_scheduler** (*Optional[_LRScheduler], optional*) – The scheduler that updates after each epoch. If None, will be generated from cfg.solver. Defaults to None.
- **step_scheduler** (*Optional[_LRScheduler], optional*) – The scheduler that updates after each optimizer-step. If None, will be generated from cfg.solver. Defaults to None.
- **tmp_dir** (*Optional[str], optional*) – A temporary directory to use for downloads etc. If None, will be auto-generated. Defaults to None.
- **model_weights_path** (*Optional[str], optional*) – URI of model weights to initialize the model with. Defaults to None.
- **model_def_path** (*Optional[str], optional*) – A local path to a directory with a hubconf.py. If provided, the model definition is imported from here. This is used when loading an external model from a model-bundle. Defaults to None.
- **loss_def_path** (*Optional[str], optional*) – A local path to a directory with a hubconf.py. If provided, the loss function definition is imported from here. This is used when loading an external loss function from a model-bundle. Defaults to None.
- **training** (*bool, optional*) – If False, the training apparatus (loss, optimizer, scheduler, logging, etc.) will not be set up and the model will be put into eval mode. If True, the training apparatus will be set up and the model will be put into training mode. Defaults to True.
- **output_dir** (*Optional[str]*) –

build_dataloaders() → Tuple[torch.utils.data.DataLoader, torch.utils.data.DataLoader, torch.utils.data.DataLoader]

Set the DataLoaders for train, validation, and test sets.

Return type
Tuple[torch.utils.data.DataLoader, torch.utils.data.DataLoader, torch.utils.data.DataLoader]

build_datasets() → Tuple[Dataset, Dataset, Dataset]

Return type
Tuple[Dataset, Dataset, Dataset]

build_epoch_scheduler(*start_epoch: int = 0*) → _LRScheduler

Returns an LR scheduler that changes the LR each epoch.

Parameters
start_epoch (*int*) –

Return type
_LRScheduler

build_loss(*loss_def_path: Optional[str] = None*) → Callable

Build a loss Callable.

Parameters
loss_def_path (*Optional[str]*) –

Return type
Callable

build_metric_names() → List[str]

Returns names of metrics used to validate model at each epoch.

Return type
List[str]

build_model(*model_def_path: Optional[str] = None*) → torch.nn.Module

Build a PyTorch model.

Parameters
model_def_path (*Optional[str]*) –

Return type
torch.nn.Module

build_optimizer() → Optimizer

Returns optimizer.

Return type
Optimizer

build_step_scheduler(*start_epoch: int = 0*) → _LRScheduler

Returns an LR scheduler that changes the LR each step.

Parameters
start_epoch (*int*) –

Return type
_LRScheduler

eval_model(*split: str*)

Evaluate model using a particular dataset split.

Gets validation metrics and saves them along with prediction plots.

Parameters
split (*str*) – the dataset split to use: train, valid, or test.

classmethod from_model_bundle(*model_bundle_uri: str*, *tmp_dir: Optional[str] = None*, *cfg: Optional[LearnerConfig] = None*, *training: bool = False*, ***kwargs*) → *Learner*

Create a Learner from a model bundle.

Note: This is the bundle saved in `train/model-bundle.zip` and not `bundle/model-bundle.zip`.

Parameters

- **model_bundle_uri** (*str*) – URI of the model bundle.
- **tmp_dir** (*Optional[str], optional*) – Optional temporary directory. Will be used for unzipping bundle and also passed to the default constructor. If None, will be auto-generated. Defaults to None.
- **cfg** (*Optional[LearnerConfig], optional*) – If None, will be read from the bundle. Defaults to None.
- **training** (*bool, optional*) – If False, the training apparatus (loss, optimizer, scheduler, logging, etc.) will not be set up and the model will be put into eval mode. If True, the training apparatus will be set up and the model will be put into training mode. Defaults to True.
- ****kwargs** – See *Learner.__init__()*.

Raises

FileNotFoundError – If using custom Albumentations transforms and definition file is not found in bundle.

Returns

Object of the Learner subclass on which this was called.

Return type

Learner

get_collate_fn() → Optional[callable]

Returns a custom collate_fn to use in DataLoader.

None is returned if default collate_fn should be used.

See https://pytorch.org/docs/stable/data.html#working-with-collate-fn

Return type

Optional[callable]

get_dataloader(*split: str*) → torch.utils.data.DataLoader

Get the DataLoader for a split.

Parameters

split (*str*) – a split name which can be train, valid, or test

Return type

torch.utils.data.DataLoader

get_start_epoch() → int

Get start epoch.

If training was interrupted, this returns the last complete epoch + 1.

Return type
int

`get_train_sampler`(*train_ds: Dataset*) → Optional[Sampler]

Return a sampler to use for the training dataloader or None to not use any.

Parameters
train_ds (*Dataset*) –

Return type
Optional[Sampler]

`get_visualizer_class`()

Returns a Visualizer class object for plotting data samples.

`load_checkpoint`()

Load last weights from previous run if available.

`load_init_weights`(*model_weights_path: Optional[str] = None*) → None

Load the weights to initialize model.

Parameters
model_weights_path (*Optional[str]*) –

Return type
None

`load_weights`(*uri: str*, ***kwargs*) → None

Load model weights from a file.

Parameters
uri (*str*) –

Return type
None

`log_data_stats`()

Log stats about each DataSet.

`main`()

Main training sequence.

This plots the dataset, runs a training and validation loop (which will resume if interrupted), logs stats, plots predictions, and syncs results to the cloud.

`normalize_input`(*x: ndarray*) → ndarray

Normalize x to [0, 1].

If x.dtype is a subtype of np.unsignedinteger, normalize it to [0, 1] using the max possible value of that dtype. Otherwise, assume it is in [0, 1] already and do nothing.

Parameters
x (*np.ndarray*) – an image or batch of images

Returns
the same array scaled to [0, 1].

Return type
ndarray

numpy_predict(*x: ndarray*, *raw_out: bool = False*) → ndarray

Make a prediction using an image or batch of images in numpy format. If x.dtype is a subtype of np.unsignedinteger, it will be normalized to [0, 1] using the max possible value of that dtype. Otherwise, x will be assumed to be in [0, 1] already and will be cast to torch.float32 directly.

Parameters

- **x** (*ndarray*) – (ndarray) of shape [height, width, channels] or [batch_sz, height, width, channels]
- **raw_out** (*bool*) – if True, return prediction probabilities

Returns
predictions using numpy arrays

Return type
ndarray

on_epoch_end(*curr_epoch*, *metrics*)

Hook that is called at end of epoch.

Writes metrics to CSV and TB, and saves model.

on_overfit_start()

Hook that is called at start of overfit routine.

on_train_start()

Hook that is called at start of train routine.

output_to_numpy(*out: torch.Tensor*) → ndarray

Convert output of model to numpy format.

Parameters
out (*torch.Tensor*) – the output of the model in PyTorch format

Return type
ndarray

Returns: the output of the model in numpy format

overfit()

Optimize model using the same batch repeatedly.

plot_dataloader(*dl: torch.utils.data.DataLoader*, *output_path: str*, *batch_limit: Optional[int] = None*, *show: bool = False*)

Plot images and ground truth labels for a DataLoader.

Parameters

- **dl** (*torch.utils.data.DataLoader*) –
- **output_path** (*str*) –
- **batch_limit** (*Optional[int]*) –
- **show** (*bool*) –

plot_dataloaders(*batch_limit: Optional[int] = None*, *show: bool = False*)

Plot images and ground truth labels for all DataLoaders.

Parameters

- **batch_limit** (*Optional[int]*) –

- **show** (*bool*) –

plot_predictions(*split: str*, *batch_limit: Optional[int] = None*, *show: bool = False*)

Plot predictions for a split.

Uses the first batch for the corresponding DataLoader.

Parameters

- **split** (*str*) – dataset split. Can be train, valid, or test.
- **batch_limit** (*Optional[int]*) – optional limit on (rendered) batch size
- **show** (*bool*) –

post_forward(*x*)

Post process output of call to model().

Useful for when predictions are inside a structure returned by model().

predict(*x: torch.Tensor*, *raw_out: bool = False*, *out_shape: Optional[Tuple[int, int]] = None*) → torch.Tensor

Make prediction for an image or batch of images.

Parameters

- **x** (*Tensor*) – Image or batch of images as a float Tensor with pixel values normalized to [0, 1].
- **raw_out** (*bool*) – if True, return prediction probabilities
- **out_shape** (*Optional[Tuple[int, int]]*) –

Returns

the predictions, in probability form if raw_out is True, in class_id form otherwise

Return type
torch.Tensor

predict_dataloader(*dl: torch.utils.data.DataLoader*, *batched_output: bool = True*, *return_format: Literal['xyz', 'yz', 'z'] = 'z'*, *raw_out: bool = True*, *predict_kw: dict = {}*) → Union[Iterator[Any], Iterator[Tuple[Any, ...]]]

Returns an iterator over predictions on the given dataloader.

Parameters

- **dl** (*DataLoader*) – The dataloader to make predictions on.
- **batched_output** (*bool, optional*) – If True, return batches of x, y, z as defined by the dataloader. If False, unroll the batches into individual items. Defaults to True.
- **return_format** (*Literal['xyz', 'yz', 'z'], optional*) – Format of the return elements of the returned iterator. Must be one of: 'xyz', 'yz', and 'z'. If 'xyz', elements are 3-tuples of x, y, and z. If 'yz', elements are 2-tuples of y and z. If 'z', elements are (non-tuple) values of z. Where x = input image, y = ground truth, and z = prediction. Defaults to 'z'.
- **raw_out** (*bool, optional*) – If true, return raw predicted scores. Defaults to True.
- **predict_kw** (*dict*) – Dict with keywords passed to Learner.predict(). Useful if a Learner subclass implements a custom predict() method.

Raises
ValueError – If return_format is not one of the allowed values.

Returns

If return_format
is 'z', the returned value is an iterator of whatever type the predictions are. Otherwise, the returned value is an iterator of tuples.

Return type
Union[Iterator[Any], Iterator[Tuple[Any, ...]]]

predict_dataset(*dataset: Dataset*, *return_format: Literal['xyz', 'yz', 'z'] = 'z'*, *raw_out: bool = True*, *numpy_out: bool = False*, *predict_kw: dict = {}*, *dataloader_kw: dict = {}*, *progress_bar: bool = True*, *progress_bar_kw: dict = {}*) → Union[Iterator[Any], Iterator[Tuple[Any, ...]]]

Returns an iterator over predictions on the given dataset.

Parameters

- **dataset** (*Dataset*) – The dataset to make predictions on.
- **return_format** (*Literal['xyz', 'yz', 'z'], optional*) – Format of the return elements of the returned iterator. Must be one of: 'xyz', 'yz', and 'z'. If 'xyz', elements are 3-tuples of x, y, and z. If 'yz', elements are 2-tuples of y and z. If 'z', elements are (non-tuple) values of z. Where x = input image, y = ground truth, and z = prediction. Defaults to 'z'.
- **raw_out** (*bool, optional*) – If true, return raw predicted scores. Defaults to True.
- **numpy_out** (*bool, optional*) – If True, convert predictions to numpy arrays before returning. Defaults to False.
- **predict_kw** (*dict*) – Dict with keywords passed to Learner.predict(). Useful if a Learner subclass implements a custom predict() method.
- **dataloader_kw** (*dict*) – Dict with keywords passed to the DataLoader constructor.
- **progress_bar** (*bool, optional*) – If True, display a progress bar. Since this function returns an iterator, the progress bar won't be visible until the iterator is consumed. Defaults to True.
- **progress_bar_kw** (*dict*) – Dict with keywords passed to tqdm.

Raises
ValueError – If return_format is not one of the allowed values.

Returns

If return_format
is 'z', the returned value is an iterator of whatever type the predictions are. Otherwise, the returned value is an iterator of tuples.

Return type
Union[Iterator[Any], Iterator[Tuple[Any, ...]]]

prob_to_pred(*x*)

Convert a Tensor with prediction probabilities to class ids.

The class ids should be the classes with the maximum probability.

run_tensorboard()

Run TB server serving logged stats.

save_model_bundle()

Save a model bundle.

This is a zip file with the model weights in .pth format and a serialized copy of the LearningConfig, which allows for making predictions in the future.

setup_data()

Set datasets and dataLoaders for train, validation, and test sets.

setup_loss(*loss_def_path: Optional[str] = None*) → None

Setup self.loss.

Parameters

- **loss_def_path** (*str, optional*) – Loss definition path. Will be
- **None.** (*available when loading from a bundle. Defaults to*) –

Return type
None

setup_model(*model_weights_path: Optional[str] = None*, *model_def_path: Optional[str] = None*) → None

Setup self.model.

Parameters

- **model_weights_path** (*Optional[str], optional*) – Path to model weights. Will be available when loading from a bundle. Defaults to None.
- **model_def_path** (*Optional[str], optional*) – Path to model definition. Will be available when loading from a bundle. Defaults to None.

Return type
None

setup_tensorboard()

Setup for logging stats to TB.

setup_training(*loss_def_path: Optional[str] = None*) → None

Parameters
loss_def_path (*Optional[str]*) –

Return type
None

stop_tensorboard()

Stop TB logging and server if it's running.

sync_from_cloud()

Sync any previous output in the cloud to output_dir.

sync_to_cloud()

Sync any output to the cloud at output_uri.

to_batch(*x: torch.Tensor*) → torch.Tensor

Ensure that image array has batch dimension.

Parameters
x (*torch.Tensor*) – assumed to be either image or batch of images

Returns
x with extra batch dimension of length 1 if needed

Return type
torch.Tensor

to_device(*x: Any*, *device: str*) → Any

Load Tensors onto a device.

Parameters

- **x** (*Any*) – some object with Tensors in it
- **device** (*str*) – 'cpu' or 'cuda'

Returns
x but with any Tensors in it on the device

Return type
Any

train(*epochs: Optional[int] = None*)

Training loop that will attempt to resume training if appropriate.

Parameters
epochs (*Optional[int]*) –

train_end(*outputs: List[Dict[str, float]]*, *num_samples: int*) → Dict[str, float]

Aggregate the ouput of train_step at the end of the epoch.

Parameters

- **outputs** (*List[Dict[str, float]]*) – a list of outputs of train_step
- **num_samples** (*int*) – total number of training samples processed in epoch

Return type
Dict[str, float]

train_epoch(*optimizer: Optimizer*, *step_scheduler: Optional[_LRScheduler] = None*) → Dict[str, float]

Train for a single epoch.

Parameters

- **optimizer** (*Optimizer*) –
- **step_scheduler** (*Optional[_LRScheduler]*) –

Return type
Dict[str, float]

train_step(*batch*, *batch_ind*)

Compute loss for a single training batch.

Parameters

- **batch** – batch data needed to compute loss
- **batch_ind** – index of batch within epoch

Returns
dict with 'train_loss' as key and possibly other losses

validate_end(*outputs*, *num_samples*)

Aggregate the ouput of validate_step at the end of the epoch.

Parameters

- **outputs** – a list of outputs of validate_step

- **num_samples** – total number of validation samples processed in epoch

validate_epoch(*dl: torch.utils.data.DataLoader*) → Dict[str, float]

Validate for a single epoch.

Parameters
dl (*torch.utils.data.DataLoader*) –

Return type
Dict[str, float]

validate_step(*batch*, *batch_ind*)

Compute metrics on validation batch.

Parameters

- **batch** – batch data needed to compute validation metrics
- **batch_ind** – index of batch within epoch

Returns
dict with metric names mapped to metric values

9.3.14 semantic_segmentation_learner_config

Classes

SemanticSegmentationDataFormat	An enumeration.

SemanticSegmentationDataFormat

class SemanticSegmentationDataFormat

Bases: Enum

An enumeration.

Attributes

default

__init__()

default = 'default'

Configs

SemanticSegmentationDataConfig

SemanticSegmentationGeoDataConfig	Configure semantic segmentation *GeoDatasets*.
SemanticSegmentationImageDataConfig	Configure *SemanticSegmentationImageDatasets*.
SemanticSegmentationLearnerConfig	Configure a *SemanticSegmentationLearner*.
SemanticSegmentationModelConfig	Configure a semantic segmentation model.

SemanticSegmentationDataConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model SemanticSegmentationDataConfig


```
{
   "title": "SemanticSegmentationDataConfig",
   "description": "Base class that can be extended to provide custom configurations.
↪\n\nThis adds some extra methods to Pydantic BaseModel.\nSee https://pydantic-
↪docs.helpmanual.io/\n\nThe general idea is that configuration schemas can be␣
↪defined by\nsubclassing this and adding class attributes with types and\ndefault␣
↪values for each field. Configs can be defined hierarchically,\nie. a Config can␣
↪have fields which are of type Config.\nValidation, serialization, deserialization,
↪ and IDE support is\nprovided automatically based on this schema.",
   "type": "object",
   "properties": {
      "type_hint": {
         "title": "Type Hint",
         "default": "semantic_segmentation_data",
         "enum": [
            "semantic_segmentation_data"
         ],
         "type": "string"
      }
   },
   "additionalProperties": false
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *type_hint (Literal['semantic_segmentation_data'])*

field type_hint: Literal['semantic_segmentation_data'] = 'semantic_segmentation_data'

build()

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(**args*, ***kwargs*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

SemanticSegmentationGeoDataConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model SemanticSegmentationGeoDataConfig

Configure semantic segmentation *GeoDatasets*.

See *rastervision.pytorch_learner.dataset.semantic_segmentation_dataset*.


```
{
   "title": "SemanticSegmentationGeoDataConfig",
   "description": "Configure semantic segmentation :class:`GeoDatasets <.GeoDataset>
↪`.\n\nSee\n:mod:`rastervision.pytorch_learner.dataset.semantic_segmentation_
↪dataset`.",
   "type": "object",
```


(continues on next page)

(continued from previous page)

```
   "properties": {
      "class_names": {
         "title": "Class Names",
         "description": "Names of classes.",
         "default": [],
         "type": "array",
         "items": {
            "type": "string"
         }
      },
      "class_colors": {
         "title": "Class Colors",
         "description": "Colors used to display classes. Can be color 3-tuples in␣
↪list form.",
         "type": "array",
         "items": {
            "anyOf": [
               {
                  "type": "string"
               },
               {
                  "type": "array",
                  "minItems": 3,
                  "maxItems": 3,
                  "items": [
                     {
                        "type": "integer"
                     },
                     {
                        "type": "integer"
                     },
                     {
                        "type": "integer"
                     }
                  ]
               }
            ]
         }
      },
      "img_channels": {
         "title": "Img Channels",
         "description": "The number of channels of the training images.",
         "exclusiveMinimum": 0,
         "type": "integer"
      },
      "img_sz": {
         "title": "Img Sz",
         "description": "Length of a side of each image in pixels. This is the size␣
↪to transform it to during training, not the size in the raw dataset.",
         "default": 256,
         "exclusiveMinimum": 0,
         "type": "integer"
```

(continues on next page)

(continued from previous page)

```
      },
      "train_sz": {
         "title": "Train Sz",
         "description": "If set, the number of training images to use. If fewer␣
↪images exist, then an exception will be raised.",
         "type": "integer"
      },
      "train_sz_rel": {
         "title": "Train Sz Rel",
         "description": "If set, the proportion of training images to use.",
         "type": "number"
      },
      "num_workers": {
         "title": "Num Workers",
         "description": "Number of workers to use when DataLoader makes batches.",
         "default": 4,
         "type": "integer"
      },
      "augmentors": {
         "title": "Augmentors",
         "description": "Names of albumentations augmentors to use for training␣
↪batches. Choices include: ['Blur', 'RandomRotate90', 'HorizontalFlip',
↪'VerticalFlip', 'GaussianBlur', 'GaussNoise', 'RGBShift', 'ToGray'].␣
↪Alternatively, a custom transform can be provided via the aug_transform option.",
         "default": [
            "RandomRotate90",
            "HorizontalFlip",
            "VerticalFlip"
         ],
         "type": "array",
         "items": {
            "type": "string"
         }
      },
      "base_transform": {
         "title": "Base Transform",
         "description": "An Albumentations transform serialized as a dict that will␣
↪be applied to all datasets: training, validation, and test. This transformation␣
↪is in addition to the resizing due to img_sz. This is useful for, for example,␣
↪applying the same normalization to all datasets.",
         "type": "object"
      },
      "aug_transform": {
         "title": "Aug Transform",
         "description": "An Albumentations transform serialized as a dict that will␣
↪be applied as data augmentation to the training dataset. This transform is␣
↪applied before base_transform. If provided, the augmentors option is ignored.",
         "type": "object"
      },
      "plot_options": {
         "title": "Plot Options",
         "description": "Options to control plotting.",
```

(continues on next page)

(continued from previous page)


```
        "default": {
           "transform": {
              "__version__": "1.3.0",
              "transform": {
                 "__class_fullname__": "rastervision.pytorch_learner.utils.utils.
↪MinMaxNormalize",
                 "always_apply": false,
                 "p": 1.0,
                 "min_val": 0.0,
                 "max_val": 1.0,
                 "dtype": 5
              }
           },
           "channel_display_groups": null,
           "type_hint": "plot_options"
        },
        "allOf": [
           {
              "$ref": "#/definitions/PlotOptions"
           }
        ]
     },
     "preview_batch_limit": {
        "title": "Preview Batch Limit",
        "description": "Optional limit on the number of items in the preview plots␣
↪produced during training.",
        "type": "integer"
     },
     "type_hint": {
        "title": "Type Hint",
        "default": "semantic_segmentation_geo_data",
        "enum": [
           "semantic_segmentation_geo_data"
        ],
        "type": "string"
     },
     "scene_dataset": {
        "$ref": "#/definitions/DatasetConfig"
     },
     "window_opts": {
        "title": "Window Opts",
        "default": {},
        "anyOf": [
           {
              "$ref": "#/definitions/GeoDataWindowConfig"
           },
           {
              "type": "object",
              "additionalProperties": {
                 "$ref": "#/definitions/GeoDataWindowConfig"
              }
           }
```


(continues on next page)

(continued from previous page)


```
        ]
      }
   },
   "additionalProperties": false,
   "definitions": {
      "PlotOptions": {
         "title": "PlotOptions",
         "description": "Config related to plotting.",
         "type": "object",
         "properties": {
            "transform": {
               "title": "Transform",
               "description": "An Albumentations transform serialized as a dict␣
↪that will be applied to each image before it is plotted. Mainly useful for␣
↪undoing any data transformation that you do not want included in the plot, such␣
↪as normalization. The default value will shift and scale the image so the values␣
↪range from 0.0 to 1.0 which is the expected range for the plotting function. This␣
↪default is useful for cases where the values after normalization are close to␣
↪zero which makes the plot difficult to see.",
               "default": {
                  "__version__": "1.3.0",
                  "transform": {
                     "__class_fullname__": "rastervision.pytorch_learner.utils.
↪utils.MinMaxNormalize",
                     "always_apply": false,
                     "p": 1.0,
                     "min_val": 0.0,
                     "max_val": 1.0,
                     "dtype": 5
                  }
               },
               "type": "object"
            },
            "channel_display_groups": {
               "title": "Channel Display Groups",
               "description": "Groups of image channels to display together as a␣
↪subplot when plotting the data and predictions. Can be a list or tuple of groups␣
↪(e.g. [(0, 1, 2), (3,)]) or a dict containing title-to-group mappings (e.g. {\
↪"RGB\": [0, 1, 2], \"IR\": [3]}), where each group is a list or tuple of channel␣
↪indices and title is a string that will be used as the title of the subplot for␣
↪that group.",
               "anyOf": [
                  {
                     "type": "object",
                     "additionalProperties": {
                        "type": "array",
                        "items": {
                           "type": "integer",
                           "minimum": 0
                        }
                     }
                  },
```


(continues on next page)

(continued from previous page)

```
                {
                    "type": "array",
                    "items": {
                        "type": "array",
                        "items": {
                            "type": "integer",
                            "minimum": 0
                        }
                    }
                }
            ]
        },
        "type_hint": {
            "title": "Type Hint",
            "default": "plot_options",
            "enum": [
                "plot_options"
            ],
            "type": "string"
        }
    },
    "additionalProperties": false
},
"ClassConfig": {
    "title": "ClassConfig",
    "description": "Configure class information for a machine learning task.",
    "type": "object",
    "properties": {
        "names": {
            "title": "Names",
            "description": "Names of classes. The i-th class in this list will␣
↪have class ID = i.",
            "type": "array",
            "items": {
                "type": "string"
            }
        },
        "colors": {
            "title": "Colors",
            "description": "Colors used to visualize classes. Can be color␣
↪strings accepted by matplotlib or RGB tuples. If None, a random color will be␣
↪auto-generated for each class.",
            "type": "array",
            "items": {
                "anyOf": [
                    {
                        "type": "string"
                    },
                    {
                        "type": "array",
                        "items": {}
                    }
```

(continues on next page)

(continued from previous page)


```
                    ]
                }
            },
            "null_class": {
                "title": "Null Class",
                "description": "Optional name of class in `names` to use as the null␣
↪class. This is used in semantic segmentation to represent the label for imagery␣
↪pixels that are NODATA or that are missing a label. If None and the class names␣
↪include \"null\", it will automatically be used as the null class. If None, and␣
↪this Config is part of a SemanticSegmentationConfig, a null class will be added␣
↪automatically.",
                "type": "string"
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "class_config",
                "enum": [
                    "class_config"
                ],
                "type": "string"
            }
        },
        "required": [
            "names"
        ],
        "additionalProperties": false
    },
    "RasterTransformerConfig": {
        "title": "RasterTransformerConfig",
        "description": "Configure a :class:`.RasterTransformer`.",
        "type": "object",
        "properties": {
            "type_hint": {
                "title": "Type Hint",
                "default": "raster_transformer",
                "enum": [
                    "raster_transformer"
                ],
                "type": "string"
            }
        },
        "additionalProperties": false
    },
    "RasterSourceConfig": {
        "title": "RasterSourceConfig",
        "description": "Configure a :class:`.RasterSource`.",
        "type": "object",
        "properties": {
            "channel_order": {
                "title": "Channel Order",
                "description": "The sequence of channel indices to use when reading␣
↪imagery.",
```


(continues on next page)

(continued from previous page)

```
                "type": "array",
                "items": {
                   "type": "integer"
                }
             },
             "transformers": {
                "title": "Transformers",
                "default": [],
                "type": "array",
                "items": {
                   "$ref": "#/definitions/RasterTransformerConfig"
                }
             },
             "extent": {
                "title": "Extent",
                "description": "Use-specified extent in pixel coords in the form␣
↪(ymin, xmin, ymax, xmax). Useful for cropping the raster source so that only part␣
↪of the raster is read from.",
                "type": "array",
                "minItems": 4,
                "maxItems": 4,
                "items": [
                   {
                      "type": "integer"
                   },
                   {
                      "type": "integer"
                   },
                   {
                      "type": "integer"
                   },
                   {
                      "type": "integer"
                   }
                ]
             },
             "type_hint": {
                "title": "Type Hint",
                "default": "raster_source",
                "enum": [
                   "raster_source"
                ],
                "type": "string"
             }
          },
          "additionalProperties": false
       },
       "LabelSourceConfig": {
          "title": "LabelSourceConfig",
          "description": "Configure a :class:`.LabelSource`.",
          "type": "object",
          "properties": {
```

(continues on next page)

(continued from previous page)


```
            "type_hint": {
               "title": "Type Hint",
               "default": "label_source",
               "enum": [
                  "label_source"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      },
      "LabelStoreConfig": {
         "title": "LabelStoreConfig",
         "description": "Configure a :class:`.LabelStore`.",
         "type": "object",
         "properties": {
            "type_hint": {
               "title": "Type Hint",
               "default": "label_store",
               "enum": [
                  "label_store"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      },
      "SceneConfig": {
         "title": "SceneConfig",
         "description": "Configure a :class:`.Scene` comprising raster data &␣
↪labels for an AOI.\n    ",
         "type": "object",
         "properties": {
            "id": {
               "title": "Id",
               "type": "string"
            },
            "raster_source": {
               "$ref": "#/definitions/RasterSourceConfig"
            },
            "label_source": {
               "$ref": "#/definitions/LabelSourceConfig"
            },
            "label_store": {
               "$ref": "#/definitions/LabelStoreConfig"
            },
            "aoi_uris": {
               "title": "Aoi Uris",
               "description": "List of URIs of GeoJSON files that define the AOIs␣
↪for the scene. Each polygon defines an AOI which is a piece of the scene that is␣
↪assumed to be fully labeled and usable for training or validation. The AOIs are␣
↪assumed to be in EPSG:4326 coordinates.",
```


(continues on next page)

(continued from previous page)


```
                "type": "array",
                "items": {
                   "type": "string"
                }
             },
             "type_hint": {
                "title": "Type Hint",
                "default": "scene",
                "enum": [
                   "scene"
                ],
                "type": "string"
             }
          },
          "required": [
             "id",
             "raster_source"
          ],
          "additionalProperties": false
       },
       "DatasetConfig": {
          "title": "DatasetConfig",
          "description": "Configure train, validation, and test splits for a dataset.
↪",
          "type": "object",
          "properties": {
             "class_config": {
                "$ref": "#/definitions/ClassConfig"
             },
             "train_scenes": {
                "title": "Train Scenes",
                "type": "array",
                "items": {
                   "$ref": "#/definitions/SceneConfig"
                }
             },
             "validation_scenes": {
                "title": "Validation Scenes",
                "type": "array",
                "items": {
                   "$ref": "#/definitions/SceneConfig"
                }
             },
             "test_scenes": {
                "title": "Test Scenes",
                "default": [],
                "type": "array",
                "items": {
                   "$ref": "#/definitions/SceneConfig"
                }
             },
             "scene_groups": {
```


(continues on next page)

(continued from previous page)


