ECS171 homework 2

Zhen Zhang

Question 1

For this question I write the algorithm all by myself in python.

The first step is use R to preprocess the data a little bit, and I attach the R code in the folder. Next I will explain the basic structure of my python code.

I define a class named Ann, that can accepts data, train percentage, learning rate and maximum iteration as the inputs. First I define some static methods in class as tools to do subsequent jobs, including sigmoid (the transformation function), theta_part2 (which calculates part two of each theta), dim_expand (add an 1 at dimension 0 to the input data), weight_init (initialize weight by the required row and column number), error_compute (compute error using RSS method).

Where n is the number of observations of the data set, m is the number of output neuros, in this problem, 10.

Now back to the code. Next comes the next part, forward_prop implements the process of going forward, and it supplies a2 and a3 to the next function, backward. There are two parts of backward function, one from output to hidden, and another from hidden to input. This code format supports multiple dimension, so adding an extra hidden layer can also be achieved. Inside them there are function to update weight (weight_update) and at last, the forward_backward function combines them together.

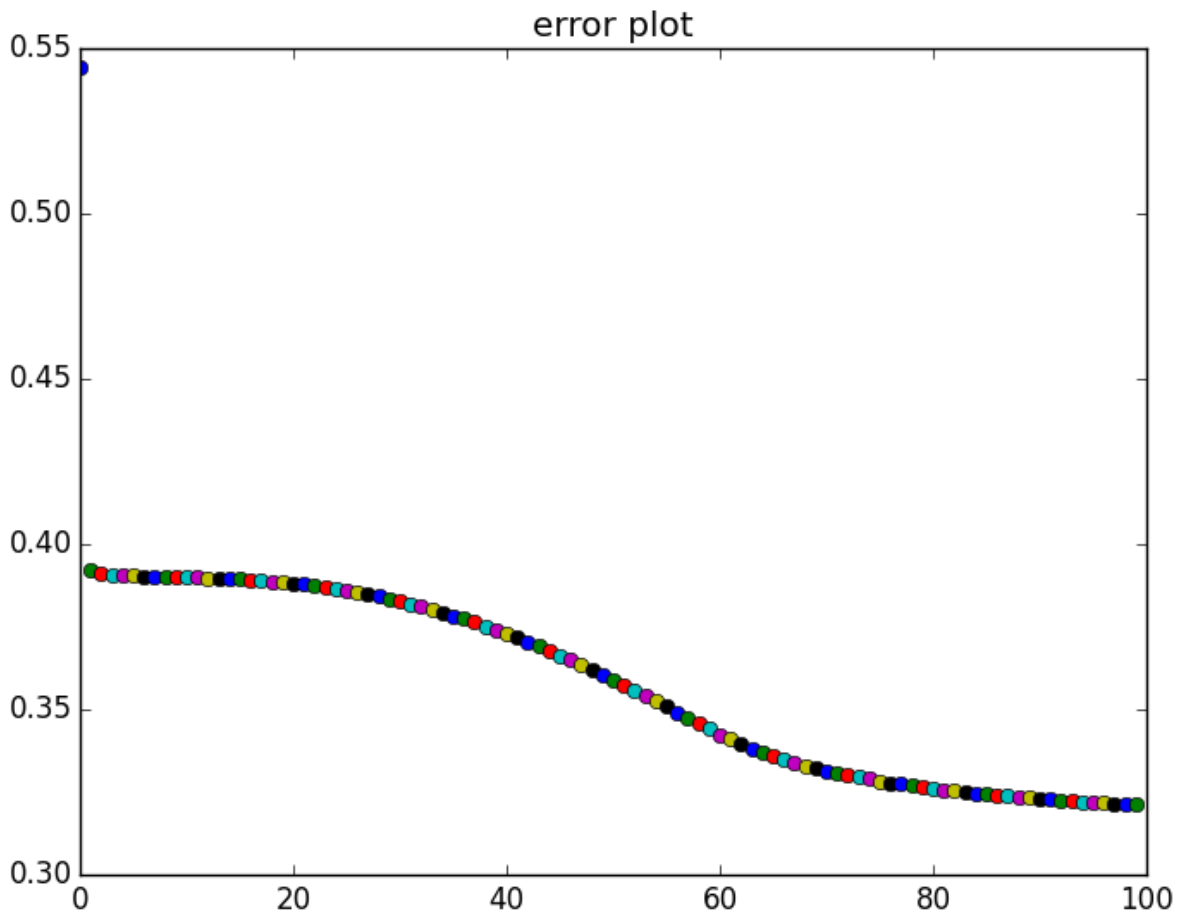
Next is the plot area. I give the error, output and weights required to plot the result, and they call the corresponding result it need to plot the diagram. Inside plot_output and plot_weight, I use them to call the plot_single function to plot each iteration by calling the map function. At last, I combine them into plot_total to do all the things altogether.

