University of Minho
School of Engineering

ISLab

Deep Learning with TensorFlow™

Connective Systems and Classifiers

Perfil ML:FA @ MiEI/4º ano - 2º Semestre

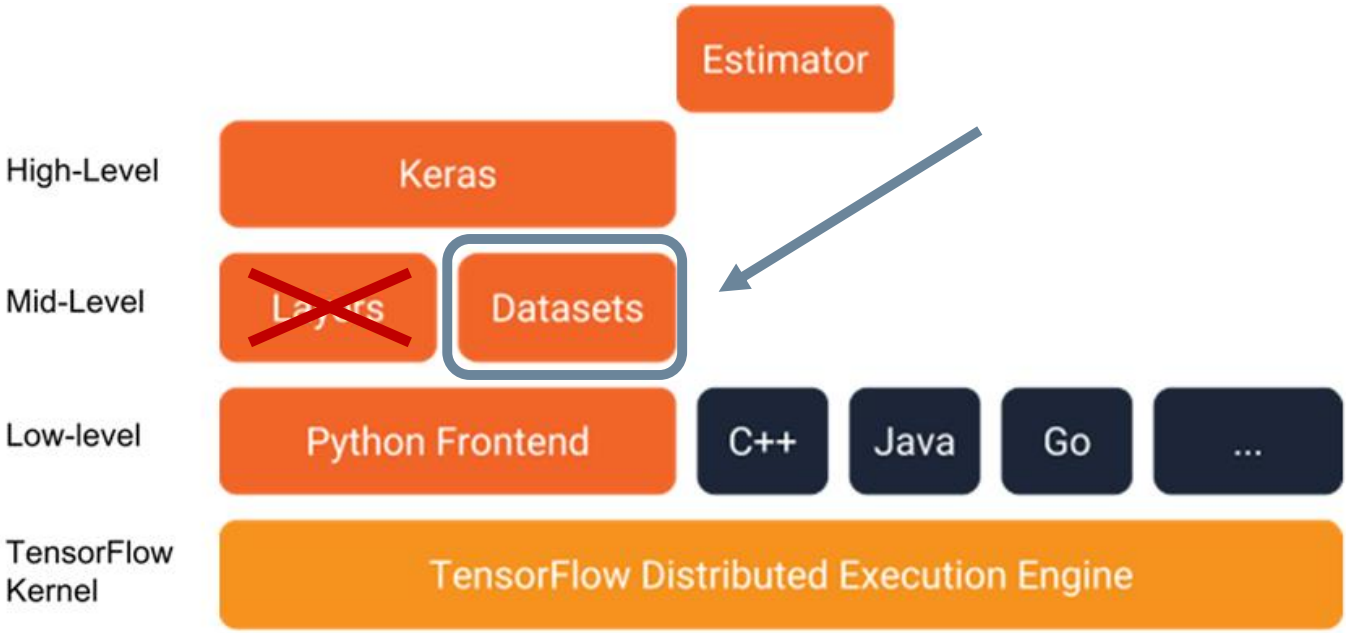
Bruno Fernandes, Victor Alves

05/03/2020

Contents

- The tf.data API (Mid-level API)
  - tf.data.Dataset
  - Loading and transforming datasets
- The tf.keras API (High-level API)
  - Keras Callbacks e Custom Callbacks
  - Checkpoints, SavedModel and saving Subclassed Models
- TensorBoard
- OpenAI Gym
- Hands On

The tf.data API



The tf.data API

TensorFlow introduced the tf.data API, a better way to handle data!

- The tf.data API makes it possible to handle large amounts of data, read from different data formats, and perform complex transformations
- The tf.data API introduces a tf.data.Dataset abstraction that represents a sequence of elements
- There are two distinct ways to create a dataset:
  - A data source builds a Dataset from data stored in memory (tf.data.Dataset.from_tensors or tf.data.Dataset.from_tensor_slices) or in files (tf.data.TFRecordDataset)
  - A data transformation to build a dataset from one or more tf.data.Dataset objects

The tf.data.Dataset API

TensorFlow introduced the tf.data API, a better way to handle data!

- The Dataset object is a Python iterable. It is possible to consume its elements using a for loop
- A dataset contains elements that each have the same (nested) structure
- The Dataset.element_spec property allows you to inspect the type of each element

```
dataset = tf.data.Dataset.from_tensor_slices([1, 2, 4, 8])

print(dataset)
print(dataset.element_spec)

for elem in dataset:
    print(elem)
```

```
<TensorSliceDataset shapes: (), types: tf.int32>

TensorSpec(shape=(), dtype=tf.int32, name=None)

tf.Tensor(1, shape=(), dtype=int32)
tf.Tensor(2, shape=(), dtype=int32)
tf.Tensor(4, shape=(), dtype=int32)
tf.Tensor(8, shape=(), dtype=int32)
```

The tf.data.Dataset API

TensorFlow introduced the tf.data API, a better way to handle data!

- The Dataset object is a Python iterable. It is possible to consume its elements using a for loop
- A dataset contains elements that each have the same (nested) structure
- The Dataset.element_spec property allows you to inspect the type of each element

```
dataset = tf.data.Dataset.from_tensor_slices([1, 2, 4, 8])

print(dataset)
print(dataset.element_spec)

for elem in dataset:
    print(elem.numpy())
```

```
<TensorSliceDataset shapes: (), types: tf.int32>

TensorSpec(shape=(), dtype=tf.int32, name=None)

1
2
4
8
```

The tf.data.Dataset API
Load from Numpy

Features and labels are supposed to be tensors, but since TensorFlow and NumPy are seamlessly integrated they can be NumPy arrays!

Assuming you have a vector of examples and the corresponding labels, pass the two vectors as a tuple into tf.data.Dataset.from_tensor_slices to create a tf.data.Dataset

```
path = tf.keras.utils.get_file(FILE_NAME, DATA_URL)

with np.load(path) as data:
  train_examples = data['x_train']
  train_labels = data['y_train']

#creating train dataset from two NumPy arrays
train_dataset = tf.data.Dataset.from_tensor_slices((train_examples, train_labels))
```

The tf.data.Dataset API
Load from pandas.DataFrame

Or pass the values of two dataframes (examples and labels) into tf.data.Dataset.from_tensor_slices to create a tf.data.Dataset

```
path = tf.keras.utils.get_file(FILE_NAME, DATA_URL)

df = pd.read_csv(path)
y = df.pop('class')

#creating train dataset from two dataframes
train_dataset = tf.data.Dataset.from_tensor_slices((df.values, y.values))
```

The tf.data.Dataset API
Load from CSV file

You can load csv content using pandas.DataFrame. But if you need to scale up to a large set of files then use the tf.data.experimental.make_csv_dataset function.

The only column you need to identify explicitly is the class.

```
#df = pd.read_csv('data.csv')
path = tf.keras.utils.get_file(FILE_NAME, DATA_URL)
#creating raw train dataset from csv file
def get_dataset(file_path, **kwargs):
    dataset = tf.data.experimental.make_csv_dataset(path, batch_size=32, label_name='survived',
        na_value='na', num_epochs=1, ignore_errors=True, **kwargs)
    return dataset

raw_train_dataset = get_dataset(path)

#if the csv file has no column names, pass them as a list of strings to the column_names argument
#instead, use the select_columns argument if you only want a subset of features
CSV_COLUMNS = ['sex', 'age', 'parch', 'fare', 'deck', 'alone', 'survived']
raw_train_dataset = get_dataset(path, column_names=CSV_COLUMNS)
```

The tf.data.Dataset API

Load text file

Use tf.data.TextLineDataset to load examples from text files, in which each example is a line of text from the original file.

Particularly interesting for NLP.

```
path = tf.keras.utils.get_file(FILE_NAME, DATA_URL)

#creating train dataset from a txt file
lines_dataset = tf.data.TextLineDataset(path)
```

The tf.data.Dataset API

Shuffling & Batching

Shuffling

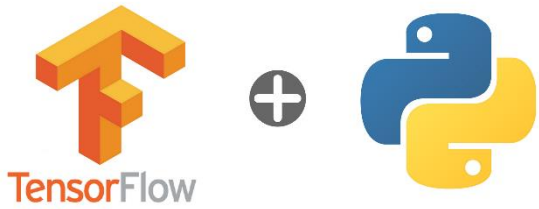
- The Dataset.shuffle() transformation maintains a fixed-size buffer and chooses the next element uniformly at random from that buffer
- Be aware that large buffer_sizes may take a lot of time and memory
- Order matters!

Batching

- The Dataset.batch() transformation stacks n consecutive elements of a dataset into a single element
- Dataset.batch results in an unknown batch size because the last batch may not be full!
- Use the drop_remainder argument to ignore that last batch, and get full shape

The tf.data.Dataset API

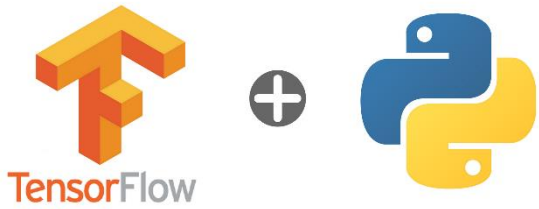
Shuffling & Batching




```
'''
Importing data
'''
def import_data():
    #load mnist training and test data
    (x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

    #some data exploration
    print('***** Log import_data *****')
    print('Train data shape', x_train.shape)
    print('Test data shape', x_test.shape)
    print('Number of training samples', x_train.shape[0])
    print('Number of testing samples', x_test.shape[0])
    for i in range(25):
        plt.subplot(5,5,i+1)    #Add a subplot as 5 x 5
        plt.xticks([])          #get rid of labels
        plt.yticks([])          #get rid of labels
        plt.imshow(x_test[i], cmap="gray")
    plt.show()
    print('***** Log import_data *****')

    #reshape the input to have a list of self.batch_size by 28*28 = 784; and normalize (/255)
    x_train = x_train.reshape(x_train.shape[0], x_train.shape[1]*x_train.shape[2]).astype('float32')/255
    x_test = x_test.reshape(x_test.shape[0], x_test.shape[1]*x_test.shape[2]).astype('float32')/255
    #reserve 5000 samples for validation
    x_validation = x_train[-5000:]
    y_validation = y_train[-5000:]
    #do not use those same 5000 samples for training!
    x_train = x_train[:-5000]
    y_train = y_train[:-5000]

    #create dataset iterator for training
    train_dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train))
    #shuffle in intervals of 1024 and slice in batchs of batch_size
    train_dataset = train_dataset.shuffle(buffer_size=1024).batch(batch_size)
    #create the validation dataset
    validation_dataset = tf.data.Dataset.from_tensor_slices((x_validation, y_validation))
    validation_dataset = validation_dataset.batch(batch_size)
    return train_dataset, validation_dataset, x_test, y_test
```

The tf.data.Dataset API

Shuffling & Batching

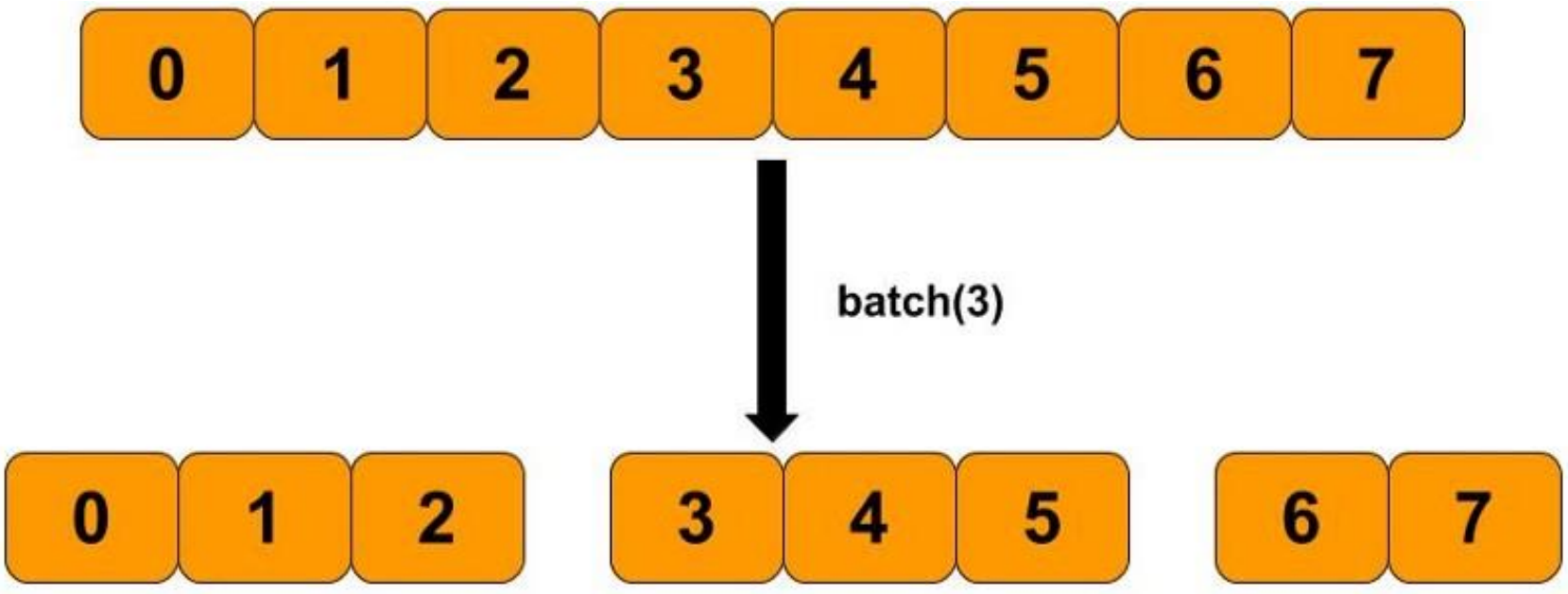



```
'''
Define a low level fit and predict making use of the tape.gradient
'''
def low_level_fit_and_predict():
    #manually, let's iterate over the epochs and fit ourselves
    for epoch in range(epochs):
        print('Epoch %d/%d' %(epoch+1, epochs))

        #to store loss values
        loss_history = []

        #iterate over all batchs
        for step, (x_batch, y_batch) in enumerate(train_dataset):
            #use a gradien tape to save computations to calculate gradient later
            with tf.GradientTape() as tape:
                #running the forward pass of all layers
                #operations being recorded into the tape
                probs = mlp(x_batch)
                #computing the loss for this batch
                #how far are we from the correct labels?
                loss_value = loss_object(y_batch, probs)

            #store loss value
            loss_history.append(loss_value.numpy().mean())
            #use the tape to automatically retrieve the gradients of the trainable variables
            #with respect to the loss
            gradients = tape.gradient(loss_value, mlp.trainable_weights)
            #running one step of gradient descent by updating (going backwards now)
            #the value of the trainable variables to minimize the loss
            optimizer.apply_gradients(zip(gradients, mlp.trainable_weights))
            # Update training metric.
            train_metric(y_batch, probs)

            #log every n batches
            if step%200 == 0:
                print('Step %s. Loss Value = %s; Mean loss = %s' %(step, str(loss_value.numpy()), np.mean(loss_history)))

        #display metrics at the end of each epoch
        train_accuracy = train_metric.result()
        print('Training accuracy for epoch %d: %s' %(epoch+1, float(train_accuracy)))
        #reset training metrics (at the end of each epoch)
        train_metric.reset_states()
```

tf.data transformations

Once you have created the Dataset, you can apply several types of transformations (order is important!):

