
Raster Vision Documentation

Release 0.8.0

Azavea

Apr 12, 2019

Contents

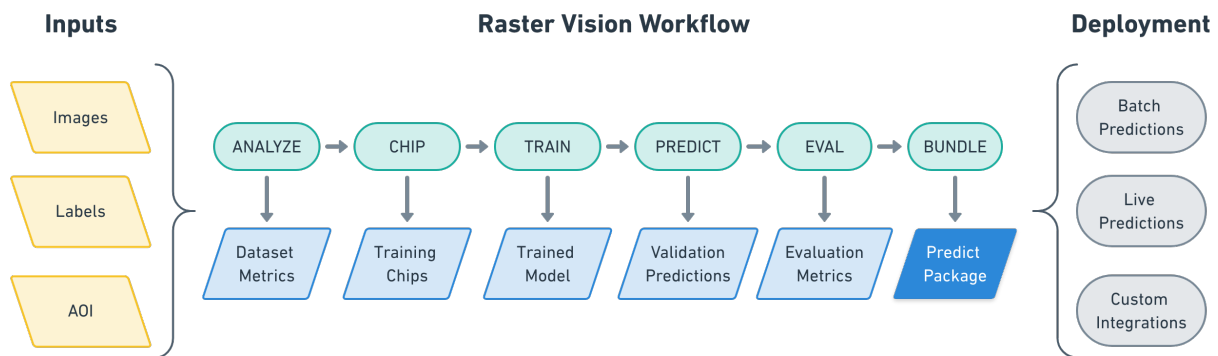
| | | |
|----------|---|-----------|
| 1 | Why Raster Vision? | 5 |
| 1.1 | Why do we need yet another deep learning library? | 5 |
| 1.2 | What are the benefits of Raster Vision? | 5 |
| 1.3 | Who is Raster Vision for? | 6 |
| 2 | Quickstart | 7 |
| 2.1 | The Data | 8 |
| 2.2 | Creating an ExperimentSet | 8 |
| 2.3 | Running an experiment | 9 |
| 2.4 | Seeing Results | 10 |
| 2.5 | Predict Packages | 11 |
| 2.6 | Next Steps | 13 |
| 3 | Setup | 15 |
| 3.1 | Installing Raster Vision | 15 |
| 3.2 | Raster Vision Configuration | 16 |
| 3.3 | Docker Containers | 17 |
| 3.4 | Running on a machine with GPUs | 17 |
| 3.5 | Setting up AWS Batch | 18 |
| 4 | Experiment Configuration | 19 |
| 4.1 | Experiment Set | 19 |
| 4.2 | ExperimentConfig | 19 |
| 4.3 | Task | 20 |
| 4.4 | Backend | 22 |
| 4.5 | Dataset | 23 |
| 4.6 | Scene | 23 |
| 4.7 | Analyzers | 26 |
| 4.8 | Evaluators | 26 |
| 4.9 | Default Providers | 26 |
| 5 | Commands | 29 |
| 5.1 | Command Types | 29 |
| 6 | Running Experiments | 31 |
| 6.1 | ExperimentRunners | 31 |
| 6.2 | Running locally | 32 |

| | | |
|-----------|--|-----------|
| 6.3 | Running on AWS Batch | 32 |
| 7 | Making Predictions (Inference) | 33 |
| 7.1 | How to make predictions with models train by Raster Vision | 33 |
| 7.2 | Predict Package | 33 |
| 8 | Command Line Interface | 35 |
| 8.1 | Commands | 35 |
| 9 | Miscellaneous Topics | 39 |
| 9.1 | FileSystems | 39 |
| 9.2 | Viewing Tensorboard | 39 |
| 9.3 | Model Defaults | 39 |
| 9.4 | Reusing models trained by Raster Vision | 40 |
| 10 | Codebase Design Patterns | 43 |
| 10.1 | Configuration vs Entity | 43 |
| 10.2 | Fluent Builder Pattern | 44 |
| 10.3 | Global Registry | 45 |
| 11 | Plugins | 47 |
| 11.1 | Creating Plugins | 47 |
| 11.2 | Registering the Plugin | 48 |
| 11.3 | Configuring Raster Vision to use your Plugins | 48 |
| 11.4 | Plugins in remote environments | 48 |
| 11.5 | Example Plugin | 48 |
| 12 | QGIS Plugin | 49 |
| 12.1 | Installing | 50 |
| 12.2 | Load Experiment | 50 |
| 12.3 | Predict | 53 |
| 12.4 | Style Profiles | 54 |
| 12.5 | Configure | 54 |
| 13 | API Reference | 57 |
| 13.1 | API Reference | 57 |
| | Python Module Index | 75 |



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). It allows for 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):
```

(continues on next page)

(continued from previous page)

```
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)

    task = rv.TaskConfig.builder(rv.OBJECT_DETECTION) \
        .with_chip_size(512) \
        .with_classes({
            'building': (1, 'red')
        }) \
        .with_chip_options(neg_ratio=1.0,
                           ioa_thresh=0.8) \
        .with_predict_options(merge_thresh=0.1,
                              score_thresh=0.5) \
        .build()

    backend = rv.BackendConfig.builder(rv.TF_OBJECT_DETECTION) \
        .with_task(task) \
        .with_debug(True) \
        .with_batch_size(8) \
        .with_num_steps(5) \
        .with_model_defaults(rv.SSD_MOBILENET_V2_COCO) \
        .build()

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

    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()

    val_raster_source = rv.RasterSourceConfig.builder(rv.GEOTIFF_SOURCE) \
        .with_uri(val_image_uri) \
        .with_stats_transformer() \
        .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) \
        .build()

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

    experiment = rv.ExperimentConfig.builder() \
```

(continues on next page)

(continued from previous page)

```

        .with_id('tiny-spacenet-experiment') \
        .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()

```

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.

Why Raster Vision?

1.1 Why do we need yet another deep learning library?

Machine learning libraries generally implement the algorithms and other core functionality needed to build models. The workflow of creating training data in a format that the machine learning library understands, running training in a highly configurable way, making predictions on validation images and performing evaluations on models is usually up to the user to figure out. This often results in a bunch of one-off scripts that are assembled per project, and not engineered to be reusable. Raster Vision is a framework that allows you to state configurations in modifiable and reusable ways, and keeps track of all the files through each step of the machine learning model building workflow. This means you can focus on running experiments to see which machine learning techniques apply best to your problems, and leave the data munging and repeatable workflow processes to Raster Vision.

In addition, the current libraries in the deep learning ecosystem don't usually work well out of the box with large imagery sets, and especially not geospatial imagery (e.g. satellite, aerial, and drone imagery). For example, in traditional object detection, each image is a small PNG file and contains a few objects. In contrast, when working with satellite and aerial imagery, each image is a set of very large GeoTIFF files and contains hundreds of objects that are sparsely distributed. In addition, annotations and predictions are represented in geospatial coordinates using GeoJSON files.

1.2 What are the benefits of Raster Vision?

- Configure *Task*, *Backend*, and other components of deep learning *ExperimentConfig* using a flexible and readable pattern that sets up all the information needed to run a machine learning workflow.
- Run *Commands* from the command line that execute locally or on AWS Batch. With AWS Batch, you can fire off jobs that run through the entire workflow on a GPU spot instance that is created for the workload and terminates immediately afterwards, saving not only money in EC2 instance hours, but also time usually spent ssh'ing into machines or babysitting processes.
- Read files from HTTP, S3, the local filesystem, or anywhere with the pluggable *FileSystems* architecture.
- Make predictions and build inference pipelines using a single file as output of the Raster Vision workflow for any experiment, which includes the trained model and configuration.

1.3 Who is Raster Vision for?

Raster Vision is 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 Tensorflow and Keras 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 know what you have to do. That's one of the core ideas of Raster Vision - the work to get an experiment you created against a tiny test set is simply to point it at more data, tweak some parameters and run it in a GPU-enabled environment. Also, we'll show how to make predictions on the data using a model we've already trained on GPUs for some time to show what you can expect to get out Raster Vision from a basic setup.

For the Quickstart we are going to be using one of the published [Docker Containers](#) 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 to install all the necessary dependencies. See [Installing Raster Vision](#) 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-0.8 /bin/bash
```

See also:

See *Docker Containers* 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)

        task = rv.TaskConfig.builder(rv.OBJECT_DETECTION) \
            .with_chip_size(300) \
            .with_classes({
                'building': (1, 'red')
            }) \
            .with_chip_options(neg_ratio=1.0,
                              ioa_thresh=0.8) \
            .with_predict_options(merge_thresh=0.1,
                                  score_thresh=0.5) \
            .build()

        backend = rv.BackendConfig.builder(rv.TF_OBJECT_DETECTION) \
            .with_task(task) \
            .with_debug(True) \
            .with_batch_size(1) \
            .with_num_steps(2) \
            .with_model_defaults(rv.SSD_MOBILENET_V2_COCO) \
            .build()

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

        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()
```

(continues on next page)

(continued from previous page)

```

val_raster_source = rv.RasterSourceConfig.builder(rv.GEOTIFF_SOURCE) \
    .with_uri(val_image_uri) \
    .with_stats_transformer() \
    .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) \
    .build()

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

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()

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 create a `TaskConfig` and `BackendConfig` that configure Raster Vision to perform object detection on buildings. In fact, Raster Vision isn't doing any of the heavy lifting of actually training the model - it's using the [TensorFlow Object Detection API](#) for that. Raster Vision just provides a configuration wrapper that sets up all of the options and data for the experiment workflow that utilizes that library.

