
Mantra Documentation

Release 0.960

Robert Stojnic & Ross Taylor

Oct 12, 2018

Contents

1	Installation	3
2	Get Started	5
2.1	Making Models with Mantra	6
2.2	Packaging Datasets with Mantra	11
2.3	Defining Tasks with Mantra	16
2.4	Training Models with Mantra	18

Mantra is a deep learning development kit that manages the various components in an deep learning project, and makes it much easier to do routine tasks like training in the cloud, model monitoring, model benchmarking and more. It works with your favourite deep learning libraries like TensorFlow, PyTorch and Keras.

You might like mantra if:

- You need to structure deep learning projects: versioning, monitoring and storage.
- You need devops tasks like cloud integration and file syncing taken care for you.
- You need to evaluate your model against benchmark tasks, e.g. CIFAR-10 accuracy.

CHAPTER 1

Installation

Mantra is a Python that you can install via pip:

```
$ pip install mantraml
```

It is currently tested on Python 3.5-7.

Additional dependencies you need to install are TensorFlow or PyTorch depending on which framework you want to use. If you want to use the TensorBoard feature of Mantra with PyTorch then you should also install TensorboardX.

CHAPTER 2

Get Started

Find a directory where you want to create a project and run:

```
$ mantra launch my_project_name
```

This will create a **my_project_name** directory with a folder structure like this:

```
data/  
models/  
tasks/  
trials/  
__init__.py  
mantra.yml  
README.md  
settings.py
```

- The `data/` folder contains your datasets
- The `models/` folder contains your models
- The `tasks/` folder contains your tasks
- The `trials/` folder contains trial data (data when you train a model)
- The `mantra.yml` file contains project metadata
- The `settings.py` file contains project settings

Now we have a project! To view the current project through the Mantra UI, execute the following from your project root:

```
$ mantra ui
```

Now that you are ready, it's time to learn how Mantra models and datasets work!

- [Get started with Mantra models](#)
- [Get started with Mantra datasets](#)

- [Get started with Mantra tasks](#)
- [Get started with Mantra training](#)

2.1 Making Models with Mantra

Mantra models allow you to take a model in an framework such as TensorFlow or PyTorch, and with a few modifications, allows them to be easily trained, deployed, evaluated and more. In these docs we are going to see how we make a model package.

2.1.1 Make a Model

Go to the root of your project. To make a new model we can use the `makemodel` command:

```
$ mantra makemodel my_model
```

If we intend to use a particular deep learning framework, we can reference a template:

```
$ mantra makemodel my_model --template tensorflow
```

```
$ mantra makemodel my_model --template keras
```

```
$ mantra makemodel my_model --template pytorch
```

Our new model folder will be located at **myproject/models/my_model**. Inside:

```
__init__.py
default.jpg
model.py
notebook.ipynb
README.md
```

- `model.py` contains your core model logic
- `notebook.ipynb` is a notebook which you can use for prototyping
- `README.md` is where you can describe the model (useful for sharing the model with others)

Let's have a look at the `model.py` file and see what the template contains:

```
from mantraml.models import MantraModel

class MyModel(MantraModel):
    model_name = "My Model"
    model_image = "default.jpg"
    model_notebook = 'notebook.ipynb'
    model_tags = ['new']

    def __init__(self, data=None, task=None, **kwargs):
        self.data = data
        self.task = task

    def run(self):
        return
```

(continues on next page)

(continued from previous page)

```
def predict(self, X):
    return
```

The first thing we observe is that we inherit from `MantraModel`. This is a simple step - if you have existing machine learning code in a class then you can just inherit from `MantraModel` to gain access to most of the Mantra integration.

The second thing we observe is we just need to pass in a data and task argument when initializing the class - this is what allows Mantra models to be composable with different datasets and evaluation criteria.

The third thing we observe is the `run` and `predict` methods. All you need to do is to trigger your training code from the `run` method. When Mantra trains the model, it will call this method first and the rest is history. For optional evaluation, you just need to write a `predict` method.

This is the core design pattern of Mantra models.

Now let's see some examples for each framework: Keras, Base TensorFlow and PyTorch.