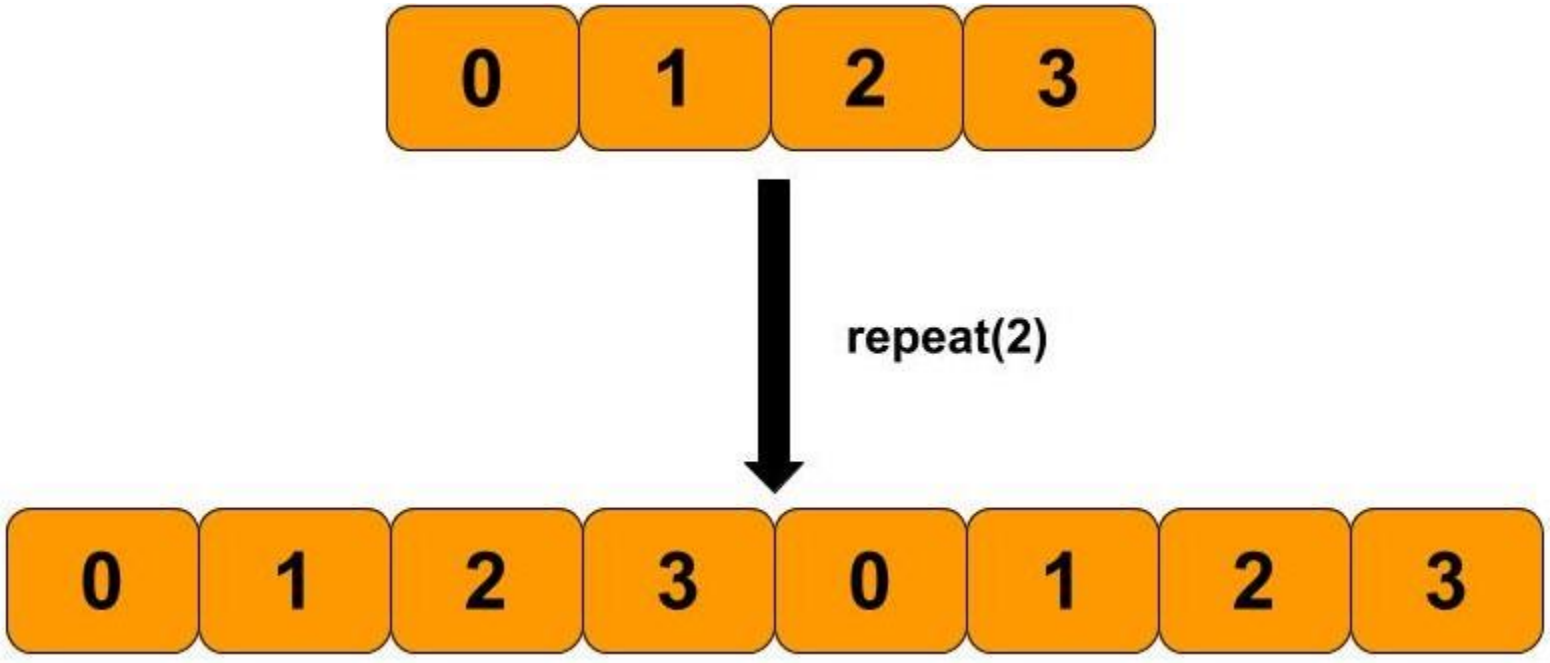
- Batch - sequentially dividing your dataset by the specified batch size



tf.data transformations

Once you have created the Dataset, you can apply several types of transformations (order is important!):

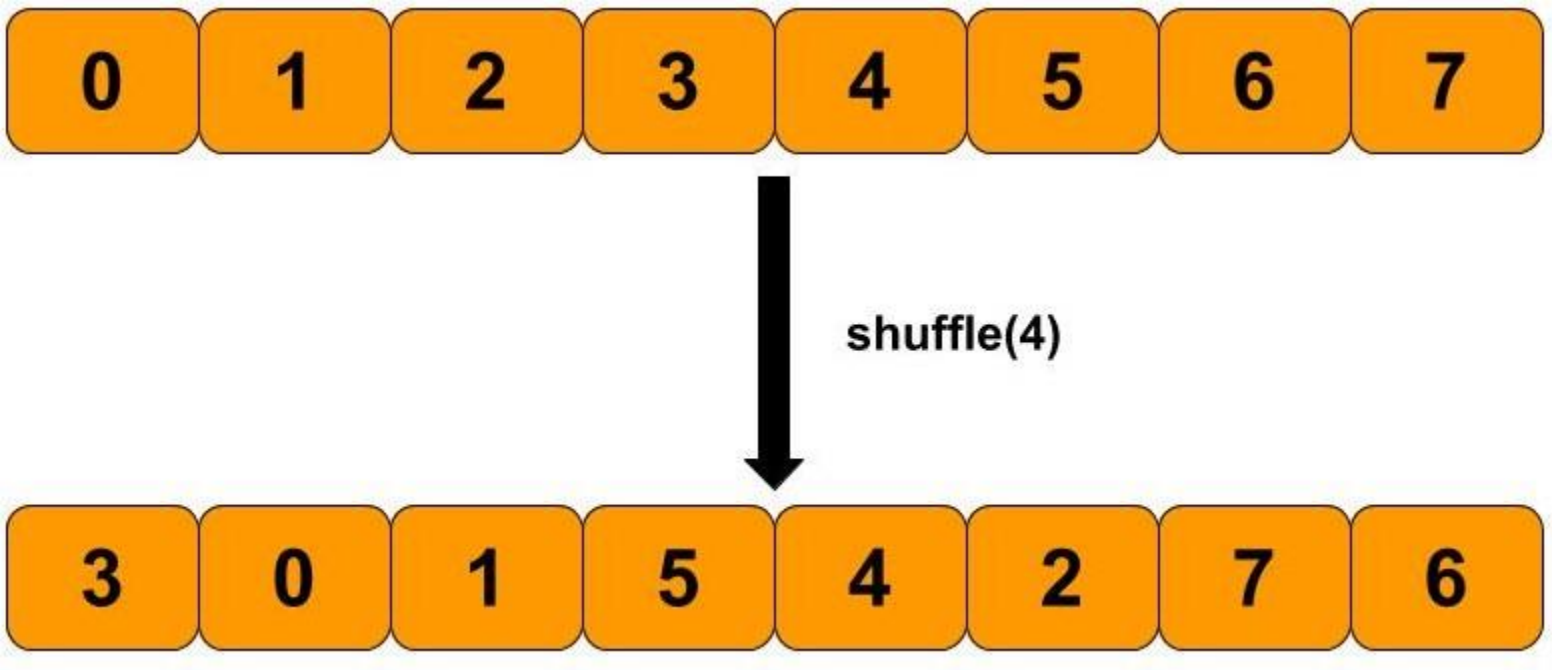
- Repeat - use this transformation to create duplicates of the existing data



tf.data transformations

Once you have created the Dataset, you can apply several types of transformations (order is important!):

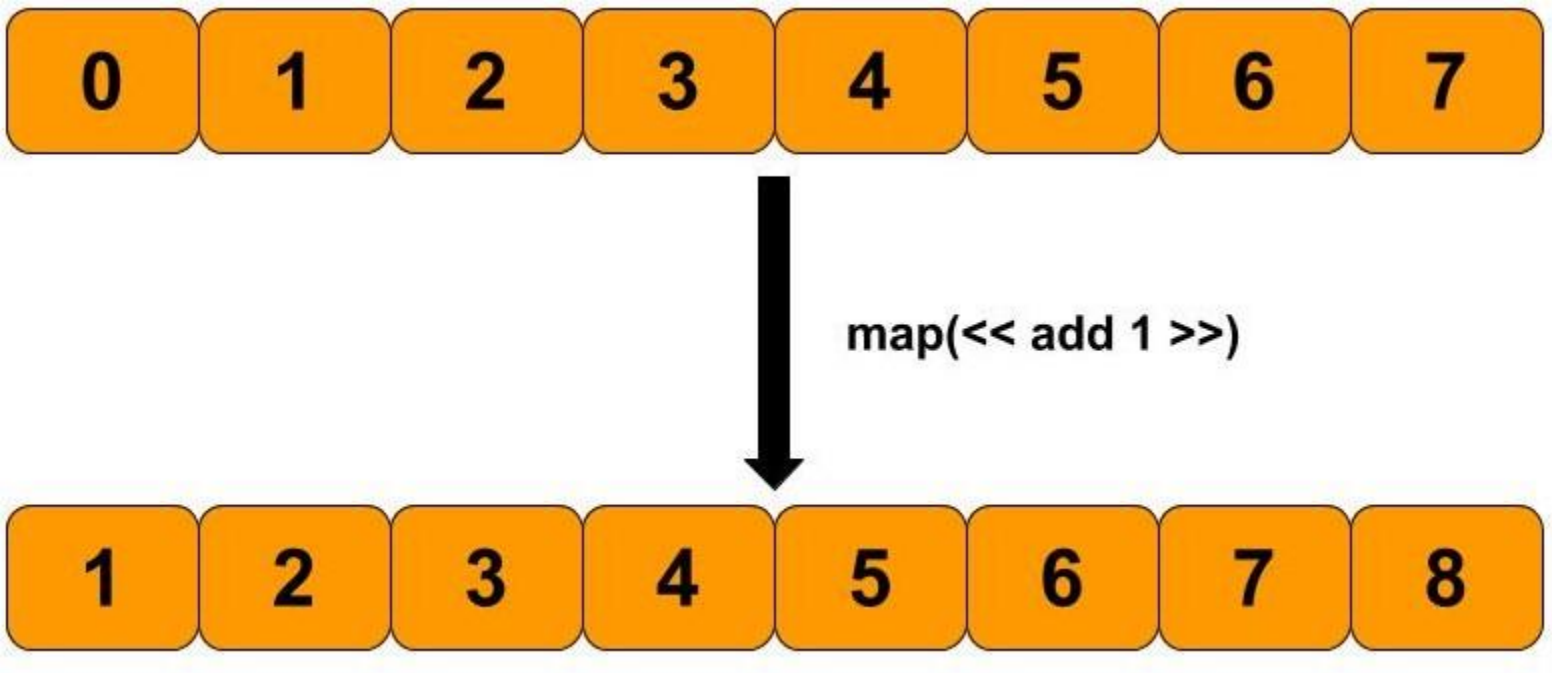
- Shuffle - randomly shuffle the data in the Dataset



tf.data transformations

Once you have created the Dataset, you can apply several types of transformations (order is important!):

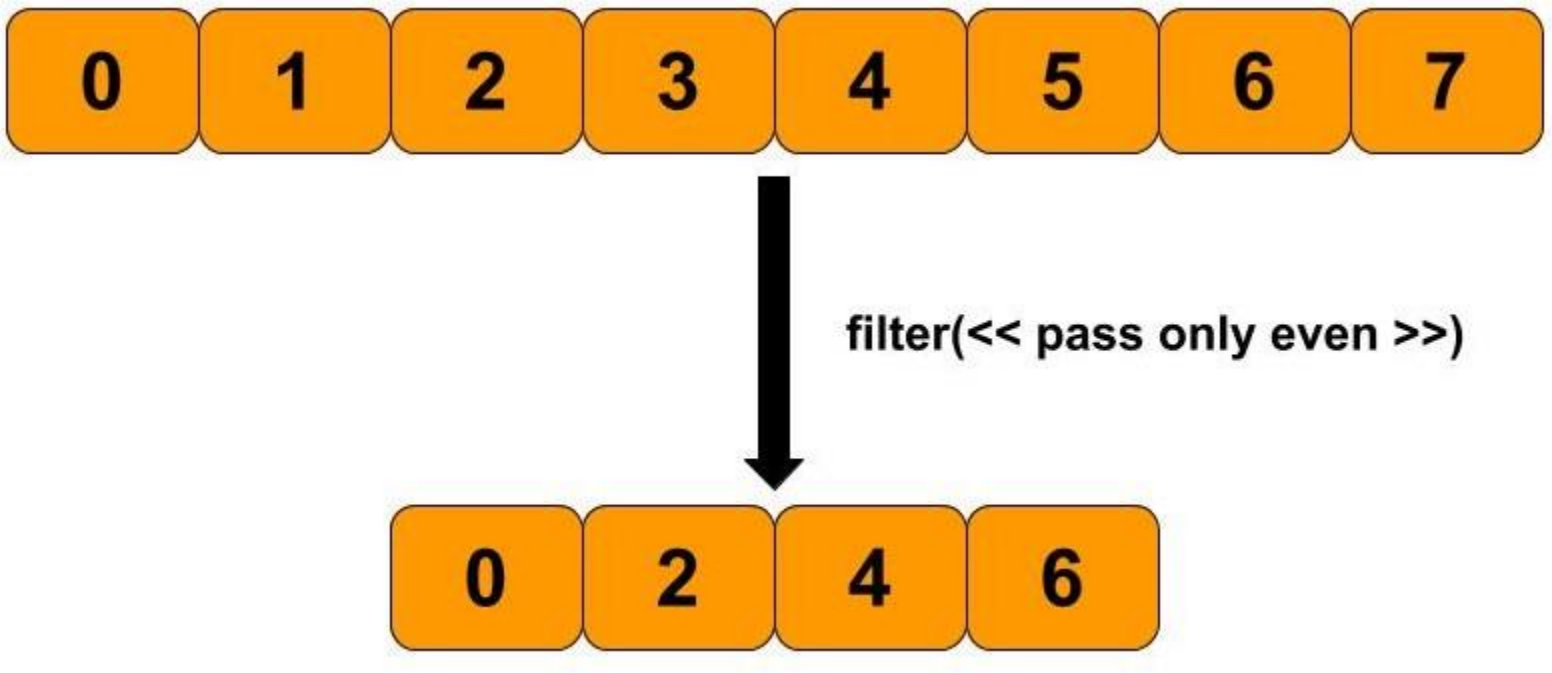
- Map - apply some operation to all the individual data elements in the dataset



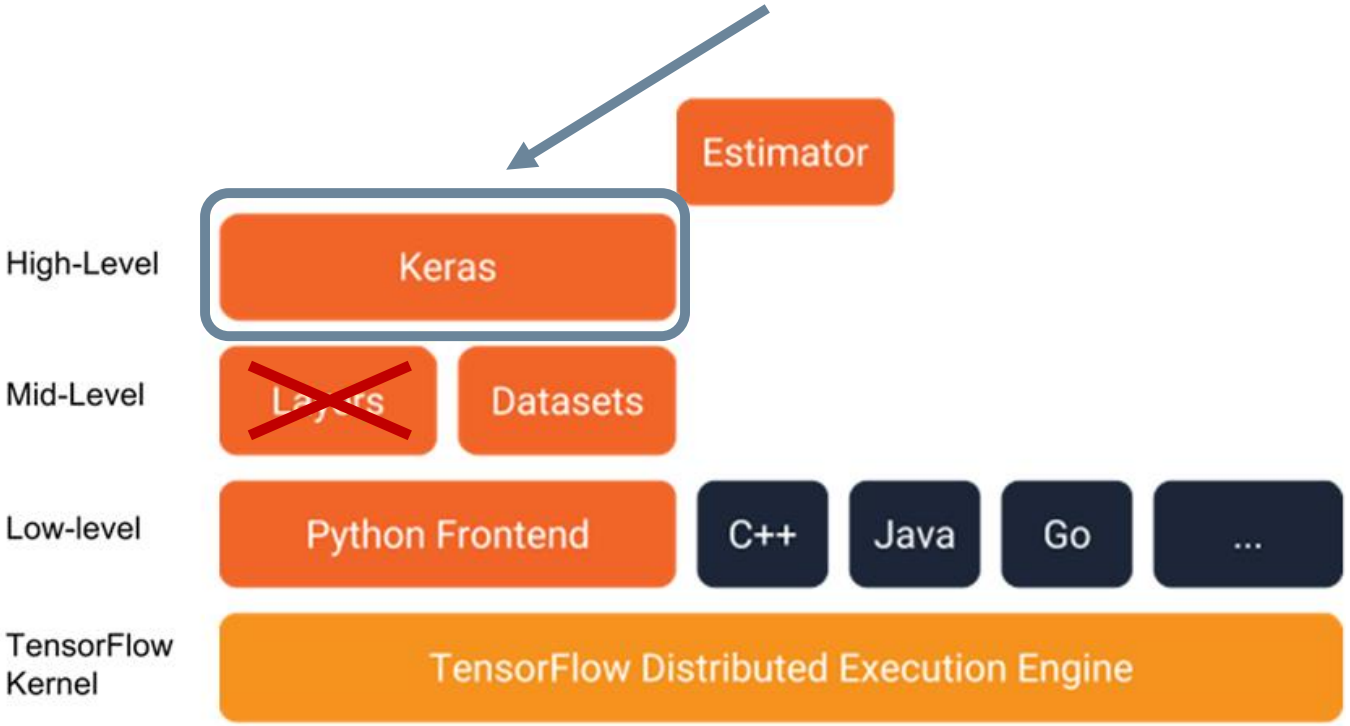
tf.data transformations

Once you have created the Dataset, you can apply several types of transformations (order is important!):