You also can see 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` represented by this configuration by calling `with_stats_transformer()`, which sets a default `StatsTransformerConfig` onto the `RasterSourceConfig` transformers.

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
  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.json
```

(continues on next page)

(continued from previous page)



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](#).


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.



img/vegas/1929.png



img/vegas/predictions.png

See also:

You can read more about the *Predict Package* 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) and view the results in QGIS, see the SpaceNet examples in the [Raster Vision Examples](#) repository.

3.1 Installing Raster Vision

You can get the library directly from PyPI:

```
> pip install rastervision
```

Note: Raster Vision requires Python 3 or later.

3.1.1 Troubleshooting macOS Installation

If you encounter problems running `pip install rastervision` 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
> pip install rastervision
```

3.1.2 Using AWS, Tensorflow, and/or Keras

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

```
> pip install rastervision[aws,tensorflow,tensorflow-gpu]
```

If you'd like to use Raster Vision with [Tensorflow Object Detection](#) or [TensorFlow DeepLab](#), you'll need to follow the instructions in their documentation about how to install, or look at our Dockerfile to see an example of setting this up.

Note: You must install Tensorflow Object Detection and Deep Lab from [Azavea's fork](#) of the models repository, since it contains some necessary changes that have not yet been merged back upstream.

Note: The usage of [Docker Containers](#) is recommended, as it provides a consistent environment for running Raster Vision.

If you have Docker installed, simply run the published container according to the instructions in [Docker Containers](#)

3.2 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.

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
```

See the root options of the [Command Line Interface](#) for the option to set the profile.

3.2.1 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.

3.2.2 PLUGINS

```
[PLUGINS]
files=[]
modules=[]
```

- `files` - Optional list of Python file URIs to gather plugins from. Must be a JSON-parsable array of values, e.g. `["analyzers.py", "backends.py"]`.
- `modules` - Optional list of modules to load plugins from. Must be a JSON-parsable array of values, e.g. `["rvplugins.analyzer", "rvplugins.backend"]`.

See [Plugins](#) for more information about the Plugin architecture.

3.2.3 Other Sections

Other configurations are documented elsewhere:

- [aws batch config section](#)

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`

3.3 Docker Containers

Using the Docker containers published for Raster Vision allows you 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 containers are published to quay.io/azavea/raster-vision. To run the raster vision container for the latest release, run:

```
> docker run --rm -it quay.io/azavea/raster-vision:cpu-0.8 /bin/bash
```

You'll likely need to load up volumes and expose ports to make this container fully useful; see the [docker/console](#) script for an example usage.

We publish containers set up for both CPU-only running and GPU-running, and tag each container as appropriate. So you can also pull down the `quay.io/azavea/raster-vision:gpu-0.8` image, as well as `quay.io/azavea/raster-vision:cpu-latest` and `quay.io/azavea/raster-vision:gpu-latest`.

You can also base your own Dockerfiles off the Raster Vision container to use with your own codebase. See the Dockerfiles in the [Raster Vision Examples](#) repository.

3.4 Running on a machine with GPUs

If you are running 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 make sure of a couple of things so that the Docker container is able to utilize the GPUs.

You'll need to install the [nvidia-docker](#) runtime on your system. Follow their [quickstart](#) and installation instructions. Make sure that your GPU is supported by NVIDIA Docker - if not you might need to find another way to have your

Docker container communicate with the GPU. If you figure out how to support more GPUs, please let us know so we can add the steps to this documentation!

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

```
> docker run --runtime=nvidia --rm -it quay.io/azavea/raster-vision:gpu-0.8 /bin/bash
```

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

If you want to run code against AWS, you'll need a specific Raster Vision AWS Batch setup on your account, which you can accomplish through the instructions at the [Raster Vision for AWS Batch setup repository](#).

Set the appropriate configuration in your *Raster Vision Configuration*:

```
[AWS_BATCH]
job_queue=rasterVisionQueue
job_definition=raster-vision-gpu
attempts=1
```

- `job_queue` - Job Queue to submit Batch jobs to.
- `job_definition` - The Job Definition that define the Batch jobs to run.
- `attempts` - Optional number of attempts to retry failed jobs.

See also:

For more information about how Raster Vision uses AWS Batch, see the section: [Running on AWS Batch](#).

Experiment Configuration

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

4.1 Experiment Set

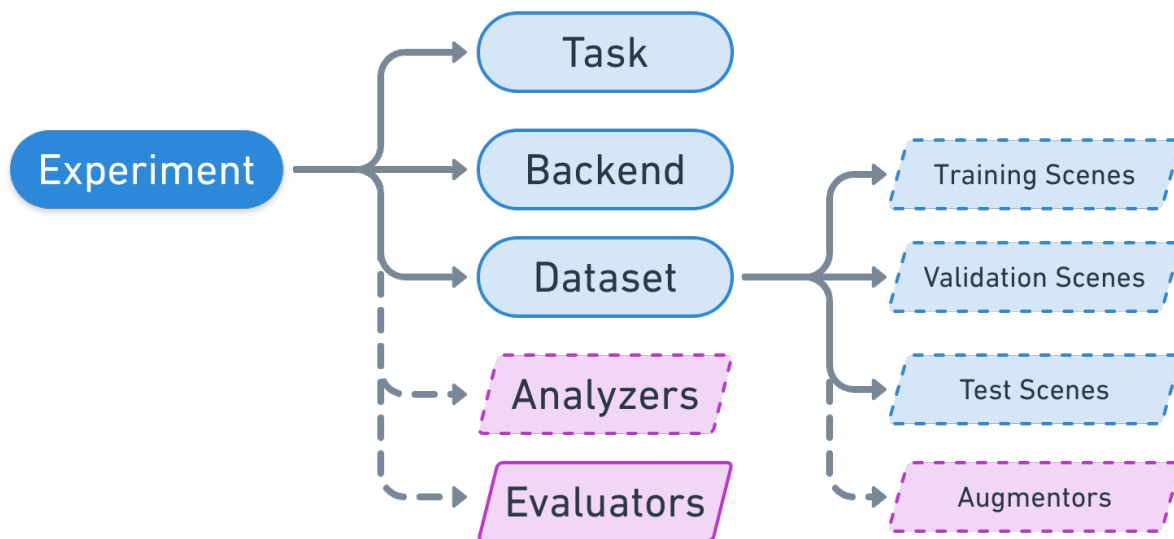
An experiment set is a set of related experiments and can be created by subclassing `ExperimentSet`. For each experiment, the class should have a method prefixed with `exp_` that returns either a single `ExperimentConfig`, or a list of `ExperimentConfig` objects.

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`.

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 a 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 Future 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 with **key** from the `.build` static method of TaskConfig. In our `tiny_spacenet.py` example, we configured an object detection task:

```
task = rv.TaskConfig.builder(rv.OBJECT_DETECTION) \
    .with_chip_size(512) \
    .with_classes({
        'building': (1, 'red')
    }) \
    .with_chip_options(neg_ratio=1.0,
                      ioa_thresh=0.8) \
    .with_predict_options(merge_thresh=0.1,
                          score_thresh=0.5) \
    .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 third-party libraries via a “backend” interface inspired by Keras. Each backend is a subclass of Backend and contains methods for translating between Raster Vision data structures and calls to a third-party library.

4.4.1 Keras Classification

rv.KERAS_CLASSIFICATION

For chip classification, the default backend is Keras Classification, which is a small, simple library for image classification using Keras. Currently, it only has support for ResNet50.

4.4.2 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.

4.4.3 TensorFlow DeepLab

rv.TF_DEEPLAB

For semantic segmentation, the default backend is Tensorflow Deeplab. It has support for the Deeplab segmentation architecture with Mobilenet and Inception as base models.

Note: For each 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.

4.4.4 BackendConfig

A BackendConfig is always constructed through a builder, which is created with **key** from the `.build` static method of BackendConfig. In our `tiny_spacenet.py` example, we configured the TensorFlow Object Detection backend:

```
backend = rv.BackendConfig.builder(rv.TF_OBJECT_DETECTION) \
    .with_task(task) \
    .with_debug(True) \
    .with_batch_size(8) \
    .with_num_steps(5) \
    .with_model_defaults(rv.SSD_MOBILENET_V2_COCO) \
    .build()
```

See also:

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

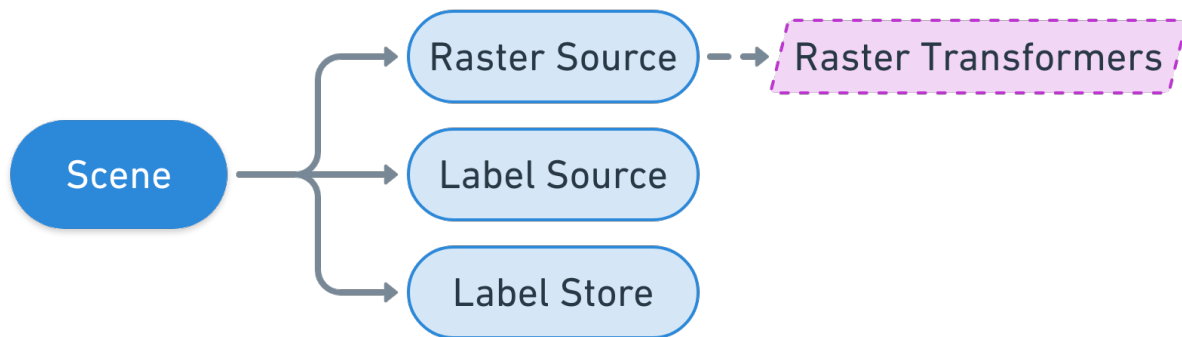
4.5 Dataset

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 [Augmentors](#), which describe 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 call `with_train_scenes` and `with_validation_scenes` with many scenes:

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

4.6 Scene



A scene represents an image, associated labels, and an optional Area of Interest (AOI) that describes what area of the scene has 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 contain none of the elements that are desired to be detected.

