Introduction to Tensorflow

Song Jiaming

Contents

1 Background

TensorFlow is an open source software library for numerical computation using data flow graphs. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them. The flexible architecture allows you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API.

TensorFlow was originally developed by researchers and engineers working on the Google Brain Team within Google's Machine Intelligence research organization for the purposes of conducting machine learning and deep neural networks[1] research, but the system is general enough to be applicable in a wide variety of other domains as well.

2 Installation

This guide help install tensorflow through pip. For other ways or more information, please visit Tensorflow Installation page

2.1 For Mac OS

Your system must contain one of the following Python versions: Python 2.7 or 3.3+. The newest version is Python 3.6.2

`pip` should be already installed with python: `pip` for Python 2.7, `pip3` for Python 3.n. Issue the following command to check if you have installed pip correctly:

```
$ pip -V # for Python 2.7
$ pip3 -V # for Python 3.n
```

To upgrade the versions:

```
$ sudo easy_install --upgrade pip
$ sudo easy_install --upgrade six
```

To install Tensor Flow, invoke one of the following commands:

```
$ pip install tensorflow # for Python 2.7
$ pip3 install tensorflow # for Python 3.n
```

2.2 For Windows

This guide is to install *TensorFlow with CPU support only* version. If your system have a NVIDIA GPU and you wish to install *TensorFlow with GPU support* version, please visit Installing Tensorflow on Windows for more details.
You should have the following version of Python installed on your machine:

- Python 3.5 x 64-bit

Python 3.5.x comes with pip3 package manager, which is the program to install TensorFlow.
After you have correctly installed Python 3.5.x, start 'Command Prompt', move to the directory where you store your python (cd 'filepath'), then enter the following command:

```
pip3 install --upgrade tensorflow
```

2.3 Validate your installation: Both Mac and Windows

Open your python shell and enter the folloing program:

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

If TensorFlow is installed successfully, you should be able to see the following:

```
Hello, TensorFlow!
```

If you see *b'Hello, TensorFlow!'*, try the following command instead:

```
>>> print(sess.run(hello).decode())
```

[1] A computer system modelled on the human brain and nervous system

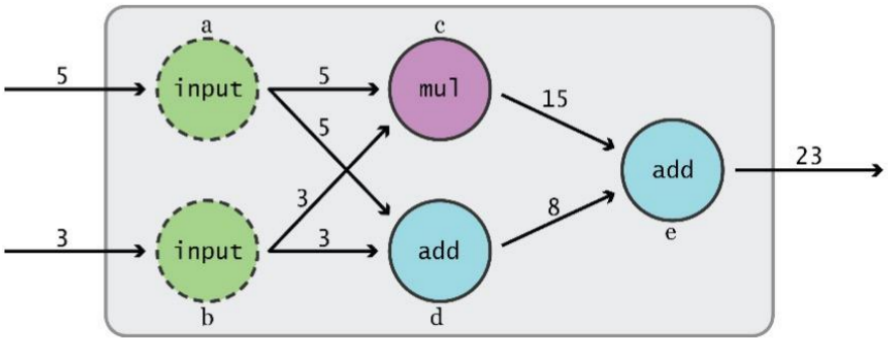
3 Basics

To use TensorFlow package

```
import tensorflow as tf
```

3.1 Data Flow Graphs

TensorFlow separates definition of computations from their execution.



Phase 1: assemble a graph
Phase 2: use a session to execute operations in the graph.

What's a tensor?

An n-dimensional matrix:

- 0-d tensor: scalar (number)
- 1-d tensor: vector
- 2-d tensor: matrix
- ⋮

3.2 Assign values

3.2.1 Constants

Assign constant values to a tensor:

```
tf.constant(value, dtype=None, shape=None,name='Const', verify_shape=False)
```

where,

- `value` : can be any dimension
- `dtype` : type of the value
- `shape` : the dimension of the value you input
- `name` : name for visualisation
- `verify_shape` : verify if the value and shape you provided are consistent, error message will be shown if they are not. If this parameter is turned false, but the value and shape is inconsistent, tf.constant will repeat the last digit of the value you input.