- Filter - filter out some elements from the Dataset



High-level TensorFlow APIs



Keras: The Python Deep Learning library

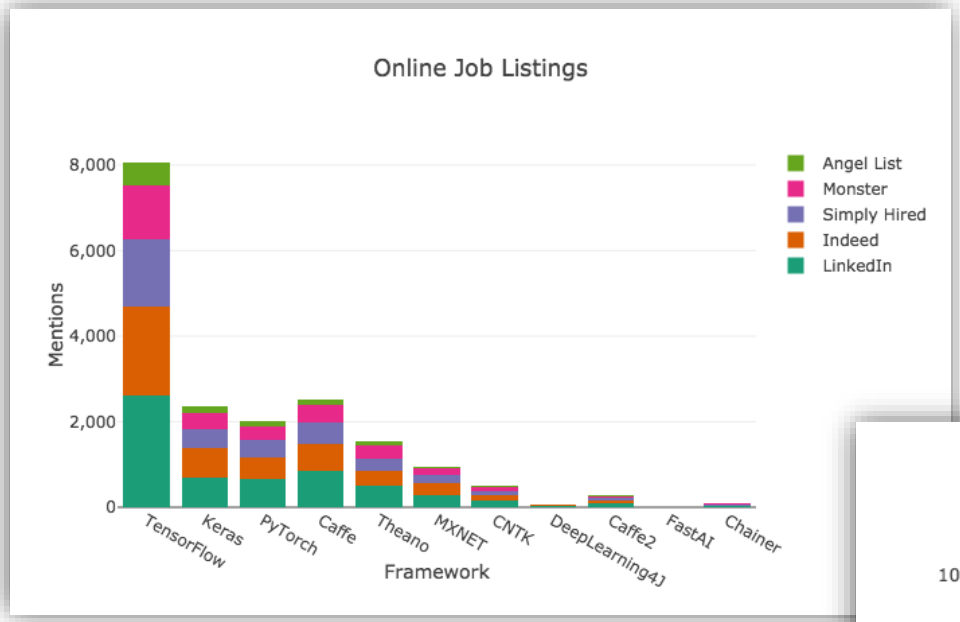
Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow. Indeed, Keras is now (since v2.0) the official high-level API of TensorFlow!

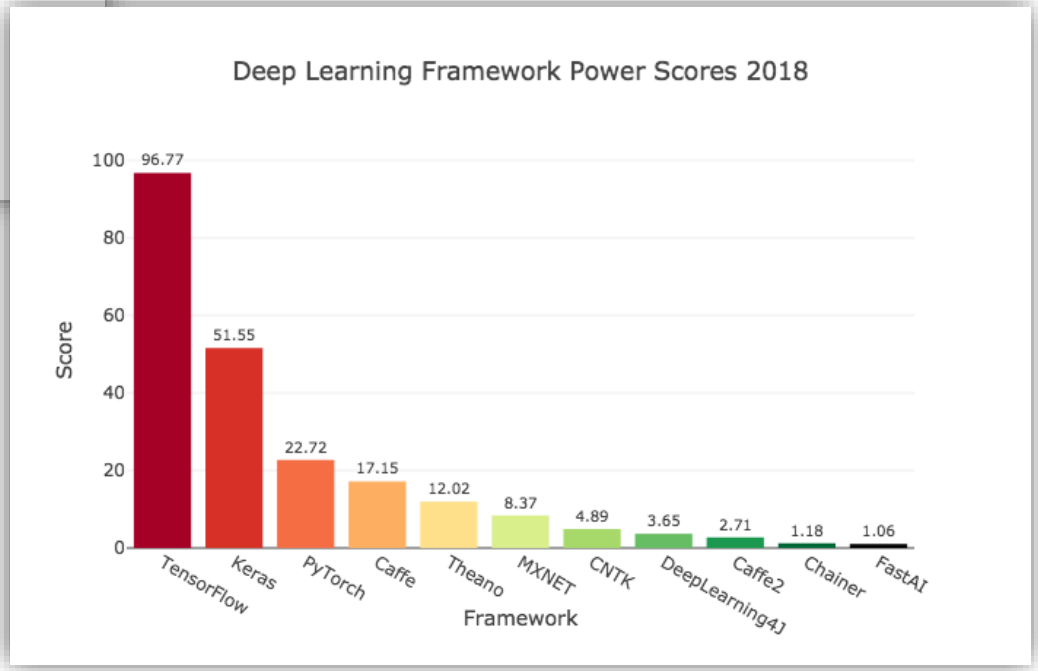
Use Keras if you need a deep learning library that:

- Allows for easy and fast prototyping (through user friendliness, modularity, and extensibility)
- Supports both convolutional networks and recurrent networks, as well as combinations of the two
- Runs seamlessly on CPU and GPU

Officially, Keras is compatible with Python 2.7 to 3.6!

Keras





What's special about Keras?

- Focused on user experience
- Large adoption in the industry and research community
- Multi-backend, multi-platform
- Easy productization of models

Keras

NETFLIX UBER Google

instacart HUAWEI NVIDIA

Square Expedia Zocdoc yelp

Get Started

tf.keras is TensorFlow's implementation of the Keras API specification. It is a high-level API to build and train models that includes support for TensorFlow-specific functionality!

tf.keras makes TensorFlow easier to use without sacrificing flexibility and performance!

To get started, import tf.keras as part of your TensorFlow program as such:

```
import tensorflow as tf

print(tf.__version__)

print(tf.keras.__version__)
```

```
2.0.0

2.2.4-tf
```

Three API styles

The Sequential Model

- As simple as it gets
- Used for single-input, single-output and sequential layer stacks
- Good for >70% of use cases

The functional API

- Like playing with Lego bricks
- Multi-input, multi-output, arbitrary static graph topologies
- Good for 95% of use cases

Model subclassing

- Maximum flexibility
- Larger potential for errors

Sequential model

The core data structure of Keras is a model, a way to organize layers. The simplest and most common type of model is the Sequential model, a linear stack of layers.

To build a simple, fully-connected network (i.e. multi-layer perceptron):

```
import tensorflow as tf
from tensorflow.keras import layers

#sequential model
model = tf.keras.Sequential()
#flatten layer to transform input from 2d (28x28) to 1d (784)
model.add(layers.Flatten(input_shape=(28, 28)))
# add a densely-connected layer with 64 neurons
model.add(layers.Dense(64, activation='relu'))
#add a softmax layer with 10 output neurons
model.add(layers.Dense(10, activation='softmax'))
```

Sequential model

The core data structure of Keras is a model, a way to organize layers. The simplest and most common type of model is the Sequential model, a linear stack of layers.

To build a simple, fully-connected network (i.e. multi-layer perceptron):

```
import tensorflow as tf
from tensorflow.keras import layers

#sequential model
model = tf.keras.Sequential([
    layers.Flatten(input_shape=(28, 28)),
    layers.Dense(64 , activation='relu'),
    layers.Dense(10, activation='softmax')
])
```

Sequential model

There are many tf.keras.layers available with some common constructor parameters:

- activation - set the activation function for the layer (defaults to no activation being applied)
- kernel_initializer and bias_initializer - the initialization schemes that create the layer's weights (defaults to Glorot/Xavier uniform initializer)
- kernel_regularizer and bias_regularizer - apply penalties on layer parameters or layer activity during optimization (penalties are incorporated in the loss function)

```
#create a sigmoid activated layer
layers.Dense(64, activation='sigmoid')
#linear layer with L1 regularization (Lasso Regression) of factor 0.01 applied to weights
layers.Dense(64, kernel_regularizer=tf.keras.regularizers.l1(0.01))
#linear layer with L2 regularization (Ridge Regression) of factor 0.01 applied to the bias
layers.Dense(64, bias_regularizer=tf.keras.regularizers.l2(0.01))
#linear layer with a bias vector initialized to 2.0s
layers.Dense(64, bias_initializer=tf.keras.initializers.constant(2.0))
```

Compile it

After building the model, configure the model for training by calling the compile method

```
#sequential model
model = tf.keras.Sequential()
#flatten layer to transform input from 2d (28x28) to 1d (784)
model.add(layers.Flatten(input_shape=(28, 28)))
# add a densely-connected layer with 64 neurons
model.add(layers.Dense(64, activation='relu'))
#add a softmax layer with 10 output neurons
model.add(layers.Dense(10, activation='softmax'))

#configure the model
model.compile(optimizer=tf.train.AdamOptimizer(0.001),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Compile it

tf.keras.Model.compile takes three important arguments:

- optimizer - specifies the training procedure. Pass optimizer instances from tf.keras.optimizers module
- loss - function to minimize during optimization. Loss functions are specified by name or by passing a callable object from the tf.keras.losses module
- metrics - used to monitor training. May be strings or callable objects from the tf.keras.metrics module

```
#configure the model
model.compile(optimizer=tf.train.AdamOptimizer(0.001),
              loss='mse',
              metrics=['mae'])
#or
model.compile(optimizer=tf.train.GradientDescentOptimizer(0.01),
              loss=tf.keras.losses.categorical_crossentropy,
              metrics=[tf.keras.metrics.categorical_accuracy])
```

Fit the model

tf.keras.Model.fit also takes three important arguments:

- epochs - training is structured into epochs. An epoch is one iteration over the entire input data (normally done in smaller batches)
- batch_size - specifies the size of each batch
- validation_data - data on which to evaluate the loss and any model metrics at the end of each epoch. The model will not be trained on this data

```
#training
model.fit(data, labels, epochs=10, batch_size=32)

#or (the tf.data.dataset was already shuffled and batched)
model.fit(train_dataset, validation_data=validation_dataset, epochs=epochs)
```

NumPy inputs

For small datasets, use in-memory NumPy arrays to train and evaluate the model

```
import numpy as np

#numpy data
data = np.random.random((1000, 32))
labels = np.random.random((1000, 10))

#validation data
val_data = np.random.random((100, 32))
val_labels = np.random.random((100, 10))

# … building the model

#train the model
model.fit(data, labels, epochs=10, batch_size=32,
          validation_data=(val_data, val_labels))
```

tf.data inputs

To scale to large datasets use the tf.data API

```
#instantiate a dataset instance
dataset = tf.data.Dataset.from_tensor_slices((data, labels))
dataset = dataset.batch(32)
dataset = dataset.repeat(2)
…
#specify steps_per_epoch when calling fit on a dataset
model.fit(dataset, epochs=10, steps_per_epoch=30)
```

Note:

- steps_per_epoch refers to the total number of steps before declaring one epoch finished and starting the next epoch. When training with TensorFlow data tensors, the default is equal to the number of samples in the dataset divided by the batch size, or 1 if that cannot be determined

Evaluate & Predict

To evaluate loss and metrics for the provided data:

```
import numpy as np
…
#numpy data
test_data = np.random.random((1000, 32))
test_labels = np.random.random((1000, 10))
…
test_loss, test_acc = model.evaluate(test_data, test_labels, batch_size=32)
#or
test_loss, test_acc = model.evaluate(dataset, steps=30)
```

And to predict the output of the last layer in inference for the provided data:

```
predictions = model.predict(pred_data, batch_size=32)
```

Functional API

A more flexible way of creating models than the Sequential API.

Use the Functional API to build complex model topologies such as:

- Multi-input models
- Multi-output models
- Models with shared layers (the same layer called several times)

Building a model with the functional API works like this:

1. A layer instance is callable and returns a tensor
2. Input tensors and output tensors are used to define a tf.keras.Model instance
3. This model is trained just like the Sequential model.

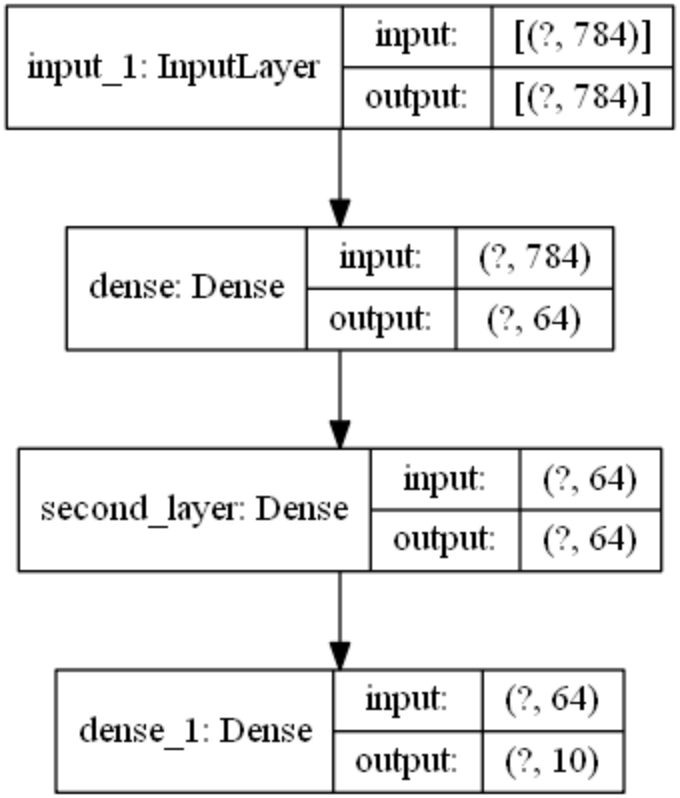
Functional API


```
#returns a placeholder input
inputs = tf.keras.Input(shape=(784,))

#a layer instance is callable on a tensor and returns a tensor.
x = tf.keras.layers.Dense(64, activation='relu')(inputs)
#or
#dense = layers.Dense(64, activation='relu')
#x = dense(inputs)

x = tf.keras.layers.Dense(64, activation='relu', name='second_layer')(x)
probs = tf.keras.layers.Dense(10, activation='softmax')(x)

#instantiate the model given the inputs and outputs
model = tf.keras.Model(inputs=inputs, outputs=probs, name='mnist')

print(model.summary())
#you must install pydot and graphviz to plot your model
tf.keras.utils.plot_model(model, 'my_mnist_model.png', show_shapes=True)

#call model.compile(), model.fit() and model.evaluate() as you do with the Sequential API
```

Functional API


```
Model: "mnist"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 784)]             0
_________________________________________________________________
dense (Dense)                (None, 64)                50240
_________________________________________________________________
second_layer (Dense)         (None, 64)                4160
_________________________________________________________________
dense_1 (Dense)              (None, 10)                650
=================================================================
Total params: 55,050
Trainable params: 55,050
Non-trainable params: 0
_________________________________________________________________
None
```

Functional API



Model Subclassing

Allows us to build a fully-customizable model by subclassing tf.keras.Model and defining our own forward pass!

We can also create our custom layer by subclassing tf.keras.layers.Layer!

Where have we seen this??

Model Subclassing


```
# ================================================================
#
#         PerceptronLayer Class
#
# ================================================================
class PerceptronLayer(Layer):

    '''
    The constructor in an object oriented perspective.
    Called when an object is created, allowing the class to initialize the attributes of a class.
    neurons corresponds to the number of neurons in this perceptron layer
    '''
    def __init__(self, neurons=16, **kwargs):
        super(PerceptronLayer, self).__init__(**kwargs)
        self.neurons = neurons

    '''
    We use the build function to deferr weight creation until the shape of the inputs is known
    '''
    def build(self, input_shape):
        pass

    '''
    Implements the function call operator (when an instance is used as a function).
    It will automatically run build the first time it is called, i.e., layer's weights are created dynamically
    '''
    def call(self, inputs):
        pass

    '''
    Enable serialization on our perceptron layer
    '''
    def get_config(self):
        pass
```

Model Subclassing