A scene is composed of the following elements:

- *Image*: Represented in Raster Vision by a `RasterSource`, an 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.
- *AOI (Optional)*: An Area 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 `RasterSource` optionally combined with a `LabelSource`, `LabelStore`, and `AOI`.

In our `tiny_spacenet.py` example, we configured the train scene with a GeoTIFF URI and a GeoJSON URI. We pass in a built `RasterSource`, however we pass in just the URI for the `LabelSource`. This is because the `SceneConfig` can construct a default `LabelSource` based on the URI using *Default Providers*.

```
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()
```

The validation scene is also constructed very similarly. However, it's worth noting that the `LabelStore` is not even mentioned in the building of the configuration. This is because the prediction label store can be determined by *Default Providers*, by finding the default `LabelStore` provider for a given task.

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 using a `RasterSource`. 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.

GeoTIFF

rv.GEOTIFF_SOURCE

Georeferenced imagery stored as GeoTIFFs can be read using a `GeoTIFFSource`. 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. This is implemented behind the scenes using `Rasterio`, which builds a VRT out of the constituent images.

Image

rv.IMAGE_SOURCE

Non-georeferenced images including `.tif`, `.png`, and `.jpg` files can be read using an `ImageSource`. This is useful for oblique drone imagery, biomedical imagery, and any other (potentially massive!) non-georeferenced images.

Segmentation GeoJSON

rv.GEOJSON_SOURCE

Semantic segmentation labels stored as polygons and lines in a GeoJSON file can be rasterized and read using a `GeoJSONSource`. 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 up the training scene raster source:

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

See also:

The [RasterSourceConfig](#) API Reference docs have more information about the RasterSource types available.

4.6.3 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.

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

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

See also:

The [LabelSourceConfig](#) API Reference docs have more information about the LabelSource types available.

4.6.4 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.LabelStoreConfig.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 its labels. We could do that, but if we leave that out the Raster Vision logic will set that path explicitly based on the experiment's root directory and the predict command's key.

See also:

The [LabelStoreConfig](#) API Reference docs have more information about the LabelStore types available.

4.6.5 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.

4.6.6 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.

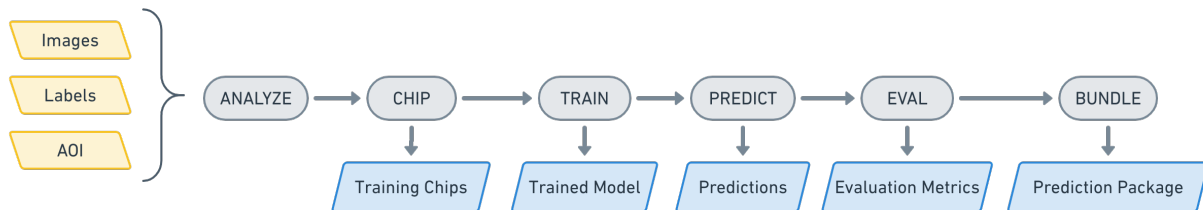
See also:

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.



5.1 Command Types

A Raster Vision experiment is a sequence of commands that each run a component of a machine learning workflow.

5.1.1 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.

5.1.2 CHIP

Scenes are comprised of large geospatial raster sources (eg. GeoTIFFs) and geospatial label sources (eg. 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.

5.1.3 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.

5.1.4 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.

5.1.5 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.

5.1.6 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.

Running Experiments

Running experiments in Raster Vision is done by the `run rastervision` 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 chosed 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 what commands are requested from the user, what commands already have been run, and what commands are common between *ExperimentConfigs*.

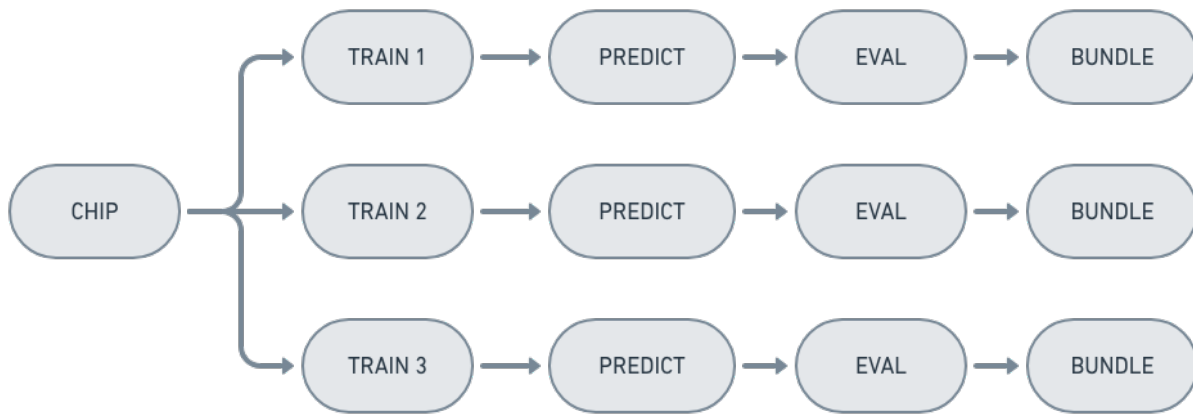
Note: Raster Vision considers two commands to be equal if their inputs, outputs and command types (e.g. `rv.CHIP`, `rv.TRAIN`, etc...) are the same. Raster Vision will avoid running multiple of the same command in one run with sameness defined in this way.

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, and declare what its inputs and outputs are. 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 parent `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 directed acyclic graph (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 chip:



6.2 Running locally

A `rastervision run local ...` command will use the `LocalExperimentRunner`, which simply executes each command in the DAG on the client machine. These run serially, without any parallelization. In future versions, we may want to split the DAG up into components that can be executed in parallel on a large machine.

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 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.

The `AWSBatchExperimentRunner` executes each command as a call to `rastervision run_command` inside of the Docker image configured in the job definition that is sent to AWS Batch. 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.

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 applied to 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 train 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.

With the command line, you are loading the model and saving the label output in a single call. If you need to call this for a large number of files, consider using the `Predictor` in Python code, 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 `rastertransformer`, and the model could have trained on a subset of bands dictated by the `rastersource`. We need to know about the configuration of what's coming out of the model as a prediction in order to properly serialize it to GeoJSON, raster data, or whatever other labelstore was used to serialize labels. The prediction logic needs to know what `Task` is being used to apply any transformations that take raw model output and transform it to meaningful classifications or other 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.

Command Line Interface

The Raster Vision command line utility, `rastervision` is installed with a `pip install of rastervision`. It consists of 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)

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

Some specific parameters to call out:

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 this, Raster Vision will remind you.

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
```

Use `-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.

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 prediction 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    String containing 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 experiments 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 serialized 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:
  --help  Show this message and exit.
```


9.1 FileSystems

The FileSystem architecture allows support of multiple filesystems 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 FileSystems via the plugin. We're happy to take contributions on new FileSystem support if it's generally useful!

9.2 Viewing Tensorboard

Backends that utilize TensorFlow will start up 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 attributes into backends, instead of having to explicitly state them for every experiment that you want to use defaults for. 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 it's model defaults differently. For more information, see the `rastervision/backend/model_defaults.json` file in the repository.

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.

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. Default includes 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 what each model is can be found in the [Tensorflow DeepLab Model Zoo](#) page. Default includes pretrained model weights and backend configurations for the following models:

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

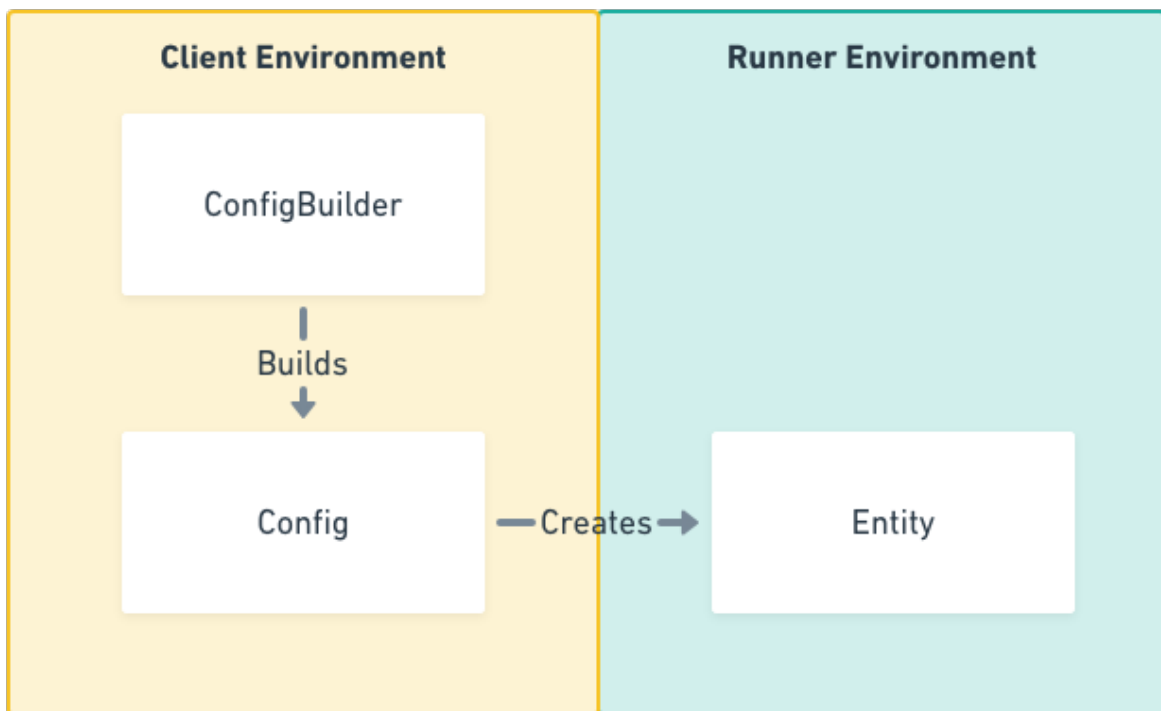
9.4 Reusing models trained by Raster Vision

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.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.