```
                "title": "Scene Groups",
                "description": "Groupings of scenes. Should be a dict of the form: {
↪<group-name>: Set(scene_id_1, scene_id_2, ...)}. Three groups are added by␣
↪default: \"train_scenes\", \"validation_scenes\", and \"test_scenes\"",
                "default": {},
                "type": "object",
                "additionalProperties": {
                   "type": "array",
                   "items": {
                      "type": "string"
                   },
                   "uniqueItems": true
                }
             },
             "type_hint": {
                "title": "Type Hint",
                "default": "dataset",
                "enum": [
                   "dataset"
                ],
                "type": "string"
             }
          },
          "required": [
             "class_config",
             "train_scenes",
             "validation_scenes"
          ],
          "additionalProperties": false
       },
       "GeoDataWindowMethod": {
          "title": "GeoDataWindowMethod",
          "description": "An enumeration.",
          "enum": [
             "sliding",
             "random"
          ]
       },
       "GeoDataWindowConfig": {
          "title": "GeoDataWindowConfig",
          "description": "Configure a :class:`.GeoDataset`.\n\nSee :mod:
↪`rastervision.pytorch_learner.dataset.dataset`.",
          "type": "object",
          "properties": {
             "method": {
                "default": "sliding",
                "allOf": [
                   {
                      "$ref": "#/definitions/GeoDataWindowMethod"
                   }
                ]
             },
```


(continues on next page)

(continued from previous page)

```
            "size": {
               "title": "Size",
               "description": "If method = sliding, this is the size of sliding␣
↪window. If method = random, this is the size that all the windows are resized to␣
↪before they are returned. If method = random and neither size_lims nor h_lims and␣
↪w_lims have been specified, then size_lims is set to (size, size + 1).",
               "anyOf": [
                  {
                     "type": "integer",
                     "exclusiveMinimum": 0
                  },
                  {
                     "type": "array",
                     "minItems": 2,
                     "maxItems": 2,
                     "items": [
                        {
                           "type": "integer",
                           "exclusiveMinimum": 0
                        },
                        {
                           "type": "integer",
                           "exclusiveMinimum": 0
                        }
                     ]
                  }
               ]
            },
            "stride": {
               "title": "Stride",
               "description": "Stride of sliding window. Only used if method =␣
↪sliding.",
               "anyOf": [
                  {
                     "type": "integer",
                     "exclusiveMinimum": 0
                  },
                  {
                     "type": "array",
                     "minItems": 2,
                     "maxItems": 2,
                     "items": [
                        {
                           "type": "integer",
                           "exclusiveMinimum": 0
                        },
                        {
                           "type": "integer",
                           "exclusiveMinimum": 0
                        }
                     ]
                  }
```

(continues on next page)

(continued from previous page)

```
            ]
         },
         "padding": {
            "title": "Padding",
            "description": "How many pixels are windows allowed to overflow the␣
↪edges of the raster source.",
            "anyOf": [
               {
                  "type": "integer",
                  "minimum": 0
               },
               {
                  "type": "array",
                  "minItems": 2,
                  "maxItems": 2,
                  "items": [
                     {
                        "type": "integer",
                        "minimum": 0
                     },
                     {
                        "type": "integer",
                        "minimum": 0
                     }
                  ]
               }
            ]
         },
         "pad_direction": {
            "title": "Pad Direction",
            "description": "If \"end\", only pad ymax and xmax (bottom and␣
↪right). If \"start\", only pad ymin and xmin (top and left). If \"both\", pad all␣
↪sides. Has no effect if paddiong is zero. Defaults to \"end\".",
            "default": "end",
            "enum": [
               "both",
               "start",
               "end"
            ],
            "type": "string"
         },
         "size_lims": {
            "title": "Size Lims",
            "description": "[min, max) interval from which window sizes will be␣
↪uniformly randomly sampled. The upper limit is exclusive. To fix the size to a␣
↪constant value, use size_lims = (sz, sz + 1). Only used if method = random.␣
↪Specify either size_lims or h_lims and w_lims, but not both. If neither size_lims␣
↪nor h_lims and w_lims have been specified, then this will be set to (size, size +␣
↪1).",
            "type": "array",
            "minItems": 2,
            "maxItems": 2,
```

(continues on next page)

(continued from previous page)

```
                "items": [
                    {
                        "type": "integer",
                        "exclusiveMinimum": 0
                    },
                    {
                        "type": "integer",
                        "exclusiveMinimum": 0
                    }
                ]
            },
            "h_lims": {
                "title": "H Lims",
                "description": "[min, max] interval from which window heights will␣
↪be uniformly randomly sampled. Only used if method = random.",
                "type": "array",
                "minItems": 2,
                "maxItems": 2,
                "items": [
                    {
                        "type": "integer",
                        "exclusiveMinimum": 0
                    },
                    {
                        "type": "integer",
                        "exclusiveMinimum": 0
                    }
                ]
            },
            "w_lims": {
                "title": "W Lims",
                "description": "[min, max] interval from which window widths will be␣
↪uniformly randomly sampled. Only used if method = random.",
                "type": "array",
                "minItems": 2,
                "maxItems": 2,
                "items": [
                    {
                        "type": "integer",
                        "exclusiveMinimum": 0
                    },
                    {
                        "type": "integer",
                        "exclusiveMinimum": 0
                    }
                ]
            },
            "max_windows": {
                "title": "Max Windows",
                "description": "Max allowed reads from a GeoDataset. Only used if␣
↪method = random.",
                "default": 10000,
```

(continues on next page)

(continued from previous page)


```
                "minimum": 0,
                "type": "integer"
            },
            "max_sample_attempts": {
                "title": "Max Sample Attempts",
                "description": "Max attempts when trying to find a window within the␣
↪AOI of a scene. Only used if method = random and the scene has aoi_polygons␣
↪specified.",
                "default": 100,
                "exclusiveMinimum": 0,
                "type": "integer"
            },
            "efficient_aoi_sampling": {
                "title": "Efficient Aoi Sampling",
                "description": "If the scene has AOIs, sampling windows at random␣
↪anywhere in the extent and then checking if they fall within any of the AOIs can␣
↪be very inefficient. This flag enables the use of an alternate algorithm that␣
↪only samples window locations inside the AOIs. Only used if method = random and␣
↪the scene has aoi_polygons specified. Defaults to True",
                "default": true,
                "type": "boolean"
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "geo_data_window",
                "enum": [
                    "geo_data_window"
                ],
                "type": "string"
            }
        },
        "required": [
            "size"
        ],
        "additionalProperties": false
    }
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *aug_transform (Optional[dict])*
- *augmentors (List[str])*
- *base_transform (Optional[dict])*
- *class_colors (Optional[List[Union[str, Tuple[int, int, int]]]])*
- *class_names (List[str])*

- *img_channels (Optional[pydantic.types.PositiveInt])*
- *img_sz (pydantic.types.PositiveInt)*
- *num_workers (int)*
- *plot_options (Optional[rastervision.pytorch_learner.learner_config.PlotOptions])*
- *preview_batch_limit (Optional[int])*
- *scene_dataset (Optional[rastervision.core.data.dataset_config.DatasetConfig])*
- *train_sz (Optional[int])*
- *train_sz_rel (Optional[float])*
- *type_hint (Literal['semantic_segmentation_geo_data'])*
- *window_opts (Union[rastervision.pytorch_learner.learner_config.GeoDataWindowConfig, Dict[str, rastervision.pytorch_learner.learner_config.GeoDataWindowConfig]])*

Validators

- ensure_class_colors » all fields
- get_class_info_from_class_config_if_needed » all fields
- *validate_albumentation_transform* » *aug_transform*
- *validate_albumentation_transform* » *base_transform*
- validate_augmentors » *augmentors*
- validate_plot_options » all fields
- validate_window_opts » *window_opts*

field aug_transform: Optional[dict] = None

An Albumentations transform serialized as a dict that will be applied as data augmentation to the training dataset. This transform is applied before base_transform. If provided, the augmentors option is ignored.

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- *validate_albumentation_transform*
- validate_plot_options

field augmentors: List[str] = ['RandomRotate90', 'HorizontalFlip', 'VerticalFlip']

Names of albumentations augmentors to use for training batches. Choices include: [‘Blur’, ‘RandomRotate90’, ‘HorizontalFlip’, ‘VerticalFlip’, ‘GaussianBlur’, ‘GaussNoise’, ‘RGBShift’, ‘ToGray’]. Alternatively, a custom transform can be provided via the aug_transform option.

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_augmentors

- validate_plot_options

field base_transform: Optional[dict] = None

An Albumentations transform serialized as a dict that will be applied to all datasets: training, validation, and test. This transformation is in addition to the resizing due to img_sz. This is useful for, for example, applying the same normalization to all datasets.

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- *validate_albumentation_transform*
- validate_plot_options

field class_colors: Optional[List[Union[str, RGBTuple]]] = None

Colors used to display classes. Can be color 3-tuples in list form.

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field class_names: List[str] = []

Names of classes.

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field img_channels: Optional[PosInt] = None

The number of channels of the training images.

Constraints

- **exclusiveMinimum** = 0

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field img_sz: PosInt = 256

Length of a side of each image in pixels. This is the size to transform it to during training, not the size in the raw dataset.

Constraints

- **exclusiveMinimum** = 0

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed

- validate_plot_options

field num_workers: int = 4

Number of workers to use when DataLoader makes batches.

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field plot_options: Optional[*PlotOptions*] = PlotOptions(transform={'__version__': '1.3.0', 'transform': {'__class_fullname__': 'rastervision.pytorch_learner.utils.utils.MinMaxNormalize', 'always_apply': False, 'p': 1.0, 'min_val': 0.0, 'max_val': 1.0, 'dtype': 5}}, channel_display_groups=None)

Options to control plotting.

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field preview_batch_limit: Optional[int] = None

Optional limit on the number of items in the preview plots produced during training.

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field scene_dataset: Optional['SceneDatasetConfig'] = None

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field train_sz: Optional[int] = None

If set, the number of training images to use. If fewer images exist, then an exception will be raised.

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field train_sz_rel: Optional[float] = None

If set, the proportion of training images to use.

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed

- validate_plot_options

field type_hint: Literal['semantic_segmentation_geo_data'] = 'semantic_segmentation_geo_data'

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options

field window_opts: Union[*GeoDataWindowConfig*, Dict[str, *GeoDataWindowConfig*]] = {}

Validated by

- ensure_class_colors
- get_class_info_from_class_config_if_needed
- validate_plot_options
- validate_window_opts

build(*tmp_dir: str*, *overfit_mode: bool = False*, *test_mode: bool = False*) → Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

Parameters

- **tmp_dir** (*str*) –
- **overfit_mode** (*bool*) –
- **test_mode** (*bool*) –

Return type
Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

build_scenes(*tmp_dir: str*) → Tuple[List[*Scene*], List[*Scene*], List[*Scene*]]

Build training, validation, and test scenes.

Parameters
tmp_dir (*str*) –

Return type
Tuple[*List*[Scene], *List*[Scene], *List*[Scene]]

validator ensure_class_colors » *all fields*

Parameters
values (*dict*) –

Return type
dict

get_bbox_params() → Optional[BboxParams]

Returns BboxParams used by albumentations for data augmentation.

Return type
Optional[*BboxParams*]

validator get_class_info_from_class_config_if_needed » *all fields*

Parameters
values (*dict*) –

Return type
dict

get_custom_albumentations_transforms() → List[dict]

Returns all custom transforms found in this config.

This should return all serialized albumentations transforms with a 'lambda_transforms_path' field contained in this config or in any of its members no matter how deeply neseted.

The pupose is to make it easier to adjust their paths all at once while saving to or loading from a bundle.

Return type
List[dict]

get_data_transforms() → Tuple[BasicTransform, BasicTransform]

Get albumentations transform objects for data augmentation.

Returns
a transform that doesn't do any data augmentation 2nd tuple arg: a transform with data augmentation

Return type
1st tuple arg

make_datasets(*tmp_dir: str*, *train_tf: Optional[BasicTransform] = None*, *val_tf: Optional[BasicTransform] = None*, *test_tf: Optional[BasicTransform] = None*, ***kwargs*) → Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

Make training, validation, and test datasets.

Parameters

- **tmp_dir** (*str*) – Temporary directory to be used for building scenes.
- **train_tf** (*Optional[A.BasicTransform], optional*) – Transform for the training dataset. Defaults to None.
- **val_tf** (*Optional[A.BasicTransform], optional*) – Transform for the validation dataset. Defaults to None.
- **test_tf** (*Optional[A.BasicTransform], optional*) – Transform for the test dataset. Defaults to None.
- **kwargs** – Kwargs to pass to self.scene_to_dataset()

Returns

PyTorch-compatiable training,
validation, and test datasets.

Return type
Tuple[Dataset, Dataset, Dataset]

random_subset_dataset(*ds: torch.utils.data.Dataset*, *size: Optional[int] = None*, *fraction: Optional[ConstrainedFloatValue] = None*) → torch.utils.data.Subset

Parameters

- **ds** (*torch.utils.data.Dataset*) –

- **size** (*Optional[int]*) –
- **fraction** (*Optional[ConstrainedFloatValue]*) –

Return type
torch.utils.data.Subset

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

scene_to_dataset(*scene:* Scene, *transform: Optional[BasicTransform] = None*) → torch.utils.data.Dataset

Make a dataset from a single scene.

Parameters

- **scene** (Scene) –
- **transform** (*Optional[BasicTransform]*) –

Return type
torch.utils.data.Dataset

update(**args*, ***kwargs*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validator validate_augmentors » augmentors

Parameters
v (*str*) –

Return type
str

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises
ConfigError – if field is invalid

validator validate_plot_options » *all fields*

Parameters
values (*dict*) –

Return type
dict

validator validate_window_opts » window_opts

Parameters

- **v** (*Union[GeoDataWindowConfig, Dict[str, GeoDataWindowConfig]]*) –
- **values** (*dict*) –

Return type
Union[GeoDataWindowConfig, *Dict*[str, GeoDataWindowConfig]]

property num_classes

SemanticSegmentationImageDataConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model SemanticSegmentationImageDataConfig

Configure *SemanticSegmentationImageDatasets*.

This assumes the following file structure:

```
<data_dir>/
    img/
        <img 1>.<extension>
        <img 2>.<extension>
        ...
        <img N>.<extension>
    labels/
        <img 1>.<extension>
        <img 2>.<extension>
        ...
        <img N>.<extension>
```


```
{
   "title": "SemanticSegmentationImageDataConfig",
   "description": "Configure :class:`SemanticSegmentationImageDatasets <.
↪SemanticSegmentationImageDataset>`.\n\nThis assumes the following file structure:\
↪n\n.. code-block:: text\n\n    <data_dir>/\n        img/\n            <img 1>.
↪<extension>\n            <img 2>.<extension>\n            ...\n            <img N>
↪.<extension>\n        labels/\n            <img 1>.<extension>\n            <img␣
↪2>.<extension>\n            ...\n            <img N>.<extension>",
   "type": "object",
   "properties": {
      "class_names": {
         "title": "Class Names",
```


(continues on next page)

(continued from previous page)

```
         "description": "Names of classes.",
         "default": [],
         "type": "array",
         "items": {
            "type": "string"
         }
      },
      "class_colors": {
         "title": "Class Colors",
         "description": "Colors used to display classes. Can be color 3-tuples in␣
↪list form.",
         "type": "array",
         "items": {
            "anyOf": [
               {
                  "type": "string"
               },
               {
                  "type": "array",
                  "minItems": 3,
                  "maxItems": 3,
                  "items": [
                     {
                        "type": "integer"
                     },
                     {
                        "type": "integer"
                     },
                     {
                        "type": "integer"
                     }
                  ]
               }
            ]
         }
      },
      "img_channels": {
         "title": "Img Channels",
         "description": "The number of channels of the training images.",
         "exclusiveMinimum": 0,
         "type": "integer"
      },
      "img_sz": {
         "title": "Img Sz",
         "description": "Length of a side of each image in pixels. This is the size␣
↪to transform it to during training, not the size in the raw dataset.",
         "default": 256,
         "exclusiveMinimum": 0,
         "type": "integer"
      },
      "train_sz": {
         "title": "Train Sz",
```

(continues on next page)

(continued from previous page)


```
         "description": "If set, the number of training images to use. If fewer␣
↪images exist, then an exception will be raised.",
         "type": "integer"
      },
      "train_sz_rel": {
         "title": "Train Sz Rel",
         "description": "If set, the proportion of training images to use.",
         "type": "number"
      },
      "num_workers": {
         "title": "Num Workers",
         "description": "Number of workers to use when DataLoader makes batches.",
         "default": 4,
         "type": "integer"
      },
      "augmentors": {
         "title": "Augmentors",
         "description": "Names of albumentations augmentors to use for training␣
↪batches. Choices include: ['Blur', 'RandomRotate90', 'HorizontalFlip',
↪'VerticalFlip', 'GaussianBlur', 'GaussNoise', 'RGBShift', 'ToGray'].␣
↪Alternatively, a custom transform can be provided via the aug_transform option.",
         "default": [
            "RandomRotate90",
            "HorizontalFlip",
            "VerticalFlip"
         ],
         "type": "array",
         "items": {
            "type": "string"
         }
      },
      "base_transform": {
         "title": "Base Transform",
         "description": "An Albumentations transform serialized as a dict that will␣
↪be applied to all datasets: training, validation, and test. This transformation␣
↪is in addition to the resizing due to img_sz. This is useful for, for example,␣
↪applying the same normalization to all datasets.",
         "type": "object"
      },
      "aug_transform": {
         "title": "Aug Transform",
         "description": "An Albumentations transform serialized as a dict that will␣
↪be applied as data augmentation to the training dataset. This transform is␣
↪applied before base_transform. If provided, the augmentors option is ignored.",
         "type": "object"
      },
      "plot_options": {
         "title": "Plot Options",
         "description": "Options to control plotting.",
         "default": {
            "transform": {
               "__version__": "1.3.0",
```


(continues on next page)

(continued from previous page)


```
                "transform": {
                   "__class_fullname__": "rastervision.pytorch_learner.utils.utils.
↪MinMaxNormalize",
                   "always_apply": false,
                   "p": 1.0,
                   "min_val": 0.0,
                   "max_val": 1.0,
                   "dtype": 5
                }
             },
             "channel_display_groups": null,
             "type_hint": "plot_options"
          },
          "allOf": [
             {
                "$ref": "#/definitions/PlotOptions"
             }
          ]
       },
       "preview_batch_limit": {
          "title": "Preview Batch Limit",
          "description": "Optional limit on the number of items in the preview plots␣
↪produced during training.",
          "type": "integer"
       },
       "type_hint": {
          "title": "Type Hint",
          "default": "semantic_segmentation_image_data",
          "enum": [
             "semantic_segmentation_image_data"
          ],
          "type": "string"
       },
       "data_format": {
          "default": "default",
          "allOf": [
             {
                "$ref": "#/definitions/SemanticSegmentationDataFormat"
             }
          ]
       },
       "uri": {
          "title": "Uri",
          "description": "One of the following:\n(1) a URI of a directory containing␣
↪\"train\", \"valid\", and (optinally) \"test\" subdirectories;\n(2) a URI of a␣
↪zip file containing (1);\n(3) a list of (2);\n(4) a URI of a directory containing␣
↪zip files containing (1).",
          "anyOf": [
             {
                "type": "string"
             },
             {
```


(continues on next page)

(continued from previous page)

```
                "type": "array",
                "items": {
                   "type": "string"
                }
             }
          ]
       },
       "group_uris": {
          "title": "Group Uris",
          "description": "This can be set instead of uri in order to specify groups␣
↪of chips. Each element in the list is expected to be an object of the same form␣
↪accepted by the uri field. The purpose of separating chips into groups is to be␣
↪able to use the group_train_sz field.",
          "type": "array",
          "items": {
             "anyOf": [
                {
                   "type": "string"
                },
                {
                   "type": "array",
                   "items": {
                      "type": "string"
                   }
                }
             ]
          }
       },
       "group_train_sz": {
          "title": "Group Train Sz",
          "description": "If group_uris is set, this can be used to specify the␣
↪number of chips to use per group. Only applies to training chips. This can either␣
↪be a single value that will be used for all groups or a list of values (one for␣
↪each group).",
          "anyOf": [
             {
                "type": "integer"
             },
             {
                "type": "array",
                "items": {
                   "type": "integer"
                }
             }
          ]
       },
       "group_train_sz_rel": {
          "title": "Group Train Sz Rel",
          "description": "Relative version of group_train_sz. Must be a float in [0,␣
↪1]. If group_uris is set, this can be used to specify the proportion of the total␣
↪chips in each group to use per group. Only applies to training chips. This can␣
↪either be a single value that will be used for all groups or a list of values␣
```

(continues on next page)

(continued from previous page)

```
↪(one for each group).",
        "anyOf": [
           {
              "type": "number",
              "minimum": 0,
              "maximum": 1
           },
           {
              "type": "array",
              "items": {
                 "type": "number",
                 "minimum": 0,
                 "maximum": 1
              }
           }
        ]
     }
  },
  "additionalProperties": false,
  "definitions": {
     "PlotOptions": {
        "title": "PlotOptions",
        "description": "Config related to plotting.",
        "type": "object",
        "properties": {
           "transform": {
              "title": "Transform",
              "description": "An Albumentations transform serialized as a dict␣
↪that will be applied to each image before it is plotted. Mainly useful for␣
↪undoing any data transformation that you do not want included in the plot, such␣
↪as normalization. The default value will shift and scale the image so the values␣
↪range from 0.0 to 1.0 which is the expected range for the plotting function. This␣
↪default is useful for cases where the values after normalization are close to␣
↪zero which makes the plot difficult to see.",
              "default": {
                 "__version__": "1.3.0",
                 "transform": {
                    "__class_fullname__": "rastervision.pytorch_learner.utils.
↪utils.MinMaxNormalize",
                    "always_apply": false,
                    "p": 1.0,
                    "min_val": 0.0,
                    "max_val": 1.0,
                    "dtype": 5
                 }
              },
              "type": "object"
           },
           "channel_display_groups": {
              "title": "Channel Display Groups",
              "description": "Groups of image channels to display together as a␣
↪subplot when plotting the data and predictions. Can be a list or tuple of groups␣
```

(continues on next page)

(continued from previous page)

```
↪(e.g. [(0, 1, 2), (3,)]) or a dict containing title-to-group mappings (e.g. {\
↪"RGB\": [0, 1, 2], \"IR\": [3]}), where each group is a list or tuple of channel␣
↪indices and title is a string that will be used as the title of the subplot for␣
↪that group.",
                "anyOf": [
                    {
                        "type": "object",
                        "additionalProperties": {
                            "type": "array",
                            "items": {
                                "type": "integer",
                                "minimum": 0
                            }
                        }
                    },
                    {
                        "type": "array",
                        "items": {
                            "type": "array",
                            "items": {
                                "type": "integer",
                                "minimum": 0
                            }
                        }
                    }
                ]
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "plot_options",
                "enum": [
                    "plot_options"
                ],
                "type": "string"
            }
        },
        "additionalProperties": false
    },
    "SemanticSegmentationDataFormat": {
        "title": "SemanticSegmentationDataFormat",
        "description": "An enumeration.",
        "enum": [
            "default"
        ]
    }
  }
}
```

Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *aug_transform (Optional[dict])*
- *augmentors (List[str])*
- *base_transform (Optional[dict])*
- *class_colors (Optional[List[Union[str, Tuple[int, int, int]]]])*
- *class_names (List[str])*
- *data_format (rastervision.pytorch_learner.semantic_segmentation_learner_config.SemanticSegmentationDataFormat)*
- *group_train_sz (Optional[Union[int, List[int]]])*
- *group_train_sz_rel (Optional[Union[rastervision.pytorch_learner.learner_config.ConstrainedFloatValue, List[rastervision.pytorch_learner.learner_config.ConstrainedFloatValue]]])*
- *group_uris (Optional[List[Union[str, List[str]]]])*
- *img_channels (Optional[pydantic.types.PositiveInt])*
- *img_sz (pydantic.types.PositiveInt)*
- *num_workers (int)*
- *plot_options (Optional[rastervision.pytorch_learner.learner_config.PlotOptions])*
- *preview_batch_limit (Optional[int])*
- *train_sz (Optional[int])*
- *train_sz_rel (Optional[float])*
- *type_hint (Literal['semantic_segmentation_image_data'])*
- *uri (Optional[Union[str, List[str]]])*

Validators

- ensure_class_colors » all fields
- *validate_albumentation_transform* » *aug_transform*
- *validate_albumentation_transform* » *base_transform*
- validate_augmentors » *augmentors*
- validate_group_uris » all fields
- validate_plot_options » all fields

field aug_transform: Optional[dict] = None

An Albumentations transform serialized as a dict that will be applied as data augmentation to the training dataset. This transform is applied before base_transform. If provided, the augmentors option is ignored.

Validated by

- ensure_class_colors
- *validate_albumentation_transform*
- validate_group_uris
- validate_plot_options

field augmentors: List[str] = ['RandomRotate90', 'HorizontalFlip', 'VerticalFlip']

Names of albumentations augmentors to use for training batches. Choices include: ['Blur', 'RandomRotate90', 'HorizontalFlip', 'VerticalFlip', 'GaussianBlur', 'GaussNoise', 'RGBShift', 'ToGray']. Alternatively, a custom transform can be provided via the aug_transform option.

Validated by

- ensure_class_colors
- validate_augmentors
- validate_group_uris
- validate_plot_options

field base_transform: Optional[dict] = None

An Albumentations transform serialized as a dict that will be applied to all datasets: training, validation, and test. This transformation is in addition to the resizing due to img_sz. This is useful for, for example, applying the same normalization to all datasets.

Validated by

- ensure_class_colors
- *validate_albumentation_transform*
- validate_group_uris
- validate_plot_options

field class_colors: Optional[List[Union[str, RGBTuple]]] = None

Colors used to display classes. Can be color 3-tuples in list form.

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field class_names: List[str] = []

Names of classes.

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field data_format: *SemanticSegmentationDataFormat* = SemanticSegmentationDataFormat.default

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field group_train_sz: Optional[Union[int, List[int]]] = None

If group_uris is set, this can be used to specify the number of chips to use per group. Only applies to training chips. This can either be a single value that will be used for all groups or a list of values (one for each group).

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field group_train_sz_rel: Optional[Union[Proportion, List[Proportion]]] = None

Relative version of group_train_sz. Must be a float in [0, 1]. If group_uris is set, this can be used to specify the proportion of the total chips in each group to use per group. Only applies to training chips. This can either be a single value that will be used for all groups or a list of values (one for each group).

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field group_uris: Optional[List[Union[str, List[str]]]] = None

This can be set instead of uri in order to specify groups of chips. Each element in the list is expected to be an object of the same form accepted by the uri field. The purpose of separating chips into groups is to be able to use the group_train_sz field.

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field img_channels: Optional[PosInt] = None

The number of channels of the training images.

Constraints

- **exclusiveMinimum** = 0

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field img_sz: PosInt = 256

Length of a side of each image in pixels. This is the size to transform it to during training, not the size in the raw dataset.

Constraints

- **exclusiveMinimum** = 0

Validated by

- ensure_class_colors

- validate_group_uris
- validate_plot_options

field num_workers: int = 4

Number of workers to use when DataLoader makes batches.

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field plot_options: Optional[*PlotOptions*] = PlotOptions(transform={'__version__': '1.3.0', 'transform': {'__class_fullname__': 'rastervision.pytorch_learner.utils.utils.MinMaxNormalize', 'always_apply': False, 'p': 1.0, 'min_val': 0.0, 'max_val': 1.0, 'dtype': 5}}, channel_display_groups=None)

Options to control plotting.

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field preview_batch_limit: Optional[int] = None

Optional limit on the number of items in the preview plots produced during training.

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field train_sz: Optional[int] = None

If set, the number of training images to use. If fewer images exist, then an exception will be raised.

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field train_sz_rel: Optional[float] = None

If set, the proportion of training images to use.

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field type_hint: Literal['semantic_segmentation_image_data'] = 'semantic_segmentation_image_data'

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

field uri: Optional[Union[str, List[str]]] = None

One of the following: (1) a URI of a directory containing "train", "valid", and (optinally) "test" subdirectories; (2) a URI of a zip file containing (1); (3) a list of (2); (4) a URI of a directory containing zip files containing (1).

Validated by

- ensure_class_colors
- validate_group_uris
- validate_plot_options

build(*tmp_dir: str*, *overfit_mode: bool = False*, *test_mode: bool = False*) → Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

Parameters

- **tmp_dir** (*str*) –
- **overfit_mode** (*bool*) –
- **test_mode** (*bool*) –

Return type
Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

dir_to_dataset(*data_dir: str*, *transform: BasicTransform*) → torch.utils.data.Dataset

Parameters

- **data_dir** (*str*) –
- **transform** (*BasicTransform*) –

Return type
torch.utils.data.Dataset

validator ensure_class_colors » *all fields*

Parameters
values (*dict*) –

Return type
dict

get_bbox_params() → Optional[BboxParams]

Returns BboxParams used by albumentations for data augmentation.

Return type
Optional[*BboxParams*]

get_custom_albumentations_transforms() → List[dict]

Returns all custom transforms found in this config.

This should return all serialized albumentations transforms with a 'lambda_transforms_path' field contained in this config or in any of its members no matter how deeply neseted.

The pupose is to make it easier to adjust their paths all at once while saving to or loading from a bundle.

Return type
List[dict]

get_data_dirs(*uri: Union[str, List[str]]*, *unzip_dir: str*) → List[str]

Extract data dirs from uri.

Data dirs are directories containing "train", "valid", and (optinally) "test" subdirectories.

Parameters

- **uri** (*Union[str, List[str]]*) – a URI or a list of URIs of one of the following:

(1) a URI of a directory containing "train", "valid", and (optinally) "test" subdirectories

(2) a URI of a zip file containing (1)

(3) a list of (2)

(4) a URI of a directory containing zip files containing (1)

- **unzip_dir** (*str*) –

Returns
paths to directories that each contain contents of one zip file

Return type
List[str]

get_data_transforms() → Tuple[BasicTransform, BasicTransform]

Get albumentations transform objects for data augmentation.

Returns
a transform that doesn't do any data augmentation 2nd tuple arg: a transform with data augmentation

Return type
1st tuple arg

get_datasets_from_group_uris(*uris: Union[str, List[str]]*, *tmp_dir: str*, *group_train_sz: Optional[int] = None*, *group_train_sz_rel: Optional[float] = None*, *overfit_mode: bool = False*, *test_mode: bool = False*) → Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

Parameters

- **uris** (*Union[str, List[str]]*) –
- **tmp_dir** (*str*) –
- **group_train_sz** (*Optional[int]*) –
- **group_train_sz_rel** (*Optional[float]*) –
- **overfit_mode** (*bool*) –
- **test_mode** (*bool*) –

Return type
Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

get_datasets_from_uri(*uri: Union[str, List[str]]*, *tmp_dir: str*, *overfit_mode: bool = False*, *test_mode: bool = False*) → Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

Get image train, validation, & test datasets from a single zip file.

