Raster Vision Documentation

Release 0.10.0

Azavea

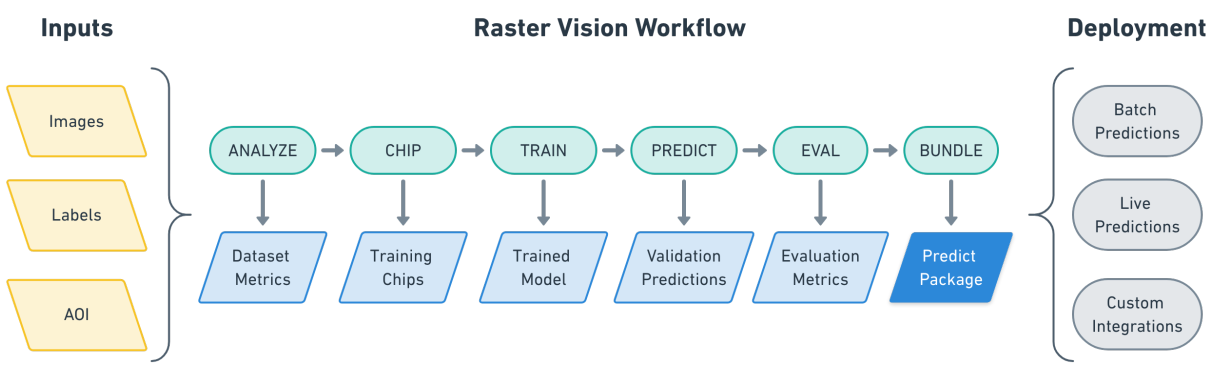
Oct 14, 2019

Contents

Raster Vision is an open source 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 and Tensorflow.

Chip Classification   Object Detection   Semantic Segmentation

Raster Vision allows engineers to quickly and repeatably configure *experiments* 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.



Raster Vision workflows begin when you have a set of images and training data, optionally with Areas of Interest (AOIs) that describe where the images are labeled. Raster Vision workflows end with a packaged model and configuration that allows you to easily utilize models in various deployment situations. Inside the Raster Vision workflow, there's the process of running multiple experiments to find the best model or models to deploy.

The process of running experiments includes executing workflows that perform 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 variety of "backends" such as TensorFlow or Keras.
- **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 into a *Predict Package*, which can be deployed in batch processes, live servers, and other workflows.

Experiments are configured using a fluent builder pattern that makes configuration easy to read, reuse and maintain.

```
# tiny_spacenet.py

import rastervision as rv

class TinySpacenetExperimentSet(rv.ExperimentSet):
    def exp_main(self):
        base_uri = ('https://s3.amazonaws.com/azavea-research-public-data/'
                    'raster-vision/examples/spacenet')
        train_image_uri = '{}/RGB-PanSharpen_AOI_2_Vegas_img205.tif'.format(base_uri)
        train_label_uri = '{}/buildings_AOI_2_Vegas_img205.geojson'.format(base_uri)
        val_image_uri = '{}/RGB-PanSharpen_AOI_2_Vegas_img25.tif'.format(base_uri)
        val_label_uri = '{}/buildings_AOI_2_Vegas_img25.geojson'.format(base_uri)
        channel_order = [0, 1, 2]
        background_class_id = 2

        # ------------- TASK -------------

        task = rv.TaskConfig.builder(rv.SEMANTIC_SEGMENTATION) \
                            .with_chip_size(300) \
                            .with_chip_options(chips_per_scene=50) \
                            .with_classes({
                                'building': (1, 'red'),
                                'background': (2, 'black')
                            }) \
                            .build()

        # ------------- BACKEND -------------

        backend = rv.BackendConfig.builder(rv.PYTORCH_SEMANTIC_SEGMENTATION) \
            .with_task(task) \
            .with_train_options(
                batch_size=2,
                num_epochs=1,
                debug=True) \
            .build()

        # ------------- TRAINING -------------

        train_raster_source = rv.RasterSourceConfig.builder(rv.RASTERIO_SOURCE) \
                                                   .with_uri(train_image_uri) \
                                                   .with_channel_order(channel_order)␣
↪\
                                                   .with_stats_transformer() \
                                                   .build()

        train_label_raster_source = rv.RasterSourceConfig.builder(rv.RASTERIZED_
↪SOURCE) \
```

(continues on next page)

(continued from previous page)

```
                                                          .with_vector_source(train_
↪label_uri) \
                                                          .with_rasterizer_
↪options(background_class_id) \
                                                          .build()
       train_label_source = rv.LabelSourceConfig.builder(rv.SEMANTIC_SEGMENTATION) \
                                                  .with_raster_source(train_label_
↪raster_source) \
                                                  .build()

       train_scene =  rv.SceneConfig.builder() \
                                    .with_task(task) \
                                    .with_id('train_scene') \
                                    .with_raster_source(train_raster_source) \
                                    .with_label_source(train_label_source) \
                                    .build()

       # ------------- VALIDATION -------------

       val_raster_source = rv.RasterSourceConfig.builder(rv.RASTERIO_SOURCE) \
                                                .with_uri(val_image_uri) \
                                                .with_channel_order(channel_order) \
                                                .with_stats_transformer() \
                                                .build()

       val_label_raster_source = rv.RasterSourceConfig.builder(rv.RASTERIZED_SOURCE)␣
↪\
                                                        .with_vector_source(val_label_
↪uri) \
                                                        .with_rasterizer_
↪options(background_class_id) \
                                                        .build()
       val_label_source = rv.LabelSourceConfig.builder(rv.SEMANTIC_SEGMENTATION) \
                                                .with_raster_source(val_label_raster_
↪source) \
                                                .build()

       val_scene = rv.SceneConfig.builder() \
                                 .with_task(task) \
                                 .with_id('val_scene') \
                                 .with_raster_source(val_raster_source) \
                                 .with_label_source(val_label_source) \
                                 .build()

       # ------------- DATASET -------------

       dataset = rv.DatasetConfig.builder() \
                                 .with_train_scene(train_scene) \
                                 .with_validation_scene(val_scene) \
                                 .build()

       # ------------- EXPERIMENT -------------

       experiment = rv.ExperimentConfig.builder() \
                                       .with_id('tiny-spacenet-experiment') \
                                       .with_root_uri('/opt/data/rv') \
                                       .with_task(task) \
```

(continues on next page)

(continued from previous page)

```
                                        .with_backend(backend) \
                                        .with_dataset(dataset) \
                                        .with_stats_analyzer() \
                                        .build()

        return experiment


if __name__ == '__main__':
    rv.main()
```

Raster Vision uses a `unittest`-like method for executing experiments. 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 aws_batch -p tiny_spacenet.py
```

See the *Quickstart* for a more complete description of using this example.

This part of the documentation guides you through all of the library's usage patterns.

CHAPTER 1

Why Raster Vision?

1.1 Why do we need yet another deep learning library?

Most machine learning libraries implement the core functionality needed to 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.

1.2 What are the benefits of using Raster Vision?

- Programmatically configure workflows in a concise, modifiable, and reusable way, using abstractions such as *ExperimentConfig*, *Task*, *Backend*, *Dataset*, and *Scene*.
- Let the framework handle the challenges and idiosyncrasies of doing machine learning on massive, geospatial imagery.
- Run experiments 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 *FileSystems* architecture.
- Make predictions and build inference pipelines using a single "prediction package" which includes the trained model and configuration.
- Add new data sources, tasks, and backends using the *Plugins* architecture.

1.3 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 hard 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 2

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 experiments 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 18, 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 source file and another for holding experiment output. Make sure these directories exist:

```
> export RV_QUICKSTART_CODE_DIR=`pwd`/code
> export RV_QUICKSTART_EXP_DIR=`pwd`/rv_root
> mkdir -p ${RV_QUICKSTART_CODE_DIR} ${RV_QUICKSTART_EXP_DIR}
```

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

```
> docker run --rm -it -p 6006:6006 \
     -v ${RV_QUICKSTART_CODE_DIR}:/opt/src/code  \
     -v ${RV_QUICKSTART_EXP_DIR}:/opt/data \
     quay.io/azavea/raster-vision:cpu-0.10 /bin/bash
```

See also:

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

2.1 The Data

2.2 Creating an ExperimentSet

Create a Python file in the `${RV_QUICKSTART_CODE_DIR}` named `tiny_spacenet.py`. Inside, you're going to create an *Experiment Set*. You can think of an `ExperimentSet` a lot like the `unittest.TestSuite`: It's a class that contains specially-named methods that are run via reflection by the `rastervision` command line tool.

```
# tiny_spacenet.py

import rastervision as rv

class TinySpacenetExperimentSet(rv.ExperimentSet):
    def exp_main(self):
        base_uri = ('https://s3.amazonaws.com/azavea-research-public-data/'
                    'raster-vision/examples/spacenet')
        train_image_uri = '{}/RGB-PanSharpen_AOI_2_Vegas_img205.tif'.format(base_uri)
        train_label_uri = '{}/buildings_AOI_2_Vegas_img205.geojson'.format(base_uri)
        val_image_uri = '{}/RGB-PanSharpen_AOI_2_Vegas_img25.tif'.format(base_uri)
        val_label_uri = '{}/buildings_AOI_2_Vegas_img25.geojson'.format(base_uri)
        channel_order = [0, 1, 2]
        background_class_id = 2

        # ------------- TASK -------------

        task = rv.TaskConfig.builder(rv.SEMANTIC_SEGMENTATION) \
                            .with_chip_size(300) \
                            .with_chip_options(chips_per_scene=50) \
                            .with_classes({
                                'building': (1, 'red'),
                                'background': (2, 'black')
                            }) \
                            .build()

        # ------------- BACKEND -------------

        backend = rv.BackendConfig.builder(rv.PYTORCH_SEMANTIC_SEGMENTATION) \
            .with_task(task) \
            .with_train_options(
                batch_size=2,
                num_epochs=1,
                debug=True) \
            .build()

        # ------------- TRAINING -------------

        train_raster_source = rv.RasterSourceConfig.builder(rv.RASTERIO_SOURCE) \
                                                   .with_uri(train_image_uri) \
                                                   .with_channel_order(channel_order)␣
↪\

                                                   .with_stats_transformer() \
                                                   .build()
```

(continues on next page)

(continued from previous page)

```
        train_label_raster_source = rv.RasterSourceConfig.builder(rv.RASTERIZED_
↪SOURCE) \
                                                            .with_vector_source(train_
↪label_uri) \
                                                            .with_rasterizer_
↪options(background_class_id) \
                                                            .build()
        train_label_source = rv.LabelSourceConfig.builder(rv.SEMANTIC_SEGMENTATION) \
                                                    .with_raster_source(train_label_
↪raster_source) \
                                                    .build()

        train_scene =  rv.SceneConfig.builder() \
                                  .with_task(task) \
                                  .with_id('train_scene') \
                                  .with_raster_source(train_raster_source) \
                                  .with_label_source(train_label_source) \
                                  .build()

        # ------------- VALIDATION -------------

        val_raster_source = rv.RasterSourceConfig.builder(rv.RASTERIO_SOURCE) \
                                                    .with_uri(val_image_uri) \
                                                    .with_channel_order(channel_order) \
                                                    .with_stats_transformer() \
                                                    .build()

        val_label_raster_source = rv.RasterSourceConfig.builder(rv.RASTERIZED_SOURCE)␣
↪\
                                                          .with_vector_source(val_label_
↪uri) \
                                                          .with_rasterizer_
↪options(background_class_id) \
                                                          .build()
        val_label_source = rv.LabelSourceConfig.builder(rv.SEMANTIC_SEGMENTATION) \
                                                  .with_raster_source(val_label_raster_
↪source) \
                                                  .build()

        val_scene = rv.SceneConfig.builder() \
                                .with_task(task) \
                                .with_id('val_scene') \
                                .with_raster_source(val_raster_source) \
                                .with_label_source(val_label_source) \
                                .build()

        # ------------- DATASET -------------

        dataset = rv.DatasetConfig.builder() \
                                .with_train_scene(train_scene) \
                                .with_validation_scene(val_scene) \
                                .build()

        # ------------- EXPERIMENT -------------

        experiment = rv.ExperimentConfig.builder() \
                                      .with_id('tiny-spacenet-experiment') \
```

(continues on next page)

(continued from previous page)

```
                                        .with_root_uri('/opt/data/rv') \
                                        .with_task(task) \
                                        .with_backend(backend) \
                                        .with_dataset(dataset) \
                                        .with_stats_analyzer() \
                                        .build()

        return experiment


if __name__ == '__main__':
    rv.main()
```

The `exp_main` method has a special name: any method starting with `exp_` is one that Raster Vision will look for experiments in. Raster Vision does this by calling the method and processing any experiments that are returned - you can either return a single experiment or a list of experiments.

Notice that we set up a `SceneConfig`, which points to a `RasterSourceConfig`, and calls `with_label_source` with a GeoJSON URI, which sets a default `LabelSourceConfig` type into the scene based on the extension of the URI. We also set a `StatsTransformer` to be used for the `RasterSource` by calling `with_stats_transformer()`, which sets a default `StatsTransformerConfig` onto the `RasterSourceConfig` transformers. This transformer is needed to convert uint16 values in the rasters to the uint8 values needed by the data loader in PyTorch. (In the future, we plan on relaxing this requirement.)

2.3 Running an experiment

Now that you've configured an experiment, we can perform a dry run of executing it to see what running the full workflow will look like:

```
> cd /opt/src/code
> rastervision run local -p tiny_spacenet.py -n

Ensuring input files exist    [####################################]  100%
Checking for existing output  [####################################]  100%

Commands to be run in this order:
ANALYZE from tiny-spacenet-experiment

CHIP from tiny-spacenet-experiment
  DEPENDS ON: ANALYZE from tiny-spacenet-experiment

TRAIN from tiny-spacenet-experiment
  DEPENDS ON: CHIP from tiny-spacenet-experiment

BUNDLE from tiny-spacenet-experiment
  DEPENDS ON: ANALYZE from tiny-spacenet-experiment
  DEPENDS ON: TRAIN from tiny-spacenet-experiment

PREDICT from tiny-spacenet-experiment
  DEPENDS ON: ANALYZE from tiny-spacenet-experiment
  DEPENDS ON: TRAIN from tiny-spacenet-experiment

EVAL from tiny-spacenet-experiment
```

(continues on next page)

(continued from previous page)

```
DEPENDS ON: ANALYZE from tiny-spacenet-experiment
DEPENDS ON: PREDICT from tiny-spacenet-experiment
```

The console output above is what you should expect - although there will be a color scheme to make things easier to read in terminals that support it.

Here we see that we're about to run the ANALYZE, CHIP, TRAIN, BUNDLE, PREDICT, and EVAL commands, and what they depend on. You can change the verbosity to get even more dry run output - we won't list the output here to save space, but give it a try:

```
> rastervision -v run local -p tiny_spacenet.py -n
> rastervision -vv run local -p tiny_spacenet.py -n
```

When we're ready to run, we just remove the `-n` flag:

```
> rastervision run local -p tiny_spacenet.py
```

2.4 Seeing Results

If you go to `${RV_QUICKSTART_EXP_DIR}` you should see a folder structure like this.

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

```
> tree -L 3
.
├── analyze
│   └── tiny-spacenet-experiment
│       ├── command-config-0.json
│       └── stats.json
├── bundle
│   └── tiny-spacenet-experiment
│       ├── command-config-0.json
│       └── predict_package.zip
├── chip
│   └── tiny-spacenet-experiment
│       ├── chips
│       └── command-config-0.json
├── eval
│   └── tiny-spacenet-experiment
│       ├── command-config-0.json
│       └── eval.json
├── experiments
│   └── tiny-spacenet-experiment.json
├── predict
│   └── tiny-spacenet-experiment
│       ├── command-config-0.json
│       └── val_scene.tif
└── train
    └── tiny-spacenet-experiment
        ├── command-config-0.json
        ├── done.txt
        ├── log.csv
```

(continues on next page)

(continued from previous page)