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 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 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 construct 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 what's called a *fluent builder patter*. 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()` call 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 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 registry, 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*.

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 parent class `__init__` 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 and 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.

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 executes 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 carry with them 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

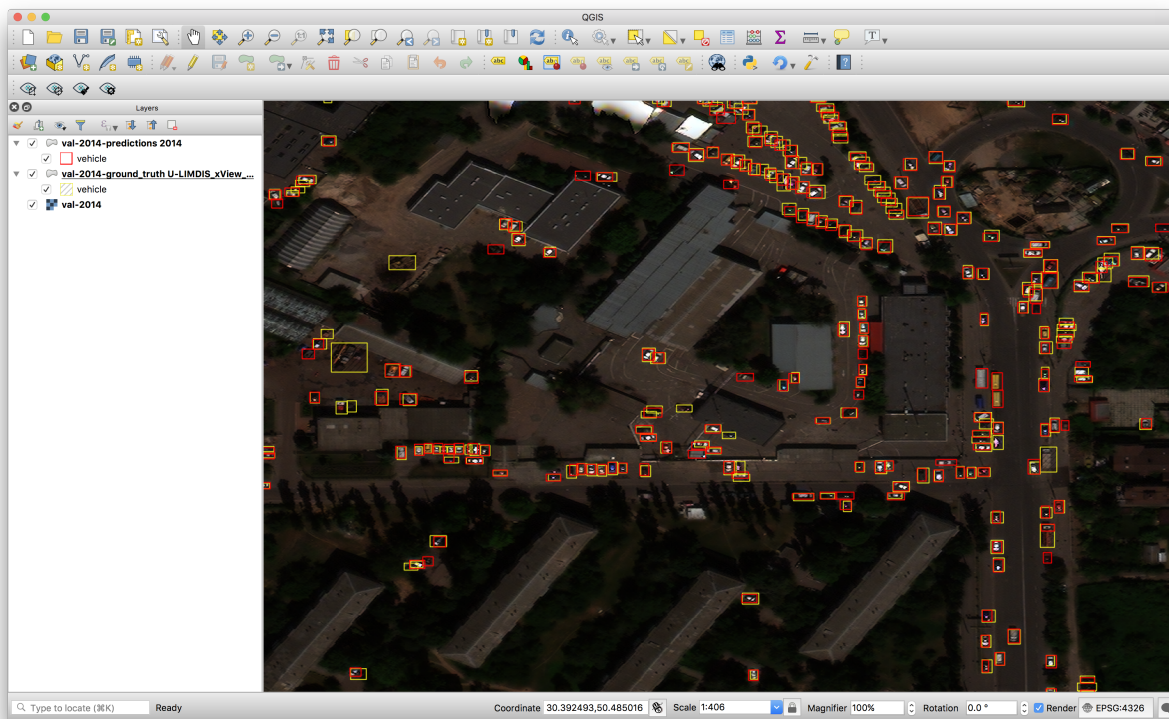
You can set the file location in the path of your Raster Vision plugin configuration in the `files` setting, and then use it in experiments like so (assuming `EASY_EVALUATOR` was defined the same as above):

```
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

QGIS Plugin



The Raster Vision QGIS plugin allows Raster Vision users to quickly view the results of experiments run against geospatial imagery. It also lets you run predictions inside of QGIS using the *Predict Package* of trained models.

12.1 Installing

To install the QGIS Plugin, you must have `rastervision` installed in the Python 3 environment that is running QGIS. Don't worry, you won't have to install all of the deep learning frameworks just to use the plugin - you can just `pip install rastervision` (or `pip3 install rastervision` if Python 3 is not the default on your system). This has been tested with Python 3.6 and QGIS 3.2.

12.1.1 Installing from Plugin Manager

A package containing all the needed dependencies can be installed through QGIS Plugin Manager. To install from plugin manager:

Click the menu “Plugins” -> “Manage and Install Plugins”. Enter ‘Raster Vision’ in search box. After installation is complete, there should be a “Raster Vision” submenu under the “Plugins” menu.

12.1.2 Installing from release

To install, grab the release `.tar.gz` file from the [GitHub Releases](#) page. Extract this into your QGIS Plugins directory, then restart QGIS and activate the plugin through QGIS menu (“Plugins” -> “Manage and Install Plugins”). You can use a command like:

```
tar -xvf rastervision_qgis-v0.8.0.tar.gz -C ${QGIS_PLUGIN_DIRECTORY}
```

Where `${QGIS_PLUGIN_DIRECTORY}` is your QGIS plugin directory. See this [GIS StackExchange post](#) if you need help finding your plugin directory.

12.1.3 QGIS Environment Setup

Note: QGIS environment variables are distinct from Bash environment variables, and can be set by going to “QGIS3” -> “Preferences” -> “System” -> “Environment” in the menu and then restarting QGIS.

Using with AWS

To use the plugin with files stored on AWS S3, you will need to have `boto3` installed, which can be done with `pip install boto3`. You'll also need to set an `AWS_PROFILE` environment variable in QGIS if you're not using the default AWS profile.

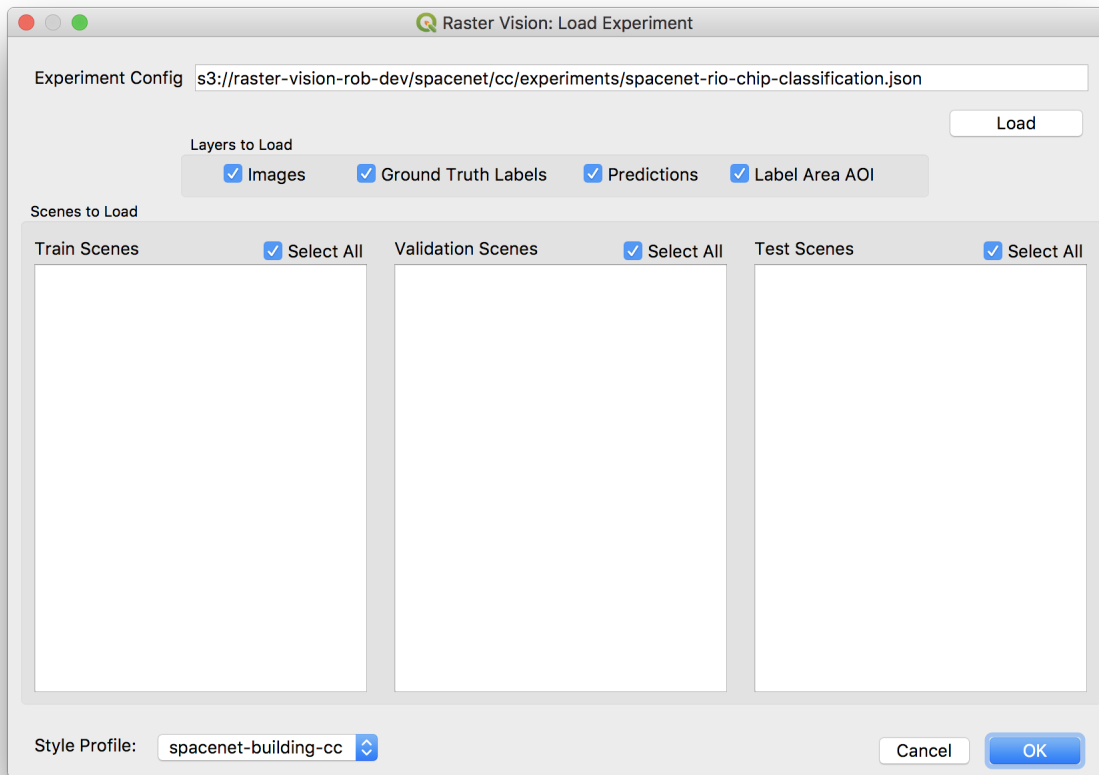
Using with Docker

To run predict through Docker, make sure that the Docker command is on the `PATH` environment variable used by QGIS.

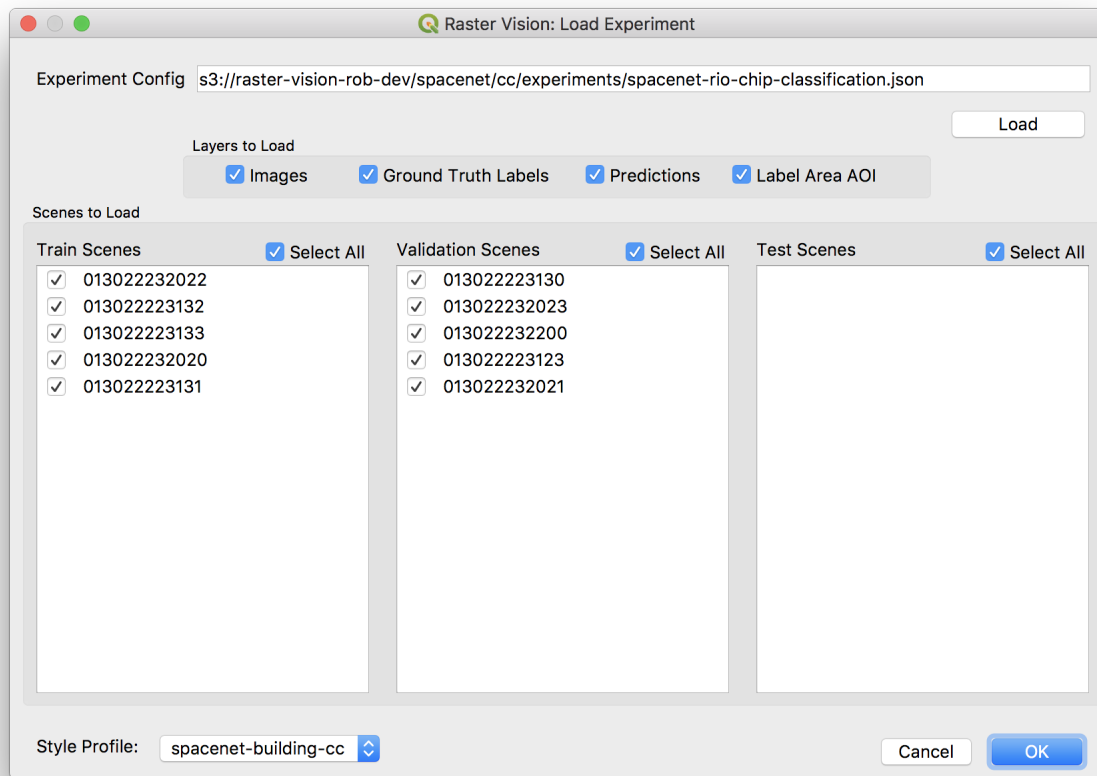
12.2 Load Experiment

The Load Experiment Dialog of the plugin lets you load results from an experiment.

The first step in using this feature is to load up an experiment configuration JSON file. You can find experiment configurations in the `experiments` directory under the `root_uri` of your experiment.



After hitting the *Load* button, you should see the dialog populate with the train, validation, and test scenes that were used in the experiment. The names that appear in the dialog are the scene's ID.



You can select which data type you want from each scene in the “Layers to Load” section. You can also select Scenes that you want to load from the list boxes.

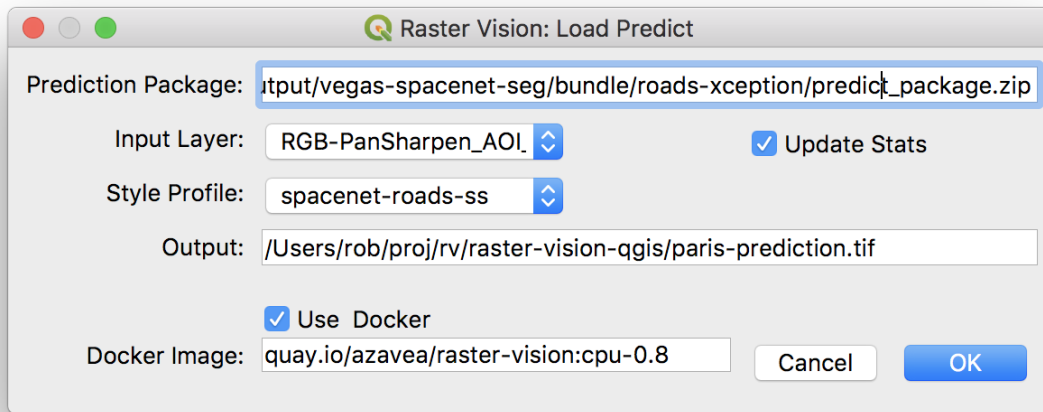
You can choose one of your configured [Style Profiles](#) from the “Style Profile” box. All incoming layers will be styled according to the style profile.

When you’re satisfied with your choices, pressing OK and the project will load in QGIS. This will clear the current project in QGIS and load the new layers - if you already have layers, it will confirm that you want to clear out your project.

The layers that load will have the following naming conventions:

- `train-*` layers are from train scenes.
- `val-*` layers are from validation scenes.
- `test-*` layers are from test scenes.
- Everything will include the scene ID
- Ground truth labels are suffixed with `-ground_truth`
- Predictions are suffixed with `-predictions`

12.3 Predict



This Dialog allows you to make predictions using a *Predict Package* from a raster vision experiment.

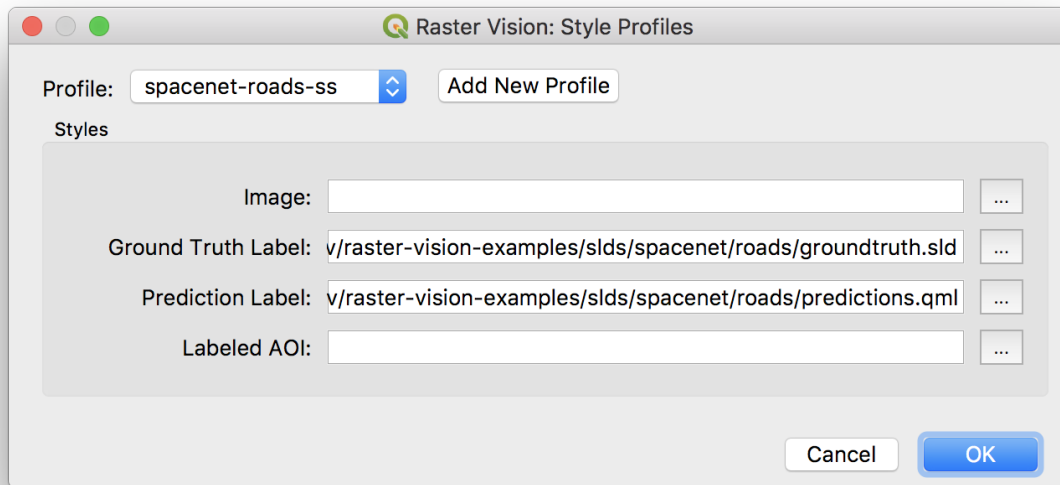
To use do the following:

- input the predict package URI
- select a layer from the “Input Layer” dropdown, which is populated from the raster layers of the current QGIS project
- Optionally choose a Style Profile
- Select whether or not to update any stats used by the model with the given image
- Give the path where the prediction labels should be saved to

You can use Docker or a local installation of Raster Vision to run the prediction. If using Docker, you’ll have to give the name of the image from which to run the container.

This runs a similar process as the *predict* CLI command, and will load the prediciton layer after prediction completes.

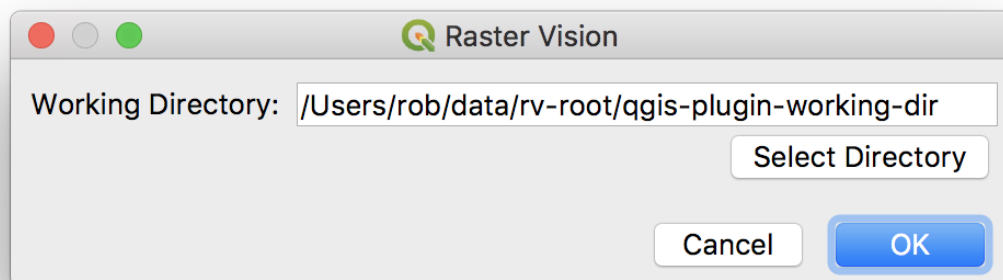
12.4 🗨️ Style Profiles



Set up style profiles so that when you load an experiment or make predictions, layers are automatically styled with given SLDs or QML files.

The best way to do this is to styl each of the types of layers you want after first loading an experiment. Export an SLD or QML of the style for each layer by using the *Style -> Save Style* command in the *Symbolology* section of the layer properties. Then, create a style profile for that experiment group, and point it to the appropriate QML or SLD files. Now you'll be able to select the style profile when loading new experiments and making predictions.

12.5 🗨️ Configure



Configure the plugin with a working directory. If the files live on S3, this plugin will download files as necessary to

your local working directory. If the file already exists in the working directory, the plugin will check the timestamps and overwrite the local file if the file on S3 is newer.

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

13.1 API Reference

This part of the documentation lists the full API reference of public classes and functions.

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

13.1.1 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.

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_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.
Args: *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.

13.1.2 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.

```

13.1.3 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.