Parameters

- **uri** (*Union[str, List[str]]*) – Uri of a zip file containing the images.
- **tmp_dir** (*str*) –
- **overfit_mode** (*bool*) –
- **test_mode** (*bool*) –

Returns

Training, validation, and test
dataSets.

Return type
Tuple[Dataset, Dataset, Dataset]

make_datasets(*train_dirs: Iterable[str]*, *val_dirs: Iterable[str]*, *test_dirs: Iterable[str]*, *train_tf: Optional[BasicTransform] = None*, *val_tf: Optional[BasicTransform] = None*, *test_tf: Optional[BasicTransform] = None*) → Tuple[torch.utils.data.Dataset, torch.utils.data.Dataset, torch.utils.data.Dataset]

Make training, validation, and test datasets.

Parameters

- **train_dirs** (*str*) – Directories where training data is located.
- **val_dirs** (*str*) – Directories where validation data is located.
- **test_dirs** (*str*) – Directories where test data is located.
- **train_tf** (*Optional[A.BasicTransform], optional*) – Transform for the training dataset. Defaults to None.
- **val_tf** (*Optional[A.BasicTransform], optional*) – Transform for the validation dataset. Defaults to None.
- **test_tf** (*Optional[A.BasicTransform], optional*) – Transform for the test dataset. Defaults to None.

Returns

PyTorch-compatiable training,
validation, and test datasets.

Return type
Tuple[Dataset, Dataset, Dataset]

random_subset_dataset(*ds: torch.utils.data.Dataset*, *size: Optional[int] = None*, *fraction: Optional[ConstrainedFloatValue] = None*) → torch.utils.data.Subset

Parameters

- **ds** (*torch.utils.data.Dataset*) –
- **size** (*Optional[int]*) –

- **fraction** (*Optional[ConstrainedFloatValue]*) –

Return type
torch.utils.data.Subset

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

unzip_data(*zip_uris: List[str]*, *unzip_dir: str*) → List[str]

Unzip dataset zip files.

Parameters

- **zip_uris** (*List[str]*) – a list of URIs of zip files:
- **unzip_dir** (*str*) – directory where zip files will be extrated to.

Returns
paths to directories that each contain contents of one zip file

Return type
List[str]

update(**args*, ***kwargs*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validator validate_augmentors » augmentors

Parameters
v (*str*) –

Return type
str

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validator validate_group_uris » *all fields*

Parameters
values (*dict*) –

Return type
dict

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises
ConfigError – if field is invalid

validator validate_plot_options » *all fields*

Parameters
values (*dict*) –

Return type
dict

property num_classes

SemanticSegmentationLearnerConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model SemanticSegmentationLearnerConfig

Configure a *SemanticSegmentationLearner*.


```
{
   "title": "SemanticSegmentationLearnerConfig",
   "description": "Configure a :class:`.SemanticSegmentationLearner`.",
   "type": "object",
   "properties": {
      "model": {
         "$ref": "#/definitions/SemanticSegmentationModelConfig"
      },
      "solver": {
         "$ref": "#/definitions/SolverConfig"
      },
      "data": {
         "title": "Data",
         "anyOf": [
            {
               "$ref": "#/definitions/SemanticSegmentationImageDataConfig"
            },
            {
               "$ref": "#/definitions/SemanticSegmentationGeoDataConfig"
            }
         ]
      },
      "predict_mode": {
         "title": "Predict Mode",
         "description": "If True, skips training, loads model, and does final eval.
↪",
         "default": false,
         "type": "boolean"
      },
```


(continues on next page)

(continued from previous page)

```
      "test_mode": {
         "title": "Test Mode",
         "description": "If True, uses test_num_epochs, test_batch_sz, truncated␣
↪datasets with only a single batch, image_sz that is cut in half, and num_workers␣
↪= 0. This is useful for testing that code runs correctly on CPU without␣
↪multithreading before running full job on GPU.",
         "default": false,
         "type": "boolean"
      },
      "overfit_mode": {
         "title": "Overfit Mode",
         "description": "If True, uses half image size, and instead of doing epoch-
↪based training, optimizes the model using a single batch repeatedly for overfit_
↪num_steps number of steps.",
         "default": false,
         "type": "boolean"
      },
      "eval_train": {
         "title": "Eval Train",
         "description": "If True, runs final evaluation on training set (in␣
↪addition to test set). Useful for debugging.",
         "default": false,
         "type": "boolean"
      },
      "save_model_bundle": {
         "title": "Save Model Bundle",
         "description": "If True, saves a model bundle at the end of training which␣
↪is zip file with model and this LearnerConfig which can be used to make␣
↪predictions on new images at a later time.",
         "default": true,
         "type": "boolean"
      },
      "log_tensorboard": {
         "title": "Log Tensorboard",
         "description": "Save Tensorboard log files at the end of each epoch.",
         "default": true,
         "type": "boolean"
      },
      "run_tensorboard": {
         "title": "Run Tensorboard",
         "description": "run Tensorboard server during training",
         "default": false,
         "type": "boolean"
      },
      "output_uri": {
         "title": "Output Uri",
         "description": "URI of where to save output",
         "type": "string"
      },
      "type_hint": {
         "title": "Type Hint",
         "default": "semantic_segmentation_learner",
```

(continues on next page)

(continued from previous page)

```
        "enum": [
            "semantic_segmentation_learner"
        ],
        "type": "string"
    }
},
"required": [
    "solver",
    "data"
],
"additionalProperties": false,
"definitions": {
    "Backbone": {
        "title": "Backbone",
        "description": "An enumeration.",
        "enum": [
            "alexnet",
            "densenet121",
            "densenet169",
            "densenet201",
            "densenet161",
            "googlenet",
            "inception_v3",
            "mnasnet0_5",
            "mnasnet0_75",
            "mnasnet1_0",
            "mnasnet1_3",
            "mobilenet_v2",
            "resnet18",
            "resnet34",
            "resnet50",
            "resnet101",
            "resnet152",
            "resnext50_32x4d",
            "resnext101_32x8d",
            "wide_resnet50_2",
            "wide_resnet101_2",
            "shufflenet_v2_x0_5",
            "shufflenet_v2_x1_0",
            "shufflenet_v2_x1_5",
            "shufflenet_v2_x2_0",
            "squeezenet1_0",
            "squeezenet1_1",
            "vgg11",
            "vgg11_bn",
            "vgg13",
            "vgg13_bn",
            "vgg16",
            "vgg16_bn",
            "vgg19_bn",
            "vgg19"
        ]
```

(continues on next page)

(continued from previous page)

```
    },
    "ExternalModuleConfig": {
       "title": "ExternalModuleConfig",
       "description": "Config describing an object to be loaded via Torch Hub.",
       "type": "object",
       "properties": {
          "uri": {
             "title": "Uri",
             "description": "Local uri of a zip file, or local uri of a directory,
↪or remote uri of zip file.",
             "minLength": 1,
             "type": "string"
          },
          "github_repo": {
             "title": "Github Repo",
             "description": "<repo-owner>/<repo-name>[:tag]",
             "pattern": ".+/.+",
             "type": "string"
          },
          "name": {
             "title": "Name",
             "description": "Name of the folder in which to extract/copy the␣
↪definition files.",
             "minLength": 1,
             "type": "string"
          },
          "entrypoint": {
             "title": "Entrypoint",
             "description": "Name of a callable present in hubconf.py. See docs␣
↪for torch.hub for details.",
             "minLength": 1,
             "type": "string"
          },
          "entrypoint_args": {
             "title": "Entrypoint Args",
             "description": "Args to pass to the entrypoint. Must be serializable.
↪",
             "default": [],
             "type": "array",
             "items": {}
          },
          "entrypoint_kwargs": {
             "title": "Entrypoint Kwargs",
             "description": "Keyword args to pass to the entrypoint. Must be␣
↪serializable.",
             "default": {},
             "type": "object"
          },
          "force_reload": {
             "title": "Force Reload",
             "description": "Force reload of module definition.",
             "default": false,
```

(continues on next page)

(continued from previous page)

```
                "type": "boolean"
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "external-module",
                "enum": [
                    "external-module"
                ],
                "type": "string"
            }
        },
        "required": [
            "entrypoint"
        ],
        "additionalProperties": false
    },
    "SemanticSegmentationModelConfig": {
        "title": "SemanticSegmentationModelConfig",
        "description": "Configure a semantic segmentation model.",
        "type": "object",
        "properties": {
            "backbone": {
                "description": "The torchvision.models backbone to use. At the␣
↪moment only resnet50 or resnet101 will work.",
                "default": "resnet50",
                "allOf": [
                    {
                        "$ref": "#/definitions/Backbone"
                    }
                ]
            },
            "pretrained": {
                "title": "Pretrained",
                "description": "If True, use ImageNet weights. If False, use random␣
↪initialization.",
                "default": true,
                "type": "boolean"
            },
            "init_weights": {
                "title": "Init Weights",
                "description": "URI of PyTorch model weights used to initialize␣
↪model. If set, this supercedes the pretrained option.",
                "type": "string"
            },
            "load_strict": {
                "title": "Load Strict",
                "description": "If True, the keys in the state dict referenced by␣
↪init_weights must match exactly. Setting this to False can be useful if you just␣
↪want to load the backbone of a model.",
                "default": true,
                "type": "boolean"
            },
```

(continues on next page)

(continued from previous page)


```
            "external_def": {
               "title": "External Def",
               "description": "If specified, the model will be built from the␣
↪definition from this external source, using Torch Hub.",
               "allOf": [
                  {
                     "$ref": "#/definitions/ExternalModuleConfig"
                  }
               ]
            },
            "type_hint": {
               "title": "Type Hint",
               "default": "semantic_segmentation_model",
               "enum": [
                  "semantic_segmentation_model"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      },
      "SolverConfig": {
         "title": "SolverConfig",
         "description": "Config related to solver aka optimizer.",
         "type": "object",
         "properties": {
            "lr": {
               "title": "Lr",
               "description": "Learning rate.",
               "default": 0.0001,
               "exclusiveMinimum": 0,
               "type": "number"
            },
            "num_epochs": {
               "title": "Num Epochs",
               "description": "Number of epochs (ie. sweeps through the whole␣
↪training set).",
               "default": 10,
               "exclusiveMinimum": 0,
               "type": "integer"
            },
            "test_num_epochs": {
               "title": "Test Num Epochs",
               "description": "Number of epochs to use in test mode.",
               "default": 2,
               "exclusiveMinimum": 0,
               "type": "integer"
            },
            "test_batch_sz": {
               "title": "Test Batch Sz",
               "description": "Batch size to use in test mode.",
               "default": 4,
```


(continues on next page)

(continued from previous page)

```
                "exclusiveMinimum": 0,
                "type": "integer"
            },
            "overfit_num_steps": {
                "title": "Overfit Num Steps",
                "description": "Number of optimizer steps to use in overfit mode.",
                "default": 1,
                "exclusiveMinimum": 0,
                "type": "integer"
            },
            "sync_interval": {
                "title": "Sync Interval",
                "description": "The interval in epochs for each sync to the cloud.",
                "default": 1,
                "exclusiveMinimum": 0,
                "type": "integer"
            },
            "batch_sz": {
                "title": "Batch Sz",
                "description": "Batch size.",
                "default": 32,
                "exclusiveMinimum": 0,
                "type": "integer"
            },
            "one_cycle": {
                "title": "One Cycle",
                "description": "If True, use triangular LR scheduler with a single␣
↪cycle across all epochs with start and end LR being lr/10 and the peak being lr.",
                "default": true,
                "type": "boolean"
            },
            "multi_stage": {
                "title": "Multi Stage",
                "description": "List of epoch indices at which to divide LR by 10.",
                "default": [],
                "type": "array",
                "items": {}
            },
            "class_loss_weights": {
                "title": "Class Loss Weights",
                "description": "Class weights for weighted loss.",
                "type": "array",
                "items": {
                    "type": "number"
                }
            },
            "ignore_class_index": {
                "title": "Ignore Class Index",
                "description": "If specified, this index is ignored when computing␣
↪the loss. See pytorch documentation for nn.CrossEntropyLoss for more details.␣
↪This can also be negative, in which case it is treated as a negative slice index␣
↪i.e. -1 = last index, -2 = second-last index, and so on.",
```

(continues on next page)

(continued from previous page)


```
                "type": "integer"
            },
            "external_loss_def": {
                "title": "External Loss Def",
                "description": "If specified, the loss will be built from the␣
↪definition from this external source, using Torch Hub.",
                "allOf": [
                    {
                        "$ref": "#/definitions/ExternalModuleConfig"
                    }
                ]
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "solver",
                "enum": [
                    "solver"
                ],
                "type": "string"
            }
        },
        "additionalProperties": false
    },
    "PlotOptions": {
        "title": "PlotOptions",
        "description": "Config related to plotting.",
        "type": "object",
        "properties": {
            "transform": {
                "title": "Transform",
                "description": "An Albumentations transform serialized as a dict␣
↪that will be applied to each image before it is plotted. Mainly useful for␣
↪undoing any data transformation that you do not want included in the plot, such␣
↪as normalization. The default value will shift and scale the image so the values␣
↪range from 0.0 to 1.0 which is the expected range for the plotting function. This␣
↪default is useful for cases where the values after normalization are close to␣
↪zero which makes the plot difficult to see.",
                "default": {
                    "__version__": "1.3.0",
                    "transform": {
                        "__class_fullname__": "rastervision.pytorch_learner.utils.
↪utils.MinMaxNormalize",
                        "always_apply": false,
                        "p": 1.0,
                        "min_val": 0.0,
                        "max_val": 1.0,
                        "dtype": 5
                    }
                },
                "type": "object"
            },
            "channel_display_groups": {
```


(continues on next page)

(continued from previous page)

```
                "title": "Channel Display Groups",
                "description": "Groups of image channels to display together as a␣
↪subplot when plotting the data and predictions. Can be a list or tuple of groups␣
↪(e.g. [(0, 1, 2), (3,)]) or a dict containing title-to-group mappings (e.g. {\
↪"RGB\": [0, 1, 2], \"IR\": [3]}), where each group is a list or tuple of channel␣
↪indices and title is a string that will be used as the title of the subplot for␣
↪that group.",
                "anyOf": [
                   {
                      "type": "object",
                      "additionalProperties": {
                         "type": "array",
                         "items": {
                            "type": "integer",
                            "minimum": 0
                         }
                      }
                   },
                   {
                      "type": "array",
                      "items": {
                         "type": "array",
                         "items": {
                            "type": "integer",
                            "minimum": 0
                         }
                      }
                   }
                ]
             },
             "type_hint": {
                "title": "Type Hint",
                "default": "plot_options",
                "enum": [
                   "plot_options"
                ],
                "type": "string"
             }
          },
          "additionalProperties": false
       },
       "SemanticSegmentationDataFormat": {
          "title": "SemanticSegmentationDataFormat",
          "description": "An enumeration.",
          "enum": [
             "default"
          ]
       },
       "SemanticSegmentationImageDataConfig": {
          "title": "SemanticSegmentationImageDataConfig",
          "description": "Configure :class:`SemanticSegmentationImageDatasets <.
↪SemanticSegmentationImageDataset>`.\n\nThis assumes the following file structure:\
```

(continues on next page)

(continued from previous page)

```
↪n\n.. code-block:: text\n\n    <data_dir>/\n        img/\n            <img 1>.
↪<extension>\n            <img 2>.<extension>\n            ...\n            <img N>
↪.<extension>\n        labels/\n            <img 1>.<extension>\n            <img␣
↪2>.<extension>\n            ...\n            <img N>.<extension>",
        "type": "object",
        "properties": {
           "class_names": {
              "title": "Class Names",
              "description": "Names of classes.",
              "default": [],
              "type": "array",
              "items": {
                 "type": "string"
              }
           },
           "class_colors": {
              "title": "Class Colors",
              "description": "Colors used to display classes. Can be color 3-
↪tuples in list form.",
              "type": "array",
              "items": {
                 "anyOf": [
                    {
                       "type": "string"
                    },
                    {
                       "type": "array",
                       "minItems": 3,
                       "maxItems": 3,
                       "items": [
                          {
                             "type": "integer"
                          },
                          {
                             "type": "integer"
                          },
                          {
                             "type": "integer"
                          }
                       ]
                    }
                 ]
              }
           },
           "img_channels": {
              "title": "Img Channels",
              "description": "The number of channels of the training images.",
              "exclusiveMinimum": 0,
              "type": "integer"
           },
           "img_sz": {
              "title": "Img Sz",
```

(continues on next page)

(continued from previous page)

```
                "description": "Length of a side of each image in pixels. This is␣
↪the size to transform it to during training, not the size in the raw dataset.",
                "default": 256,
                "exclusiveMinimum": 0,
                "type": "integer"
            },
            "train_sz": {
                "title": "Train Sz",
                "description": "If set, the number of training images to use. If␣
↪fewer images exist, then an exception will be raised.",
                "type": "integer"
            },
            "train_sz_rel": {
                "title": "Train Sz Rel",
                "description": "If set, the proportion of training images to use.",
                "type": "number"
            },
            "num_workers": {
                "title": "Num Workers",
                "description": "Number of workers to use when DataLoader makes␣
↪batches.",
                "default": 4,
                "type": "integer"
            },
            "augmentors": {
                "title": "Augmentors",
                "description": "Names of albumentations augmentors to use for␣
↪training batches. Choices include: ['Blur', 'RandomRotate90', 'HorizontalFlip',
↪'VerticalFlip', 'GaussianBlur', 'GaussNoise', 'RGBShift', 'ToGray'].␣
↪Alternatively, a custom transform can be provided via the aug_transform option.",
                "default": [
                    "RandomRotate90",
                    "HorizontalFlip",
                    "VerticalFlip"
                ],
                "type": "array",
                "items": {
                    "type": "string"
                }
            },
            "base_transform": {
                "title": "Base Transform",
                "description": "An Albumentations transform serialized as a dict␣
↪that will be applied to all datasets: training, validation, and test. This␣
↪transformation is in addition to the resizing due to img_sz. This is useful for,␣
↪for example, applying the same normalization to all datasets.",
                "type": "object"
            },
            "aug_transform": {
                "title": "Aug Transform",
                "description": "An Albumentations transform serialized as a dict␣
↪that will be applied as data augmentation to the training dataset. This transform␣
```

(continues on next page)

(continued from previous page)


```
↪is applied before base_transform. If provided, the augmentors option is ignored.",
                "type": "object"
            },
            "plot_options": {
                "title": "Plot Options",
                "description": "Options to control plotting.",
                "default": {
                    "transform": {
                        "__version__": "1.3.0",
                        "transform": {
                            "__class_fullname__": "rastervision.pytorch_learner.utils.
↪utils.MinMaxNormalize",
                            "always_apply": false,
                            "p": 1.0,
                            "min_val": 0.0,
                            "max_val": 1.0,
                            "dtype": 5
                        }
                    },
                    "channel_display_groups": null,
                    "type_hint": "plot_options"
                },
                "allOf": [
                    {
                        "$ref": "#/definitions/PlotOptions"
                    }
                ]
            },
            "preview_batch_limit": {
                "title": "Preview Batch Limit",
                "description": "Optional limit on the number of items in the preview␣
↪plots produced during training.",
                "type": "integer"
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "semantic_segmentation_image_data",
                "enum": [
                    "semantic_segmentation_image_data"
                ],
                "type": "string"
            },
            "data_format": {
                "default": "default",
                "allOf": [
                    {
                        "$ref": "#/definitions/SemanticSegmentationDataFormat"
                    }
                ]
            },
            "uri": {
                "title": "Uri",
```


(continues on next page)

(continued from previous page)

```
                "description": "One of the following:\n(1) a URI of a directory␣
↪containing \"train\", \"valid\", and (optinally) \"test\" subdirectories;\n(2) a␣
↪URI of a zip file containing (1);\n(3) a list of (2);\n(4) a URI of a directory␣
↪containing zip files containing (1).",
                "anyOf": [
                   {
                      "type": "string"
                   },
                   {
                      "type": "array",
                      "items": {
                         "type": "string"
                      }
                   }
                ]
             },
             "group_uris": {
                "title": "Group Uris",
                "description": "This can be set instead of uri in order to specify␣
↪groups of chips. Each element in the list is expected to be an object of the same␣
↪form accepted by the uri field. The purpose of separating chips into groups is to␣
↪be able to use the group_train_sz field.",
                "type": "array",
                "items": {
                   "anyOf": [
                      {
                         "type": "string"
                      },
                      {
                         "type": "array",
                         "items": {
                            "type": "string"
                         }
                      }
                   ]
                }
             },
             "group_train_sz": {
                "title": "Group Train Sz",
                "description": "If group_uris is set, this can be used to specify␣
↪the number of chips to use per group. Only applies to training chips. This can␣
↪either be a single value that will be used for all groups or a list of values␣
↪(one for each group).",
                "anyOf": [
                   {
                      "type": "integer"
                   },
                   {
                      "type": "array",
                      "items": {
                         "type": "integer"
                      }
```

(continues on next page)

(continued from previous page)

```
                }
             ]
          },
          "group_train_sz_rel": {
             "title": "Group Train Sz Rel",
             "description": "Relative version of group_train_sz. Must be a float␣
↪in [0, 1]. If group_uris is set, this can be used to specify the proportion of␣
↪the total chips in each group to use per group. Only applies to training chips.␣
↪This can either be a single value that will be used for all groups or a list of␣
↪values (one for each group).",
             "anyOf": [
                {
                   "type": "number",
                   "minimum": 0,
                   "maximum": 1
                },
                {
                   "type": "array",
                   "items": {
                      "type": "number",
                      "minimum": 0,
                      "maximum": 1
                   }
                }
             ]
          }
       },
       "additionalProperties": false
    },
    "ClassConfig": {
       "title": "ClassConfig",
       "description": "Configure class information for a machine learning task.",
       "type": "object",
       "properties": {
          "names": {
             "title": "Names",
             "description": "Names of classes. The i-th class in this list will␣
↪have class ID = i.",
             "type": "array",
             "items": {
                "type": "string"
             }
          },
          "colors": {
             "title": "Colors",
             "description": "Colors used to visualize classes. Can be color␣
↪strings accepted by matplotlib or RGB tuples. If None, a random color will be␣
↪auto-generated for each class.",
             "type": "array",
             "items": {
                "anyOf": [
                   {
```

(continues on next page)

(continued from previous page)


```
                        "type": "string"
                     },
                     {
                        "type": "array",
                        "items": {}
                     }
                  ]
               }
            },
            "null_class": {
               "title": "Null Class",
               "description": "Optional name of class in `names` to use as the null␣
↪class. This is used in semantic segmentation to represent the label for imagery␣
↪pixels that are NODATA or that are missing a label. If None and the class names␣
↪include \"null\", it will automatically be used as the null class. If None, and␣
↪this Config is part of a SemanticSegmentationConfig, a null class will be added␣
↪automatically.",
               "type": "string"
            },
            "type_hint": {
               "title": "Type Hint",
               "default": "class_config",
               "enum": [
                  "class_config"
               ],
               "type": "string"
            }
         },
         "required": [
            "names"
         ],
         "additionalProperties": false
      },
      "RasterTransformerConfig": {
         "title": "RasterTransformerConfig",
         "description": "Configure a :class:`.RasterTransformer`.",
         "type": "object",
         "properties": {
            "type_hint": {
               "title": "Type Hint",
               "default": "raster_transformer",
               "enum": [
                  "raster_transformer"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      },
      "RasterSourceConfig": {
         "title": "RasterSourceConfig",
         "description": "Configure a :class:`.RasterSource`.",
```


(continues on next page)

(continued from previous page)

```
        "type": "object",
        "properties": {
           "channel_order": {
              "title": "Channel Order",
              "description": "The sequence of channel indices to use when reading␣
↪imagery.",
              "type": "array",
              "items": {
                 "type": "integer"
              }
           },
           "transformers": {
              "title": "Transformers",
              "default": [],
              "type": "array",
              "items": {
                 "$ref": "#/definitions/RasterTransformerConfig"
              }
           },
           "extent": {
              "title": "Extent",
              "description": "Use-specified extent in pixel coords in the form␣
↪(ymin, xmin, ymax, xmax). Useful for cropping the raster source so that only part␣
↪of the raster is read from.",
              "type": "array",
              "minItems": 4,
              "maxItems": 4,
              "items": [
                 {
                    "type": "integer"
                 },
                 {
                    "type": "integer"
                 },
                 {
                    "type": "integer"
                 },
                 {
                    "type": "integer"
                 }
              ]
           },
           "type_hint": {
              "title": "Type Hint",
              "default": "raster_source",
              "enum": [
                 "raster_source"
              ],
              "type": "string"
           }
        },
        "additionalProperties": false
```

(continues on next page)

(continued from previous page)


```
      },
      "LabelSourceConfig": {
         "title": "LabelSourceConfig",
         "description": "Configure a :class:`.LabelSource`.",
         "type": "object",
         "properties": {
            "type_hint": {
               "title": "Type Hint",
               "default": "label_source",
               "enum": [
                  "label_source"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      },
      "LabelStoreConfig": {
         "title": "LabelStoreConfig",
         "description": "Configure a :class:`.LabelStore`.",
         "type": "object",
         "properties": {
            "type_hint": {
               "title": "Type Hint",
               "default": "label_store",
               "enum": [
                  "label_store"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      },
      "SceneConfig": {
         "title": "SceneConfig",
         "description": "Configure a :class:`.Scene` comprising raster data &␣
↪labels for an AOI.\n    ",
         "type": "object",
         "properties": {
            "id": {
               "title": "Id",
               "type": "string"
            },
            "raster_source": {
               "$ref": "#/definitions/RasterSourceConfig"
            },
            "label_source": {
               "$ref": "#/definitions/LabelSourceConfig"
            },
            "label_store": {
               "$ref": "#/definitions/LabelStoreConfig"
            },
```


(continues on next page)

(continued from previous page)


```
            "aoi_uris": {
               "title": "Aoi Uris",
               "description": "List of URIs of GeoJSON files that define the AOIs␣
↪for the scene. Each polygon defines an AOI which is a piece of the scene that is␣
↪assumed to be fully labeled and usable for training or validation. The AOIs are␣
↪assumed to be in EPSG:4326 coordinates.",
               "type": "array",
               "items": {
                  "type": "string"
               }
            },
            "type_hint": {
               "title": "Type Hint",
               "default": "scene",
               "enum": [
                  "scene"
               ],
               "type": "string"
            }
         },
         "required": [
            "id",
            "raster_source"
         ],
         "additionalProperties": false
      },
      "DatasetConfig": {
         "title": "DatasetConfig",
         "description": "Configure train, validation, and test splits for a dataset.
↪",
         "type": "object",
         "properties": {
            "class_config": {
               "$ref": "#/definitions/ClassConfig"
            },
            "train_scenes": {
               "title": "Train Scenes",
               "type": "array",
               "items": {
                  "$ref": "#/definitions/SceneConfig"
               }
            },
            "validation_scenes": {
               "title": "Validation Scenes",
               "type": "array",
               "items": {
                  "$ref": "#/definitions/SceneConfig"
               }
            },
            "test_scenes": {
               "title": "Test Scenes",
               "default": [],
```


(continues on next page)

(continued from previous page)


```
                "type": "array",
                "items": {
                   "$ref": "#/definitions/SceneConfig"
                }
             },
             "scene_groups": {
                "title": "Scene Groups",
                "description": "Groupings of scenes. Should be a dict of the form: {
↪<group-name>: Set(scene_id_1, scene_id_2, ...)}. Three groups are added by␣
↪default: \"train_scenes\", \"validation_scenes\", and \"test_scenes\"",
                "default": {},
                "type": "object",
                "additionalProperties": {
                   "type": "array",
                   "items": {
                      "type": "string"
                   },
                   "uniqueItems": true
                }
             },
             "type_hint": {
                "title": "Type Hint",
                "default": "dataset",
                "enum": [
                   "dataset"
                ],
                "type": "string"
             }
          },
          "required": [
             "class_config",
             "train_scenes",
             "validation_scenes"
          ],
          "additionalProperties": false
       },
       "GeoDataWindowMethod": {
          "title": "GeoDataWindowMethod",
          "description": "An enumeration.",
          "enum": [
             "sliding",
             "random"
          ]
       },
       "GeoDataWindowConfig": {
          "title": "GeoDataWindowConfig",
          "description": "Configure a :class:`.GeoDataset`.\n\nSee :mod:
↪`rastervision.pytorch_learner.dataset.dataset`.",
          "type": "object",
          "properties": {
             "method": {
                "default": "sliding",
```


(continues on next page)

(continued from previous page)

```
                "allOf": [
                    {
                        "$ref": "#/definitions/GeoDataWindowMethod"
                    }
                ]
            },
            "size": {
                "title": "Size",
                "description": "If method = sliding, this is the size of sliding␣
↪window. If method = random, this is the size that all the windows are resized to␣
↪before they are returned. If method = random and neither size_lims nor h_lims and␣
↪w_lims have been specified, then size_lims is set to (size, size + 1).",
                "anyOf": [
                    {
                        "type": "integer",
                        "exclusiveMinimum": 0
                    },
                    {
                        "type": "array",
                        "minItems": 2,
                        "maxItems": 2,
                        "items": [
                            {
                                "type": "integer",
                                "exclusiveMinimum": 0
                            },
                            {
                                "type": "integer",
                                "exclusiveMinimum": 0
                            }
                        ]
                    }
                ]
            },
            "stride": {
                "title": "Stride",
                "description": "Stride of sliding window. Only used if method =␣
↪sliding.",
                "anyOf": [
                    {
                        "type": "integer",
                        "exclusiveMinimum": 0
                    },
                    {
                        "type": "array",
                        "minItems": 2,
                        "maxItems": 2,
                        "items": [
                            {
                                "type": "integer",
                                "exclusiveMinimum": 0
                            },
```

(continues on next page)