Next is the main regression function ann_regression_one, it first initialize the

weight vector, then depending on I need the train data or the whole data, it calls the corresponding property of the class object. It also initializes the output, weights and error to fill in afterwards. After that it updates the weights in each iteration by each observation, and it is what should be done in stochastic gradient descent (SGD). In the iteration, it fills output, weights and error by the right value. It returns the output, weights and error as the return value to be used by other functions.

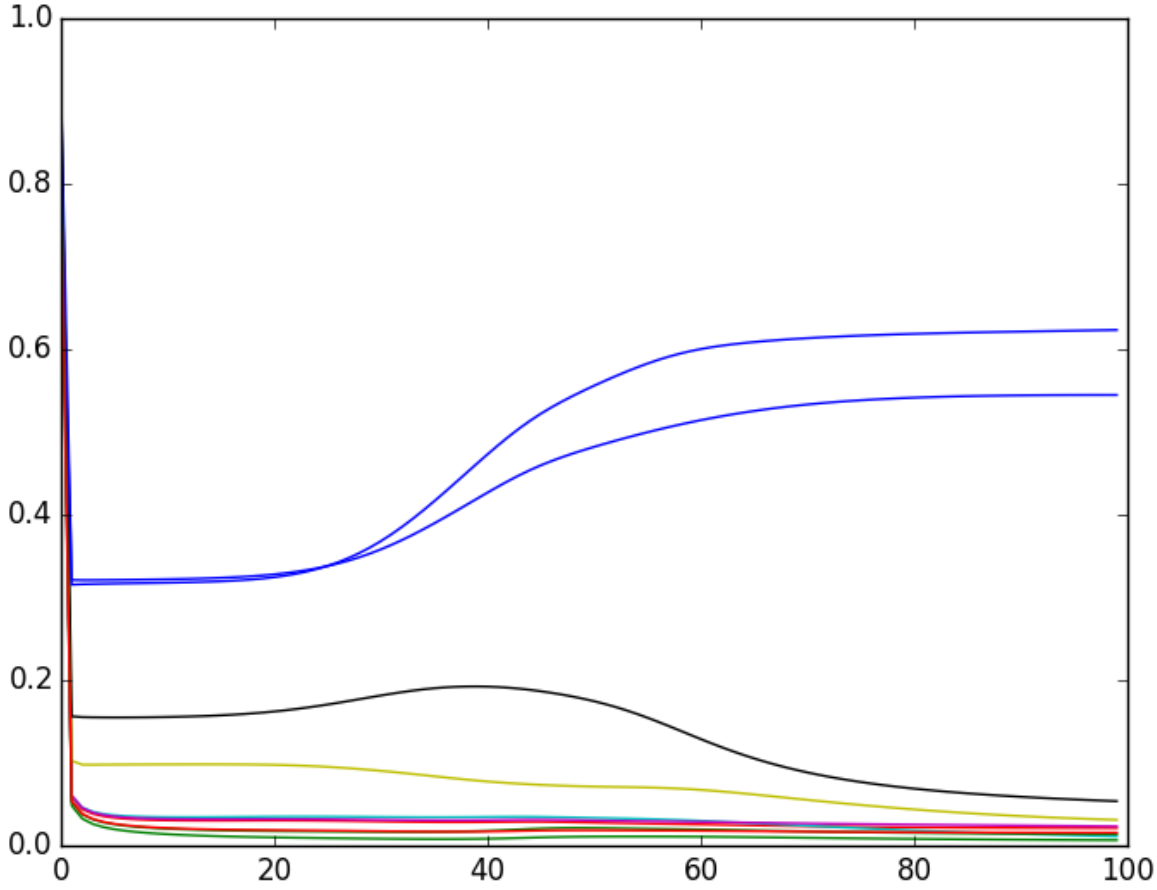
They are enough for the first question, and the remaining function will be introduced in the following question. Now I will show the results.

