# Department of Computing

**CS471: Machine Learning**

**Class: BESE-7AB**

**Lab 5: Artificial Neural Network**

**Kashif Nazir Khan**

**BESE 7B**

**193105**

**CL03: Understand over fitting, bias-variance tradeoff and parameter optimization**

**Date: 22-02-2019**

**Time: 10:00 am– 1:00 pm & 2:00 pm-5:00 pm**

**Instructor: Dr. Pakeeza Akram**

**Lab 10: Artificial Neural Network**

**Introduction**

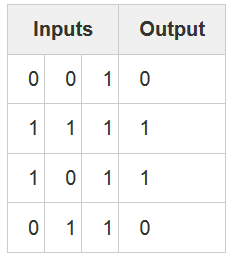
In this lab you will develop your own ANN (Artificial Neural Network) from the basic ANN development code via python implementation.

**Tools/Software Requirement**

Python, numpy

**Tasks**

Consider the example of the tiny toy problem where the neural network is attempting to use input to predict output.



We are trying to predict the output column given the three input columns.

import numpy as np

A [sigmoid function](https://en.wikipedia.org/wiki/Sigmoid_function) maps any value to a value between 0 and 1. We use it to convert numbers to probabilities.

# sigmoid function

def nonlin(x,deriv=False):

if(deriv==True):

return x\*(1-x)

return 1/(1+np.exp(-x))

Notice that this function can also generate the derivative of a sigmoid (when deriv=True).

Next we have to initialize our input dataset as a numpy matrix. Each row is a single "training example". Each column corresponds to one of our input nodes. Thus, we have 3 input nodes to the network and 4 training examples.

# input dataset

X = np.array([  [0,0,1],

[0,1,1],

[1,0,1],

[1,1,1] ])

Then we’ll initializes our output dataset. In this case, we’ll generate the dataset horizontally (with a single row and 4 columns) for space. ".T" is the transpose function. After the transpose, this y matrix has 4 rows with one column. Just like our input, each row is a training example, and each column (only one) is an output node. So, our network has 3 inputs and 1 output.

# output dataset

y = np.array([[0,0,1,1]]).T

It's good practice to seed your random numbers. Your numbers will still be randomly distributed, but they'll be randomly distributed in exactly the same way each time you train. This makes it easier to see how your changes affect the network.

# seed random numbers to make calculation

# deterministic (just a good practice)

np.random.seed(1)

Next, is our weight matrix for this neural network. It's called "syn0" to imply "synapse zero". Since we only have 2 layers (input and output), we only need one matrix of weights to connect them. Its dimension is (3,1) because we have 3 inputs and 1 output.

# initialize weights randomly with mean 0

syn0 = 2\*np.random.random((3,1)) – 1

Our actual network training code starts from here. This for loop "iterates" multiple times over the training code to optimize our network to the dataset.

for iter in xrange(10000):

# forward propagation

l0 = X

l1 = nonlin(np.dot(l0,syn0))

# how much did we miss?

l1\_error = y - l1

# multiply how much we missed by the

# slope of the sigmoid at the values in l1

l1\_delta = l1\_error \* nonlin(l1,True)

# update weights

syn0 += np.dot(l0.T,l1\_delta)

Since our first layer, l0, is simply our data. We explicitly describe it as such at this point. Remember that X contains 4 training examples (rows). We're going to process all of them at the same time in this implementation. This is known as "full batch" training. Thus, we have 4 different l0 rows, but you can think of it as a single training example if you want. It makes no difference at this point. (We could load in 1000 or 10,000 if we wanted to without changing any of the code).

The code marked in red is our prediction step. Basically, we first let the network "try" to predict the output given the input. We will then study how it performs so that we can adjust it to do a bit better for each iteration.

Since we loaded in 4 training examples, we ended up with 4 guesses for the correct answer, a (4 x 1) matrix. Each output corresponds with the network's guess for a given input.

Further details on how the network updates can be found here:

<https://iamtrask.github.io/2015/07/12/basic-python-network/>

At the end print the output.

print "Output After Training:"

print l1

Considering our example when both an input and a output are 1, we increase the weight between them. When an input is 1 and an output is 0, we decrease the weight between them.

