

# First Neurons

```
In [25]: import numpy as np
import tensorflow as tf

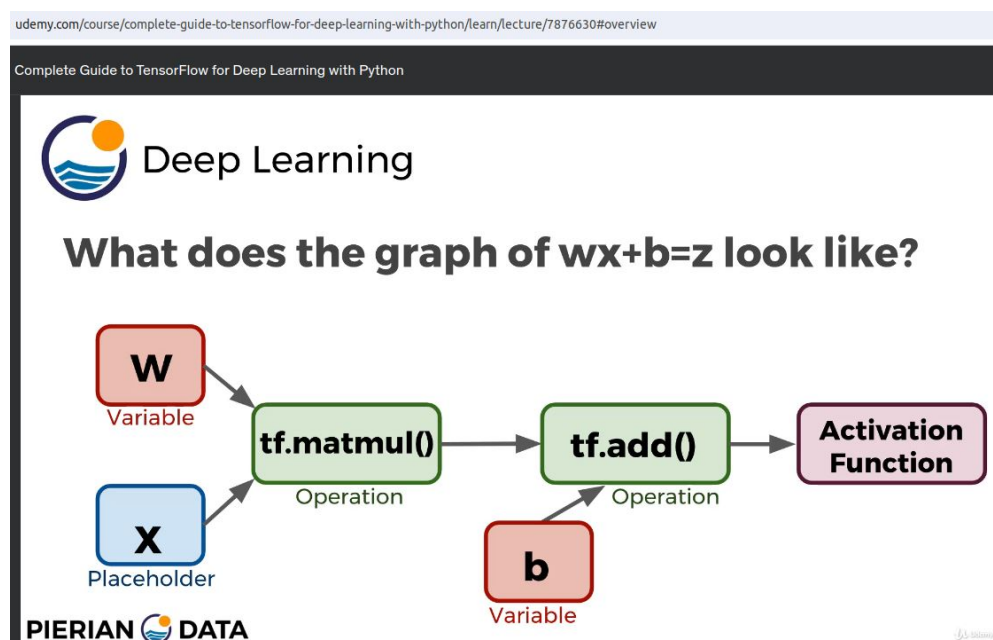
import matplotlib.pyplot as plt
%matplotlib inline
```

## Set Random Seeds for same results

```
In [2]: np.random.seed(101)
tf.set_random_seed(101)
```

## The graph we will be building

Getting a very simple linear fit to some 2d data.



## Data Setup

Setting Up some Random Data for Demonstration Purposes

```
In [3]: rand_a = np.random.uniform(0, 100, (5, 5))
        rand_a
```

```
Out[3]: array([[51.63986277, 57.06675869,  2.84742265, 17.15216562, 68.52769817],
               [83.38968626, 30.69662197, 89.36130797, 72.15438618, 18.99389542],
               [55.42275911, 35.2131954 , 18.18924027, 78.56017619, 96.54832224],
               [23.23536618,  8.35614337, 60.35484223, 72.89927573, 27.62388285],
               [68.53063288, 51.78674742,  4.84845374, 13.78692376, 18.69674261]])
```

```
In [4]: rand_b = np.random.uniform(0, 100, (5, 1))  
rand_b
```

```
Out[4]: array([[99.43179012],  
               [52.06653967],  
               [57.87895355],  
               [73.48190583],  
               [54.19617722]])
```

## Placeholders

```
In [5]: a = tf.placeholder(tf.float32)
```

```
In [6]: b = tf.placeholder(tf.float32) # we're not using the shape parameter for now.
```

## Operations

```
In [7]: add_op = a + b # Could do tf.add(a, b), and [Shift]+[Tab] to look at the docs
```

```
In [8]: mult_op = a * b # Could do tf.multiply(a, b)
```

## Running Sessions to create Graphs with Feed Dictionaries

```
In [9]: with tf.Session() as sess:  
        add_result = sess.run(add_op, feed_dict={a:10, b:20})  
        print(add_result)
```

```
##endof: with tf.Session() as sess:
```