(continued from previous page)

```
                        {
                            "type": "integer",
                            "exclusiveMinimum": 0
                        }
                    ]
                }
            ]
        },
        "padding": {
            "title": "Padding",
            "description": "How many pixels are windows allowed to overflow the␣
↪edges of the raster source.",
            "anyOf": [
                {
                    "type": "integer",
                    "minimum": 0
                },
                {
                    "type": "array",
                    "minItems": 2,
                    "maxItems": 2,
                    "items": [
                        {
                            "type": "integer",
                            "minimum": 0
                        },
                        {
                            "type": "integer",
                            "minimum": 0
                        }
                    ]
                }
            ]
        },
        "pad_direction": {
            "title": "Pad Direction",
            "description": "If \"end\", only pad ymax and xmax (bottom and␣
↪right). If \"start\", only pad ymin and xmin (top and left). If \"both\", pad all␣
↪sides. Has no effect if paddiong is zero. Defaults to \"end\".",
            "default": "end",
            "enum": [
                "both",
                "start",
                "end"
            ],
            "type": "string"
        },
        "size_lims": {
            "title": "Size Lims",
            "description": "[min, max) interval from which window sizes will be␣
↪uniformly randomly sampled. The upper limit is exclusive. To fix the size to a␣
↪constant value, use size_lims = (sz, sz + 1). Only used if method = random.␣
```

(continues on next page)

(continued from previous page)

```
↪Specify either size_lims or h_lims and w_lims, but not both. If neither size_lims␣
↪nor h_lims and w_lims have been specified, then this will be set to (size, size +␣
↪1).",
                "type": "array",
                "minItems": 2,
                "maxItems": 2,
                "items": [
                   {
                      "type": "integer",
                      "exclusiveMinimum": 0
                   },
                   {
                      "type": "integer",
                      "exclusiveMinimum": 0
                   }
                ]
             },
             "h_lims": {
                "title": "H Lims",
                "description": "[min, max] interval from which window heights will␣
↪be uniformly randomly sampled. Only used if method = random.",
                "type": "array",
                "minItems": 2,
                "maxItems": 2,
                "items": [
                   {
                      "type": "integer",
                      "exclusiveMinimum": 0
                   },
                   {
                      "type": "integer",
                      "exclusiveMinimum": 0
                   }
                ]
             },
             "w_lims": {
                "title": "W Lims",
                "description": "[min, max] interval from which window widths will be␣
↪uniformly randomly sampled. Only used if method = random.",
                "type": "array",
                "minItems": 2,
                "maxItems": 2,
                "items": [
                   {
                      "type": "integer",
                      "exclusiveMinimum": 0
                   },
                   {
                      "type": "integer",
                      "exclusiveMinimum": 0
                   }
                ]
```

(continues on next page)

(continued from previous page)


```
            },
            "max_windows": {
               "title": "Max Windows",
               "description": "Max allowed reads from a GeoDataset. Only used if␣
↪method = random.",
               "default": 10000,
               "minimum": 0,
               "type": "integer"
            },
            "max_sample_attempts": {
               "title": "Max Sample Attempts",
               "description": "Max attempts when trying to find a window within the␣
↪AOI of a scene. Only used if method = random and the scene has aoi_polygons␣
↪specified.",
               "default": 100,
               "exclusiveMinimum": 0,
               "type": "integer"
            },
            "efficient_aoi_sampling": {
               "title": "Efficient Aoi Sampling",
               "description": "If the scene has AOIs, sampling windows at random␣
↪anywhere in the extent and then checking if they fall within any of the AOIs can␣
↪be very inefficient. This flag enables the use of an alternate algorithm that␣
↪only samples window locations inside the AOIs. Only used if method = random and␣
↪the scene has aoi_polygons specified. Defaults to True",
               "default": true,
               "type": "boolean"
            },
            "type_hint": {
               "title": "Type Hint",
               "default": "geo_data_window",
               "enum": [
                  "geo_data_window"
               ],
               "type": "string"
            }
         },
         "required": [
            "size"
         ],
         "additionalProperties": false
      },
      "SemanticSegmentationGeoDataConfig": {
         "title": "SemanticSegmentationGeoDataConfig",
         "description": "Configure semantic segmentation :class:`GeoDatasets <.
↪GeoDataset>`.\n\nSee\n:mod:`rastervision.pytorch_learner.dataset.semantic_
↪segmentation_dataset`.",
         "type": "object",
         "properties": {
            "class_names": {
               "title": "Class Names",
               "description": "Names of classes.",
```


(continues on next page)

(continued from previous page)

```
                "default": [],
                "type": "array",
                "items": {
                   "type": "string"
                }
             },
             "class_colors": {
                "title": "Class Colors",
                "description": "Colors used to display classes. Can be color 3-
↪tuples in list form.",
                "type": "array",
                "items": {
                   "anyOf": [
                      {
                         "type": "string"
                      },
                      {
                         "type": "array",
                         "minItems": 3,
                         "maxItems": 3,
                         "items": [
                            {
                               "type": "integer"
                            },
                            {
                               "type": "integer"
                            },
                            {
                               "type": "integer"
                            }
                         ]
                      }
                   ]
                }
             },
             "img_channels": {
                "title": "Img Channels",
                "description": "The number of channels of the training images.",
                "exclusiveMinimum": 0,
                "type": "integer"
             },
             "img_sz": {
                "title": "Img Sz",
                "description": "Length of a side of each image in pixels. This is␣
↪the size to transform it to during training, not the size in the raw dataset.",
                "default": 256,
                "exclusiveMinimum": 0,
                "type": "integer"
             },
             "train_sz": {
                "title": "Train Sz",
                "description": "If set, the number of training images to use. If␣
```

(continues on next page)

(continued from previous page)

```
↪fewer images exist, then an exception will be raised.",
                "type": "integer"
            },
            "train_sz_rel": {
                "title": "Train Sz Rel",
                "description": "If set, the proportion of training images to use.",
                "type": "number"
            },
            "num_workers": {
                "title": "Num Workers",
                "description": "Number of workers to use when DataLoader makes␣
↪batches.",
                "default": 4,
                "type": "integer"
            },
            "augmentors": {
                "title": "Augmentors",
                "description": "Names of albumentations augmentors to use for␣
↪training batches. Choices include: ['Blur', 'RandomRotate90', 'HorizontalFlip',
↪'VerticalFlip', 'GaussianBlur', 'GaussNoise', 'RGBShift', 'ToGray'].␣
↪Alternatively, a custom transform can be provided via the aug_transform option.",
                "default": [
                    "RandomRotate90",
                    "HorizontalFlip",
                    "VerticalFlip"
                ],
                "type": "array",
                "items": {
                    "type": "string"
                }
            },
            "base_transform": {
                "title": "Base Transform",
                "description": "An Albumentations transform serialized as a dict␣
↪that will be applied to all datasets: training, validation, and test. This␣
↪transformation is in addition to the resizing due to img_sz. This is useful for,␣
↪for example, applying the same normalization to all datasets.",
                "type": "object"
            },
            "aug_transform": {
                "title": "Aug Transform",
                "description": "An Albumentations transform serialized as a dict␣
↪that will be applied as data augmentation to the training dataset. This transform␣
↪is applied before base_transform. If provided, the augmentors option is ignored.",
                "type": "object"
            },
            "plot_options": {
                "title": "Plot Options",
                "description": "Options to control plotting.",
                "default": {
                    "transform": {
                        "__version__": "1.3.0",
```

(continues on next page)

(continued from previous page)


```
                    "transform": {
                        "__class_fullname__": "rastervision.pytorch_learner.utils.
↪utils.MinMaxNormalize",
                        "always_apply": false,
                        "p": 1.0,
                        "min_val": 0.0,
                        "max_val": 1.0,
                        "dtype": 5
                    }
                },
                "channel_display_groups": null,
                "type_hint": "plot_options"
            },
            "allOf": [
                {
                    "$ref": "#/definitions/PlotOptions"
                }
            ]
        },
        "preview_batch_limit": {
            "title": "Preview Batch Limit",
            "description": "Optional limit on the number of items in the preview␣
↪plots produced during training.",
            "type": "integer"
        },
        "type_hint": {
            "title": "Type Hint",
            "default": "semantic_segmentation_geo_data",
            "enum": [
                "semantic_segmentation_geo_data"
            ],
            "type": "string"
        },
        "scene_dataset": {
            "$ref": "#/definitions/DatasetConfig"
        },
        "window_opts": {
            "title": "Window Opts",
            "default": {},
            "anyOf": [
                {
                    "$ref": "#/definitions/GeoDataWindowConfig"
                },
                {
                    "type": "object",
                    "additionalProperties": {
                        "$ref": "#/definitions/GeoDataWindowConfig"
                    }
                }
            ]
        }
    },
```


(continues on next page)

(continued from previous page)

```
            "additionalProperties": false
        }
    }
}
```

Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *data (Union[rastervision.pytorch_learner.semantic_segmentation_learner_config.SemanticSegmentationImageDataConfig, rastervision.pytorch_learner.semantic_segmentation_learner_config.SemanticSegmentationGeoDataConfig])*
- *eval_train (bool)*
- *log_tensorboard (bool)*
- *model (Optional[rastervision.pytorch_learner.semantic_segmentation_learner_config.SemanticSegmentationModelConfig])*
- *output_uri (Optional[str])*
- *overfit_mode (bool)*
- *predict_mode (bool)*
- *run_tensorboard (bool)*
- *save_model_bundle (bool)*
- *solver (rastervision.pytorch_learner.learner_config.SolverConfig)*
- *test_mode (bool)*
- *type_hint (Literal['semantic_segmentation_learner'])*

Validators

- `update_for_mode` » `all fields`
- `validate_class_loss_weights` » `all fields`
- `validate_run_tensorboard` » *run_tensorboard*

field data: Union[*SemanticSegmentationImageDataConfig*, *SemanticSegmentationGeoDataConfig*] [Required]

Validated by

- `update_for_mode`
- `validate_class_loss_weights`

field eval_train: bool = False

If True, runs final evaluation on training set (in addition to test set). Useful for debugging.

Validated by

- `update_for_mode`

- validate_class_loss_weights

field log_tensorboard: bool = True

Save Tensorboard log files at the end of each epoch.

Validated by

- update_for_mode
- validate_class_loss_weights

field model: Optional[*SemanticSegmentationModelConfig*] = None

Validated by

- update_for_mode
- validate_class_loss_weights

field output_uri: Optional[str] = None

URI of where to save output

Validated by

- update_for_mode
- validate_class_loss_weights

field overfit_mode: bool = False

If True, uses half image size, and instead of doing epoch-based training, optimizes the model using a single batch repeatedly for overfit_num_steps number of steps.

Validated by

- update_for_mode
- validate_class_loss_weights

field predict_mode: bool = False

If True, skips training, loads model, and does final eval.

Validated by

- update_for_mode
- validate_class_loss_weights

field run_tensorboard: bool = False

run Tensorboard server during training

Validated by

- update_for_mode
- validate_class_loss_weights
- validate_run_tensorboard

field save_model_bundle: bool = True

If True, saves a model bundle at the end of training which is zip file with model and this LearnerConfig which can be used to make predictions on new images at a later time.

Validated by

- update_for_mode
- validate_class_loss_weights

field solver: *SolverConfig* [Required]

Validated by

- update_for_mode
- validate_class_loss_weights

field test_mode: bool = False

If True, uses test_num_epochs, test_batch_sz, truncated datasets with only a single batch, image_sz that is cut in half, and num_workers = 0. This is useful for testing that code runs correctly on CPU without multithreading before running full job on GPU.

Validated by

- update_for_mode
- validate_class_loss_weights

field type_hint: Literal['semantic_segmentation_learner'] = 'semantic_segmentation_learner'

Validated by

- update_for_mode
- validate_class_loss_weights

build(*tmp_dir=None*, *model_weights_path=None*, *model_def_path=None*, *loss_def_path=None*, *training=True*)

Returns a Learner instantiated using this Config.

Parameters

- **tmp_dir** (*str*) – Root of temp dirs.
- **model_weights_path** (*str, optional*) – A local path to model weights. Defaults to None.
- **model_def_path** (*str, optional*) – A local path to a directory with a hubconf.py. If provided, the model definition is imported from here. Defaults to None.
- **loss_def_path** (*str, optional*) – A local path to a directory with a hubconf.py. If provided, the loss function definition is imported from here. Defaults to None.
- **training** (*bool, optional*) – Whether the model is to be used for training or prediction. If False, the model is put in eval mode and the loss function, optimizer, etc. are not initialized. Defaults to True.

get_model_bundle_uri() → str

Returns the URI of where the model bundle is stored.

Return type
str

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(**args*, ***kwargs*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

validator update_for_mode » *all fields*

Parameters
values (*dict*) –

Return type
dict

validator validate_class_loss_weights » *all fields*

Parameters
values (*dict*) –

Return type
dict

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises
ConfigError – if field is invalid

validator validate_run_tensorboard » run_tensorboard

Parameters

- **v** (*bool*) –
- **values** (*dict*) –

Return type
bool

SemanticSegmentationModelConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model SemanticSegmentationModelConfig

Configure a semantic segmentation model.


```
{
   "title": "SemanticSegmentationModelConfig",
   "description": "Configure a semantic segmentation model.",
   "type": "object",
   "properties": {
      "backbone": {
         "description": "The torchvision.models backbone to use. At the moment only␣
↪resnet50 or resnet101 will work.",
         "default": "resnet50",
         "allOf": [
            {
               "$ref": "#/definitions/Backbone"
            }
         ]
      },
      "pretrained": {
         "title": "Pretrained",
         "description": "If True, use ImageNet weights. If False, use random␣
↪initialization.",
         "default": true,
         "type": "boolean"
      },
      "init_weights": {
         "title": "Init Weights",
         "description": "URI of PyTorch model weights used to initialize model. If␣
↪set, this supercedes the pretrained option.",
         "type": "string"
      },
      "load_strict": {
         "title": "Load Strict",
         "description": "If True, the keys in the state dict referenced by init_
↪weights must match exactly. Setting this to False can be useful if you just want␣
↪to load the backbone of a model.",
         "default": true,
         "type": "boolean"
      },
      "external_def": {
         "title": "External Def",
         "description": "If specified, the model will be built from the definition␣
↪from this external source, using Torch Hub.",
         "allOf": [
            {
               "$ref": "#/definitions/ExternalModuleConfig"
            }
         ]
```


(continues on next page)

(continued from previous page)


```
    },
    "type_hint": {
      "title": "Type Hint",
      "default": "semantic_segmentation_model",
      "enum": [
        "semantic_segmentation_model"
      ],
      "type": "string"
    }
  },
  "additionalProperties": false,
  "definitions": {
    "Backbone": {
      "title": "Backbone",
      "description": "An enumeration.",
      "enum": [
        "alexnet",
        "densenet121",
        "densenet169",
        "densenet201",
        "densenet161",
        "googlenet",
        "inception_v3",
        "mnasnet0_5",
        "mnasnet0_75",
        "mnasnet1_0",
        "mnasnet1_3",
        "mobilenet_v2",
        "resnet18",
        "resnet34",
        "resnet50",
        "resnet101",
        "resnet152",
        "resnext50_32x4d",
        "resnext101_32x8d",
        "wide_resnet50_2",
        "wide_resnet101_2",
        "shufflenet_v2_x0_5",
        "shufflenet_v2_x1_0",
        "shufflenet_v2_x1_5",
        "shufflenet_v2_x2_0",
        "squeezenet1_0",
        "squeezenet1_1",
        "vgg11",
        "vgg11_bn",
        "vgg13",
        "vgg13_bn",
        "vgg16",
        "vgg16_bn",
        "vgg19_bn",
        "vgg19"
      ]
```


(continues on next page)

(continued from previous page)

```
      },
      "ExternalModuleConfig": {
         "title": "ExternalModuleConfig",
         "description": "Config describing an object to be loaded via Torch Hub.",
         "type": "object",
         "properties": {
            "uri": {
               "title": "Uri",
               "description": "Local uri of a zip file, or local uri of a directory,
↪or remote uri of zip file.",
               "minLength": 1,
               "type": "string"
            },
            "github_repo": {
               "title": "Github Repo",
               "description": "<repo-owner>/<repo-name>[:tag]",
               "pattern": ".+/.+",
               "type": "string"
            },
            "name": {
               "title": "Name",
               "description": "Name of the folder in which to extract/copy the␣
↪definition files.",
               "minLength": 1,
               "type": "string"
            },
            "entrypoint": {
               "title": "Entrypoint",
               "description": "Name of a callable present in hubconf.py. See docs␣
↪for torch.hub for details.",
               "minLength": 1,
               "type": "string"
            },
            "entrypoint_args": {
               "title": "Entrypoint Args",
               "description": "Args to pass to the entrypoint. Must be serializable.
↪",
               "default": [],
               "type": "array",
               "items": {}
            },
            "entrypoint_kwargs": {
               "title": "Entrypoint Kwargs",
               "description": "Keyword args to pass to the entrypoint. Must be␣
↪serializable.",
               "default": {},
               "type": "object"
            },
            "force_reload": {
               "title": "Force Reload",
               "description": "Force reload of module definition.",
               "default": false,
```

(continues on next page)

(continued from previous page)


```
                "type": "boolean"
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "external-module",
                "enum": [
                    "external-module"
                ],
                "type": "string"
            }
        },
        "required": [
            "entrypoint"
        ],
        "additionalProperties": false
    }
}
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *backbone (rastervision.pytorch_learner.learner_config.Backbone)*
- *external_def (Optional[rastervision.pytorch_learner.learner_config.ExternalModuleConfig])*
- *init_weights (Optional[str])*
- *load_strict (bool)*
- *pretrained (bool)*
- *type_hint (Literal['semantic_segmentation_model'])*

Validators

- *only_valid_backbones* » *backbone*

field backbone: *Backbone* = Backbone.resnet50

The torchvision.models backbone to use. At the moment only resnet50 or resnet101 will work.

Validated by

- *only_valid_backbones*

field external_def: Optional[*ExternalModuleConfig*] = None

If specified, the model will be built from the definition from this external source, using Torch Hub.

field init_weights: Optional[str] = None

URI of PyTorch model weights used to initialize model. If set, this supercedes the pretrained option.

field load_strict: bool = True

If True, the keys in the state dict referenced by init_weights must match exactly. Setting this to False can be useful if you just want to load the backbone of a model.

field pretrained: bool = True

If True, use ImageNet weights. If False, use random initialization.

field type_hint: Literal['semantic_segmentation_model'] = 'semantic_segmentation_model'

build(*num_classes: int*, *in_channels: int*, *save_dir: Optional[str] = None*, *hubconf_dir: Optional[str] = None*, ***kwargs*) → torch.nn.Module

Build and return a model based on the config.

Parameters

- **num_classes** (*int*) – Number of classes.
- **in_channels** (*int, optional*) – Number of channels in the images that will be fed into the model. Defaults to 3.
- **save_dir** (*Optional[str], optional*) – Used for building external_def if specified. Defaults to None.
- **hubconf_dir** (*Optional[str], optional*) – Used for building external_def if specified. Defaults to None.

Returns
a PyTorch nn.Module.

Return type
nn.Module

build_default_model(*num_classes: int*, *in_channels: int*) → torch.nn.Module

Build and return the default model.

Parameters

- **num_classes** (*int*) – Number of classes.
- **in_channels** (*int, optional*) – Number of channels in the images that will be fed into the model. Defaults to 3.

Returns
a PyTorch nn.Module.

Return type
nn.Module

build_external_model(*save_dir: str*, *hubconf_dir: Optional[str] = None*) → torch.nn.Module

Build and return an external model.

Parameters

- **save_dir** (*str*) – The module def will be saved here.
- **hubconf_dir** (*Optional[str], optional*) – Path to existing definition. Defaults to None.

Returns
a PyTorch nn.Module.

Return type
nn.Module

`get_backbone_str`()

`validator only_valid_backbones » backbone`

`recursive_validate_config`()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

`revalidate`()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

`update`(**args*, ***kwargs*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

`validate_config`()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

`validate_list`(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises
ConfigError – if field is invalid

9.3.15 utils

Modules

torch_hub

utils

torch_hub

Functions

get_hubconf_dir_from_cfg(cfg[, parent])	Determine destination directory name from an ExternalModuleConfig.
torch_hub_load_github(repo, hubconf_dir, ...)	Load an entrypoint from a github repo using torch.hub.load().
torch_hub_load_local(hubconf_dir, ...)	
torch_hub_load_uri(uri, hubconf_dir, ...)	Load an entrypoint from a uri.

get_hubconf_dir_from_cfg

get_hubconf_dir_from_cfg(*cfg*, *parent: Optional[str] = ''*) → str

Determine destination directory name from an ExternalModuleConfig.

If a parent path is provided, the dir name is appended to it.

Parameters

- **cfg** (*ExternalModuleConfig*) – an ExternalModuleConfig
- **parent** (*str, optional*) – Parent path. Defaults to ‘’.

Returns
directory name or path

Return type
str

torch_hub_load_github

torch_hub_load_github(*repo: str*, *hubconf_dir: str*, *entrypoint: str*, **args*, ***kwargs*) → Any

Load an entrypoint from a github repo using torch.hub.load().

Parameters

- **repo** (*str*) – <repo-owner>/<erpo-name>[:tag]
- **hubconf_dir** (*str*) – Where the contents from the uri will finally be saved to.
- **entrypoint** (*str*) – Name of a callable present in hubconf.py.
- ***args** – Args to be passed to the entrypoint.
- ****kwargs** – Keyword args to be passed to the entrypoint.

Returns
The output from calling the entrypoint.

Return type
Any

torch_hub_load_local

torch_hub_load_local(*hubconf_dir: str*, *entrypoint: str*, **args*, ***kwargs*) → Any

Parameters

- **hubconf_dir** (*str*) –
- **entrypoint** (*str*) –

Return type
Any

torch_hub_load_uri

torch_hub_load_uri(*uri: str*, *hubconf_dir: str*, *entrypoint: str*, **args*, ***kwargs*) → Any

Load an entrypoint from a uri.

Load an entrypoint from:

- a local uri of a zip file, or
- a local uri of a directory, or
- a remote uri of zip file.

The zip file should either have hubconf.py at the top level or contain a single sub-directory that contains hubconf.py at its top level. In the latter case, the sub-directory will be copied to hubconf_dir.

Parameters

- **uri** (*str*) – A URI.
- **hubconf_dir** (*str*) – The target directory where the contents from the uri will finally be saved to.
- **entrypoint** (*str*) – Name of a callable present in hubconf.py.
- ***args** – Args to be passed to the entrypoint.
- ****kwargs** – Keyword args to be passed to the entrypoint.

Returns
The output from calling the entrypoint.

Return type
Any

utils

Classes

AddTensors	Adds all its inputs together.
MinMaxNormalize	Albumentations transform that normalizes image to desired min and max values.
Parallel	Passes inputs through multiple `nn.Module`s in parallel. Returns a tuple of outputs.
SplitTensor	Wrapper around *torch.split*

AddTensors

class AddTensors

Bases: Module

Adds all its inputs together.

__init__(**args: Any*, ***kwargs: Any*) → None

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

Methods

__init__(*args, **kwargs)
forward(xs)

__init__(**args: Any*, ***kwargs: Any*) → None

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

static __new__(*cls*, **args: Any*, ***kwargs: Any*) → Any

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
Any

forward(*xs*)

MinMaxNormalize

class MinMaxNormalize

Bases: ImageOnlyTransform

Albumentations transform that normalizes image to desired min and max values.

This will shift and scale the image appropriately to achieve the desired min and max.

Attributes

call_backup

target_dependence

targets

targets_as_params

__init__(*min_val=0.0*, *max_val=1.0*, *dtype=5*, *always_apply=False*, *p=1.0*)

Constructor.

Parameters

- **min_val** – the minimum value that output should have
- **max_val** – the maximum value that output should have
- **dtype** – the dtype of output image

Methods

__init__([min_val, max_val, dtype, ...])	Constructor.
add_targets(additional_targets)	Add targets to transform them the same way as one of existing targets ex: {'target_image': 'image'} ex: {'obj1_mask': 'mask', 'obj2_mask': 'mask'} by the way you must have at least one object with key 'image'
apply(image, **params)	
apply_with_params(params, **kwargs)	
get_base_init_args()	
get_class_fullname()	
get_dict_with_id()	
get_params()	
get_params_dependent_on_targets(params)	
get_transform_init_args()	
get_transform_init_args_names()	
is_serializable()	
set_deterministic(flag[, save_key])	
to_dict([on_not_implemented_error])	Take a transform pipeline and convert it to a serializable representation that uses only standard python data types: dictionaries, lists, strings, integers, and floats.
update_params(params, **kwargs)	

__init__(*min_val=0.0*, *max_val=1.0*, *dtype=5*, *always_apply=False*, *p=1.0*)

Constructor.

Parameters

- **min_val** – the minimum value that output should have
- **max_val** – the maximum value that output should have
- **dtype** – the dtype of output image

add_targets(*additional_targets: Dict[str, str]*)

Add targets to transform them the same way as one of existing targets ex: {'target_image': 'image'} ex: {'obj1_mask': 'mask', 'obj2_mask': 'mask'} by the way you must have at least one object with key 'image'

Parameters

additional_targets (*dict*) – keys - new target name, values - old target name. ex: {'image2': 'image'}

apply(*image*, ***params*)

apply_with_params(*params: Dict[str, Any]*, ***kwargs*) → Dict[str, Any]

Parameters
params (*Dict[str, Any]*) –

Return type
Dict[str, *Any*]

get_base_init_args() → Dict[str, Any]

Return type
Dict[str, *Any*]

classmethod get_class_fullname() → str

Return type
str

get_dict_with_id() → Dict[str, Any]

Return type
Dict[str, *Any*]

get_params() → Dict

Return type
Dict

get_params_dependent_on_targets(*params: Dict[str, Any]*) → Dict[str, Any]

Parameters
params (*Dict[str, Any]*) –

Return type
Dict[str, *Any*]

get_transform_init_args() → Dict[str, Any]

Return type
Dict[str, *Any*]

get_transform_init_args_names()

classmethod is_serializable()

set_deterministic(*flag: bool*, *save_key: str = 'replay'*) → BasicTransform

Parameters

- **flag** (*bool*) –
- **save_key** (*str*) –

Return type
BasicTransform

to_dict(*on_not_implemented_error: str = 'raise'*) → Dict[str, Any]

Take a transform pipeline and convert it to a serializable representation that uses only standard python data types: dictionaries, lists, strings, integers, and floats.

Parameters

- **self** – A transform that should be serialized. If the transform doesn't implement the *to_dict* method and *on_not_implemented_error* equals to 'raise' then *NotImplementedError* is raised. If *on_not_implemented_error* equals to 'warn' then *NotImplementedError* will be ignored but no transform parameters will be serialized.
- **on_not_implemented_error** (*str*) – *raise* or *warn*.

Return type
Dict[str, *Any*]

update_params(*params: Dict[str, Any]*, ***kwargs*) → Dict[str, Any]

Parameters
params (*Dict[str, Any]*) –

Return type
Dict[str, *Any*]

call_backup = None

fill_value: Any

interpolation: Any

mask_fill_value: Any

property target_dependence: Dict

property targets: Dict[str, Callable]

property targets_as_params: List[str]

Parallel

class Parallel

Bases: ModuleList

Passes inputs through multiple `nn.Module`s in parallel. Returns a tuple of outputs.

__init__(**args*)

Methods

__init__(*args)
forward(xs)

__init__(**args*)

static __new__(*cls*, **args: Any*, ***kwargs: Any*) → Any

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
Any

forward(*xs*)

SplitTensor

class SplitTensor

Bases: Module

Wrapper around *torch.split*

__init__(*size_or_sizes*, *dim*)

Methods

__init__(size_or_sizes, dim)
forward(X)

__init__(*size_or_sizes*, *dim*)

static __new__(*cls*, **args: Any*, ***kwargs: Any*) → Any

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
Any

forward(*X*)

Functions

adjust_conv_channels(old_conv, in_channels)	
channel_groups_to_imgs(x, channel_groups)	
color_to_triple([color])	Given a PIL ImageColor string, return a triple of integers representing the red, green, and blue values.
compute_conf_mat(out, y, num_labels)	
compute_conf_mat_metrics(conf_mat, label_names)	
deserialize_albumentation_transform(tf_dict)	Deserialize an albumentations transform serialized by *serialize_albumentation_transform()*.
plot_channel_groups(axs, imgs, channel_groups)	
serialize_albumentation_transform(tf[, ...])	Serialize an albumentations transform to a dict.
validate_albumentation_transform(tf_dict)	Validate a serialized albumentation transform by attempting to deserialize it.

adjust_conv_channels

adjust_conv_channels(*old_conv: torch.nn.Conv2d*, *in_channels: int*, *pretrained: bool = True*) → Union[torch.nn.Conv2d, torch.nn.Sequential]

Parameters

- **old_conv** (*torch.nn.Conv2d*) –
- **in_channels** (*int*) –
- **pretrained** (*bool*) –

Return type
Union[torch.nn.Conv2d, torch.nn.Sequential]

channel_groups_to_imgs

channel_groups_to_imgs(*x: torch.Tensor*, *channel_groups: Dict[str, Sequence[int]]*) → List[torch.Tensor]

Parameters

- **x** (*torch.Tensor*) –
- **channel_groups** (*Dict[str, Sequence[int]]*) –

Return type
List[torch.Tensor]

color_to_triple

color_to_triple(*color: Optional[str] = None*) → Tuple[int, int, int]

Given a PIL ImageColor string, return a triple of integers representing the red, green, and blue values.

If color is None, return a random color.

Parameters
color (*Optional[str]*) – A PIL ImageColor string

Returns
An triple of integers

Return type
Tuple[int, int, int]

compute_conf_mat

compute_conf_mat(*out*, *y*, *num_labels*)

compute_conf_mat_metrics

compute_conf_mat_metrics(*conf_mat*, *label_names*, *eps=1e-06*)

deserialize_albumentation_transform

deserialize_albumentation_transform(*tf_dict: dict*) → BasicTransform

Deserialize an albumentations transform serialized by *serialize_albumentation_transform()*.

If the input dict contains a *lambda_transforms_path*, the *lambda_transforms* dict is dynamically imported from it and passed to *A.from_dict()*. See https://albumentations.ai/docs/examples/serialization/ for details

Parameters
tf_dict (*dict*) – Serialized albumentations transform.

Returns
Deserialized transform.

Return type
A.BasicTransform

plot_channel_groups

plot_channel_groups(*axs: Iterable*, *imgs: Iterable[Union[array, torch.Tensor]]*, *channel_groups: dict*) → None

Parameters

- **axs** (*Iterable*) –
- **imgs** (*Iterable[Union[array, torch.Tensor]]*) –
- **channel_groups** (*dict*) –

Return type
None

serialize_albumentation_transform

serialize_albumentation_transform(*tf: BasicTransform*, *lambda_transforms_path: Optional[str] = None*, *dst_dir: Optional[str] = None*) → dict

Serialize an albumentations transform to a dict.

If the transform includes a Lambda transform, a *lambda_transforms_path* should be provided. This should be a path to a python file that defines a dict named *lambda_transforms* as required by *A.from_dict()*. See https://albumentations.ai/docs/examples/serialization/ for details. This path is saved as a field in the returned dict so that it is available at the time of deserialization.

Parameters

- **tf** (*A.BasicTransform*) – The transform to serialize.
- **lambda_transforms_path** (*Optional[str], optional*) – Path to a python file that defines a dict named *lambda_transforms* as required by *A.from_dict()*. Defaults to None.
- **dst_dir** (*Optional[str], optional*) – Directory to copy the transforms file to. Useful for copying the file to S3 when running on Batch. Defaults to None.

Returns
The serialized transform.

Return type
dict

validate_albumentation_transform

validate_albumentation_transform(*tf_dict: Optional[dict]*) → dict

Validate a serialized albumentation transform by attempting to deserialize it.

Parameters
tf_dict (*Optional[dict]*) –

Return type
dict

9.4 pytorch_backend

Modules

pytorch_chip_classification

pytorch_chip_classification_config

pytorch_learner_backend

pytorch_learner_backend_config

pytorch_object_detection

pytorch_object_detection_config

pytorch_semantic_segmentation

pytorch_semantic_segmentation_config

9.4.1 pytorch_chip_classification

Classes

PyTorchChipClassification

PyTorchChipClassificationSampleWriter

PyTorchChipClassification

class PyTorchChipClassification

Bases: *PyTorchLearnerBackend*

__init__(*pipeline_cfg:* RVPipelineConfig, *learner_cfg:* LearnerConfig, *tmp_dir:* *str*)

Parameters

- **pipeline_cfg** (RVPipelineConfig) –
- **learner_cfg** (LearnerConfig) –
- **tmp_dir** (*str*) –

Methods

__init__(pipeline_cfg, learner_cfg, tmp_dir)	

get_sample_writer()	Returns a SampleWriter for this Backend.
load_model()	Load the model in preparation for one or more prediction calls.
predict_scene(scene, chip_sz[, stride])	Return predictions for an entire scene using the model.
train([source_bundle_uri])	Train a model.

__init__(*pipeline_cfg:* RVPipelineConfig, *learner_cfg:* LearnerConfig, *tmp_dir:* *str*)

Parameters

- **pipeline_cfg** (RVPipelineConfig) –
- **learner_cfg** (LearnerConfig) –
- **tmp_dir** (*str*) –

get_sample_writer()

Returns a SampleWriter for this Backend.

load_model()

Load the model in preparation for one or more prediction calls.

predict_scene(*scene:* Scene, *chip_sz:* *int*, *stride:* *Optional[int]* = *None*) → *ChipClassificationLabels*

Return predictions for an entire scene using the model.

Parameters

- **scene** (Scene) – Scene to run inference on.
- **chip_sz** (*int*) –
- **stride** (*Optional[int]*) –

Returns

Labels object containing predictions

Return type

ChipClassificationLabels

train(*source_bundle_uri=None*)

Train a model.

This should download chips created by the SampleWriter, train the model, and then saving it to disk.

PyTorchChipClassificationSampleWriter

class PyTorchChipClassificationSampleWriter

Bases: *PyTorchLearnerSampleWriter*

__init__(*output_uri: str*, *class_config:* ClassConfig, *tmp_dir: str*)

Constructor.

Parameters

- **output_uri** (*str*) – URI of directory where zip file of chips should be placed.
- **class_config** (ClassConfig) – used to convert class ids to names which may be needed for some training data formats.
- **tmp_dir** (*str*) – local directory which is root of any temporary directories that are created.

Methods

__init__(output_uri, class_config, tmp_dir)	Constructor.
get_image_ext(chip)	Decide which format to store the image in.
get_image_path(split_name, sample, class_id)	Decide the save location of the image.
write_chip(chip, path)	Save chip as either a PNG image or a numpy array.
write_sample(sample)	This writes a training or validation sample to (train|valid)/{class_name}/{scene_id}-{ind}.png

__init__(*output_uri: str*, *class_config:* ClassConfig, *tmp_dir: str*)

Constructor.

Parameters

- **output_uri** (*str*) – URI of directory where zip file of chips should be placed.
- **class_config** (ClassConfig) – used to convert class ids to names which may be needed for some training data formats.
- **tmp_dir** (*str*) – local directory which is root of any temporary directories that are created.

get_image_ext(*chip: ndarray*) → str

Decide which format to store the image in.

Parameters
chip (*ndarray*) –

Return type
str

get_image_path(*split_name: str*, *sample: DataSample*, *class_id: int*) → str

Decide the save location of the image. Also, ensure that the target directory exists.

Parameters

- **split_name** (*str*) –
- **sample** (*DataSample*) –
- **class_id** (*int*) –

Return type
str

write_chip(*chip: ndarray*, *path: str*) → None

Save chip as either a PNG image or a numpy array.

Parameters

- **chip** (*ndarray*) –
- **path** (*str*) –

Return type
None

write_sample(*sample: DataSample*)

This writes a training or validation sample to (train|valid)/{class_name}/{scene_id}-{ind}.png

Parameters
sample (*DataSample*) –

9.4.2 pytorch_chip_classification_config

Configs

PyTorchChipClassificationConfig	Configure a *PyTorchChipClassification* backend.

PyTorchChipClassificationConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model PyTorchChipClassificationConfig

Configure a *PyTorchChipClassification* backend.