Thus, in our four training examples, the weight from the first input to the output would **consistently increment or remain unchanged**, whereas the other two weights would find themselves **both increasing and decreasing across training examples** (cancelling out progress). This phenomenon is what causes our network to learn based on correlations between the input and output.

**Tasks**

Develop your own ANN

* **Iamtrask code for basic ANN development**

(Described above)

Code:

import numpy as np

# sigmoid function

def nonlin(x,deriv=False):

if(deriv==True):

return x\*(1-x)

return 1/(1+np.exp(-x))

# input dataset

X = np.array([ [0,0,1],

[0,1,1],

[1,0,1],

[1,1,1] ])

# output dataset

y = np.array([[0,0,1,1]]).T

# seed random numbers to make calculation

# deterministic (just a good practice)

np.random.seed(1)

# initialize weights randomly with mean 0

syn0 = 2\*np.random.random((3,1)) - 1

for iter in range(10000):

# forward propagation

l0 = X

l1 = nonlin(np.dot(l0,syn0))

# how much did we miss?

l1\_error = y - l1

# multiply how much we missed by the

# slope of the sigmoid at the values in l1

l1\_delta = l1\_error \* nonlin(l1,True)

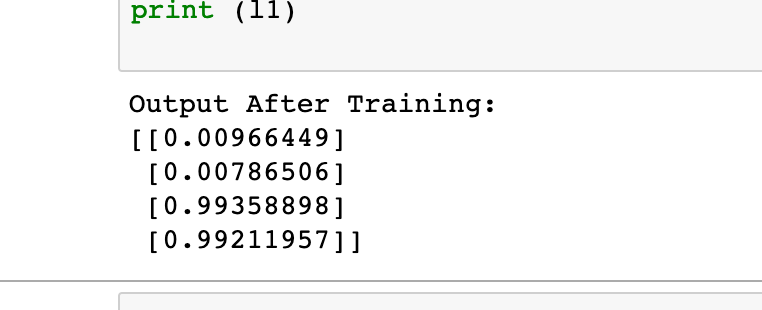
# update weights

syn0 += np.dot(l0.T,l1\_delta)

print ("Output After Training:")

print (l1)

Screenshot:



* **Change the number of layers and neurons**

For adding layer more syn type calculations need to be done. Essentially adding syn2, l3, and l3\_delta and their calculations in line 6 - 11. (Checkout Reference link for line 6-11)

Line 6 and 7 make the neurons logreg. To make any neuron linear regression then for that value this step would be skipped. (Checkout Reference link for line 6-11)

Code:

import numpy as np

def nonlin(x,deriv=False):

if(deriv==True):

return x\*(1-x)

return 1/(1+np.exp(-x))

X = np.array([[0,0,1],

[0,1,1],

[1,0,1],

[1,1,1]])

y = np.array([[0],

[1],

[1],

[0]])

np.random.seed(1)

# randomly initialize our weights with mean 0

syn0 = 2\*np.random.random((3,4)) - 1

syn1 = 2\*np.random.random((4,3)) - 1

syn2 = 2\*np.random.random((3,1)) - 1

for j in range(60000):

# Feed forward through layers 0, 1, and 2

l0 = X

l1 = nonlin(np.dot(l0,syn0))

l2 = nonlin(np.dot(l1,syn1))

l3 = nonlin(np.dot(l2,syn2))

# how much did we miss the target value?

l3\_error = y - l3

if (j% 10000) == 0:

print ("Error:" + str(np.mean(np.abs(l3\_error))))

# in what direction is the target value?

# were we really sure? if so, don't change too much.

l3\_delta = l3\_error\*nonlin(l3,deriv=True)

# how much did we miss the target value?

l2\_error = y - l2

# in what direction is the target value?

# were we really sure? if so, don't change too much.

l2\_delta = l2\_error\*nonlin(l2,deriv=True)

# how much did each l1 value contribute to the l2 error (according to the weights)?

l1\_error = l2\_delta.dot(syn1.T)

# in what direction is the target l1?

# were we really sure? if so, don't change too much.

l1\_delta = l1\_error \* nonlin(l1,deriv=True)