2.1.2 Model Building in Keras



Here is an example of a deep CNN using Keras:

```
import tensorflow
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Activation, Flatten
from tensorflow.keras.layers import Conv2D, MaxPooling2D

from mantraml.models import MantraModel
from mantraml.models.keras.callbacks import TensorBoard, StoreTrial, EvaluateTask, \
↳ ModelCheckpoint

class DeepCNN(MantraModel):
    model_name = "Deep Convolutional Model"
    model_image = "default.jpg"
    model_notebook = 'notebook.ipynb'
    model_tags = ['cnn', 'classification']

    def __init__(self, data=None, task=None, **kwargs):
        self.data = data
        self.task = task

        self.dropout = kwargs.get('dropout', 0.25)
        self.optimizer = kwargs.get('optimizer', 'adam')
        self.loss = kwargs.get('loss', 'categorical_crossentropy')
        self.metrics = kwargs.get('metrics', ['accuracy'])

        if self.task:
            self.X = self.task.X_train
            self.y = self.task.y_train
        else:
            self.X = self.data.X
```

(continues on next page)

(continued from previous page)

```

        self.y = self.data.y

    def run(self):
        num_classes = self.data.X.shape[1]

        model = Sequential()
        model.add(Conv2D(32, (3, 3), padding='same', input_shape=self.data.X.
↪shape[1:]))
        model.add(Activation('relu'))
        model.add(Conv2D(32, (3, 3)))
        model.add(Activation('relu'))
        model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(Dropout(self.dropout))

        model.add(Conv2D(64, (3, 3), padding='same'))
        model.add(Activation('relu'))
        model.add(Conv2D(64, (3, 3)))
        model.add(Activation('relu'))
        model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(Dropout(self.dropout))

        model.add(Flatten())
        model.add(Dense(512))
        model.add(Activation('relu'))
        model.add(Dropout(self.dropout))
        model.add(Dense(num_classes))
        model.add(Activation('softmax'))

        model.compile(loss=self.loss, optimizer=self.optimizer, metrics=self.metrics)

        self.model = model

        tb_callback = TensorBoard(mantra_model=self, write_graph=True, write_
↪images=True)
        exp_callback = StoreTrial(mantra_model=self)
        eval_callback = EvaluateTask(mantra_model=self)
        checkpoint_callback = ModelCheckpoint(mantra_model=self)

        callbacks = [tb_callback, eval_callback, checkpoint_callback, exp_callback]

        self.model.fit(self.X, self.y, epochs=self.n_epochs, batch_size=self.n_batch,
                        callbacks=callbacks)

    def predict(self, X):
        return self.model.predict(X)

```

Let's briefly analyse this code:

- We've inherited from MantraModel
- We've specified our data `self.X` and `self.y`
- We've written our model logic in `run`

There's one final thing to note. Keras reports results to you through callbacks. To get nice results reported to us through the Mantra UI, we simply need to add some or all of the following callbacks:

```
tb_callback = TensorBoard(mantra_model=self, write_graph=True, write_images=True)
exp_callback = StoreTrial(mantra_model=self)
eval_callback = EvaluateTask(mantra_model=self)
checkpoint_callback = ModelCheckpoint(mantra_model=self)
```

This will configure things so your logs, media, and weights are managed and versioned correctly; and that you can monitor and evaluate results through the UI.

And that's it, your model is Mantra ready!

For more more Keras model examples, check out the [Mantra examples repository](#).

2.1.3 Model Building in TensorFlow



The class structure is the same as the Keras example. But now we use the following callbacks:

```
from mantraml.models.tensorflow.summary import FileWriter
from mantraml.models.tensorflow.callbacks import ModelCheckpoint, EvaluateTask, \
↳StoreTrial, SavePlot
```

To configure your TensorFlow code for Mantra, use the mantra FileWriter instead of the TensorFlow FileWriter. For example:

```
self.writer = FileWriter(mantra_model=self)
```

Then at the end of each epoch of training, use the following callbacks:

```
ModelCheckpoint(mantra_model=self, session=self.session)
if self.task:
    EvaluateTask(mantra_model=self)
StoreTrial(mantra_model=self, epoch=epoch)
```

Just as with Keras callbacks, this will configure things so your logs, media, and weights are managed and versioned correctly; and that you can monitor and evaluate results through the UI.