30.0

```
In [10]: with tf.Session() as sess:

    add_result = sess.run(add_op, feed_dict={a:rand_a, b:rand_b})
    print(add_result)

    print('\n')

    mult_result = sess.run(mult_op, feed_dict={a:rand_a, b:rand_b})
    print(mult_result)

##endof: with tf.Session as sess
```

```
[ [151.07166  156.49855  102.27921  116.58396  167.95949 ]
  [135.45622   82.76316  141.42784  124.22093   71.06043 ]
  [113.30171   93.09215   76.06819  136.43912  154.42728 ]
  [ 96.71727   81.83804  133.83675  146.38118  101.10579 ]
  [122.72681  105.982925  59.044632  67.9831   72.89292 ] ]
```

```
[ [5134.644  5674.25    283.12433 1705.4707  6813.8315 ]
  [4341.8125 1598.267   4652.734  3756.8293  988.94635]
  [3207.8113 2038.1029  1052.7742  4546.9805  5588.1157 ]
  [1707.379   614.02527 4434.989   5356.7773  2029.8555 ]
  [3714.0984 2806.6438   262.76764  747.19855 1013.292  ] ]
```

Multiplication was element-by-element rather than a dot multiply.

## Example Neural Network

```
In [11]: n_features = 10
         n_dense_neurons = 3
```

```
In [12]: # Placeholder for x  
x = tf.placeholder(tf.float32,) # shape is defined, though the 'None' is  
    ## because it depends on how big of a  
    ## batch of data you're feeding into your NN  
    ## features. We should be expecting 'x'  
    ## to be receiving an array with n_samples  
    ## by n_features
```

```
In [13]: W = tf.Variable(tf.random_normal([n_features, n_dense_neurons]))  
  
b = tf.Variable(tf.ones([n_dense_neurons])) # dimension must allow  
    # matrix multiplication
```

### Operation Activation Function

```
In [14]: xW = tf.matmul(x, W) # It will do the dot multiplication, now.
```

```
In [15]: z = tf.add(xW, b)
```

```
In [16]: a = tf.sigmoid(z)
```

### Variable Initializer!

```
In [17]: init = tf.global_variables_initializer()
```

```
In [18]: with tf.Session() as sess:  
    sess.run(init) # do this for a session  
  
    layer_out = sess.run(a, feed_dict={x:np.random.random([1, n_features])})  
    # feeding it one sample  
  
    ##endof: with ... sess
```

```
In [19]: # print the result
print(layer_out)
# Mine doesn't match his Jose's, even though I used
# the same random seed and had the same results above
# ... (?)
# He says it might be different, depending on how many
# times we had run random. I guess he ran random a few
# times in between
```

```
[[0.19592889 0.8423014 0.36188066]]
```

We just passed in random numbers for  $w$  and  $b$  ; we're not adjusting them.

We still need to finish off this process with optimization! Let's learn how to do this next.

---

## Full Network Example

Let's work on a regression example, we are trying to solve a very simple equation:

$$y = mx + b$$

$y$  will be the  $y\_labels$  and  $x$  is the  $x\_data$ . We are trying to figure out the slope and the intercept for the line that best fits our data!

## Artificial Data (Some Made Up Regression Data)

a.k.a. (what the lecture calls it)

## Simple Regression Example

```
In [20]: x_data = np.linspace(0, 10, 10) + np.random.uniform(-1.5, 1.5, 10)
```

```
In [21]: x_data
```

```
Out[21]: array([-1.20856056, -0.08034641,  2.82674411,  4.50477294,  3.42312535,  
                4.88227319,  7.18414126,  6.77068715,  9.4930023 ,  9.96290567])
```

```
In [22]: y_label = np.linspace(0, 10, 10) + np.random.uniform(-1.5, 1.5, 10)
```

