I'm a developer first and foremost, so for my analysis I will be speaking from a code point of view and how it relates to the math. The first thing I notice is that this file is a tutorial file with comments everywhere, I will spend some time creating my own code to aid in my understanding of this process set in my own code standards.

The first thing I come across is the transfer function definitions. First a definition; a transfer or activation is the part of the equation where were increasing the preliminary output of the neuron by an amount in preparation to create the final output. The transfer function requires a preliminary input from the neuron as well as the alpha value. Alpha is defined as the slope of the transfer function.

Next we find the derivative of the transfer function, the purpose of this is to show the rate of change of the transfer function. I haven't gone through the rest of the code and as I said I'm more development and less math. My initial reaction is; Is this the learning rate?

Next I find a function that sets some variables, not quite sure why this is a function onto itself. We can set these variables at the top of the page without having to call anything. Re-coded on my own file to accommodate.

Next we have a function that initializes the weights, fairly standard I think.

Next we have a weight array initialization function along with the bias weight array, nothing strange here other than some coding changes.

Next we have a function that generates some training data for the XOR problem. My understanding from the code is that we get a random set of values each time. Made some slight code changes.

Next we have a function that defines a single neuron activation. From the code I understand that it takes in some inputs, weights, and a debug switch. The function sends some of these parameters to the transfer function defined up top and returns the activation output.

Next up is quite a large function defined as the single feed forward pass. The majority of it appears to be setting arrays and checking the debug option. The function also sends data into the single neuron activation function we defined earlier.

Next up we have another function that defines the SSE values. Couldn't remember what SSE stood for but that's what google is for. This function should help us measure our accuracy by taking the sum of squared errors. I see some potential here for refactoring the 3 chunks of code that get the input data and pass it into the feed forward by turning that into a for loop that would help minimize the amount of code we have to look at. Seeing as I'm still feeling cautious ill just leave it for now to ensure I don't break anything.

Now we're getting into the back propagation piece with the function print and trace back propagation to hidden (node?). I notice the alpha isn't used so I've removed that from my code, along with the deltavwt variables which don't appear to be used either.

Next looks like something very similar, tracing the back propagate to the hidden input. At a glance they look exactly the same the difference being their direction. Opportunity exists here to combine into a single function making the direction a parameter. Will leave as is until I feel more comfortable with the process.

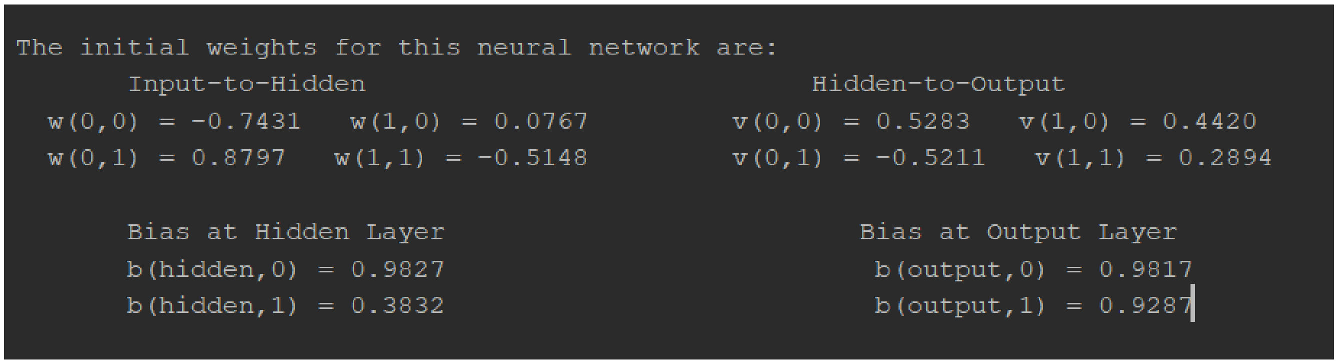
About halfway through the code, awesome! Looks like another few back propagation trace functions right before the main is set.