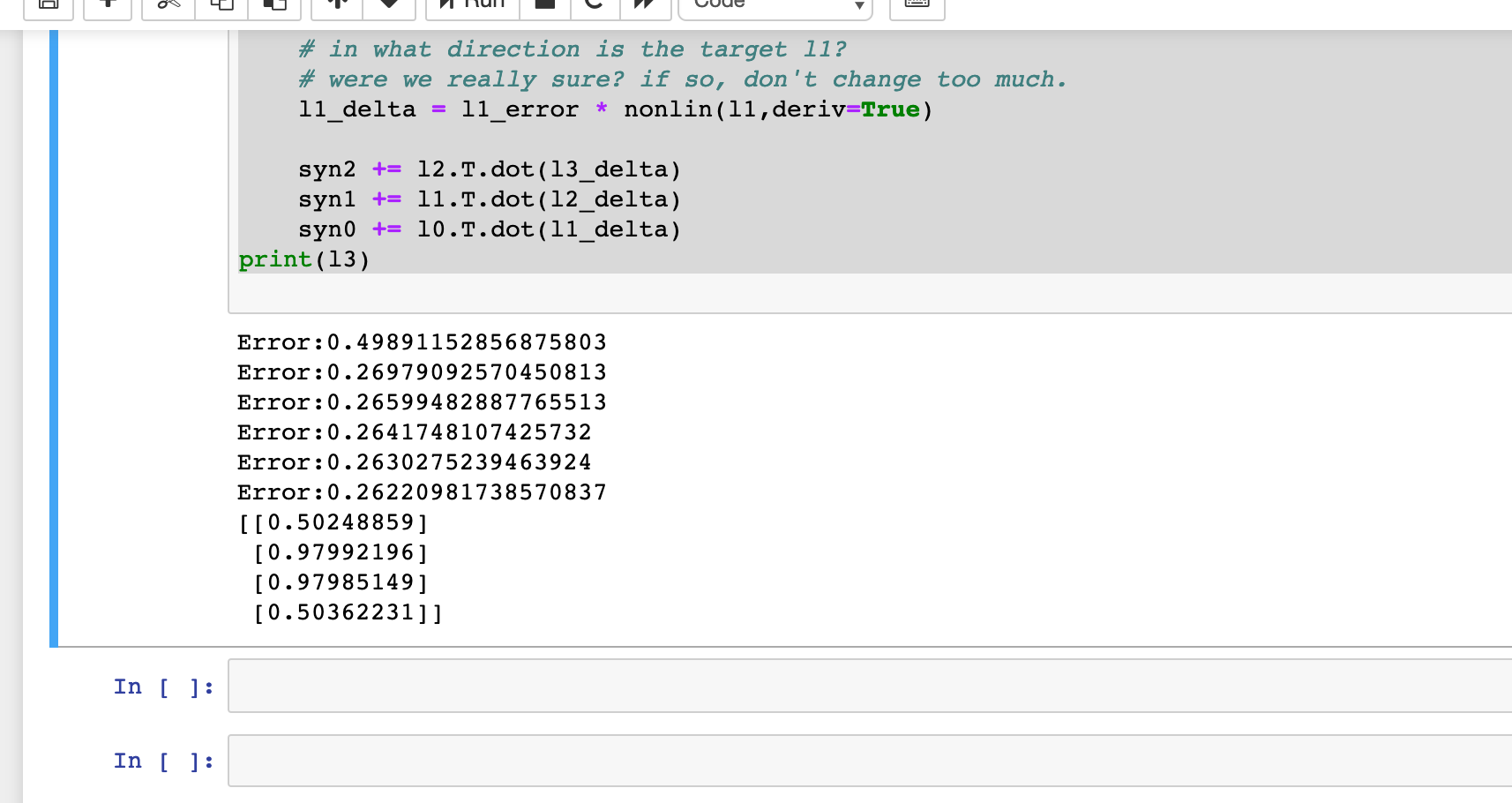
syn2 += l2.T.dot(l3\_delta)

syn1 += l1.T.dot(l2\_delta)

syn0 += l0.T.dot(l1\_delta)

print(l3)

Screenshot:



* **Multi-class classification**

Changing sizes of syn0 and syn1 changes the number of neurons in layers. To make it multi-class classification syn1 has to have columns= # of classes.

