[Lab] Introduction to Tensorflow 1.x

Jae Yun JUN KIM*

November 9, 2020

Source:

- Nishant Shukla, Machine Learning with TensorFlow, Manning Publications, Shelter Island, NY, USA, 2018
- Armando Fandango, Mastering TensorFlow 1.x, Packt, 2018

Notice: The version of TensorFlow that we employ in this document is of version 1.x.

1 Installation

Step 1: Verify the Python version by typing the following command in a terminal:

```
python
```

Step 2: Before we install TensorFlow, install Anaconda framework in the system. Go to anaconda project site, download it and install it. After successful installation, verify that it properly works by typing the following command in a terminal:

```
conda
```

Step 3: Execute the following command to initialize the installation of TensorFlow, supposing that your Python version is 3.5. If it is a different version, then type the one that suits for you:

```
conda create --name tensorflow python=3.5
```

Ste 4: After successful environmental setup, activate TensorFlow module by typing the following command in a terminal:

```
activate tensorflow
```

Step 5: Use *pip* to install "TensorFlow" in the system by typing the following command in a terminal:

```
pip install tensorflow
```

and also

*ECE Paris Graduate School of Engineering, 37 quai de Grenelle 75015 Paris, France; jae-yun.jun-kim@ece.fr

```
pip install tensorflow-gpu
```

Step 6: Then, type the following commands in a terminal:

```
activate tensorflow
python
```

Then, the Python shell will be activated, from which you write the first TensorFlow program:

```
import tensorflow as tf
hello = tf.constant('Hello, Tensorflow!')
sess = tf.Session()
print(sess.run(hello))
```

2 Essentials of TensorFlow

2.1 Representing tensors

The first thing that you need to define to work with TensorFlow is a tensor. There are three ways to do so:

```
import tensorflow as tf
import numpy as np

# define a 2 x 2 matrix in three ways.
m1 = [[1.0, 2.0],
      [3.0, 4.0]]
m2 = np.array([[1.0, 2.0],
               [3.0, 4.0]], dtype=np.float32)
m3 = tf.constant([[1.0, 2.0],
                  [3.0, 4.0]])

# print these matrices
print(type(m1))
print(type(m2))
print(type(m3))

# create tensor objects out of the above three types
t1 = tf.convert_to_tensor(m1, dtype=tf.float32)
t2 = tf.convert_to_tensor(m2, dtype=tf.float32)
t3 = tf.convert_to_tensor(m3, dtype=tf.float32)

# notice that the types are the same now.
print(type(t1))
print(type(t2))
print(type(t3))
```

Some other examples for creating tensors are below:

```
import tensorflow as tf

# a 2x1 matrix
m1 = tf.constant([[1.0, 2.0]])

# a 1x2 matrix
m2 = tf.constant([[1.0], [2.0]])

# a rank-3 tensor
m3 = tf.constant([ [[1.,2.],
                    [3.,4.],
                    [5.,6.]],
                    [[7.,8.],
                    [9.,10.],
                    [11.,12.]] ])

# print these tensors
print(m1)
print(m2)
print(m3)
```

TensorFlow also comes with a few convenient constructors for some simple tensors. For example, **tf.zeros(shape)** and **tf.ones(shape)**, where *shape* is a one-dimensional (1D) tensor of type *int32* (a list of integers) describing the dimensions of the tensor.

2.2 Creating operators

Useful TensorFlow operators are:

- **tf.add(x,y)**: adds two tensors of the same type, x+y
- **tf.subtract(x,y)**: subtracts tensors of the same type, x-y
- **tf.multiply(x,y)**: multiplies two tensors element-wise
- **tf.pow(x,y)**: takes the element-wise x to the power of y
- **tf.exp(x)**: equivalent to $pow(e, x)$, where e is Euler's number (2.718...)
- **tf.sqrt(x)**: equivalent to $pow(x, 0.5)$
- **tf.div(x,y)**: takes the element-wise division of x and y
- **tf.truediv(x,y)**: same as $tf.div$, except casts the arguments as a float
- **tf.floordiv(x,y)**: same as *truediv*, except rounds down the final answer into an integer
- **tf.mod(x,y)**: takes the element-wise remainder from division
- **tf.negative(x)**: negates the tensor x

As an example:

```
import tensorflow as tf

x = tf.constant([[1.0, 2.0]])
negMatrix = tf.negative(x)
print(negMatrix)
```

Notice that the output is not [[-1,2]]. That is because we are printing out the definition of the negation operator, not the actual evaluation of the operation. The printed output shows that the negation operation is a **Tensor class** with a name, shape, and data type. The name was autoamtically assigned, but you could have provided it explicitly as well. Similarly, the shape and data type were inferred from the [[1,2]] that you passed in.

2.3 Executing operators with sessions

A **session** is an environment of a software system that describes how the lines of code should run. In TensorFlow, a session sets up how the hardware devices (such as CPU and GPU) talk to each other. That way, you can design your machine-learning algorithm without worrying aobut micromanaging the hardware it runs on. You can later configure the session to change its behavior without changing a line of the machine-learning code. You must create a session class by using **tf.Session()** and tell it to run as operator.

```
import tensorflow as tf

x = tf.constant([[1.0, 2.0]])
neg_op = tf.negative(x)

# starts a session to be able to run operations
with tf.Session() as sess:
  # tells the session to evaluate negMatrix
  result = sess.run(negMatrix)
# prints the resulting matrix
print(negMatrix)
```

A session not only configures *where* your code will be computed on your machine, but also crafts *how* the computation will be laid out in order to parallelize computation.
Every **Tensor** object has an **eval()** function to evaluate the mathematical operations that define its value. But the **eval()** function requires defining a session object for the library to understand how to best use the underlying hardware. Notice that **sess.run()** is equivalent to **Tensor's eval()** function in the context of the session.

When you run TensorFlow code through an interactive environment *for debugging or presentation purposes), it is often easier to create the session in interactive mode, where the session is implicitly part of any call to **eval()**. That way, the session variable does not need to be passed around throughout the code, making it easier to focus on the relevant parts of the algorithm:

```
import tensorflow as tf

# starts an interactive session so the sess variable
# no longer needs to be passed around.
```

```
sess = tf.InteractiveSession()

x = tf.constant([[1.0, 2.0]])
negMatrix = tf.negative(x)

# evaluate now negMatrix without explicitly specifying a session
result = negMatrix.eval()
print(result)

# remember to close the session to free up resources.
sess.close()
```

2.3.1 Understanding code as a graph

Think of every operator as a **node** in a graph. The **edges** between these nodes represent the composition of mathematical functions. Specifically, the **negative** operator we have been studying is a node, and the incoming/outgoing edges of this node are how the **Tensor** transforms. A tensor flows through the graph, which is why this library is called TensorFlow.