For TensorFlow model examples, check out the [Mantra examples repository](#).

2.1.4 Model Building in PyTorch



The class structure is the same as the previous examples. But now we use the following callbacks:

```
from mantraml.models.pytorch.summary import SummaryWriter
from mantraml.models.pytorch.callbacks import ModelCheckpoint, EvaluateTask, \
↳StoreTrial, SavePlot
```

Mantra works with TensorBoardX for PyTorch. Use the mantra SummaryWriter instead of the TensorBoardX SummaryWriter:

```
self.writer = SummaryWriter (mantra_model=self)
```

Then at the end of each epoch of training, use the following callbacks:

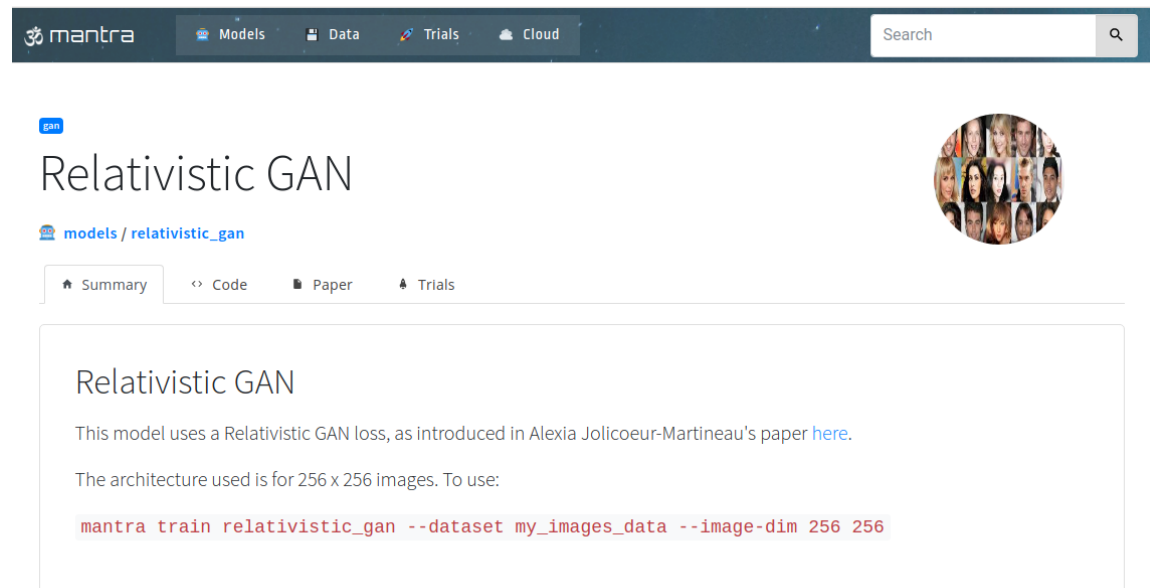
```
ModelCheckpoint (mantra_model=self, session=self.session)
if self.task:
    EvaluateTask (mantra_model=self)
StoreTrial (mantra_model=self, epoch=epoch)
```

For PyTorch model examples, check out the [Mantra examples repository](#).

2.1.5 Visualizing Your Model Projects

Load up the UI and click on a model:

```
$ mantra ui
```



In order to customise how the UI looks for your model you can add metadata to your model classes:

```
class DeepCNN (MantraModel) :

    # The Name of the Model
    model_name = "Relativistic GAN"

    # The Model Image
    model_image = "default.jpg"

    # Link to a Notebook
    model_notebook = 'notebook.ipynb'

    # Tags for the Model
    model_tags = ['cnn', 'classification']

    # ArXiv Link
```

(continues on next page)

(continued from previous page)

```
model_arxiv_id = '1807.00734'

# Custom Paper PDF (instead of an ArXiv PDF)
model_pdf = 'my_paper.pdf'
```

Then when you share the model with your collaborators, they won't just get code - they'll get a whole project they can visualize and interact with: including notebooks, the paper the model was based on, and more!