Now we get into our main function, before actually stepping through the code lets clean things up a bit. I think I now understand why they were initializing an array, in order to ensure that each variable had a separate random set of numbers. If I remember my operations right, because we are already calling a function within the array to determine the weight we shouldn't need to call the initialization function since each weight within the array is a function in itself. My refactor should be fine (famous last words).

With the refactor done it's time to run the file and see what happens, start up the debugger and step into the code line by line.

First we set our alpha to 1, iterations to 2k and eta to .1. Then we set our hidden array and output array lengths which are defined near the top of the code.
Unfortunately my refactor did break the weight array variables which means ill have to go back in and return them to a function, after that and a few tweaks we can keep going.

We set our v and w weights, we also define our bias arrays. Next we set our initial v and w weights and start setting some output;



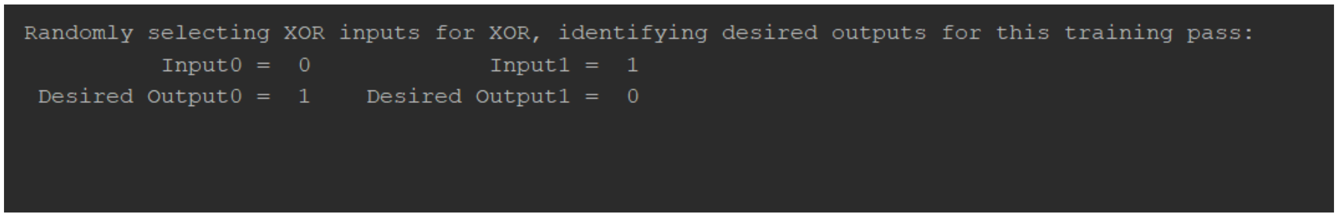
```
The initial weights for this neural network are:
       Input-to-Hidden                           Hidden-to-Output
  w(0,0) = -0.7431   w(1,0) = 0.0767        v(0,0) = 0.5283   v(1,0) = 0.4420
  w(0,1) = 0.8797   w(1,1) = -0.5148        v(0,1) = -0.5211   v(1,1) = 0.2894

       Bias at Hidden Layer                          Bias at Output Layer
       b(hidden,0) = 0.9827                           b(output,0) = 0.9817
       b(hidden,1) = 0.3832                           b(output,1) = 0.9287
```

Next we set epsilon and our iteration variables then set our sse array along with the initial total of 2.88.

Next we set the random xor data, which turned out to be 0,1,1,0,1 this run

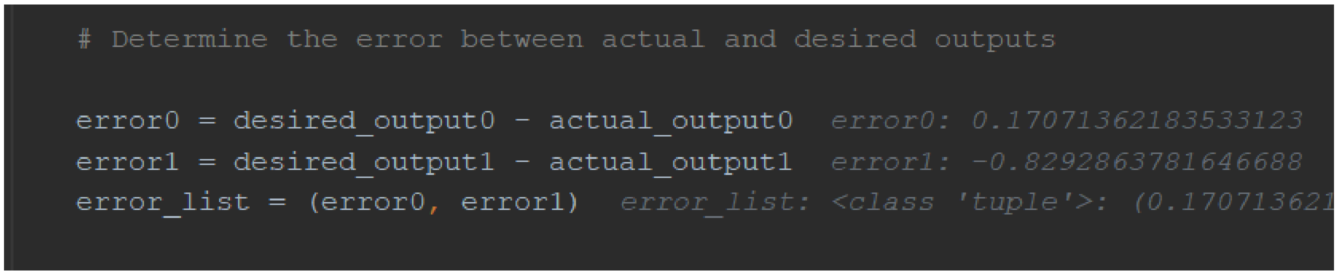
Then we set our input values and our desired output valies along with the set number.