    Args: chip_size: Integer value chip size

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

```

Args:

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

Args:

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.

Args: chip_size: Integer value chip size

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.

Args:

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.

Args:

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

Args:

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.

Args: chip_size: Integer value chip size

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

Set the classes for this task.

Args:

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.

13.1.4 BackendConfig

BackendConfigBuilders are created by calling

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

Where BACKEND_TYPE is one of the following:

rv.KERAS_CLASSIFICATION

class rastervision.backend.KerasClassificationConfigBuilder (*prev=None*)

build ()

Build this configuration, setting 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 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*)
 Defines the name of the model file that will be created for this model after training.

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.
Args: task: A TaskConfig object.

with_template (*template*)
 Use a template from the dict, string or uri as the base for the Keras Classification API.

with_train_options (*sync_interval=600, do_monitoring=True, replace_model=False*)
 Sets the train options for this backend.
Args:
 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 checkpointing if exists, if the backend allows for this.

with_training_data_uri (*training_data_uri*)
 Whence comes the training data?
Args: training_data_uri: The location of the training data.

with_training_output_uri (*training_output_uri*)
 Whither goes the training output?
Args:
 training_output_uri: The location where the training output will be stored.

rv.TF_OBJECT_DETECTION

```
class rastervision.backend.TFObjectDetectionConfigBuilder (prev=None)
```

build ()
 Build this configuration, setting 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 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 that will be created for this model after training.

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.
Args: task: A TaskConfig object.

with_template (*template*)
 Use a template for TF Object Detection pipeline config.
Args:
 template: A dict, string or uri as the base for the tensorflow object detection API model training pipeline, for example those found here: [#https://github.com/tensorflow/models/tree/eef6bb5bd3b3cd5fcf54306bf29750b7f9f9a5ea/research/object_detection/samples/configs](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.
Args:
 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 checkpointing if exists, if the backend allows for this.

with_training_data_uri (*training_data_uri*)
 Whence comes the training data?
Args: training_data_uri: The location of the training data.

with_training_output_uri (*training_output_uri*)
 Whither goes the training output?
Args:
 training_output_uri: The location where the training output will be stored.

rv.TF_DEEPLAB

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

build ()
 Build this configuration, setting any values into the TFDL 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 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*)
Defines the name of the fine tune checkpoint that will be created for this model after training.

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

with_task (*task*)
Sets a specific task type.
Args: 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*)
Sets the train options for this backend.
Args:
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 checkpointing if exists, if the backend allows for this.

with_training_data_uri (*training_data_uri*)
Whence comes the training data?
Args: training_data_uri: The location of the training data.

with_training_output_uri (*training_output_uri*)
Whither goes the training output?
Args:
training_output_uri: The location where the training output will be stored.

13.1.5 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_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.

Args:

uri: The URI points to the AoI (nominally a GeoJSON polygon).

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.

Args:

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.

Args:

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 registry 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.

Args:

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.

13.1.6 RasterSourceConfig

RasterSourceConfigBuilders are created by calling

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

Where SOURCE_TYPE is one of the following:

rv.GEOTIFF_SOURCE

```
class rastervision.data.GeoTiffSourceConfigBuilder (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.

Args:

channel_order: numpy array of length n where n is the number of channels to use and the values are channel indices

```
with_stats_transformer ()
```

Add a stats transformer to the raster source.

```
with_transformer (transformer)
```

A transformer to be applied to the raster data.

Args:

transformer: A transformer to apply to the raster data.

```
with_transformers (transformers)
```

Transformers to be applied to the raster data.

Args:

transformers: A list of transformers to apply to the raster data.

```
with_uri (uri)
```

Set URI for a GeoTIFF containing raster data.

```
with_uris (uris)
```

Set URIs for a GeoTIFFs containing as raster data.

rv.IMAGE_SOURCE

```
class rastervision.data.ImageSourceConfigBuilder (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.

Args:

channel_order: numpy array of length n where n is the number of channels to use and the values are channel indices

```
with_stats_transformer ()
```

Add a stats transformer to the raster source.

```
with_transformer (transformer)
```

A transformer to be applied to the raster data.

Args:

transformer: A transformer to apply to the raster data.

with_transformers (*transformers*)

Transformers to be applied to the raster data.

Args:

transformers: A list of transformers to apply to the raster data.

with_uri (*uri*)

Set URI for an image.

Args:

uri: A URI pointing to some (non-georeferenced) raster file (TIFs, PNGs, and JPEGs are supported, and possibly others).

rv.GEOJSON_SOURCE

class rastervision.data.GeoJSONSourceConfigBuilder (*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.

Args:

channel_order: numpy array of length *n* where *n* is the number of channels to use and the values are channel indices

with_rasterizer_options (*background_class_id*, *line_buffer=15*)

Specify options for converting GeoJSON to raster.

Args:

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

line_buffer: Number of pixels to add to each side of line when rasterized.

with_stats_transformer ()

Add a stats transformer to the raster source.

with_transformer (*transformer*)

A transformer to be applied to the raster data.

Args:

transformer: A transformer to apply to the raster data.

with_transformers (*transformers*)

Transformers to be applied to the raster data.

Args:

transformers: A list of transformers to apply to the raster data.

with_uri (*uri*)

Set URI for a GeoJSON file used to read labels.

13.1.7 LabelSourceConfig

LabelSourceConfigBuilders are created by calling


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

Where SOURCE_TYPE is one of the following:

rv.CHIP_CLASSIFICATION_GEOJSON

```
class rastervision.data.ChipClassificationGeoJSONSourceConfigBuilder (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.

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 of the GeoJSON. 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*)

Set URI for a GeoJSON used to read/write predictions.

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.

For rv.OBJECT_DETECTION:

rv.OBJECT_DETECTION_GEOJSON

```
class rastervision.data.ObjectDetectionGeoJSONSourceConfigBuilder (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.SemanticSegmentationRasterSourceConfigBuilder (prev=None)
```

```
    build()
```

Returns the configuration that is built by this builder.

```
    with_raster_source (source, channel_order=None)
```

Set raster_source.

Args:

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

Returns: SemanticSegmentationRasterSourceConfigBuilder

```
    with_rgb_class_map (rgb_class_map)
```

Set rgb_class_map.

Args:

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

Returns: SemanticSegmentationRasterSourceConfigBuilder

13.1.8 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.
```

13.1.9 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.
    Args: stats_uri: URI to the stats json to use
```

13.1.10 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.
```

Args: aug_prob: Float value between 0.0 and 1.0

13.1.11 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_stats_uri (stats_uri)
```

Set the stats_uri.

Args: stats_uri: URI to the stats json to use

13.1.12 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.

Args: class_map: The class map to be used

```
with_output_uri (output_uri)
```

Set the output_uri.

Args: output_uri: URI to the stats json to use

```
with_task (task)
```

Sets a specific task type, e.g. rv.OBJECT_DETECTION.

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.
    Args: class_map: The class map to be used

with_output_uri (output_uri)
    Set the output_uri.
    Args: output_uri: URI to the stats json to use

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

```

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.
    Args: class_map: The class map to be used

with_output_uri (output_uri)
    Set the output_uri.
    Args: output_uri: URI to the stats json to use

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

```

13.1.13 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.
        Args:
            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 di-
                rectory 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 indicating a new channel order to use for the imagery being pre-
                dicted 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.

Args:

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.

13.1.14 Plugin Registry

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

register_config_builder (*group*, *key*, *builder_class*)

Registers a ConfigBuilder as a plugin.

Args: **group** - The Config group, e.g. rv.BACKEND, rv.TASK. **key** - The key used for this plugin. This will be used to

construct the builder in a “.builder(key)” call.

builder_class - The subclass of ConfigBuilder that builds 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_experiment_runner (*runner_key*, *runner_class*)

Registers an ExperimentRunner as a plugin.

Args:

runner_key - The key used to reference this plugin runner. 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.

runner_class - The class of the ExperimentRunner plugin.

register_filesystem (*filesystem_class*)

Registers a FileSystem as a plugin.

r

`rastervision`, [57](#)

Symbols

`__init__()` (*rastervision.Predictor* method), 73

B

`build()` (*rastervision.analyzer.StatsAnalyzerConfigBuilder* method), 72

`build()` (*rastervision.augmentor.NodataAugmentorConfigBuilder* method), 71

`build()` (*rastervision.backend.KerasClassificationConfigBuilder* method), 62

`build()` (*rastervision.backend.TFDeeplabConfigBuilder* method), 64

`build()` (*rastervision.backend.TFObjectDetectionConfigBuilder* method), 63

`build()` (*rastervision.data.ChipClassificationGeoJSONSourceConfigBuilder* method), 69

`build()` (*rastervision.data.ChipClassificationGeoJSONStoreConfigBuilder* method), 70

`build()` (*rastervision.data.DatasetConfigBuilder* method), 59

`build()` (*rastervision.data.GeoJSONSourceConfigBuilder* method), 68

`build()` (*rastervision.data.GeoTiffSourceConfigBuilder* method), 67

`build()` (*rastervision.data.ImageSourceConfigBuilder* method), 67

`build()` (*rastervision.data.ObjectDetectionGeoJSONSourceConfigBuilder* method), 69

`build()` (*rastervision.data.ObjectDetectionGeoJSONStoreConfigBuilder* method), 70

`build()` (*rastervision.data.SceneConfigBuilder* method), 66

`build()` (*rastervision.data.SemanticSegmentationRasterSourceConfigBuilder* method), 70

`build()` (*rastervision.data.SemanticSegmentationRasterStoreConfigBuilder* method), 71

`build()` (*rastervision.data.StatsTransformerConfigBuilder* method), 71

`build()` (*rastervision.evaluation.ChipClassificationEvaluatorConfigBuilder* method), 72

`build()` (*rastervision.evaluation.ObjectDetectionEvaluatorConfigBuilder* method), 72

`build()` (*rastervision.evaluation.SemanticSegmentationEvaluatorConfigBuilder* method), 73