```
{
   "title": "PyTorchChipClassificationConfig",
   "description": "Configure a :class:`.PyTorchChipClassification` backend.",
   "type": "object",
   "properties": {
      "type_hint": {
         "title": "Type Hint",
         "default": "pytorch_chip_classification_backend",
         "enum": [
            "pytorch_chip_classification_backend"
         ],
         "type": "string"
      },
      "model": {
         "$ref": "#/definitions/ClassificationModelConfig"
      },
      "solver": {
         "$ref": "#/definitions/SolverConfig"
      },
      "data": {
```


(continues on next page)

(continued from previous page)


```
            "$ref": "#/definitions/DataConfig"
        },
        "log_tensorboard": {
            "title": "Log Tensorboard",
            "description": "If True, log events to Tensorboard log files.",
            "default": true,
            "type": "boolean"
        },
        "run_tensorboard": {
            "title": "Run Tensorboard",
            "description": "If True, run Tensorboard server pointing at log files.",
            "default": false,
            "type": "boolean"
        },
        "test_mode": {
            "title": "Test Mode",
            "description": "This field is passed along to the LearnerConfig which is␣
↪returned by get_learner_config(). For more info, see the docs forpytorch_learner.
↪learner_config.LearnerConfig.test_mode.",
            "default": false,
            "type": "boolean"
        }
    },
    "required": [
        "model",
        "solver",
        "data"
    ],
    "additionalProperties": false,
    "definitions": {
        "Backbone": {
            "title": "Backbone",
            "description": "An enumeration.",
            "enum": [
                "alexnet",
                "densenet121",
                "densenet169",
                "densenet201",
                "densenet161",
                "googlenet",
                "inception_v3",
                "mnasnet0_5",
                "mnasnet0_75",
                "mnasnet1_0",
                "mnasnet1_3",
                "mobilenet_v2",
                "resnet18",
                "resnet34",
                "resnet50",
                "resnet101",
                "resnet152",
                "resnext50_32x4d",
```


(continues on next page)

(continued from previous page)

```
            "resnext101_32x8d",
            "wide_resnet50_2",
            "wide_resnet101_2",
            "shufflenet_v2_x0_5",
            "shufflenet_v2_x1_0",
            "shufflenet_v2_x1_5",
            "shufflenet_v2_x2_0",
            "squeezenet1_0",
            "squeezenet1_1",
            "vgg11",
            "vgg11_bn",
            "vgg13",
            "vgg13_bn",
            "vgg16",
            "vgg16_bn",
            "vgg19_bn",
            "vgg19"
         ]
      },
      "ExternalModuleConfig": {
         "title": "ExternalModuleConfig",
         "description": "Config describing an object to be loaded via Torch Hub.",
         "type": "object",
         "properties": {
            "uri": {
               "title": "Uri",
               "description": "Local uri of a zip file, or local uri of a directory,
↪or remote uri of zip file.",
               "minLength": 1,
               "type": "string"
            },
            "github_repo": {
               "title": "Github Repo",
               "description": "<repo-owner>/<repo-name>[:tag]",
               "pattern": ".+/.+",
               "type": "string"
            },
            "name": {
               "title": "Name",
               "description": "Name of the folder in which to extract/copy the␣
↪definition files.",
               "minLength": 1,
               "type": "string"
            },
            "entrypoint": {
               "title": "Entrypoint",
               "description": "Name of a callable present in hubconf.py. See docs␣
↪for torch.hub for details.",
               "minLength": 1,
               "type": "string"
            },
            "entrypoint_args": {
```

(continues on next page)

(continued from previous page)


```
                "title": "Entrypoint Args",
                "description": "Args to pass to the entrypoint. Must be serializable.
↪",
                "default": [],
                "type": "array",
                "items": {}
            },
            "entrypoint_kwargs": {
                "title": "Entrypoint Kwargs",
                "description": "Keyword args to pass to the entrypoint. Must be␣
↪serializable.",
                "default": {},
                "type": "object"
            },
            "force_reload": {
                "title": "Force Reload",
                "description": "Force reload of module definition.",
                "default": false,
                "type": "boolean"
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "external-module",
                "enum": [
                    "external-module"
                ],
                "type": "string"
            }
        },
        "required": [
            "entrypoint"
        ],
        "additionalProperties": false
    },
    "ClassificationModelConfig": {
        "title": "ClassificationModelConfig",
        "description": "Configure a classification model.",
        "type": "object",
        "properties": {
            "backbone": {
                "description": "The torchvision.models backbone to use.",
                "default": "resnet18",
                "allOf": [
                    {
                        "$ref": "#/definitions/Backbone"
                    }
                ]
            },
            "pretrained": {
                "title": "Pretrained",
                "description": "If True, use ImageNet weights. If False, use random␣
↪initialization.",
```


(continues on next page)

(continued from previous page)


```
                "default": true,
                "type": "boolean"
            },
            "init_weights": {
                "title": "Init Weights",
                "description": "URI of PyTorch model weights used to initialize␣
↪model. If set, this supercedes the pretrained option.",
                "type": "string"
            },
            "load_strict": {
                "title": "Load Strict",
                "description": "If True, the keys in the state dict referenced by␣
↪init_weights must match exactly. Setting this to False can be useful if you just␣
↪want to load the backbone of a model.",
                "default": true,
                "type": "boolean"
            },
            "external_def": {
                "title": "External Def",
                "description": "If specified, the model will be built from the␣
↪definition from this external source, using Torch Hub.",
                "allOf": [
                    {
                        "$ref": "#/definitions/ExternalModuleConfig"
                    }
                ]
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "classification_model",
                "enum": [
                    "classification_model"
                ],
                "type": "string"
            }
        },
        "additionalProperties": false
    },
    "SolverConfig": {
        "title": "SolverConfig",
        "description": "Config related to solver aka optimizer.",
        "type": "object",
        "properties": {
            "lr": {
                "title": "Lr",
                "description": "Learning rate.",
                "default": 0.0001,
                "exclusiveMinimum": 0,
                "type": "number"
            },
            "num_epochs": {
                "title": "Num Epochs",
```


(continues on next page)

(continued from previous page)

```
                "description": "Number of epochs (ie. sweeps through the whole␣
↪training set).",
                "default": 10,
                "exclusiveMinimum": 0,
                "type": "integer"
            },
            "test_num_epochs": {
                "title": "Test Num Epochs",
                "description": "Number of epochs to use in test mode.",
                "default": 2,
                "exclusiveMinimum": 0,
                "type": "integer"
            },
            "test_batch_sz": {
                "title": "Test Batch Sz",
                "description": "Batch size to use in test mode.",
                "default": 4,
                "exclusiveMinimum": 0,
                "type": "integer"
            },
            "overfit_num_steps": {
                "title": "Overfit Num Steps",
                "description": "Number of optimizer steps to use in overfit mode.",
                "default": 1,
                "exclusiveMinimum": 0,
                "type": "integer"
            },
            "sync_interval": {
                "title": "Sync Interval",
                "description": "The interval in epochs for each sync to the cloud.",
                "default": 1,
                "exclusiveMinimum": 0,
                "type": "integer"
            },
            "batch_sz": {
                "title": "Batch Sz",
                "description": "Batch size.",
                "default": 32,
                "exclusiveMinimum": 0,
                "type": "integer"
            },
            "one_cycle": {
                "title": "One Cycle",
                "description": "If True, use triangular LR scheduler with a single␣
↪cycle across all epochs with start and end LR being lr/10 and the peak being lr.",
                "default": true,
                "type": "boolean"
            },
            "multi_stage": {
                "title": "Multi Stage",
                "description": "List of epoch indices at which to divide LR by 10.",
                "default": [],
```

(continues on next page)

(continued from previous page)

```
                "type": "array",
                "items": {}
            },
            "class_loss_weights": {
                "title": "Class Loss Weights",
                "description": "Class weights for weighted loss.",
                "type": "array",
                "items": {
                    "type": "number"
                }
            },
            "ignore_class_index": {
                "title": "Ignore Class Index",
                "description": "If specified, this index is ignored when computing␣
↪the loss. See pytorch documentation for nn.CrossEntropyLoss for more details.␣
↪This can also be negative, in which case it is treated as a negative slice index␣
↪i.e. -1 = last index, -2 = second-last index, and so on.",
                "type": "integer"
            },
            "external_loss_def": {
                "title": "External Loss Def",
                "description": "If specified, the loss will be built from the␣
↪definition from this external source, using Torch Hub.",
                "allOf": [
                    {
                        "$ref": "#/definitions/ExternalModuleConfig"
                    }
                ]
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "solver",
                "enum": [
                    "solver"
                ],
                "type": "string"
            }
        },
        "additionalProperties": false
    },
    "PlotOptions": {
        "title": "PlotOptions",
        "description": "Config related to plotting.",
        "type": "object",
        "properties": {
            "transform": {
                "title": "Transform",
                "description": "An Albumentations transform serialized as a dict␣
↪that will be applied to each image before it is plotted. Mainly useful for␣
↪undoing any data transformation that you do not want included in the plot, such␣
↪as normalization. The default value will shift and scale the image so the values␣
↪range from 0.0 to 1.0 which is the expected range for the plotting function. This␣
```

(continues on next page)

(continued from previous page)

```
↪default is useful for cases where the values after normalization are close to␣
↪zero which makes the plot difficult to see.",
               "default": {
                  "__version__": "1.3.0",
                  "transform": {
                     "__class_fullname__": "rastervision.pytorch_learner.utils.
↪utils.MinMaxNormalize",
                     "always_apply": false,
                     "p": 1.0,
                     "min_val": 0.0,
                     "max_val": 1.0,
                     "dtype": 5
                  }
               },
               "type": "object"
            },
            "channel_display_groups": {
               "title": "Channel Display Groups",
               "description": "Groups of image channels to display together as a␣
↪subplot when plotting the data and predictions. Can be a list or tuple of groups␣
↪(e.g. [(0, 1, 2), (3,)]) or a dict containing title-to-group mappings (e.g. {\
↪"RGB\": [0, 1, 2], \"IR\": [3]}), where each group is a list or tuple of channel␣
↪indices and title is a string that will be used as the title of the subplot for␣
↪that group.",
               "anyOf": [
                  {
                     "type": "object",
                     "additionalProperties": {
                        "type": "array",
                        "items": {
                           "type": "integer",
                           "minimum": 0
                        }
                     }
                  },
                  {
                     "type": "array",
                     "items": {
                        "type": "array",
                        "items": {
                           "type": "integer",
                           "minimum": 0
                        }
                     }
                  }
               ]
            },
            "type_hint": {
               "title": "Type Hint",
               "default": "plot_options",
               "enum": [
                  "plot_options"
```

(continues on next page)

(continued from previous page)


```
                ],
                "type": "string"
            }
        },
        "additionalProperties": false
    },
    "DataConfig": {
        "title": "DataConfig",
        "description": "Config related to dataset for training and testing.",
        "type": "object",
        "properties": {
            "class_names": {
                "title": "Class Names",
                "description": "Names of classes.",
                "default": [],
                "type": "array",
                "items": {
                    "type": "string"
                }
            },
            "class_colors": {
                "title": "Class Colors",
                "description": "Colors used to display classes. Can be color 3-
↪tuples in list form.",
                "type": "array",
                "items": {
                    "anyOf": [
                        {
                            "type": "string"
                        },
                        {
                            "type": "array",
                            "minItems": 3,
                            "maxItems": 3,
                            "items": [
                                {
                                    "type": "integer"
                                },
                                {
                                    "type": "integer"
                                },
                                {
                                    "type": "integer"
                                }
                            ]
                        }
                    ]
                }
            },
            "img_channels": {
                "title": "Img Channels",
                "description": "The number of channels of the training images.",
```


(continues on next page)

(continued from previous page)

```
                "exclusiveMinimum": 0,
                "type": "integer"
            },
            "img_sz": {
                "title": "Img Sz",
                "description": "Length of a side of each image in pixels. This is␣
↪the size to transform it to during training, not the size in the raw dataset.",
                "default": 256,
                "exclusiveMinimum": 0,
                "type": "integer"
            },
            "train_sz": {
                "title": "Train Sz",
                "description": "If set, the number of training images to use. If␣
↪fewer images exist, then an exception will be raised.",
                "type": "integer"
            },
            "train_sz_rel": {
                "title": "Train Sz Rel",
                "description": "If set, the proportion of training images to use.",
                "type": "number"
            },
            "num_workers": {
                "title": "Num Workers",
                "description": "Number of workers to use when DataLoader makes␣
↪batches.",
                "default": 4,
                "type": "integer"
            },
            "augmentors": {
                "title": "Augmentors",
                "description": "Names of albumentations augmentors to use for␣
↪training batches. Choices include: ['Blur', 'RandomRotate90', 'HorizontalFlip',
↪'VerticalFlip', 'GaussianBlur', 'GaussNoise', 'RGBShift', 'ToGray'].␣
↪Alternatively, a custom transform can be provided via the aug_transform option.",
                "default": [
                    "RandomRotate90",
                    "HorizontalFlip",
                    "VerticalFlip"
                ],
                "type": "array",
                "items": {
                    "type": "string"
                }
            },
            "base_transform": {
                "title": "Base Transform",
                "description": "An Albumentations transform serialized as a dict␣
↪that will be applied to all datasets: training, validation, and test. This␣
↪transformation is in addition to the resizing due to img_sz. This is useful for,␣
↪for example, applying the same normalization to all datasets.",
                "type": "object"
```

(continues on next page)

(continued from previous page)


```
            },
            "aug_transform": {
               "title": "Aug Transform",
               "description": "An Albumentations transform serialized as a dict␣
↪that will be applied as data augmentation to the training dataset. This transform␣
↪is applied before base_transform. If provided, the augmentors option is ignored.",
               "type": "object"
            },
            "plot_options": {
               "title": "Plot Options",
               "description": "Options to control plotting.",
               "default": {
                  "transform": {
                     "__version__": "1.3.0",
                     "transform": {
                        "__class_fullname__": "rastervision.pytorch_learner.utils.
↪utils.MinMaxNormalize",
                        "always_apply": false,
                        "p": 1.0,
                        "min_val": 0.0,
                        "max_val": 1.0,
                        "dtype": 5
                     }
                  },
                  "channel_display_groups": null,
                  "type_hint": "plot_options"
               },
               "allOf": [
                  {
                     "$ref": "#/definitions/PlotOptions"
                  }
               ]
            },
            "preview_batch_limit": {
               "title": "Preview Batch Limit",
               "description": "Optional limit on the number of items in the preview␣
↪plots produced during training.",
               "type": "integer"
            },
            "type_hint": {
               "title": "Type Hint",
               "default": "data",
               "enum": [
                  "data"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      }
   }
}
```

Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *data (rastervision.pytorch_learner.learner_config.DataConfig)*
- *log_tensorboard (bool)*
- *model (rastervision.pytorch_learner.classification_learner_config.ClassificationModelConfig)*
- *run_tensorboard (bool)*
- *solver (rastervision.pytorch_learner.learner_config.SolverConfig)*
- *test_mode (bool)*
- *type_hint (Literal['pytorch_chip_classification_backend'])*

field data: *DataConfig* [Required]

field log_tensorboard: bool = True

If True, log events to Tensorboard log files.

field model: *ClassificationModelConfig* [Required]

field run_tensorboard: bool = False

If True, run Tensorboard server pointing at log files.

field solver: *SolverConfig* [Required]

field test_mode: bool = False

This field is passed along to the LearnerConfig which is returned by get_learner_config(). For more info, see the docs forpytorch_learner.learner_config.LearnerConfig.test_mode.

field type_hint: Literal['pytorch_chip_classification_backend'] = 'pytorch_chip_classification_backend'

build(*pipeline*, *tmp_dir*)

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

filter_commands(*commands: List[str]*) → List[str]

Filter out any commands that are not needed or supported.

Parameters
commands (*List[str]*) –

Return type
List[str]

get_bundle_filenames()

Returns the names of files that should be included in a model bundle.

The files are assumed to be in the train/ directory generated by the train command. Note that only the names, not the full paths should be returned.

get_img_channels(*pipeline_cfg:* RVPipelineConfig) → int

Determine img_channels from scenes.

Parameters
pipeline_cfg (RVPipelineConfig) –

Return type
int

get_learner_config(*pipeline*)

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*pipeline: Optional[*RVPipelineConfig*] = None*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

Parameters
pipeline (*Optional[*RVPipelineConfig*]*) –

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises
ConfigError – if field is invalid

9.4.3 pytorch_learner_backend

Classes

PyTorchLearnerBackend	Backend that uses the rastervision.pytorch_learner package to train models.
PyTorchLearnerSampleWriter	

PyTorchLearnerBackend

class PyTorchLearnerBackend

Bases: *Backend*

Backend that uses the rastervision.pytorch_learner package to train models.

__init__(*pipeline_cfg: RVPipelineConfig*, *learner_cfg: LearnerConfig*, *tmp_dir: str*)

Parameters

- **pipeline_cfg** (RVPipelineConfig) –
- **learner_cfg** (LearnerConfig) –
- **tmp_dir** (*str*) –

Methods

__init__(pipeline_cfg, learner_cfg, tmp_dir)	
get_sample_writer()	Returns a SampleWriter for this Backend.
load_model()	Load the model in preparation for one or more prediction calls.
predict_scene(scene, chip_sz[, stride])	Return predictions for an entire scene using the model.
train([source_bundle_uri])	Train a model.

__init__(*pipeline_cfg: RVPipelineConfig*, *learner_cfg: LearnerConfig*, *tmp_dir: str*)

Parameters

- **pipeline_cfg** (RVPipelineConfig) –
- **learner_cfg** (LearnerConfig) –
- **tmp_dir** (*str*) –

get_sample_writer()

Returns a SampleWriter for this Backend.

load_model()

Load the model in preparation for one or more prediction calls.

predict_scene(*scene: Scene*, *chip_sz: int*, *stride: Optional[int] = None*)

Return predictions for an entire scene using the model.

Parameters

- **scene** (Scene) – Scene to run inference on.
- **chip_sz** (*int*) –
- **stride** (*Optional[int]*) –

Returns

Labels object containing predictions

train(*source_bundle_uri=None*)

Train a model.

This should download chips created by the SampleWriter, train the model, and then saving it to disk.

PyTorchLearnerSampleWriter

class PyTorchLearnerSampleWriter

Bases: *SampleWriter*

__init__(*output_uri: str*, *class_config:* ClassConfig, *tmp_dir: str*)

Constructor.

Parameters

- **output_uri** (*str*) – URI of directory where zip file of chips should be placed.
- **class_config** (ClassConfig) – used to convert class ids to names which may be needed for some training data formats.
- **tmp_dir** (*str*) – local directory which is root of any temporary directories that are created.

Methods

__init__(output_uri, class_config, tmp_dir)	Constructor.
get_image_ext(chip)	Decide which format to store the image in.
get_image_path(split_name, sample)	Decide the save location of the image.
write_chip(chip, path)	Save chip as either a PNG image or a numpy array.
write_sample(sample)	Write a single sample to disk.

__init__(*output_uri: str*, *class_config:* ClassConfig, *tmp_dir: str*)

Constructor.

Parameters

- **output_uri** (*str*) – URI of directory where zip file of chips should be placed.
- **class_config** (ClassConfig) – used to convert class ids to names which may be needed for some training data formats.
- **tmp_dir** (*str*) – local directory which is root of any temporary directories that are created.

get_image_ext(*chip: ndarray*) → str

Decide which format to store the image in.

Parameters
chip (*ndarray*) –

Return type
str

get_image_path(*split_name: str*, *sample: DataSample*) → str

Decide the save location of the image. Also, ensure that the target directory exists.

Parameters

- **split_name** (*str*) –
- **sample** (*DataSample*) –

Return type
str

write_chip(*chip: ndarray*, *path: str*) → None

Save chip as either a PNG image or a numpy array.

Parameters

- **chip** (*ndarray*) –
- **path** (*str*) –

Return type
None

write_sample(*sample: DataSample*) → None

Write a single sample to disk.

Parameters
sample (*DataSample*) –

Return type
None

Functions

get_image_ext(chip)	Decide which format to store the image in.
write_chip(chip, path)	Save chip as either a PNG image or a numpy array.

get_image_ext

get_image_ext(*chip: ndarray*) → str

Decide which format to store the image in.

Parameters
chip (*ndarray*) –

Return type
str

write_chip

write_chip(*chip: ndarray*, *path: str*) → None

Save chip as either a PNG image or a numpy array.

Parameters

- **chip** (*ndarray*) –
- **path** (*str*) –

Return type
None

9.4.4 pytorch_learner_backend_config

Configs

PyTorchLearnerBackendConfig	Configure a *PyTorchLearnerBackend*.

PyTorchLearnerBackendConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model PyTorchLearnerBackendConfig

Configure a *PyTorchLearnerBackend*.