Next we step into more output;



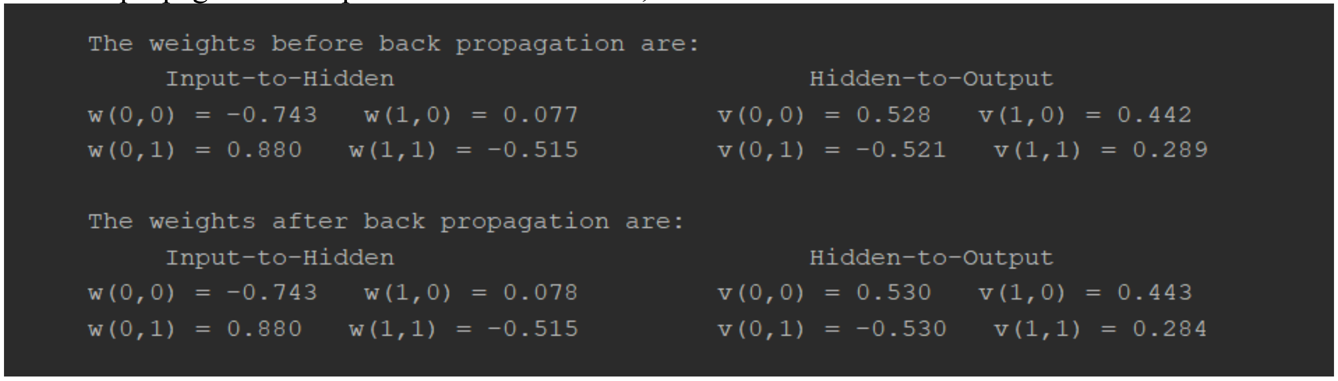
```
Randomly selecting XOR inputs for XOR, identifying desired outputs for this training pass:
         Input0 =  0             Input1 =  1
 Desired Output0 =  1    Desired Output1 =  0
```

Then we get our actual all nodes output list and go into our hidden actual and actual outputs followed by our errors;



```
# Determine the error between actual and desired outputs

error0 = desired_output0 - actual_output0  error0: 0.17071362183533123
error1 = desired_output1 - actual_output1  error1: -0.8292863781646688
error_list = (error0, error1)  error_list: <class 'tuple'>: (0.170713621
```

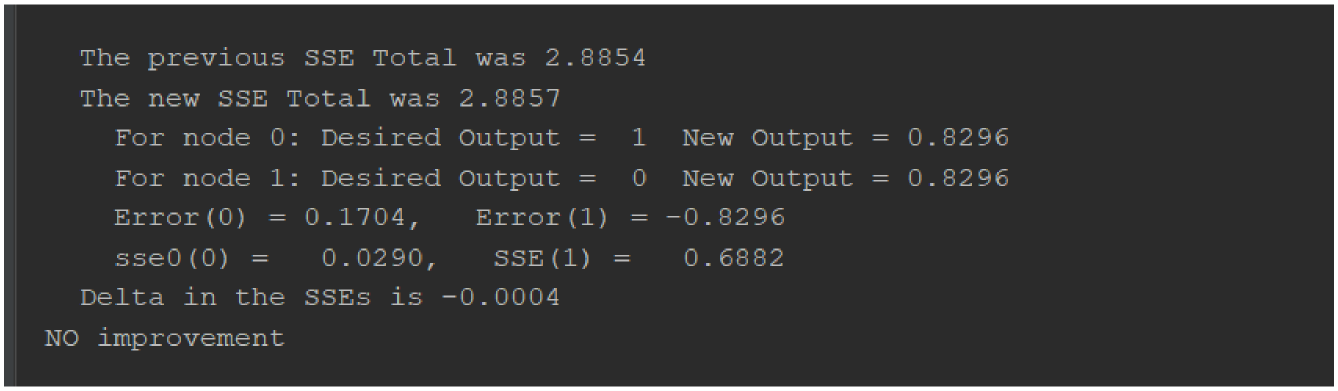
Next we propagate the output to the hidden nodes;



```
The weights before back propagation are:
     Input-to-Hidden                          Hidden-to-Output
w(0,0) = -0.743   w(1,0) = 0.077         v(0,0) = 0.528   v(1,0) = 0.442
w(0,1) = 0.880   w(1,1) = -0.515         v(0,1) = -0.521   v(1,1) = 0.289

The weights after back propagation are:
     Input-to-Hidden                          Hidden-to-Output
w(0,0) = -0.743   w(1,0) = 0.078         v(0,0) = 0.530   v(1,0) = 0.443
w(0,1) = 0.880   w(1,1) = -0.515         v(0,1) = -0.530   v(1,1) = 0.284
```