```
# ==============================================================================
#
#         Multilayer Perceptron Class
#
# ==============================================================================
class MultilayerPerceptron(Model):

    '''
    The Layers of our MLP (with a fixed number of neurons)
    '''
    def __init__(self, output_neurons=10, name='multilayerPerceptron', **kwargs):
        super(MultilayerPerceptron, self).__init__(name=name, **kwargs)
        self.perceptron_layer_1 = PerceptronLayer(16)
        self.perceptron_layer_2 = PerceptronLayer(32)
        self.perceptron_layer_3 = PerceptronLayer(output_neurons)

    '''
    Layers are recursively composable, i.e.,
    if you assign a Layer instance as attribute of another Layer, the outer layer will start tracking the weights of the inner layer.
    Remember that the build of each layer is called automatically (thus creating the weights).
    '''
    def feed_model(self, input_data):
        pass

    """
    Compute softmax values for the logits
    """
    def softmax(self, logits):
        pass

    def print_trainable_weights(self):
        pass

    def call(self, input_data):
        pass
```

Model Subclassing


```
'''
Layers are recursively composable, i.e.,
if you assign a Layer instance as attribute of another Layer, the outer layer will start tracking the weights of the inner layer.
Remember that the build of each layer is called automatically (thus creating the weights).
'''
def feed_model(self, input_data):
    x = self.perceptron_layer_1(input_data)
    #activation function applied to the output of the perceptron layer
    x = tf.nn.relu(x)
    #the output, now normalized, is fed as input to the second perceptron layer
    x = self.perceptron_layer_2(x)
    #again, activation function applied to the output of the second perceptron layer
    x = tf.nn.relu(x)
    #which, again, is fed as input to the third layer, which returns its output
    logits = self.perceptron_layer_3(x)
    #the output of the last layer going over a softmax activation
    #so, we will not be outputting logits but "probabilities"
    return self.softmax(logits) #equivalent of tf.nn.softmax(logits)
```

Using the model
High Level


```
'''
Define a high level fit and predict making use tf.Keras APIs
'''
def high_level_fit_and_predict():
    #shortcut to compile and fit a model!
    #able to do this because our model subclasses tf.keras.Model
    mlp.compile(optimizer, loss=loss_object, metrics=[train_metric])
    #since the train_dataset already takes care of batching, we don't pass a batch_size argument
    #passing validation data for monitoring validation loss and metrics at the end of each epoch
    history = mlp.fit(train_dataset, validation_data=validation_dataset, epochs=epochs)
    #print('\nHistory values per epoch:', history.history)

    #evaluating the model on the test data
    print('\nEvaluating model on test data...')
    scores = mlp.evaluate(x_test, y_test, batch_size=batch_size, verbose=0)
    print('Evaluation %s: %s' %(mlp.metrics_names, str(scores)))

    #finally, generating predictions (the output of the last layer)
    print('\nGenerating predictions for ten samples...')
    predictions = mlp.predict(x_test[:10])
    #now, for each prediction in predictions, get the value with higher "probability"
    #look at the shape, it is as (3, 10). For each prediction, we have the prob of beeing 0, beeing 1, etc...
    #we now choose the index of the list with higher "probability"
    #if pos=3 is the one with higher probability it means it predicts a 3
    print('Predictions shape:', predictions.shape)
    for i, prediction in enumerate(predictions):
        #tf.argmax returns the INDEX with the largest value across axes of a tensor
        predicted_value = tf.argmax(prediction)
        label = y_test[i]
        print('Predicted a %d. Real value is %d.' %(predicted_value, label))
```

Using the model

Low Level


```
'''
Define a low level fit and predict making use of the tape.gradient
'''
def low_level_fit_and_predict():
    #manually, let's iterate over the epochs and fit ourselves
    for epoch in range(epochs):
        print('Epoch %d/%d' %(epoch+1, epochs))

        #to store loss values
        loss_history = []

        #iterate over all batchs
        for step, (x_batch, y_batch) in enumerate(train_dataset):
            #use a gradien tape to save computations to calculate gradient later
            with tf.GradientTape() as tape:
                #running the forward pass of all layers
                #operations being recorded into the tape
                probs = mlp(x_batch)
                #computing the loss for this batch
                #how far are we from the correct labels?
                loss_value = loss_object(y_batch, probs)

            #store loss value
            loss_history.append(loss_value.numpy().mean())
            #use the tape to automatically retrieve the gradients of the trainable variables
            #with respect to the loss
            gradients = tape.gradient(loss_value, mlp.trainable_weights)
            #running one step of gradient descent by updating (going backwards now)
            #the value of the trainable variables to minimize the loss
            optimizer.apply_gradients(zip(gradients, mlp.trainable_weights))
            # Update training metric.
            train_metric(y_batch, probs)

            #log every n batches
            if step%200 == 0:
                print('Step %s. Loss Value = %s; Mean Loss = %s' %(step, str(loss_value.numpy()), np.mean(loss_history)))

        #display metrics at the end of each epoch
        train_accuracy = train_metric.result()
        print('Training accuracy for epoch %d: %s' %(epoch+1, float(train_accuracy)))
        #reset training metrics (at the end of each epoch)
        train_metric.reset_states()
```

Callbacks

A callback is an object passed to a model to customize and extend its behavior during training:

- tf.keras.callbacks.LearningRateScheduler - dynamically change the learning rate
- tf.keras.callbacks.EarlyStopping - interrupt training when validation performance has stopped improving
- tf.keras.callbacks.ModelCheckpoint - model is automatically saved during training
- tf.keras.callbacks.TensorBoard - write TensorBoard logs

```
#defining callbacks
callbacks = [
        #interrupt training if loss stops improving for over 2 epochs
        tf.keras.callbacks.EarlyStopping(patience=2, monitor='cost'),
        #write TensorBoard logs to ./tf_logs directory
        tf.keras.callbacks.TensorBoard(log_dir='./logs')
]
model.fit(data, labels, batch_size=32, epochs=5, callbacks=callbacks,…)
```

Custom Callbacks

We can write our own custom callback too!

```
#extend tf.keras.callbacks.Callback
class OurCustomCallback(tf.keras.callbacks.Callback):

    #logs dict contains the loss value and all the metrics at the end of a batch/epoch
    def on_train_batch_begin(self, batch, logs=None):
        print('Training: batch %d begins.', %batch)

    def on_train_batch_end(self, batch, logs=None):
        print('Training: batch %d ends.', %batch)

    def on_test_batch_begin(self, batch, logs=None):
        print('Evaluating: batch %d begins.', %batch)

    def on_test_batch_end(self, batch, logs=None):
        print('Evaluating: batch %d ends.', %batch)


model.fit(data, labels, batch_size=32, epochs=5, callbacks=[OurCustomCallback()])
```

Custom Callbacks

We can write our own custom callback too!

Override:

- on_(train|test|predict)_(begin|end)(self, logs=None)
  - Called at the beginning/end of fit/evaluate/predict.
- on_(train|test|predict)_batch_(begin|end)(self, batch, logs=None)
  - Called right before/after processing a batch during training/testing/predicting.
- on_epoch_(begin|end)(self, epoch, logs=None)
  - For training only

Checkpoints

There may be times where we need to let our models run for days… And then:

- We decide to adjust something...!!
- The computer crashes!
- Power goes off!
- A memory error emerges!
- … the list goes on…

Training is interrupted and we will need to run our model all over again!!!

Checkpoints

It is crucial to have the ability to stop training at any point, for any reason, and resume training later as if nothing happened!

- Checkpoints capture the exact value of all tf.Variable objects used by a model
- Checkpoints do not contain any description of the computation defined by the model
- Typically useful when source code of the original model is available

Tip:

- A good practice is to periodically save the model's parameters so that we can later restore or retrain our model from a previous step, if required.

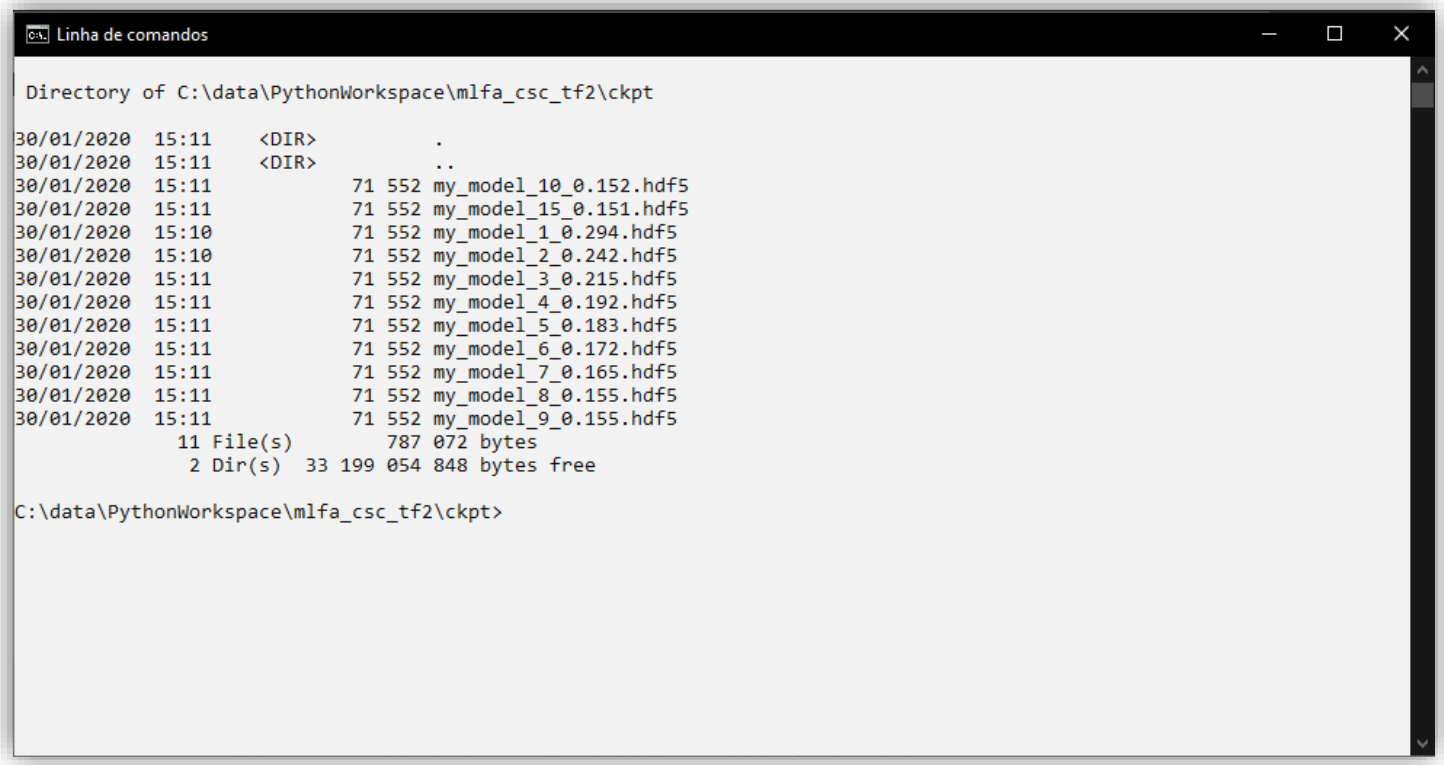
Checkpoints

We can use the tf.keras.callbacks.ModelCheckpoint callback to automatically save checkpoints during training

```
#defining callbacks
callbacks = [
    tf.keras.callbacks.ModelCheckpoint(
        #saving in Keras HDF5 (or h5), a binary data format
        filepath='ckpt/my_model_{epoch}_{val_loss:.3f}.hdf5', #path where to save the model
        save_best_only=True, #overwrite the current checkpoint if and only if
        monitor='val_loss', #the val_loss score has improved
        save_weights_only=False, #if True, only the weights are saved
        verbose=1) #verbosity mode
]

#if the filepath does not contain formatting options like {epoch} or {val_loss}
#then filepath will be overwritten by each new better model
model.fit(data, labels, batch_size=32, epochs=5, callbacks=callbacks,...)
```

Checkpoints

In the filesystem…



Hands On
Checkpoints

Implement this callback in our low-level MLP (@ function high_level_fit_and_predict)!

Note: you will need to set the name for each weight and bias! (issue https://github.com/tensorflow/tensorflow/issues/26811)

```
#defining callbacks
callbacks = [
    tf.keras.callbacks.ModelCheckpoint(
        #saving in Keras HDF5 (or h5), a binary data format
        filepath='ckpt/my_model_{epoch}_{val_loss:.3f}.hdf5', #path where to save the model
        save_best_only=True, #overwrite the current checkpoint if and only if
        monitor='val_loss', #the val_loss score has improved
        save_weights_only=False, #if True, only the weights are saved
        verbose=1) #verbosity mode
]

#if the filepath does not contain formatting options like {epoch} or {val_loss}
#then filepath will be overwritten by each new better model
model.fit(data, labels, batch_size=32, epochs=5, callbacks=callbacks,...)
```

HANDS ON

Whole-model Saving

tf.keras.Model.save() function includes:

- The model's architecture
- The model's weight values (learned during training)
- The model's training config (passed to the compile function)
- The optimizer and its state, if any (enables you to restart training)

```
#save the model
model.save('my_model.h5')

#recreate the exact same model from file
new_model = tf.keras.models.load_model('my_model.h5')
new_predictions = new_model.predict(x_test)
```

Whole-model Saving

However if we apply it to our sub-classed, low-level MLP…

NotImplementedError: Saving the model to HDF5 format requires the model to be a Functional model or a Sequential model.
It does not work for subclassed models, because such models are defined via the body of a Python method, which isn't safely serializable.
Consider saving to the Tensorflow SavedModel format (by setting save_format="tf") or using `save_weights`.

The tf.keras.Model.save() function works only for Functional or Sequential models!

SavedModel

We can also export a whole model to the TensorFlow SavedModel format:

- SavedModel is a standalone serialization format for TensorFlow objects
- It contains a complete TensorFlow program, including weights and computation
- Models in this format are independent of the source code that created the model, which makes it useful for sharing and deploying

SavedModel

The SavedModel files include:

- A TensorFlow checkpoint containing the model weights
- A SavedModel proto containing the underlying TensorFlow graph

The SavedModel also works for Functional or Sequential models!