2.1.6 Magic Hyperparameters

Write some custom hyperparameters in your `__init__` function:

```
def __init__(self, data=None, task=None, **kwargs):
    self.dropout = kwargs.get('dropout', 0.25)
    self.optimizer = kwargs.get('my_optimizer', 'adam')

    ...
```

When you train you can automatically reference these hyperparameters without writing command parser code, i.e. this works out of the box:

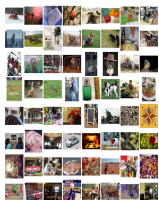
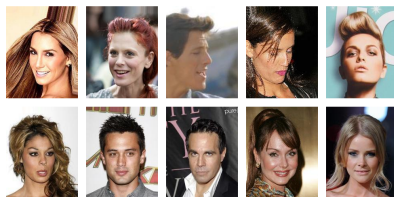
```
$ mantra train my_model --dataset my_dat --dropout 0.5 --my-optimizer 'adam'
```

Note that there are two core hyperparameters that are supported by default: `self.batch_size` and `self.epochs`. These can be adjusted through the command line:

```
$ mantra train my_model --dataset my_dat --epochs 100 --batch-size 32
```

2.2 Packaging Datasets with Mantra

```
0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4 4
5 5 5 5 5 5 5 5 5 5
6 6 6 6 6 6 6 6 6 6
7 7 7 7 7 7 7 7 7 7
8 8 8 8 8 8 8 8 8 8
9 9 9 9 9 9 9 9 9 9
```



With **Mantra** it's easy to package data for deep learning. In these docs, we are going to see how we make a data package, and how we process it using the powerful `Dataset` class.

2.2.1 Make a Dataset

Go to the root of your project. To make a dataset use the `makedata` command. We can make an empty dataset as follows:

```
$ mantra makedata first_dataset
```

Or if we already have a `tar.gz` file with some data, we can reference it as follows:

```
$ mantra makedata first_dataset --tar-path tar_path_here
```

Our new data folder will be located at **myproject/data/first_dataset**. Inside:

```
raw/
__init__.py
data.py
README.md
```

- `data.py` contains the core `Dataset` class that is used to process your data
- `raw/` contains the `tar.gz` file with the raw data
- `README.md` is where you can describe the model (useful for sharing the model with others)

If we don't need flat files, but want to import data through an API, we can use the `no-tar` flag:

```
$ mantra makedata first_dataset --no-tar
```

We now need to extract the input and output vectors `X`, `y` that are used to train models ...

2.2.2 Magic Data Templates

Many datasets are standardised, such as a folder of images or a csv file with columns of features and labels. Mantra provides magic templates so you don't have to write the entire class yourself.

Images

If we have a `tar` file that contains a folder of images, we can use the `images` template:

```
$ mantra makedata celeba --template 'images' --tar-path celebA.tar.gz --image-dim 128_
↪128
```

Above we are using the `images` template. This will create an `ImageDataset` class using the `tar` file provided. We can also specify additional default options for the template:

Parameter	Type	Example	Description
<code>-image-dim</code>	list	64 64	Desired image dimension (height, output)
<code>-normalize</code>	bool (flag)	<code>-normalize</code>	Whether to normalize the images for training

Once we have executed the command, we can open the `data.py` file:


```
import numpy as np

from mantraml.data import Dataset, cachedata
from mantraml.data import ImageDataset

class MyImageDataset(ImageDataset):
    data_name = 'My Image Dataset'
    data_tags = ['example', 'new', 'images']
    files = ['celebA.tar.gz']
    image_dataset = 'celebA.tar.gz' # referring to the file that contains the images

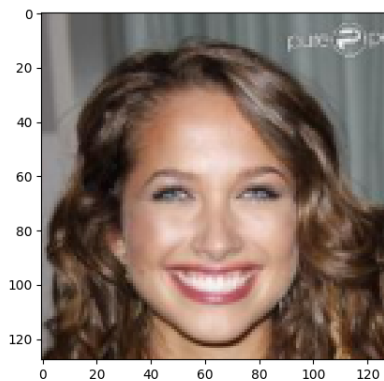
    # additional default data
    has_labels = False
    image_dim = (128, 128)
    normalized = True

    @cachedata
    def y(self):
        # return your labels here as an np.ndarray
        # if no labels, e.g. generative models, then you can remove this method
        return
```

We can see that we are inheriting from ImageDataset. We can also see our input dimensions have entered as a default argument. We can use `sample` to eyeball the data:

```
from data.celeba.data import MyImageDataset

dataset = MyImageDataset(name='celeba')
dataset.sample()
```



So the advantage of using a template is that we didn't have to write any code. We could, if we wish though, write on top of these templates for some further customisation if we needed it.