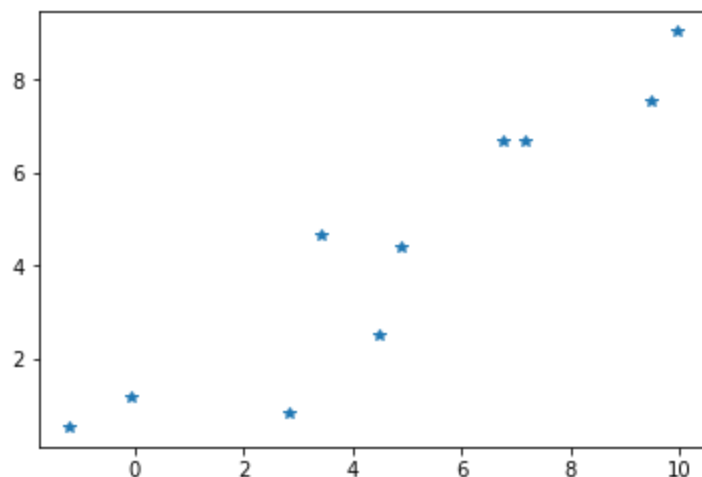
```
In [23]: y_label
```

```
Out[23]: array([0.5420333 , 1.17575569, 0.85241231, 2.50514314, 4.67005971,  
                4.41685654, 6.66701681, 6.69180648, 7.54731409, 9.03483077])
```

But that one matched (?). Maybe since it's np.random.uniform ...

```
In [26]: plt.plot(x_data, y_label, '*') # What have we created?  
        #+ Hopefully something with  
        #+ a linear trend.
```

```
Out[26]: [<matplotlib.lines.Line2D at 0x1ce6fd1ba58>]
```



## Variables

Remember, trying to solve  $y = mx + b$

```
In [27]: np.random.rand(2)
```

```
Out[27]: array([0.44236813, 0.87758732])
```

Just copying them in - 0.44 and 0.88. Wait, he wants us to use something different.

```
In [30]: m, b = np.random.rand(2)
```

```
In [41]: print("m = " + str(m) + " ; " + "b = " + str(b))
```

```
m = 0.4611193422864347 ; b = 0.637289031017357
```



The optimizer needs tensors. Otherwise, trying to run 'optimizer.minimize(error)' below will give:

```
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-40-f5a4eb180e> in <module>()
      1 optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001)
----> 2 train = optimizer.minimize(error)

~\.conda\envs\tfdeeplearning\lib\site-packages\tensorflow\python\training\optimizer.py in minimize(self, loss, glo
bal_step, var_list, gate_gradients, aggregation_method, colocate_gradients_with_ops, name, grad_loss)
    398         aggregation_method=aggregation_method,
    399         colocate_gradients_with_ops=colocate_gradients_with_ops,
--> 400         grad_loss=grad_loss)
    401
    402     vars_with_grad = [v for g, v in grads_and_vars if g is not None]

~\.conda\envs\tfdeeplearning\lib\site-packages\tensorflow\python\training\optimizer.py in compute_gradients(self,
loss, var_list, gate_gradients, aggregation_method, colocate_gradients_with_ops, grad_loss)
    492         "Optimizer.GATE_OP, Optimizer.GATE_GRAPH.  Not %s" %
    493         gate_gradients)
--> 494     self._assert_valid_dtypes([loss])
    495     if grad_loss is not None:
    496         self._assert_valid_dtypes([grad_loss])

~\.conda\envs\tfdeeplearning\lib\site-packages\tensorflow\python\training\optimizer.py in _assert_valid_dtypes(sel
f, tensors)
    870     valid_dtypes = self._valid_dtypes()
    871     for t in tensors:
--> 872         dtype = t.dtype.base_dtype
    873         if dtype not in valid_dtypes:
    874             raise ValueError(

AttributeError: 'numpy.dtype' object has no attribute 'base_dtype'
```

That's why I can't *just* pull them out of the array, like I did. I'll cast them.

```
In [43]: m = tf.Variable(m)
         b = tf.Variable(b)
         # it's pythonic, but I don't like it.
```

## Cost Function

```
In [44]: error = 0
```

```
In [46]: # zip will do tuples with points

for x, y in zip(x_data, y_label):
    y_hat = m*x + b # predicted y, m and b are tries

    error += (y - y_hat) ** 2
##endof: for x, y ...
```

## Optimizer

Learning rate: we don't want to overshoot the value, nor do we want to take forever to train things. Computations are expensive, and that matters with networks, because time matters.

```
In [47]: optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001)
         train = optimizer.minimize(error)
```