Here is an example: every operator is a strongly typed function that takes input tensors of a dimension and produces output of the same dimension. Figure 2.3.1 shows how the Gaussian function can be designed using TensorFlow. The function is represented as a graph in which operators are nodes and edges represent interactions between nodes. This graph, as a whole, represents the Gaussian function

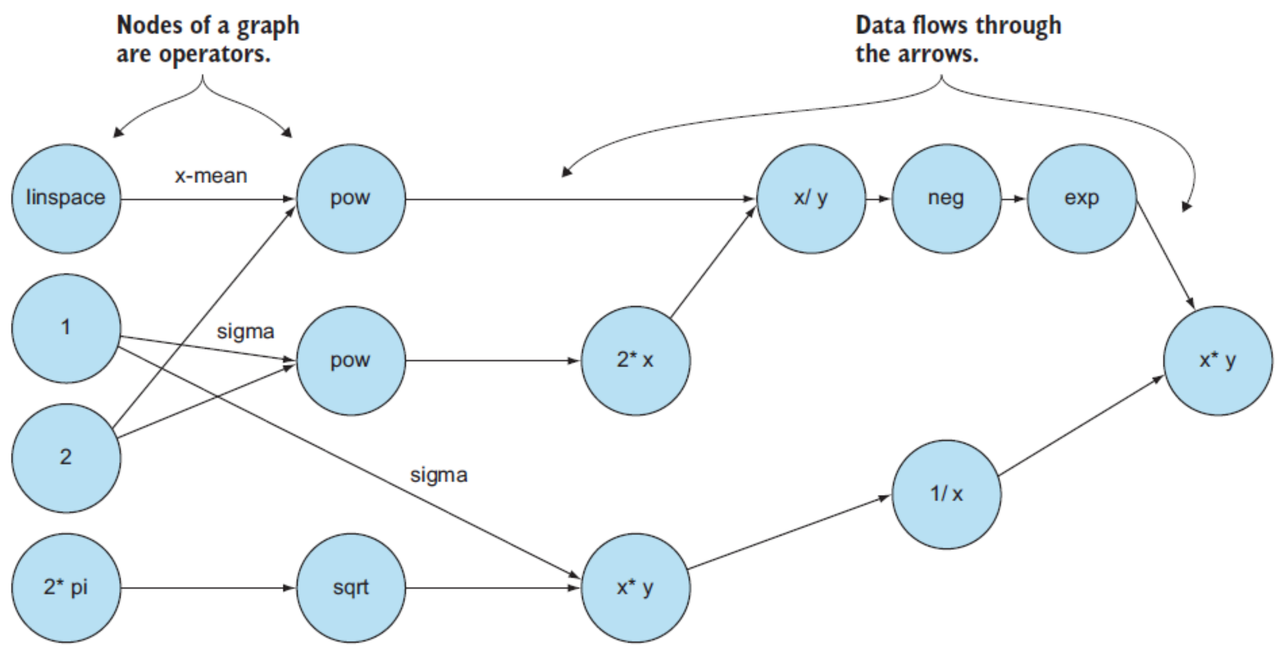

Figure 1: TensorFlow: a dataflow graph

The technical term for such a flowchart is a **dataflow graph**. Every arrow in a dataflow graph is called an **edge**. In addition, every state of the dataflow graph is called a **node**. The purpose of the session is to interpret your Python code into a dataflow graph, and then associate the computation of each node of the graph to the CPU or GPU.

2.3.2 Setting session configurations

You can also pass options to **tf.Session**. For example, TensorFlow automatically determines the best way to assign a GPU or CPU device to an operation, depending on what is available. You can pass an additional option, **log_device_placements=True**, when creating a session, which will show you exactly where on your hardware the computations are evoked.

```
import tensorflow as tf

# defines a matrix and negates it
x = tf.constant([[1.0, 2.0]])
negMatrix = tf.negative(x)

# starts the session with a special config passed
with tf.Session(config=tf.ConfigProto(log_device_placement=True)) as sess:
  # evaluate now negMatrix
  result = sess.run(negMatrix)

# printing the result value
print(result)
```

This outputs info about which CPU/GPU devices are used in the session for each operation.

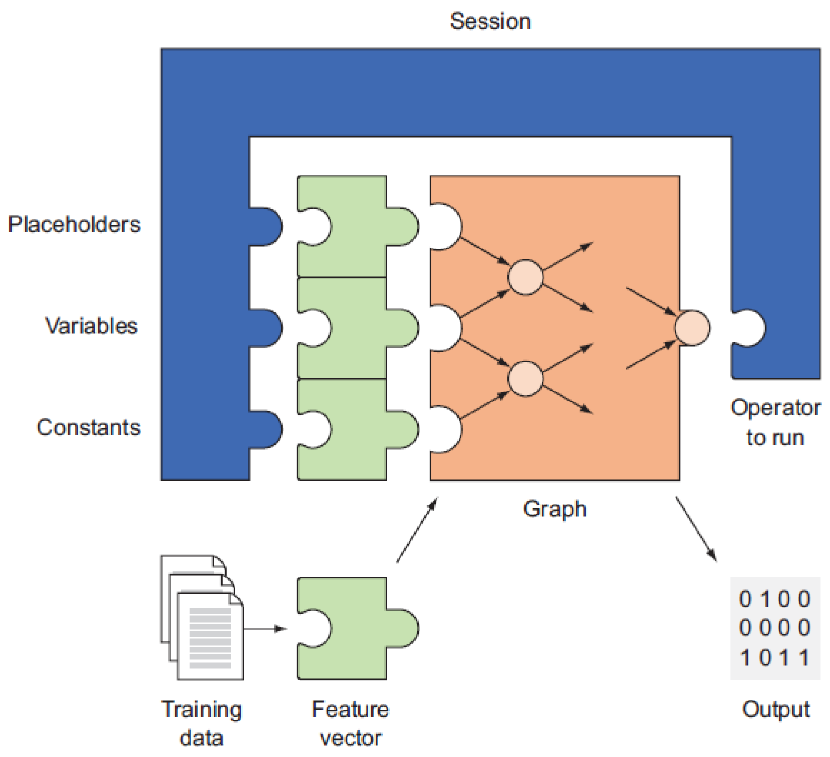

Figure 2: Session, placeholders, variables, and constants

Sessions are essential in TensorFlow code. You need to call a session to "run" the math. Figure 2.3.2 maps out how the components on TensorFlow interact with the machine-learning pipeline. A session not only runs a graph operation, but also can take placeholders, variables, and constants as input. We have used constants so far, but in later sections we start to use variables and placeholders. Here is a quick overview of these three types of values:

- **Placeholder**: A value that is unassigned but will be initialized by the session wherever it is run. Typically, placeholders are the input and output of your model.
- **Variable**: A value that can change, such as parameters of a machine-learning model. Variables must be initialized by the session before they are used.
- **Constant**: A value that does not change, such as hyperparameters or settings.