Tables

If we have a flat csv file, we can use the `tabular` template to configure it:

```
$ mantraml makedata table_data --template 'tabular' --tar-path mydata.tar.gz
$ --file-name 'my_flat_file.csv' --target 'target_column'
$ --features 'feature_1' 'feature_2'
```

This will create an `TabularDataset` class. We can also specify additional options for the template.

Parameter	Type	Example	Description
<code>--file-name</code>	str	<code>'my_flat_file.csv'</code>	The name of the flat file inside the tar
<code>--target</code>	str	<code>'target_column'</code>	The column name to extract as the target
<code>--features</code>	list	<code>'feature_1' 'feature_2'</code>	The columns to extract as the features
<code>--target-index</code>	int	0	The column index of the target
<code>--features-index</code>	list	1 2	The column indices to extract as features

The index options are there if we want to refer to the table by indices rather than column names; if we just want to use column names then we can ignore these options.

Once we have executed the command, we can open the `data.py` file:

```
import numpy as np

from mantraml.data import TabularDataset

class MyTabularDataset(TabularDataset):

    data_name = 'Example Table Data'
    files = ['mydata.tar.gz']
    data_file = 'my_flat_file.csv'
    data_tags = ['tabular']
    has_labels = True
    target = 'target_column'
    features = ['feature_1', 'feature_2']
```

We can see that we are inheriting from `TabularDataset`. We can also see our feature and target options are now default argument options. This dataset is now Mantra ready. If we want to alter features from the command line:

```
$ mantraml train my_model --dataset table_data --features feature_1 feature_2 feature_
→ 3
```

2.2.3 Custom Data Processing

If the magic templates aren't useful, you can write your own data processing logic. Open up the `data.py` file:

```
import numpy as np

from mantraml.data import Dataset, cachedata

class MyImageDataset(Dataset):
    data_name = 'My Image Dataset'
    data_tags = ['example', 'new', 'images']
    files = ['myfiles.tar.gz']
    has_labels = False

    @cachedata
    def X(self):
        # return your features here as an np.ndarray
        return

    @cachedata
```

(continues on next page)

(continued from previous page)

```
def y(self):
    # return your labels here as an np.ndarray
    return
```

Simply write your logic for extracting X, y in the above. Your dependency data in `files` will be extracted to a path at `self.self.extracted_data_path`. So if you are extracting data from these files, just open the files from this directory and do what you want with them.

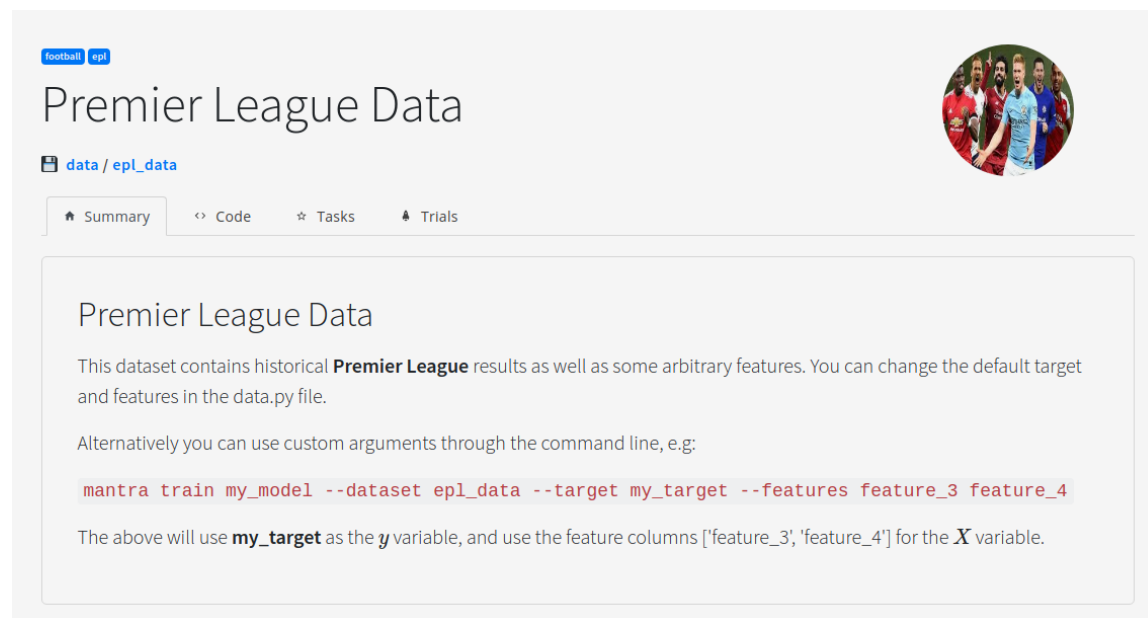
You might be wondering what the `@cachedata` decorator does. It does two things. First it is a property based decorator so you can access the data at `MyImageDataset().X` and `MyImageDataset().y` respectively. Secondly, it caches the data to RAM upon the first call so the processing logic doesn't have to be run twice. If you don't want the caching, then just replace this decorator with `@property`.