Use `print(sth)` to see the information of the Tensor but use `sth.eval()` to see the exact value of the Tensor.

Note: To execute `sth.eval()`, either `tf.InteractiveSession()` or `with tf.Session() as sess:` is needed. This session will be explained in the later part of the note.
Examples:

```
a = tf.constant(2,shape=[2,2], verify_shape = True)
# output: error
a = tf.constant(3,shape=[2,2], verify_shape = False) # or just remove verify_shape
# print(a) => Tensor("Const:0", shape=(2, 2), dtype=int32)
# a.eval() => array([[3, 3],[3, 3]])
```

```
b = tf.constant([2,1],shape=[3,3],name="b")
# print(b) => Tensor("b:0", shape=(3, 3), dtype=int32)
# b.eval() => array([[2, 1, 1],[1, 1, 1],[1, 1, 1]])
c = tf.constant([[0,1],[2,3],[3,4]])
```

To interpret Tensor("Const:0", shape=(2, 2), dtype=int32):

- "const" is the default name given when there is no input for 'name' parameter. Sometimes you will see "const_1", "const_2", etc, this means that you have constructed multiple "const" before.
- 0: that's the position of this value in the tensor. An operation with n tensor outputs name these tensors "op_name:0", "op_name:1", ... "op_name:n-1". Since the output here is just one tensor, it will be positioned at the start: 0.

Try out the following codes :)

```
a = tf.constant([2,2],name="a")
b = tf.constant([[0,1],[2,3]], name = "b")
x = tf.add(a, b, name="add") #a is added to both rows of b
y = tf.multiply(a, b, name="mul") # a is mul picewisely to both rows of b
with tf.Session() as sess:
    x, y = sess.run([x,y])
    print(x)
    print(y)
```

3.2.2 Zeros Tensor

Which gives all 0s

```
tf.zeros(shape,dtype=tf.float32, name=None)
#For e.g.
c = tf.zeros([2,3],tf.int32)
# [0,0,0][0,0,0]
```

If you have an input_tensor, but you want a zero tensor which has the same shape of that tensor, use:

```
tf.zeros_like(input_tensor,dtype=None, name=None, opimize=True)
# For e.g. (Pesudo code)
input_tensor is [[0,1],[2,3],[4,5]]
tf.zeros_like(input_tensor) >>> [[0,0],[0,0],[0,0]]
```

3.2.3 Ones Tensor

Which gives all ones. It is similar to zeros tensor,

```
tf.ones(shape,dtype=tf.float32, name=None)
tf.ones_like(input_tensor,dtype=None, name=None, opimize=True)
```

3.2.4 Fill with a specific value

The tensor will have all elements equal to the defined value.

```
tf.fill(dims,value,name=None) # "dims" -> dimensions
# e.g.
tf.fill([2,3],8) >>> [[8,8,8],[8,8,8]]
```

3.2.5 Constants as sequences

Getting a list of sequence:

- tf.linspace: A sequence of 'num' evenly-spaced values are generated beginning at 'start'. If num $>$ 1, the values in the sequence increase by (stop - start)/(num - 1), so that the last one is exactly 'stop'.

  ```
  tf.linspace(start, stop, num, name=None)
  # e.g.
  tf.linspace(10.0, 13.0, 4) #==> [10.0 11.0 12.0 13.0]
  ```

- tf.range : Creates a sequence of numbers that begins at 'start' and extends by increments of 'delta' up to but not including 'limit'.

```
tf.range(start, limit=None, delta=1, dtype=None, name='range')
# e.g
# 'start' is 3, 'limit' is 18, 'delta' is 3
tf.range(3, 18, 3) #==> [3, 6, 9, 12, 15]
# 'limit' is 5
tf.range(5) #==> [0, 1, 2, 3, 4]
```