Most of the code in TensorFlow consists of setting up the graph and session.

2.4 Using variables

You can use the **Variable** class to represent a node whose value changes over time.
In the following code, start with importing TensorFlow, and use **tf.InteractiveSession()**. When you are declared an interactive session, TensorFlow functions do not require the session attribute they would otherwise.

```
import tensorflow as tf
# start the session in interactive mode, so you won't need to pass around sess
sess = tf.InteractiveSession()

# let us say that you have some raw data like this
raw_data = [1., 2., 8., -1., 0., 5.5, 6., 13]

# create a Boolean variable called spike to detect a sudden increase in a series of numbers.
spike = tf.Variable(False)

# because all variables must be initialized,
# initialize the variable by calling run() on its initializer
spike.initializer.run()

# loops through the data (skipping the first element)
# and updates the spike variable when there is a significant increase.
for i in range(1, len(raw_data)):
  if raw_data[i] - raw_data[i-1] > 5:

    # to update a variable, assign it a new value using tf.assign(<var name>, <new value>).
    # Evaluate it to see the change.
    updater = tf.assign(spike, True)
    updater.eval()
  else:
    tf.assign(spike, False).eval()
  print("Spike", spike.eval())

# remember to close the session after it will no longer be used.
sess.close()
```

3 Regression

3.1 Liner regression

Let us first consider a linear regression problem

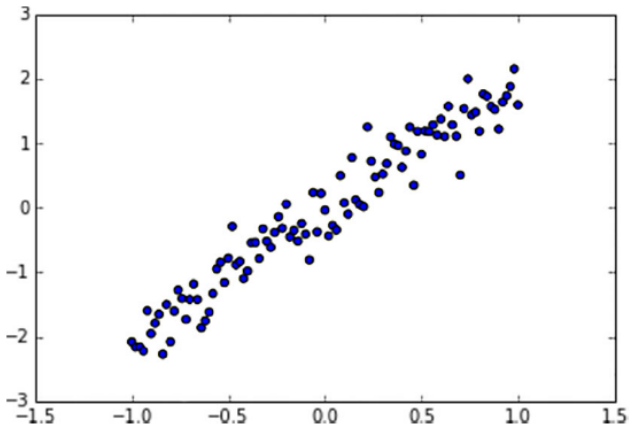

Figure 3: Data for linear regression

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
learning_rate = 0.01
training_epochs = 100

# generate synthetically training data
x_train = np.linspace(-1, 1, 101)
y_train = 2 * x_train + np.random.randn(*x_train.shape) * 0.33

# Sets up the input and output nodes as placeholders
# because the value will be injected by x_train and y_train
X = tf.placeholder(tf.float32)
Y = tf.placeholder(tf.float32)


# Defines the model as y = w*X
def model(X, w):
  return tf.multiply(X, w)

# Sets up the weights variable
w = tf.Variable(0.0, name="weights")

# Defines the cost function
y_model = model(X, w)
cost = tf.square(Y-y_model)

# Defines the operation that will be called on each iteration of the learning algorithm
train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)

# Sets up a session and initializes all variables
sess = tf.Session()
init = tf.global_variables_initializer()
sess.run(init)

# Loops through the dataset multiple times
for epoch in range(training_epochs):
  # Loops through each item in the dataset
  for (x, y) in zip(x_train, y_train):
    # Updates the model parameter(s) to try to minimize the cost function
    sess.run(train_op, feed_dict={X: x, Y: y})

# Obtains the final parameter value
w_val = sess.run(w)
```

```
# Closes the session
sess.close()
# Plots the original data
plt.scatter(x_train, y_train)

# Plots the best-fit line
y_learned = x_train*w_val
plt.plot(x_train, y_learned, 'r')
plt.show()
```

Once you execute the above code you should get the results similar to those shown in Figure 3.1.

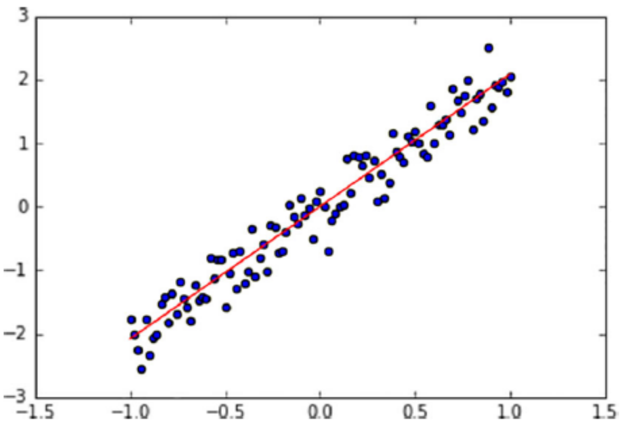

Figure 4: Regression solution

3.2 Polynomial regression

Let us first consider a polynomial regression problem

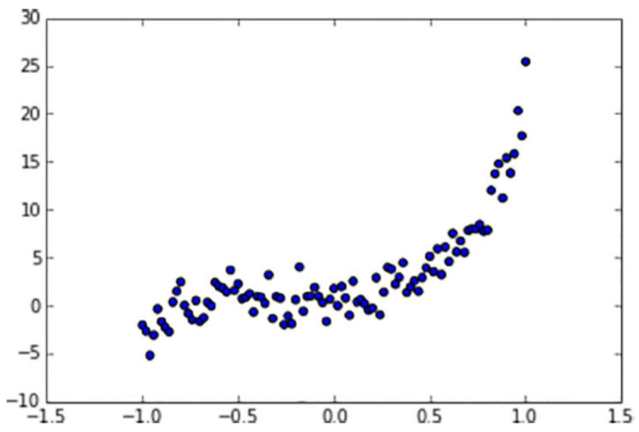

Figure 5: Data for polynomial regression

```
# Imports the relevant libraries and initializes the hyperparameters
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
learning_rate = 0.01
training_epochs = 40

# Sets up synthetic raw input data
trX = np.linspace(-1, 1, 101)

# Sets up raw output data based on a fifth degree polynomial
num_coeffs = 6
trY_coeffs = [1, 2, 3, 4, 5, 6]
trY = 0
for i in range(num_coeffs):
  trY += trY_coeffs[i] * np.power(trX, i)

# Adds noise
trY += np.random.randn(*trX.shape) * 1.5

# Shows a scatter plot of the raw data
plt.scatter(trX, trY)
plt.show()

# Defines the nodes to hold values for input/output pairs
X = tf.placeholder(tf.float32)
Y = tf.placeholder(tf.float32)

# Defines the polynomial model
def model(X, w):
  terms = []
  for i in range(num_coeffs):
    term = tf.multiply(w[i], tf.pow(X, i))
    terms.append(term)
  return tf.add_n(terms)