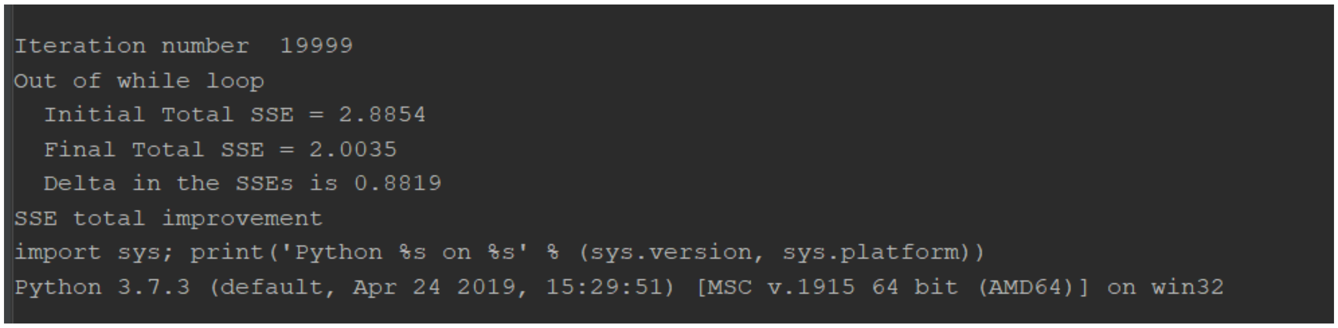
Looks like there was a slight change to the weights since we passed in our initial output.

Next we get an all new set of outputs using the adjusted weights and create and output to measure progress;



```
  The previous SSE Total was 2.8854
  The new SSE Total was 2.8857
    For node 0: Desired Output =  1  New Output = 0.8296
    For node 1: Desired Output =  0  New Output = 0.8296
    Error(0) = 0.1704,   Error(1) = -0.8296
    sse0(0) =   0.0290,   SSE(1) =   0.6882
  Delta in the SSEs is -0.0004
NO improvement
```

And that's it, that's a whole iteration stepped through. Now lets do it 20k more times.

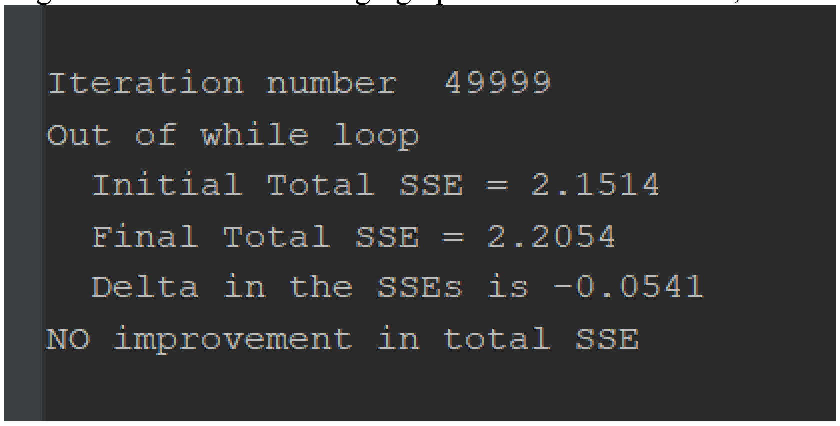


```
Iteration number  19999
Out of while loop
  Initial Total SSE = 2.8854
  Final Total SSE = 2.0035
  Delta in the SSEs is 0.8819
SSE total improvement
import sys; print('Python %s on %s' % (sys.version, sys.platform))
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)] on win32
```

The code analysis and re-write helped me get a better understanding of how back propagation works.

We take an input and an expected output and compare the two. Using SSE we adjust our weights in order to move the actual output closer in the direction of the desired output. We run this process over as many iterations, as we deem necessary.