3.2.6 Randomly generated constants

```
tf.random_normal(shape,mean=0.0,stddev=1.0,dtype=tf.float32,seed=None,name=None)
tf.truncated_normal(shape,mean=0.0,stddev=1.0,dtype=tf.float32,seed=None,
name=None) # similar to normal, but value is denser
tf.random_uniform(shape,minval=0,maxval=None,dtype=tf.float32,seed=None,
name=None)
tf.random_shuffle(value,seed=None,name=None) # always shuffle on the 1st dimension
tf.random_crop(value,size,seed=None,name=None) #randomly crop a shape size
tf.multinomial(logits,num_samples,seed=None,name=None)
tf.random_gamma(shape,alpha,beta=None,dtype=tf.float32,seed=None,name=None)
tf.set_random_seed(seed)  # values won't be deterministic
```

3.3 Operations

Category	Examples
Element-wise math ops	add, subtract, multiply, div, log, greater, less, equal,...
Array operations	concat, slice, split, constant, rank, shape, shuffle,...
Matrix operations	matmul,matrixInverse, matrixDeterminant,...
Stateful operations	variable, assign, assignAdd,...
Neural network building blocks	softmax, sigmoid, relu, convolution2D, maxpool,...
Checkpointing operations	save, restore
Queue and synchronization operations	enqueue, dequeue, mutexacquire, mutexrelease
Control flow operations	merge, switch, enter, leave, nextiteration

Try the following examples:

```
a = tf.constant([3, 6])
b = tf.constant([2, 2])
tf.add(a, b) # >> [5 8]
tf.add_n([a, b, b]) # >> [7 10]. Equivalent to a + b + b
tf.multiply(a, b) # >> [6 12] because mul is element wise
tf.matmul(a, b) # >> ValueError
tf.matmul(tf.reshape(a, [1, 2]), tf.reshape(b, [2, 1])) # >> [[18]]
tf.div(a, b) # >> [1 3]
tf.mod(a, b) # >> [1 0]
```

3.4 Ranks, Shapes, Data Types

In Tensorflow system, tensors are described by their rank(aka degree, order), shape (dimension) and data types.

- 0-d tensor, or "scalar", (Rank = 0, shape = [])

```
t_0 = 19
tf.zeros_like(t_0) # ==> 0
tf.ones_like(t_0) # ==> 1
```

- 1-d tensor, or "vector", (Rank = 1, shape = [1xn])

```
t_1 = ['apple', 'peach', 'banana']
tf.zeros_like(t_1) # ==> ['' '' '']
tf.ones_like(t_1) # ==> TypeError: Expected string, got 1 of type 'int' instead
    .
```

- 2x2 tensor, or "matrix", (Rank = 2, shape = [nxn])

```
t_2 = [[True, False, False],
[False, False, True],
[False, True, False]]
tf.zeros_like(t_2) # ==> 2x2 tensor, all elements are False
tf.ones_like(t_2) # ==> 2x2 tensor, all elements are True
```

Figure 1: Tensorflow data types

Data type	Python type	Description
DT_FLOAT	tf.float32	32 bits floating point.
DT_DOUBLE	tf.float64	64 bits floating point.
DT_INT8	tf.int8	8 bits signed integer.
DT_INT16	tf.int16	16 bits signed integer.
DT_INT32	tf.int32	32 bits signed integer.
DT_INT64	tf.int64	64 bits signed integer.
DT_UINT8	tf.uint8	8 bits unsigned integer.
DT_UINT16	tf.uint16	16 bits unsigned integer.
DT_STRING	tf.string	Variable length byte arrays. Each element of a Tensor is a byte array.
DT_BOOL	tf.bool	Boolean.
DT_COMPLEX64	tf.complex64	Complex number made of two 32 bits floating points: real and imaginary parts.
DT_COMPLEX128	tf.complex128	Complex number made of two 64 bits floating points: real and imaginary parts.
DT_QINT8	tf.qint8	8 bits signed integer used in quantized Ops.
DT_QINT32	tf.qint32	32 bits signed integer used in quantized Ops.
DT_QUINT8	tf.quint8	8 bits unsigned integer used in quantized Ops.