# Sets up the parameter vector to all zeros
w = tf.Variable([0.] * num_coeffs, name="parameters")
y_model = model(X, w)

# Defines the cost function just as before
cost = (tf.pow(Y-y_model, 2))
train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)

# Sets up the session and runs the learning algorithm just as before
sess = tf.Session()
init = tf.global_variables_initializer()
```

```
sess.run(init)

for epoch in range(training_epochs):
  for (x, y) in zip(trX, trY):
    sess.run(train_op, feed_dict={X: x, Y: y})

w_val = sess.run(w)
print(w_val)

# Closes the session when done
sess.close()

# Plots the result
# raw data
plt.scatter(trX, trY)

# polynomial regressor
trY2 = 0
for i in range(num_coeffs):
  trY2 += w_val[i] * np.power(trX, i)

plt.plot(trX, trY2, 'r')
plt.show()
```

Once you execute the above code you should get the results similar to those shown in Figure 3.2.

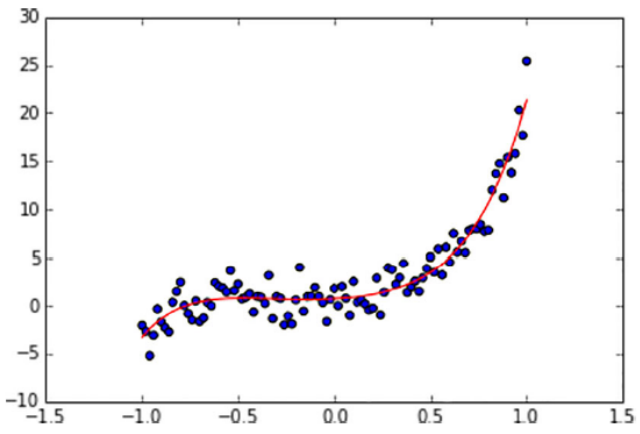

Figure 6: Polynomial regression solution

4 Classification

4.1 Logistic regression

Let us first consider the logistic regression. Recall that the logistic regression function has the shape shown in Figure 4.1.

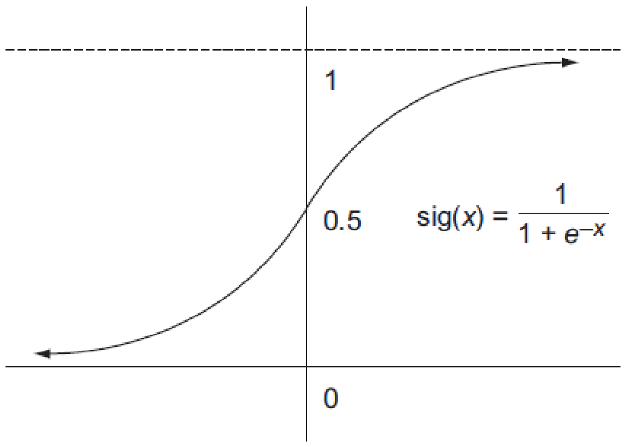

Figure 7: logistic regression

4.1.1 The one-dimensional classification solved using the logistic regression

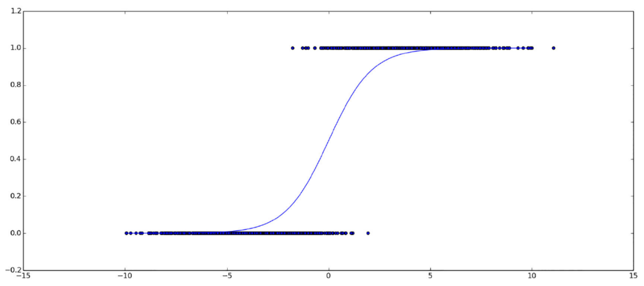

Figure 8: Data for one-dimensional logistic regression

```
# Imports relevant libraries
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

# Sets the hyperparameters
learning_rate = 0.01
training_epochs = 1000

# Defines a helper function to calculate the sigmoid function
def sigmoid(x):
  return 1. / (1. + np.exp(-x))

# Initializes synthetic data
x1 = np.random.normal(-4, 2, 1000)
x2 = np.random.normal(4, 2, 1000)
xs = np.append(x1, x2)
ys = np.asarray([0.] * len(x1) + [1.] * len(x2))

# Visualizes the data
plt.scatter(xs, ys)
```

```
# Defines the input/output placeholders
X = tf.placeholder(tf.float32, shape=(None,), name="x")
Y = tf.placeholder(tf.float32, shape=(None,), name="y")

# Defines the parameter node
w = tf.Variable([0., 0.], name="parameter", trainable=True)

# Defines the model using TensorFlow's sigmoid function
y_model = tf.sigmoid(w[1] * X + w[0])

# Defines the cross-entropy loss function
cost = tf.reduce_mean(-Y * tf.log(y_model) - (1 - Y) * tf.log(1 - y_model))


# Defines the minimizer to use
train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)

# Opens a session, and defines all variables
with tf.Session() as sess:
  sess.run(tf.global_variables_initializer())

  # Defines a variable to keep track of the previous error
  prev_err = 0

  # Iterates until convergence or until the maximum number of epochs is reached
  for epoch in range(training_epochs):

    # Computes the cost, and updates the learning parameters
    err, _ = sess.run([cost, train_op], {X: xs, Y: ys})
    print(epoch, err)

    # Checks for convergence:
    # if you are changing by < .01% per iteration, you are done.
    if abs(prev_err - err) < 0.0001:
      break

    # Updates the previous error value
    prev_err = err

  # Obtains the learned parameter value
  w_val = sess.run(w, {X: xs, Y: ys})


# Plots the learned sigmoid function
all_xs = np.linspace(-10, 10, 100)
plt.plot(all_xs, sigmoid((all_xs * w_val[1] + w_val[0])))
plt.show()
```

Now, run the above code and observe the results.

4.1.2 The two-dimensional classification solved using the logistic regression

Suppose that you have the input dataset with its solutions shown in Figure 4.1.2.

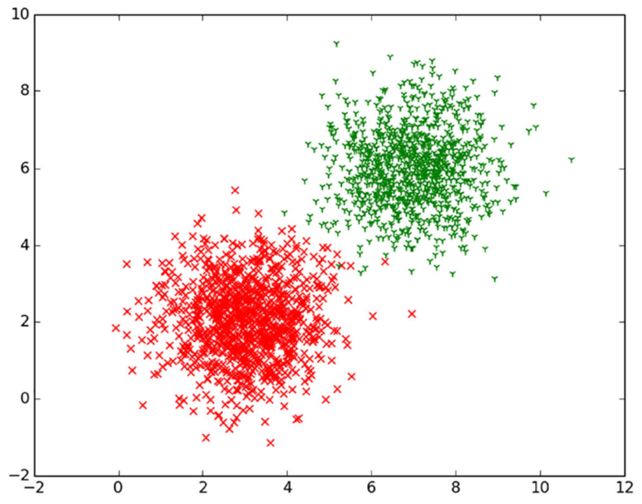

Figure 9: Data for two-dimensional logistic regression