Code:

import numpy as np

def nonlin(x,deriv=False):

if(deriv==True):

return x\*(1-x)

return 1/(1+np.exp(-x))

X = np.array([[0,0,1],

[0,1,1],

[1,0,1],

[1,1,1]])

y = np.array([[0, 1],

[1, 0],

[1, 0],

[0, 1]])

np.random.seed(1)

# randomly initialize our weights with mean 0

syn0 = 2\*np.random.random((3,4)) - 1

syn1 = 2\*np.random.random((4,2)) - 1

for j in range(60000):

# Feed forward through layers 0, 1, and 2

l0 = X

l1 = nonlin(np.dot(l0,syn0))

l2 = nonlin(np.dot(l1,syn1))

# how much did we miss the target value?

l2\_error = y - l2

if (j% 10000) == 0:

print ("Error:" + str(np.mean(np.abs(l2\_error))))

# in what direction is the target value?

# were we really sure? if so, don't change too much.

l2\_delta = l2\_error\*nonlin(l2,deriv=True)

# how much did each l1 value contribute to the l2 error (according to the weights)?

l1\_error = l2\_delta.dot(syn1.T)

# in what direction is the target l1?

# were we really sure? if so, don't change too much.

l1\_delta = l1\_error \* nonlin(l1,deriv=True)

syn1 += l1.T.dot(l2\_delta)

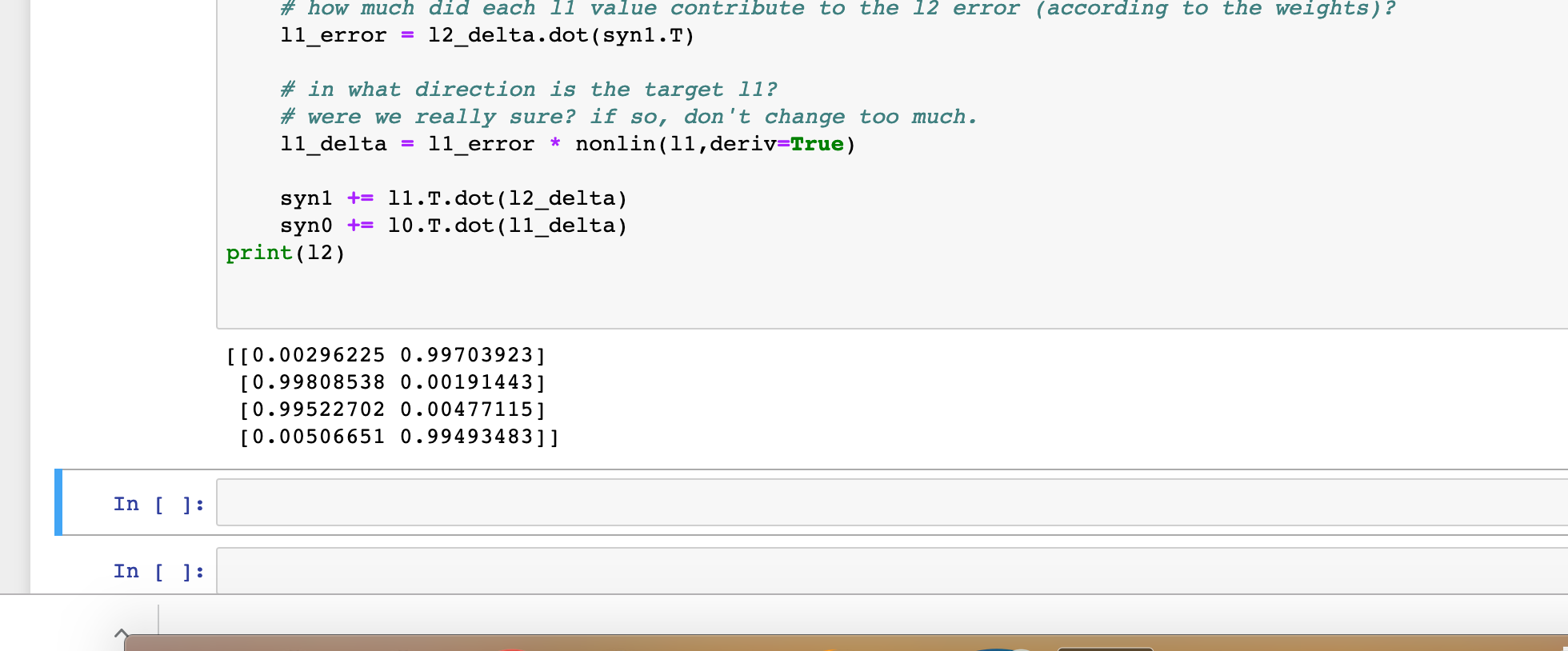
syn0 += l0.T.dot(l1\_delta)

print(l2)

for i in l2:

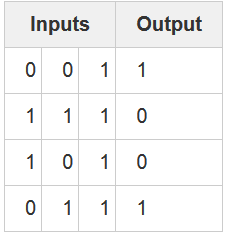
print(np.argmax(i))

Screenshot:



* **Regression function by using different layers as regressors rather than logreg.**

Input for logistic regression (logreg) should be:



Code:

import numpy as np

def nonlin(x,deriv=False):

if(deriv==True):

return x\*(1-x)

return 1/(1+np.exp(-x))

X = np.array([[0,0,1],

[0,1,1],

[1,0,1],

[1,1,1]])

y = np.array([[0, 1],

[1, 0],

[1, 0],

[0, 1]])

np.random.seed(1)

# randomly initialize our weights with mean 0

syn0 = 2\*np.random.random((3,4)) - 1

syn1 = 2\*np.random.random((4,2)) - 1

for j in range(60000):

# Feed forward through layers 0, 1, and 2

l0 = X

l1 = nonlin(np.dot(l0,syn0))

l2 = nonlin(np.dot(l1,syn1))

# how much did we miss the target value?

l2\_error = y - l2

# in what direction is the target value?

# were we really sure? if so, don't change too much.

l2\_delta = l2\_error\*nonlin(l2,deriv=True)

# how much did each l1 value contribute to the l2 error (according to the weights)?

l1\_error = l2\_delta.dot(syn1.T)

# in what direction is the target l1?

# were we really sure? if so, don't change too much.

l1\_delta = l1\_error \* nonlin(l1,deriv=True)

syn1 += l1.T.dot(l2\_delta)

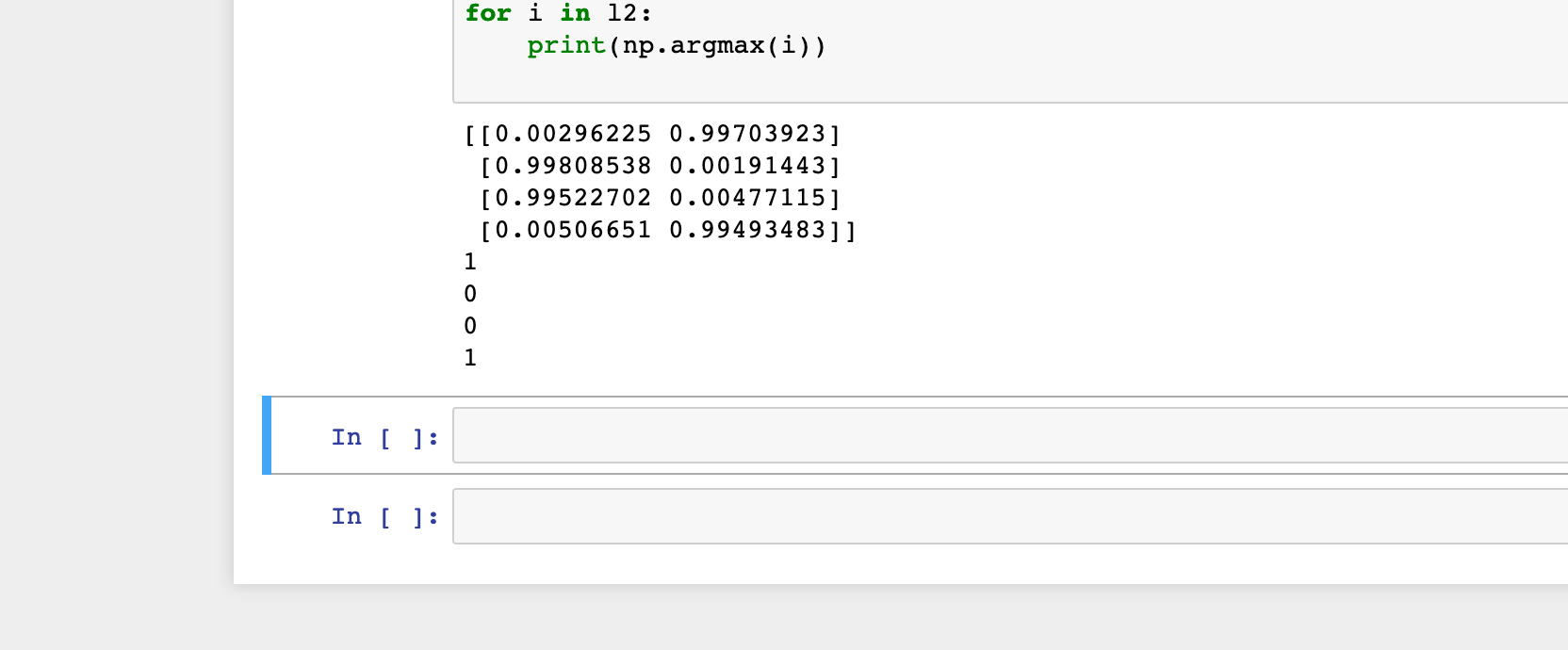
syn0 += l0.T.dot(l1\_delta)