```
#save a savedmodel
model.save('my_model', save_format='tf')

#recreate the exact same savedmodel from file
new_model = tf.keras.models.load_model('my_model')
new_predictions = new_model.predict(x_test)
```

Saving Subclassed Models

- Sequential and Functional models are data structures that represent a Directed Acyclic Graph of layers that can be safely serialized and deserialized
- On the other hand, a subclassed model is a piece of code, where the architecture of the model is defined via the body of the call method
- To load a subclassed model we need to have access to the code that created it
- Also, a subclassed model that has never been used/trained cannot be saved! Remember that a subclassed model needs to be called on some data in order to build the weights (otherwise the shape and dtype of the input data is unknown)!

Saving Subclassed Models

Hence, the recommended way to save a subclassed model is to use save_weights function, creating a SavedModel checkpoint that contains the value of all variables:

- Layer's Weights
- Optimizer's state
- Other variables associated with stateful model metrics

Some notes on loading a subclassed model:

- To restore the optimizer state and the state of any stateful metric you should compile the model with the exact same arguments as before
- Then call it on some data
- Then call load_weights

Saving Subclassed Models

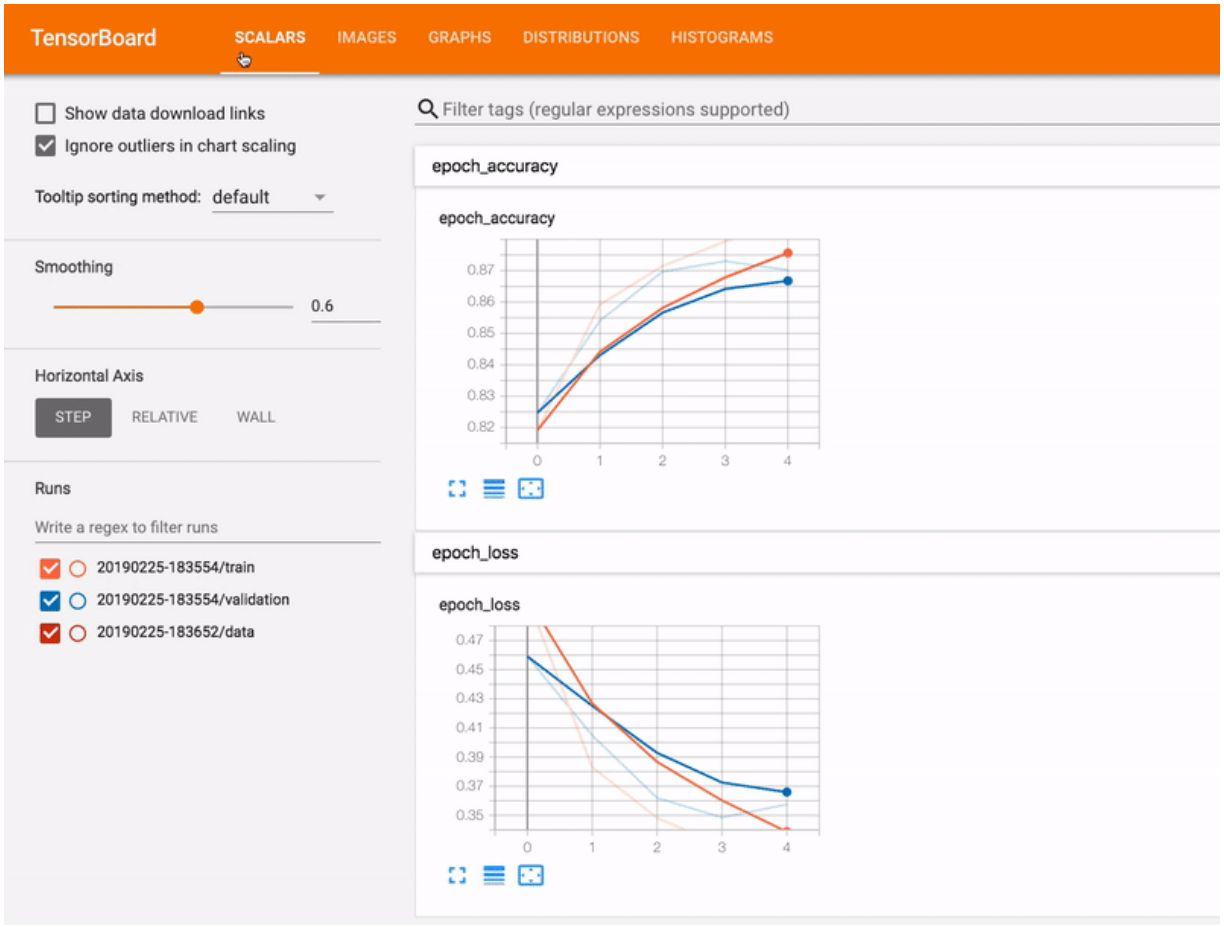

```
#save a savedmodel
model.save_weights('my_weights', save_format='tf')
---
#to restore the optimizer state and the state of any stateful metric
#you should compile the model with the exact same arguments as before
new_model = MultilayerPerceptron(output_neurons=output_neurons)
new_model.compile(optimizer, loss=loss_object, metrics=[train_metric])

#and call it on some data before calling load_weights
#initialize the variables used by the optimizer as well as any stateful metric variables
new_model.train_on_batch(x_train[:1], y_train[:1])

#load the weights of the old model
new_model.load_weights('my_weights')

new_predictions = new_model.predict(x_test)
```

Saving to json

Saving the model architecture to json as well as the weights of a trained model. However, it will not include the training configuration and the optimizer. You will need to call compile() again before using the model.

```
#serialize a model to JSON format
json_config = model.to_json()
with open('model_config.json', 'w') as json_file:
    json_file.write(json_config)
#save weights
model.save_weights('my_weights.h5') #Keras HDF5 format
#model.save_weights('my_tf_checkpoint', save_format='tf') #tf checkpoint format

#load json model from the 2 files
with open('model_config.json') as json_file:
    json_config = json_file.read()
new_model = tf.keras.models.model_from_json(json_config)
new_model.load_weights('my_weights.h5')
```

TensorBoard



What is it?

TensorFlow's visualization toolkit:

- The computations we will be using TensorFlow for can be complex and confusing
- TensorBoard is a set of visualization tools to make it easier to understand, debug and optimize TensorFlow programs
- TensorBoard can be used to visualize our TensorFlow graph, plot quantitative metrics about the execution of the graph or show additional data
- The graph visualizer is a component of TensorBoard that renders the structure of your graph visually in a browser

Why?

TensorBoard provides the visualization and tooling needed for ML experimentation:

- Tracking and visualizing metrics such as loss and accuracy
- Visualizing the model graph (ops and layers)
- Viewing histograms of weights, biases, or other tensors as they change over time
- Projecting embeddings to a lower dimensional space
- Displaying images, text, and audio data
- Profiling TensorFlow programs
- …

However, it tends to be more complex than it should…

Using TensorBoard


```
#load the TensorBoard notebook extension (run it in the IPython shell)
%load_ext tensorboard
```

```
#a simple ANN
import tensorflow as tf
import datetime
import os
#mnist dataset
mnist = tf.keras.datasets.mnist
#data normalization
(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train/255.0, x_test/255.0
#model creation
def create_model():
    return tf.keras.models.Sequential([
        tf.keras.layers.Flatten(input_shape=(28, 28)),
        tf.keras.layers.Dense(512, activation='relu'),
        tf.keras.layers.Dropout(0.2),
        tf.keras.layers.Dense(10, activation='softmax')
    ])
```

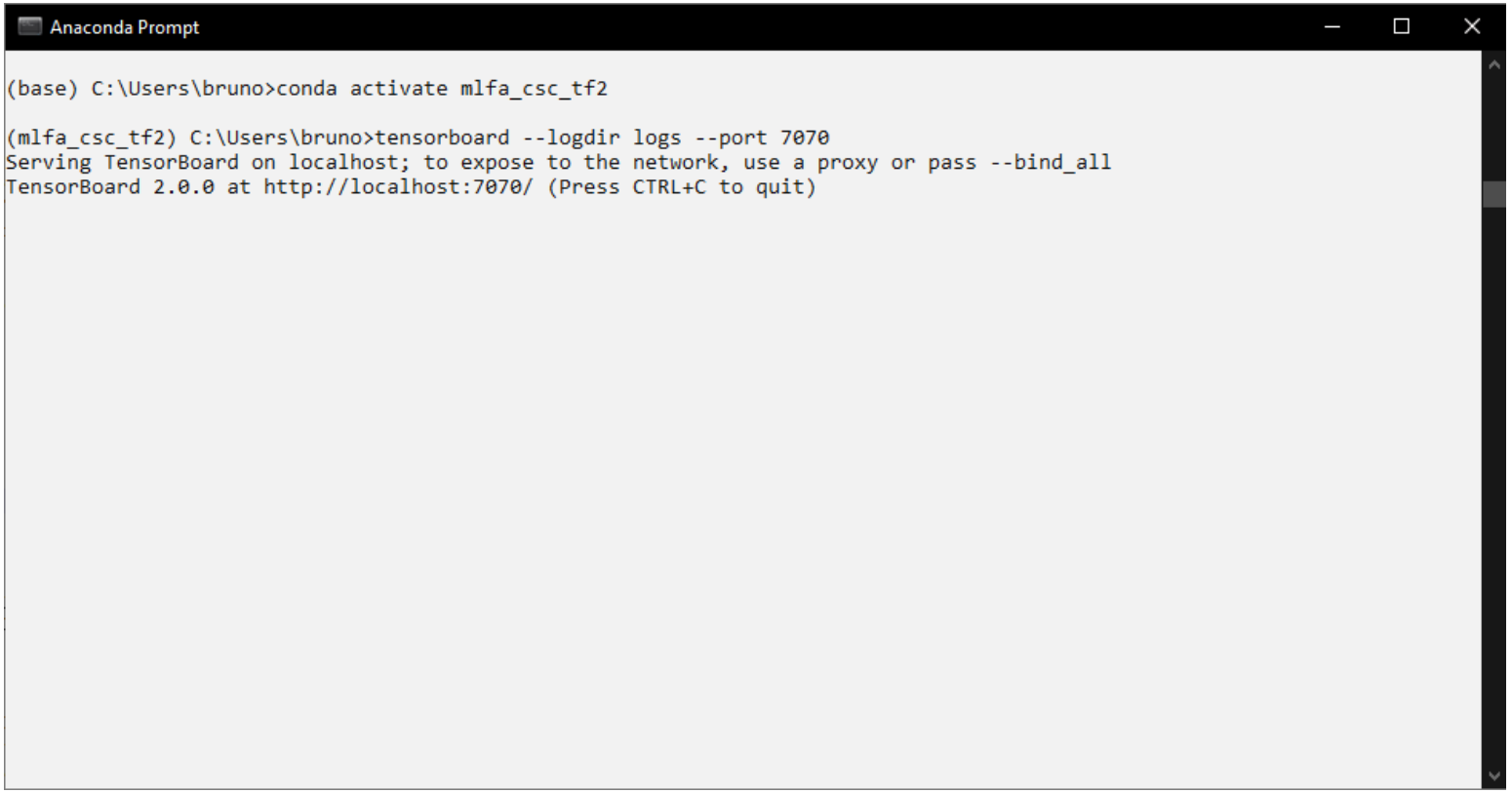
Using TensorBoard


```
model = create_model()
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

#placing the logs in a timestamped subdirectory
log_dir = os.path.join('logs', datetime.datetime.now().strftime('%Y%m%d-%H%M%S'))
#this callback is used to ensure that logs are created and stored
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1)

#fit the model using the training data, for 5 epochs, passing the previous callback
model.fit(x=x_train,
          y=y_train,
          epochs=5,
          validation_data=(x_test, y_test),
          callbacks=[tensorboard_callback])
```