## Initialize Variables

```
In [48]: init = tf.global_variables_initializer()
```

## Create Session and Run!

```
In [51]: with tf.Session() as sess:

    sess.run(init)

    training_steps = 1 # probably way too low

    for i in range(training_steps):

        sess.run(train)

    ##endof: for i ...

    final_slope, final_intercept = sess.run([m, b]) # expecting a very bad job

    #endof: with ... sess
```

```
In [52]: final_slope
```

```
Out[52]: 0.6862147644515957
```

```
In [53]: final_intercept
```

```
Out[53]: 0.6687047474012019
```

## Evaluate Results

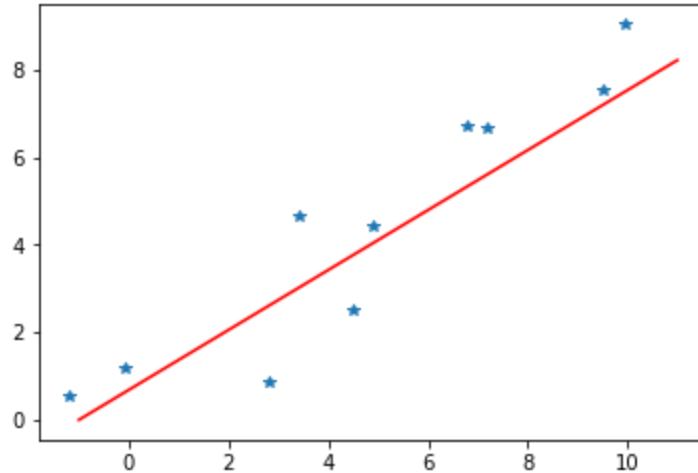
```
In [54]: x_test = np.linspace(-1, 11, 10) # Make plot look a little nicer

# y = mx + b
y_pred_plot = final_slope * x_test + final_intercept

plt.plot(x_test, y_pred_plot, 'r')

plt.plot(x_data, y_label, '*')
```

Out[54]: [<matplotlib.lines.Line2D at 0x1ce71146080>]



It did decently with one epoch (one step). Let's go with 100.

## Do it with 100 epochs

```
In [55]: with tf.Session() as sess_100:

    sess_100.run(init)

    #training_steps = 1 # 100 will be better
    epochs = 100

    for i in range(epochs):

        sess_100.run(train)

    ##endof: for i ...

    # Fetch the results
    final_slope_100, final_intercept_100 = sess_100.run([m, b])

#endof: with ... sess_100
```

```
In [56]: final_slope_100
```

```
Out[56]: 0.7693320305000105
```

```
In [57]: final_intercept_100
```

```
Out[57]: 0.7217691615910771
```

## Evaluate with 100 epochs

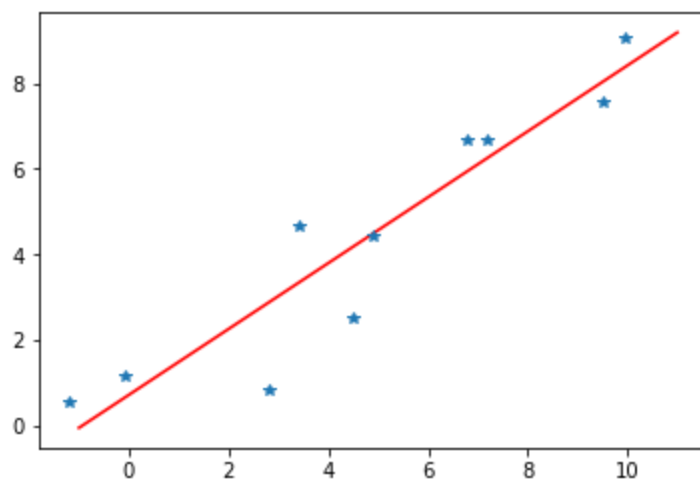
```
In [58]: x_test = np.linspace(-1, 11, 10) # Make plot look a little nicer

# y = mx + b
y_pred_plot_100 = final_slope_100 * x_test + final_intercept_100

plt.plot(x_test, y_pred_plot_100, 'r')

plt.plot(x_data, y_label, '*')
```

Out[58]: [<matplotlib.lines.Line2D at 0x1ce711bf4a8>]



That does indeed seem better

*That's all for now!*