```
{
   "title": "PyTorchLearnerBackendConfig",
   "description": "Configure a :class:`.PyTorchLearnerBackend`.",
   "type": "object",
   "properties": {
      "type_hint": {
         "title": "Type Hint",
         "default": "pytorch_learner_backend",
         "enum": [
            "pytorch_learner_backend"
         ],
         "type": "string"
      },
      "model": {
         "$ref": "#/definitions/ModelConfig"
      },
      "solver": {
         "$ref": "#/definitions/SolverConfig"
      },
      "data": {
         "$ref": "#/definitions/DataConfig"
      },
      "log_tensorboard": {
         "title": "Log Tensorboard",
         "description": "If True, log events to Tensorboard log files.",
         "default": true,
         "type": "boolean"
      },
      "run_tensorboard": {
         "title": "Run Tensorboard",
         "description": "If True, run Tensorboard server pointing at log files.",
         "default": false,
         "type": "boolean"
      },
      "test_mode": {
         "title": "Test Mode",
         "description": "This field is passed along to the LearnerConfig which is␣
```


(continues on next page)

(continued from previous page)

```
↪returned by get_learner_config(). For more info, see the docs forpytorch_learner.
↪learner_config.LearnerConfig.test_mode.",
        "default": false,
        "type": "boolean"
     }
  },
  "required": [
     "model",
     "solver",
     "data"
  ],
  "additionalProperties": false,
  "definitions": {
     "Backbone": {
        "title": "Backbone",
        "description": "An enumeration.",
        "enum": [
           "alexnet",
           "densenet121",
           "densenet169",
           "densenet201",
           "densenet161",
           "googlenet",
           "inception_v3",
           "mnasnet0_5",
           "mnasnet0_75",
           "mnasnet1_0",
           "mnasnet1_3",
           "mobilenet_v2",
           "resnet18",
           "resnet34",
           "resnet50",
           "resnet101",
           "resnet152",
           "resnext50_32x4d",
           "resnext101_32x8d",
           "wide_resnet50_2",
           "wide_resnet101_2",
           "shufflenet_v2_x0_5",
           "shufflenet_v2_x1_0",
           "shufflenet_v2_x1_5",
           "shufflenet_v2_x2_0",
           "squeezenet1_0",
           "squeezenet1_1",
           "vgg11",
           "vgg11_bn",
           "vgg13",
           "vgg13_bn",
           "vgg16",
           "vgg16_bn",
           "vgg19_bn",
           "vgg19"
```

(continues on next page)

(continued from previous page)


```
        ]
     },
     "ExternalModuleConfig": {
        "title": "ExternalModuleConfig",
        "description": "Config describing an object to be loaded via Torch Hub.",
        "type": "object",
        "properties": {
           "uri": {
              "title": "Uri",
              "description": "Local uri of a zip file, or local uri of a directory,
↪or remote uri of zip file.",
              "minLength": 1,
              "type": "string"
           },
           "github_repo": {
              "title": "Github Repo",
              "description": "<repo-owner>/<repo-name>[:tag]",
              "pattern": ".+/.+",
              "type": "string"
           },
           "name": {
              "title": "Name",
              "description": "Name of the folder in which to extract/copy the␣
↪definition files.",
              "minLength": 1,
              "type": "string"
           },
           "entrypoint": {
              "title": "Entrypoint",
              "description": "Name of a callable present in hubconf.py. See docs␣
↪for torch.hub for details.",
              "minLength": 1,
              "type": "string"
           },
           "entrypoint_args": {
              "title": "Entrypoint Args",
              "description": "Args to pass to the entrypoint. Must be serializable.
↪",
              "default": [],
              "type": "array",
              "items": {}
           },
           "entrypoint_kwargs": {
              "title": "Entrypoint Kwargs",
              "description": "Keyword args to pass to the entrypoint. Must be␣
↪serializable.",
              "default": {},
              "type": "object"
           },
           "force_reload": {
              "title": "Force Reload",
              "description": "Force reload of module definition.",
```


(continues on next page)

(continued from previous page)

```
                "default": false,
                "type": "boolean"
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "external-module",
                "enum": [
                    "external-module"
                ],
                "type": "string"
            }
        },
        "required": [
            "entrypoint"
        ],
        "additionalProperties": false
    },
    "ModelConfig": {
        "title": "ModelConfig",
        "description": "Config related to models.",
        "type": "object",
        "properties": {
            "backbone": {
                "description": "The torchvision.models backbone to use.",
                "default": "resnet18",
                "allOf": [
                    {
                        "$ref": "#/definitions/Backbone"
                    }
                ]
            },
            "pretrained": {
                "title": "Pretrained",
                "description": "If True, use ImageNet weights. If False, use random␣
↪initialization.",
                "default": true,
                "type": "boolean"
            },
            "init_weights": {
                "title": "Init Weights",
                "description": "URI of PyTorch model weights used to initialize␣
↪model. If set, this supercedes the pretrained option.",
                "type": "string"
            },
            "load_strict": {
                "title": "Load Strict",
                "description": "If True, the keys in the state dict referenced by␣
↪init_weights must match exactly. Setting this to False can be useful if you just␣
↪want to load the backbone of a model.",
                "default": true,
                "type": "boolean"
            },
```

(continues on next page)

(continued from previous page)


```
            "external_def": {
               "title": "External Def",
               "description": "If specified, the model will be built from the␣
↪definition from this external source, using Torch Hub.",
               "allOf": [
                  {
                     "$ref": "#/definitions/ExternalModuleConfig"
                  }
               ]
            },
            "type_hint": {
               "title": "Type Hint",
               "default": "model",
               "enum": [
                  "model"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      },
      "SolverConfig": {
         "title": "SolverConfig",
         "description": "Config related to solver aka optimizer.",
         "type": "object",
         "properties": {
            "lr": {
               "title": "Lr",
               "description": "Learning rate.",
               "default": 0.0001,
               "exclusiveMinimum": 0,
               "type": "number"
            },
            "num_epochs": {
               "title": "Num Epochs",
               "description": "Number of epochs (ie. sweeps through the whole␣
↪training set).",
               "default": 10,
               "exclusiveMinimum": 0,
               "type": "integer"
            },
            "test_num_epochs": {
               "title": "Test Num Epochs",
               "description": "Number of epochs to use in test mode.",
               "default": 2,
               "exclusiveMinimum": 0,
               "type": "integer"
            },
            "test_batch_sz": {
               "title": "Test Batch Sz",
               "description": "Batch size to use in test mode.",
               "default": 4,
```


(continues on next page)

(continued from previous page)

```
            "exclusiveMinimum": 0,
            "type": "integer"
         },
         "overfit_num_steps": {
            "title": "Overfit Num Steps",
            "description": "Number of optimizer steps to use in overfit mode.",
            "default": 1,
            "exclusiveMinimum": 0,
            "type": "integer"
         },
         "sync_interval": {
            "title": "Sync Interval",
            "description": "The interval in epochs for each sync to the cloud.",
            "default": 1,
            "exclusiveMinimum": 0,
            "type": "integer"
         },
         "batch_sz": {
            "title": "Batch Sz",
            "description": "Batch size.",
            "default": 32,
            "exclusiveMinimum": 0,
            "type": "integer"
         },
         "one_cycle": {
            "title": "One Cycle",
            "description": "If True, use triangular LR scheduler with a single␣
↪cycle across all epochs with start and end LR being lr/10 and the peak being lr.",
            "default": true,
            "type": "boolean"
         },
         "multi_stage": {
            "title": "Multi Stage",
            "description": "List of epoch indices at which to divide LR by 10.",
            "default": [],
            "type": "array",
            "items": {}
         },
         "class_loss_weights": {
            "title": "Class Loss Weights",
            "description": "Class weights for weighted loss.",
            "type": "array",
            "items": {
               "type": "number"
            }
         },
         "ignore_class_index": {
            "title": "Ignore Class Index",
            "description": "If specified, this index is ignored when computing␣
↪the loss. See pytorch documentation for nn.CrossEntropyLoss for more details.␣
↪This can also be negative, in which case it is treated as a negative slice index␣
↪i.e. -1 = last index, -2 = second-last index, and so on.",
```

(continues on next page)

(continued from previous page)


```
                "type": "integer"
            },
            "external_loss_def": {
                "title": "External Loss Def",
                "description": "If specified, the loss will be built from the␣
↪definition from this external source, using Torch Hub.",
                "allOf": [
                    {
                        "$ref": "#/definitions/ExternalModuleConfig"
                    }
                ]
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "solver",
                "enum": [
                    "solver"
                ],
                "type": "string"
            }
        },
        "additionalProperties": false
    },
    "PlotOptions": {
        "title": "PlotOptions",
        "description": "Config related to plotting.",
        "type": "object",
        "properties": {
            "transform": {
                "title": "Transform",
                "description": "An Albumentations transform serialized as a dict␣
↪that will be applied to each image before it is plotted. Mainly useful for␣
↪undoing any data transformation that you do not want included in the plot, such␣
↪as normalization. The default value will shift and scale the image so the values␣
↪range from 0.0 to 1.0 which is the expected range for the plotting function. This␣
↪default is useful for cases where the values after normalization are close to␣
↪zero which makes the plot difficult to see.",
                "default": {
                    "__version__": "1.3.0",
                    "transform": {
                        "__class_fullname__": "rastervision.pytorch_learner.utils.
↪utils.MinMaxNormalize",
                        "always_apply": false,
                        "p": 1.0,
                        "min_val": 0.0,
                        "max_val": 1.0,
                        "dtype": 5
                    }
                },
                "type": "object"
            },
            "channel_display_groups": {
```


(continues on next page)

(continued from previous page)

```
                "title": "Channel Display Groups",
                "description": "Groups of image channels to display together as a␣
↪subplot when plotting the data and predictions. Can be a list or tuple of groups␣
↪(e.g. [(0, 1, 2), (3,)]) or a dict containing title-to-group mappings (e.g. {\
↪"RGB\": [0, 1, 2], \"IR\": [3]}), where each group is a list or tuple of channel␣
↪indices and title is a string that will be used as the title of the subplot for␣
↪that group.",
                "anyOf": [
                   {
                      "type": "object",
                      "additionalProperties": {
                         "type": "array",
                         "items": {
                            "type": "integer",
                            "minimum": 0
                         }
                      }
                   },
                   {
                      "type": "array",
                      "items": {
                         "type": "array",
                         "items": {
                            "type": "integer",
                            "minimum": 0
                         }
                      }
                   }
                ]
             },
             "type_hint": {
                "title": "Type Hint",
                "default": "plot_options",
                "enum": [
                   "plot_options"
                ],
                "type": "string"
             }
          },
          "additionalProperties": false
       },
       "DataConfig": {
          "title": "DataConfig",
          "description": "Config related to dataset for training and testing.",
          "type": "object",
          "properties": {
             "class_names": {
                "title": "Class Names",
                "description": "Names of classes.",
                "default": [],
                "type": "array",
                "items": {
```

(continues on next page)

(continued from previous page)

```
                "type": "string"
            }
        },
        "class_colors": {
            "title": "Class Colors",
            "description": "Colors used to display classes. Can be color 3-
↪tuples in list form.",
            "type": "array",
            "items": {
                "anyOf": [
                    {
                        "type": "string"
                    },
                    {
                        "type": "array",
                        "minItems": 3,
                        "maxItems": 3,
                        "items": [
                            {
                                "type": "integer"
                            },
                            {
                                "type": "integer"
                            },
                            {
                                "type": "integer"
                            }
                        ]
                    }
                ]
            }
        },
        "img_channels": {
            "title": "Img Channels",
            "description": "The number of channels of the training images.",
            "exclusiveMinimum": 0,
            "type": "integer"
        },
        "img_sz": {
            "title": "Img Sz",
            "description": "Length of a side of each image in pixels. This is␣
↪the size to transform it to during training, not the size in the raw dataset.",
            "default": 256,
            "exclusiveMinimum": 0,
            "type": "integer"
        },
        "train_sz": {
            "title": "Train Sz",
            "description": "If set, the number of training images to use. If␣
↪fewer images exist, then an exception will be raised.",
            "type": "integer"
        },
```

(continues on next page)

(continued from previous page)

```
            "train_sz_rel": {
               "title": "Train Sz Rel",
               "description": "If set, the proportion of training images to use.",
               "type": "number"
            },
            "num_workers": {
               "title": "Num Workers",
               "description": "Number of workers to use when DataLoader makes␣
↪batches.",
               "default": 4,
               "type": "integer"
            },
            "augmentors": {
               "title": "Augmentors",
               "description": "Names of albumentations augmentors to use for␣
↪training batches. Choices include: ['Blur', 'RandomRotate90', 'HorizontalFlip',
↪'VerticalFlip', 'GaussianBlur', 'GaussNoise', 'RGBShift', 'ToGray'].␣
↪Alternatively, a custom transform can be provided via the aug_transform option.",
               "default": [
                  "RandomRotate90",
                  "HorizontalFlip",
                  "VerticalFlip"
               ],
               "type": "array",
               "items": {
                  "type": "string"
               }
            },
            "base_transform": {
               "title": "Base Transform",
               "description": "An Albumentations transform serialized as a dict␣
↪that will be applied to all datasets: training, validation, and test. This␣
↪transformation is in addition to the resizing due to img_sz. This is useful for,␣
↪for example, applying the same normalization to all datasets.",
               "type": "object"
            },
            "aug_transform": {
               "title": "Aug Transform",
               "description": "An Albumentations transform serialized as a dict␣
↪that will be applied as data augmentation to the training dataset. This transform␣
↪is applied before base_transform. If provided, the augmentors option is ignored.",
               "type": "object"
            },
            "plot_options": {
               "title": "Plot Options",
               "description": "Options to control plotting.",
               "default": {
                  "transform": {
                     "__version__": "1.3.0",
                     "transform": {
                        "__class_fullname__": "rastervision.pytorch_learner.utils.
↪utils.MinMaxNormalize",
```

(continues on next page)

(continued from previous page)


```
                        "always_apply": false,
                        "p": 1.0,
                        "min_val": 0.0,
                        "max_val": 1.0,
                        "dtype": 5
                     }
                  },
                  "channel_display_groups": null,
                  "type_hint": "plot_options"
               },
               "allOf": [
                  {
                     "$ref": "#/definitions/PlotOptions"
                  }
               ]
            },
            "preview_batch_limit": {
               "title": "Preview Batch Limit",
               "description": "Optional limit on the number of items in the preview␣
↪plots produced during training.",
               "type": "integer"
            },
            "type_hint": {
               "title": "Type Hint",
               "default": "data",
               "enum": [
                  "data"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      }
   }
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *data (rastervision.pytorch_learner.learner_config.DataConfig)*
- *log_tensorboard (bool)*
- *model (rastervision.pytorch_learner.learner_config.ModelConfig)*
- *run_tensorboard (bool)*
- *solver (rastervision.pytorch_learner.learner_config.SolverConfig)*
- *test_mode (bool)*
- *type_hint (Literal['pytorch_learner_backend'])*

field data: *DataConfig* [Required]

field log_tensorboard: bool = True

If True, log events to Tensorboard log files.

field model: *ModelConfig* [Required]

field run_tensorboard: bool = False

If True, run Tensorboard server pointing at log files.

field solver: *SolverConfig* [Required]

field test_mode: bool = False

This field is passed along to the LearnerConfig which is returned by get_learner_config(). For more info, see the docs forpytorch_learner.learner_config.LearnerConfig.test_mode.

field type_hint: Literal['pytorch_learner_backend'] = 'pytorch_learner_backend'

build(*pipeline: Optional[*RVPipelineConfig*]*, *tmp_dir: str*)

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

Parameters

- **pipeline** (*Optional[*RVPipelineConfig*]*) –
- **tmp_dir** (*str*) –

filter_commands(*commands: List[str]*) → List[str]

Filter out any commands that are not needed or supported.

Parameters
commands (*List[str]*) –

Return type
List[str]

get_bundle_filenames()

Returns the names of files that should be included in a model bundle.

The files are assumed to be in the train/ directory generated by the train command. Note that only the names, not the full paths should be returned.

get_img_channels(*pipeline_cfg:* RVPipelineConfig) → int

Determine img_channels from scenes.

Parameters
pipeline_cfg (RVPipelineConfig) –

Return type
int

get_learner_config(*pipeline: Optional[*RVPipelineConfig*]*)

Parameters
pipeline (*Optional[*RVPipelineConfig*]*) –

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*pipeline: Optional[RVPipelineConfig] = None*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

Parameters

pipeline (*Optional[RVPipelineConfig]*) –

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

9.4.5 pytorch_object_detection

Classes

PyTorchObjectDetection	
PyTorchObjectDetectionSampleWriter	Writes data in COCO format.

PyTorchObjectDetection

class PyTorchObjectDetection

Bases: *PyTorchLearnerBackend*

__init__(*pipeline_cfg: RVPipelineConfig*, *learner_cfg: LearnerConfig*, *tmp_dir: str*)

Parameters

- **pipeline_cfg** (RVPipelineConfig) –
- **learner_cfg** (LearnerConfig) –

- **tmp_dir** (*str*) –

Methods

__init__(pipeline_cfg, learner_cfg, tmp_dir)	
get_sample_writer()	Returns a SampleWriter for this Backend.
load_model()	Load the model in preparation for one or more prediction calls.
predict_scene(scene, chip_sz[, stride])	Return predictions for an entire scene using the model.
train([source_bundle_uri])	Train a model.

__init__(*pipeline_cfg: RVPipelineConfig*, *learner_cfg: LearnerConfig*, *tmp_dir: str*)

Parameters

- **pipeline_cfg** (RVPipelineConfig) –
- **learner_cfg** (LearnerConfig) –
- **tmp_dir** (*str*) –

get_sample_writer()

Returns a SampleWriter for this Backend.

load_model()

Load the model in preparation for one or more prediction calls.

predict_scene(*scene: Scene*, *chip_sz: int*, *stride: Optional[int] = None*) → *ObjectDetectionLabels*

Return predictions for an entire scene using the model.

Parameters

- **scene** (Scene) – Scene to run inference on.
- **chip_sz** (*int*) –
- **stride** (*Optional[int]*) –

Returns

Labels object containing predictions

Return type

ObjectDetectionLabels

train(*source_bundle_uri=None*)

Train a model.

This should download chips created by the SampleWriter, train the model, and then saving it to disk.

PyTorchObjectDetectionSampleWriter

class PyTorchObjectDetectionSampleWriter

Bases: *PyTorchLearnerSampleWriter*

Writes data in COCO format.

__init__(*output_uri: str*, *class_config:* ClassConfig, *tmp_dir: str*)

Constructor.

Parameters

- **output_uri** (*str*) – URI of directory where zip file of chips should be placed.
- **class_config** (ClassConfig) – used to convert class ids to names which may be needed for some training data formats.
- **tmp_dir** (*str*) – local directory which is root of any temporary directories that are created.

Methods

__init__(output_uri, class_config, tmp_dir)	Constructor.
get_image_ext(chip)	Decide which format to store the image in.
get_image_path(split_name, sample)	Decide the save location of the image.
update_coco_data(split_name, sample, img_path)	
write_chip(chip, path)	Save chip as either a PNG image or a numpy array.
write_sample(sample)	This writes a training or validation sample to (train|valid)/img/{scene_id}-{ind}.png and updates some COCO data structures.

__init__(*output_uri: str*, *class_config:* ClassConfig, *tmp_dir: str*)

Constructor.

Parameters

- **output_uri** (*str*) – URI of directory where zip file of chips should be placed.
- **class_config** (ClassConfig) – used to convert class ids to names which may be needed for some training data formats.
- **tmp_dir** (*str*) – local directory which is root of any temporary directories that are created.

get_image_ext(*chip: ndarray*) → str

Decide which format to store the image in.

Parameters
chip (*ndarray*) –

Return type
str

get_image_path(*split_name: str*, *sample: DataSample*) → str

Decide the save location of the image. Also, ensure that the target directory exists.

Parameters

- **split_name** (*str*) –

- **sample** (*DataSample*) –

Return type
str

update_coco_data(*split_name: str*, *sample: DataSample*, *img_path: str*)

Parameters

- **split_name** (*str*) –
- **sample** (*DataSample*) –
- **img_path** (*str*) –

write_chip(*chip: ndarray*, *path: str*) → None

Save chip as either a PNG image or a numpy array.

Parameters

- **chip** (*ndarray*) –
- **path** (*str*) –

Return type
None

write_sample(*sample: DataSample*)

This writes a training or validation sample to (train|valid)/img/{scene_id}-{ind}.png and updates some COCO data structures.

Parameters
sample (*DataSample*) –

9.4.6 pytorch_object_detection_config

Configs

PyTorchObjectDetectionConfig	Configure a *PyTorchObjectDetection* backend.

PyTorchObjectDetectionConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model PyTorchObjectDetectionConfig

Configure a *PyTorchObjectDetection* backend.


```
{
   "title": "PyTorchObjectDetectionConfig",
   "description": "Configure a :class:`.PyTorchObjectDetection` backend.",
   "type": "object",
   "properties": {
      "type_hint": {
         "title": "Type Hint",
         "default": "pytorch_object_detection_backend",
```


(continues on next page)

(continued from previous page)


```
            "enum": [
               "pytorch_object_detection_backend"
            ],
            "type": "string"
         },
         "model": {
            "$ref": "#/definitions/ObjectDetectionModelConfig"
         },
         "solver": {
            "$ref": "#/definitions/SolverConfig"
         },
         "data": {
            "$ref": "#/definitions/DataConfig"
         },
         "log_tensorboard": {
            "title": "Log Tensorboard",
            "description": "If True, log events to Tensorboard log files.",
            "default": true,
            "type": "boolean"
         },
         "run_tensorboard": {
            "title": "Run Tensorboard",
            "description": "If True, run Tensorboard server pointing at log files.",
            "default": false,
            "type": "boolean"
         },
         "test_mode": {
            "title": "Test Mode",
            "description": "This field is passed along to the LearnerConfig which is␣
↪returned by get_learner_config(). For more info, see the docs forpytorch_learner.
↪learner_config.LearnerConfig.test_mode.",
            "default": false,
            "type": "boolean"
         }
      },
      "required": [
         "model",
         "solver",
         "data"
      ],
      "additionalProperties": false,
      "definitions": {
         "Backbone": {
            "title": "Backbone",
            "description": "An enumeration.",
            "enum": [
               "alexnet",
               "densenet121",
               "densenet169",
               "densenet201",
               "densenet161",
               "googlenet",
```


(continues on next page)

(continued from previous page)

```
            "inception_v3",
            "mnasnet0_5",
            "mnasnet0_75",
            "mnasnet1_0",
            "mnasnet1_3",
            "mobilenet_v2",
            "resnet18",
            "resnet34",
            "resnet50",
            "resnet101",
            "resnet152",
            "resnext50_32x4d",
            "resnext101_32x8d",
            "wide_resnet50_2",
            "wide_resnet101_2",
            "shufflenet_v2_x0_5",
            "shufflenet_v2_x1_0",
            "shufflenet_v2_x1_5",
            "shufflenet_v2_x2_0",
            "squeezenet1_0",
            "squeezenet1_1",
            "vgg11",
            "vgg11_bn",
            "vgg13",
            "vgg13_bn",
            "vgg16",
            "vgg16_bn",
            "vgg19_bn",
            "vgg19"
         ]
      },
      "ExternalModuleConfig": {
         "title": "ExternalModuleConfig",
         "description": "Config describing an object to be loaded via Torch Hub.",
         "type": "object",
         "properties": {
            "uri": {
               "title": "Uri",
               "description": "Local uri of a zip file, or local uri of a directory,
↪or remote uri of zip file.",
               "minLength": 1,
               "type": "string"
            },
            "github_repo": {
               "title": "Github Repo",
               "description": "<repo-owner>/<repo-name>[:tag]",
               "pattern": ".+/.+",
               "type": "string"
            },
            "name": {
               "title": "Name",
               "description": "Name of the folder in which to extract/copy the␣
```

(continues on next page)

(continued from previous page)

```
↪definition files.",
                "minLength": 1,
                "type": "string"
            },
            "entrypoint": {
                "title": "Entrypoint",
                "description": "Name of a callable present in hubconf.py. See docs␣
↪for torch.hub for details.",
                "minLength": 1,
                "type": "string"
            },
            "entrypoint_args": {
                "title": "Entrypoint Args",
                "description": "Args to pass to the entrypoint. Must be serializable.
↪",
                "default": [],
                "type": "array",
                "items": {}
            },
            "entrypoint_kwargs": {
                "title": "Entrypoint Kwargs",
                "description": "Keyword args to pass to the entrypoint. Must be␣
↪serializable.",
                "default": {},
                "type": "object"
            },
            "force_reload": {
                "title": "Force Reload",
                "description": "Force reload of module definition.",
                "default": false,
                "type": "boolean"
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "external-module",
                "enum": [
                    "external-module"
                ],
                "type": "string"
            }
        },
        "required": [
            "entrypoint"
        ],
        "additionalProperties": false
    },
    "ObjectDetectionModelConfig": {
        "title": "ObjectDetectionModelConfig",
        "description": "Configure an object detection model.",
        "type": "object",
        "properties": {
            "backbone": {
```

(continues on next page)

(continued from previous page)


```
                "description": "The torchvision.models backbone to use, which must␣
↪be in the resnet* family.",
                "default": "resnet50",
                "allOf": [
                   {
                      "$ref": "#/definitions/Backbone"
                   }
                ]
             },
             "pretrained": {
                "title": "Pretrained",
                "description": "If True, use ImageNet weights. If False, use random␣
↪initialization.",
                "default": true,
                "type": "boolean"
             },
             "init_weights": {
                "title": "Init Weights",
                "description": "URI of PyTorch model weights used to initialize␣
↪model. If set, this supercedes the pretrained option.",
                "type": "string"
             },
             "load_strict": {
                "title": "Load Strict",
                "description": "If True, the keys in the state dict referenced by␣
↪init_weights must match exactly. Setting this to False can be useful if you just␣
↪want to load the backbone of a model.",
                "default": true,
                "type": "boolean"
             },
             "external_def": {
                "title": "External Def",
                "description": "If specified, the model will be built from the␣
↪definition from this external source, using Torch Hub.",
                "allOf": [
                   {
                      "$ref": "#/definitions/ExternalModuleConfig"
                   }
                ]
             },
             "type_hint": {
                "title": "Type Hint",
                "default": "object_detection_model",
                "enum": [
                   "object_detection_model"
                ],
                "type": "string"
             }
          },
          "additionalProperties": false
       },
       "SolverConfig": {
```


(continues on next page)

(continued from previous page)


```
        "title": "SolverConfig",
        "description": "Config related to solver aka optimizer.",
        "type": "object",
        "properties": {
           "lr": {
              "title": "Lr",
              "description": "Learning rate.",
              "default": 0.0001,
              "exclusiveMinimum": 0,
              "type": "number"
           },
           "num_epochs": {
              "title": "Num Epochs",
              "description": "Number of epochs (ie. sweeps through the whole␣
↪training set).",
              "default": 10,
              "exclusiveMinimum": 0,
              "type": "integer"
           },
           "test_num_epochs": {
              "title": "Test Num Epochs",
              "description": "Number of epochs to use in test mode.",
              "default": 2,
              "exclusiveMinimum": 0,
              "type": "integer"
           },
           "test_batch_sz": {
              "title": "Test Batch Sz",
              "description": "Batch size to use in test mode.",
              "default": 4,
              "exclusiveMinimum": 0,
              "type": "integer"
           },
           "overfit_num_steps": {
              "title": "Overfit Num Steps",
              "description": "Number of optimizer steps to use in overfit mode.",
              "default": 1,
              "exclusiveMinimum": 0,
              "type": "integer"
           },
           "sync_interval": {
              "title": "Sync Interval",
              "description": "The interval in epochs for each sync to the cloud.",
              "default": 1,
              "exclusiveMinimum": 0,
              "type": "integer"
           },
           "batch_sz": {
              "title": "Batch Sz",
              "description": "Batch size.",
              "default": 32,
              "exclusiveMinimum": 0,
```


(continues on next page)

(continued from previous page)

```
            "type": "integer"
         },
         "one_cycle": {
            "title": "One Cycle",
            "description": "If True, use triangular LR scheduler with a single␣
↪cycle across all epochs with start and end LR being lr/10 and the peak being lr.",
            "default": true,
            "type": "boolean"
         },
         "multi_stage": {
            "title": "Multi Stage",
            "description": "List of epoch indices at which to divide LR by 10.",
            "default": [],
            "type": "array",
            "items": {}
         },
         "class_loss_weights": {
            "title": "Class Loss Weights",
            "description": "Class weights for weighted loss.",
            "type": "array",
            "items": {
               "type": "number"
            }
         },
         "ignore_class_index": {
            "title": "Ignore Class Index",
            "description": "If specified, this index is ignored when computing␣
↪the loss. See pytorch documentation for nn.CrossEntropyLoss for more details.␣
↪This can also be negative, in which case it is treated as a negative slice index␣
↪i.e. -1 = last index, -2 = second-last index, and so on.",
            "type": "integer"
         },
         "external_loss_def": {
            "title": "External Loss Def",
            "description": "If specified, the loss will be built from the␣
↪definition from this external source, using Torch Hub.",
            "allOf": [
               {
                  "$ref": "#/definitions/ExternalModuleConfig"
               }
            ]
         },
         "type_hint": {
            "title": "Type Hint",
            "default": "solver",
            "enum": [
               "solver"
            ],
            "type": "string"
         }
      },
      "additionalProperties": false
```

(continues on next page)

(continued from previous page)


```
      },
      "PlotOptions": {
         "title": "PlotOptions",
         "description": "Config related to plotting.",
         "type": "object",
         "properties": {
            "transform": {
               "title": "Transform",
               "description": "An Albumentations transform serialized as a dict␣
↪that will be applied to each image before it is plotted. Mainly useful for␣
↪undoing any data transformation that you do not want included in the plot, such␣
↪as normalization. The default value will shift and scale the image so the values␣
↪range from 0.0 to 1.0 which is the expected range for the plotting function. This␣
↪default is useful for cases where the values after normalization are close to␣
↪zero which makes the plot difficult to see.",
               "default": {
                  "__version__": "1.3.0",
                  "transform": {
                     "__class_fullname__": "rastervision.pytorch_learner.utils.
↪utils.MinMaxNormalize",
                     "always_apply": false,
                     "p": 1.0,
                     "min_val": 0.0,
                     "max_val": 1.0,
                     "dtype": 5
                  }
               },
               "type": "object"
            },
            "channel_display_groups": {
               "title": "Channel Display Groups",
               "description": "Groups of image channels to display together as a␣
↪subplot when plotting the data and predictions. Can be a list or tuple of groups␣
↪(e.g. [(0, 1, 2), (3,)]) or a dict containing title-to-group mappings (e.g. {\
↪"RGB\": [0, 1, 2], \"IR\": [3]}), where each group is a list or tuple of channel␣
↪indices and title is a string that will be used as the title of the subplot for␣
↪that group.",
               "anyOf": [
                  {
                     "type": "object",
                     "additionalProperties": {
                        "type": "array",
                        "items": {
                           "type": "integer",
                           "minimum": 0
                        }
                     }
                  },
                  {
                     "type": "array",
                     "items": {
                        "type": "array",
```


(continues on next page)

(continued from previous page)

```
                        "items": {
                            "type": "integer",
                            "minimum": 0
                        }
                    }
                }
            ]
        },
        "type_hint": {
            "title": "Type Hint",
            "default": "plot_options",
            "enum": [
                "plot_options"
            ],
            "type": "string"
        }
    },
    "additionalProperties": false
},
"DataConfig": {
    "title": "DataConfig",
    "description": "Config related to dataset for training and testing.",
    "type": "object",
    "properties": {
        "class_names": {
            "title": "Class Names",
            "description": "Names of classes.",
            "default": [],
            "type": "array",
            "items": {
                "type": "string"
            }
        },
        "class_colors": {
            "title": "Class Colors",
            "description": "Colors used to display classes. Can be color 3-
↪tuples in list form.",
            "type": "array",
            "items": {
                "anyOf": [
                    {
                        "type": "string"
                    },
                    {
                        "type": "array",
                        "minItems": 3,
                        "maxItems": 3,
                        "items": [
                            {
                                "type": "integer"
                            },
                            {
```

(continues on next page)

(continued from previous page)

```
                                "type": "integer"
                            },
                            {
                                "type": "integer"
                            }
                        ]
                    }
                ]
            }
        },
        "img_channels": {
            "title": "Img Channels",
            "description": "The number of channels of the training images.",
            "exclusiveMinimum": 0,
            "type": "integer"
        },
        "img_sz": {
            "title": "Img Sz",
            "description": "Length of a side of each image in pixels. This is␣
↪the size to transform it to during training, not the size in the raw dataset.",
            "default": 256,
            "exclusiveMinimum": 0,
            "type": "integer"
        },
        "train_sz": {
            "title": "Train Sz",
            "description": "If set, the number of training images to use. If␣
↪fewer images exist, then an exception will be raised.",
            "type": "integer"
        },
        "train_sz_rel": {
            "title": "Train Sz Rel",
            "description": "If set, the proportion of training images to use.",
            "type": "number"
        },
        "num_workers": {
            "title": "Num Workers",
            "description": "Number of workers to use when DataLoader makes␣
↪batches.",
            "default": 4,
            "type": "integer"
        },
        "augmentors": {
            "title": "Augmentors",
            "description": "Names of albumentations augmentors to use for␣
↪training batches. Choices include: ['Blur', 'RandomRotate90', 'HorizontalFlip',
↪'VerticalFlip', 'GaussianBlur', 'GaussNoise', 'RGBShift', 'ToGray'].␣
↪Alternatively, a custom transform can be provided via the aug_transform option.",
            "default": [
                "RandomRotate90",
                "HorizontalFlip",
                "VerticalFlip"
```

(continues on next page)

(continued from previous page)


```
                ],
                "type": "array",
                "items": {
                   "type": "string"
                }
             },
             "base_transform": {
                "title": "Base Transform",
                "description": "An Albumentations transform serialized as a dict␣
↪that will be applied to all datasets: training, validation, and test. This␣
↪transformation is in addition to the resizing due to img_sz. This is useful for,␣
↪for example, applying the same normalization to all datasets.",
                "type": "object"
             },
             "aug_transform": {
                "title": "Aug Transform",
                "description": "An Albumentations transform serialized as a dict␣
↪that will be applied as data augmentation to the training dataset. This transform␣
↪is applied before base_transform. If provided, the augmentors option is ignored.",
                "type": "object"
             },
             "plot_options": {
                "title": "Plot Options",
                "description": "Options to control plotting.",
                "default": {
                   "transform": {
                      "__version__": "1.3.0",
                      "transform": {
                         "__class_fullname__": "rastervision.pytorch_learner.utils.
↪utils.MinMaxNormalize",
                         "always_apply": false,
                         "p": 1.0,
                         "min_val": 0.0,
                         "max_val": 1.0,
                         "dtype": 5
                      }
                   },
                   "channel_display_groups": null,
                   "type_hint": "plot_options"
                },
                "allOf": [
                   {
                      "$ref": "#/definitions/PlotOptions"
                   }
                ]
             },
             "preview_batch_limit": {
                "title": "Preview Batch Limit",
                "description": "Optional limit on the number of items in the preview␣
↪plots produced during training.",
                "type": "integer"
             },
```


(continues on next page)

(continued from previous page)

```
            "type_hint": {
               "title": "Type Hint",
               "default": "data",
               "enum": [
                  "data"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      }
   }
}
```

Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *data (rastervision.pytorch_learner.learner_config.DataConfig)*
- *log_tensorboard (bool)*
- *model (rastervision.pytorch_learner.object_detection_learner_config.ObjectDetectionModelConfig)*
- *run_tensorboard (bool)*
- *solver (rastervision.pytorch_learner.learner_config.SolverConfig)*
- *test_mode (bool)*
- *type_hint (Literal['pytorch_object_detection_backend'])*

field data: *DataConfig* [Required]

field log_tensorboard: bool = True

If True, log events to Tensorboard log files.

field model: *ObjectDetectionModelConfig* [Required]

field run_tensorboard: bool = False

If True, run Tensorboard server pointing at log files.

field solver: *SolverConfig* [Required]

field test_mode: bool = False

This field is passed along to the LearnerConfig which is returned by get_learner_config(). For more info, see the docs forpytorch_learner.learner_config.LearnerConfig.test_mode.

field type_hint: Literal['pytorch_object_detection_backend'] = 'pytorch_object_detection_backend'

build(*pipeline*, *tmp_dir*)

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

filter_commands(*commands: List[str]*) → List[str]

Filter out any commands that are not needed or supported.