```
        ├── logs
        ├── model
        ├── models
        ├── train-debug-chips.zip
        └── val-debug-chips.zip
```

Each directory with a command name contains output for that command type across experiments. The directory inside those have our experiment ID as the name - this is so different experiments can share `root_uri`'s without overwriting each other's output. You can also use "keys", e.g. `.with_chip_key('chip-size-300')` on an `ExperimentConfigBuilder` to set the directory for a command across experiments, so that they can share command output. This is useful in the case where many experiments have the same CHIP output, and so you only want to run that once for many train commands from various experiments. The experiment configuration is also saved off in the `experiments` directory.

Don't get too excited to look at the evaluation results in `eval/tiny-spacenet-experiment/` - we trained a model for 1 step, 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.

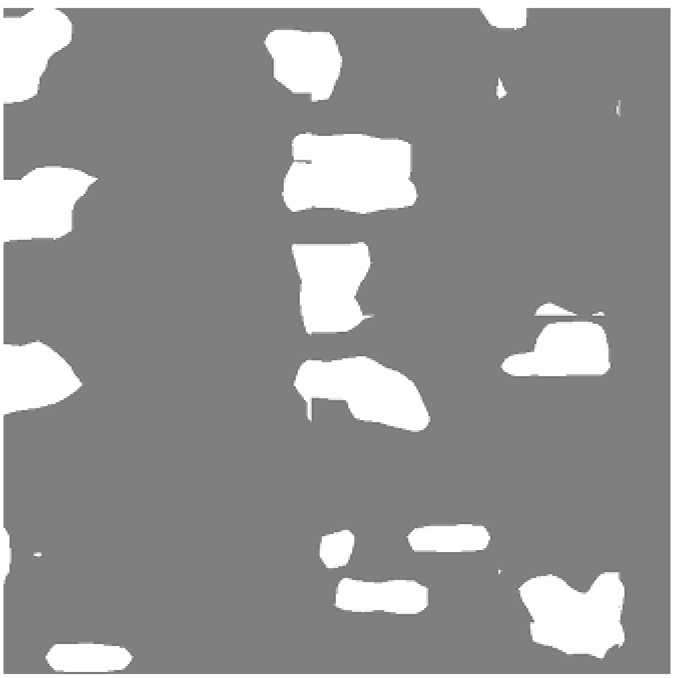
2.5 Predict Packages

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 perform semantic segmentation using a MobileNet-based DeepLab model that has been pretrained for Las Vegas, one can type:

```
> rastervision predict https://s3.amazonaws.com/azavea-research-public-data/raster-
↪vision/examples/model-zoo/vegas-building-seg/predict_package.zip https://s3.
↪amazonaws.com/azavea-research-public-data/raster-vision/examples/model-zoo/vegas-
↪building-seg/1929.tif predictions.tif
```

This will perform a prediction on the image `1929.tif` using the provided prediction package, and will produce a file called `predictions.tif` that contains the predictions. 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 *Predict Package* concept and the *predict* CLI command in the documentation.

2.6 Next Steps

This is just a quick example of a Raster Vision workflow. For a more complete example of how to train a model on SpaceNet (optionally using GPUs on AWS Batch), see the SpaceNet examples in the Raster Vision Examples repository.

CHAPTER 3

Setup

3.1 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 18, although you may be able to use a lower version.

Docker images 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-0.10 /bin/bash
```

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.

There are Raster Vision backends for PyTorch and Tensorflow – the Tensorflow ones are being sunsetted. We publish separate Docker images with the dependencies necessary for using the PyTorch and Tensorflow backends, and there are CPU and GPU variants for the Tensorflow images. There are also images with the *-latest* suffix for the latest commits on the `master` branch. The available images include:

- `quay.io/azavea/raster-vision:tf-gpu-0.10` and `quay.io/azavea/raster-vision:tf-gpu-latest`
- `quay.io/azavea/raster-vision:tf-cpu-0.10` and `quay.io/azavea/raster-vision:tf-cpu-latest`
- `quay.io/azavea/raster-vision:pytorch-0.10` and `quay.io/azavea/raster-vision:pytorch-latest`

You can also base your own Dockerfiles off the Raster Vision image to use with your own codebase. See the Dockerfiles in the Raster Vision Examples repository.

3.1.1 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.

After cloning the repo, you can build all the Docker images using:

```
> docker/build
```

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.

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

Remember to use the correct image for the backend you are using!

```
> ./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)
AWS_PROFILE (optional AWS profile)
RASTER_VISION_REPO (optional path to main RV repo; mounted to /opt/src)

Options:
--aws forwards AWS credentials (sets AWS_PROFILE env var and mounts ~/.aws to /root/.
↪aws)
--tensorboard maps port 6006
--gpu use the NVIDIA runtime and GPU image
--name sets the name of the running container
--jupyter forwards port 8888, mounts ./notebooks to /opt/notebooks, and runs Jupyter
--debug maps port 3007 on localhost to 3000 inside container
--tf-gpu use raster-vision-examples-tf-gpu image and nvidia runtime
--tf-cpu use raster-vision-examples-tf-cpu image
--pytorch-gpu use raster-vision-examples-pytorch image and nvidia runtime

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

3.2 Installing via pip

Rather than running Raster Vision from inside a Docker container, you can directly install the library using `pip`. However, we recommend using the Docker images since it can be difficult to install some of the dependencies.

```
> pip install rastervision==0.10.0
```

Note: Raster Vision requires Python 3 or later. Use `pip3 install rastervision==0.10.0` if you have more than one version of Python installed.

3.2.1 Troubleshooting macOS Installation

If you encounter problems running `pip install rastervision==0.10.0` on macOS, you may have to manually install Cython and pyproj.

To circumvent a problem installing pyproj with Python 3.7, you may also have to install that library using `git+https`:

```
> pip install cython
> pip install git+https://github.com/jswhit/pyproj.
↪git@e56e879438f0a1688b89b33228ebda0f0d885c19
> pip install rastervision==0.10.0
```

3.2.2 Using AWS, Tensorflow, and/or Keras

If you'd like to use AWS, PyTorch, Tensorflow and/or Keras with Raster Vision, you can include any of these extras:

```
> pip install rastervision[aws,pytorch,tensorflow-cpu,tensorflow-gpu]==0.10.0
```

If you'd like to use Raster Vision with Tensorflow Object Detection or TensorFlow DeepLab, you'll need to install these from Azavea's fork of the models repository, since it contains some necessary changes that have not yet been merged back upstream.

You will also need to install Tippecanoe if you would like to do vector tile processing. For an example of setting these up, see the various Dockerfiles.

3.3 Raster Vision Configuration

Raster Vision is configured via the everett library.

Raster Vision 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.

3.3.1 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.

3.3.2 Configuration File Sections

RV

```
[RV]
model_defaults_uri = ""
```

- `model_defaults_uri` - Specifies the URI of the *Model Defaults* JSON. Leave this option out to use the Raster Vision supplied model defaults.

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.

PLUGINS

```
[PLUGINS]
files=analyzers.py,backends.py
modules=rvplugins.analyzer,rvplugins.backend
```

- `files` - Optional list of Python file URIs to gather plugins from as a comma-separated list of values, e.g. `analyzers.py,backends.py`.
- `modules` - Optional list of modules to load plugins from as a comma-separated list of values, e.g. `rvplugins.analyzer,rvplugins.backend`.

See *Plugins* for more information about the Plugin architecture.

3.3.3 Other Sections

Other configurations are documented elsewhere:

- *AWS Batch Configuration Section*

3.3.4 Environment Variables

Any INI file option can also be stated in the environment. Just prepend the section name to the setting name, e.g. `RV_MODEL_DEFAULTS_URI`.

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`

3.4 Running on a machine with GPUs

If you would like to run Raster Vision in a Docker container with GPUs - e.g. if you have your own GPU machine or you spun up a GPU-enabled machine on a cloud provider like a p3.2xlarge on AWS - you'll need to check some things so that the Docker container can utilize the GPUs.

Here are some (slightly out of date, but still useful) instructions written by a community member on setting up an AWS account and a GPU-enabled EC2 instance to run Raster Vision.

3.4.1 Install nvidia-docker

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!

3.4.2 Use the nvidia-docker runtime

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-0.10 /bin/
↪bash
```

3.4.3 Ensure your setup sees the GPUS

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 TensorFlow can see the GPU(s). To do this, open up an ipython console and initialize TensorFlow:

```
> ipython
In [1]: import tensorflow as tf
In [2]: sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))
```

This should print out console output that looks something like:

```
.../gpu/gpu_device.cc:1405] Found device 0 with properties: name: GeForce GTX
```

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
```

3.5 Setting up AWS Batch

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 the CloudFormation template in the Raster Vision AWS Batch repository.

3.5.1 AWS Batch Configuration Section

After creating the resources on AWS, set the corresponding configuration in your *Raster Vision Configuration*:

```
[AWS_BATCH]
job_queue=RasterVisionGpuJobQueue
job_definition=RasterVisionHostedPyTorchGpuJobDefinition
cpu_job_queue=RasterVisionCpuJobQueue
cpu_job_definition=RasterVisionHostedPyTorchCpuJobDefinition
attempts=5
```

- `job_queue` - Job Queue to submit GPU Batch jobs to.
- `cpu_job_queue` - Job Queue to submit CPU-only jobs to.
- `job_definition` - The Job Definition that defines the Batch jobs to run on GPU.
- `cpu_job_definition` - The Job Definition that defines the Batch jobs to run on CPU (which might be the same as the `job_definition`)
- `attempts` - Optional number of attempts to retry failed jobs.

Check the AWS Batch console to see the names of the resources that were created, as they vary depending on how CloudFormation was configured.

If you would like the ability to switch between PyTorch and Tensorflow-based jobs, you should create separate Raster Vision profiles for each of the two sets of resources.

See also:

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

CHAPTER 4

Experiment Configuration

Experiments are configured programmatically using a compositional API based on the *Fluent Builder Pattern*.

4.1 Experiment Set

An experiment set is a set of related experiments and can be created by subclassing `ExperimentSet`. For each experiment, the class should have a method prefixed with `exp_` that returns either a single `ExperimentConfig`, or a list of `ExperimentConfig` objects. You can also return a `CommandConfig` directly or multiple in a list; this is useful when running *Auxiliary (Aux) Commands*.

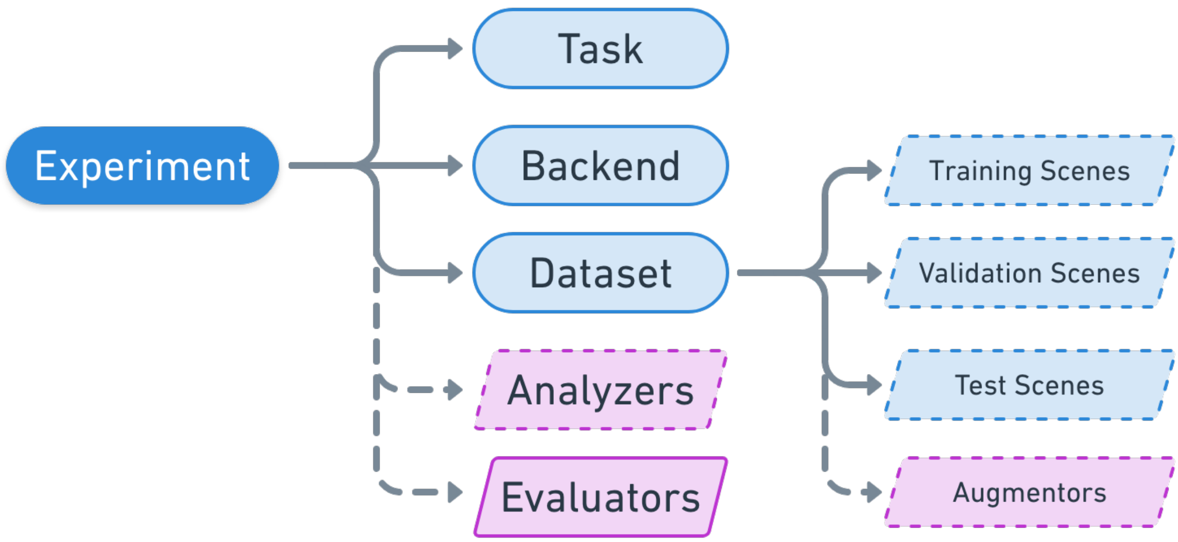
In the `tiny_spacenet.py` example from the *Quickstart*, the `TinySpacenetExperimentSet` is the `ExperimentSet` that Raster Vision finds when executing `rastervision run -p tiny_spacenet.py`.

```
import rastervision as rv

class TinySpacenetExperimentSet(rv.ExperimentSet):
    def exp_main(self):
        # Here we return an experiment or list of experiments
        pass

    # We could also add other experiment methods
    def exp_other_examples(self):
        pass


if __name__ == '__main__':
    rv.main()
```

4.2 ExperimentConfig

An experiment is a sequence of commands that represents a machine learning workflow. The way those workflows are configured is by constructing an `ExperimentConfig`. An `ExperimentConfig` is what is returned from the experiment methods of an `ExperimentSet`, and are used by Raster Vision to determine what and how *Commands* will be run. While the actual execution of the commands, be it locally or on AWS Batch, are determined by *ExperimentRunners*, all the details about how the commands will execute (which files, what methods, what hyperparameters, etc.) are determined by the `ExperimentConfig`.

The following diagram shows the hierarchy of the high level components that comprise an experiment configuration:



In the `tiny_spacenet.py` example, we can see that the experiment is the very last thing constructed and returned.

```
experiment = rv.ExperimentConfig.builder() \
                                .with_id('tiny-spacenet-experiment') \
                                .with_root_uri('/opt/data/rv') \
                                .with_task(task) \
                                .with_backend(backend) \
                                .with_dataset(dataset) \
                                .with_stats_analyzer() \
                                .build()
```

4.3 Task

A `Task` is a computer vision task such as chip classification, object detection, or semantic segmentation. Tasks are configured using a `TaskConfig`, which is then set into the experiment with the `.with_task(task)` method.

Chip Classification

Object Detection

Semantic Segmentation

4.3.1 Chip Classification

rv.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.

4.3.2 Object Detection

rv.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.

4.3.3 Semantic Segmentation

rv.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.

4.3.4 New Tasks

It is possible to add support for new tasks by extending the Task class. Some potential tasks to add are chip regression (goal: predict a number for each chip) and instance segmentation (goal: predict a segmentation mask for each individual object).

4.3.5 TaskConfig

A `TaskConfig` is always constructed through a builder, which is created by passing a **key** to the `.builder` static method of `TaskConfig`. In our `tiny_spacenet.py` example, we configured a semantic segmentation task:

```
task = rv.TaskConfig.builder(rv.SEMANTIC_SEGMENTATION) \
                    .with_chip_size(300) \
                    .with_chip_options(chips_per_scene=50) \
                    .with_classes({
                        'building': (1, 'red')
                    }) \
                    .build()
```

See also:

The *TaskConfigBuilder* API Reference docs have more information about the Task types available.

4.4 Backend

To avoid reinventing the wheel, Raster Vision relies on third-party libraries to implement core functionality around building and training models for the various computer vision tasks it supports. To maintain flexibility and avoid being tied to any one library, Raster Vision tasks interact with other libraries via a "backend" interface inspired by Keras. Each backend is a subclass of `Backend` and mediates between Raster Vision data structures and another library. Backends are configured using a `BackendConfig, which is then set into the experiment using the ``.with_backend(backend).`

We are in the process of sunsetting the Tensorflow-based backends in favor of backends based on PyTorch.

4.4.1 PyTorch Chip Classification

rv.PYTORCH_CHIP_CLASSIFICATION

For chip classification, the default backend is PyTorch Chip Classification. It trains classification models from torchvision.

4.4.2 PyTorch Semantic Segmentation

rv.PYTORCH_SEMANTIC_SEGMENTATION

For semantic segmentation, the default backend is PyTorch Semantic Segmentation. It trains the DeepLabV3 model in torchvision.

4.4.3 PyTorch Object Detection

rv.PYTORCH_OBJECT_DETECTION

For object detection, the default backend is PyTorch Object Detection. It trains the Faster-RCNN model in torchvision.

4.4.4 TensorFlow Object Detection

rv.TF_OBJECT_DETECTION

For object detection, the default backend is the Tensorflow Object Detection API. It supports a variety of object detection architectures such as SSD, Faster-RCNN, and RetinaNet with Mobilenet, ResNet, and Inception as base models.

4.4.5 Keras Classification

rv.KERAS_CLASSIFICATION

This backend uses Keras Classification, a small, simple interal library for image classification using Keras. Currently, it only has support for ResNet50.

4.4.6 TensorFlow DeepLab

rv.TF_DEEPLAB

This backend has support for the Deeplab segmentation architecture with Mobilenet and Inception as base models.

Note: For each Tensorflow-based backend included with Raster Vision there is a list of *Model Defaults* with a default configuration for each model architecture. Each default can be considered a good starting point for configuring that model.

4.4.7 BackendConfig

A `BackendConfig` is always constructed through a builder, which is created with a **key** using the `.builder` static method of `BackendConfig`. In our `tiny_spacenet.py` example, we configured the PyTorch semantic segmentation backend:

```
backend = rv.BackendConfig.builder(rv.PYTORCH_SEMANTIC_SEGMENTATION) \
    .with_task(task) \
    .with_train_options(
        batch_size=2,
        num_epochs=1,
        debug=True) \
    .build()
```

See also:

The *BackendConfig* API Reference docs have more information about the Backend types available.

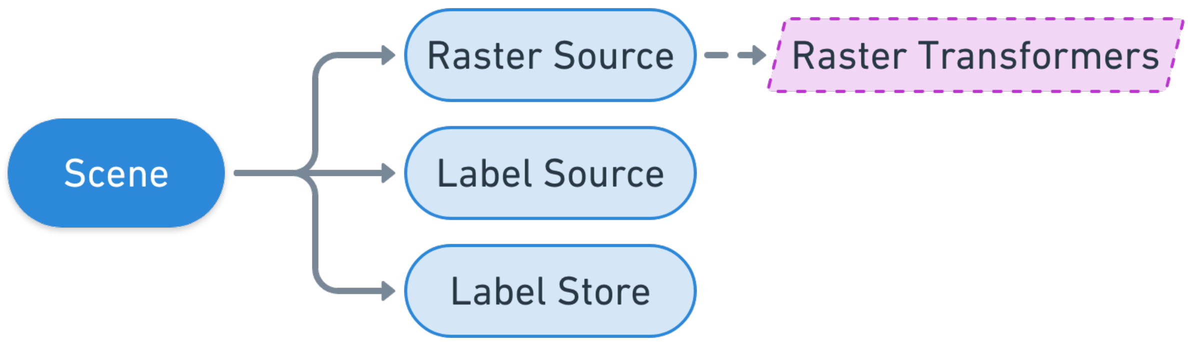
4.5 Dataset

A `Dataset` contains the training, validation, and test splits needed to train and evaluate a model. Each dataset split is a list of scenes. A dataset can also hold an *Augmentors*, which describes how to augment the training scenes (but not the validation and test scenes).

In our `tiny_spacenet.py` example, we configured the dataset with single scenes, though more often in real use cases you would call `with_train_scenes` and `with_validation_scenes` with many scenes:

```
dataset = rv.DatasetConfig.builder() \
                          .with_train_scenes(train_scenes) \
                          .with_validation_scenes(val_scenes) \
                          .build()
```

4.6 Scene



A scene represents an image, associated labels, and an optional list of areas of interest (AOIs) that describes which parts of the scene have been exhaustively labeled. Labels are task-specific annotations, and can represent geometries (bounding boxes for object detection or chip classification), rasters (semantic segmentation), or even numerical values (for regression tasks, not yet implemented). Specifying an AOI allows Raster Vision to understand not only where it can pull "positive" chips from, or subsets of imagery that contain the target class we are trying to identify, but also lets Raster Vision know where it is able to pull "negative" examples, or subsets of imagery that are missing the target class.

A scene is composed of the following elements:

- *Image*: Represented in Raster Vision by a `RasterSource`, a large scene image can contain multiple sub-images or a single file.
- *Labels*: Represented in Raster Vision as a `LabelSource`, this is what provides the annotations or labels for the scene. The nature of the labels that are produced by the LabelSource are specific to the *Task* that the machine learning model is performing.
- *AOIs* (Optional): An optional list of areas of interest that describes which sections of the scene image (RasterSource) are exhaustively labeled.

In addition to the outline above, which describes training data completely, a *LabelStore* is also associated with scenes on which Raster Vision will perform prediction. The label store determines how to store and retrieve the predictions from a scene.

4.6.1 SceneConfig

A `SceneConfig` consists of a `RasterSourceConfig` optionally combined with a `LabelSourceConfig`, `LabelStoreConfig`, and list of AOIs. Each AOI is expected to be a URI to a GeoJSON file containing polygons.

In our `tiny_spacenet.py` example, we configured the train scene with a GeoTIFF URI and a GeoJSON URI. We pass in a `RasterSourceConfig` object to the `with_raster_source` method, but just pass the URI to `with_label_source`. This is because the `SceneConfig` can construct a default `LabelSourceConfig` based on the URI using *Default Providers*. The `LabelStoreConfig` is not explicitly set in the building of the

SceneConfig. This is because the prediction label store can be determined by *Default Providers* by finding the default LabelStore provider for a given task.

```
train_scene =  rv.SceneConfig.builder() \
                             .with_task(task) \
                             .with_id('train_scene') \
                             .with_raster_source(train_raster_source) \
                             .with_label_source(train_label_uri) \
                             .build()
```

4.6.2 RasterSource

A RasterSource represents a source of raster data for a scene, and has subclasses for various data sources. They are used to retrieve small windows of raster data from larger scenes. You can also set a subset of channels (i.e. bands) that you want to use and their order. For example, satellite imagery often contains more than three channels, but pretrained models trained on datasets like Imagenet only support three (RGB) input channels. In order to cope with this situation, we can select three of the channels to utilize.

Imagery

rv.RASTERIO_SOURCE

Any images that can be read by GDAL/Rasterio can be handled by the RasterioSource. This includes georeferenced imagery such as GeoTIFFs. If there are multiple image files that cover a single scene, you can pass the corresponding list of URIs using with_uris(), and read from the RasterSource as if it were a single stitched-together image.

The RasterioSource 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.

Rasterized Vectors

rv.RASTERIZED_SOURCE

Semantic segmentation labels stored as polygons in a VectorSource can be rasterized and read using a RasterizedSource. This is a slightly unusual use of a RasterSource as we're using it to read labels, and not images to use as input to a model.

RasterSourceConfig

In the tiny_spacenet.py example, we build the training scene raster source:

```
train_raster_source = rv.RasterSourceConfig.builder(rv.RASTERIO_SOURCE) \
                                           .with_uri(train_image_uri) \
                                           .with_stats_transformer() \
                                           .build()
```

See also:

The *RasterSourceConfig* API Reference docs have more information about RasterSources.

4.6.3 VectorSource

A VectorSource is an object that supports reading vector data like polygons and lines from various places. It is used by ObjectDetectionLabelSource and ChipClassificationLabelSource, as well as the RasterizedSource (a type of RasterSource).

VectorSourceConfig

Here is an example of configuring a VectorTileVectorSource which uses Mapbox vector tiles as a source of labels. A complete example using this is in the Spacenet Vegas example.

```
uri = 'http://foo.com/{z}/{x}/{y}.mvt'
class_id_to_filter = {1: ['has', 'building']}

b = rv.VectorSource.builder(rv.VECTOR_TILE_SOURCE) \
    .with_class_inference(class_id_to_filter=class_id_to_filter,
                          default_class_id=None) \
    .with_uri(uri) \
    .with_zoom(14) \
    .build()
```

See also:

The *VectorSourceConfig* API Reference docs have more information about the VectorSource types available.

4.6.4 LabelSource

A LabelSource is an object that allows reading ground truth labels for a scene. There are subclasses for different tasks and data formats. 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 validation scenes.

Here is an example of configuring a SemanticSegmentationLabelSource using rasterized vector data. A complete example using this is in the Spacenet Vegas example.

```
label_raster_source = rv.RasterSourceConfig.builder(rv.RASTERIZED_SOURCE) \
    .with_vector_source(vector_source) \
    .with_rasterizer_options(background_class_id, line_buffer=line_buffer) \
    .build()

label_source = rv.LabelSourceConfig.builder(rv.SEMANTIC_SEGMENTATION) \
    .with_raster_source(label_raster_source) \
    .build()
```

See also:

The *LabelSourceConfig* API Reference docs have more information about the LabelSource types available.

4.6.5 LabelStore

A LabelStore is an object that allows reading and writing predicted labels for a scene. There are subclasses for different tasks and data formats. They are used for saving predictions and then loading them during evaluation.

In the tiny_spacenet.py example, there is no explicit LabelStore supplied on the validation scene. It instead relies on the *Default Providers* architecture to determine the correct label store to use. If we wanted to state the label store explicitly, the following code would be equivalent:

```
val_label_store = rv.LabelStoreConfing.builder(rv.OBJECT_DETECTION_GEOJSON) \
                                      .build()

val_scene = rv.SceneConfig.builder() \
                          .with_task(task) \
                          .with_id('val_scene') \
                          .with_raster_source(val_raster_source) \
                          .with_label_source(val_label_uri) \
                          .with_label_store(val_label_store) \
                          .build()
```

Notice the above example does not set the explicit URI for where the `LabelStore` will store it's labels. We could do that, but if we leave that out the Raster Vision logic will set that path explicitly based on the exeriment's root directory and the predict command's key.

See also:

The *LabelStoreConfig* API Reference docs have more information about the LabelStore types available.

4.6.6 Raster Transformers

A `RasterTransformer` is a mechanism for transforming raw raster data into a form that is more suitable for being fed into a model.

See also:

The *RasterTransformerConfig* API Reference docs have more information about the RasterTransformer types available.

4.6.7 Augmentors

Data augmentation is a technique used to increase the effective size of a training dataset. It consists of transforming the images (and labels) using random shifts in position, rotation, zoom level, and color distribution. Each backend has its own ways of doing data augmentation inherited from its underlying third-party library, but some additional forms of data augmentation are implemented within Raster Vision as `Augmentors`. For instance, there is a `NodataAugmentor` which adds blocks of NODATA values to images to learn to avoid making spurious predictions over NODATA regions.

See also:

The *AugmentorConfig* API Reference docs have more information about the Augmentors available.

4.7 Analyzers

Analyzers are used to gather dataset-level statistics and metrics for use in downstream processes. Currently the only analyzer available is the `StatsAnalyzer`, which determines the distribution of values over the imagery in order to normalize values to `uint8` values in a `StatsTransformer`.

See also:

The *AnalyzerConfig* API Reference docs have more information about the Analyzers available.

4.8 Evaluators

For each task, there is an evaluator 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.

Normally you will not have to set any evaluators into the `ExperimentConfig`, as the default architecture will choose the evaluator that applies to the specific `Task` the experiment pertains to.

See also:

The *EvaluatorConfig* API Reference docs have more information about the Evaluators available.

4.9 Default Providers

Default Providers allow Raster Vision users to either state configuration simply, i.e. give a URI instead of a full configuration, or not at all. Defaults are provided for a number of configurations. There is also the ability to add new defaults via the *Plugins* architecture.

For instance, you can specify a RasterSource and LabelSource just by a URI, and the Defaults registered with the *Global Registry* will find a default that pertains to that URI. There are default LabelStores and Evaluators per Task, so you won't have to state them explicitly unless you need additional configuration or are using a non-default type.

CHAPTER 5

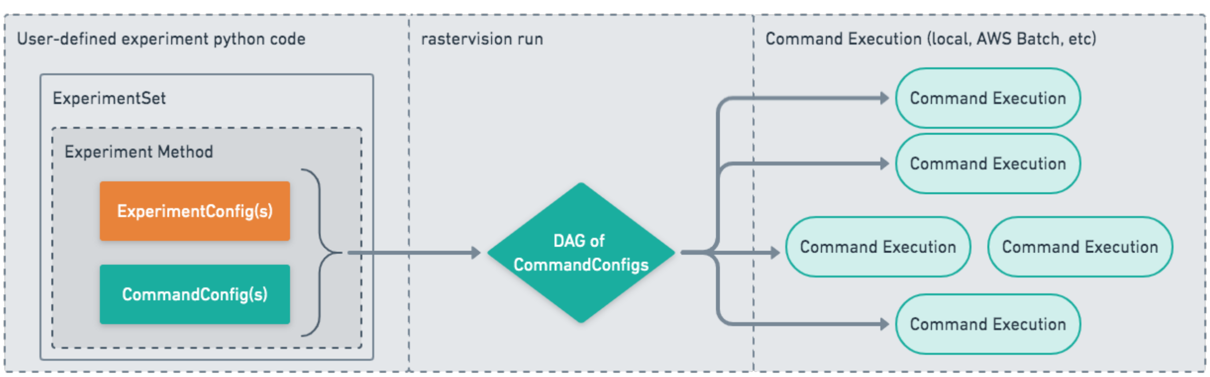
Commands

Commands are at the heart of how Raster Vision turns configuration into actions that can run in various environments (e.g. locally or on AWS Batch). When a user runs an Experiment through Raster Vision, every *ExperimentConfig* is transformed into one or more commands configurations, which are then tied together through their inputs and outputs, and used to generate the commands to be run. Without commands, experiments are simply configuration.

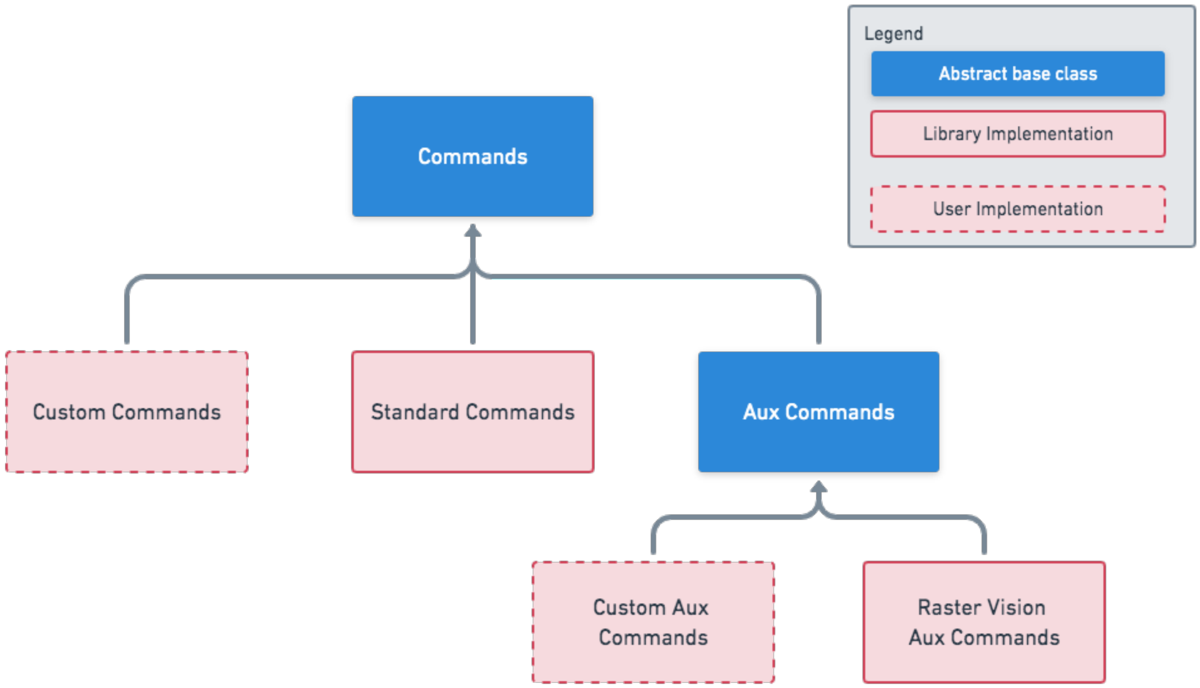
5.1 Command Generation and Execution

Commands are generated from CommandConfigs in the runner environment. Commands follow the same *Configuration vs Entity* differentiation that ExperimentConfig elements do - they are only created when and where they are to be executed. For example, if you are running Raster Vision against AWS Batch, the Commands themselves are only created in the AWS Batch task that is going to run the command.

Each `CommandConfig` is initially generated in the client environment. They can be created directly from a `CommandConfigBuilder`, or generated as part of an internal Raster Vision process that generates CommandConfigs from ExperimentConfigs. The flowchart below shows how all configurations are eventually decomposed into CommandConfigs, and then executed in the runner environment as Commands:



5.2 Command Architecture

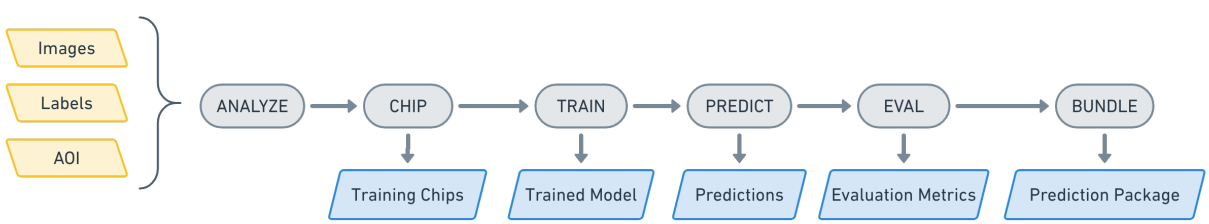


Every command derives from the `Command` abstract class, and is associated with a `CommandConfig` and `CommandConfigBuilder`. Every command must implement methods that describe the input and output of the command; this is how commands are structured in the Directed Acyclic Graph (DAG) of commands - if command B declares an input that is declared as output from command A, then there will be an edge (Command A)->(Command B) in the DAG of commands. This ensures that commands are run in the proper order. Commands often will declare their inputs implicitly based on configuration, so that you do not have to specify full URIs for inputs and outputs. However, this is command specific; e.g. Aux Commands are often more explicitly configured.

Commands are further differentiated between standard commands and auxiliary commands. Auxiliary commands are a simplified version of commands are less flexible as far as implicit configuration setting, but are often easier to utilize and implement for explicitly configured commands such as those used for preprocessing data.

5.3 Standard Commands

There are several commands that are commonly at the core to machine learning workflow, which are implemented as standard commands in Raster Vision:



5.3.1 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 *Raster Transformers*) to do the conversion.

5.3.2 CHIP

Scenes are comprised of 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.

5.3.3 TRAIN

The TRAIN command is used to train a model using the dataset generated by the CHIP command. The command is a thin wrapper around the train method in the backend that synchronizes files with the cloud, configures and calls the training routine provided by the associated third-party machine learning library, and sets up a log visualization server in some cases (e.g. Tensorboard). The output is a trained model that can be used to make predictions and fine-tune on another dataset.

5.3.4 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.

5.3.5 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.

5.3.6 BUNDLE

The BUNDLE command gathers files necessary to create a prediction package from the output of the previous commands. A prediction package contains a model file plus associated configuration data, and can be used to make predictions on new imagery in a deployed application.

5.4 Auxiliary (Aux) Commands

Raster Vision utilizes *auxiliary commands* for things like data preparation. These are commands that do not run in the normal ML pipeline (e.g., if one were to run run `rastervision run` without an command specified). Auxiliary commands normally do not have the same type of implicit configuration setting as normal commands; because of this, file paths are often set explicitly, and these commands are often configured and returned from an ExperimentSet method directly, instead of implicitly created through the ExperimentConfig.

5.4.1 Configuring Aux Commands

There are two ways to configure an Aux command: one is through custom configuration set on an ExperimentConfig, and the other is to directly return a CommandConfig instance from an experiment method. Normally Aux Commands are run separately from the normal experiment workflow, so we suggest returning command configurations as a default.

Configuring an Aux Command from an ExperimentConfig

In order to pass an Aux Command configuration through the experiment, you must set the configuration on the custom configuration of the experiment, as a dictionary of aux command configuration values, set onto a property that is the command name.

The aux command configuration dict must either have a root_uri property set, which will determine the root URI to store command configuration, or a key property, which will be used to implicitly construct the root URI based on the Experiment's overall root URI.

The aux command configuration must also have a config key, which holds the configuration values for that particular command as a dict.

For example, to set the configuration for the CogifyCommand on your experiment, you would do the following:

```
import rastervision as rv

class ExampleExperiments(rv.ExperimentSet):
   def exp_example(self):

       # Full experiment configuration builder generated elsewhere...
       experiment_builder = get_experiment_builder()

       # Before building the ExperimentConfig, set custom configuration
       # for the COGIFY Aux Command.
       e = experiment_builder \
           .with_root_uri(tmp_dir) \
           .with_custom_config({
               'cogify': {
                   'key': 'test',
                   'config': {
                       'uris': [(src_path, cog_path)],
                       'block_size': 128
                   }
               }
           }) \
           .build()

       return e
```

Configuring an Aux Command directly

You can configure the command configuration using the builder pattern directly. Aux Command builders all have the *with_root_uri* method, to set the root URI that will store command configuration, as well as the with_config method. This with_config method accepts **kwargs for configuration values.

You can return one or more command configuration directly from an experiment method, as a single command configuration or a list of configs.

Below is an example of an ExperimentSet that has one experiment method, that returns a configuration for a cogify command.

```
import rastervision as rv

class Preprocess(rv.ExperimentSet):
    def exp_cogify(self):
        root_uri = 's3://my-bucket/cogify'
        uris = [('s3://my-bucket/original/some.tif', 's3://my-bucket/cogs/some-cog.tif
↪')]

        cmd_config = rv.CommandConfig.builder(rv.COGIFY) \
                                     .with_root_uri(root_uri) \
                                     .with_config(uris=uris,
                                                  resample_method='bilinear',
                                                  compression='jpeg') \
                                     .build()

        return cmd_config
```

Running Aux Commands

By default Aux Commands won't run without explicitly being run. That means

```
> rastervision -p example run local -e example.Preprocess
```

Will not run the above Cogify command, however this will:

```
> rastervision -p example run local -e example.Preprocess cogify
```

5.5 Aux Commands included with Raster Vision

5.5.1 COGIFY

The `COGIFY` command will turn GDAL-readable images and turn them into Cloud Optimized GeoTiffs.

See the CogifyCommand entry in the *Aux Commands* API docs for configuration options.

5.6 Custom Commands

Custom Commands allow advanced Raster Vision users to implement their own commands using the *Plugins* architecture.

To create a standard custom command, you will need to create implementations of the `Command`, `CommandConfig`, and `CommandConfigBuilder` interfaces. You then need to register the `CommandConfigBuilder` using the `register_command_config_builder` method of the plugin registry.

Custom commands that are built as standard commands will by default always be run - that is, if you run *rastervision run . . .* without any specific command, your custom command will be run by default. The order in which it is run will be determined by how the inputs and outputs it declares are connected with other command definitions. One detail to note is the `update_for_command` method of custom commands will be called *after* it is called for the standard commands, in the order in which the custom commands were registered with Raster Vision.

5.7 Custom Aux Commands

Custom Aux Commands are more simple to write than a standard custom command. For instance, the following example creates and registers a custom AuxCommand that copies a file from one location to the other, with a no-op processing:

```
import rastervision as rv
from rastervision.utils.files import (download_or_copy, upload_or_copy)

def process_file(local_file_path, options):
    # Do something
    local_output_path = local_file_path
    return local_output_path

class ExampleCommand(rv.AuxCommand):
    command_type = "EXAMPLE"
    options = rv.AuxCommandOptions(
        split_on='uris',
        inputs=lambda conf: map(lambda tup: tup[0], conf['uris']),
        outputs=lambda conf: map(lambda tup: tup[1], conf['uris']),
        required_fields=['uris', 'options'])

    def run(self, tmp_dir=None):
        if not tmp_dir:
            tmp_dir = self.get_tmp_dir()

        options = self.command_config['options']
        for src, dest in self.command_config['uris']:
            src_local = download_or_copy(src, tmp_dir)
            output_local = process_file(src_local, options)
            upload_or_copy(output_local, dest)

def register_plugin(plugin_registry):
    plugin_registry.register_aux_command("EXAMPLE",
                                         ExampleCommand)
```

Notice there is only one class to implement: the `rv.AuxCommand` class.

When creating an custom AuxCommand, be sure to set the options correctly - see the *Aux Command Options* API docs for more information about options.

To use a custom command, refer to it by the `command_type` in the `rv.CommandConfig.builder(...)` method, like so:

```
import rastervision as rv

class Preprocess(rv.ExperimentSet):
   def exp_example_command(self):
       root_uri = 's3://my-bucket/example'
       uris = [('s3://my-bucket/original/some.tif', 's3://my-bucket/processed/some.tif
↪')]
       options = { 'something_useful': 'yes' }

```

(continues on next page)

(continued from previous page)

```
        cmd_config = rv.CommandConfig.builder("EXAMPLE") \
                                     .with_root_uri(root_uri) \
                                     .with_config(uris=uris,
                                                  options=options) \
                                     .build()

        return cmd_config
```

To run the command, use the `command_type` name on the command line, e.g.:

```
> rastervision -p example run local -e example.Preprocess example
```

CHAPTER 6

Running Experiments

Running experiments in Raster Vision is done using the `rastervision` *run* command. This looks in all the places stated by the command for *Experiment Set* classes and executes methods to get a collection of *ExperimentConfig* objects. These are fed into the `ExperimentRunner` that is chosen as a command line argument, which then determines how the commands derived from the experiments should be executed.

6.1 ExperimentRunners

An `ExperimentRunner` takes a collection of *ExperimentConfig* objects and executes commands derived from those configurations. The commands it chooses to run are based on which commands are requested from the user, which commands already have been run, and which commands are common between ExperimentConfigs.

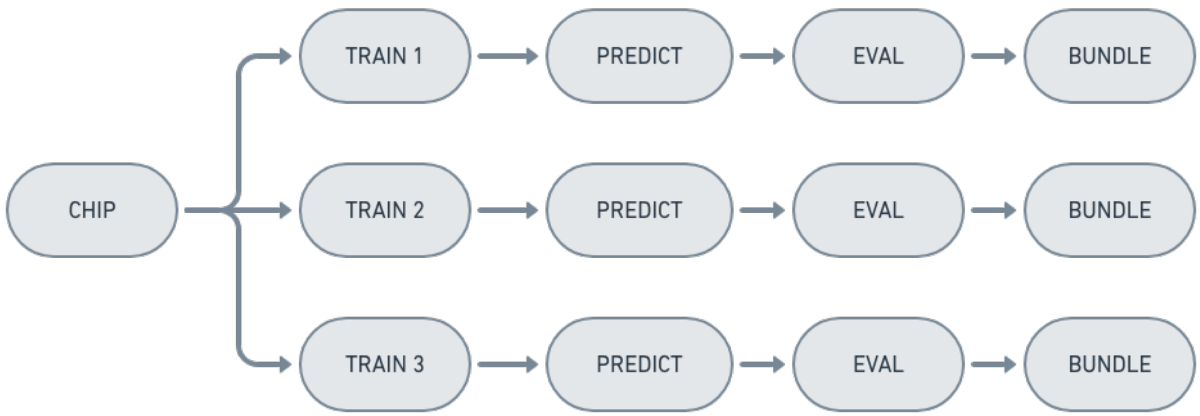
Note: Raster Vision considers two commands to be equal if their inputs, outputs and command types (e.g. rv.CHIP, rv.TRAIN, etc…) are the same. Raster Vision will avoid running multiple of the same command in one run with sameness defined in this way.

During the process of deriving commands from the ExperimentConfigs, each Config object in the experiment has the chance to update itself for a specific command (using the `update_for_command` method), and report what its inputs and outputs are (using the `report_io` method). This is an internal mechanism, so you won't have to dive too deeply into this unless you are a contributor or a plugin author. However, it's good to know that this is when some of the implicit values are set into the configuration. For instance, the `model_uri` property can be set on a `rv.BackendConfig` by using the `with_model_uri` on the builder; however the more standard practice is to let Raster Vision set this property during the `update_for_command` process described above, which it will do based on the `root_uri` of the `ExperimentConfig` as well as other factors.

The base `ExperimentRunner` class constructs a Directed Acyclic Graph (DAG) of the commands based on which commands consume as input other command's outputs, and passes that off to the implementation to be executed. The specific implementation will choose how to actually execute each command.

When an `ExperimentSet` is executed by an `ExperimentRunner`, it is first converted into a `CommandDAG` representing a DAG of commands. In this graph, there is a node for each command, and an edge from X to Y if X produces the input of Y. The commands are then executed according to a topological sort of the graph, so as to respect dependencies between commands.

Two optimizations are performed to eliminate duplicated computation. The first is to only execute commands whose outputs don't exist. The second is to eliminate duplicate nodes that are present when experiments partially overlap, like when an `ExperimentSet` is created with multiple experiments that generate the same chips:



6.2 Running locally

A `rastervision run local ...` command will use the `LocalExperimentRunner`, which builds a Makefile based on the DAG and then executes it on the host machine. This will run multiple experiments in parallel.

6.3 Running on AWS Batch

`rastervision run aws_batch ...` will execute the commands on AWS Batch. This provides a powerful mechanism for running Raster Vision experiment workflows. It allows for queues of CPU and GPU instances to have 0 instances running when not in use. With the running of a single command on your own machine, AWS Batch 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.

The `AWSBatchExperimentRunner` 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 AWS 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 - this includes plugin files.

Note: To run on AWS Batch, you'll need the proper setup. See *Setting up AWS Batch* for instructions.

6.4 Running commands in Parallel

Raster Vision can run certain commands in parallel, such as the *CHIP* and *PREDICT* commands. To do so, use the *–splits* option in the `run` command of the CLI.

Commands implement a `split` method on them, that either returns the original command if they cannot be split, e.g. with training, or a sequence of commands that each do a subset of the work. For instance, using `--splits 5` on a `CHIP` command over 50 training scenes and 25 validation scenes will result in 5 CHIP commands, that can be run in parallel, that will each create chips for 15 scenes.

The command DAG that is given to the experiment 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 submit jobs for each of the command splits, and the Batch Compute Environment will dictate how many resources are available to run Batch jobs simultaneously.

CHAPTER 7

Making Predictions (Inference)

A major focus of Raster Vision is to generate models that can quickly be used to predict, or run inference, on new imagery. To accomplish this, the last step in the chain of commands that comprise an experiment is the `BUNDLE` command, which generates a "predict package". This predict package contains all the necessary model files and configuration to make predictions using the model that was trained by an experiment.

7.1 How to make predictions with models trained by Raster Vision

With a predict package, we can call the *predict* command from the command line client, or use the *Predictor* class to generate predictions from a predict package directly from Python code.

Using the command line tool 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` 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 `Predictor` class is the most flexible way to integrate Raster Vision models into other systems, whether in large PySpark batch jobs or in web servers running on GPU systems.

7.2 Predict Package

The predict package is a zip file containing the model file and the configuration necessary for Raster Vision to use the model. The model file or files are specific to the backend: for Keras, there's a single serialized Keras model file, and for TensorFlow there is the protobuf serialized inference graph. But this is not all that is needed to create predictions. The data that was trained on was potentially processed in specific ways by *Raster Transformers*, and the model could have trained on a subset of bands dictated by the *RasterSource*. We need to know about the *LabelStore* that was used to serialize the predictions to GeoJSON, GeoTIFF, or something else. The prediction logic also needs to know which *Task* was used to apply any transformations that take raw model output and transform it to meaningful predictions.

The predict package 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 experiment that produced the predict package. 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.

CHAPTER 8

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 subcommands, with some top level options:

```
> rastervision --help
 Usage: python -m rastervision [OPTIONS] COMMAND [ARGS]...

 Options:
   -p, --profile TEXT  Sets the configuration profile name to use.
   -v, --verbose       Sets the output to  be verbose.
   --help              Show this message and exit.

 Commands:
   ls           Print out a list of Experiment IDs.
   predict      Make predictions using a predict package.
   run          Run Raster Vision commands against Experiments.
   run_command  Run a command from configuration file.
```

8.1 Commands

8.1.1 run

Run is the main interface into running `ExperimentSet` workflows.

```
> rastervision run --help
Usage: python -m rastervision run [OPTIONS] RUNNER [COMMANDS]...

Run Raster Vision commands from experiments, using the experiment runner
named RUNNER.

Options:
  -e, --experiment_module TEXT  Name of an importable module to look for
```

(continues on next page)

(continued from previous page)

```
                              experiment sets in. If not supplied,
                              experiments will be loaded from __main__
  -p, --path PATTERN          Path of file containing ExprimentSet to run.
  -n, --dry-run               Execute a dry run, which will print out
                              information about the commands to be run, but
                              will not actually run the commands
  -x, --skip-file-check       Skip the step that verifies that file exist.
  -a, --arg KEY VALUE         Pass a parameter to the experiments if the
                              method parameter list takes in a parameter
                              with that key. Multiple args can be supplied
  --prefix PREFIX             Prefix for methods containing experiments.
                              (default: "exp_")
  -m, --method PATTERN        Pattern to match method names to run.
  -f, --filter PATTERN        Pattern to match experiment names to run.
  -r, --rerun                 Rerun commands, regardless if their output
                              files already exist.
  --tempdir TEXT              Temporary directory to use for this run.
  -s, --splits INTEGER        The number of processes to attempt to split
                              each stage into.
  --help                      Show this message and exit.
```

Some specific parameters to call out:

–arg

Use `-a` to pass arguments into the experiment methods; many of which take a `root_uri` which is where Raster Vision will store all the output of the experiment. If you forget to supply an argument, Raster Vision will remind you.

–dry-run

Using the `-n` or `--dry-run` flag is useful to see what you're about to run before you run it. Combine this with the verbose flag for different levels of output:

```
> rastervision run spacenet.chip_classification -a root_uri s3://example/ --dry_run
> rastervision -v run spacenet.chip_classification -a root_uri s3://example/ --dry_run
> rastervision -vv run spacenet.chip_classification -a root_uri s3://example/ --dry_
↪run
```

–skip-file-check

Use `--skip-file-check` or `-x` to avoid checking if files exist, which can take a long time for large experiments. This is useful to do the first run, but if you haven't changed anything about the experiment and are sure the files are there, it's often nice to skip that step.

–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.

8.1.2 predict

Use `predict` to make predictions on new imagery given a *Predict Package*.

