Hugin EO

Release 0.2.0b3

Nov 12, 2019

Contents

Hugin helps scientists run Machine Learning experiments on geospatial raster data.

Hugin was developed as part of the ESA funded ML4EO

Overall Hugin aims to facilitate experimentation with multiple machine learning problems, like:

- Classification
- Segmentation
- Super-Resolution

Currently Hugin builds on top of the Keras machine learning library but it also aims to support, in the future, additional backends like scikit-learn.

CHAPTER 1

Installation

1.1 Prerequisites

Hugin builds functionality on top of existing technology, primarily it uses Keras, SciKit-Learn, OpenCV and RasterIO.

The exact prerequisites are specified in the *requirements.txt* and *setup.py* files. Normally you package manager will handle requirement installation automatically.

1.2 Using pip

1.2.1 From PyPi

```
pip install hugin
```

1.2.2 From GitHub

You can install Hugin using the following command:

```
pip install git+http://github.com/sage-group/hugin#egg=hugin
```

1.3 From source code

When installing from source code we recommend installation inside a specially created virtual environment.

Installing from source code involves running the *setup.py* inside you python environment.

```
python setup.py install
```

CHAPTER 2

Introduction

Hugin is meant to be used in two scenarios:

- as a standalone tool driven a by an experiment configuration file
- as a library in your code

Both scenarios share same concepts with the main difference that the standalone tool connects all the Hugin components together.

2.1 Standalone

Using Hugin involves two steps:

- training
- prediction

Both steps are driven using dedicated configuration files. The configuration files are normal YAML files referencing various Hugin components.

This configuration files allow the end user to customize pre-processing, model and post-processing operations.

2.1.1 Training

The training process involves the preparation of a training scenario configuration file. This configuration file is composed out of multiple sections, particularly:

- Global configuration (the *configuration* key)

- Data source specification (the *trainer* key)
- Trainer specification (the *data_source* key)

2.1.1.1 Global Configuration

Currently in this section (the *configuration* key in YAML file) you can specify:

- *model_path*: a string specifying the "workspace" used for saving the model, and depending on the backend it will hold checkpoints, metrics, etc. This string allows interpolation of trainer attributes.

An example configuration specification could be:

```
configuration:
 model_path: "/home/user/experiments/{name}"
```

2.1.1.2 Data Source Specification

The data source is intended for locating the data we wish to use in our experiments. As part of Hugin there are multiple data source implementations, particularly:

- *FileSystemLoader*: capable of scanning, recursively, a directory for input files and group them together according to a specified pattern.
- *FileLoader*: capable of reading file names from an input file. The main purpose of this file is for supporting GDAL Virtual File Systems, for example:
  - */vsicurl/*: for retrieving files using cURL (HTTP, FTP, etc)
  - */vsis3/*: for retrieving files from AWS S3
  - */vsigs/*: for retrieving files from Google Cloud Storage

The data source that should be used is introduced using the YAML *data_source* key in the YAML file and is an explicit reference to the data source implementation.

The aforementioned data sources can have the following configuration options:

- *data_pattern* **(mandatory)**: used for specifying a regular expression matching files that should be taken into consideration
- *id_format* **(mandatory)**: used for constructing an *scene id* used by Hugin for identifying a particular scene. This option is similar to the SQL *GROUP BY* statement
- *type_format* **(mandatory)**: used for identifying the various potential types of data in a scene
- *validation_percent* **(optional)**: used for specifying the number of scenes that should be kept for validation purposes
- *randomise* **(optional, default: 'False')**: asks the data source to provide the scenes to the other components in a randomized order

- *persist_file* **(optional)**: specifies a path where the data source should save the detected files. In case it exists it is used as source for further operation. The main benefit of this configuration option is the ability to reuse the same training/validation split between multiple runs.
- *input_source* **(mandatory)**: specifies a location for loading the data. For the *FileSystemLoader* it represents a directory that should be scanned. For *FileLoader* it represents an input text file listing all files that should be taken into consideration (on file path per line)

An example configuration for loading the data from the SpaceNet5 competition:

```
data_source: !!python/object/apply:hugin.io.FileSystemLoader
 kwds:
   data_pattern: '(?P<category>[0-9A-Za-z_]+)_AOI_(?P<location>\d+(_[A-Za-z0-
↪9]+)+)_(?P<type>(PS-MS|PS-RGB|MS|PAN))_(?P<idx>[A-Za-z0-9]+)(?P<gti>_GTI)?.(?P
↪<extension>(tif|tiff|png|jpg|jp2))$'
   id_format: '{location}-{idx}'
   type_format: '{type}{gti}'
   validation_percent: 0.2
   randomise: True
   persist_file: "/storage/spacenet5/split1.yaml"
   input_source: "/storage/spacenet5"
```

2.1.1.3 Model Configuration

This section is aimed for configuring the effective training operation.