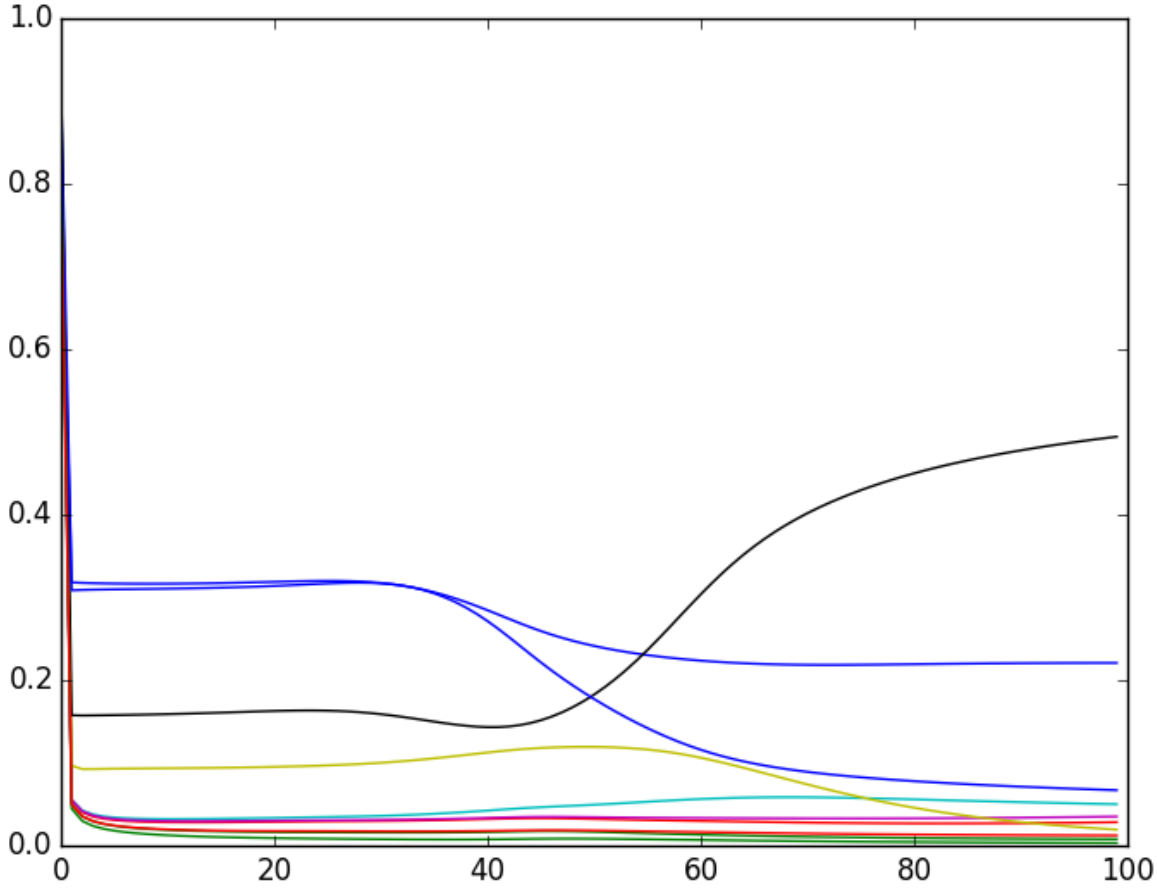
First I will show you the error plot for each iteration:

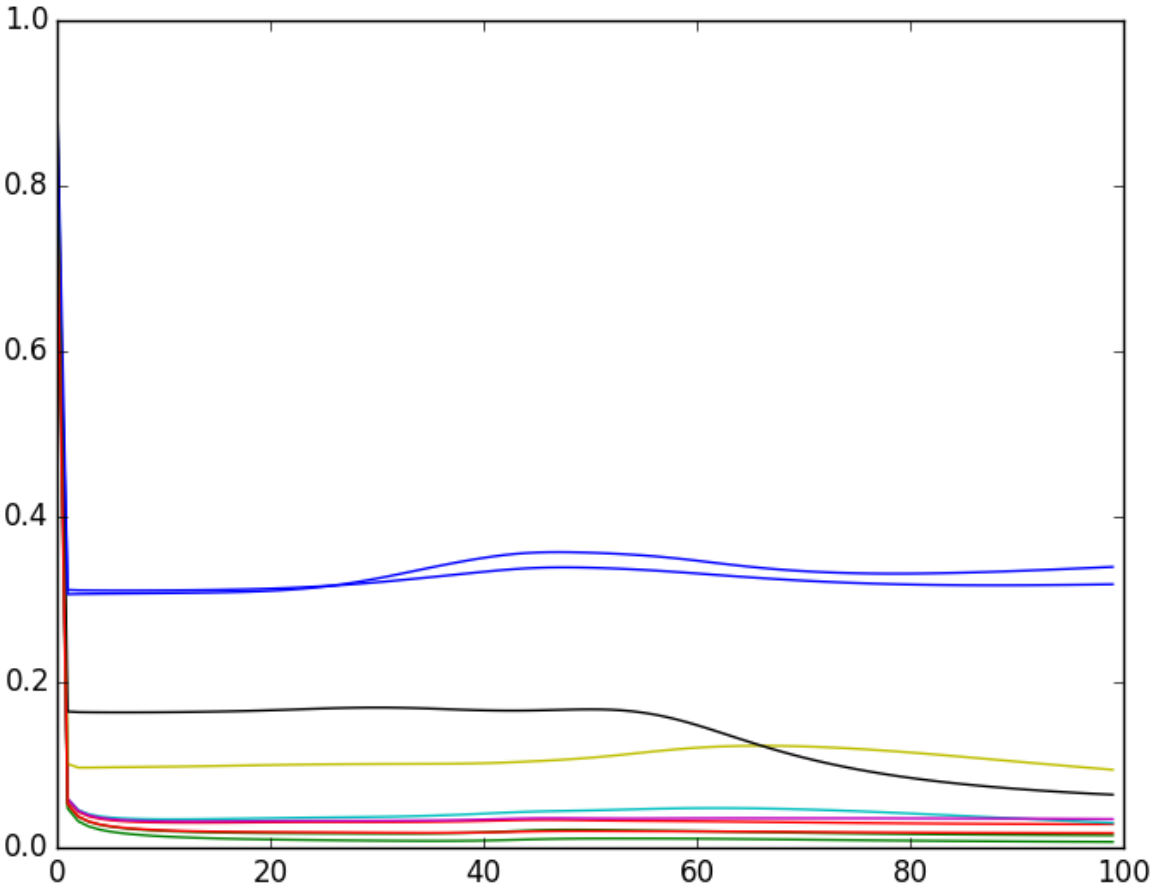


It seems the error is decreasing in most of the time, and the final train error becomes about 0.32.

Second I will show the first three outputs:

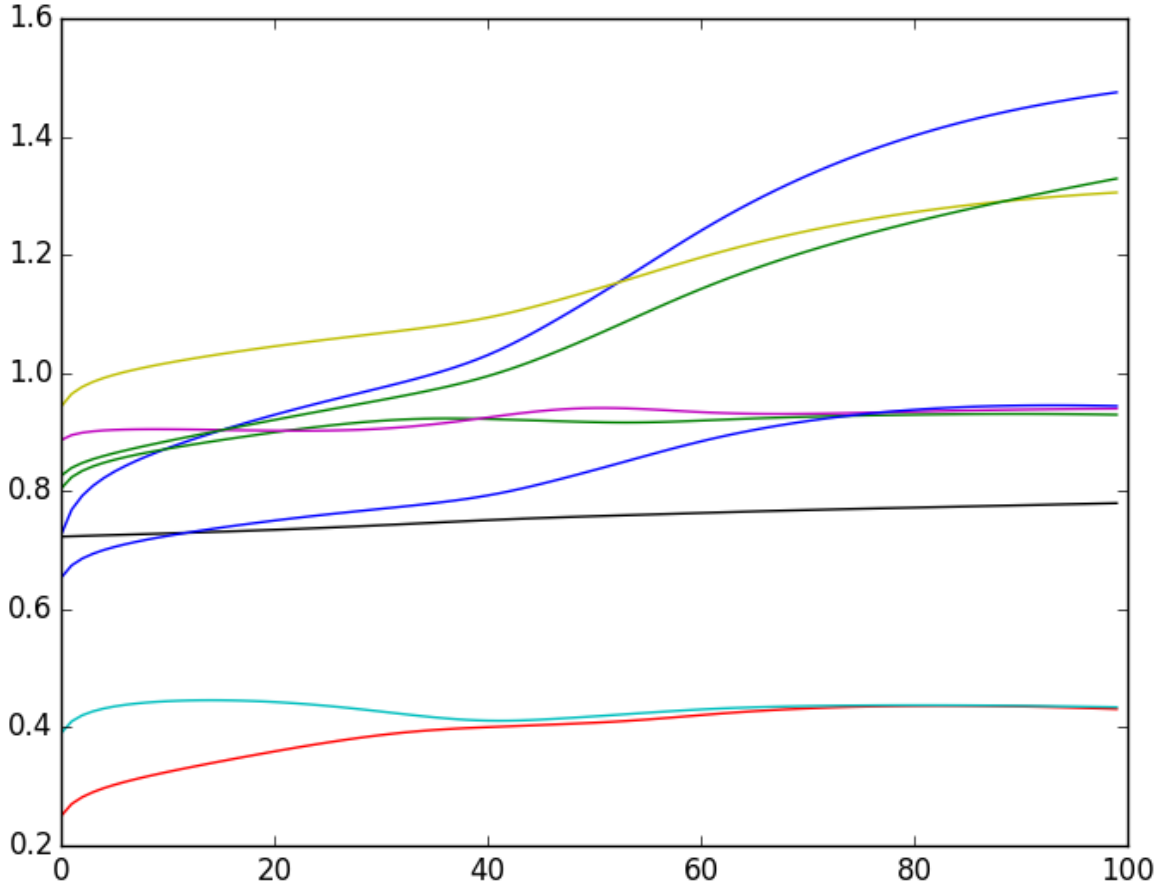

1.0
0.8
0.6
0.4
0.2
0.0
0
20
40
60
80
100

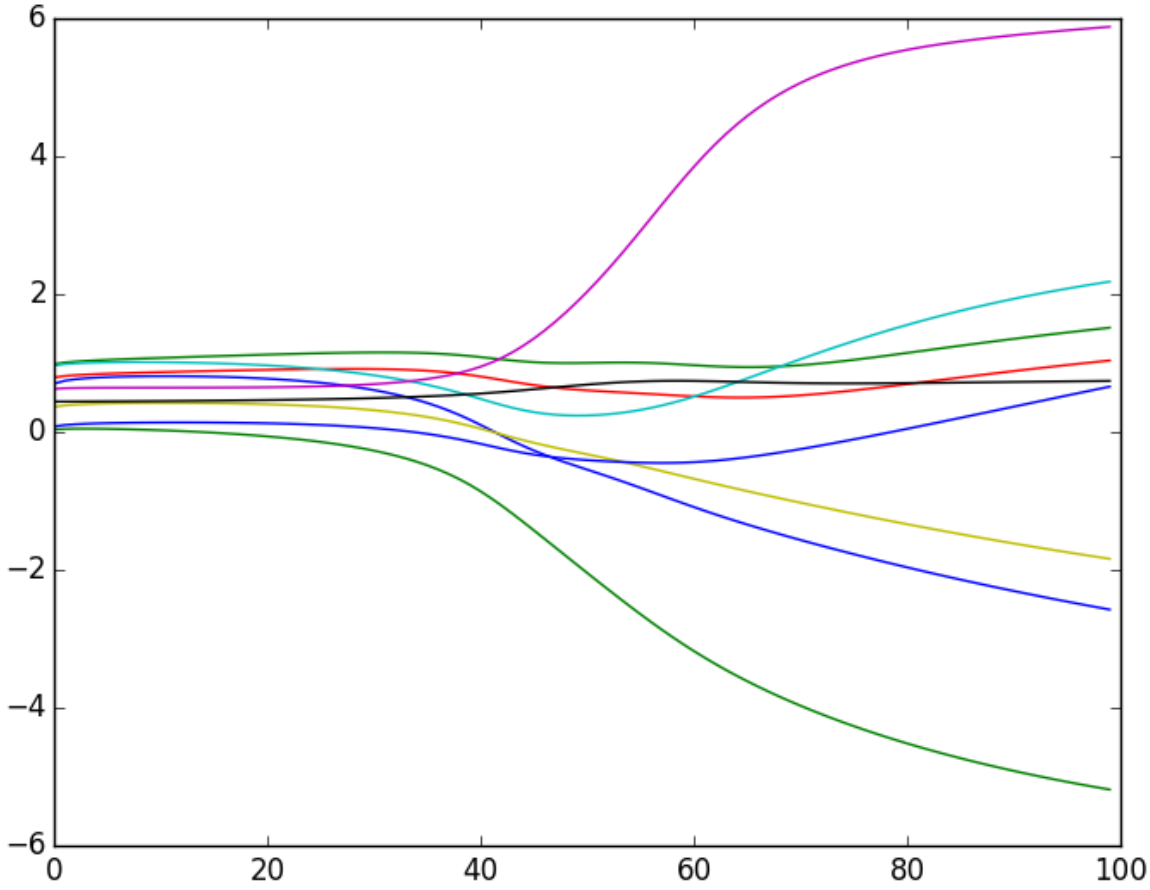

1.0
0.8
0.6
0.4
0.2
0.0
0
20
40
60
80
100

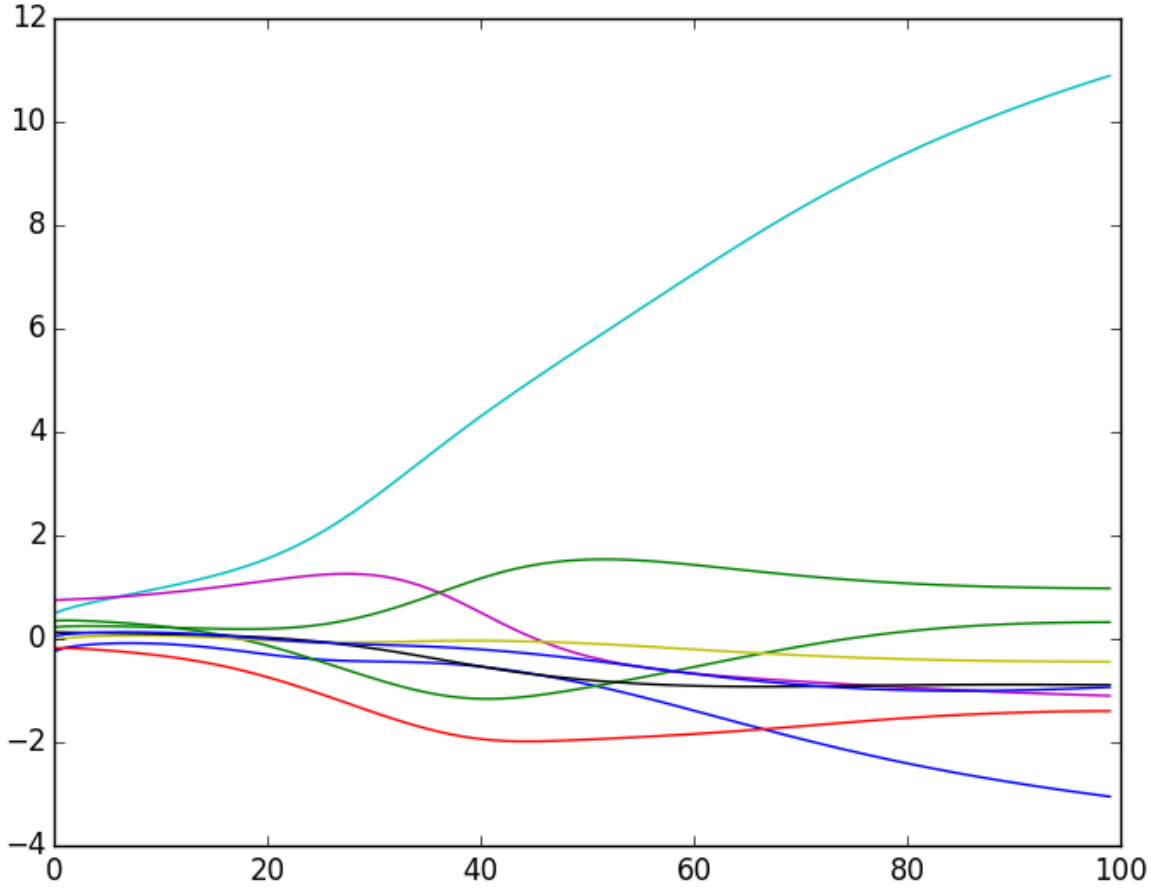