```
#start TensorBoard through the notebook (works nicely with jupyter and colab)
%tensorboard --logdir logs
#in anaconda prompt/terminal run as follows (if you use spyder or pycharm)
tensorboard --logdir logs --port 7070
```

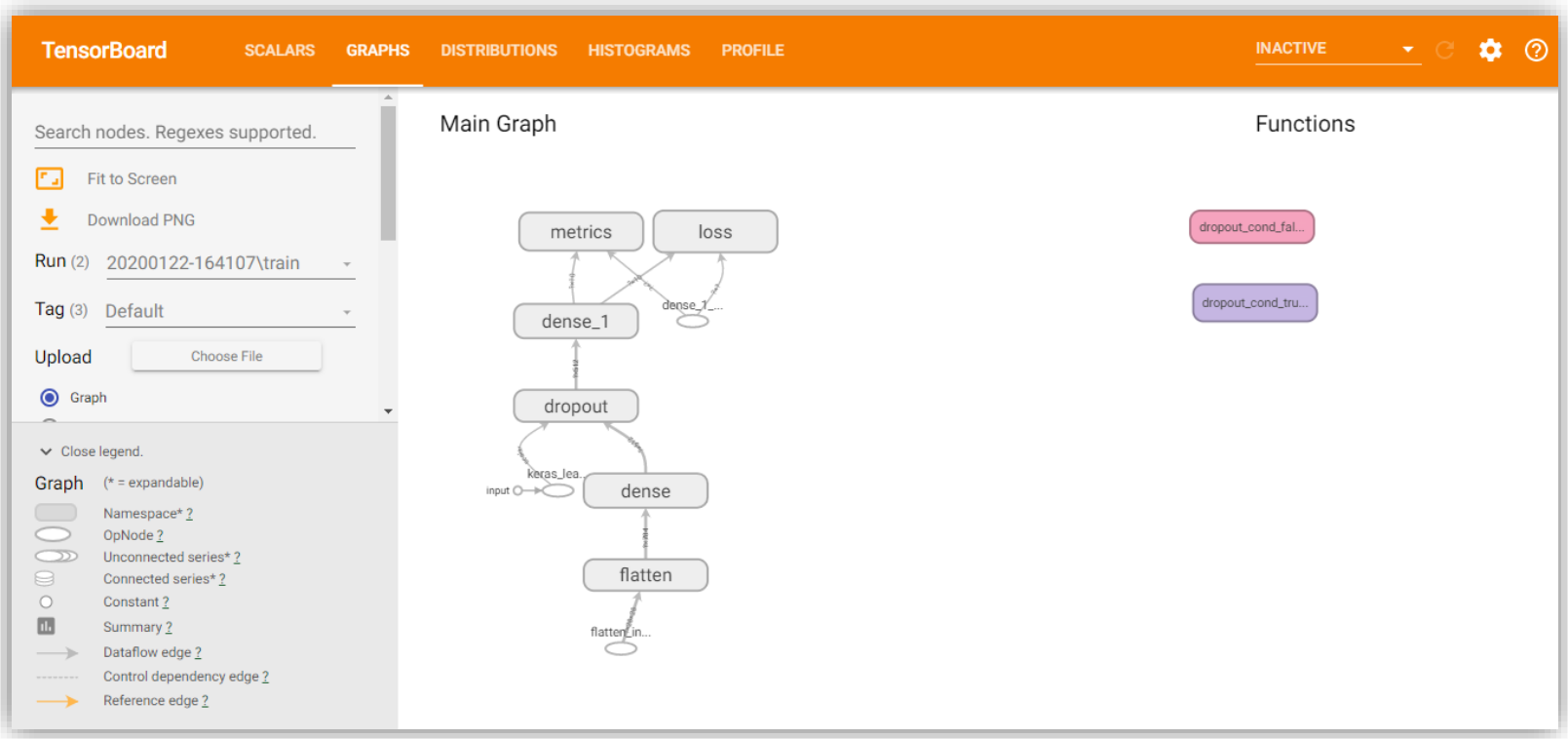
Running it

After executing our program, go to the terminal (or Anaconda Prompt for windows users) and run:



Running it

Then, just open your browser and go to http://localhost:7070/
(change to the selected port)



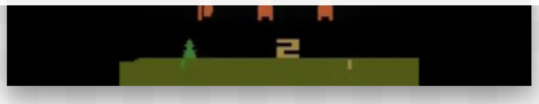
OpenAI Gym

Environments Documentation

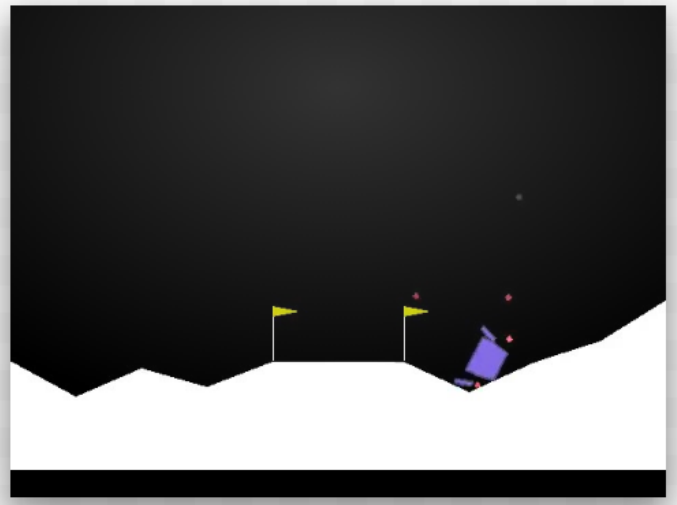
Gym

Gym is a toolkit for developing and comparing reinforcement learning algorithms. It supports teaching agents everything from walking to playing games like Pong or Pinball.

View documentation ›
View on GitHub ›



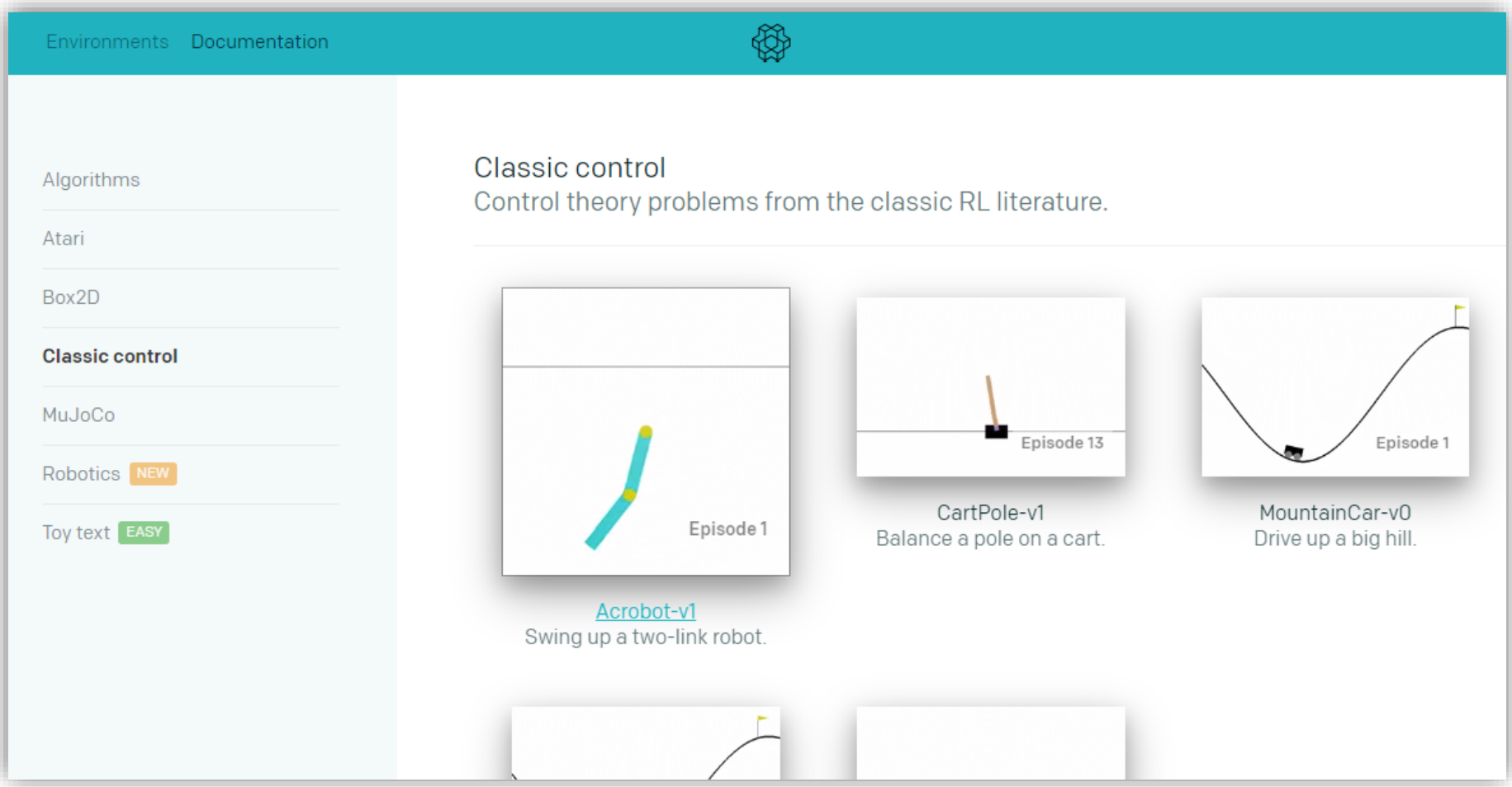
RandomAgent on SpaceInvaders-v0



RandomAgent on LunarLander-v2

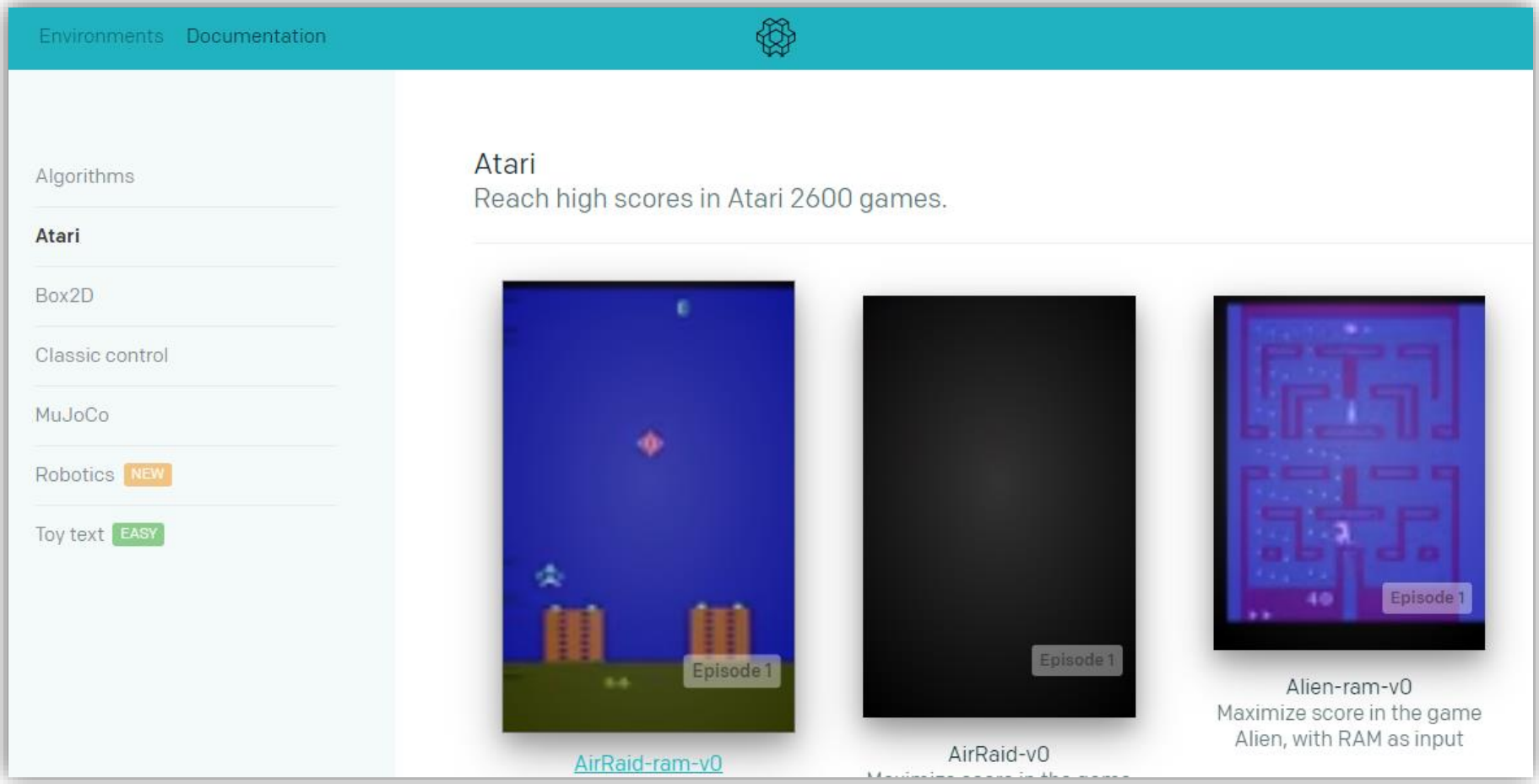
(https://gym.openai.com/)

OpenAI Gym



(https://gym.openai.com/envs)

OpenAI Gym



(https://gym.openai.com/envs)

OpenAI Gym

Environments Documentation

Getting Started with Gym

Gym is a toolkit for developing and comparing reinforcement learning algorithms. It makes no assumptions about the structure of your agent, and is compatible with any numerical computation library, such as TensorFlow or Theano.

The gym library is a collection of test problems — **environments** — that you can use to work out your reinforcement learning algorithms. These environments have a shared interface, allowing you to write general algorithms.

Installation

To get started, you'll need to have Python 3.5+ installed. Simply install `gym` using `pip`:

```
pip install gym
```

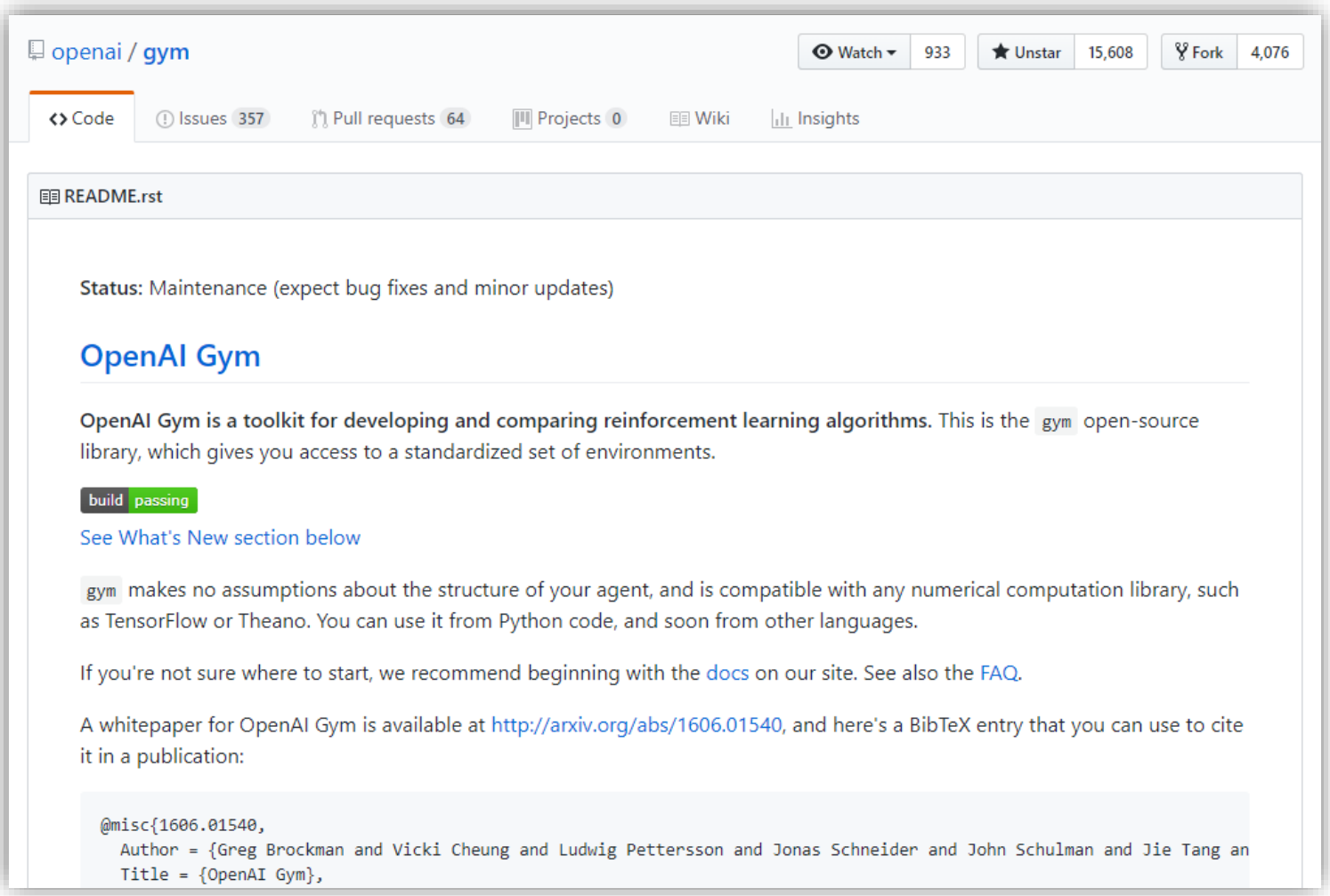
And you're good to go!

Building from Source

If you prefer, you can also clone the Git repository directly. This is particularly useful when you're

(https://gym.openai.com/docs/)

OpenAI Gym



openai / gym

Watch 933 Unstar 15,608 Fork 4,076

Code Issues 357 Pull requests 64 Projects 0 Wiki Insights

README.rst

Status: Maintenance (expect bug fixes and minor updates)

OpenAI Gym

OpenAI Gym is a toolkit for developing and comparing reinforcement learning algorithms. This is the `gym` open-source library, which gives you access to a standardized set of environments.

build passing

See What's New section below

`gym` makes no assumptions about the structure of your agent, and is compatible with any numerical computation library, such as TensorFlow or Theano. You can use it from Python code, and soon from other languages.

If you're not sure where to start, we recommend beginning with the docs on our site. See also the FAQ.

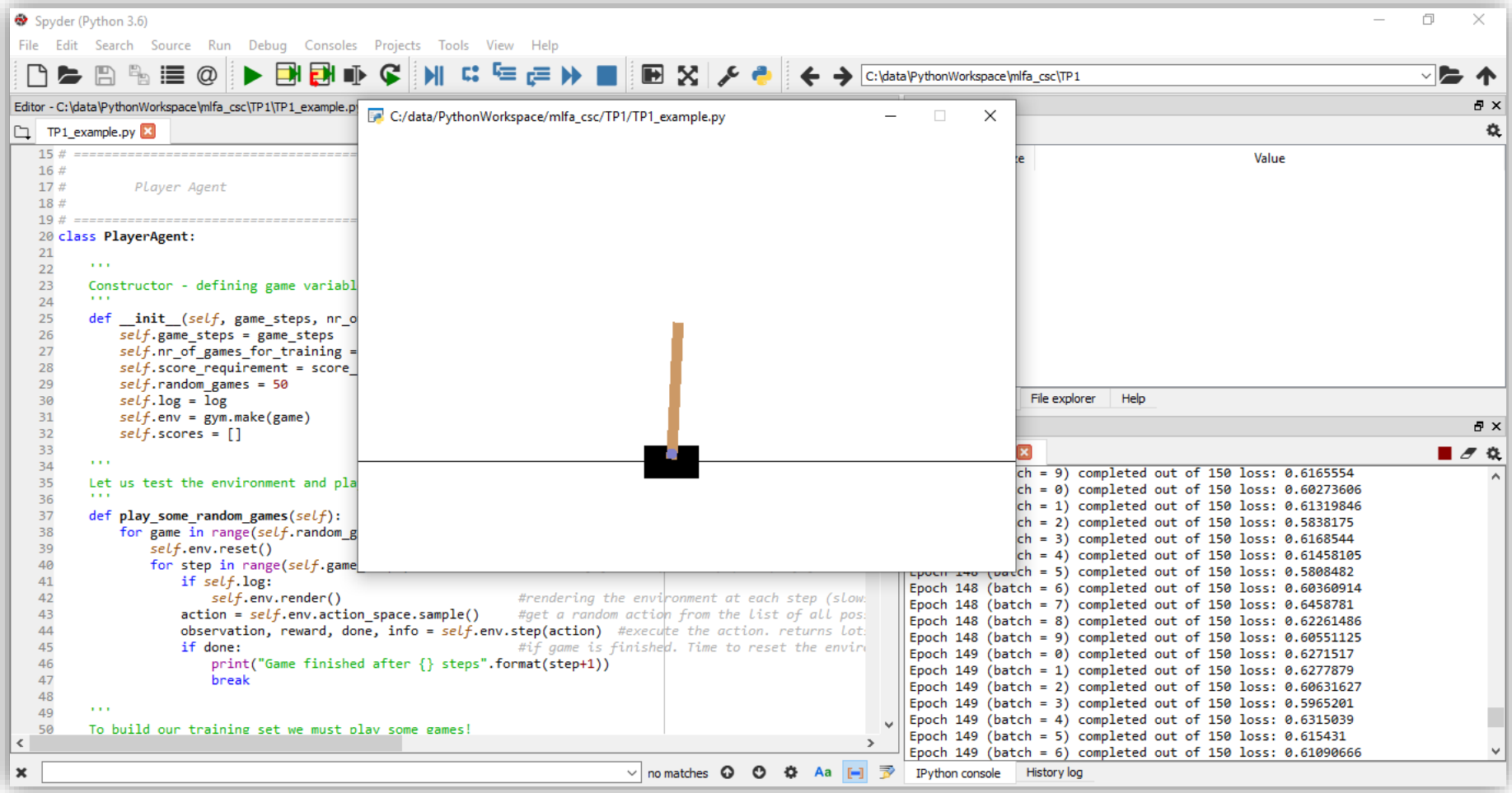
A whitepaper for OpenAI Gym is available at http://arxiv.org/abs/1606.01540, and here's a BibTeX entry that you can use to cite it in a publication:

```
@misc{1606.01540,
  Author = {Greg Brockman and Vicki Cheung and Ludwig Pettersson and Jonas Schneider and John Schulman and Jie Tang an
  Title = {OpenAI Gym},
```

(https://github.com/openai/gym/)

OpenAI Gym

The CartPole-v1 Example

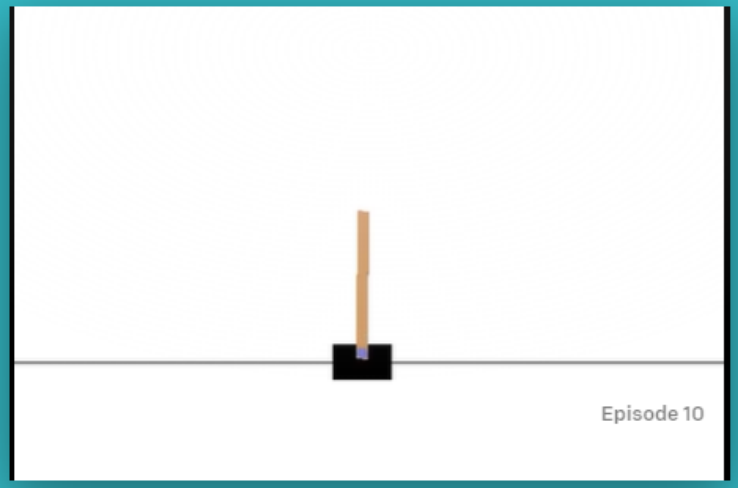


OpenAI Gym

The CartPole-v1 Example

CartPole-v1

A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every timestep that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center.



RandomAgent on CartPole-v1

This environment corresponds to the version of the cart-pole problem described by Barto, Sutton, and Anderson [Barto83].

[Barto83] *AG Barto, RS Sutton and CW Anderson, "Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problem", IEEE Transactions on Systems, Man, and Cybernetics, 1983.*

VIEW SOURCE ON GITHUB

(https://gym.openai.com/envs/CartPole-v1/)

OpenAI Gym

The CartPole-v1 Example


```
class CartPoleEnv(gym.Env):
    """
    Description:
        A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The pendulum starts upright, and the

    Source:
        This environment corresponds to the version of the cart-pole problem described by Barto, Sutton, and Anderson

    Observation:
        Type: Box(4)
        Num     Observation               Min         Max
        0       Cart Position             -4.8            4.8
        1       Cart Velocity             -Inf            Inf
        2       Pole Angle                 -24 deg        24 deg
        3       Pole Velocity At Tip      -Inf            Inf

    Actions:
        Type: Discrete(2)
        Num     Action
        0       Push cart to the left
        1       Push cart to the right

        Note: The amount the velocity that is reduced or increased is not fixed; it depends on the angle the pole is pointing. This is beca

    Reward:
        Reward is 1 for every step taken, including the termination step

    Starting State:
        All observations are assigned a uniform random value in [-0.05..0.05]

    Episode Termination:
        Pole Angle is more than 12 degrees
        Cart Position is more than 2.4 (center of the cart reaches the edge of the display)
        Episode length is greater than 200
        Solved Requirements
        Considered solved when the average reward is greater than or equal to 195.0 over 100 consecutive trials.
    """
```

(https://github.com/openai/gym/blob/master/gym/envs/classic_control/cartpole.py)

OpenAI Gym

The CartPole-v1 Example


```
# ========================================================================
#
#         Main Execution
#
# ========================================================================

playerAgent = PlayerAgent(500, 1000, 75, game='CartPole-v1', log=True)
#playerAgent.play_random_games()
playerAgent.play_the_game()
```

OpenAI Gym
The CartPole-v1 Example


```
# ======================================================================
#
#         Player Agent
#
# ======================================================================
class PlayerAgent:

    '''
    Constructor - defining game variables
    '''
    def __init__(self, game_steps, nr_of_games_for_training, score_requirement, game='CartPole-v1', log=False):
        self.game_steps = game_steps                                    #number of frames (aka steps) to play for each game
        self.nr_of_games_for_training = nr_of_games_for_training        #the number of games that will be played to build the training set
        self.score_requirement = score_requirement                      #the minimum score for a decent game (also for training purposes)
        self.random_games = 50                                          #initial random games to play
        self.log = log                                                  #log info
        self.env = gym.make(game)                                       #make the gaming environment
        self.scores = []                                                #list with all scores when playing


    '''
    Let us test the environment and play some random games!
    '''
    def play_random_games(self):
        pass

    '''
    To build our training set we must play some games!
    Let us start by playing completly random games and save those where we achieve an interesting score.
    You should then use this training set to train the model!!
    '''
    def build_training_set(self):
        pass

    '''
    Let us play the game after been trained to do that!
    '''
    def play_the_game(self):
        pass
```

OpenAI Gym
The CartPole-v1 Example


```
# ==============================================================================
#
#         Player Agent
#
# ==============================================================================
class PlayerAgent:

    '''
    Constructor - defining game variables
    '''
    def __init__(self, game_steps, nr_of_games_for_training, score_requirement, game='CartPole-v1', log=False):
        self.game_steps = game_steps                                    #number of frames (aka steps) to play for each game
        self.nr_of_games_for_training = nr_of_games_for_training        #the number of games that will be played to build the training set
        self.score_requirement = score_requirement                      #the minimum score for a decent game (also for training purposes)
        self.random_games = 50                                          #initial random games to play
        self.log = log                                                  #log info
        self.env = gym.make(game)                                       #make the gaming environment
        self.scores = []                                                #list with all scores when playing
```

- Start by calling gym.make('SOME_GAME') to prepare the game environment!
- Save it to a variable (let's call it *env*).

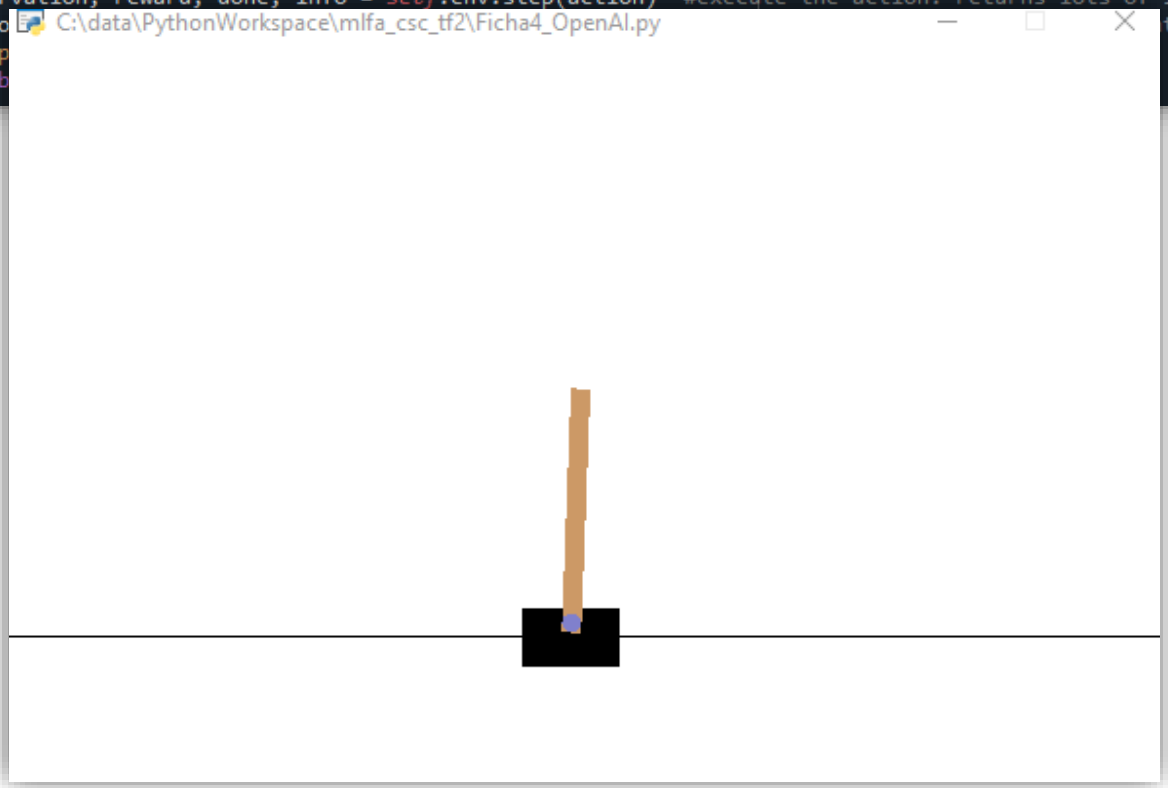
OpenAI Gym
The CartPole-v1 Example


```
'''
Let us test the environment and play some random games!
'''
def play_random_games(self):
    for game in range(self.random_games):              #let's play self.random_games games
        self.env.reset()                               #reset the environment to initial state. returns the first observation of th
        for step in range(self.game_steps):            #number of frames (aka steps) to play for each game
            if self.log:
                self.env.render()                      #rendering the environment at each step (slows things down)
            action = self.env.action_space.sample()    #get a random action from the list of all possible actions
            observation, reward, done, info = self.env.step(action)  #execute the action. returns lots of information
            if done:                                   #if game is finished. Time to reset the environment again
                print("Game finished after {} steps".format(step+1))
                break
```

Let’s play some random games (fool behaviour is expected!!)

OpenAI Gym
The CartPole-v1 Example


```
'''
Let us test the environment and play some random games!
'''
def play_random_games(self):
    for game in range(self.random_games):              #let's play self.random_games games
        self.env.reset()                               #reset the environment to initial state. returns the first observation of th
        for step in range(self.game_steps):            #number of frames (aka steps) to play for each game
            if self.log:
                self.env.render()                      #rendering the environment at each step (slows things down)
            action = self.env.action_space.sample()    #get a random action from the list of all possible actions
            observation, reward, done, info = self.env.step(action)  #execute the action. returns lots of information
            if do                                                    t again
                p
                b
```



OpenAI Gym
The CartPole-v1 Example


```
    '''
    Let us test the environment and play some random games!
    '''
    def play_random_games(self):
        for game in range(self.random_games):                #let's play self.random_games games
            self.env.reset()                                 #reset the environment to initial state. returns the first observation of th
            for step in range(self.game_steps):              #number of frames (aka steps) to play for each game
                if self.log:
                    self.env.render()                        #rendering the environment at each step (slows things down)
                action = self.env.action_space.sample()      #get a random action from the list of all possible actions
                observation, reward, done, info = self.env.step(action)  #execute the action. returns lots of information
                if done:                                     #if game is finished. Time to reset the environment again
                    print("Game finished after {} steps".format(step+1))
                    break
```

- Call env.reset() to make the game environment ready to be played!
- Call env.render() to render the environment at each step (slows everything considerably)
- Call env.action_space.sample() to get a random action from all available actions (left | right)

OpenAI Gym
The CartPole-v1 Example


```
    '''
    Let us test the environment and play some random games!
    '''
    def play_random_games(self):
        for game in range(self.random_games):                #let's play self.random_games games
            self.env.reset()                                 #reset the environment to initial state. returns the first observation of th
            for step in range(self.game_steps):              #number of frames (aka steps) to play for each game
                if self.log:
                    self.env.render()                        #rendering the environment at each step (slows things down)
                action = self.env.action_space.sample()      #get a random action from the list of all possible actions
                observation, reward, done, info = self.env.step(action)  #execute the action. returns lots of information
                if done:                                     #if game is finished. Time to reset the environment again
                    print("Game finished after {} steps".format(step+1))
                    break
```

- Call env.step(*action*) to execute the *action* and get information about the consequence of having executed that same action!
  - observation: an environment-specific object representing your observation of the environment (check slide 74 to see what this environment returns)
  - reward: mount of reward achieved by the executed action
  - done: is time to reset the environment again?
  - info: useful for debugging!

OpenAI Gym
The CartPole-v1 Example


```
def build_training_set(self):
    training_set = []
    score_set = []

    for game in range(self.nr_of_games_for_training):
        cumulative_game_score = 0
        game_memory = []
        obs_prev = self.env.reset()
```

How can we train our agent so that it can play the game?

We must first collect some data!

OpenAI Gym
The CartPole-v1 Example


```
def build_training_set(self):
    training_set = []
    score_set = []

    for game in range(self.nr_of_games_for_training):
        cumulative_game_score = 0
        game_memory = []
        obs_prev = self.env.reset()

        for step in range(self.game_steps):
            #env.render()                                          #best not to render the game!
            action = self.env.action_space.sample()               #some random action
            obs_next, reward, done, info = self.env.step(action)  #execute it and gather the results
            game_memory.append([action, obs_prev])                #store the execute action and the corresponding observation
            cumulative_game_score += reward                       #store cumulative reward per game
            obs_prev = obs_next
            if done:
                #print("Game finished after {} steps".format(step+1))   #for debug purposes
                break
```

How can we train our agent so that it can play the game?

We must first collect some data!

OpenAI Gym
The CartPole-v1 Example


```
def build_training_set(self):
    training_set = []
    score_set = []

    for game in range(self.nr_of_games_for_training):
        cumulative_game_score = 0
        game_memory = []
        obs_prev = self.env.reset()

        for step in range(self.game_steps):
            #env.render()                                          #best not to render the game!
            action = self.env.action_space.sample()               #some random action
            obs_next, reward, done, info = self.env.step(action)  #execute it and gather the results
            game_memory.append([action, obs_prev])                #store the execute action and the corresponding observation
            cumulative_game_score += reward                       #store cumulative reward per game
            obs_prev = obs_next
            if done:
                #print("Game finished after {} steps".format(step+1))   #for debug purposes
                break

        #the game is finished. Was it a decent game?
        if cumulative_game_score > self.score_requirement:
            score_set.append(cumulative_game_score)
            for play in game_memory:
                if play[0] == 0:
                    one_hot_action = [1, 0]
                elif play[0] == 1:
                    one_hot_action = [0, 1]

                training_set.append([one_hot_action, play[1]])
```

How can we train our agent so that it can play the game?

We must first collect some data!

OpenAI Gym
The CartPole-v1 Example


```
def build_training_set(self):
    training_set = []
    score_set = []

    for game in range(self.nr_of_games_for_training):
        cumulative_game_score = 0
        game_memory = []
        obs_prev = self.env.reset()

        for step in range(self.game_steps):
            #env.render()                                          #best not to render the game!
            action = self.env.action_space.sample()               #some random action
            obs_next, reward, done, info = self.env.step(action)  #execute it and gather the results
            game_memory.append([action, obs_prev])                #store the execute action and the corresponding observation
            cumulative_game_score += reward                       #store cumulative reward per game
            obs_prev = obs_next
            if done:
                #print("Game finished after {} steps".format(step+1))   #for debug purposes
                break

        #the game is finished. Was it a decent game?
        if cumulative_game_score > self.score_requirement:
            score_set.append(cumulative_game_score)
            for play in game_memory:
                if play[0] == 0:
                    one_hot_action = [1, 0]
                elif play[0] == 1:
                    one_hot_action = [0, 1]

                training_set.append([one_hot_action, play[1]])