Remark. *TensorFlow takes Python natives types: boolean, numeric(int, float), strings. However, do not use Python naive types because TensorFlow has to infer Python type*

3.5 Variables

Though constants are easier to define, there is something wrong with constants, other than being constant. That is, they are stored in the graph definition, so if you are to print out the graph definition:

```
import tensorflow as tf
my_const = tf.constant([1.0, 2.0], name="my_const")
with tf.Session() as sess:
  print(sess.graph.as_graph_def())
# you will see value of my_const stored in the graph s definition
```

This makes loading graph expensive when constants are big.
Therefore, we only use constants for primitive type and use **variables** or readers for data that requires more memory.

```
# create variable a with scalar value
a = tf.Variable(2, name="scalar")

# create variable b as a vector
b = tf.Variable([2, 3], name="vector")
```

```
# create variable c as a 2x2 matrix
c = tf.Variable([[0, 1], [2, 3]], name="matrix")

# create variable W as 784 x 10 tensor, filled with zeros
W = tf.Variable(tf.zeros([784,10]))
```

Define the variables, input the value or sizes that you want it to start with.

Remark. *tf.Variable is a class, but tf.constant is an op.*
tf.Variable holds several ops: x.initializer, x.value(), x.assign(...), x.assign_add(...). Note that all ops have to be called by using 'session' so that they can make effect

3.5.1 Initialize your variables

You have to initialize your variables for them to work properly:

- Initialize all variables at once:

```
init = tf.global_variables_initializer()
with tf.Session() as sess:
  sess.run(init) # goes to look at all variables and initialize them
```

- Initialize only a subset of variables:

```
init_ab = tf.variables_initializer([a,b],name="init_ab")
with tf.Session() as sess:
  sess.run(init_ab)
```

- Initialize a single variable:

```
W = tf.Variable(tf.zeros([784,10]))
with tf.Session() as sess:
  sess.run(W.initializer)
```

3.5.2 Get the variable's value

To see the value of a variable: theVar.eval()

```
# let's say W is a random 700*100 variable object
W = tf.Variable(tf.truncated_normal([784,10]))
with tf.Session() as sess:
  sess.run(W.initializer)
  print(W.eval())
```

3.5.3 Assign values to variable

To give a value for the variable, use assign()

```
W = tf.Variable(10)
W.assign(100)
with tf.Session() as sess:
  sess.run(W.initializer)
  print(W.eval()) # >> 10 but why not 100?
```

W.assign(100) will not assign the value 100 to W unless we run it. The correct way to do that is:

```
W = tf.Variable(10)
assign_op = W.assign(100)
with tf.Session() as sess:
  sess.run(W.initializer)
  sess.run(assign_op)
  print(W.eval()) # >> 100
```

Remark. *Even if the W.initializer is not called at this stage, the code will still output 100, because assign_op does the initialization, too. In fact, initializer op is the assign op that assigns the variables' initial value to the variable itself.*

Try the following code:

```
# create a variable whose original value is 2
my_var = tf.Variable(2, name="my_var")

# assign a * 2 to a and call that op a_times_two
my_var_times_two = my_var.assign(2 * my_var)

with tf.Session() as sess:
  sess.run(my_var.initializer)
  sess.run(my_var_times_two) # >> 4
  sess.run(my_var_times_two) # >> 8
  sess.run(my_var_times_two) # >> 16
```

assign_add() and assign_sub() are related operations

```
my_var = tf.Variable(10)
with tf.Session() as sess:
  sess.run(my_var.initializer)  # >> 10

  # increment by 10
  sess.run(my_var.assign_add(10)) # >> 20

  # decrement by 2
  sess.run(my_var.assign_sub(2)) # >>18
```

Remark. *assign_add() and assign_sub() cannot initialize the variable my_var because these ops need the original value of my_var*