The primary key specifying the training operation is the *trainer* key in the YAML file. Currently Hugin only supports handling of raster operation (handling images of various kinds) through the *RasterSceneTrainer*

The *RasterSceneTrainer* implementation offers multiple features like:

- **Tiling** (subsampling): splitting input scenes in multiple smaller scenes. This is particularly useful for large inputs where the input can not fit in GPU memory. Hugin support overlapping tiles using a specific stride.
- **Co-registration**: synchronize input tiles from the various components forming a scene (Eg. a scene might be composed out of data in multiple resolutions: for WorldView-3 we might have an panchromatic channel with *0.31m* spatial resolution and multi-spectral data with *1.24m* resolution per pixel)
- **Pre-Processing**: applying a series of preprocessing operation on the data before it is ingested by models. Some of the operations supported include standardization, augmentation, etc.

The *RasterSceneTrainer* assembles the data according to a user specified mapping and feeds the data to a model implementation specified by the user. Both the mapping and the model implementation will be discussed in the following sections.

The options supported by the *RasterSceneTrainer* are:

- *name* **(mandatory)**: specifies a name for the trainer. This name is used in multiple locations, particularly for identifying the model in the experiment workspace (discussed in *Global Configuration*)
- *window_size* **(optional)**: specifies the size of the sliding window used for subsampling. If omitted Hugin assumes that it equals the size of one of the randomly picked scenes
- *stride_size* **(optional)**: specifies the stride size to be used in case subsampling is needed. If omitted it is inferred from the window size
- *mapping* **(mandatory)**: this configuration option specifies how the input to the model should be assembled. This configuration might be shared both between training and prediction time. It is further discussed in (discussed in *Mapping* section)
- *model* **(mandatory)** specifies to model to be used for training

Mapping

The mapping concept is further discussed in the *Mapping* section. One specific requirement related to training is the presence of the *target* mapping. It is needed for specifying the expected output (ground truth) from the various machine learning models.

Model

This configuration option specifies the model to be trained. It is a reference to one of the backend implementations offered by Hugin:

- *KerasModel*: The backend supporting running Keras based models
- *SkLearnStandardizer*: A custom backend based on SciKit-Learn for training an SciKit-Learn data standardizer
- *SciKitLearnModel*: A backend for supporting model complient to the SciKit-Learn interface (ToDo)

Keras Model

The *KerasModel* implementation allow running models defined using Keras. It exposes the following options:

- *name* **(mandatory)**: Option specifying the name of the model
- *model_path* **(optional)**: The location of the trained model. If it exists it is loaded and training resumes from the loaded state. This is particularly useful for transfer learning
- *model_builder* **(mandatory)**: The function to be called for building the model

- *loss* **(mandatory)**: Loss function to be used by Keras during training. Any Keras loss can be referenced, or used defined functions
- *optimizer* **(optional)**: Optimizer function to be used during training. Any Keras optimizer can be referenced
- *batch_size* **(mandatory)**: The batch size to be used for feeding the data to the model
- *epochs* **(mandatory)**: The maximum number of epochs to run
- *metrics* **(optional)**: A list of metrics to be computed during training
- *checkpoint* **(optional)**: If defined it enables model checkpoints according to specified configuration. It allows setting the following options:
  - *save_best_only* **(default: False)**: Saves only the best model
  - *save_weights_only* **(default: False)**: Save only the model weights
  - *mode* **(valid options: auto, min, max)**: Save models based on either the maximization or the minimization of the monitored quantity. This only applies when *save_best_only* is enabled
  - *monitor*: quantity to be monitored (eg. *val_loss* or any user defined metric)
- *enable_multi_gpu* **(optional, default=False)**: enable multiple GPU usage
- *num_gpus* **(optional)**: number of GPUs to be used by Keras
- *callbacks* **(optional)**: list of Keras callbacks to be enabled. List is composed out of Keras callbacks or compatible user defined callbacks.