Parameters
commands (*List[str]*) –

Return type
List[str]

get_bundle_filenames()

Returns the names of files that should be included in a model bundle.

The files are assumed to be in the train/ directory generated by the train command. Note that only the names, not the full paths should be returned.

get_img_channels(*pipeline_cfg: RVPipelineConfig*) → int

Determine img_channels from scenes.

Parameters
pipeline_cfg (RVPipelineConfig) –

Return type
int

get_learner_config(*pipeline*)

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*pipeline: Optional[RVPipelineConfig] = None*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

Parameters
pipeline (*Optional[RVPipelineConfig]*) –

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises
ConfigError – if field is invalid

9.4.7 pytorch_semantic_segmentation

Classes

PyTorchSemanticSegmentation
PyTorchSemanticSegmentationSampleWriter

PyTorchSemanticSegmentation

class PyTorchSemanticSegmentation

Bases: *PyTorchLearnerBackend*

__init__(*pipeline_cfg:* RVPipelineConfig, *learner_cfg:* LearnerConfig, *tmp_dir:* *str*)

Parameters

- **pipeline_cfg** (RVPipelineConfig) –
- **learner_cfg** (LearnerConfig) –
- **tmp_dir** (*str*) –

Methods

__init__(pipeline_cfg, learner_cfg, tmp_dir)

get_sample_writer()	Returns a SampleWriter for this Backend.
load_model()	Load the model in preparation for one or more prediction calls.
predict_scene(scene, chip_sz[, stride, crop_sz])	Return predictions for an entire scene using the model.
train([source_bundle_uri])	Train a model.

__init__(*pipeline_cfg:* RVPipelineConfig, *learner_cfg:* LearnerConfig, *tmp_dir:* *str*)

Parameters

- **pipeline_cfg** (RVPipelineConfig) –
- **learner_cfg** (LearnerConfig) –
- **tmp_dir** (*str*) –

get_sample_writer()

Returns a SampleWriter for this Backend.

load_model()

Load the model in preparation for one or more prediction calls.

predict_scene(*scene:* Scene, *chip_sz:* *int*, *stride:* *Optional[int]* = *None*, *crop_sz:* *Optional[int]* = *None*) → *SemanticSegmentationLabels*

Return predictions for an entire scene using the model.

Parameters

- **scene** (Scene) – Scene to run inference on.
- **chip_sz** (*int*) –
- **stride** (*Optional[int]*) –
- **crop_sz** (*Optional[int]*) –

Returns

Labels object containing predictions

Return type

SemanticSegmentationLabels

train(*source_bundle_uri=None*)

Train a model.

This should download chips created by the SampleWriter, train the model, and then saving it to disk.

PyTorchSemanticSegmentationSampleWriter

class PyTorchSemanticSegmentationSampleWriter

Bases: *PyTorchLearnerSampleWriter*

__init__(*output_uri:* *str*, *class_config:* ClassConfig, *tmp_dir:* *str*)

Constructor.

Parameters

- **output_uri** (*str*) – URI of directory where zip file of chips should be placed.
- **class_config** (ClassConfig) – used to convert class ids to names which may be needed for some training data formats.
- **tmp_dir** (*str*) – local directory which is root of any temporary directories that are created.

Methods

__init__(output_uri, class_config, tmp_dir)	Constructor.
get_image_ext(chip)	Decide which format to store the image in.
get_image_path(split_name, sample)	Decide the save location of the image.
get_label_path(split_name, sample, label_arr)	
write_chip(chip, path)	Save chip as either a PNG image or a numpy array.
write_sample(sample)	This writes a training or validation sample to (train|valid)/img/{scene_id}-{ind}.png and (train|valid)/labels/{scene_id}-{ind}.png

__init__(*output_uri: str*, *class_config:* ClassConfig, *tmp_dir: str*)

Constructor.

Parameters

- **output_uri** (*str*) – URI of directory where zip file of chips should be placed.
- **class_config** (ClassConfig) – used to convert class ids to names which may be needed for some training data formats.
- **tmp_dir** (*str*) – local directory which is root of any temporary directories that are created.

get_image_ext(*chip: ndarray*) → str

Decide which format to store the image in.

Parameters
chip (*ndarray*) –

Return type
str

get_image_path(*split_name: str*, *sample: DataSample*) → str

Decide the save location of the image. Also, ensure that the target directory exists.

Parameters

- **split_name** (*str*) –
- **sample** (*DataSample*) –

Return type
str

get_label_path(*split_name: str*, *sample: DataSample*, *label_arr: ndarray*) → str

Parameters

- **split_name** (*str*) –
- **sample** (*DataSample*) –
- **label_arr** (*ndarray*) –

Return type
str

write_chip(*chip: ndarray*, *path: str*) → None

Save chip as either a PNG image or a numpy array.

Parameters

- **chip** (*ndarray*) –
- **path** (*str*) –

Return type
None

write_sample(*sample: DataSample*)

This writes a training or validation sample to (train|valid)/img/{scene_id}-{ind}.png and (train|valid)/labels/{scene_id}-{ind}.png

Parameters
sample (*DataSample*) –

9.4.8 pytorch_semantic_segmentation_config

Configs

PyTorchSemanticSegmentationConfig	Configure a *PyTorchSemanticSegmentation* backend.

PyTorchSemanticSegmentationConfig

Note: All Configs are derived from *rastervision.pipeline.config.Config*, which itself is a pydantic Model.

pydantic model PyTorchSemanticSegmentationConfig

Configure a *PyTorchSemanticSegmentation* backend.


```
{
   "title": "PyTorchSemanticSegmentationConfig",
   "description": "Configure a :class:`.PyTorchSemanticSegmentation` backend.",
   "type": "object",
   "properties": {
      "type_hint": {
         "title": "Type Hint",
         "default": "pytorch_semantic_segmentation_backend",
         "enum": [
            "pytorch_semantic_segmentation_backend"
         ],
         "type": "string"
      },
      "model": {
         "$ref": "#/definitions/SemanticSegmentationModelConfig"
      },
      "solver": {
         "$ref": "#/definitions/SolverConfig"
      },
      "data": {
         "$ref": "#/definitions/DataConfig"
      },
      "log_tensorboard": {
         "title": "Log Tensorboard",
         "description": "If True, log events to Tensorboard log files.",
         "default": true,
         "type": "boolean"
      },
      "run_tensorboard": {
         "title": "Run Tensorboard",
         "description": "If True, run Tensorboard server pointing at log files.",
         "default": false,
         "type": "boolean"
      },
      "test_mode": {
         "title": "Test Mode",
```


(continues on next page)

(continued from previous page)

```
        "description": "This field is passed along to the LearnerConfig which is␣
↪returned by get_learner_config(). For more info, see the docs forpytorch_learner.
↪learner_config.LearnerConfig.test_mode.",
        "default": false,
        "type": "boolean"
     }
  },
  "required": [
     "model",
     "solver",
     "data"
  ],
  "additionalProperties": false,
  "definitions": {
     "Backbone": {
        "title": "Backbone",
        "description": "An enumeration.",
        "enum": [
           "alexnet",
           "densenet121",
           "densenet169",
           "densenet201",
           "densenet161",
           "googlenet",
           "inception_v3",
           "mnasnet0_5",
           "mnasnet0_75",
           "mnasnet1_0",
           "mnasnet1_3",
           "mobilenet_v2",
           "resnet18",
           "resnet34",
           "resnet50",
           "resnet101",
           "resnet152",
           "resnext50_32x4d",
           "resnext101_32x8d",
           "wide_resnet50_2",
           "wide_resnet101_2",
           "shufflenet_v2_x0_5",
           "shufflenet_v2_x1_0",
           "shufflenet_v2_x1_5",
           "shufflenet_v2_x2_0",
           "squeezenet1_0",
           "squeezenet1_1",
           "vgg11",
           "vgg11_bn",
           "vgg13",
           "vgg13_bn",
           "vgg16",
           "vgg16_bn",
           "vgg19_bn",
```

(continues on next page)

(continued from previous page)


```
            "vgg19"
         ]
      },
      "ExternalModuleConfig": {
         "title": "ExternalModuleConfig",
         "description": "Config describing an object to be loaded via Torch Hub.",
         "type": "object",
         "properties": {
            "uri": {
               "title": "Uri",
               "description": "Local uri of a zip file, or local uri of a directory,
↪or remote uri of zip file.",
               "minLength": 1,
               "type": "string"
            },
            "github_repo": {
               "title": "Github Repo",
               "description": "<repo-owner>/<repo-name>[:tag]",
               "pattern": ".+/.+",
               "type": "string"
            },
            "name": {
               "title": "Name",
               "description": "Name of the folder in which to extract/copy the␣
↪definition files.",
               "minLength": 1,
               "type": "string"
            },
            "entrypoint": {
               "title": "Entrypoint",
               "description": "Name of a callable present in hubconf.py. See docs␣
↪for torch.hub for details.",
               "minLength": 1,
               "type": "string"
            },
            "entrypoint_args": {
               "title": "Entrypoint Args",
               "description": "Args to pass to the entrypoint. Must be serializable.
↪",
               "default": [],
               "type": "array",
               "items": {}
            },
            "entrypoint_kwargs": {
               "title": "Entrypoint Kwargs",
               "description": "Keyword args to pass to the entrypoint. Must be␣
↪serializable.",
               "default": {},
               "type": "object"
            },
            "force_reload": {
               "title": "Force Reload",
```


(continues on next page)

(continued from previous page)


```
                "description": "Force reload of module definition.",
                "default": false,
                "type": "boolean"
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "external-module",
                "enum": [
                    "external-module"
                ],
                "type": "string"
            }
        },
        "required": [
            "entrypoint"
        ],
        "additionalProperties": false
    },
    "SemanticSegmentationModelConfig": {
        "title": "SemanticSegmentationModelConfig",
        "description": "Configure a semantic segmentation model.",
        "type": "object",
        "properties": {
            "backbone": {
                "description": "The torchvision.models backbone to use. At the␣
↪moment only resnet50 or resnet101 will work.",
                "default": "resnet50",
                "allOf": [
                    {
                        "$ref": "#/definitions/Backbone"
                    }
                ]
            },
            "pretrained": {
                "title": "Pretrained",
                "description": "If True, use ImageNet weights. If False, use random␣
↪initialization.",
                "default": true,
                "type": "boolean"
            },
            "init_weights": {
                "title": "Init Weights",
                "description": "URI of PyTorch model weights used to initialize␣
↪model. If set, this supercedes the pretrained option.",
                "type": "string"
            },
            "load_strict": {
                "title": "Load Strict",
                "description": "If True, the keys in the state dict referenced by␣
↪init_weights must match exactly. Setting this to False can be useful if you just␣
↪want to load the backbone of a model.",
                "default": true,
```


(continues on next page)

(continued from previous page)


```
                "type": "boolean"
            },
            "external_def": {
                "title": "External Def",
                "description": "If specified, the model will be built from the␣
↪definition from this external source, using Torch Hub.",
                "allOf": [
                    {
                        "$ref": "#/definitions/ExternalModuleConfig"
                    }
                ]
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "semantic_segmentation_model",
                "enum": [
                    "semantic_segmentation_model"
                ],
                "type": "string"
            }
        },
        "additionalProperties": false
    },
    "SolverConfig": {
        "title": "SolverConfig",
        "description": "Config related to solver aka optimizer.",
        "type": "object",
        "properties": {
            "lr": {
                "title": "Lr",
                "description": "Learning rate.",
                "default": 0.0001,
                "exclusiveMinimum": 0,
                "type": "number"
            },
            "num_epochs": {
                "title": "Num Epochs",
                "description": "Number of epochs (ie. sweeps through the whole␣
↪training set).",
                "default": 10,
                "exclusiveMinimum": 0,
                "type": "integer"
            },
            "test_num_epochs": {
                "title": "Test Num Epochs",
                "description": "Number of epochs to use in test mode.",
                "default": 2,
                "exclusiveMinimum": 0,
                "type": "integer"
            },
            "test_batch_sz": {
                "title": "Test Batch Sz",
```


(continues on next page)

(continued from previous page)

```
            "description": "Batch size to use in test mode.",
            "default": 4,
            "exclusiveMinimum": 0,
            "type": "integer"
         },
         "overfit_num_steps": {
            "title": "Overfit Num Steps",
            "description": "Number of optimizer steps to use in overfit mode.",
            "default": 1,
            "exclusiveMinimum": 0,
            "type": "integer"
         },
         "sync_interval": {
            "title": "Sync Interval",
            "description": "The interval in epochs for each sync to the cloud.",
            "default": 1,
            "exclusiveMinimum": 0,
            "type": "integer"
         },
         "batch_sz": {
            "title": "Batch Sz",
            "description": "Batch size.",
            "default": 32,
            "exclusiveMinimum": 0,
            "type": "integer"
         },
         "one_cycle": {
            "title": "One Cycle",
            "description": "If True, use triangular LR scheduler with a single␣
↪cycle across all epochs with start and end LR being lr/10 and the peak being lr.",
            "default": true,
            "type": "boolean"
         },
         "multi_stage": {
            "title": "Multi Stage",
            "description": "List of epoch indices at which to divide LR by 10.",
            "default": [],
            "type": "array",
            "items": {}
         },
         "class_loss_weights": {
            "title": "Class Loss Weights",
            "description": "Class weights for weighted loss.",
            "type": "array",
            "items": {
               "type": "number"
            }
         },
         "ignore_class_index": {
            "title": "Ignore Class Index",
            "description": "If specified, this index is ignored when computing␣
↪the loss. See pytorch documentation for nn.CrossEntropyLoss for more details.␣
```

(continues on next page)

(continued from previous page)

```
↪This can also be negative, in which case it is treated as a negative slice index␣
↪i.e. -1 = last index, -2 = second-last index, and so on.",
                "type": "integer"
            },
            "external_loss_def": {
                "title": "External Loss Def",
                "description": "If specified, the loss will be built from the␣
↪definition from this external source, using Torch Hub.",
                "allOf": [
                    {
                        "$ref": "#/definitions/ExternalModuleConfig"
                    }
                ]
            },
            "type_hint": {
                "title": "Type Hint",
                "default": "solver",
                "enum": [
                    "solver"
                ],
                "type": "string"
            }
        },
        "additionalProperties": false
    },
    "PlotOptions": {
        "title": "PlotOptions",
        "description": "Config related to plotting.",
        "type": "object",
        "properties": {
            "transform": {
                "title": "Transform",
                "description": "An Albumentations transform serialized as a dict␣
↪that will be applied to each image before it is plotted. Mainly useful for␣
↪undoing any data transformation that you do not want included in the plot, such␣
↪as normalization. The default value will shift and scale the image so the values␣
↪range from 0.0 to 1.0 which is the expected range for the plotting function. This␣
↪default is useful for cases where the values after normalization are close to␣
↪zero which makes the plot difficult to see.",
                "default": {
                    "__version__": "1.3.0",
                    "transform": {
                        "__class_fullname__": "rastervision.pytorch_learner.utils.
↪utils.MinMaxNormalize",
                        "always_apply": false,
                        "p": 1.0,
                        "min_val": 0.0,
                        "max_val": 1.0,
                        "dtype": 5
                    }
                },
                "type": "object"
```

(continues on next page)

(continued from previous page)

```
            },
            "channel_display_groups": {
               "title": "Channel Display Groups",
               "description": "Groups of image channels to display together as a␣
↪subplot when plotting the data and predictions. Can be a list or tuple of groups␣
↪(e.g. [(0, 1, 2), (3,)]) or a dict containing title-to-group mappings (e.g. {\
↪"RGB\": [0, 1, 2], \"IR\": [3]}), where each group is a list or tuple of channel␣
↪indices and title is a string that will be used as the title of the subplot for␣
↪that group.",
               "anyOf": [
                  {
                     "type": "object",
                     "additionalProperties": {
                        "type": "array",
                        "items": {
                           "type": "integer",
                           "minimum": 0
                        }
                     }
                  },
                  {
                     "type": "array",
                     "items": {
                        "type": "array",
                        "items": {
                           "type": "integer",
                           "minimum": 0
                        }
                     }
                  }
               ]
            },
            "type_hint": {
               "title": "Type Hint",
               "default": "plot_options",
               "enum": [
                  "plot_options"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      },
      "DataConfig": {
         "title": "DataConfig",
         "description": "Config related to dataset for training and testing.",
         "type": "object",
         "properties": {
            "class_names": {
               "title": "Class Names",
               "description": "Names of classes.",
               "default": [],
```

(continues on next page)

(continued from previous page)


```
                "type": "array",
                "items": {
                   "type": "string"
                }
             },
             "class_colors": {
                "title": "Class Colors",
                "description": "Colors used to display classes. Can be color 3-
↪tuples in list form.",
                "type": "array",
                "items": {
                   "anyOf": [
                      {
                         "type": "string"
                      },
                      {
                         "type": "array",
                         "minItems": 3,
                         "maxItems": 3,
                         "items": [
                            {
                               "type": "integer"
                            },
                            {
                               "type": "integer"
                            },
                            {
                               "type": "integer"
                            }
                         ]
                      }
                   ]
                }
             },
             "img_channels": {
                "title": "Img Channels",
                "description": "The number of channels of the training images.",
                "exclusiveMinimum": 0,
                "type": "integer"
             },
             "img_sz": {
                "title": "Img Sz",
                "description": "Length of a side of each image in pixels. This is␣
↪the size to transform it to during training, not the size in the raw dataset.",
                "default": 256,
                "exclusiveMinimum": 0,
                "type": "integer"
             },
             "train_sz": {
                "title": "Train Sz",
                "description": "If set, the number of training images to use. If␣
↪fewer images exist, then an exception will be raised.",
```


(continues on next page)

(continued from previous page)

```
                "type": "integer"
            },
            "train_sz_rel": {
                "title": "Train Sz Rel",
                "description": "If set, the proportion of training images to use.",
                "type": "number"
            },
            "num_workers": {
                "title": "Num Workers",
                "description": "Number of workers to use when DataLoader makes␣
↪batches.",
                "default": 4,
                "type": "integer"
            },
            "augmentors": {
                "title": "Augmentors",
                "description": "Names of albumentations augmentors to use for␣
↪training batches. Choices include: ['Blur', 'RandomRotate90', 'HorizontalFlip',
↪'VerticalFlip', 'GaussianBlur', 'GaussNoise', 'RGBShift', 'ToGray'].␣
↪Alternatively, a custom transform can be provided via the aug_transform option.",
                "default": [
                    "RandomRotate90",
                    "HorizontalFlip",
                    "VerticalFlip"
                ],
                "type": "array",
                "items": {
                    "type": "string"
                }
            },
            "base_transform": {
                "title": "Base Transform",
                "description": "An Albumentations transform serialized as a dict␣
↪that will be applied to all datasets: training, validation, and test. This␣
↪transformation is in addition to the resizing due to img_sz. This is useful for,␣
↪for example, applying the same normalization to all datasets.",
                "type": "object"
            },
            "aug_transform": {
                "title": "Aug Transform",
                "description": "An Albumentations transform serialized as a dict␣
↪that will be applied as data augmentation to the training dataset. This transform␣
↪is applied before base_transform. If provided, the augmentors option is ignored.",
                "type": "object"
            },
            "plot_options": {
                "title": "Plot Options",
                "description": "Options to control plotting.",
                "default": {
                    "transform": {
                        "__version__": "1.3.0",
                        "transform": {
```

(continues on next page)

(continued from previous page)


```
                        "__class_fullname__": "rastervision.pytorch_learner.utils.
↪utils.MinMaxNormalize",
                        "always_apply": false,
                        "p": 1.0,
                        "min_val": 0.0,
                        "max_val": 1.0,
                        "dtype": 5
                     }
                  },
                  "channel_display_groups": null,
                  "type_hint": "plot_options"
               },
               "allOf": [
                  {
                     "$ref": "#/definitions/PlotOptions"
                  }
               ]
            },
            "preview_batch_limit": {
               "title": "Preview Batch Limit",
               "description": "Optional limit on the number of items in the preview␣
↪plots produced during training.",
               "type": "integer"
            },
            "type_hint": {
               "title": "Type Hint",
               "default": "data",
               "enum": [
                  "data"
               ],
               "type": "string"
            }
         },
         "additionalProperties": false
      }
   }
}
```


Config

- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

Fields

- *data (rastervision.pytorch_learner.learner_config.DataConfig)*
- *log_tensorboard (bool)*
- *model (rastervision.pytorch_learner.semantic_segmentation_learner_config.SemanticSegmentationModelConfig)*
- *run_tensorboard (bool)*
- *solver (rastervision.pytorch_learner.learner_config.SolverConfig)*

- *test_mode (bool)*
- *type_hint (Literal['pytorch_semantic_segmentation_backend'])*

field data: *DataConfig* [Required]

field log_tensorboard: bool = True

If True, log events to Tensorboard log files.

field model: *SemanticSegmentationModelConfig* [Required]

field run_tensorboard: bool = False

If True, run Tensorboard server pointing at log files.

field solver: *SolverConfig* [Required]

field test_mode: bool = False

This field is passed along to the LearnerConfig which is returned by get_learner_config(). For more info, see the docs forpytorch_learner.learner_config.LearnerConfig.test_mode.

field type_hint: Literal['pytorch_semantic_segmentation_backend'] = 'pytorch_semantic_segmentation_backend'

build(*pipeline*, *tmp_dir*)

Build an instance of the corresponding type of object using this config.

For example, BackendConfig will build a Backend object. The arguments to this method will vary depending on the type of Config.

filter_commands(*commands: List[str]*) → List[str]

Filter out any commands that are not needed or supported.

Parameters
commands (*List[str]*) –

Return type
List[str]

get_bundle_filenames()

Returns the names of files that should be included in a model bundle.

The files are assumed to be in the train/ directory generated by the train command. Note that only the names, not the full paths should be returned.

get_img_channels(*pipeline_cfg:* RVPipelineConfig) → int

Determine img_channels from scenes.

Parameters
pipeline_cfg (RVPipelineConfig) –

Return type
int

get_learner_config(*pipeline*)

recursive_validate_config()

Recursively validate hierarchies of Configs.

This uses reflection to call validate_config on a hierarchy of Configs using a depth-first pre-order traversal.

revalidate()

Re-validate an instantiated Config.

Runs all Pydantic validators plus self.validate_config().

Adapted from: https://github.com/samuelcolvin/pydantic/issues/1864#issuecomment-679044432

update(*pipeline: Optional[RVPipelineConfig] = None*)

Update any fields before validation.

Subclasses should override this to provide complex default behavior, for example, setting default values as a function of the values of other fields. The arguments to this method will vary depending on the type of Config.

Parameters

pipeline (*Optional[RVPipelineConfig]*) –

validate_config()

Validate fields that should be checked after update is called.

This is to complement the builtin validation that Pydantic performs at the time of object construction.

validate_list(*field: str*, *valid_options: List[str]*)

Validate a list field.

Parameters

- **field** (*str*) – name of field to validate
- **valid_options** (*List[str]*) – values that field is allowed to take

Raises

ConfigError – if field is invalid

9.5 aws_s3

Modules

s3_file_system

9.5.1 s3_file_system

Classes

S3FileSystem	A FileSystem for interacting with files stored on AWS S3.

S3FileSystem

class S3FileSystem

Bases: *FileSystem*

A FileSystem for interacting with files stored on AWS S3.

Uses Everett configuration of form: ` [AWS_S3] requester_pays=True `

__init__()

Methods

__init__()	
copy_from(src_uri, dst_path)	Copy a source file to a local destination.
copy_to(src_path, dst_uri)	Copy a local source file to a destination.
file_exists(uri[, include_dir])	Check if a file exists.
get_file_system(uri[, mode])	Return FileSystem that should be used for the given URI/mode pair.
get_request_payer()	
get_session()	
last_modified(uri)	Get the last modified date of a file.
list_paths(uri[, ext])	List paths rooted at URI.
local_path(uri, download_dir)	Return the path where a local copy should be stored.
matches_uri(uri, mode)	Returns True if this FS can be used for the given URI/mode pair.
parse_uri(uri)	Parse bucket name and key from an S3 URI.
read_bytes(uri)	Read contents of URI to bytes.
read_str(uri)	Read contents of URI to a string.
sync_from_dir(src_dir_uri, dst_dir[, delete])	Syncs a source directory to a local destination directory.
sync_to_dir(src_dir, dst_dir_uri[, delete])	Syncs a local source directory to a destination directory.
write_bytes(uri, data)	Write bytes in data to URI.
write_str(uri, data)	Write string in data to URI.

static copy_from(*src_uri: str*, *dst_path: str*) → None

Copy a source file to a local destination.

If the FileSystem is remote, this involves downloading.

Parameters

- **src_uri** (*str*) – uri of source that can be copied from by this FileSystem
- **dst_path** (*str*) – local path to destination file

Return type
None

static copy_to(*src_path: str*, *dst_uri: str*) → None

Copy a local source file to a destination.

If the FileSystem is remote, this involves uploading.

Parameters

- **src_path** (*str*) – local path to source file
- **dst_uri** (*str*) – uri of destination that can be copied to by this FileSystem

Return type
None

static file_exists(*uri: str*, *include_dir: bool = True*) → bool

Check if a file exists.

Parameters

- **uri** (*str*) – The URI to check
- **include_dir** (*bool*) – Include directories in check, if this file_system supports directory reads. Otherwise only return true if a single file exists at the URI.

Return type
bool

static get_file_system(*uri: str*, *mode: str = 'r'*) → *FileSystem*

Return FileSystem that should be used for the given URI/mode pair.

Parameters

- **uri** (*str*) – URI of file
- **mode** (*str*) – mode to open file in, 'r' or 'w'

Return type
FileSystem

static get_request_payer()

static get_session()

static last_modified(*uri: str*) → datetime

Get the last modified date of a file.

Parameters
uri (*str*) – the URI of the file

Returns
the last modified date in UTC of a file or None if this FileSystem does not support this operation.

Return type
datetime

static list_paths(*uri*, *ext=''*)

List paths rooted at URI.

Optionally only includes paths with a certain file extension.

Parameters

- **uri** – the URI of a directory

- **ext** – the optional file extension to filter by

static local_path(*uri: str*, *download_dir: str*) → None

Return the path where a local copy should be stored.

Parameters

- **uri** (*str*) – the URI of the file to be copied
- **download_dir** (*str*) – path of the local directory in which files should be copied

Return type
None

static matches_uri(*uri: str*, *mode: str*) → bool

Returns True if this FS can be used for the given URI/mode pair.

Parameters

- **uri** (*str*) – URI of file
- **mode** (*str*) – mode to open file in, 'r' or 'w'

Return type
bool

static parse_uri(*uri: str*) → Tuple[str, str]

Parse bucket name and key from an S3 URI.

Parameters
uri (*str*) –

Return type
Tuple[str, str]

static read_bytes(*uri: str*) → bytes

Read contents of URI to bytes.

Parameters
uri (*str*) –

Return type
bytes

static read_str(*uri: str*) → str

Read contents of URI to a string.

Parameters
uri (*str*) –

Return type
str

static sync_from_dir(*src_dir_uri: str*, *dst_dir: str*, *delete: bool = False*) → None

Syncs a source directory to a local destination directory.

If the FileSystem is remote, this involves downloading.

Parameters

- **src_dir_uri** (*str*) – source directory that can be synced from by this FileSystem
- **dst_dir** (*str*) – A local destination directory
- **delete** (*bool*) – True if the destination should be deleted first.

Return type
None

static sync_to_dir(*src_dir: str*, *dst_dir_uri: str*, *delete: bool = False*) → None

Syncs a local source directory to a destination directory.

If the FileSystem is remote, this involves uploading.

Parameters

- **src_dir** (*str*) – local source directory to sync from
- **dst_dir_uri** (*str*) – A destination directory that can be synced to by this FileSystem
- **delete** (*bool*) – True if the destination should be deleted first.

Return type
None

static write_bytes(*uri: str*, *data: bytes*) → None

Write bytes in data to URI.

Parameters

- **uri** (*str*) –
- **data** (*bytes*) –

Return type
None

static write_str(*uri: str*, *data: str*) → None

Write string in data to URI.

Parameters

- **uri** (*str*) –
- **data** (*str*) –

Return type
None

Functions

get_matching_s3_keys(bucket[, prefix, ...])	Generate the keys in an S3 bucket.
get_matching_s3_objects(bucket[, prefix, ...])	Generate objects in an S3 bucket.
progressbar(total_size, desc)	

get_matching_s3_keys

get_matching_s3_keys(*bucket*, *prefix=''*, *suffix=''*, *request_payer='None'*)

Generate the keys in an S3 bucket.