3.5.4 Sessions and variables

Each session maintains it's own copy of variable.

```
W = tf.Variable(10)
sess1 = tf.Session()
sess2 = tf.Session()
sess1.run(W.initializer)
sess2.run(W.initializer)
print(sess1.run(W.assign_add(10))) # >> 20
print(sess2.run(W.assign_sub(2))) # >> 8
sess1.close()
sess2.close()
```

3.5.5 Use a variable to initialize another variable

Let's say we want have a variable W, and we want to have a variable U such that U = 2*W.

```
# let W be a random 700*100 tensor
W = tf.Variable(tf.truncated_normal([700,100]))
U = tf.Variable(2*W)
```

However, using 2*W may not be very safe as we are not sure if W is initialised, thus replaying the last line of code with the following code is better: `U = tf.Variable(2*W.initialized_value())`

3.6 Session vs InteractiveSession

The only difference is that an Interactive Session makes itself the default.

```
sess = tf.InteractiveSession()
a = tf.constant(5.0)
b = tf.constant(6.0)
c = a * b
# We can just use 'c.eval()' without specifying the context 'sess'
print(c.eval())
sess.close()
```

3.7 Control Dependencies

Sometimes we want to make sure some ops have run, before we run some other ops, the way to do it is to use `tf.Graph.control_dependencies(control_inputs)`
For example, your graph g has 5 ops: a, b, c, d, e:

```
with g.control_dependencies([a,b,c]):
  d = ...
  e = ...
  # 'd' and 'e' will only run after 'a','b','c' have executes
```

3.8 Placeholders

Recall that a TF program often has 2 phases: assemble a graph, then use a session to execute operations in the graph
A placeholder is something to allow you assemble the graph first without knowing the values needed for computation. It is in analogy to defining a function: $f(x, y) = x \times 2 + y$ without knowing the value of x or y. In this case, x and y are placeholders for the actual values. Placeholders allow us to later supply our data when they need to execute the computation. i.e. When we write our models to classify images, and the training set has millions of images, we do not want to load all these images first.

```
### format ###
tf.placeholder(dtype, shape=None, name=None)

# create a placeholder of type float 32-bit, shape is a vector of 3 elements
a = tf.placeholder(tf.float32, shape=[3])

# create a constant of type float 32-bit, shape is a vector of 3 elements
b = tf.constant([5, 5, 5], tf.float32)

# use the placeholder as you would a constant or a variable
c = a + b # Short for tf.add(a, b)

with tf.Session() as sess:
  print(sess.run(c)) # Error because a has no value
```

We can feed values to placeholders using a dictionary:

```
a = tf.placeholder(tf.float32, shape=[3])
b = tf.constant([5, 5, 5], tf.float32)
c = a + b # Short for tf.add(a, b)

with tf.Session() as sess:
  # feed [1, 2, 3] to placeholder a via the dict {a: [1, 2, 3]}
  print(sess.run(c, {a: [1, 2, 3]})) # >> [6,7,8]
```

Placeholders are valid ops, and can be put into graphs similarly like putting a constant

Remark. *shape = None means that tensor of any shape will be accepted as value for placeholder. It is easy for constructing graphs, but nightmarish for debugging.*

3.8.1 Feeding multiple data points

What if we want to feed all the values in, one at a time? Use a loop.

```
with tf.Session() as sess:
  for a_value in list_of_values_for_a:
    print(sess.run(c, {a: a_value}))
```

3.8.2 Feeding values to normal tensors

We can feed values to tensors that are not placeholders provided that these tensors are feedable. To check if a tensor is feedable, try:

```
tf.Graph.is_feedable(tensor) # True if and only if tensor is feedable

#e.g.
W = tf.Variable(10)
with tf.Session() as sess:
    print(tf.get_default_graph().is_feedable(b))
```

feed_dict is an useful way to feed data in a dictionary-like way. For example:

```
# create operations, tensors, etc (using the default graph)
a = tf.add(2, 5)
b = tf.multiply(a, 3)

# start up a `Session` using the default graph
sess = tf.Session()

# define a dictionary that says to replace the value of `a` with 15
replace_dict = {a: 15}

# Run the session, passing in `replace_dict` as the value to `feed_dict`
sess.run(b, feed_dict=replace_dict) # returns 45
```