```
> rastervision predict --help
Usage: python -m rastervision predict [OPTIONS] PREDICT_PACKAGE IMAGE_URI
                                      OUTPUT_URI

  Make predictions on the image at IMAGE_URI using PREDICT_PACKAGE and store
  the prediciton output at OUTPUT_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
  --export-config PATH  Exports the configuration to the given output file.
  --help                Show this message and exit.
```

8.1.3 ls

The `ls` command very simply lists the IDs of experiments in the given module or file. This functionality is likely to expand to give more information about expriments discovered in a project in later versions.

```
> rastervision ls --help
Usage: python -m rastervision ls [OPTIONS]

  Print out a list of Experiment IDs.

Options:
  -e, --experiment-module TEXT  Name of an importable module to look for
                                experiment sets in. If not supplied,
                                experiments will be loaded from __main__
  -a, --arg KEY VALUE           Pass a parameter to the experiments if the
                                method parameter list takes in a parameter
                                with that key. Multiple args can be supplied
  --help                        Show this message and exit.
```

8.1.4 run_command

The `run_command` is used to run a specific command from a serialized command configuration. This is likely only useful to people writing *ExperimentRunners* that want to run commands remotely from serialzed command JSON.

```
> rastervision run_command --help
Usage: python -m rastervision run_command [OPTIONS] COMMAND_CONFIG_URI

Run a command from a serialized command configuration at
COMMAND_CONFIG_URI.

Options:
--tempdir TEXT
--help          Show this message and exit.
```

CHAPTER 9

Miscellaneous Topics

9.1 FileSystems

The `FileSystem` architecture allows support of multiple file systems through an interface, that is chosen by URI. We currently support the local file system, AWS S3, and HTTP. Some filesystems support read only (HTTP), while others are read/write.

If you need to support other file storage systems, you can add new `FileSystem` classes via the plugin. We're happy to take contributions on new `FileSystem` support if it's generally useful!

9.2 Viewing Tensorboard

The built-in backends will start an instance of TensorBoard while training. 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), <public_dns> is the public DNS of the machine running the training command.

9.3 Model Defaults

Model Defaults allow you to use a single key to set default attributes into backends instead of having to explicitly state them. This is useful for, say, using a key to refer to the pretrained model weights and hyperparameter configuration of various models. Each `Backend` can interpret its model defaults differently. For more information, see the rastervision/backend/model_defaults.json file.

You can set the model defaults to use a different JSON file, so that plugin backends can create model defaults or so that you can override the defaults provided by Raster Vision. See the *RV* Configuration Section for that config value.

Note that model defaults are only used for the Tensorflow-based backends.

9.3.1 TensorFlow Object Detection

This is a list of model defaults for use with the `rv.TF_OBJECT_DETECTION` backend. They come from the TensorFlow Object Detection project, and more information about what each model is can be found in the Tensorflow Object Detection Model Zoo page. These defaults include pretrained model weights and TensorFlow Object Detection `pipeline.conf` templates for the following models:

- `rv.SSD_MOBILENET_V1_COCO`
- `rv.SSD_MOBILENET_V2_COCO`
- `rv.SSDLITE_MOBILENET_V2_COCO`
- `rv.SSD_INCEPTION_V2_COCO`
- `rv.FASTER_RCNN_INCEPTION_V2_COCO`
- `rv.FASTER_RCNN_RESNET50_COCO`
- `rv.RFCN_RESNET101_COCO`
- `rv.FASTER_RCNN_RESNET101_COCO`
- `rv.FASTER_RCNN_INCEPTION_RESNET_V2_ATROUS_COCO`
- `rv.FASTER_RCNN_NAS`
- `rv.MASK_RCNN_INCEPTION_RESNET_V2_ATROUS_COCO`
- `rv.MASK_RCNN_INCEPTION_V2_COCO`
- `rv.MASK_RCNN_RESNET101_ATROUS_COCO`
- `rv.MASK_RCNN_RESNET50_ATROUS_COCO`

9.3.2 Keras Classification

This is a list of model defaults for use with the `rv.KERAS_CLASSIFICATION` backend. Keras Classification only supports one model for now, but more will be added in the future. The pretrained weights come from https://github.com/fchollet/deep-learning-models

- `rv.RESNET50_IMAGENET`

9.3.3 Tensorflow DeepLab

This is a list of model defaults for use with the `rv.TF_DEEPLAB` backend. They come from the TensorFlow DeepLab project, and more information about each model can be found in the Tensorflow DeepLab Model Zoo. These defaults include pretrained model weights and backend configurations for the following models:

- `rv.XCEPTION_65`
- `rv.MOBILENET_V2`

9.4 Reusing models trained by Raster Vision

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 files are listed per backend here:

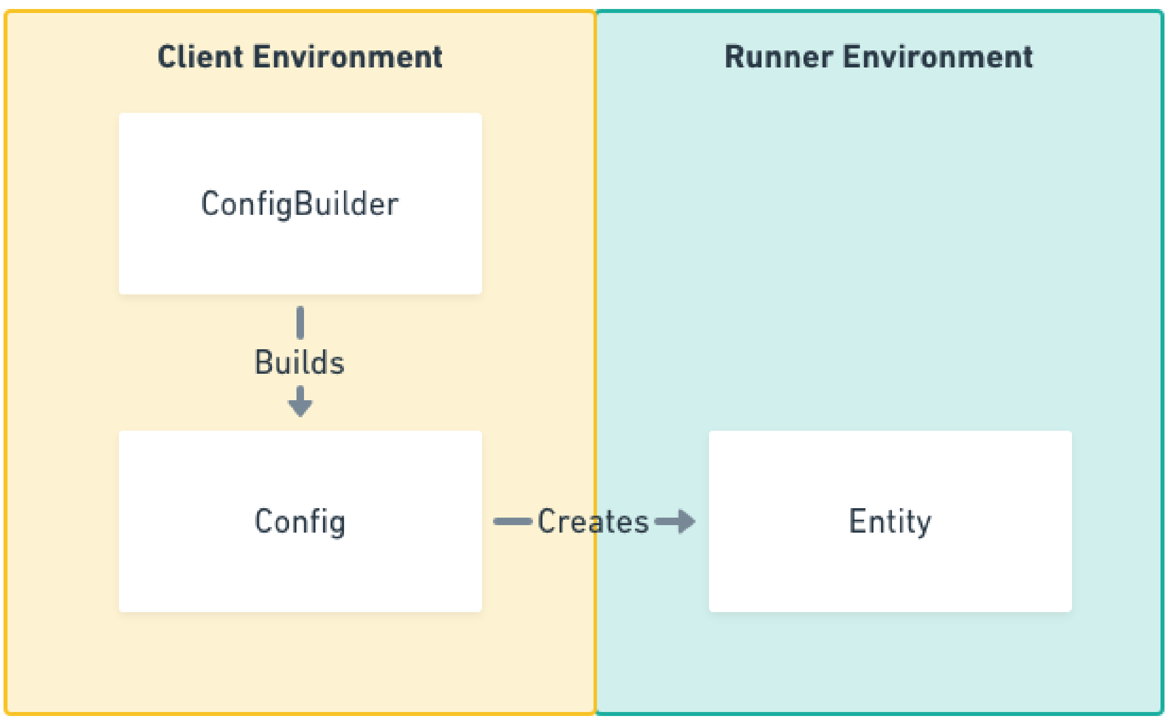
- `rv.PYTORCH_CHIP_CLASSIFICATION`: You can use the `model` file in the train command output as a pretrained model.

- `rv.PYTORCH_SEMANTIC_SEGMENTATION`: You can use the `model` file in the train command output as a pretrained model.
- `rv.PYTORCH_OBJECT_DETECTION`: You can use the `model` file in the train command output as a pretrained model.
- `rv.KERAS_CLASSIFICATION`: You can use the `model_weights.hdf5` file in the train command output as a pretrained model.
- `rv.TF_OBJECT_DETECTION`: Use the `<experiment_id>.tar.gz` that is in the train command output as a pretrained model. The default name of the file is the experiment ID, however you can change the backend configuration to use another name with the `.with_fine_tune_checkpoint_name` method.
- `rv.TF_DEEPLAB`: Use the `<experiment_id>.tar.gz` that is in the TRAIN command output as a pretrained model. The default name of the file is the experiment ID, however you can change the backend configuration to use another name with the `.with_fine_tune_checkpoint_name` method.

CHAPTER 10

Codebase Design Patterns

10.1 Configuration vs Entity



In Raster Vision we keep a separation between configuration of a thing and the creation of the thing itself. This allows us to keep the *client environment*, i.e. the environment that is running the `rastervision` CLI application, and the *runner environment*, i.e. the environment that is actually running commands, totally separate. This means you can install Raster Vision and run experiments on a machine that doesn't have a GPU or any machine learning library installed, but can issue commands to an environment that does. This also lets us work with configuration on the client side very quickly, and leave all the heavy lifting to the runner side.

This separation is expressed in a core design principle that is seen across the codebase: the use of the `Config` and `ConfigBuilder` classes.

10.1.1 Config

The `Config` class represents the configuration of a component of the experiment. It is a declarative encapsulation of exactly what we want to run, without actually running anything. We are able to serialize Configs, and because they describe exactly what we want to do, they become historical artifacts about what happened, messages for running on remote systems, and records that let us repeat experiments and verify results.

The construction of configuration can include some heavy logic, and we want a clean separation from the `Config` and the way we build it. This is why each `Config` has a separate `ConfigBuilder` class.

10.1.2 ConfigBuilder

The `ConfigBuilder` classes are the main interaction point for users of Raster Vision. They are generally instantiated when client code calls the static `.builder()` method on the `Config`. If there are multiple types of builders, a key is used to state which builder should be returned (e.g. with `rv.BackendConfig.builder(rv.KERAS_CLASSIFICATION)`. The usage of keys to return specific builder types allows for two things: 1. a standard interface for constructing builders that only changes based on the parameter passed in, and 2. a way for plugins to register their own keys, so that using plugins feels exactly like using core Raster Vision code.

The ConfigBuilders are immutable data structures that use a *fluent builder pattern*. When you call a method on a builder that sets a property, what you're actually doing is creating a copy of the builder and returning it. Not modifying internal state allows us to fork builders into different transformed objects without having to worry about modifying the internal properties of the builders earlier in the chain of modifications. Using a fluent builder pattern also gives us a readable and standard way of creating and transforming ConfigBuilders and Configs.

The `ConfigBuilder` also has a `.validate()` method that is called whenever `.build()` is called, which gives the `ConfigBuilder` the chance to make sure all required properties are set and are sane. One major advantage of using the `ConfigBuilder` pattern over simply having long `__init__` methods on `Config` objects is that you can set up builders in one part of the code, without setting required properties, and pass it off to another decoupled part of the code that can use the builder further. As long as the required properties are set before `build()` is called, you can set as little or as many properties as you want.

10.2 Fluent Builder Pattern

The ConfigBuilders in Raster Vision use a fluent builder design pattern. This allows the composition and chaining together of transformations on builders, which encourages readable configuration code. The usage of builders is always as follows:

- The `Config` type (`SceneConfig`, `TaskConfig`, etc) will always be available through the top level import (which generally is `import rastervision as rv`)
- The `ConfigBuilder` is created from the static `builder` method on the `Config` class, e.g. `rv.TaskConfig.builder(rv.OBJECT_DETECTION)`. Keys for builder types are also always exposed in the top level package (unless your key is for a custom plugin, in which case you're on your own).

- The builder is then transformed using the *.with_*()* methods. Each call to a *.with_*()* method returns a new copy of the builder with the modifications set, which means you can chain them together. This is the "fluent" part of the fluent builder pattern.
- You call `.build()` when you are ready for your fully baked `Config` object.

You can also call `.to_builder()` on any `Config` object, which lets you move between the `Config` and `ConfigBuilder` space easily. This is useful when you want to take a config that was deserialized or constructed in some other way and use it as a base for further transformation.

10.3 Global Registry

Another major design pattern of Raster Vision is the use of a global registry. This is what gives the ability for the single interface to construct all subclass builders through the static `builder()` method on the `Config` via a key, e.g. `rv.RasterSourceConfig.builder(rv.GEOTIFF_SOURCE)`. The key is used to look up what ConfigBuilders are registered inside the global registery, and the registry determines what builder to return from the `build()` call. More importantly, this enables Raster Vision to have a flexible system to create *Plugins* out of anything that has a keyed `ConfigBuilder`. The registry pattern goes beyond Configs and ConfigBuilders, though: this is also how internal classes and plugins are chosen for *Default Providers*, *ExperimentRunners*, and *FileSystems*.

10.4 Configuration Topics

Configuration objects have a couple of methods that require some understanding if you'd like deeper knowledge of how Raster Vision works - for example if you are creating plugins.

10.4.1 Implicit Configuration

Configuration values can be set implicitly from other configuration. For example, if my backend requires a `model_uri` to save a model to, and it is not set, the configuration may set it to `/opt/data/rv_root/train/experiment-name/model.hdf`. This was implicitly set by knowing the root URI for the train command is `/opt/data/rv_root/train/experiment-name`, which is set on the experiment (by default constructed from the `root_uri` and `experiment_id`). The mechanism that allows this is that configurations implement a method called `update_for_command`, with the following signature:

class rastervision.core.**Config**

update_for_command(*command_type*, *experiment_config*, *context=None*, *io_def=None*)

Updates this configuration for the given command

Note: While configuration is immutable for client facing operations, this is an internal operation and mutates the configuration.

Parameters

- **command_type** – The command type that is currently being preprocessed. experiment_config: The experiment configuration that this configuration is a part of.
- **context** – Optional list of parent configurations, to allow for child configurations contained in collections to understand their context in the experiment configuration.

Returns Nothing. Call should mutate the configuration object itself.

This method is called before running commands on an experiment, and gives the configuration a chance to update any values it needs to based on the experiment and any other context it needs. The context argument is, for example, the `SceneConfig` that the configuration is attached to (e.g. a `RasterSourceConfig`). Context should be set whenever a parent configuration calls `update_for_command` on child configuration, when that parent configuration is part of a collection of configurations (e.g., the collection of `SceneConfig`s in a `DataSetConfig`).

10.4.2 Reporting IO

Raster Vision requires that configuration reports on its input and output files, which allows it to tie together commands into a Directed Acyclic Graph of operations that the `ExperimentRunners` can execute. The way this reporting happens is through the `report_io` method on `Config`.

class `rastervision.core.`**`Config`**

`report_io`(*command_type*, *io_def*)

Updates the given CommandIODefinition.

So that it includes the inputs, outputs, and missing files for this configuration at this command.

Parameters

- **`command_type`** – The command type that is currently being preprocessed.
- **`io_def`** – The CommandIODefinition that this call should modify.

Returns: Nothing. This call should make the appropriate calls to the given io_def to mutate its state.

For each specific command, configuration should set any input files or directories onto the `io_def` through the add `add_input` method, and set any output files or directories using the `add_output` method.

If a configuration does not correctly report on its IO, it could result in commands not running or rerunning happening even though output already exists and the `--rerun` flag is not used. This can be a common pitfall for plugin development, and care should be taken to ensure that IO is properly being reported. The `--dry-run` flag with the `-v` verbosity flag can be useful here for ensuring the IO that is reported is what is expected.

CHAPTER 11

Plugins

You can extend Raster Vision easily by writing Plugins. Any `Config` that is created using the *Fluent Builder Pattern*, that is based on a key (e.g. `rv.BackendConfig.builder(rv.KERAS_CLASSIFICATION)`) can use plugins.

All of the configurable entities that are constructed like this in the Raster Vision codebase use the same sort of registration process as Plugins - the difference is that they are registered internally in the main Raster Vision *Global Registry*. Because of this, the best way to figure out how to build components of Raster Vision that can be plugged in is to study the codebase.

11.1 Creating Plugins

You'll need to implement an interface for the Plugin, by inheriting from `Task`, `Backend`, etc. You will also have to implement a `Config` and `ConfigBuilder` for your type. The `Config` and `ConfigBuilder` should likewise inherit from the appropriate parent class - for example, if you are implementing a backend plugin, you'll need to develop implementations of `Backend`, `BackendConfig`, and `BackendConfigBuilder`. The `__init__` method of `BackendConfig` takes a `backend_type`, which you will have to assign a unique string. This will be the key that you later refer to in your experiment configurations. For instance, if you developed a new backend that passed in the `backend_type = "AWESOME"`, you could reference that backend configuration in an experiment like this:

```
backend = rv.BackendConfig.builder("AWESOME") \
            .with_awesome_property("etc") \
            .build()
```

You'll need to implement the `to_proto` method on the `Config` and the `from_proto` method on `ConfigBuilder`. In the `.proto` files for the entity you are creating a plugin for, you'll see a `google.protobuf.Struct custom_config` section. This is the field in the protobuf that can handle arbitrary JSON, and should be used in plugins for configuration.

Note: Be sure to review the *Configuration Topics* and ensure you're implementing `report_io` and `update_for_command` properly in your configuration.

Note: A common pitfall is implementing the `ConfigBuilder.from_proto` and `Config.to_proto` methods correctly. Look to other `Config` and `ConfigBuilder` implementations in the Raster Vision codebase for examples on how to do this correctly - and utilize the `custom_config` in the protobufs to be able to set arbitrary configuration that is specific to your plugin implementation.

11.2 Registering the Plugin

Your plugin file or module must define a `register_plugin` method with the following signature:

```
def register_plugin(plugin_registry):
    pass
```

The `plugin_registry` that is passed in has a number of methods that allow for registering the plugin with Raster Vision. This is the method that is called on startup of Raster Vision for any plugin configured in the configuration file. See the *Plugin Registry* API reference for more information on registration methods.

11.3 Configuring Raster Vision to use your Plugins

Raster Vision searches for `register_plugin` methods in all the files and modules listed in the Raster Vision configuration. See documentation on the *PLUGINS* section of the configuration for more info on how to set this up.

11.4 Plugins in remote environments

In order for plugins to work with any *ExperimentRunners* that execute commands remotely, the configured files or modules will have to be available to the remote machines. For example, if you are using AWS Batch, then your plugin cannot be something that is only stored on your local machine. In that case, you could store the file in S3 or in a repository that the instances will have access to through HTTP, or you can ensure that the module containing the plugin is also installed in the remote runner environment (e.g. by baking a Docker container based on the Raster Vision container that has your plugins installed, and setting up the AWS Batch job definition to use that container).

Command configurations contain the paths and module names of the plugins they use. This way, the remote environment knows what plugins to load in order to successfully run the commands.

11.5 Example Plugin

```
# easy_evaluator.py

from copy import deepcopy

import rastervision as rv
from rastervision.evaluation import (Evaluator, EvaluatorConfig,
                                     EvaluatorConfigBuilder)
from rastervision.protos.evaluator_pb2 import EvaluatorConfig as EvaluatorConfigMsg

EASY_EVALUATOR = 'EASY_EVALUATOR'
```

(continues on next page)

(continued from previous page)

```
class EasyEvaluator(Evaluator):
    def __init__(self, message):
        self.message

    def process(self, scenes, tmp_dir):
        print(self.message)


class EasyEvaluatorConfig(EvaluatorConfig):
    def __init__(self, message):
        super().__init__(EASY_EVALUATOR)

    def to_proto(self):
        msg = EvaluatorConfigMsg(
            evaluator_type=self.evaluator_type, custom_config={ "message": self.
↪message })
        return msg

    def create_evaluator(self):
        return NoopEvaluator(self.message)

    def update_for_command(self, command_type, experiment_config, context=[]):
        return (self, rv.core.CommandIODefinition())


class NoopEvaluatorConfigBuilder(EvaluatorConfigBuilder):
    def __init__(self, prev=None):
        self.config = {}
        if prev:
            self.config = {
                'message': prev.message
            }

        super().__init__(EasyEvaluatorConfig, {})

    def from_proto(self, msg):
        return self.with_message(msg.custom_config.get("message"))

    def with_message(self, message):
        b = deepcopy(self)
        b.config['message'] = message
        return b


def register_plugin(plugin_registry):
    plugin_registry.register_config_builder(rv.EVALUATOR, NOOP_EVALUATOR,
                                            NoopEvaluatorConfigBuilder)
```

You can set the file location in the path of your Raster Vision plugin configuration in the `files` setting, and then use it in experiments like so (assuming EASY_EVALUATOR was defined the same as above):

```
evaluator = rv.EvaluatorConfig.builder(EASY_EVALUATOR) \
                              .with_message("Great job!") \
                              .build()
```

You could then set this evaluator on an experiment just as you would an internal evaluator.

CHAPTER 12

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.

12.1 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`

CHAPTER 13

Release Process

This is a guide to the process of creating a new release, and is meant for the maintainers of Raster Vision. It describes how to create a new bug fix release, using incrementing from 0.8.0 to 0.8.1 as an example. The process for minor and major releases are somewhat different, and will be documented in the future.

Note: The following instructions assume that Python 3 is the default Python on your local system. Using Python 2 will not work.

13.1 Prepare branch

This assumes that there is already a branch for a minor release called `0.8`. To create a bug fix release (version 0.8.1), we need to backport all the bug fix commits on the `master` branch into the `0.8` branch that have been added since the last bug fix release. For each bug fix PR on `master` we need to create a PR against `0.8` 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]. Our goal is to create and merge each backport PR immediately after each bug fix PR is merged, so hopefully the preceding is already done by the time we are creating a bug fix release.

Make and merge a PR against `0.8` (but not `master`) that increments `version.py` to `0.8.1`. Then wait for the `0.8` branch to be built by Travis 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.

13.2 Make Github release

Using the Github UI, make a new release. Use `0.8.1` as the tag, and `0.8` as the target.

13.3 Make Docker image

The image for `0.8` is created automatically by Travis, but we need to manually create images for `0.8.1`. For this you will need an account on Quay.io under the Azavea organization.

```
docker login quay.io

docker pull quay.io/azavea/raster-vision:cpu-0.8
docker tag quay.io/azavea/raster-vision:cpu-0.8 quay.io/azavea/raster-vision:cpu-0.8.1
docker push quay.io/azavea/raster-vision:cpu-0.8.1

docker pull quay.io/azavea/raster-vision:gpu-0.8
docker tag quay.io/azavea/raster-vision:gpu-0.8 quay.io/azavea/raster-vision:gpu-0.8.1
docker push quay.io/azavea/raster-vision:gpu-0.8.1
```

13.4 Make release on PyPI

Once a release is created on PyPI it can't be deleted, so be careful. 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
```

To create the release distribution, navigate to the `raster-vision` repo on your local filesystem on an up-to-date branch `0.8.`. Then run

```
python setup.py sdist bdist_wheel
```

The contents of the distribution will be in `dist/`. When you are ready to upload to PyPI, run:

```
twine upload dist/*
```

13.5 Announcement

Let people in the Gitter channel know there is a new version.

CHAPTER 14

API Reference

If you are looking for information on a specific function, class, or method, this part of the documentation is for you.

14.1 API Reference

This API documentation is not exhaustive, but covers most of the public API that is important to typical Raster Vision usage.

14.1.1 ExperimentConfigBuilder

An ExperimentConfigBuilder is created by calling

```
rv.ExperimentConfig.builder()
```

class rastervision.experiment.**ExperimentConfigBuilder**(*prev=None*)

build()
Returns the configuration that is built by this builder.

clear_command_uris()
Clears existing command URIs and keys. Useful for re-using experiment configs for new builders.

with_analyze_key(*key*)
Sets the key associated with the analysis stage.

with_analyze_uri(*uri*)
Sets the location where the results of the analysis stage will be stored.

with_analyzer(*analyzer*)
Add an analyzer to be used in the analysis stage.

with_analyzers(*analyzers*)
Add analyzers to be used in the analysis stage.

with_backend(*backend*)
Specifies the backend to be used, e.g. rv.TF_DEEPLAB.

with_bundle_key(*key*)
Sets the key associated with the bundling stage.

with_bundle_uri(*uri*)
Sets the location where the results of the bundling stage will be stored.

with_chip_key(*key*)
Sets the key associated with the "chip" stage.

with_chip_uri(*uri*)
Sets the location where the results of the "chip" stage will be stored.

with_custom_config(*config*)
Sets custom configuration for this experiment. This can be used by plugins such as custom commands.

with_dataset(*dataset*)
Specifies the dataset to be used.

with_eval_key(*key*)
Sets the key associated with the evaluation stage.

with_eval_uri(*uri*)
Sets the location where the results of the evaluation stage will be stored.

with_evaluator(*evaluator*)
Sets the evaluator to use for the evaluation stage.

with_evaluators(*evaluators*)
Sets the evaluators to use for the evaluation stage.

with_id(*id*)
Sets an id for the experiment.

with_predict_key(*key*)
Sets the key associated with the prediction stage.

with_predict_uri(*uri*)
Sets the location where the results of the prediction stage will be stored.

with_root_uri(*uri*)
Sets the root directory where all output will be stored unless subsequently overridden.

with_stats_analyzer()
Add a stats analyzer to be used in the analysis stage.

with_task(*task*)
Sets a specific task type.

Parameters **task** – A TaskConfig object.

with_train_key(*key*)
Sets the key associated with the training stage.

with_train_uri(*uri*)
Sets the location where the results of the training stage will be stored.

14.1.2 DatasetConfigBuilder

A DatasetConfigBuilder is created by calling

```
rv.DatasetConfig.builder()
```

class rastervision.data.**DatasetConfigBuilder**(*prev=None*)

build()
Returns the configuration that is built by this builder.

with_augmentor(*augmentor*)
Sets the data augmentor to be used.

with_augmentors(*augmentors*)
Sets the data augmentors to be used.

with_test_scene(*scene*)
Sets the scene to be used for testing.

with_test_scenes(*scenes*)
Sets the scenes to be used for testing.

with_train_scene(*scene*)
Sets the scene to be used for training.

with_train_scenes(*scenes*)
Sets the scenes to be used for training.

with_validation_scene(*scene*)
Sets the scene to be used for validation.

with_validation_scenes(*scenes*)
Sets the scenes to be used for validation.

14.1.3 TaskConfigBuilder

TaskConfigBuilders are created by calling

```
rv.TaskConfig.builder(TASK_TYPE)
```

Where TASK_TYPE is one of the following:

rv.CHIP_CLASSIFICATION

class rastervision.task.**ChipClassificationConfigBuilder**(*prev=None*)

build()
Returns the configuration that is built by this builder.

with_chip_size(*chip_size*)
Set the chip_size for this task.

Note that some model implementations have a minimum size of input they can handle. A value of > 200 is usually safe.

Parameters **chip_size** – (int) chip size in units of pixels

with_classes(*classes: Union[rastervision.core.class_map.ClassMap, List[str], List[rastervision.protos.class_item_pb2.ClassItem], List[rastervision.core.class_map.ClassItem], Dict[str, int], Dict[str, Tuple[int, str]]]*)
Set the classes for this task.

Parameters **classes** – Either a list of class names, a dict which maps class names to class ids, or a dict which maps class names to a tuple of (class_id, color), where color is a PIL color string.

with_debug(*debug*)
Flag for producing debug products.

with_predict_batch_size(*predict_batch_size*)
Sets the batch size to use during prediction.

with_predict_debug_uri(*predict_debug_uri*)
Set the directory to place prediction debug images

with_predict_package_uri(*predict_package_uri*)
Sets the URI to save a predict package URI to during bundle.

rv.OBJECT_DETECTION

class rastervision.task.**ObjectDetectionConfigBuilder**(*prev=None*)

build()
Returns the configuration that is built by this builder.

with_chip_options(*neg_ratio=1*, *ioa_thresh=0.8*, *window_method='chip'*, *label_buffer=0.0*)
Sets object detection configurations for the Chip command

Parameters

- **neg_ratio** – 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. This option is not used when window_method is *sliding*.
- **ioa_thresh** – 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.
- **window_method** – Different models in the Object Detection API have different inputs. Some models allow variable size inputs so several methods of building training data are required

  Valid values are: - chip (default) - label

  – each label's bounding box is the positive window

  – **image**

    * each image is the positive window

  – **sliding**

    * each image is from a sliding window with 50% overlap

- **label_buffer** – If method is “label”, the positive window can be buffered. If value is >= 0. and < 1., the value is treated as a percentage If value is >= 1., the value is treated in number of pixels

with_chip_size (*chip_size*)
Set the chip_size for this task.

Note that some model implementations have a minimum size of input they can handle. A value of > 200 is usually safe.

Parameters **chip_size** – (int) chip size in units of pixels

with_classes (*classes: Union[rastervision.core.class_map.ClassMap, List[str], List[rastervision.protos.class_item_pb2.ClassItem], List[rastervision.core.class_map.ClassItem], Dict[str, int], Dict[str, Tuple[int, str]]]*)
Set the classes for this task.

Parameters **classes** – Either a list of class names, a dict which maps class names to class ids, or a dict which maps class names to a tuple of (class_id, color), where color is a PIL color string.

with_debug (*debug*)
Flag for producing debug products.

with_predict_batch_size (*predict_batch_size*)
Sets the batch size to use during prediction.

with_predict_debug_uri (*predict_debug_uri*)
Set the directory to place prediction debug images

with_predict_options (*merge_thresh=0.5*, *score_thresh=0.5*)
Prediction options for this task.

Parameters

- **merge_thresh** – 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.
- **score_thresh** – Predicted boxes are only output if their score is above score_thresh.

with_predict_package_uri (*predict_package_uri*)
Sets the URI to save a predict package URI to during bundle.

rv.SEMANTIC_SEGMENTATION

class rastervision.task.**SemanticSegmentationConfigBuilder** (*prev=None*)

build ()
Returns the configuration that is built by this builder.

with_chip_options (*window_method='random_sample'*, *target_classes=None*, *debug_chip_probability=0.25*, *negative_survival_probability=1.0*, *chips_per_scene=1000*, *target_count_threshold=1000*, *stride=None*)
Sets semantic segmentation configurations for the Chip command.

Parameters

- **window_method** – Window method to use for chipping. Options are: random_sample, sliding
- **target_classes** – list of class ids to train model on
- **debug_chip_probability** – probability of generating a debug chip. Applies to the 'random_sample' window method.
- **negative_survival_probability** – probability that a sampled negative chip will be utilized if it does not contain more pixels than target_count_threshold. Applies to the 'random_sample' window method.
- **chips_per_scene** – number of chips to generate per scene. Applies to the 'random_sample' window method.
- **target_count_threshold** – minimum number of pixels covering target_classes that a chip must have. Applies to the 'random_sample' window method.
- **stride** – Stride of windows across image. Defaults to half the chip size. Applies to the 'sliding_window' method.

Returns SemanticSegmentationConfigBuilder

with_chip_size(*chip_size*)
Set the chip_size for this task.

Note that some model implementations have a minimum size of input they can handle. A value of > 200 is usually safe.

Parameters **chip_size** – (int) chip size in units of pixels

with_classes(*classes: Union[rastervision.core.class_map.ClassMap, List[str], List[rastervision.protos.class_item_pb2.ClassItem], List[rastervision.core.class_map.ClassItem], Dict[str, int], Dict[str, Tuple[int, str]]]*)
Set the classes for this task.

Parameters **classes** – Either a list of class names, a dict which maps class names to class ids, or a dict which maps class names to a tuple of (class_id, color), where color is a PIL color string.

with_debug(*debug*)
Flag for producing debug products.

with_predict_batch_size(*predict_batch_size*)
Sets the batch size to use during prediction.

with_predict_chip_size(*chip_size*)
Set the chip_size to use only at prediction time for this task.

Parameters **chip_size** – Integer value chip size

with_predict_debug_uri(*predict_debug_uri*)
Set the directory to place prediction debug images

with_predict_package_uri(*predict_package_uri*)
Sets the URI to save a predict package URI to during bundle.

14.1.4 BackendConfig

There are backends based on PyTorch and Tensorflow. Remember to use the appropriate Docker image depending on the backend. Note that the Tensorflow backends are being sunsetted. BackendConfigBuilders are created by calling

```
rv.BackendConfig.builder(BACKEND_TYPE)
```

Where `BACKEND_TYPE` is one of the following:

rv.PYTORCH_SEMANTIC_SEGMENTATION

class rastervision.backend.pytorch_semantic_segmentation_config.**PyTorchSemanticSegmentatio**

build()
Returns the configuration that is built by this builder.

config_class
alias of `PyTorchSemanticSegmentationConfig`

with_model_defaults(*model_defaults_key*)
Sets the backend configuration and pretrained model defaults according to the model defaults configuration.

with_model_uri(*model_uri*)

with_pretrained_model(*uri*)
Set a pretrained model URI. The filetype and meaning for this model will be different based on the backend implementation.

with_pretrained_uri(*pretrained_uri*)
pretrained_uri should be uri of exported model file.

with_task(*task*)
Sets a specific task type.

Parameters **task** – A TaskConfig object.

with_train_options(*batch_size=8*, *lr=0.0001*, *one_cycle=True*, *num_epochs=5*, *model_arch='resnet50'*, *sync_interval=1*, *debug=False*, *log_tensorboard=True*, *run_tensorboard=True*)
Set options for training models.