`build()` (*rastervision.experiment.ExperimentConfigBuilder* method), 57

`build()` (*rastervision.task.ChipClassificationConfigBuilder* method), 59

`build()` (*rastervision.task.ObjectDetectionConfigBuilder* method), 60

`build()` (*rastervision.task.SemanticSegmentationConfigBuilder* method), 61

C

`ChipClassificationConfigBuilder` (class in *rastervision.task*), 59

`ChipClassificationEvaluatorConfigBuilder` (class in *rastervision.evaluation*), 72

`ChipClassificationGeoJSONSourceConfigBuilder` (class in *rastervision.data*), 69

`ChipClassificationGeoJSONStoreConfigBuilder` (class in *rastervision.data*), 70

`clear_label_source()` (*rastervision.data.SceneConfigBuilder* method), 66

`clear_label_store()` (*rastervision.data.SceneConfigBuilder* method), 66

D

`DatasetConfigBuilder` (class in *rastervision.data*), 59

E

`ExperimentConfigBuilder` (class in *rastervision.experiment*), 57

G

`GeoJSONSourceConfigBuilder` (class in *rastervision.data*), 68

GeoTiffSourceConfigBuilder (class in *rastervision.data*), 67

I

ImageSourceConfigBuilder (class in *rastervision.data*), 67

K

KerasClassificationConfigBuilder (class in *rastervision.backend*), 62

L

load_model() (*rastervision.Predictor* method), 73

N

NodataAugmentorConfigBuilder (class in *rastervision.augmentor*), 71

O

ObjectDetectionConfigBuilder (class in *rastervision.task*), 60

ObjectDetectionEvaluatorConfigBuilder (class in *rastervision.evaluation*), 72

ObjectDetectionGeoJSONSourceConfigBuilder (class in *rastervision.data*), 69

ObjectDetectionGeoJSONStoreConfigBuilder (class in *rastervision.data*), 70

P

PluginRegistry (class in *rastervision.plugin*), 74

predict() (*rastervision.Predictor* method), 74

Predictor (class in *rastervision*), 73

R

rastervision (module), 57

register_config_builder() (*rastervision.plugin.PluginRegistry* method), 74

register_default_evaluator() (*rastervision.plugin.PluginRegistry* method), 74

register_default_label_source() (*rastervision.plugin.PluginRegistry* method), 74

register_default_label_store() (*rastervision.plugin.PluginRegistry* method), 74

register_default_raster_source() (*rastervision.plugin.PluginRegistry* method), 74

register_experiment_runner() (*rastervision.plugin.PluginRegistry* method), 74

register_filesystem() (*rastervision.plugin.PluginRegistry* method), 74

S

SceneConfigBuilder (class in *rastervision.data*), 66

SemanticSegmentationConfigBuilder (class in *rastervision.task*), 61

SemanticSegmentationEvaluatorConfigBuilder (class in *rastervision.evaluation*), 73

SemanticSegmentationRasterSourceConfigBuilder (class in *rastervision.data*), 70

SemanticSegmentationRasterStoreConfigBuilder (class in *rastervision.data*), 71

StatsAnalyzerConfigBuilder (class in *rastervision.analyzer*), 72

StatsTransformerConfigBuilder (class in *rastervision.data*), 71

T

TFDeepLabConfigBuilder (class in *rastervision.backend*), 64

TFOBJECTDetectionConfigBuilder (class in *rastervision.backend*), 63

W

with_analyze_key() (*rastervision.experiment.ExperimentConfigBuilder* method), 57

with_analyze_uri() (*rastervision.experiment.ExperimentConfigBuilder* method), 57

with_analyzer() (*rastervision.experiment.ExperimentConfigBuilder* method), 57

with_analyzers() (*rastervision.experiment.ExperimentConfigBuilder* method), 57

with_aoi_uri() (*rastervision.data.SceneConfigBuilder* method), 66

with_augmentor() (*rastervision.data.DatasetConfigBuilder* method), 59

with_augmentors() (*rastervision.data.DatasetConfigBuilder* method), 59

with_backend() (*rastervision.experiment.ExperimentConfigBuilder* method), 58

with_background_class_id() (*rastervision.data.ChipClassificationGeoJSONSourceConfigBuilder* method), 69

with_batch_size() (*rastervision.backend.KerasClassificationConfigBuilder* method), 62

with_batch_size() (*rastervision.backend.TFDeepLabConfigBuilder* method), 64

with_batch_size() (*rastervision.backend.TFOBJECTDetectionConfigBuilder* method), 63

[method](#)), 63
[with_bundle_key\(\)](#) ([rastervision.experiment.ExperimentConfigBuilder](#) [method](#)), 58
[with_bundle_uri\(\)](#) ([rastervision.experiment.ExperimentConfigBuilder](#) [method](#)), 58
[with_cell_size\(\)](#) ([rastervision.data.ChipClassificationGeoJSONSourceConfigBuilder](#) [method](#)), 69
[with_channel_order\(\)](#) ([rastervision.data.GeoJSONSourceConfigBuilder](#) [method](#)), 68
[with_channel_order\(\)](#) ([rastervision.data.GeoTiffSourceConfigBuilder](#) [method](#)), 67
[with_channel_order\(\)](#) ([rastervision.data.ImageSourceConfigBuilder](#) [method](#)), 67
[with_chip_key\(\)](#) ([rastervision.experiment.ExperimentConfigBuilder](#) [method](#)), 58
[with_chip_options\(\)](#) ([rastervision.task.ObjectDetectionConfigBuilder](#) [method](#)), 60
[with_chip_options\(\)](#) ([rastervision.task.SemanticSegmentationConfigBuilder](#) [method](#)), 61
[with_chip_size\(\)](#) ([rastervision.task.ChipClassificationConfigBuilder](#) [method](#)), 59
[with_chip_size\(\)](#) ([rastervision.task.ObjectDetectionConfigBuilder](#) [method](#)), 61
[with_chip_size\(\)](#) ([rastervision.task.SemanticSegmentationConfigBuilder](#) [method](#)), 62
[with_chip_uri\(\)](#) ([rastervision.experiment.ExperimentConfigBuilder](#) [method](#)), 58
[with_class_map\(\)](#) ([rastervision.evaluation.ChipClassificationEvaluatorConfigBuilder](#) [method](#)), 72
[with_class_map\(\)](#) ([rastervision.evaluation.ObjectDetectionEvaluatorConfigBuilder](#) [method](#)), 73
[with_class_map\(\)](#) ([rastervision.evaluation.SemanticSegmentationEvaluatorConfigBuilder](#) [method](#)), 73
[with_classes\(\)](#) ([rastervision.task.ChipClassificationConfigBuilder](#) [method](#)), 59
[with_classes\(\)](#) ([rastervision.task.ObjectDetectionConfigBuilder](#) [method](#)), 61
[with_classes\(\)](#) ([rastervision.task.SemanticSegmentationConfigBuilder](#) [method](#)), 62
[with_config\(\)](#) ([rastervision.backend.KerasClassificationConfigBuilder](#) [method](#)), 62
[with_config\(\)](#) ([rastervision.backend.TFDeeplabConfigBuilder](#) [method](#)), 65
[with_config\(\)](#) ([rastervision.backend.TFObjectDetectionConfigBuilder](#) [method](#)), 63
[with_dataset\(\)](#) ([rastervision.experiment.ExperimentConfigBuilder](#) [method](#)), 58
[with_debug\(\)](#) ([rastervision.backend.KerasClassificationConfigBuilder](#) [method](#)), 62
[with_debug\(\)](#) ([rastervision.backend.TFDeeplabConfigBuilder](#) [method](#)), 65
[with_debug\(\)](#) ([rastervision.backend.TFObjectDetectionConfigBuilder](#) [method](#)), 63
[with_debug\(\)](#) ([rastervision.task.ChipClassificationConfigBuilder](#) [method](#)), 60
[with_debug\(\)](#) ([rastervision.task.ObjectDetectionConfigBuilder](#) [method](#)), 61
[with_debug\(\)](#) ([rastervision.task.SemanticSegmentationConfigBuilder](#) [method](#)), 62
[with_eval_key\(\)](#) ([rastervision.experiment.ExperimentConfigBuilder](#) [method](#)), 58
[with_eval_uri\(\)](#) ([rastervision.experiment.ExperimentConfigBuilder](#) [method](#)), 58
[with_evaluator\(\)](#) ([rastervision.experiment.ExperimentConfigBuilder](#) [method](#)), 58
[with_evaluators\(\)](#) ([rastervision.experiment.ExperimentConfigBuilder](#) [method](#)), 58
[with_fine_tune_checkpoint_name\(\)](#) ([rastervision.backend.TFDeeplabConfigBuilder](#) [method](#)), 65
[with_fine_tune_checkpoint_name\(\)](#) ([rastervision.backend.TFObjectDetectionConfigBuilder](#) [method](#)), 64
[with_id\(\)](#) ([rastervision.data.SceneConfigBuilder](#) [method](#)), 66