print(l2)

for i in l2:

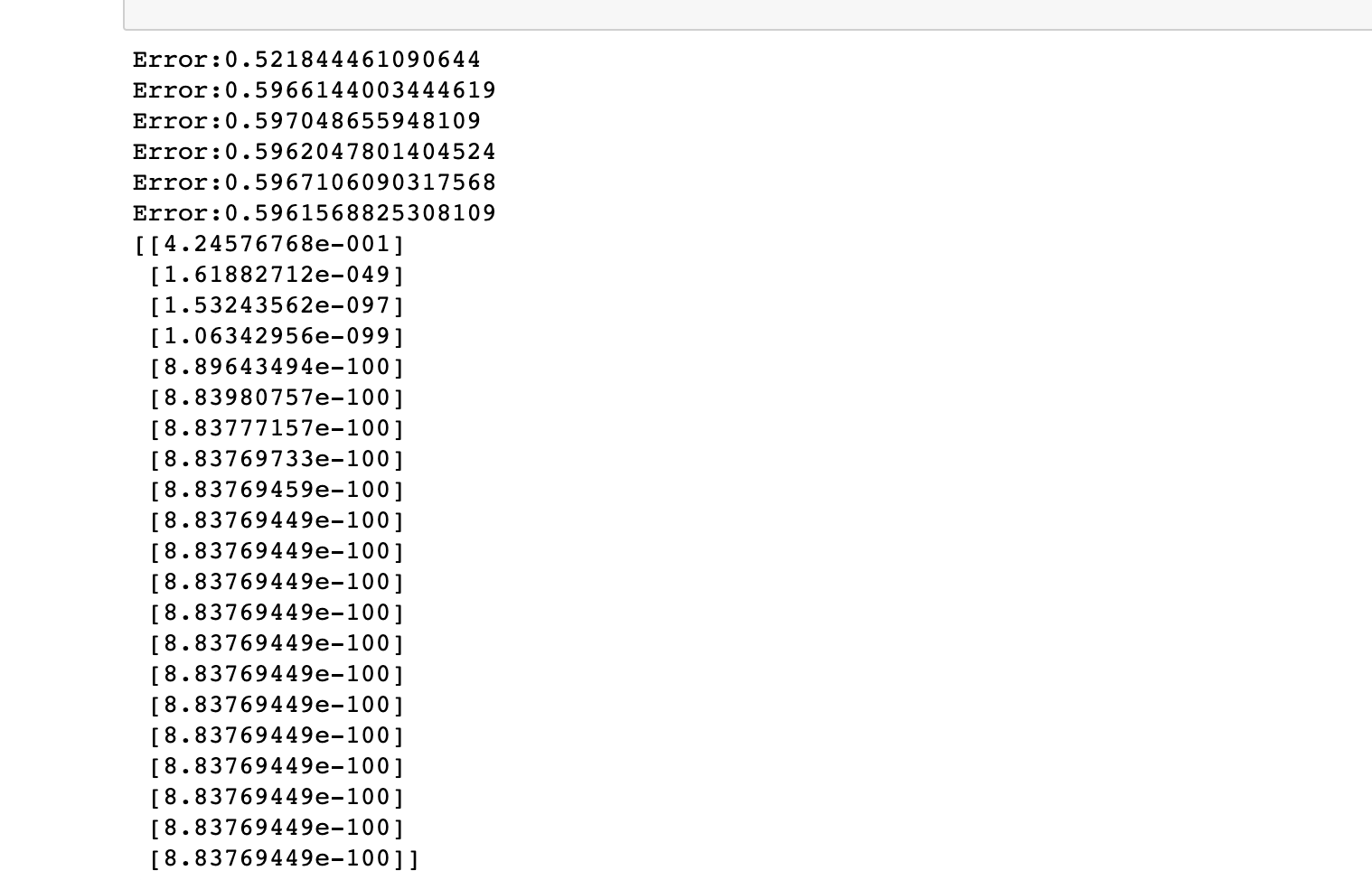
print(np.argmax(i))