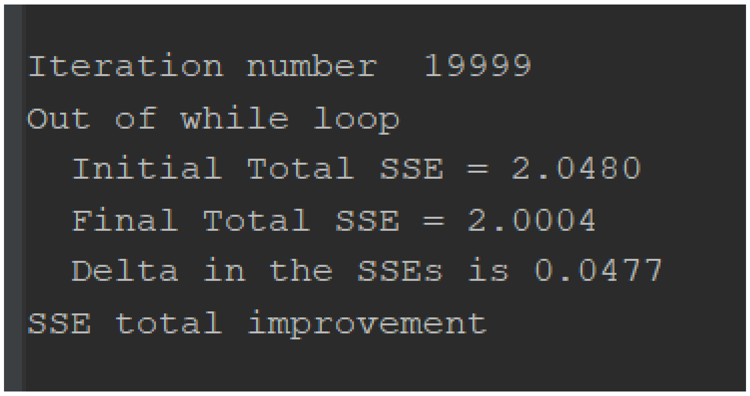
I started experimenting with the code and changing up some of the variables;



```
Iteration number  49999
Out of while loop
  Initial Total SSE = 2.1514
  Final Total SSE = 2.2054
  Delta in the SSEs is -0.0541
NO improvement in total SSE
```

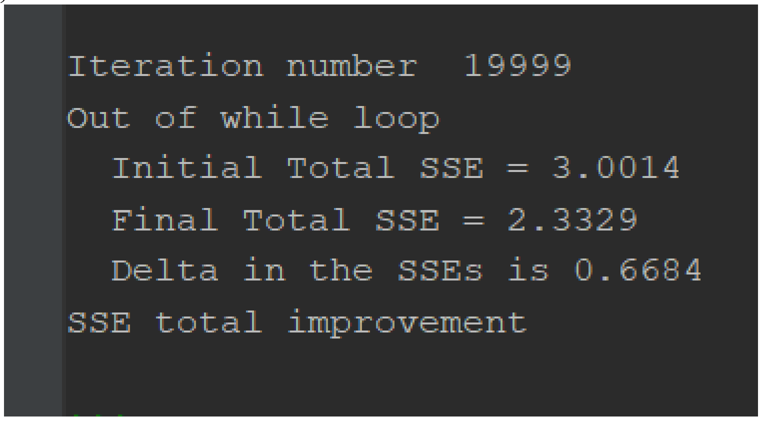
More iterations didn't result in better accuracy.

Next I set the alpha to .5

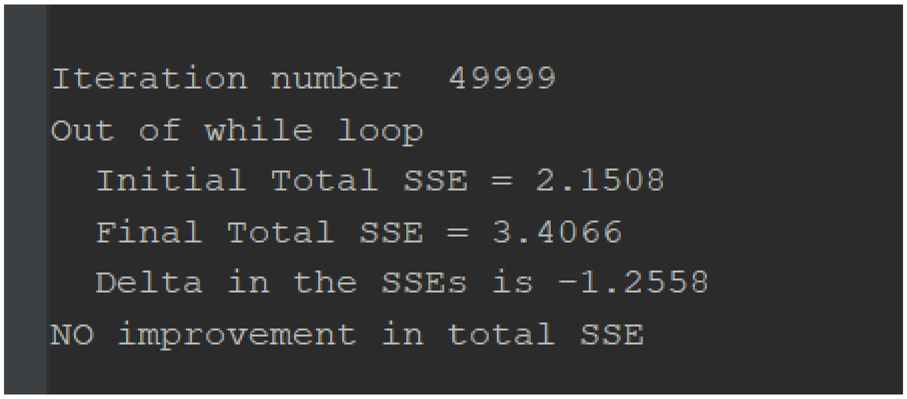


```
Iteration number  19999
Out of while loop
  Initial Total SSE = 2.0480
  Final Total SSE = 2.0004
  Delta in the SSEs is 0.0477
SSE total improvement
```

Not much of a change, how about 2?