```
# Imports relevant libraries
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

# Sets the hyperparameters
learning_rate = 0.01
training_epochs = 2000

# Defines a helper function to calculate the sigmoid function
def sigmoid(x):
  return 1. / (1. + np.exp(-x))

# Initializes synthetic data
x1_label1 = np.random.normal(3, 1, 1000)
x2_label1 = np.random.normal(2, 1, 1000)
x1_label2 = np.random.normal(7, 1, 1000)
x2_label2 = np.random.normal(6, 1, 1000)
x1s = np.append(x1_label1, x1_label2)
x2s = np.append(x2_label1, x2_label2)
ys = np.asarray([0.] * len(x1_label1) + [1.] * len(x1_label2))

# Visualizes the data
# plt.scatter(xs, ys)

# Defines the input/output placeholders
X1 = tf.placeholder(tf.float32, shape=(None,), name="x1")
X2 = tf.placeholder(tf.float32, shape=(None,), name="x2")
Y = tf.placeholder(tf.float32, shape=(None,), name="y")

# Defines the parameter node
w = tf.Variable([0., 0., 0.], name="w", trainable=True)

# Defines the model using TensorFlow's sigmoid function
y_model = tf.sigmoid(w[2] * X2 + w[1] * X1 + w[0])

# Defines the cross-entropy loss function
cost = tf.reduce_mean(-Y * tf.log(y_model) - (1 - Y) * tf.log(1 - y_model))
# cost = tf.reduce_mean(-tf.log(y_model * Y + (1 - y_model) * (1 - Y)))


# Defines the minimizer to use
train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
```

```
# Opens a session, and defines all variables
with tf.Session() as sess:
  sess.run(tf.global_variables_initializer())

  # Defines a variable to keep track of the previous error
  prev_err = 0

  # Iterates until convergence or until the maximum number of epochs is reached
  for epoch in range(training_epochs):

    # Computes the cost, and updates the learning parameters
    err, _ = sess.run([cost, train_op], {X1: x1s, X2: x2s, Y: ys})
    print(epoch, err)

    # Checks for convergence:
    # if you are changing by < .01% per iteration, you are done.
    if abs(prev_err - err) < 0.0001:
      break

    # Updates the previous error value
    prev_err = err

  # Obtains the learned parameter value
  w_val = sess.run(w, {X1: x1s, X2: x2s, Y: ys})

# Defines arrays to hold boundary points
x1_boundary, x2_boundary = [], []
# Loops through a window of points
for x1_test in np.linspace(0, 10, 100):
  for x2_test in np.linspace(0, 10, 100):
    # If the model response is close the 0.5, updates the boundary points
    z = sigmoid(-x2_test*w_val[2] - x1_test*w_val[1] - w_val[0])
    if abs(z - 0.5) < 0.01:
      x1_boundary.append(x1_test)
      x2_boundary.append(x2_test)

# Shows the boundary line along with the data
plt.scatter(x1_boundary, x2_boundary, c='b', marker='o', s=20)
plt.scatter(x1_label1, x2_label1, c='r', marker='x', s=20)
plt.scatter(x1_label2, x2_label2, c='g', marker='1', s=20)
```

Now, run the above code and observe the results.

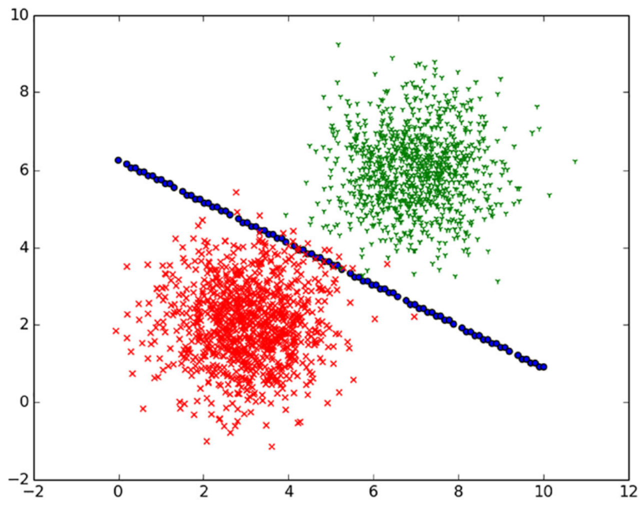

Figure 10: Two-dimensional logistic regression with boundary

5 Keras

Keras is a high-level library that allows the use of TensorFlow as a backend deep learning library. TensorFlow team has included Keras in TensorFlow Core as module **tf.keras**.

5.1 Installation

Keras can be installed in Python 3 with the following command:

```
pip3 install keras
```

5.2 Neural network models in Keras

Neural network models in Keras are defined as the graph of layers. The models in Keras can be created using the sequential or the functional APIs. Both the sequential and functional APIs can be used to build any kind of models. The functional API makes it easier to build the complex models that have multiple inputs, multiple outputs and shared layers. The sequential APIs are used for simple models built from simple layers.

5.3 Workflow for building models in Keras

The simple workflow in Keras is as follows

- Create the model
- Create and add layers to the model
- Compile the model
- Train the model
- User the model for prediction or evaluation

5.4 Creating the Keras model

The Keras model can be created usign the sequential API or functional API. The examples of creating models in both ways are given in the following sections.

5.4.1 Sequential API for creating the Keras model

In the sequential API, create the empty model with the following code:

```
model=Sequential()
```

You can now add the layers to this model, which we will see in the next section.

Alternatively, you can also pass all the layers as a list to the constructor. As an example, we add four layers by passing them to the constructor using the following code:

```
model = Sequential([Dense(10, input_shape=(256,)),
                    Activation('tanh'),
                    Dense(10),
                    Activation('softmax')
```

```
                ])
```

5.4.2 Functional API for creating the Keras model

In the functional API, you create the model as an instance of the **Model** class that takes an input and output parameter. The input and output parameters represent one or more input and output tensors, respectively.
As an example, use the following code to instantiate a model from the functional API:

```
model = Model(inputs=tensor1, outputs=tensor2)
```

If there are more than one input and output tensors, they can be passed as a list:

```
model = Model(inputs=[i1,i2,i3], outputs=[o1,o2,o3]
```

5.5 Keras Layers

Keras provides several built-in layer classes for the easy construction of the network architecture.

5.5.1 Keras core layers

The Keras core layers implement fundamental operations that are used in almost every kind of network architecture. These are: Dense, Activation, Dropout, Flatten, Reshape, Permute, RepeatVector, Lambda, ActivityRegularization, Masking.

5.5.2 Keras convolutional layers

These layers implement the different type of convolution, sampling and cropping operations for convolutional neural networks: Conv1D, Conv2D, SeparableConv2D, Conv2DTranspose, Conv3D, Cropping1D, Cropping2D, Cropping3D, UpSampling1D, UpSampling2D, UpSampling3D, ZeroPadding1D, ZeroPadding2D, ZeroPadding3D.

5.5.3 Keras pooling layers

These layers implement the different pooling operations for convlolutional neural networks: MaxPooling1D, MaxPooling2D, MaxPooling3D, AveragePooling1D, AveragePooling2D, AveragePooling3D, GlobalMaxPooling1D, GlobalAverageMaxPooling1D, GlobalMaxPooling2D, GlobalAverageMaxPooling2D.

5.5.4 Keras locally-connected layers

These layers are useful in convolutional neural networks: LocallyConnected1D,LocallyConnected2D

5.5.5 Keras recurrent layers

These layers implement different variants of recurrent neural networks: SimpleRNN, GRU, LSTM.

5.5.6 Keras embedding layers

Embedding layers

5.5.7 Keras merge layers

These layers merge two or more input tensors and produce a single output tensor by applying a specific oepration taht each layer represents: Add, Multiply, Average, Maximum, Concatenate, Dot.

5.5.8 Keras advanced activation layers

These layers implement advanced activation functions that cannot be implemented as a simple underlying backend function. They operate similarly to the **Activation()** layer (seen previously): LeakyReLU, PReLU, ELU, ThresholdedReLU.

5.5.9 Keras normalization layers

BatchNormalization.

5.5.10 Keras noise layers

These layers can be added to the model to prevent overfitting by adding noise; they are also known as regularization layers. These layers operate the same way as the **Dropout()** and **ActivityRegularizer()** layers in the core layers: GaussianNoise, GaussianDroptout, AlphaDropout.

5.6 Adding layers to the Keras model

All the layers that we just listed need to be added to the model we created earlier.

5.6.1 Sequential API to add layers to the Keras model

In the sequential API, you can create layers by instantiating an object of one of the layer types given previously. The created layers are then added to the model using the **model.add()** function. As an example, we will create a model and then add two layers to it:

```
model = Sequential()
model.add(Dense(10, input_shape=(256,))
model.add(Activation('tanh'))
model.add(Dense(10))
model.add(Activation('softmax'))
```

5.6.2 Functional API to add layers to the Keras model

In the functional API, the layers are created first in a functional manner, and then while creating the model, the input and output layers are provided as tensor arguments.
Here is an example:
1. First, create the input layer:

```
input = Input(shape=(64,))
```

2. Next, create the dense layer from the input layer in a functional way:

```
hidden = Dense(10)(inputs)
```

3. In the same way, create further hidden layers building from the previous layers in a functional way:

```
hidden = Activation('tanh')(hidden)
hidden = Dense(10)(hidden)
output = Activation('tanh')(hidden)
```

4. Finally, instantiate the model object with the input and output layers:

```
model = Model(inputs=input, outputs=output)
```

5.7 Compiling the Keras model

The model built previously needs to be compiled with the **model.compile()** method before it can be used for training and prediction. The full signature of the *compile()* method is as follows:

```
compile(self, optimizer, loss, metrics=None, sample_weight_mode=None)
```

The compile method takes three arguments:

- **optimizer**: You can specify your own function or one of the functions provided by Keras. This function is used to update the parameters in the optimization iterations. Keras offers the following built-in optimizer functions:
  - SGD
  - RMSprop
  - Adagrad
  - Adadelta
  - Adam
  - Adamax
  - Nadam
- **loss**: You can specify your own loss function or use one of the provided loss functions. The optimizer function optimizes the parameters so that the output of this loss function is minimized. Keras provides the following loss functions:
  - mean_squared_error
  - mean_absolute_error
  - mean_absolute_pecentage_error
  - mean_squared_logarithmic_error
  - squared_hinge
  - hinge
  - categorical_hinge
  - sparse_categorical_crossentropy
  - binary_crossentropy
  - poisson
  - cosine_proximity

  - binary_accuracy
  - categorical_accuracy
  - sparse_categorical_accuracy
  - top_k_categorical_accuracy
  - sparse_top_k_categorical_accuracy

- **metrics**: The third argument is a list of metrics that need to be collected while training the model. If verbose output is on, then the metrics are printed for each iteration. The metrics are like loss functions; some are provided by Keras with the ability to write your own metrics functions. All the loss functions also work as the metric function.

5.8 Training the Keras model

Training a Keras model is as simple as calling the **model.fit()** method. The full signature of this method is as follows:

```
fit(self, x, y, batch_size=32, epochs=10, verbose=1, callbacks=None,
  validation_split=0.0, validation_data=None, shuffle=True,
  class_weight=None, sample_weight=None, initial_epoch=0)
```

For the example model that we created earlier, train the model with the following code:

```
model.fit(x_data, y_labels)
```

5.9 Predicting with the Keras model

The trained model can be used either to predict the value with the **model.predict()** method or to evaluate the model with the **model.evaluate()** method.
The signatures of both the methods are as follows:

```
predict(self, x, batch_size=32, verbose=0)

evaluate(self, x, y, batch_size=32, verbose=1, sample_weight=None)
```

5.10 Additional modules in Keras

Keras provides several additional modules that supplement the basic workflow with additional functionalities. Some of the modules are as follows:

- The **preprocessing** module provides several functions for the preprocessing of sequence, image, and text data.

- The **datasets** module provides several functions for quick access to several popular datasets, such as CIFAR10 images, CIFAR100 images, IMDB movie reviews, Reuters newswire topics, MNIST handwritten digits, and Boston housing prices.

- The **initializers** module provides several functions to set initial random weight parameters of layers, such as **Zeros, Ones, Constant, RandomNormal, RandomUniform, TruncatedNormal, VarianceScaling, Orthogonal, Identity, lecun_normal, lecun_uniform, glorot_normal, glorot_uniform, he_normal, and he_uniform**.

- The **models** module provides several functions to restore the model architectures and weights, such as **model_from_json**, **model_from_yaml**, and **load_model**. The model architectures can be saved using the **model.to_yaml()** and **model.to_json()** methods. The model weights can be saved by calling the **model.save()** method. The weights get saved in an HDF5 file.
- The applications module provides several pre-built and pre-trained models such as Xception, VGG16, VGG19, ResNet50, Inception V3, InceptionResNet V2, and MobileNet. We shall learn how to use the pre-built models to predict with our datasets. We shall also learn how to retrain the pre-trained models in the **applications** module with our datasets from a slightly different domain.

5.11 Keras sequential model example for MNIST dataset

It is an example to classify handwritten digits from the MNIST set:

```
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.optimizers import SGD
from keras import utils
import numpy as np