For more more Mantra dataset examples, check out the [Mantra examples repository](#).

2.2.4 Visualizing Your Data Projects

Load up the UI and click on a model:

```
$ mantra ui
```



In order to customise how the UI looks for your dataset you can add metadata to your dataset classes:

```
class PremierLeagueData(MantraModel):

    # The Name of the Dataset
    data_name = 'Premier League Data'

    # The Dataset Image
    data_image = "default.jpeg"

    # Link to a Notebook
    data_notebook = 'notebook.ipynb'
```

(continues on next page)

(continued from previous page)

```
# Tags for the Model
data_tags = ['football', 'epl']
```

2.2.5 Accessing Datasets in Models

When you define a model you pass in a `data` and `task` parameter:

```
def __init__(self, data=None, task=None, **kwargs):
    self.data = data
    self.task = task
```

If you don't have a task, then you have no training/test split, and you can simply access the data at `self.data.X` and `self.data.y`.

If you have a task then you can train your model on the training set explicitly at `self.task.X_train` and `self.task.y_train`.

2.3 Defining Tasks with Mantra

In **mantra** a task combines evaluation metrics with definitions of the training/validation/test data. Making a task is like creating your own machine learning competition : here is the training and test set, and here is what your models will be evaluated on.

2.3.1 Make a Task

Go to the root of your project. To make a new model we can use the `makemodel` command:

```
$ mantra maketask my_task
```

Our new task folder will be located at **myproject/tasks/my_task**. Inside:

```
__init__.py
task.py
```

- `task.py` contains your core task logic

Let's have a look at the `task.py` file and see what the template contains:

```
from mantraml.tasks import Task

class MyTask(Task):
    task_name = 'My Example Task'
    evaluation_name = 'My Evaluation Metric'
    training_split = (0.5, 0.25, 0.25)

    def evaluate(self, model):
        # Return an evaluation metric scalar here. For example, categorical cross_
        ↪entropy on the validation set:
        # predictions = model.predict(self.X_val)
```

(continues on next page)

(continued from previous page)

```

        # return -np.nansum(self.y_val*np.log(predictions) + (1-self.y_val)*np.log(1-
↪predictions)) / predictions.shape[0]
        return

```

The `training_split` variable is a tuple that specifies what proportion of the data to use for training, validation and test respectively.

In the `evaluate` function we take a mantra model as an input and evaluate it according to a logic of our choice. In the commented out notes, we see an example for categorical crossentropy that is referencing the validation set in the task.

What this code will do is when we run a model for a dataset on a task, at the end of each epoch, we'll call the `evaluate` function to obtain a metric and this will be stored. So whatever you write in this method will be your evaluation metric.