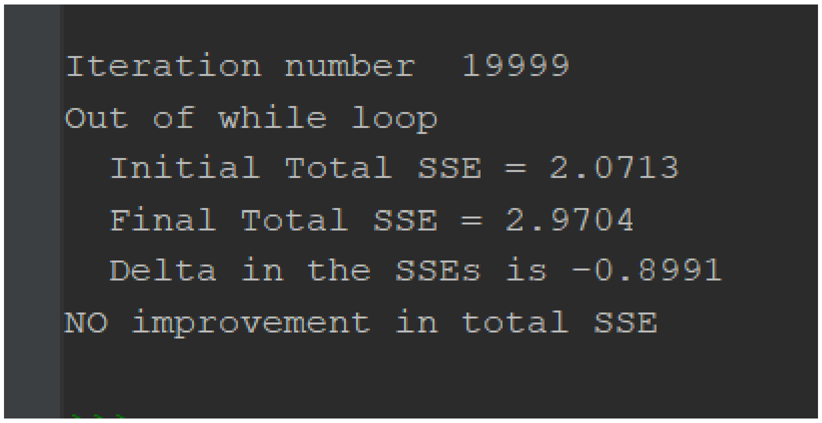


```
Iteration number  19999
Out of while loop
  Initial Total SSE = 3.0014
  Final Total SSE = 2.3329
  Delta in the SSEs is 0.6684
SSE total improvement
```



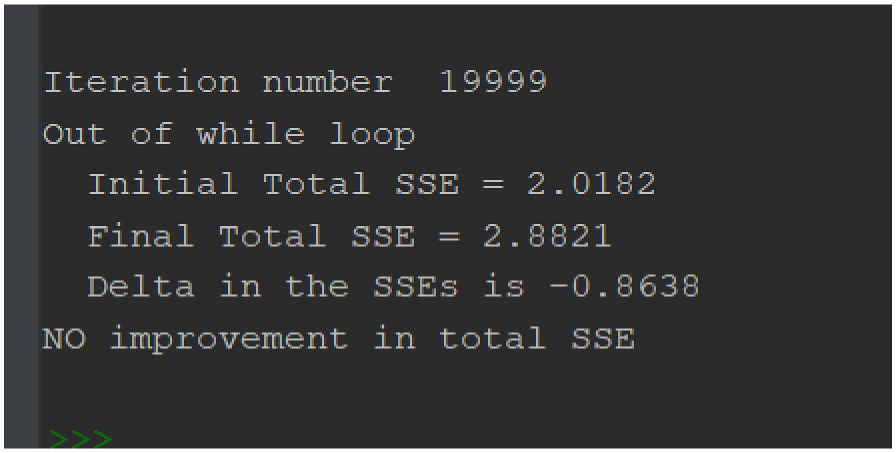
```
Iteration number  49999
Out of while loop
  Initial Total SSE = 2.1508
  Final Total SSE = 3.4066
  Delta in the SSEs is -1.2558
NO improvement in total SSE
```

Next tweaking the learning rate or the eta;
1st run at .5



```
Iteration number  19999
Out of while loop
  Initial Total SSE = 2.0713
  Final Total SSE = 2.9704
  Delta in the SSEs is -0.8991
NO improvement in total SSE
```