3.8.3 Differences between placeholders and variables

	Placeholder	Variable
Need Initial Value?	No	Yes
Changes over model training?	No	Yes
Used for?	Input data	Weights, biases, etc

3.9 TensorBoard

TensorBoard is tool to help visualise the models, it comes with the TensorFlow package. It is extremely useful to help you see what is going on and debug when your model is complicated. Here is your first TensorFlow program:

```
import tensorflow as tf
a = tf.constant(2)
b = tf.constant(3)
x = tf.add(a,b)
with tf.Session() as sess:
    print(sess.run(x))
```

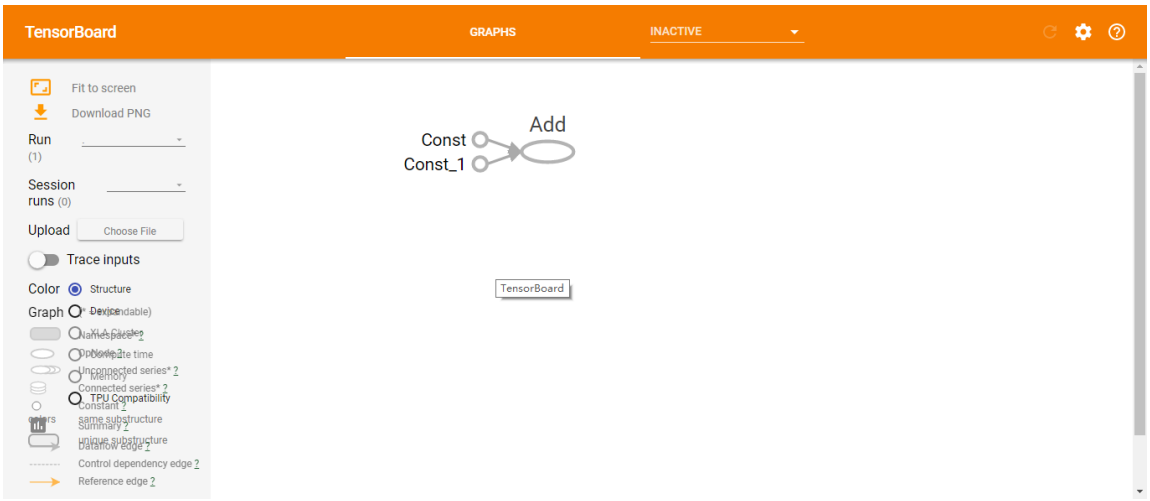
To visualize the above program with Tensor Board:

```
import tensorflow as tf
a = tf.constant(2)
b = tf.constant(3)
x = tf.add(a,b)
with tf.Session() as sess:
  writer = tf.summary.FileWriter('./graphs', sess.graph)
  #'./graphs' is the path of where you want to store your event log
  print(sess.run(x))
```

Then, go to terminal, move to the correct directory, run:

```
$ python [yourprogram].py
$ tensorboard --logdir="./graphs" --port 6006
```

Then open your browser and go to: http://localhost:6006/
You should be able to see the following:



Since the name 'const' is not intuitive, we can change it to a predefined name by:

```
a = tf.constant(2,name="a")
b = tf.constant(3,name="b")
x = tf.add(a,b,name="add")
with tf.Session() as sess:
  writer = tf.summary.FileWriter('./graphs', sess.graph)
  #'./graphs' is the path of where you want to store your event log
  print(sess.run(x))
```

3.10 (Optional)Bad example: Lazy Loading

Defer creating or initializing an object until it is needed An example of normal loading: you create the op z when you assemble the graph

```
x = tf.Variable(10, name='x')
y = tf.Variable(20, name='y')
z = tf.add(x, y) # you create the node for add node before executing the graph
with tf.Session() as sess:
  sess.run(tf.global_variables_initializer())
  writer = tf.summary.FileWriter('./my_graph/l2', sess.graph)
  for _ in range(10):
    sess.run(z)
  writer.close()
```