Let's look at a full example for binary crossentropy:

```

import numpy as np
from sklearn.metrics import accuracy_score

from mantraml.tasks import Task

class BinaryCrossEntropy(Task):
    """
    This class defines a task with binary cross entropy; with a 0.50/0.25/0.25_
↪training/test split
    """

    task_name = 'Classifier Evaluation'
    evaluation_name = 'Binary Crossentropy'
    training_split = (0.5, 0.25, 0.25)
    secondary_metrics = ['accuracy']

    def evaluate(self, model):
        predictions = model.predict(self.X_val)
        return -np.nansum(self.y_val*np.log(predictions) + (1-self.y_val)*np.log(1-
↪predictions)) / predictions.shape[0]

    def accuracy(self, model):
        predictions = model.predict(self.X_val)
        predictions[predictions > 0.5] = 1
        predictions[predictions <= 0.5] = 0
        return accuracy_score(self.y_val, predictions)

```

Here we have also specified secondary metrics - 'accuracy' - that will also be recorded during training.

2.3.2 Using a Task and Comparing Models

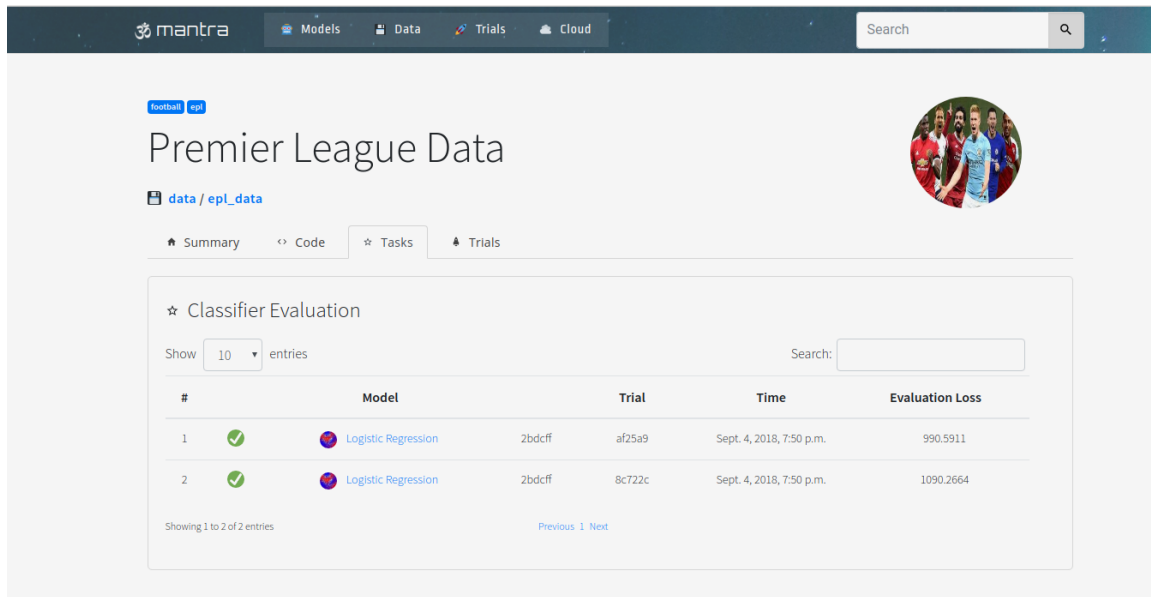
During training, reference the task as follows:

```
$ mantra train my_model --dataset my_data --task my_task
```

Using the UI we can visualize the performance of different models against tasks.

```
$ mantra ui
```

Below we can see a leaderboard for our task:



2.4 Training Models with Mantra

Once you have some datasets and models, you can easily train them (a) locally and (b) on the cloud.

2.4.1 Training Locally

From your project root:

```
$ mantra train my_model --dataset my_dataset --batch-size 64 --epochs 50
```

If you have a task that you want to train with:

```
$ mantra train my_model --dataset my_dataset --task my_task --batch-size 64 --epochs 50
```

Additional magic model hyperparameters can be referenced:

```
$ mantra train my_model --dataset my_dataset --batch-size 64 --epochs 50 --dropout 0.5
```

For image datasets, you can specify things like dimensions:

```
$ mantra train my_model --dataset my_dataset --batch-size 64 --epochs 50 --image-dim 256 256
```

For table datasets, you can specify features and targets:

```
$ mantra train my_model --dataset my_dataset --batch-size 64 --epochs 50 --target my_target --features feature_1 feature_2
```

If you only want to save the best model weights:

```
$ mantra train my_model --dataset my_dataset --batch-size 64 --epochs 50 --savebestonly
```

2.4.2 Training on AWS

To train a model on AWS, first configure your AWS credentials

```
$ mantra cloud
```

You will be asked for your AWS API keys and AWS region preferences. Once complete make sure you have AWS CLI installed - this is a necessary dependency! You will also need to ensure your security group has the right permissions - e.g. ability to create and shut down instances.