An example configuration:

```
model: !!python/object/apply:hugin.engine.keras.KerasModel
  kwds:
    name: keras_model1
    model_builder: sn5.models.wnet.wnetv9:build_wnetv9
    batch_size: 200
    epochs: 9999
    metrics:
      - accuracy
      - !!python/name:hugin.tools.utils.dice_coef
      - !!python/name:hugin.tools.utils.jaccard_coef
    loss: categorical_crossentropy
    checkpoint:
      monitor: val_loss
    enable_multi_gpu: True
    num_gpus: 4
    optimizer: !!python/object/apply:keras.optimizers.Adam
      kwds:
        lr: !!float 0.0001
        beta_1: !!float 0.9
        beta_2: !!float 0.999
```

(continues on next page)

(continued from previous page)

```
        epsilon: !!float 1e-8
    callbacks:
      - !!python/object/apply:keras.callbacks.EarlyStopping
        kwds:
          monitor: 'val_dice_coef'
          min_delta: 0
          patience: 40
          verbose: 1
          mode: 'auto'
          baseline: None
          restore_best_weights: False
```

Limitations

- Hugin assumes all scenes have an equal size per data type (eg. all multispectral data has the same size).
- Hugin only support square sliding windows. This is expected to be fixed in an upcoming version
- Hugin only support the same stride size both horizontally and vertically

2.1.1.4 Example Experiment

A complete example configuration is depicted bellow:

```
configuration:
 model_path: "/home/user/experiments/{name}"
data_source: !!python/object/apply:hugin.io.FileSystemLoader
 kwds:
   data_pattern: '(?P<category>[0-9A-Za-z_]+)_AOI_(?P<location>\d+(_[A-Za-z0-
↪9]+)+)_(?P<type>(PS-MS|PS-RGB|MS|PAN))_(?P<idx>[A-Za-z0-9]+)(?P<gti>_GTI)?.(?P
↪<extension>(tif|tiff|png|jpg|jp2))$'
   id_format: '{location}-{idx}'
   type_format: '{type}{gti}'
   validation_percent: 0.2
   randomise: True
   persist_file: "/storage/spacenet5/split1.yaml"
   input_source: "/storage/spacenet5"
trainer: !!python/object/apply:hugin.infer.scene.RasterSceneTrainer
         kwds:
           name: raster_keras_trainerv2
           stride_size: 100
           window_size: [256, 256]
           model: !!python/object/apply:hugin.engine.keras.KerasModel
             kwds:
               name: keras_model1
```

(continues on next page)

(continued from previous page)

```
            model_builder: sn5.models.wnet.wnetv9:build_wnetv9
            batch_size: 200
            epochs: 9999
            metrics:
              - accuracy
              - !!python/name:hugin.tools.utils.dice_coef
              - !!python/name:hugin.tools.utils.jaccard_coef
            loss: categorical_crossentropy
            checkpoint:
              monitor: val_loss
            enable_multi_gpu: True
            num_gpus: 4
            optimizer: !!python/object/apply:keras.optimizers.Adam
              kwds:
                lr: !!float 0.0001
                beta_1: !!float 0.9
                beta_2: !!float 0.999
                epsilon: !!float 1e-8
            callbacks:
              - !!python/object/apply:keras.callbacks.EarlyStopping
                kwds:
                  monitor: 'val_dice_coef'
                  min_delta: 0
                  patience: 40
                  verbose: 1
                  mode: 'auto'
                  baseline: None
                  restore_best_weights: False
        mapping:
          inputs:
            input_1:
              primary: True
              channels:
                - [ "PAN", 1 ]
              window_size: [256, 256]
            input_2:
              window_size: [64, 64]
              channels:
                - [ "MS", 1 ]
                - [ "MS", 5 ]
                - [ "MS", 4 ]
                - [ "MS", 8 ]
          target:
            output_1:
              channels:
                - [ "PAN_GTI", 1 ]
              preprocessing:
                - !!python/object/apply:hugin.io.loader.
↪BinaryCategoricalConverter
```

(continues on next page)

(continued from previous page)

```
                    kwds:
                      do_categorical: False
```

Assuming that the above configuration is saved in a file named *experiment.yaml*, training can be started as follows:

```
hugin trainv2 --config experiment.yaml
```

2.1.2 Prediction

2.1.3 Mapping

The data mapping functionality represents one of the core features of Hugin. It is used by the *RasterSceneTrainer* and *RasterScenePredictor* for assembling input data that is sent to the underlying models.

CHAPTER 3

Indices and Tables