Parameters

- **batch_size** – (int) the batch size
- **lr** – (float) the learning rate if using a fixed LR (ie. one_cycle is False), or the maximum LR to use if one_cycle is True
- **one_cycle** – (bool) True if cyclic learning rate scheduler should be used. This cycles the LR once during the course of training and seems to result in a pretty consistent improvement. See lr for more details.
- **num_epochs** – (int) number of epochs (sweeps through training set) to train model for
- **model_arch** – (str) classification model backbone to use for DeepLabV3 architecture. Currently, only Resnet50 works.
- **sync_interval** – (int) sync training directory to cloud every sync_interval epochs.

- **debug** – (bool) if True, save debug chips (ie. visualizations of input to model during training) during training and use single-core for creating minibatches.
- **log_tensorboard** – (bool) if True, write events to Tensorboard log file
- **run_tensorboard** – (bool) if True, run a Tensorboard server at port 6006 that uses the logs generated by the log_tensorboard option

rv.PYTORCH_CHIP_CLASSIFICATION

class rastervision.backend.pytorch_chip_classification_config.**PyTorchChipClassificationCon**

build()
Returns the configuration that is built by this builder.

config_class
alias of PyTorchChipClassificationConfig

with_model_defaults(*model_defaults_key*)
Sets the backend configuration and pretrained model defaults according to the model defaults configuration.

with_model_uri(*model_uri*)

with_pretrained_model(*uri*)
Set a pretrained model URI. The filetype and meaning for this model will be different based on the backend implementation.

with_pretrained_uri(*pretrained_uri*)
pretrained_uri should be uri of exported model file.

with_task(*task*)
Sets a specific task type.

Parameters **task** – A TaskConfig object.

with_train_options(*batch_size=8, lr=0.0001, one_cycle=True, num_epochs=1, model_arch='resnet18', sync_interval=1, debug=False, log_tensorboard=True, run_tensorboard=True*)
Set options for training models.

Parameters

- **batch_size** – (int) the batch size
- **weight_decay** – (float) the weight decay
- **lr** – (float) the learning rate if using a fixed LR (ie. one_cycle is False), or the maximum LR to use if one_cycle is True
- **one_cycle** – (bool) True if cyclic learning rate scheduler should be used. This cycles the LR once during the course of training and seems to result in a pretty consistent improvement. See lr for more details.
- **num_epochs** – (int) number of epochs (sweeps through training set) to train model for
- **model_arch** – (str) Any classification model option in torchvision.models is valid, for example, resnet18.
- **sync_interval** – (int) sync training directory to cloud every sync_interval epochs.

- **debug** – (bool) if True, save debug chips (ie. visualizations of input to model during training) during training and use single-core for creating minibatches.
- **log_tensorboard** – (bool) if True, write events to Tensorboard log file
- **run_tensorboard** – (bool) if True, run a Tensorboard server at port 6006 that uses the logs generated by the log_tensorboard option

rv.PYTORCH_OBJECT_DETECTION

class rastervision.backend.pytorch_object_detection_config.**PyTorchObjectDetectionConfigBui**

Object detection using PyTorch and Faster-RCNN/Resnet50 from torchvision.

build()

Returns the configuration that is built by this builder.

config_class

alias of PyTorchObjectDetectionConfig

with_model_defaults(*model_defaults_key*)

Sets the backend configuration and pretrained model defaults according to the model defaults configuration.

with_model_uri(*model_uri*)

with_pretrained_model(*uri*)

Set a pretrained model URI. The filetype and meaning for this model will be different based on the backend implementation.

with_pretrained_uri(*pretrained_uri*)

pretrained_uri should be uri of exported model file.

with_task(*task*)

Sets a specific task type.

Parameters **task** – A TaskConfig object.

with_train_options(*batch_size=8*, *lr=0.0001*, *one_cycle=True*, *num_epochs=5*, *model_arch='resnet18'*, *sync_interval=1*, *log_tensorboard=True*, *run_tensorboard=True*, *debug=False*)

Set options for training models.

Parameters

- **batch_size** – (int) the batch size
- **lr** – (float) the learning rate if using a fixed LR (ie. one_cycle is False), or the maximum LR to use if one_cycle is True
- **one_cycle** – (bool) True if cyclic learning rate scheduler should be used. This cycles the LR once during the course of training and seems to result in a pretty consistent improvement. See lr for more details.
- **num_epochs** – (int) number of epochs (sweeps through training set) to train model for
- **model_arch** – (str) classification model backbone to use. Any Resnet option in torchvision.models is valid, for example, resnet18.
- **sync_interval** – (int) sync training directory to cloud every sync_interval epochs.
- **log_tensorboard** – (bool) if True, write events to Tensorboard log file

- **run_tensorboard** – (bool) if True, run a Tensorboard server at port 6006 that uses the logs generated by the log_tensorboard option
- **debug** – (bool) if True, save debug chips (ie. visualizations of input to model during training) during training and use single-core for creating minibatches.

rv.KERAS_CLASSIFICATION

class rastervision.backend.keras_classification_config.**KerasClassificationConfigBuilder**(*pre*

build()
Build this configuration.

with_batch_size(*batch_size*)
Sets the training batch size.

with_config(*config_mod*, *ignore_missing_keys=False*, *set_missing_keys=False*)
Modify the backend configuration.

Given a dict, modify the tensorflow pipeline configuration such that keys that are found recursively in the configuration are replaced with those values. TODO: better explanation.

with_debug(*debug*)
Sets the debug flag for this backend.

with_model_defaults(*model_defaults_key*)
Sets the backend configuration and pretrained model defaults according to the model defaults configuration.

with_model_uri(*model_uri*)
Sets the filename of the trained model.

with_num_epochs(*num_epochs*)
Sets the number of training epochs.

with_pretrained_model(*uri*)
Set a pretrained model URI. The filetype and meaning for this model will be different based on the backend implementation.

with_task(*task*)
Sets a specific task type.

Parameters **task** – A TaskConfig object.

with_template(*template*)
Use a template as the base for configuring Keras Classification.

Parameters **template** – dict, string or uri

with_train_options(*sync_interval=600*, *do_monitoring=True*, *replace_model=False*)
Sets the train options for this backend.

Parameters

- **sync_interval** – How often to sync output of training to the cloud (in seconds).
- **do_monitoring** – Run process to monitor training (eg. Tensorboard)
- **replace_model** – Replace the model checkpoint if exists. If false, this will continue training from the checkpoint if it exists, if the backend allows for this.

with_training_data_uri (*training_data_uri*)
Whence comes the training data?

Parameters **training_data_uri** – The location of the training data.

with_training_output_uri (*training_output_uri*)
Whither goes the training output?

Parameters **training_output_uri** – The location where the training output will be stored.

rv.TF_OBJECT_DETECTION

class rastervision.backend.tf_object_detection_config.**TFObjectDetectionConfigBuilder** (*prev=N*

build ()
Build this configuration.

Set any values into the TF object detection pipeline config as necessary.

with_batch_size (*batch_size*)
Sets the training batch size.

with_config (*config_mod*, *ignore_missing_keys=False*, *set_missing_keys=False*)
Given a dict, modify the tensorflow pipeline configuration.

Modify it such that keys that are found recursively in the configuration are replaced with those values. TODO: better explanation.

with_debug (*debug*)
Sets the debug flag for this backend.

with_fine_tune_checkpoint_name (*fine_tune_checkpoint_name*)
Defines the name of the fine tune checkpoint for this model.

with_model_defaults (*model_defaults_key*)
Sets the backend configuration and pretrained model defaults according to the model defaults configuration.

with_model_uri (*model_uri*)
Defines the name of the model file that will be created for this model after training.

with_num_steps (*num_steps*)
Sets the number of training steps.

with_pretrained_model (*uri*)
Set a pretrained model URI. The filetype and meaning for this model will be different based on the backend implementation.

with_script_locations (*model_main_uri='/opt/tf-models/object_detection/model_main.py'*, *export_uri='/opt/tf-models/object_detection/export_inference_graph.py'*)

with_task (*task*)
Sets a specific task type.

Parameters **task** – A TaskConfig object.

with_template (*template*)
Use a template for TF Object Detection pipeline config.

Parameters **template** – A dict, string or uri as the base for the TF Object Detection API model training pipeline, for example those found here: https://github.com/tensorflow/models/tree/eef6bb5bd3b3cd5fcf54306bf29750b7f9f9a5ea/research/object_detection/samples/configs # noqa

with_train_options (*sync_interval=600*, *do_monitoring=True*, *replace_model=False*)
Sets the train options for this backend.

Parameters

- **sync_interval** – How often to sync output of training to the cloud (in seconds).
- **do_monitoring** – Run process to monitor training (eg. Tensorboard)
- **replace_model** – Replace the model checkpoint if exists. If false, this will continue training from checkpoing if exists, if the backend allows for this.

with_training_data_uri (*training_data_uri*)
Whence comes the training data?

Parameters **training_data_uri** – The location of the training data.

with_training_output_uri (*training_output_uri*)
Whither goes the training output?

Parameters **training_output_uri** – The location where the training output will be stored.

rv.TF_DEEPLAB

class rastervision.backend.tf_deeplab_config.**TFDeeplabConfigBuilder** (*prev=None*)

build ()
Build this configuration.

with_batch_size (*batch_size*)
Sets the training batch size.

with_config (*config_mod*, *ignore_missing_keys=False*, *set_missing_keys=False*)
Given a dict, modify the tensorflow pipeline configuration.

Modify it such that keys that are found recursively in the configuration are replaced with those values.

with_debug (*debug*)
Sets the debug flag for this backend.

with_fine_tune_checkpoint_name (*fine_tune_checkpoint_name*)
Sets the name of the fine tune checkpoint for the model.

with_model_defaults (*model_defaults_key*)
Sets the backend configuration and pretrained model defaults according to the model defaults configuration.

with_model_uri (*model_uri*)
Sets the filename for the model that will be trained.

with_num_clones (*num_clones*)
Sets the number of clones (useful for multi-GPU training).

with_num_steps (*num_steps*)
Sets the number of training steps.

with_pretrained_model(*uri*)
Set a pretrained model URI. The filetype and meaning for this model will be different based on the backend implementation.

with_script_locations(*train_py='/opt/tf-models/deeplab/train.py'*, *export_py='/opt/tf-models/deeplab/export_model.py'*, *eval_py='/opt/tf-models/deeplab/eval.py'*)

with_task(*task*)
Sets a specific task type.

Parameters **task** – A TaskConfig object.

with_template(*template*)
Use a TFDL config template from dict, string or uri.

with_train_options(*train_restart_dir=None*, *sync_interval=600*, *do_monitoring=True*, *replace_model=False*, *do_eval=False*)
Sets the train options for this backend.

Parameters

- **sync_interval** – How often to sync output of training to the cloud (in seconds).
- **do_monitoring** – Run process to monitor training (eg. Tensorboard)
- **replace_model** – Replace the model checkpoint if exists. If false, this will continue training from checkpoint if exists, if the backend allows for this.
- **do_eval** – Boolean determining whether to run the eval script.

with_training_data_uri(*training_data_uri*)
Whence comes the training data?

Parameters **training_data_uri** – The location of the training data.

with_training_output_uri(*training_output_uri*)
Whither goes the training output?

Parameters **training_output_uri** – The location where the training output will be stored.

14.1.5 SceneConfig

SceneConfigBuilders are created by calling

```
rv.SceneConfig.builder()
```

class rastervision.data.**SceneConfigBuilder**(*prev=None*)

build()
Returns the configuration that is built by this builder.

clear_aois()
Clears the AOIs for this scene

clear_label_source()
Clears the label source for this scene

clear_label_store()
Clears the label store for this scene

with_aoi_uri (*uri*)
Sets the Area of Interest for the scene.

Parameters **uri** – a URI of a GeoJSON file with polygons.

with_aoi_uris (*uris*)
Sets the areas of interest for the scene.

Parameters **uris** – List of URIs each to a GeoJSON file with polygons.

with_id (*id*)
Sets an id for the scene.

with_label_source (*label_source: Union[str, rastervision.data.label_source.label_source_config.LabelSourceConfig]*)
Sets the raster source for this scene.

Parameters **label_source** – Can either be a label source configuration, or a string. If a string, the registry will be queried to grab the default LabelSourceConfig for the string.

Note: A task must be set with *with_task* before calling this, if calling with a string.

with_label_store (*label_store: Union[str, rastervision.data.label_store.label_store_config.LabelStoreConfig, None] = None*)
Sets the raster store for this scene.

Parameters **label_store** – Can either be a label store configuration, or a string, or None. If a string, the registry will be queried to grab the default LabelStoreConfig for the string. If None, then the default for the task from the regsitry will be used.

Note: A task must be set with *with_task* before calling this, if calling with a string.

with_raster_source (*raster_source: Union[str, rastervision.data.raster_source.raster_source_config.RasterSourceConfig], channel_order=None*)
Sets the raster source for this scene.

Parameters

- **raster_source** – Can either be a raster source configuration, or a string. If a string, the registry will be queried to grab the default RasterSourceConfig for the string.
- **channel_order** – Optional channel order for this raster source.

with_task (*task*)
Sets a specific task type, e.g. rv.OBJECT_DETECTION.

14.1.6 RasterSourceConfig

RasterSourceConfigBuilders are created by calling

```
rv.RasterSourceConfig.builder(SOURCE_TYPE)
```

Where `SOURCE_TYPE` is one of the following:

rv.RASTERIO_SOURCE

class rastervision.data.**RasterioSourceConfigBuilder**(*prev=None*)
This RasterSource can read any file that can be opened by Rasterio/GDAL.

This includes georeferenced formats such as GeoTIFF and non-georeferenced formats such as JPG. See https://www.gdal.org/formats_list.html for more details.

build()
Returns the configuration that is built by this builder.

with_channel_order(*channel_order*)
Defines the channel order for this raster source.

This defines the subset of channel indices and their order to use when extracting chips from raw imagery.

Parameters **channel_order** – list of channel indices

with_shifts(*x, y*)
Set the x- and y-shifts in meters.

This will only have an effect on georeferenced imagery.

Parameters

- **x** – A number of meters to shift along the x-axis. A positive shift moves the "camera" to the right.
- **y** – A number of meters to shift along the y-axis. A positive shift moves the "camera" down.

with_stats_transformer()
Add a stats transformer to the raster source.

with_transformer(*transformer*)
A transformer to be applied to the raster data.

Parameters **transformer** – A transformer to apply to the raster data.

with_transformers(*transformers*)
Transformers to be applied to the raster data.

Parameters **transformers** – A list of transformers to apply to the raster data.

with_uri(*uri*)
Set URI for raster files that can be read by Rasterio.

with_uris(*uris*)
Set URIs for raster files that can be read by Rasterio.

rv.RASTERIZED_SOURCE

class rastervision.data.**RasterizedSourceConfigBuilder**(*prev=None*)

build()
Returns the configuration that is built by this builder.

with_channel_order(*channel_order*)
Defines the channel order for this raster source.

This defines the subset of channel indices and their order to use when extracting chips from raw imagery.

Parameters **channel_order** – list of channel indices

with_rasterizer_options (*background_class_id*, *all_touched=False*)
Specify options for converting GeoJSON to raster.

Parameters

- **background_class_id** – The class_id to use for background pixels that don't overlap with any shapes in the GeoJSON file.
- **all_touched** – 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).

with_stats_transformer ()
Add a stats transformer to the raster source.

with_transformer (*transformer*)
A transformer to be applied to the raster data.

Parameters **transformer** – A transformer to apply to the raster data.

with_transformers (*transformers*)
Transformers to be applied to the raster data.

Parameters **transformers** – A list of transformers to apply to the raster data.

with_uri (*uri*)

with_vector_source (*vector_source*)
Set the vector_source.

Parameters **vector_source** (*str or VectorSource*) – a URI and use the default provider to construct a VectorSource.

14.1.7 LabelSourceConfig

LabelSourceConfigBuilders are created by calling

```
rv.LabelSourceConfig.builder(SOURCE_TYPE)
```

Where `SOURCE_TYPE` is one of the following:

rv.CHIP_CLASSIFICATION

class rastervision.data.**ChipClassificationLabelSourceConfigBuilder** (*prev=None*)

build ()
Returns the configuration that is built by this builder.

with_background_class_id (*background_class_id*)
Sets the background class ID.

Optional class_id to use as the background class; ie. the one that is used when a window contains no boxes. If not set, empty windows have None set as their class_id.

with_cell_size (*cell_size*)
Sets the cell size of the chips.

If not explicitly set, the chip size will be used if this object is created as part of an experiment.

Parameters **cell_size** – (int) the size of the cells in units of pixels

with_infer_cells(*infer_cells*)
Set if this label source should infer cells.

If true, the label source will infer the cell polygon and label from the polygons in the vector_source. If the labels are already cells and properly labeled, this can be False.

with_ioa_thresh(*ioa_thresh*)
The minimum IOA of a polygon and cell.

with_pick_min_class_id(*pick_min_class_id*)
Set this label source to pick min class ID

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.

with_uri(*uri*)

with_use_intersection_over_cell(*use_intersection_over_cell*)
Set this label source to use intersection over cell or not.