Parameters

- **bucket** – Name of the S3 bucket.
- **prefix** – Only fetch keys that start with this prefix (optional).
- **suffix** – Only fetch keys that end with this suffix (optional).

get_matching_s3_objects

get_matching_s3_objects(*bucket*, *prefix=''*, *suffix=''*, *request_payer='None'*)

Generate objects in an S3 bucket.

Parameters

- **bucket** – Name of the S3 bucket.
- **prefix** – Only fetch objects whose key starts with this prefix (optional).
- **suffix** – Only fetch objects whose keys end with this suffix (optional).

progressbar

progressbar(*total_size: int*, *desc: str*)

Parameters

- **total_size** (*int*) –
- **desc** (*str*) –

9.6 aws_batch

Modules

aws_batch_runner

9.6.1 aws_batch_runner

Classes

AWSBatchRunner	Runs pipelines remotely using AWS Batch.

AWSBatchRunner

class AWSBatchRunner

Bases: *Runner*

Runs pipelines remotely using AWS Batch.

Requires Everett configuration of form:

```
[AWS_BATCH] cpu_job_queue= cpu_job_def= gpu_job_queue= gpu_job_def= attempts=
```

__init__()

Methods

__init__()	
get_split_ind()	Get the split_ind for the process.
run(cfg_json_uri, pipeline, commands[, ...])	Run commands in a Pipeline using a serialized PipelineConfig.

get_split_ind()

Get the split_ind for the process.

For split commands, the split_ind determines which split of work to perform within the current OS process. The CLI has a –split-ind option, but some runners may have their own means of communicating the split_ind, and this method should be overridden in such cases. If this method returns None, then the –split-ind option will be used. If both are null, then it won't be possible to run the command.

run(*cfg_json_uri*, *pipeline*, *commands*, *num_splits=1*, *pipeline_run_name: str = 'raster-vision'*)

Run commands in a Pipeline using a serialized PipelineConfig.

Parameters

- **cfg_json_uri** – URI of a JSON file with a serialized PipelineConfig
- **pipeline** – the Pipeline to run
- **commands** – names of commands to run
- **num_splits** – number of splits to use for splittable commands
- **pipeline_run_name** (*str*) –

Functions

submit_job(cmd, job_name[, debug, profile, ...])	Submit a job to run on AWS Batch.

submit_job

submit_job(*cmd: List[str]*, *job_name: str*, *debug: bool = False*, *profile: str = False*, *attempts: int = 5*, *parent_job_ids: Optional[List[str]] = None*, *num_array_jobs: Optional[int] = None*, *use_gpu: bool = False*, *job_queue: Optional[str] = None*, *job_def: Optional[str] = None*) → str

Submit a job to run on AWS Batch.

Parameters

- **cmd** (*List[str]*) – a command to run in the Docker container for the remote job
- **debug** (*bool*) – if True, run the command using a ptvsd wrapper which sets up a remote VS Code Python debugger server
- **profile** (*str*) – if True, run the command using kernprof, a line profiler
- **attempts** (*int*) – the number of times to try running the command which is useful in case of failure.
- **parent_job_ids** (*Optional[List[str]]*) – optional list of parent Batch job ids. The job created by this will only run after the parent jobs complete successfully.
- **num_array_jobs** (*Optional[int]*) – if set, make this a Batch array job with size equal to num_array_jobs
- **use_gpu** (*bool*) – if True, run the job in a GPU-enabled queue
- **job_queue** (*Optional[str]*) – if set, use this job queue
- **job_def** (*Optional[str]*) – if set, use this job definition
- **job_name** (*str*) –

Return type

str

CHAPTER

TEN

CONTRIBUTING

We are happy to take contributions! It is best to get in touch with the maintainers about larger features or design changes *before* starting the work, as it will make the process of accepting changes smoother.

10.1 Contributor License Agreement (CLA)

Everyone who contributes code to Raster Vision will be asked to sign the Azavea CLA, which is based off of the Apache CLA.

1. Download a copy of the Raster Vision Individual Contributor License Agreement or the Raster Vision Corporate Contributor License Agreement
2. Print out the CLAs and sign them, or use PDF software that allows placement of a signature image.
3. Send the CLAs to Azavea by one of: - Scanning and emailing the document to cla@azavea.com - Faxing a copy to +1-215-925-2600. - Mailing a hardcopy to: `Azavea, 990 Spring Garden Street, 5th Floor, Philadelphia, PA 19107 USA`

10.1.1 Release Process

This is a guide to the process of creating a new release, and is meant for the maintainers of Raster Vision.

Note: The following instructions assume that Python 3 is the default Python on your local system. Using Python 2 will not work.

Minor or Major Version Release

1. It's a good idea to update any major dependencies before the release.
2. Update the docs if needed. See the docs README for instructions.
3. Checkout the `master` branch, re-build the docker image (`docker/build`), and push it to ECR (`docker/ecr_publish`).
4. Execute all tutorial notebooks and make sure they work correctly. Do not commit output changes unless code behavior has changed.
5. Run all *Examples* and check that evaluation metrics are close to the scores from the last release. (For each example, there should be a link to a JSON file with the evaluation metrics from the last release.) This stage often uncovers bugs, and is the most time consuming part of the release process. There is a script to help run the examples and collect their outputs. See the associated README for details.

6. Collect all model bundles, and check that they work with the `predict` command and sanity check output in QGIS.
7. Update the *Model Zoo* by uploading model bundles and sample images to the right place on S3. If you use the `collect` command (described here), you should be able to sync the `collect_dir` to `s3://azavea-research-public-data/raster-vision/examples/model-zoo-<version>`.
8. Update the notebooks that use models from the model zoo so that they use the latest version and re-run.
9. Update tiny_spacenet.py if needed and ensure the line numbers in every `literalinclude` of that file are correct. Tip: you can find all instances by searching the repo using the regex: `\.\. literalinclude:: .+tiny_spacenet\.py$`.
10. Test *Installation* and *Quickstart* instructions and make sure they work.
11. Test examples from *Writing pipelines and plugins*.

```
rastervision run inprocess rastervision.pipeline_example_plugin1.config1 -a root_
↪uri /opt/data/pipeline-example/1/ --splits 2
rastervision run inprocess rastervision.pipeline_example_plugin1.config2 -a root_
↪uri /opt/data/pipeline-example/2/ --splits 2
rastervision run inprocess rastervision.pipeline_example_plugin2.config3 -a root_
↪uri /opt/data/pipeline-example/3/ --splits 2
```

12. Test examples from *Bootstrap new projects with a template*.

```
cookiecutter /opt/src/cookiecutter_template
```

13. Update the the changelog, and point out API changes.
14. Fix any broken badges on the GitHub repo readme.
15. Update the version number. This occurs in several places, so it's best to do this with a find and replace over the entire repo.
16. Make a PR to the `master` branch with the preceding updates. In the PR, there should be a link to preview the docs. Check that they are building and look correct.
17. Make a git branch with the version as the name, and push to GitHub.
18. Ensure that the docs are building correctly for the new version branch on readthedocs. You will need to have admin access on your RTD account. Once the branch is building successfully, Under *Versions* -> *Activate a Version*, you can activate the version to add it to the sidebar of the docs for the latest version. (This might require manually triggering a rebuild of the docs.) Then, under *Admin* -> *Advanced Settings*, change the default version to the new version.
19. GitHub Actions is supposed to publish an image whenever there is a push to a branch with a version number as the name. If this doesn't work or you want to publish it immediately, then you can manually make a Docker image for the new version and push to Quay. For this you will need an account on Quay.io under the Azavea organization.

```
./docker/build
docker login quay.io
docker tag raster-vision-pytorch:latest quay.io/azavea/raster-vision:pytorch-
↪<version>
docker push quay.io/azavea/raster-vision:pytorch-<version>
```

20. Make a GitHub tag and release using the previous release as a template.
21. Publish all packages to PyPI. This step requires twine which you can install with

```
pip install twine
```

To store settings for PyPI you can setup a `~/.pypirc` file containing:

```
[pypi]
username = azavea
```

Once packages are published they cannot be changed so be careful. (It's possible to practice using testpypi.) Navigate to the `raster-vision` repo on your local filesystem. With the version branch checked out, run something like the following to publish each plugin, and then the top-level package.

```
export RV="/Users/lfishgold/projects/raster-vision"
```

```
cd $RV/rastervision_pipeline
python setup.py sdist bdist_wheel
twine upload dist/*
```

```
cd $RV/rastervision_aws_batch
python setup.py sdist bdist_wheel
twine upload dist/*
```

```
cd $RV/rastervision_aws_s3
python setup.py sdist bdist_wheel
twine upload dist/*
```

```
cd $RV/rastervision_core
python setup.py sdist bdist_wheel
twine upload dist/*
```

```
cd $RV/rastervision_pytorch_learner
python setup.py sdist bdist_wheel
twine upload dist/*
```

```
cd $RV/rastervision_pytorch_backend
python setup.py sdist bdist_wheel
twine upload dist/*
```

```
cd $RV/rastervision_gdal_vsi
python setup.py sdist bdist_wheel
twine upload dist/*
```

```
cd $RV
python setup.py sdist bdist_wheel
twine upload dist/*
```

22. Announce new release in our forum, and with blog post if it's a big release.
23. Make a PR to the master branch that updates the version number to the next development version. For example, if the last release was `0.20.1`, update the version

Bug Fix Release

This describes how to create a new bug fix release, using incrementing from 0.8.0 to 0.8.1 as an example. This assumes that there is already a branch for a minor release called `0.8`.

1. To create a bug fix release (version 0.8.1), we need to backport all the bug fix commits on the `master` branch that have been added since the last bug fix release onto the `0.8` branch. For each bug fix PR on `master`, we need to create a PR against the `0.8` branch based on a branch of `0.8` that has cherry-picked the commits from the original PR. The title of the PR should start with [BACKPORT].
2. Make and merge a PR against `0.8` (but not `master`) that increments the version in each `setup.py` file to `0.8.1`. Then wait for the `0.8` branch to be built by GitHub Actions and the `0.8` Docker images to be published to Quay. If that is successful, we can proceed to the next steps of actually publishing a release.
3. Using the GitHub UI, make a new release. Use `0.8.1` as the tag, and the `0.8` branch as the target.
4. Publish the new version to PyPI. Follow the same instructions for PyPI that are listed above for minor/major version releases.

10.1.2 Code of Conduct

Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to make participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

Scope

This Code of Conduct applies within all project spaces, and it also applies when an individual is representing the project or its community in public spaces. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team: Lewis Fishgold lfishgold@azavea.com or Adeel Hassan ahassan@azavea.com. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

Attribution

This Code of Conduct is adapted from the PyTorch Code of Conduct, which is adapted from the Contributor Covenant, version 1.4, available at https://www.contributor-covenant.org/version/1/4/code-of-conduct.html

For answers to common questions about this code of conduct, see https://www.contributor-covenant.org/faq

CHAPTER

ELEVEN

CHANGELOG

11.1 CHANGELOG

11.1.1 Raster Vision 0.20.1

Fixes

- Do not install `rastervision_gdal_vsi` by default (#1622)
- Do not set `cfg.model.pretrained=False` in `Learner.from_model_bundle()` (#1626)
- Fix docker build errors (#1629)
- Documentation:
  - Improve docstrings for most commonly used classes and configs (#1630)
  - Minor textual fixes for the pre-chipped datasets tutorial (#1623)
  - Add comment about password for the ISPRS Potsdam dataset (#1627)
- README:
  - fix broken links (#1608)
  - make CV-tasks image slightly smaller (#1624)

11.1.2 Raster Vision 0.20

This release brings major improvements to Raster Vision's **usability** as well as its **usefulness**.

Whereas previously Raster Vision was a **framework** where users could configure a *pipeline* and then let it run, it is now *also* a **library** from which users can pick individual components and use them to build new things.

We have also significantly improved the documentation. Most notably, it now contains detailed *tutorial notebooks* as well a full *API reference*. The documentation for the Raster Vision pipeline, which used to make up most of the documentation in previous versions, is now located in the *The Raster Vision Pipeline* section.

In terms of features, some highlights are:

- Support for multiband imagery, introduced in v0.13 for semantic segmentation, is now also available for chip classification and object detection. (#1345)
- Improved data fusion: the *`MultiRasterSource`* can now combine *`RasterSources`* with varying extents and resolutions. (#1308)

- You can now discard edges of predicted chips in semantic segmentation in order to reduce boundary artifacts (#1486). This can be used *in addition* to the previously introduced ability to average overlapping regions in adjacent chips.
- Progress-bars will now be shown for all downloads and uploads as well as other time-consuming operations that take longer than 5 seconds.
- Improved caching of downloads: Raster Vision can now cache downloads. Also a bug that caused Raster Vision to download the same image multiple times has been fixed, resulting in significant speedups.

Warning: This release breaks backward-compatibility with previous versions.

Features

- Extend multiband support to all tasks (#1345)
- Add support for external models for object detection (#1337)
- Allow `MultiRasterSource` to read from sub raster sources with non-identical extents and resolutions (#1308)
- Allow discarding edges of predicted chips in semantic segmentation (#1486)
- Add numpy-like array indexing and slicing to `RasterSource` and `LabelSource` (#1470)
- Make `RandomWindowGeoDataset` more efficient when sampling chips from scenes with sparse AOIs (#1225)
- Add support for Albumentations' lambda transforms (#1368)
- Provide grouping mechanism for scenes and use it in the `analyze` and `eval` stages (#1375)
- Update STAC-reading functionality to make it compatible with STAC v1.0.* (#1243)
- Add progress bars for downloads and uploads (#1343)
- Allow caching downloads (#1450)

Refactoring

- Refactor `Learner` and related configs to be more flexible and easier to use in a notebook (#1413)
- Refactor to make it easier to programmatically make predictions on new scenes (#1434)
- Refactor: make `Evaluator` easier to use independently (#1438)
- Refactor vector data handling (#1437, #1461)
- Add `GeoDataset.from_uris()` for convenient initialization of `GeoDatasets` (#1462, #1588)
- Add `Labels.save()` convenience method (#1486)
- Factor out dataset visualization into a `Visualizer` class (#1476)
- Replace `STRTree` with GeoPandas `GeoDataFrame`-based spatial joins in `ChipClassificaitonLabelSource` and `RasterizedSource` (#1470)
- Remove `ActivateMixin` entirely (#1470)
- Remove the `mask-to-polygons` dependency (#1470)

Documentation

- Update documentation site (#1501, #1589)
- Refactor documentation (#1561)
- Add tutorial notebooks (#1470, #1506, #1586, #1546)
- Add code of conduct (#1160)

Fixes

- Speed up `RGBClassTransformer` by an order of magnitude (#1485)
- Fix `rastervision_pipeline` entry point to ensure commands from other plugins are available (#1250)
- Fix incorrect F1 scores when aggregating evals for scenes in the eval stage (#1386)
- Fix bug in semantic segmentation prediction output paths (#1354)
- Do not zero out null class pixels when creating semantic segmentation training chips (#1556)
- Fix a bug in `DataConfig` validation and refactor `ClassConfig` (#1436)
- Fix #1052 (#1451)
- Fix #991 and #1452 (#1484)
- Fix #1430 (#1495)
- Misc. fixes (#1260, #1281, #1453)

Development/maintenance

- Make the semantic segmentation integration test more deterministic (#1261)
- Migrate from Travis to GitHub Actions (#1218)
- Add Github issue templates (#1242, #1288, #1420)
- Switch from Gitter to Github Discussions (#1464, #1465)
- Update cloudformation template to allow use of on-demand GPU instances (#1482)
- Add option to build ARM64 Docker image (#1545, #1559)
- Make `docker/run` automatically find a free port for Jupyter server if the default port is already taken (#1558)
- Set tutorial-notebooks path as the default jupyter path in `docker/run` (#1595)

11.1.3 Raster Vision 0.13.1

Bug Fixes

- Fix image plot by adding default plot transform #1144

11.1.4 Raster Vision 0.13

This release presents a major jump in Raster Vision's power and flexibility. The most significant changes are:

Support arbitrary models and loss functions (#985, #992)

Raster Vision is no longer restricted to using the built in models and loss functions. It is now possible to import models and loss functions from a GitHub repo or a URI or a zip file as long as they interface correctly with RV's learner code. This means that you can now easily swap models in your existing training pipelines, allowing you to take advantage of the latest models or to make customizations that help with your specific task; all with minimal changes.

This is made possible by PyTorch's `hub` module.

Currently not supported for Object Detection.

Support for multiband images (even with Transfer Learning) (#972)

It is now possible to train on imagery with more than 3 channels. Raster Vision automatically modifies the model to be able to accept more than 3 channels. If using pretrained models, the pre-learned weights are retained.

The model modification cannot be performed automatically when using an external model. But as long as the external model supports multiband inputs, it will work correctly with RV.

Currently only supported for Semantic Segmentation.

Support for reading directly from raster sources during training without chipping (#1046)

It is no longer necessary to go through a `chip` stage to produce a training dataset. You can instead provide the `DatasetConfig` directly to the PyTorch backend and RV will sample training chips on the fly during training. All the examples now use this as the default. Check them out to see how to use this feature.

Support for arbitrary Albumentations transforms (#1001)

It is now possible to supply an arbitrarily complicated Albumentations transform for data augmentation. In the `DataConfig` subclasses, you can specify a `base_transform` that is applied every time (i.e. in training, validation, and prediction), an `aug_transform` that is only applied during training, and a `plot_transform` (via `PlotOptions`) to ensure that sample images are plotted correctly (e.g. use `plot_transform` to rescale a normalized image to 0-1).

Allow streaming reads from Rasterio sources (#1020)

It is now possible to stream chips from a remote `RasterioSource` without first downloading the entire file. To enable, set `allow_streaming=True` in the `RasterioSourceConfig`.

Analyze stage no longer necessary when using non-uint8 rasters (#972)

It is no longer necessary to go through an `analyze` stage to be able to convert non-`uint8` rasters to `uint8` chips. Chips can now be stored as `numpy` arrays, and will be normalized to `float` during training/prediction based on their specific data type. See `spacenet_vegas.py` for example usage.

Currently only supported for Semantic Segmentation.

Features

- Add support for multiband images #972
- Add support for vector output to predict command #980
- Add support for weighted loss for classification and semantic segmentation #977
- Add multi raster source #978
- Add support for fetching and saving external model definitions #985
- Add support for external loss definitions #992
- Upgrade to pyproj 2.6 #1000
- Add support for arbitrary albumentations transforms #1001
- Minor tweaks to regression learner #1013
- Add ability to specify number of PyTorch reader processes #1008
- Make img_sz specifiable #1012
- Add ignore_last_class capability to segmentation #1017
- Add filtering capability to segmentation sliding window chip generation #1018
- Add raster transformer to remove NaNs from float rasters, add raster transformers to cast to arbitrary numpy types #1016
- Add plot options for regression #1023
- Add ability to use fewer channels w/ pretrained models #1026
- Remove 4GB file size limit from VSI file system, allow streaming reads #1020
- Add reclassification transformer for segmentation label rasters #1024
- Allow filtering out chips based on proportion of NODATA pixels #1025
- Allow ignore_last_class to take either a boolean or the literal 'force'; in the latter case validation of that argument is skipped so that it can be used with external loss functions #1027
- Add ability to crop raster source extent #1030
- Accept immediate geometries in SceneConfig #1033
- Only perform normalization on unsigned integer types #1028
- Make group_uris specifiable and add group_train_sz_rel #1035
- Make number of training and dataloader previews independent of batch size #1038
- Allow continuing training from a model bundle #1022
- Allow reading directly from raster source during training without chipping #1046
- Remove external commands (obsoleted by external architectures and loss functions) #1047

- Allow saving SS predictions as probabilities #1057
- Update CUDA version from 10.1 to 10.2 #1115
- Add integration tests for the nochip functionality #1116
- Update examples to make use of the nochip functionality by default #1116

Bug Fixes

- Update all relevant saved URIs in config before instantiating Pipeline #993
- Pass verbose flag to batch jobs #988
- Fix: Ensure Integer class_id #990
- Use `--ipc=host` by default when running the docker container #1077

11.1.5 Raster Vision 0.12

This release presents a major refactoring of Raster Vision intended to simplify the codebase, and make it more flexible and customizable.

To learn about how to upgrade existing experiment configurations, perhaps the best approach is to read the source code of the *Examples* to get a feel for the new syntax. Unfortunately, existing predict packages will not be usable with this release, and upgrading and re-running the experiments will be necessary. For more advanced users who have written plugins or custom commands, the internals have changed substantially, and we recommend reading *Architecture and Customization*.

Since the changes in this release are sweeping, it is difficult to enumerate a list of all changes and associated PRs. Therefore, this change log describes the changes at a high level, along with some justifications and pointers to further documentation.

Simplified Configuration Schema

We are still using a modular, programmatic approach to configuration, but have switched to using a `Config` base class which uses the Pydantic library. This allows us to define configuration schemas in a declarative fashion, and let the underlying library handle serialization, deserialization, and validation. In addition, this has allowed us to DRY up the configuration code, eliminate the use of Protobufs, and represent configuration from plugins in the same fashion as built-in functionality. To see the difference, compare the configuration code for `ChipClassificationLabelSource` in 0.11 (label_source.proto and chip_classification_label_source_config.py), and in 0.12 (chip_classification_label_source_config.py).

Abstracted out Pipelines

Raster Vision includes functionality for running computational pipelines in local and remote environments, but previously, this functionality was tightly coupled with the "domain logic" of machine learning on geospatial data in the `Experiment` abstraction. This made it more difficult to add and modify commands, as well as use this functionality in other projects. In this release, we factored out the experiment running code into a separate *rastervision.pipeline* package, which can be used for defining, configuring, customizing, and running arbitrary computational pipelines.

Reorganization into Plugins

The rest of Raster Vision is now written as a set of optional plugins that have `Pipelines` which implement the "domain logic" of machine learning on geospatial data. Implementing everything as optional (`pip` installable) plugins makes it easier to install subsets of Raster Vision functionality, eliminates separate code paths for built-in and plugin functionality, and provides (de facto) examples of how to write plugins. See *Codebase Overview* for more details.

More Flexible PyTorch Backends

The 0.10 release added PyTorch backends for chip classification, semantic segmentation, and object detection. In this release, we abstracted out the common code for training models into a flexible `Learner` base class with subclasses for each of the computer vision tasks. This code is in the `rastervision.pytorch_learner` plugin, and is used by the `Backends` in `rastervision.pytorch_backend`. By decoupling `Backends` and `Learners`, it is now easier to write arbitrary `Pipelines` and new `Backends` that reuse the core model training code, which can be customized by overriding methods such as `build_model`. See *Customizing Raster Vision*.

Removed Tensorflow Backends

The Tensorflow backends and associated Docker images have been removed. It is too difficult to maintain backends for multiple deep learning frameworks, and PyTorch has worked well for us. Of course, it's still possible to write `Backend` plugins using any framework.

Other Changes

- For simplicity, we moved the contents of the raster-vision-examples and raster-vision-aws repos into the main repo. See *Examples* and *Setup AWS Batch using CloudFormation*.
- To help people bootstrap new projects using RV, we added *Bootstrap new projects with a template*.
- All the PyTorch backends now offer data augmentation using albumentations.
- We removed the ability to automatically skip running commands that already have output, "tree workflows", and "default providers". We also unified the `Experiment`, `Command`, and `Task` classes into a single `Pipeline` class which is subclassed for different computer vision (or other) tasks. These features and concepts had little utility in our experience, and presented stumbling blocks to outside contributors and plugin writers.
- Although it's still possible to add new `VectorSources` and other classes for reading data, our philosophy going forward is to prefer writing pre-processing scripts to get data into the format that Raster Vision can already consume. The `VectorTileVectorSource` was removed since it violates this new philosophy.
- We previously attempted to make predictions for semantic segmentation work in a streaming fashion (to avoid running out of RAM), but the implementation was buggy and complex. So we reverted to holding all predictions for a scene in RAM, and now assume that scenes are roughly < 20,000 x 20,000 pixels. This works better anyway from a parallelization standponit.
- We switched to writing chips to disk incrementally during the `CHIP` command using a `SampleWriter` class to avoid running out of RAM.
- The term "predict package" has been replaced with "model bundle", since it rolls off the tongue better, and `BUNDLE` is the name of the command that produces it.
- Class ids are now indexed starting at 0 instead of 1, which seems more intuitive. The "null class", used for marking pixels in semantic segmentation that have not been labeled, used to be 0, and is now equal to `len(class_ids)`.
- The `aws_batch` runner was renamed `batch` due to a naming conflict, and the names of the configuration variables for Batch changed. See *Running on AWS Batch*.

Future Work

The next big features we plan on developing are:

- the ability to read and write data in STAC format using the label extension. This will facilitate integration with other tools such as GroundWork.

11.1.6 Raster Vision 0.11

Features

- Added the possibility for chip classification to use data augmentors from the albumentations libary to enhance the training data. #859
- Updated the Quickstart doc with pytorch docker image and model #863
- Added the possibility to deal with class imbalances through oversampling. #868

Raster Vision 0.11.0

Bug Fixes

- Ensure randint args are ints #849
- The augmentors were not serialized properly for the chip command #857
- Fix problems with pretrained flag #860
- Correctly get_local_path for some zxy tile URIS #865

11.1.7 Raster Vision 0.10

Raster Vision 0.10.0

Notes on switching to PyTorch-based backends

The current backends based on Tensorflow have several problems:

- They depend on third party libraries (Deeplab, TF Object Detection API) that are complex, not well suited to being used as dependencies within a larger project, and are each written in a different style. This makes the code for each backend very different from one other, and unnecessarily complex. This increases the maintenance burden, makes it difficult to customize, and makes it more difficult to implement a consistent set of functionality between the backends.
- Tensorflow, in the maintainer's opinion, is more difficult to write and debug than PyTorch (although this is starting to improve).
- The third party libraries assume that training images are stored as PNG or JPG files. This limits our ability to handle more than three bands and more that 8-bits per channel. We have recently completed some research on how to train models on > 3 bands, and we plan on adding this functionality to Raster Vision.

Therefore, we are in the process of sunsetting the Tensorflow backends (which will probably be removed) and have implemented replacement PyTorch-based backends. The main things to be aware of in upgrading to this version of Raster Vision are as follows:

- Instead of there being CPU and GPU Docker images (based on Tensorflow), there are now tf-cpu, tf-gpu, and pytorch (which works on both CPU and GPU) images. Using `./docker/build --tf` or `./docker/build --pytorch` will only build the TF or PyTorch images, respectively.
- Using the TF backends requires being in the TF container, and similar for PyTorch. There are now `--tf-cpu`, `--tf-gpu`, and `--pytorch-gpu` options for the `./docker/run` command. The default setting is to use the PyTorch image in the standard (CPU) Docker runtime.
- The raster-vision-aws CloudFormation setup creates Batch resources for TF-CPU, TF-GPU, and PyTorch. It also now uses default AMIs provided by AWS, simplifying the setup process.
- To easily switch between running TF and PyTorch jobs on Batch, we recommend creating two separate Raster Vision profiles with the Batch resources for each of them.
- The way to use the `ConfigBuilders` for the new backends can be seen in the examples repo and the *Backend* reference

Features

- Add confusion matrix as metric for semantic segmentation #788
- Add predict_chip_size as option for semantic segmentation #786
- Handle "ignore" class for semantic segmentation #783
- Add stochastic gradient descent ("SGD") as an optimizer option for chip classification #792
- Add option to determine if all touched pixels should be rasterized for rasterized RasterSource #803
- Script to generate GeoTIFF from ZXY tile server #811
- Remove QGIS plugin #818
- Add PyTorch backends and add PyTorch Docker image #821 and #823.

Bug Fixes

- Fixed issue with configuration not being able to read lists #784
- Fixed ConfigBuilders not supporting type annotations in __init__ #800

11.1.8 Raster Vision 0.9

Raster Vision 0.9.0

Features

- Add requester_pays RV config option #762
- Unify Docker scripts #743
- Switch default branch to master #726
- Merge GeoTiffSource and ImageSource into RasterioSource #723
- Simplify/clarify/test/validate RasterSource #721
- Simplify and generalize geom processing #711
- Predict zero for nodata pixels on semantic segmentation #701

- Add support for evaluating vector output with AOIs #698
- Conserve disk space when dealing with raster files #692
- Optimize StatsAnalyzer #690
- Include per-scene eval metrics #641
- Make and save predictions and do eval chip-by-chip #635
- Decrease semseg memory usage #630
- Add support for vector tiles in .mbtiles files #601
- Add support for getting labels from zxy vector tiles #532
- Remove custom `__deepcopy__` implementation from `ConfigBuilder`s. #567
- Add ability to shift raster images by given numbers of meters. #573
- Add ability to generate GeoJSON segmentation predictions. #575
- Add ability to run the DeepLab eval script. #653
- Submit CPU-only stages to a CPU queue on Aws. #668
- Parallelize CHIP and PREDICT commands #671
- Refactor `update_for_command` to split out the IO reporting into `report_io`. #671
- Add Multi-GPU Support to DeepLab Backend #590
- Handle multiple AOI URIs #617
- Give `train_restart_dir` Default Value #626
- Use `make` to manage local execution #664
- Optimize vector tile processing #676

Bug Fixes

- Fix Deeplab resume bug: update path in checkpoint file #756
- Allow Spaces in `--channel-order` Argument #731
- Fix error when using predict packages with AOIs #674
- Correct checkpoint name #624
- Allow using default stride for semseg sliding window #745
- Fix filter_by_aoi for ObjectDetectionLabels #746
- Load null channel_order correctly #733
- Handle Rasterio crs that doesn't contain EPSG #725
- Fixed issue with saving semseg predictions for non-georeferenced imagery #708
- Fixed issue with handling width > height in semseg eval #627
- Fixed issue with experiment configs not setting key names correctly #576
- Fixed issue with Raster Sources that have channel order #576

11.1.9 Raster Vision 0.8

Raster Vision 0.8.1

Bug Fixes

- Allow multiploygon for chip classification #523
- Remove unused args for AWS Batch runner #503
- Skip over lines when doing chip classification, Use background_class_id for scenes with no polygons #507
- Fix issue where `get_matching_s3_keys` fails when `suffix` is `None` #497

PYTHON MODULE INDEX

r

INDEX

Symbols

A

B

C

D

E

F

G

H

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z