Screenshot:



And for regression (This is simply sin(x)):

|  |  |
| --- | --- |
| 0 | 0 |
| 0.2 | 0.198669 |
| 0.4 | 0.389418 |
| 0.6 | 0.564642 |
| 0.8 | 0.717356 |
| 1 | 0.841471 |
| 1.2 | 0.932039 |
| 1.4 | 0.98545 |
| 1.6 | 0.999574 |
| 1.8 | 0.973848 |
| 2 | 0.909297 |
| 2.2 | 0.808496 |
| 2.4 | 0.675463 |
| 2.6 | 0.515501 |
| 2.8 | 0.334988 |
| 3 | 0.14112 |
| 3.2 | -0.05837 |
| 3.4 | -0.25554 |
| 3.6 | -0.44252 |
| 3.8 | -0.61186 |
| 4 | -0.7568 |

****

import numpy as np

def nonlin(x,deriv=False):

if(deriv==True):

return x\*(1-x)

return 1/(1+np.exp(-x))

X = np.array([[0],

[0.2],

[0.4],

[0.6],

[0.8],

[1],

[1.2],

[1.4],

[1.6],

[1.8],

[2],

[2.2],

[2.4],

[2.6],

[2.8],

[3],

[3.2],

[3.4],

[3.6],

[3.8],

[4]])

y = np.array([[0],

[0.198669],

[0.389418],

[0.564642],

[0.717356],

[0.841471],

[0.932039],

[0.98545],

[0.999574],

[0.973848],

[0.909297],

[0.808496],

[0.675463],

[0.515501],

[0.334988],

[0.14112],

[-0.05837],

[-0.25554],

[-0.44252],

[-0.61186],

[-0.7568]])

np.random.seed(1)

# randomly initialize our weights with mean 0

syn0 = 2\*np.random.random((1,100)) - 1

syn1 = 2\*np.random.random((100,100)) - 1

syn2 = 2\*np.random.random((100,1)) - 1

for j in range(60000):

# Feed forward through layers 0, 1, and 2

l0 = X

l1 = nonlin(np.dot(l0,syn0))

l2 = nonlin(np.dot(l1,syn1))

l3 = nonlin(np.dot(l2,syn2))

# how much did we miss the target value?

l3\_error = y - l3

if (j% 10000) == 0:

print ("Error:" + str(np.mean(np.abs(l3\_error))))

# in what direction is the target value?

# were we really sure? if so, don't change too much.

l3\_delta = l3\_error\*nonlin(l3,deriv=True)

# how much did we miss the target value?

l2\_error = y - l2

# in what direction is the target value?

# were we really sure? if so, don't change too much.

l2\_delta = l2\_error\*nonlin(l2,deriv=True)

# how much did each l1 value contribute to the l2 error (according to the weights)?

l1\_error = l2\_delta.dot(syn1.T)

# in what direction is the target l1?

# were we really sure? if so, don't change too much.

l1\_delta = l1\_error \* nonlin(l1,deriv=True)

syn2 += l2.T.dot(l3\_delta)

syn1 += l1.T.dot(l2\_delta)

syn0 += l0.T.dot(l1\_delta)

print(l3)

for i in l3:

print(np.argmax(i))

**Deliverables**

* Word File with all codes and screen shots of results.
* Pynb files zipped in a folder.