Make sure to check the **settings.py** file and ensure that the instance type and AMI you want to launch are right for you. Most of the functionality has been tested with the AWS Deep Learning AMI. Depending on what type of instance you want to launch, you might need to contact AWS to ask them to increase your instance limit.

Danger: RESERVED AWS GPU INSTANCES CAN BE VERY EXPENSIVE TO TRAIN ON. ALWAYS ENSURE YOU ARE AWARE WHAT INSTANCES ARE RUNNING AND IF THEY HAVE BEEN PROPERLY SHUT DOWN OR TERMINATED

To train with the cloud, there are two main options. First you can spin up a reserved instance and close once training is complete. To do this just use the cloud flag:

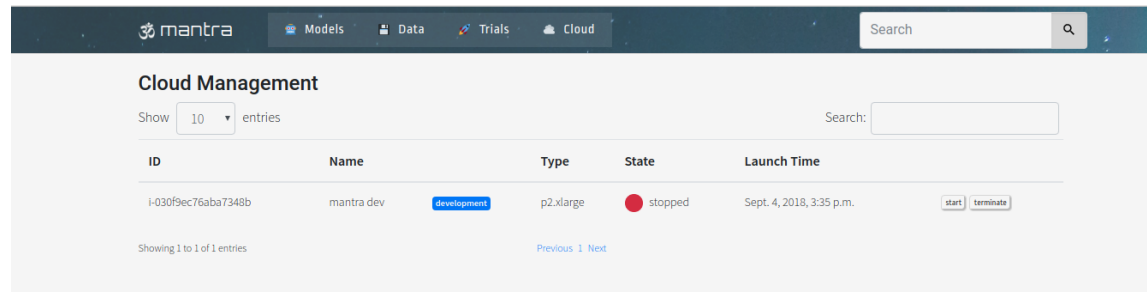
```
$ mantra train my_model --dataset my_dataset --batch-size 64 --epochs 50 --cloud
```

For model development, it's recommended to using a `--dev` flag:

```
$ mantra train my_model --dataset my_dataset --batch-size 64 --epochs 50 --cloud --dev
```

This will create a development instance that isn't terminated when training completes. This means you can use the same instance to run models on - it means setup time is a lot quicker (as all the dependencies are already sorted out). You can still shut this instance down when you're not using it - and when you do need to use it again, training will automatically turn the instance on again.

You can see what mantra GPU instances are running on the cloud tab of the UI:



The screenshot shows the Mantra Cloud Management interface. At the top, there's a navigation bar with tabs for Models, Data, Trials, and Cloud. Below this, the 'Cloud Management' section is visible, featuring a search bar and a table of instances. The table has columns for ID, Name, Type, State, and Launch Time. One instance is listed with ID 'i-030f9ec76aba7348b', Name 'mantra dev', Type 'p2.xlarge', State 'stopped', and Launch Time 'Sept. 4, 2018, 3:35 p.m.'. There are 'start' and 'terminate' buttons for this instance. The interface also shows 'Showing 1 to 1 of 1 entries' and 'Previous 1 Next' navigation links.

ID	Name	Type	State	Launch Time
i-030f9ec76aba7348b	mantra dev	p2.xlarge	stopped	Sept. 4, 2018, 3:35 p.m.

This is no substitute for checking on AWS itself what instances are running - always stay aware!

The other thing to be aware of is S3 storage costs. Mantra uses S3 as a central storage backend for datasets and also data that is generated during training - such as model weights. You can see your bucket name in **settings.py**. Be aware of how much you are currently storing, and if you are cost conscious, then remove files in S3 that you are no longer using.