# define some hyper parameters
batch_size = 100
n_inputs = 784
n_classes = 10
n_epochs = 10

# get the data
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# reshape the two dimensional 28 x 28 pixels
# sized images into a single vector of 784 pixels
x_train = x_train.reshape(60000, n_inputs)
x_test = x_test.reshape(10000, n_inputs)

# convert the input values to float32
x_train = x_train.astype(np.float32)
x_test = x_test.astype(np.float32)

# normalize the values of image vectors to fit under 1
x_train /= 255
x_test /= 255

# convert output data into one hot encoded format
y_train = utils.to_categorical(y_train, n_classes)
y_test = utils.to_categorical(y_test, n_classes)

# build a sequential model
model = Sequential()
# the first layer has to specify the dimensions of the input vector
model.add(Dense(units=128, activation='sigmoid', input_shape=(n_inputs,)))
# add dropout layer for preventing overfitting
model.add(Dropout(0.1))
model.add(Dense(units=128, activation='sigmoid'))
model.add(Dropout(0.1))
# output layer can only have the neurons equal to the number of outputs
model.add(Dense(units=n_classes, activation='softmax'))

# print the summary of our model
model.summary()

# compile the model
model.compile(loss='categorical_crossentropy',
  optimizer=SGD(),
  metrics=['accuracy'])

# train the model
model.fit(x_train, y_train,
  batch_size=batch_size,
  epochs=n_epochs)

# evaluate the model and print the accuracy score
scores = model.evaluate(x_test, y_test)

print('\n loss:', scores[0])
print('\n accuracy:', scores[1])
```

Observe the output from running the above code.

6 Reinforcement learning

Recall the Q-learning algorithm that you saw in a previous lecture:

$$Q^{(itera)}(s,a) \leftarrow Q^{(itera-1)}(s,a) + \alpha \left(r(s,a) + \gamma \max_{a'} Q^{(itera-1)}(s',a') - Q^{(itera)}(s,a) \right),$$

for all s and a possible values.
And, once a convergence occurred, we can estimate the optimal policy function π^* as

$$\pi^*(s) = \arg\max_{a} Q(s,a)),$$

for all s possible values.
We can approximate Q function using neural networks given enough training data.

6.1 Applying approximate Q-learning algorithm to the stock market

In this case, the states can be defined as a vector containing information about the current budget, the current number of stocks, and a recent history of stock prices (the last 200 stock prices). Hence, each state is a 202-dimensional vector.

For simplicity, there are only three actions: buy, sell, and hold:

- Buying a stock at the current stock price decreases the budget while incrementing the current stock count.
- Selling a stock trades it in for money at the current share price.
- Holding does neither. This action waits a single time period and yields no reward.

The goal is to learn a policy that gains the maximum net worth from trading in a stock market.

To gather stock prices, you will use the **yahoo_finance** library in Python. You can install it using **pip** or follow the official guide (https://pypi.python.org/pypi/yahoo-finance). The command to install it using pip is as follows:

```
pip install yahoo-finance
```

Once this library is installed, then we can start to use it.

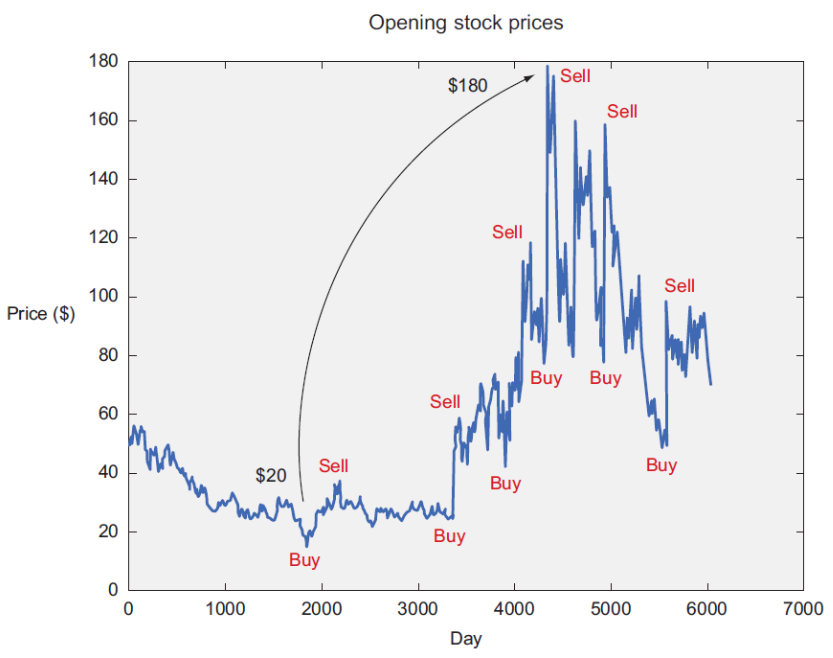

Figure 11: Stock prices and desired action plan

```
from yahoo_finance import Share
from matplotlib import pyplot as plt
import numpy as np
import tensorflow as tf
import random

# define a helper function to get prices
# by specifying the share symbol, start date and end date.
def get_prices(share_symbol, start_date, end_date,
               cache_filename='stock_prices.npy'):
  try:
    # Tries to load the data from file if it has already been computed
    stock_prices = np.load(cache_filename)
  except IOError:
    # Retrieves stock prices from the library
    share = Share(share_symbol)
    # Extracts only relevant info from the raw data
    stock_hist = share.get_historical(start_date, end_date)
    stock_prices = [stock_price['Open'] for stock_price in stock_hist]
    # Caches the result
    np.save(cache_filename, stock_prices)

  return stock_prices.astype(float)

# visualize the stock-price data
def plot_prices(prices):
plt.title('Opening stock prices')
plt.xlabel('day')
plt.ylabel('price ($)')
plt.plot(prices)
plt.savefig('prices.png')
plt.show()

# Defining a superclass for all decision policies
class DecisionPolicy:
  def select_action(self, current_state):
    pass
  def update_q(self, state, action, reward, next_state):
    pass

# Implementing a random decision policy
# Inherits from DecisionPolicy to implement its functions
class RandomDecisionPolicy(DecisionPolicy):
  def __init__(self, actions):
    self.actions = actions

  # Randomly chooses the next action
  def select_action(self, current_state):
    action = random.choice(self.actions)
    return action

# Implementing a more intelligent decision policy
class QLearningDecisionPolicy(DecisionPolicy):
  def __init__(self, actions, input_dim):
    # Sets the hyperparameters from the Q-function
    self.epsilon = 0.95
    self.gamma = 0.3
    self.actions = actions
    output_dim = len(actions)
    # Sets the number of hidden nodes in the neural networks
    h1_dim = 20

    # Defines the input and output tensors
    self.x = tf.placeholder(tf.float32, [None, input_dim])
    self.y = tf.placeholder(tf.float32, [output_dim])

    # Designs the neural network architecture
    W1 = tf.Variable(tf.random_normal([input_dim, h1_dim]))
    b1 = tf.Variable(tf.constant(0.1, shape=[h1_dim]))
    h1 = tf.nn.relu(tf.matmul(self.x, W1) + b1)
    W2 = tf.Variable(tf.random_normal([h1_dim, output_dim]))
    b2 = tf.Variable(tf.constant(0.1, shape=[output_dim]))

    # Defines the op to compute the utility
    self.q = tf.nn.relu(tf.matmul(h1, W2) + b2)

    # Sets theloss as the square error
    loss = tf.square(self.y - self.q)

    # Uses an optimizer to update model parameters to minimize the loss
    self.train_op = tf.train.AdagradOptimizer(0.01).minimize(loss)

    # Sets up the session, and initializes variables
    self.sess = tf.Session()
    self.sess.run(tf.global_variables_initializer())

  def select_action(self, current_state, step):
    threshold = min(self.epsilon, step / 1000.)