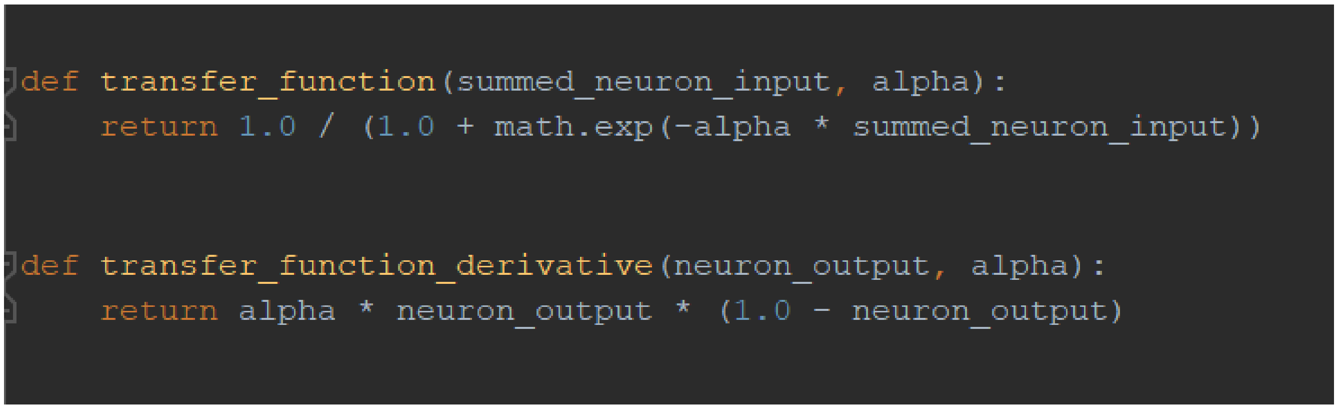
2nd run at .5



```
Iteration number  19999
Out of while loop
  Initial Total SSE = 2.0182
  Final Total SSE = 2.8821
  Delta in the SSEs is -0.8638
NO improvement in total SSE

>>>
```

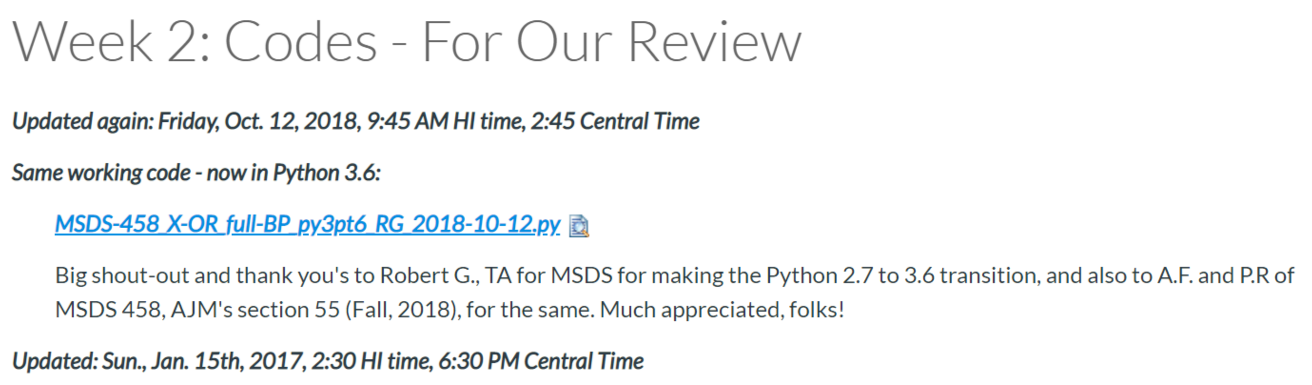
Increasing the learning rate seems to have given me worse results so I tried lowering it and didn't get a negative delta.

I used the sigmoid functions in this code since that's what was coded already. From the readings, which function you use depends what your trying to do. Sigmoid is great for probability since it exists between 0 and 1. Sigmoid converted to code looks like the below;



```
def transfer_function(summed_neuron_input, alpha):
    return 1.0 / (1.0 + math.exp(-alpha * summed_neuron_input))


def transfer_function_derivative(neuron_output, alpha):
    return alpha * neuron_output * (1.0 - neuron_output)
```

The code referenced in this assignment comes from the week 2: Codes – For Our Review page



Week 2: Codes - For Our Review

Updated again: Friday, Oct. 12, 2018, 9:45 AM HI time, 2:45 Central Time

Same working code - now in Python 3.6:

MSDS-458_X-OR_full-BP_py3pt6_RG_2018-10-12.py

Big shout-out and thank you's to Robert G., TA for MSDS for making the Python 2.7 to 3.6 transition, and also to A.F. and P.R of MSDS 458, AJM's section 55 (Fall, 2018), for the same. Much appreciated, folks!

Updated: Sun., Jan. 15th, 2017, 2:30 HI time, 6:30 PM Central Time

I've also attached my code if anyone's interested.