If use_intersection_over_cell is true, then use the area of the cell as the denominator in the IOA. Otherwise, use the area of the polygon.

with_vector_source(*vector_source*)
Set the vector_source.

Parameters **vector_source** (`str or VectorSource`) – a URI and use the default provider to construct a VectorSource.

rv.OBJECT_DETECTION

class rastervision.data.**ObjectDetectionLabelSourceConfigBuilder**(*prev=None*)

build()
Returns the configuration that is built by this builder.

with_uri(*uri*)

with_vector_source(*vector_source*)
Set the vector_source.

Parameters **vector_source** (`str or VectorSource`) – a URI and use the default provider to construct a VectorSource.

rv.SEMANTIC_SEGMENTATION

class rastervision.data.**SemanticSegmentationLabelSourceConfigBuilder**(*prev=None*)

build()
Returns the configuration that is built by this builder.

with_raster_source(*source*, *channel_order=None*)
Set raster_source.

Parameters **source** – (RasterSourceConfig) A RasterSource assumed to have RGB values that are mapped to class_ids using the rgb_class_map.

Returns SemanticSegmentationLabelSourceConfigBuilder

with_rgb_class_map(*rgb_class_map*)
Set rgb_class_map.

Parameters **rgb_class_map** – (something accepted by ClassMap.construct_from) a class map with color values used to map RGB values to class ids

Returns SemanticSegmentationLabelSourceConfigBuilder

14.1.8 VectorSourceConfig

VectorSourceConfigBuilders are created by calling

```
rv.VectorSourceConfig.builder(SOURCE_TYPE)
```

Where `SOURCE_TYPE` is one of the following:

rv.GEOJSON_SOURCE

class rastervision.data.**GeoJSONVectorSourceConfigBuilder**(*prev=None*)

build()
Returns the configuration that is built by this builder.

with_buffers(*line_bufs=None*, *point_bufs=None*)
Set options for buffering lines and points into polygons.

For example, this is useful for buffering lines representing roads so that their width roughly matches the width of roads in the imagery.

Parameters

- **line_bufs** – (dict or None) If none, uses default buffer value of 1. Otherwise, a map from class_id to number of pixels to buffer by. If the buffer value is None, then no buffering will be performed and the LineString or Point 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.
- **point_bufs** – (dict or None) same as above, but used for buffering Points into Polygons.

with_class_inference(*class_id_to_filter=None*, *default_class_id=1*)
Set options for inferring the class of each feature.

For more info on how class inference works, see ClassInference.infer_class()

Parameters

- **class_id_to_filter** – (dict) map from class_id to JSON filter. The filter schema is according to https://github.com/mapbox/mapbox-gl-js/blob/c9900db279db776f493ce8b6749966cedc2d6b8a/src/style-spec/feature_filter/index.js # noqa
- **default_class_id** – (int) the default class_id to use if class can't be inferred using other mechanisms. If a feature defaults to a class_id of None, then that feature will be deleted.

with_uri(*uri*)

rv.VECTOR_TILE_SOURCE

class rastervision.data.**VectorTileVectorSourceConfigBuilder**(*prev=None*)

build()
Returns the configuration that is built by this builder.

with_buffers(*line_bufs=None*, *point_bufs=None*)
Set options for buffering lines and points into polygons.

For example, this is useful for buffering lines representing roads so that their width roughly matches the width of roads in the imagery.

Parameters

- **line_bufs** – (dict or None) If none, uses default buffer value of 1. Otherwise, a map from class_id to number of pixels to buffer by. If the buffer value is None, then no buffering will be performed and the LineString or Point 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.
- **point_bufs** – (dict or None) same as above, but used for buffering Points into Polygons.

with_class_inference(*class_id_to_filter=None*, *default_class_id=1*)
Set options for inferring the class of each feature.

For more info on how class inference works, see ClassInference.infer_class()

Parameters

- **class_id_to_filter** – (dict) map from class_id to JSON filter. The filter schema is according to https://github.com/mapbox/mapbox-gl-js/blob/c9900db279db776f493ce8b6749966cedc2d6b8a/src/style-spec/feature_filter/index.js # noqa
- **default_class_id** – (int) the default class_id to use if class can't be inferred using other mechanisms. If a feature defaults to a class_id of None, then that feature will be deleted.

with_id_field(*id_field='@id'*)
Set the name of the id field.

Parameters **id_field** – (str) name of field in feature['properties'] that contains the feature's unique id. Used for merging features that are split across tile boundaries.

with_uri(*uri*)
Set the URI of the vector tiles.

Parameters **uri** – (str) URI of vector tile endpoint. Should either contain {z}/{x}/{y} or point to .mbtiles file.

with_zoom(*zoom*)
Set the zoom level to use when accessing vector tiles.

Note: the vector tiles need to support the zoom level. Typically only a subset of zoom levels are supported.

14.1.9 LabelStoreConfig

LabelStoreConfigBuilders are created by calling

```
rv.LabelStoreConfig.builder(STORE_TYPE)
```

Where `STORE_TYPE` is one of the following:

rv.CHIP_CLASSIFICATION_GEOJSON

class rastervision.data.**ChipClassificationGeoJSONStoreConfigBuilder**(*prev=None*)

build()
Returns the configuration that is built by this builder.

with_uri(*uri*)
Set URI for a GeoJSON used to read/write predictions.

For `rv.OBJECT_DETECTION`:

rv.OBJECT_DETECTION_GEOJSON

class rastervision.data.**ObjectDetectionGeoJSONStoreConfigBuilder**(*prev=None*)

build()
Returns the configuration that is built by this builder.

with_uri(*uri*)
Set URI for a GeoJSON used to read/write predictions.

rv.SEMANTIC_SEGMENTATION_RASTER

class rastervision.data.**SemanticSegmentationRasterStoreConfigBuilder**(*prev=None*)

build()
Returns the configuration that is built by this builder.

with_rgb(*rgb*)
Set flag for writing RGB data using the class map.

Otherwise this method will write the class ID into a single band.

with_uri(*uri*)
Set URI for a GeoTIFF used to read/write predictions.

with_vector_output(*vector_output*)
Configure vector output for predictions.

Parameters vector_output – Either a list of dictionaries or a protobuf object. The dictionary or the object contain (respectively) keys (attributes) called 'denoise', 'uri', 'class_id', and 'mode'. The value associated with the 'denoise' key specifies the radius of the structural element used to perform a low-pass filtering process on the mask (see https://en.wikipedia.org/wiki/Mathematical_morphology#Opening). The value associated with the 'uri' key is either a file where the GeoJSON prediction will be written, or "" indicating that the filename should be auto-generated. 'class_id' is the integer prediction class that is of interest. The 'mode' key must be set to 'buildings' or 'polygons'.

14.1.10 RasterTransformerConfig

RasterTransformerConfigBuilders are created by calling

```
rv.RasterTransformerConfig.builder(TRANSFORMER_TYPE)
```

Where `TRANSFORMER_TYPE` is one of the following:

rv.STATS_TRANSFORMER

class rastervision.data.**StatsTransformerConfigBuilder**(*prev=None*)

build()
Returns the configuration that is built by this builder.

with_stats_uri(*stats_uri*)
Set the stats_uri.

Parameters **stats_uri** – URI to the stats json to use

14.1.11 AugmentorConfig

AugmentorConfigBuilders are created by calling

```
rv.AugmentorConfig.builder(AUGMENTOR_TYPE)
```

Where `AUGMENTOR_TYPE` is one of the following:

rv.NODATA_AUGMENTOR

class rastervision.augmentor.**NodataAugmentorConfigBuilder**(*prev=None*)

build()
Returns the configuration that is built by this builder.

with_probability(*aug_prob*)
Sets the probability for this augmentation.

Determines how probable this augmentation will happen to negative chips.

Parameters **aug_prob** – Float value between 0.0 and 1.0

14.1.12 AnalyzerConfig

AnalyzerConfigBuilders are created by calling

```
rv.AnalyzerConfig.builder(ANALYZER_TYPE)
```

Where `ANALYZER_TYPE` is one of the following:

rv.STATS_ANALYZER

class rastervision.analyzer.**StatsAnalyzerConfigBuilder**(*prev=None*)

build()
Returns the configuration that is built by this builder.

with_sample_prob(*sample_prob*)
Set the sample_prob used to sample a subset of each scene.

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 **sample_prob** – (float or None) between 0 and 1

with_stats_uri(*stats_uri*)
Set the stats_uri.

Parameters **stats_uri** – URI to the stats json to use

14.1.13 EvaluatorConfig

EvaluatorConfigBuilders are created by calling

```
rv.EvaluatorConfig.builder(Evaluator_TYPE)
```

Where `Evaluator_TYPE` is one of the following:

rv.CHIP_CLASSIFICATION_EVALUATOR

class rastervision.evaluation.**ChipClassificationEvaluatorConfigBuilder**(*prev=None*)

build()
Returns the configuration that is built by this builder.

with_class_map(*class_map*)
Set the class map to be used for evaluation.

Parameters **class_map** – The class map to be used

with_output_uri(*output_uri*)
Set the output_uri.

Parameters **output_uri** – URI to the stats json to use

with_task(*task*)
Sets a specific task type, e.g. rv.OBJECT_DETECTION.

with_vector_output_uri(*vector_output_uri*)
Set the vector_output_uri.

Parameters **vector_output_uri** – URI to the vector stats json to use

rv.OBJECT_DETECTION_EVALUATOR

class rastervision.evaluation.**ObjectDetectionEvaluatorConfigBuilder**(*prev=None*)

build()
Returns the configuration that is built by this builder.

with_class_map(*class_map*)
Set the class map to be used for evaluation.

Parameters **class_map** – The class map to be used

with_output_uri(*output_uri*)
Set the output_uri.

Parameters **output_uri** – URI to the stats json to use

with_task(*task*)
Sets a specific task type, e.g. rv.OBJECT_DETECTION.

with_vector_output_uri(*vector_output_uri*)
Set the vector_output_uri.

Parameters **vector_output_uri** – URI to the vector stats json to use

rv.SEMANTIC_SEGMENTATION_EVALUATOR

class rastervision.evaluation.**SemanticSegmentationEvaluatorConfigBuilder**(*prev=None*)

build()
Returns the configuration that is built by this builder.

with_class_map(*class_map*)
Set the class map to be used for evaluation.

Parameters **class_map** – The class map to be used

with_output_uri(*output_uri*)
Set the output_uri.

Parameters **output_uri** – URI to the stats json to use

with_task(*task*)
Sets a specific task type, e.g. rv.OBJECT_DETECTION.

with_vector_output_uri(*vector_output_uri*)
Set the vector_output_uri.

Parameters **vector_output_uri** – URI to the vector stats json to use

14.1.14 Aux Commands

class rastervision.command.aux.**CogifyCommand**(*command_config*)
Turns a GDAL-readable raster into a Cloud Optimized GeoTiff.

Configuration:

uris: A list of tuples of (src_path, dest_path) where dest_path is the COG URI.

block_size: The tile size for the COG. Defaults to 512.

resample_method: The resample method to use for overviews. Defaults to 'near'.

compression: The compression method to use. Defaults to 'deflate'. Use 'none' for no compression.

overviews: The overview levels to create. Defaults to [2, 4, 8, 16, 32]

14.1.15 Aux Command Options

class rastervision.command.aux_command.**AuxCommandOptions**(*split_on=None, inputs=<function AuxCommandOptions.<lambda>>, outputs=<function AuxCommandOptions.<lambda>>, include_by_default=False, required_fields=None*)

__init__(*split_on=None, inputs=<function AuxCommandOptions.<lambda>>, outputs=<function AuxCommandOptions.<lambda>>, include_by_default=False, required_fields=None*)

Instantiate an AuxCommandOptions object.

Parameters

- **split_on** (*str*) – The property of the configuration to use when splitting.
- **configuration at this property must be a list.** (*The*) –
- **inputs** – A function that, given the configuration, returns a list of
- **that are inputs into the command. Along with outputs, this allows** (*URIs*) –
- **Vision to correctly determine if there are any missing inputs, or** (*Raster*) –
- **the command has already been run. It will also allow the command to** (*if*) –
- **run in the right sequence if run with other commands that will produce** (*be*) –
- **command's inputs as their outputs.** (*this*) –
- **outputs** – A function that, given the configuration, returns a list of
- **that are outputs of the command. See the details on inputs.** (*URIs*) –
- **include_by_default** – Set this to True if you want this command to run
- **default, meaning it will run every time no specific commands are issued** (*by*) –
- **the command line** (*on*) –
- **required_fields** – Set this to properties of the configuration that are

- **If the user of the command does not set values into those** (*required.*) –
- **properties, an error will be thrown at configuration building** (*configuration*) –
- **time.** –

14.1.16 Predictor

class rastervision.**Predictor**(*prediction_package_uri*, *tmp_dir*, *update_stats=False*, *channel_order=None*)

Class for making predictions based off of a prediction package.

__init__(*prediction_package_uri*, *tmp_dir*, *update_stats=False*, *channel_order=None*)

Creates a new Predictor.

Parameters

- **prediction_package_uri** – The URI of the prediction package to use. Can be any type of URI that Raster Vision can read.
- **tmp_dir** – Temporary directory in which to store files that are used by the Predictor. This directory is not cleaned up by this class.
- **update_stats** – Option indicating if any Analyzers should be run on the image to be predicted on. Otherwise, the Predictor will use the output of Analyzers that are bundled with the predict package. This is useful, for instance, if you are predicting against imagery that needs to be normalized with a StatsAnalyzer, and the color profile of the new imagery is significantly different then the imagery the model was trained on.
- **channel_order** – 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.

load_model()

Load the model for this Predictor.

This is useful if you are going to make multiple predictions with the model, and want it to be fast on the first prediction.

Note: This is called implicitly on the first call of 'predict' if it hasn't been called already.

predict(*image_uri*, *label_uri=None*, *config_uri=None*)

Generate predictions for the given image.

Parameters

- **image_uri** – URI of the image to make predictions against. This can be any type of URI readable by Raster Vision FileSystems.
- **label_uri** – Optional URI to save labels off into.
- **config_uri** – Optional URI in which to save the bundle_config, which can be useful to client applications for understanding how to interpret the labels.
- **Returns** – rastervision.data.labels.Labels containing the predicted labels.

14.1.17 Plugin Registry

class rastervision.plugin.**PluginRegistry**(*plugin_config*, *rv_home*)

register_aux_command(*command_type*, *command_class*)
Registers a custom AuxCommand as a plugin.

Parameters

- **– The key used for this plugin. This will be used to** (*command_type*) – construct the builder in a ".builder(key)" call.
- **– The subclass of AuxCommand subclass to register.** (*command_class*) –

register_command_config_builder(*command_type*, *builder_class*)
Registers a ConfigBuilder as a plugin.

Parameters

- **– The key used for this plugin. This will be used to** (*command_type*) – construct the builder in a ".builder(key)" call.
- **– The subclass of CommandConfigBuilder that builds** (*builder_class*) – the CommandConfig for this plugin.

register_config_builder(*group*, *key*, *builder_class*)
Registers a ConfigBuilder as a plugin.

Parameters

- **– The Config group, e.g. rv.BACKEND, rv.TASK.** (*group*) –
- **– The key used for this plugin. This will be used to** (*key*) – construct the builder in a ".builder(key)" call.
- **– The subclass of ConfigBuilder that builds** (*builder_class*) – the Config for this plugin.

register_default_evaluator(*provider_class*)
Registers an EvaluatorDefaultProvider for use as a plugin.

register_default_label_source(*provider_class*)
Registers a LabelSourceDefaultProvider for use as a plugin.

register_default_label_store(*provider_class*)
Registers a LabelStoreDefaultProvider for use as a plugin.

register_default_raster_source(*provider_class*)
Registers a RasterSourceDefaultProvider for use as a plugin.

register_default_vector_source(*provider_class*)
Registers a VectorSourceDefaultProvider for use as a plugin.

register_experiment_runner(*runner_key*, *runner_class*)
Registers an ExperimentRunner as a plugin.

Parameters

- **– The key used to reference this plugin runner.** (*runner_key*) – This is a string that will match the command line argument used to reference this runner; e.g. if the key is "FOO_RUNNER", then users can use the runner by issuing a "rastervision run foo_runner . . . " command.

- **– The class of the ExperimentRunner plugin.** (*runner_class*) –

register_filesystem (*filesystem_class*)
Registers a FileSystem as a plugin.

CHAPTER 15

CHANGELOG

15.1 CHANGELOG

15.1.1 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 *BackendConfig*

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

15.1.2 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 `ConfigBuilders`. #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

15.1.3 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