    #print some stats
    if score_set:
        print('Average score:', mean(score_set))
        print('Number of stored games per score', Counter(score_set))

    return training_set
```

How can we train our agent so that it can play the game?

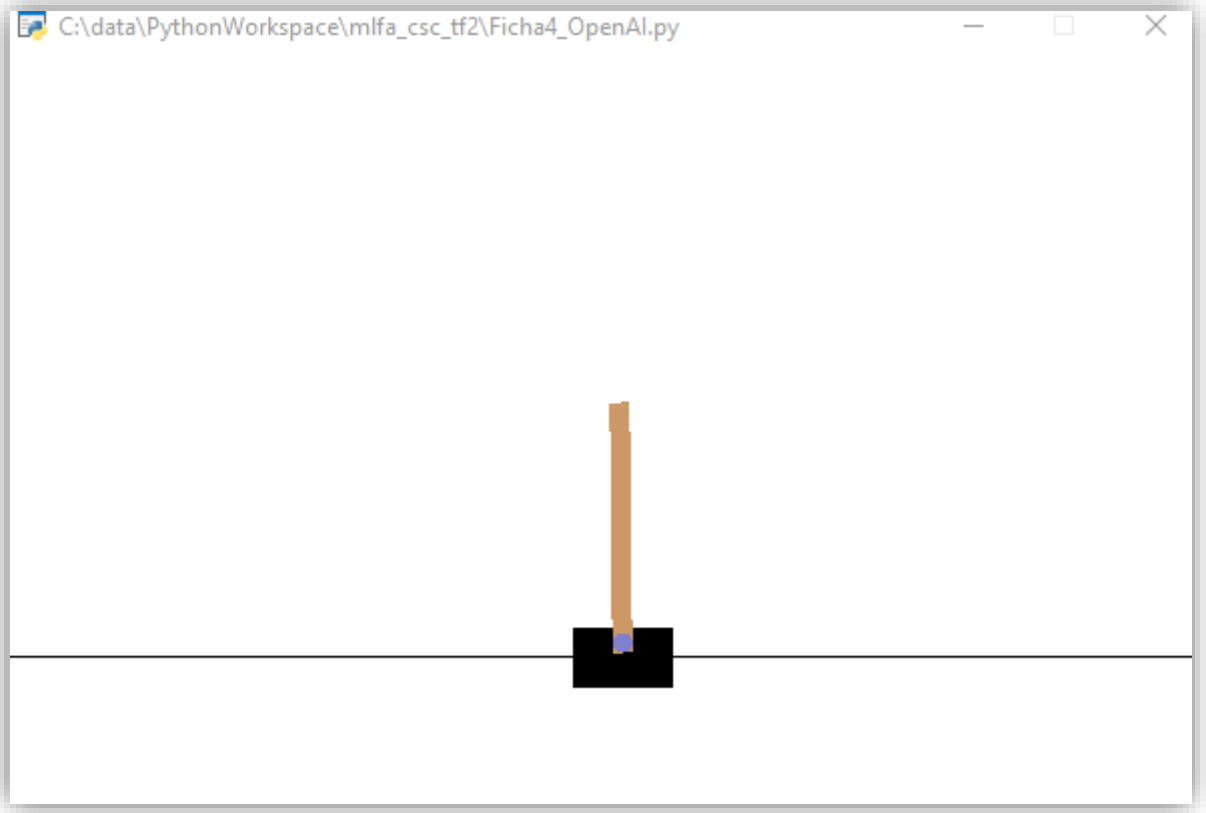
We must first collect some data!

OpenAI Gym
The CartPole-v1 Example


```
'''
Let us play the game after been trained to do that!
'''
def play_the_game(self):
    #build training set
    training_set = self.build_training_set()

    #using the model
    mlp = MLP()
    mlp.build()
    mlp.fit(training_set)

    for game in range(self.nr_of_games_for_training):
        score = 0
        obs_prev = self.env.reset()
        done = False

        while not done:
            if self.log:
                self.env.render()
            action = mlp.predict(obs_prev)
            obs_next, reward, done, _ = self.env.step(action)
            score += reward
            obs_prev = obs_next

        self.scores.append(score)
        print('Current score: ' + str(score) + '; Average score:', sum(self.scores)/len(self.scores))
```

To play the game we (1) collect the training set (or persist first and load later), (2) build our MLP, (3) train it with the training set, and (4) we start playing the game using the action returned by our model when it receives the current observation of the environment!

OpenAI Gym
The CartPole-v1 Example



Much better!! But it can be (substantially) improved!!!

How?

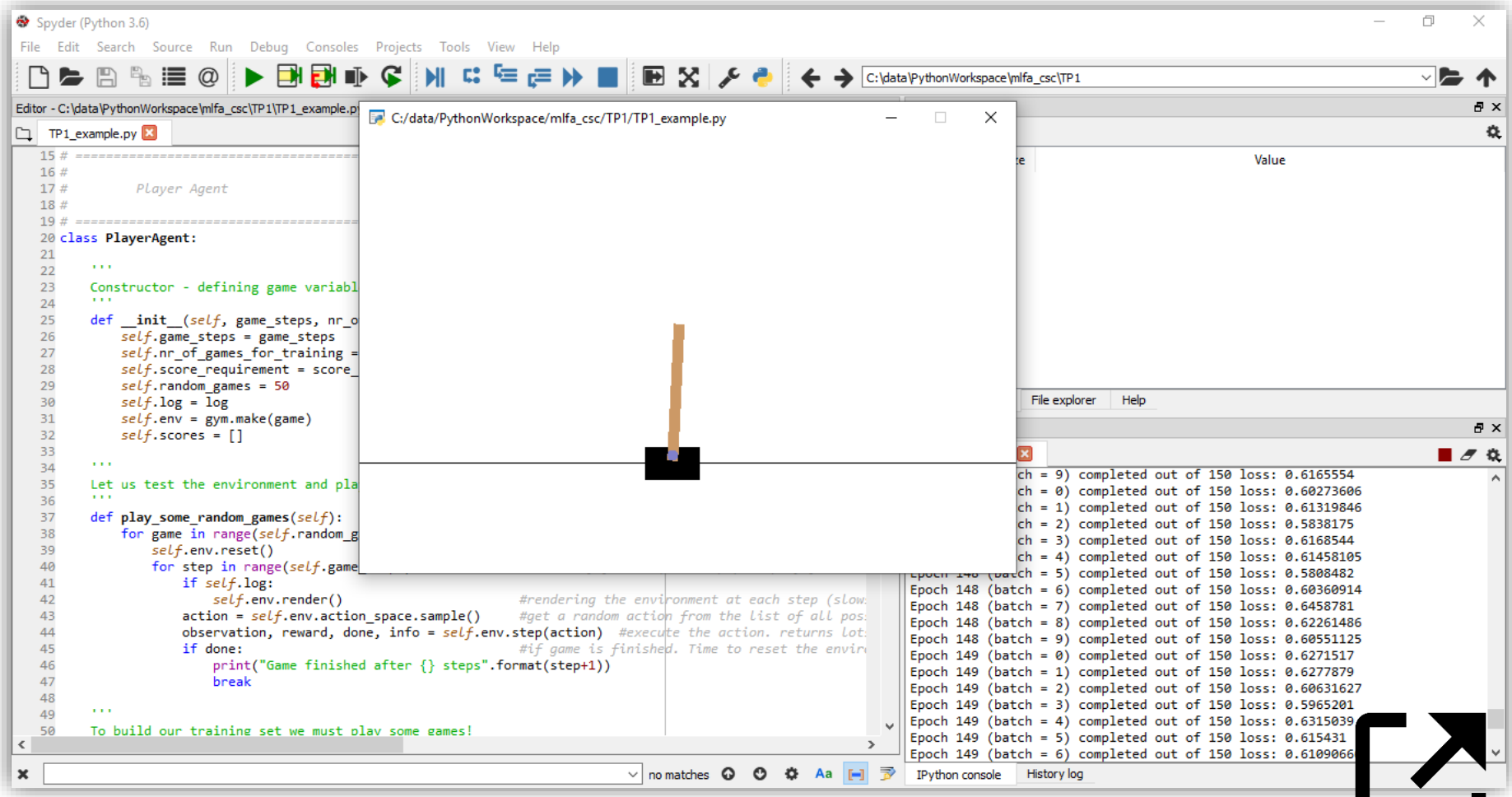
OpenAI Gym
The CartPole-v1 Example


```
# ============================================================================
#
#         Multi-Layer Perceptron
#
# ============================================================================
class MLP:

    '''
    Constructor - defining game variables
    '''
    def __init__(self, epochs=1, batch_size=32, output_neurons=2):
        self.epochs = epochs
        self.batch_size = batch_size
        self.output_neurons = output_neurons

    def prepare_data(self, training_set):
        #reshape X to list of inputs
        X = np.array([i[1] for i in training_set], dtype="float32").reshape(-1, len(training_set[0][1]))
        #y are the actions
        y = np.array([i[0] for i in training_set])
        return X, y

    def build(self):
        pass

    def fit(self, training_set):
        pass

    def predict(self, obs):
       pass
```

Now… You just need to build a MLP!!

OpenAI Gym

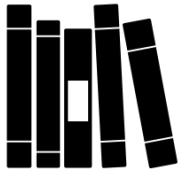
The CartPole-v1 Example



Glossary

- Callbacks: an object passed to a model to customize and extend its behavior during training
- Checkpoints: a file that captures the values of all tf.Variable objects used by a model allowing the ability to stop the training at any point, for any reason, and resume it later as if nothing happened
- Keras: a high-level neural networks API, written in Python and capable of running on top of TensorFlow. tf.Keras is TensorFlow's official high-level API
- OpenAI Gym: a toolkit for developing and comparing reinforcement learning algorithms. It supports teaching agents everything from walking to playing games like Pong or Pinball
- Tensorboard: a set of visualization tools to make it easier to understand, debug and optimize TensorFlow programs
- tf.Data: a TensorFlow API to handle large amounts of data, read from different data formats, and perform complex transformations

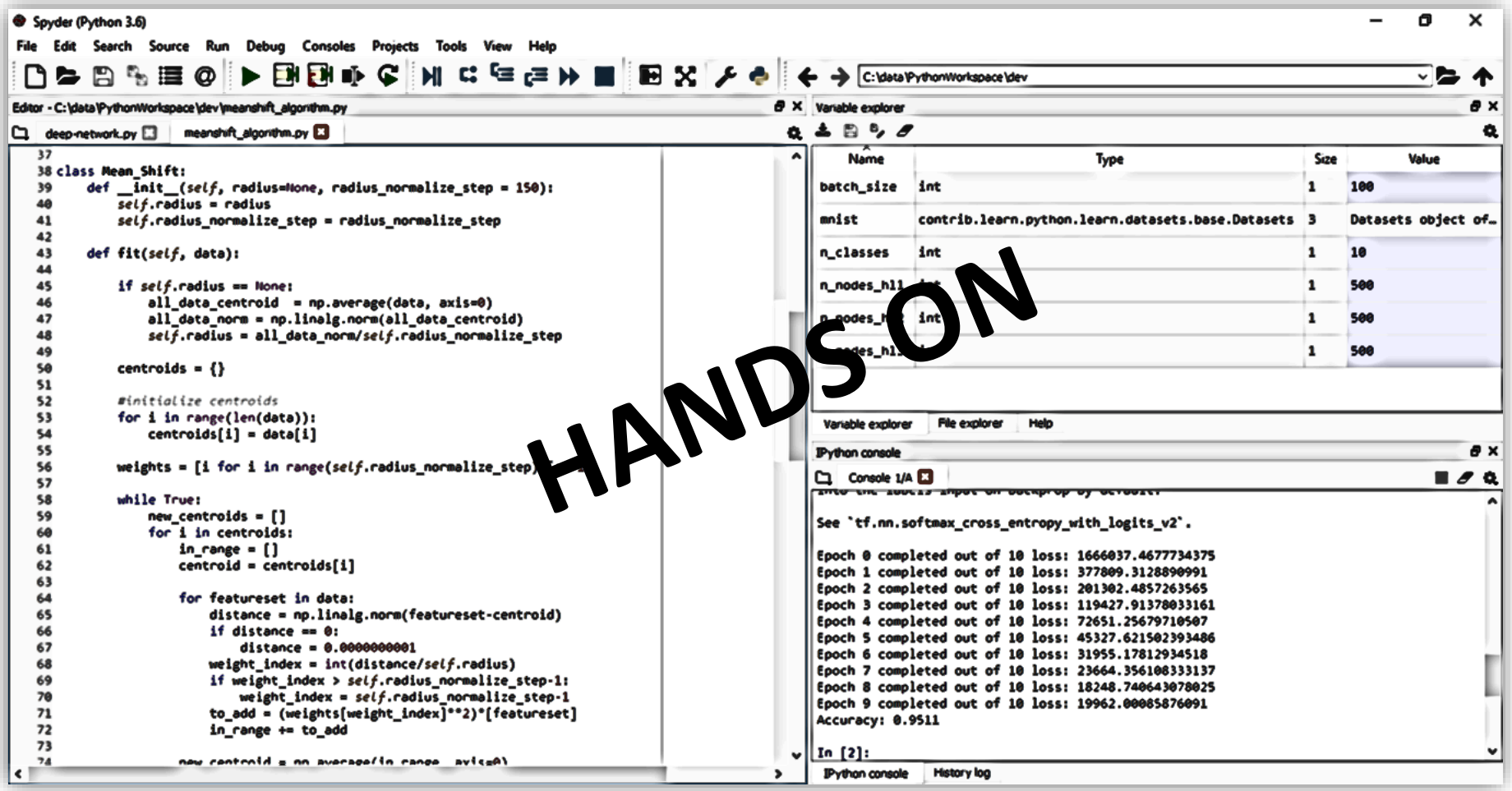
Resources



- Official Documentation
  - https://www.tensorflow.org/api_docs
  - https://www.tensorflow.org/guide/data
  - https://www.tensorflow.org/tutorials/load_data
  - https://www.tensorflow.org/guide/keras/custom_callback
  - https://www.tensorflow.org/guide/saved_model
  - https://gym.openai.com/docs/
  - https://github.com/openai/gym
  - …

Hands On


K
Keras
TensorFlow


Spyder (Python 3.6)
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\data\PythonWorkspace\dev
Editor - C:\data\PythonWorkspace\dev\meanshift_algorithm.py
deep-network.py
meanshift_algorithm.py
class Mean_Shift:
def __init__(self, radius=None, radius_normalize_step = 150):
self.radius = radius
self.radius_normalize_step = radius_normalize_step
def fit(self, data):
if self.radius == None:
all_data_centroid = np.average(data, axis=0)
all_data_norm = np.linalg.norm(all_data_centroid)
self.radius = all_data_norm/self.radius_normalize_step
centroids = {}
#initialize centroids
for i in range(len(data)):
centroids[i] = data[i]
weights = [i for i in range(self.radius_normalize_step)]
while True:
new_centroids = []
for i in centroids:
in_range = []
centroid = centroids[i]
for featureset in data:
distance = np.linalg.norm(featureset-centroid)
if distance == 0:
distance = 0.0000000001
weight_index = int(distance/self.radius)
if weight_index > self.radius_normalize_step-1:
weight_index = self.radius_normalize_step-1
to_add = (weights[weight_index]**2)*[featureset]
in_range += to_add
Variable explorer
Name Type Size Value
batch_size int 1 100
mnist contrib.learn.python.learn.datasets.base.Datasets 3 Datasets object of…
n_classes int 1 10
n_nodes_hl1 1 500
n_nodes_hl2 int 1 500
n_nodes_hl3 1 500
Variable explorer File explorer Help
IPython console
Console 1/A
See `tf.nn.softmax_cross_entropy_with_logits_v2`.
Epoch 0 completed out of 10 loss: 1666037.4677734375
Epoch 1 completed out of 10 loss: 377809.3128890991
Epoch 2 completed out of 10 loss: 201302.4857263565
Epoch 3 completed out of 10 loss: 119427.91378033161
Epoch 4 completed out of 10 loss: 72651.25679710507
Epoch 5 completed out of 10 loss: 45327.621502393486
Epoch 6 completed out of 10 loss: 31955.17812934518
Epoch 7 completed out of 10 loss: 23664.356108333137
Epoch 8 completed out of 10 loss: 18248.740643078025
Epoch 9 completed out of 10 loss: 19962.00085876091
Accuracy: 0.9511
In [2]:
IPython console History log
HANDS ON