Then for lazy loading:

```
x = tf.Variable(10, name='x')
y = tf.Variable(20, name='y')

with tf.Session() as sess:
  sess.run(tf.global_variables_initializer())
  writer = tf.summary.FileWriter('./my_graph/l2', sess.graph)
  for _ in range(10):
    sess.run(sess.run(tf.add(x, y))) # just to save one line of the code
  writer.close()
```

Both will create 'add', but what is the problem?
Normal loading just add 'add' op once into the graph, for lazy loading, you create multiple add node.
Solution: Separate definition of ops from computing/running ops and use python property to ensure function is also loaded once at the first time it is called

4 Basic Models

4.1 Linear regression

Linear regression is a model relationship between a scalar dependent variable y and independent variables X. We will use The fire and theft example of Chicago from CS20ST to illustrate. [2]
In this case, X is the number of incidents of fire, Y is the number of incidents of theft. We want to predict Y from X.
Model: $Y_{\text{predicted}}$ = w * X + b (w is the parameter, b is the bias)
s.t. $\min_{w,b}(Y - Y_{\text{predicted}})^2$ (The loss function)

```
# necessary packages
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
import xlrd

# load the data
DATA_FILE = 'fire_theft.xls'

# Step 1: read in data from the .xls file
book = xlrd.open_workbook(DATA_FILE, encoding_override="utf-8")
sheet = book.sheet_by_index(0)
data = np.asarray([sheet.row_values(i) for i in range(1, sheet.nrows)])
n_samples = sheet.nrows - 1

# Step 2: create placeholders for input X (number of fire) and label Y (number of
    theft)
X = tf.placeholder(tf.float32, name='X')
Y = tf.placeholder(tf.float32, name='Y')

# Step 3: create weight and bias, initialized to 0
w = tf.Variable(0.0, name='weights')
b = tf.Variable(0.0, name='bias')

# Step 4: build model to predict Y
Y_predicted = X * w + b

# Step 5: use the square error as the loss function
loss = tf.square(Y - Y_predicted, name='loss')
# loss = utils.huber_loss(Y, Y_predicted)

# Step 6: using gradient descent with learning rate of 0.01 to minimize loss
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001).minimize(loss)

with tf.Session() as sess:
  # Step 7: initialize the necessary variables, in this case, w and b
  sess.run(tf.global_variables_initializer())

  writer = tf.summary.FileWriter('./graphs/linear_reg', sess.graph)
  # use: tensorboard --logdir = "./graphs" to visualise the graph

  # Step 8: train the model
  for i in range(50): # train the model 100 epochs
    total_loss = 0
    for x, y in data:
      # Session runs train_op and fetch values of loss
      _, l = sess.run([optimizer, loss], feed_dict={X: x, Y:y})
      total_loss += l
    print('Epoch {0}: {1}'.format(i, total_loss/n_samples))

  # close the writer when you're done using it
```

[2] All CS20SI's tutorials and solutions can be found here: https://github.com/chiphuyen/stanford-tensorflow-tutorials

```
  writer.close()

  # Step 9: output the values of w and b
  w, b = sess.run([w, b])

# plot the results
X, Y = data.T[0], data.T[1]
plt.plot(X, Y, 'bo', label='Real data')
plt.plot(X, X * w + b, 'r', label='Predicted data')
plt.legend()
plt.show()
```

References

[1] Siraj Raval, *"Intro to Tensorflow"*. https://youtu.be/2FmcHiLCwTU

[2] CS 20SI: Tensorflow for Deep Learning Research, *Standford University*. The course materials are available at https://web.stanford.edu/class/cs20si/syllabus.html

[3] Labhesh Patel, a Youtuber who made videos on CS20SI standford materials. The course is at https://www.youtube.com/watch?v=g-EvyKpZjmQ&list=PLSPPwKHXGS2110rEaNH7amFGmaD5hsObs

[4] Tensorflow official website, https://www.tensorflow.org/