    # Exploits the best option with probability epsilon
    if random.random() < threshold:
      # Exploit best option with probability epsilon
      action_q_vals = self.sess.run(self.q, feed_dict={self.x: current_state})
      action_idx = np.argmax(action_q_vals)
      action = self.actions[action_idx]

    # Explores a random option with probability 1 - epsilon
    else:
      # Explore random option with probability 1 - epsilon
      action = self.actions[random.randint(0, len(self.actions) - 1)]
    return action

  # Updates the Q-function by updating its model parameters
  def update_q(self, state, action, reward, next_state):
    action_q_vals = self.sess.run(self.q, feed_dict={self.x: state})
    next_action_q_vals = self.sess.run(self.q, feed_dict={self.x: next_state})
    next_action_idx = np.argmax(next_action_q_vals)
    current_action_idx = self.actions.index(action)
    action_q_vals[0, current_action_idx] = reward + self.gamma *
      next_action_q_vals[0, next_action_idx]
    action_q_vals = np.squeeze(np.asarray(action_q_vals))
    self.sess.run(self.train_op, feed_dict={self.x: state, self.y: action_q_vals})

# Using a given policy to make decisions, and returning the performance
def run_simulation(policy, initial_budget, initial_num_stocks, prices, hist):
  # Initializes values that depend on computing the net worth of a portfolio
  budget = initial_budget
  num_stocks = initial_num_stocks
  share_value = 0
  transitions = list()

  for i in range(len(prices) - hist - 1):
    if i % 1000 == 0:
      print('progress {:.2f}%'.format(float(100*i) / (len(prices) - hist - 1)))
    # The state is a hist + 2 dimensional vector.
    # You'll force it to be a NumPy matrix.
    current_state = np.asmatrix(np.hstack((prices[i:i+hist], budget, num_stocks)))
    # Calculates the portfolio value
    current_portfolio = budget + num_stocks * share_value
    # Selects an action from the current policy
    action = policy.select_action(current_state, i)
    share_value = float(prices[i + hist])

    # Updates portfolio values
    if action == 'Buy' and budget >= share_value:
      budget -= share_value
      num_stocks += 1
    elif action == 'Sell' and num_stocks > 0:
      budget += share_value
      num_stocks -= 1
    else:
      action = 'Hold'

    # Computes a new portfolio value after taking action
    new_portfolio = budget + num_stocks * share_value

    # Computes the reward from taking an action at a state
    reward = new_portfolio - current_portfolio
    next_state = np.asmatrix(np.hstack((prices[i+1:i+hist+1], budget, num_stocks)))
    transitions.append((current_state, action, reward, next_state))
    # Updates the policy after experiencing
    policy.update_q(current_state, action, reward, next_state)

  # Computes the final portfolio worth
  portfolio = budget + num_stocks * share_value
  return portfolio


# Running multiple simulations to calculate an average performance
def run_simulations(policy, budget, num_stocks, prices, hist):
  # Decides the number of times to rerun the simulations
  num_tries = 10
  # Stores the portfolio worth of each run in this array
  final_portfolios = list()
  for i in range(num_tries):
    # Runs this simulation
    final_portfolio = run_simulation(policy, budget, num_stocks, prices, hist)
    final_portfolios.append(final_portfolio)
    print('Final portfolio: ${}'.format(final_portfolio))

plt.title('Final Portfolio Value')
plt.xlabel('Simulation #')
plt.ylabel('Net worth')
plt.plot(final_portfolios)
plt.show()

# the main function
if __name__ == '__main__':
  # You can grab some data and visualize it
  prices = get_prices('MSFT', '1992-07-22', '2016-07-22')
  plot_prices(prices)

  # Defines the list of actions the agent can take
  actions = ['Buy', 'Sell', 'Hold']
  hist = 3

  # Initializes a random decision policy
  policy = RandomDecisionPolicy(actions)
  # policy = QLearningDecisionPolicy(actions)
  # Sets the initial amount of money available to use
  budget = 100000.0

  # Sets the number of stocks already owned
  num_stocks = 0

  # Runs simulations multiple times
  # to compute the expected value of your final net worth
  run_simulations(policy, budget, num_stocks, prices, hist)
```

Once you run the above code, you may results similar to those shown in Figure 6.1.

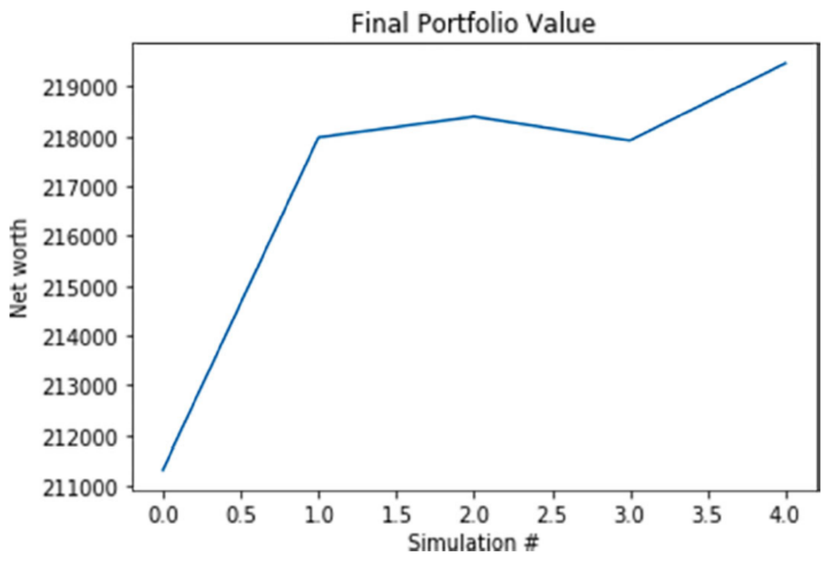

Figure 12: Final portfolio value