| | | | |
|--|---|---|--|
| <code>with_id()</code> | (<i>rastervision.experiment.ExperimentConfigBuilder</i> method), 58 | <code>with_predict_batch_size()</code> | (<i>rastervision.task.ObjectDetectionConfigBuilder</i> method), 61 |
| <code>with_infer_cells()</code> | (<i>rastervision.data.ChipClassificationGeoJSONSourceConfigBuilder</i> method), 69 | <code>predict_batch_size()</code> | (<i>rastervision.task.SemanticSegmentationConfigBuilder</i> method), 62 |
| <code>with_ioa_thresh()</code> | (<i>rastervision.data.ChipClassificationGeoJSONSourceConfigBuilder</i> method), 69 | <code>predict_debug_uri()</code> | (<i>rastervision.task.ChipClassificationConfigBuilder</i> method), 60 |
| <code>with_label_source()</code> | (<i>rastervision.data.SceneConfigBuilder</i> method), 66 | <code>with_predict_debug_uri()</code> | (<i>rastervision.task.ObjectDetectionConfigBuilder</i> method), 61 |
| <code>with_label_store()</code> | (<i>rastervision.data.SceneConfigBuilder</i> method), 66 | <code>with_predict_debug_uri()</code> | (<i>rastervision.task.SemanticSegmentationConfigBuilder</i> method), 62 |
| <code>with_model_defaults()</code> | (<i>rastervision.backend.KerasClassificationConfigBuilder</i> method), 63 | <code>with_predict_key()</code> | (<i>rastervision.experiment.ExperimentConfigBuilder</i> method), 58 |
| <code>with_model_defaults()</code> | (<i>rastervision.backend.TFDeeplabConfigBuilder</i> method), 65 | <code>with_predict_options()</code> | (<i>rastervision.task.ObjectDetectionConfigBuilder</i> method), 61 |
| <code>with_model_defaults()</code> | (<i>rastervision.backend.TFObjectDetectionConfigBuilder</i> method), 64 | <code>with_predict_package_uri()</code> | (<i>rastervision.task.ChipClassificationConfigBuilder</i> method), 60 |
| <code>with_model_uri()</code> | (<i>rastervision.backend.KerasClassificationConfigBuilder</i> method), 63 | <code>with_predict_package_uri()</code> | (<i>rastervision.task.ObjectDetectionConfigBuilder</i> method), 61 |
| <code>with_model_uri()</code> | (<i>rastervision.backend.TFDeeplabConfigBuilder</i> method), 65 | <code>with_predict_package_uri()</code> | (<i>rastervision.task.SemanticSegmentationConfigBuilder</i> method), 62 |
| <code>with_model_uri()</code> | (<i>rastervision.backend.TFObjectDetectionConfigBuilder</i> method), 64 | <code>with_predict_uri()</code> | (<i>rastervision.experiment.ExperimentConfigBuilder</i> method), 58 |
| <code>with_num_epochs()</code> | (<i>rastervision.backend.KerasClassificationConfigBuilder</i> method), 63 | <code>with_pretrained_model()</code> | (<i>rastervision.backend.KerasClassificationConfigBuilder</i> method), 63 |
| <code>with_num_steps()</code> | (<i>rastervision.backend.TFDeeplabConfigBuilder</i> method), 65 | <code>with_pretrained_model()</code> | (<i>rastervision.backend.TFDeeplabConfigBuilder</i> method), 65 |
| <code>with_num_steps()</code> | (<i>rastervision.backend.TFObjectDetectionConfigBuilder</i> method), 64 | <code>with_pretrained_model()</code> | (<i>rastervision.backend.TFObjectDetectionConfigBuilder</i> method), 64 |
| <code>with_output_uri()</code> | (<i>rastervision.evaluation.ChipClassificationEvaluatorConfigBuilder</i> method), 72 | <code>with_probability()</code> | (<i>rastervision.augmentor.NodataAugmentorConfigBuilder</i> method), 71 |
| <code>with_output_uri()</code> | (<i>rastervision.evaluation.ObjectDetectionEvaluatorConfigBuilder</i> method), 73 | <code>with_raster_source()</code> | (<i>rastervision.data.SceneConfigBuilder</i> method), 66 |
| <code>with_output_uri()</code> | (<i>rastervision.evaluation.SemanticSegmentationEvaluatorConfigBuilder</i> method), 73 | <code>with_raster_source()</code> | (<i>rastervision.data.SemanticSegmentationRasterSourceConfigBuilder</i> method), 70 |
| <code>with_pick_min_class_id()</code> | (<i>rastervision.data.ChipClassificationGeoJSONSourceConfigBuilder</i> method), 69 | <code>with_rasterizer_options()</code> | (<i>rastervision.data.GeoJSONSourceConfigBuilder</i> method), 68 |
| <code>with_predict_batch_size()</code> | (<i>rastervision.task.ChipClassificationConfigBuilder</i> method), 60 | | |

`with_rgb()` (*rastervision.data.SemanticSegmentationRasterStoreConfigBuilder* method), 58
`with_rgb_class_map()` (*rastervision.data.SemanticSegmentationRasterSourceConfigBuilder* method), 63
`with_root_uri()` (*rastervision.experiment.ExperimentConfigBuilder* method), 58
`with_script_locations()` (*rastervision.backend.TFDeeplabConfigBuilder* method), 65
`with_script_locations()` (*rastervision.backend.TFObjectDetectionConfigBuilder* method), 64
`with_stats_analyzer()` (*rastervision.experiment.ExperimentConfigBuilder* method), 58
`with_stats_transformer()` (*rastervision.data.GeoJSONSourceConfigBuilder* method), 68
`with_stats_transformer()` (*rastervision.data.GeoTiffSourceConfigBuilder* method), 67
`with_stats_transformer()` (*rastervision.data.ImageSourceConfigBuilder* method), 67
`with_stats_uri()` (*rastervision.analyzer.StatsAnalyzerConfigBuilder* method), 72
`with_stats_uri()` (*rastervision.data.StatsTransformerConfigBuilder* method), 71
`with_task()` (*rastervision.backend.KerasClassificationConfigBuilder* method), 63
`with_task()` (*rastervision.backend.TFDeeplabConfigBuilder* method), 65
`with_task()` (*rastervision.backend.TFObjectDetectionConfigBuilder* method), 64
`with_task()` (*rastervision.data.SceneConfigBuilder* method), 66
`with_task()` (*rastervision.evaluation.ChipClassificationEvaluatorConfigBuilder* method), 72
`with_task()` (*rastervision.evaluation.ObjectDetectionEvaluatorConfigBuilder* method), 73
`with_task()` (*rastervision.evaluation.SemanticSegmentationEvaluatorConfigBuilder* method), 73
`with_task()` (*rastervision.experiment.ExperimentConfigBuilder* method), 58
`with_template()` (*rastervision.backend.KerasClassificationConfigBuilder* method), 63
`with_template()` (*rastervision.backend.TFDeeplabConfigBuilder* method), 65
`with_template()` (*rastervision.backend.TFObjectDetectionConfigBuilder* method), 64
`with_test_scene()` (*rastervision.data.DatasetConfigBuilder* method), 59
`with_test_scenes()` (*rastervision.data.DatasetConfigBuilder* method), 59
`with_train_key()` (*rastervision.experiment.ExperimentConfigBuilder* method), 58
`with_train_options()` (*rastervision.backend.KerasClassificationConfigBuilder* method), 63
`with_train_options()` (*rastervision.backend.TFDeeplabConfigBuilder* method), 65
`with_train_options()` (*rastervision.backend.TFObjectDetectionConfigBuilder* method), 64
`with_train_scene()` (*rastervision.data.DatasetConfigBuilder* method), 59
`with_train_scenes()` (*rastervision.data.DatasetConfigBuilder* method), 59
`with_train_uri()` (*rastervision.experiment.ExperimentConfigBuilder* method), 58
`with_training_data_uri()` (*rastervision.backend.KerasClassificationConfigBuilder* method), 63
`with_training_data_uri()` (*rastervision.backend.TFDeeplabConfigBuilder* method), 65
`with_training_data_uri()` (*rastervision.backend.TFObjectDetectionConfigBuilder* method), 64
`with_training_output_uri()` (*rastervision.backend.KerasClassificationConfigBuilder* method), 63
`with_training_output_uri()` (*rastervision.backend.TFDeeplabConfigBuilder* method), 65
`with_training_output_uri()` (*rastervision.backend.TFObjectDetectionConfigBuilder* method), 64


```

        sion.backend.TFObjectDetectionConfigBuilder
        method), 64
with_transformer() (rastervision.data.GeoJSONSourceConfigBuilder
        method), 68
with_transformer() (rastervision.data.GeoTiffSourceConfigBuilder
        method), 67
with_transformer() (rastervision.data.ImageSourceConfigBuilder method),
        67
with_transformers() (rastervision.data.GeoJSONSourceConfigBuilder
        method), 68
with_transformers() (rastervision.data.GeoTiffSourceConfigBuilder
        method), 67
with_transformers() (rastervision.data.ImageSourceConfigBuilder method),
        68
with_uri() (rastervision.data.ChipClassificationGeoJSONSourceConfigBuilder
        method), 69
with_uri() (rastervision.data.ChipClassificationGeoJSONStoreConfigBuilder
        method), 70
with_uri() (rastervision.data.GeoJSONSourceConfigBuilder
        method), 68
with_uri() (rastervision.data.GeoTiffSourceConfigBuilder
        method), 67
with_uri() (rastervision.data.ImageSourceConfigBuilder method),
        68
with_uri() (rastervision.data.ObjectDetectionGeoJSONSourceConfigBuilder
        method), 69
with_uri() (rastervision.data.ObjectDetectionGeoJSONStoreConfigBuilder
        method), 70
with_uri() (rastervision.data.SemanticSegmentationRasterStoreConfigBuilder
        method), 71
with_uris() (rastervision.data.GeoTiffSourceConfigBuilder
        method), 67
with_use_intersection_over_cell()
        (rastervision.data.ChipClassificationGeoJSONSourceConfigBuilder
        method), 69
with_validation_scene() (rastervision.data.DatasetConfigBuilder
        method),
        59
with_validation_scenes() (rastervi-
```