Each line represents the value in each of the ten classes. Now we can see that in each plot, there is one of more value larger than the rest, which will be the classified class. After I check the result, they are the same.

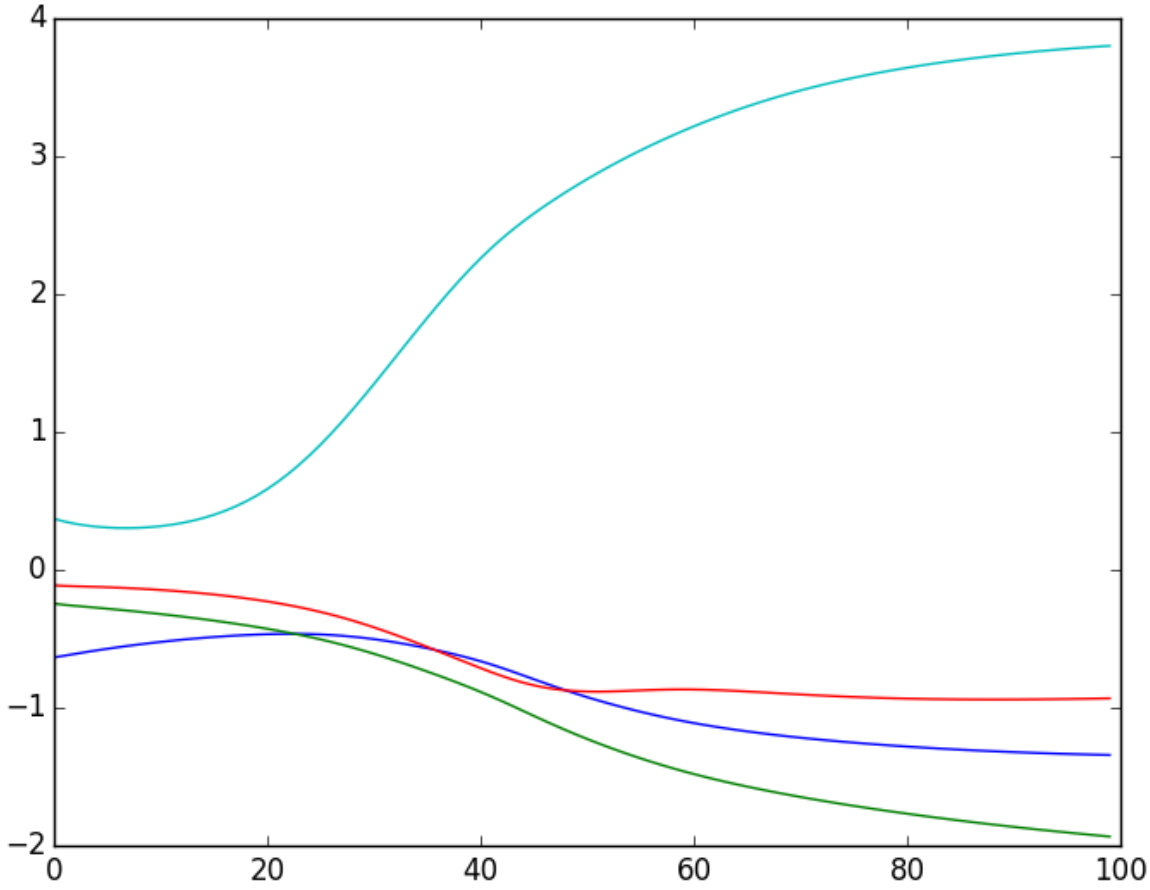
Now I will show the weights of input to hidden. Since there are three neuros in hidden layer, I will display three plots, one for each neuros, with nine lines representing the link from the previous node to this one:

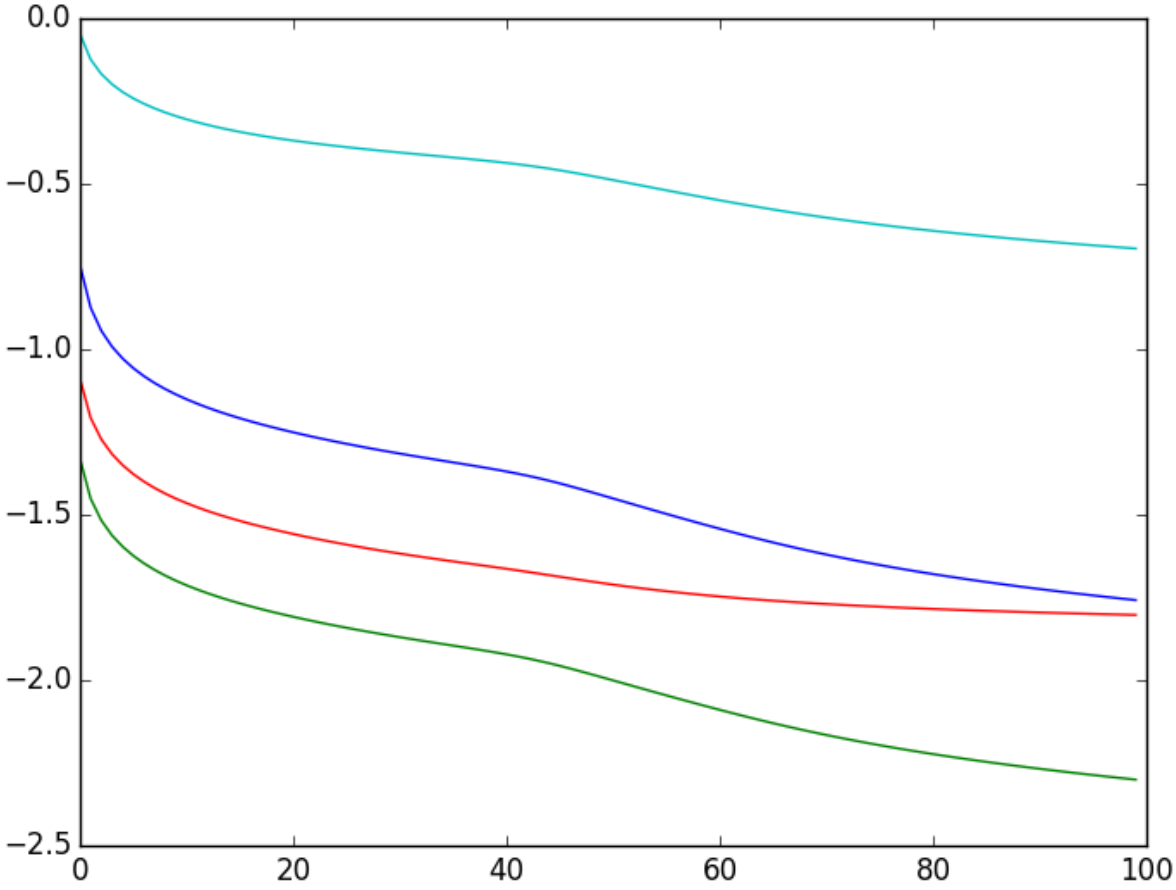

1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0
20
40
60
80
100

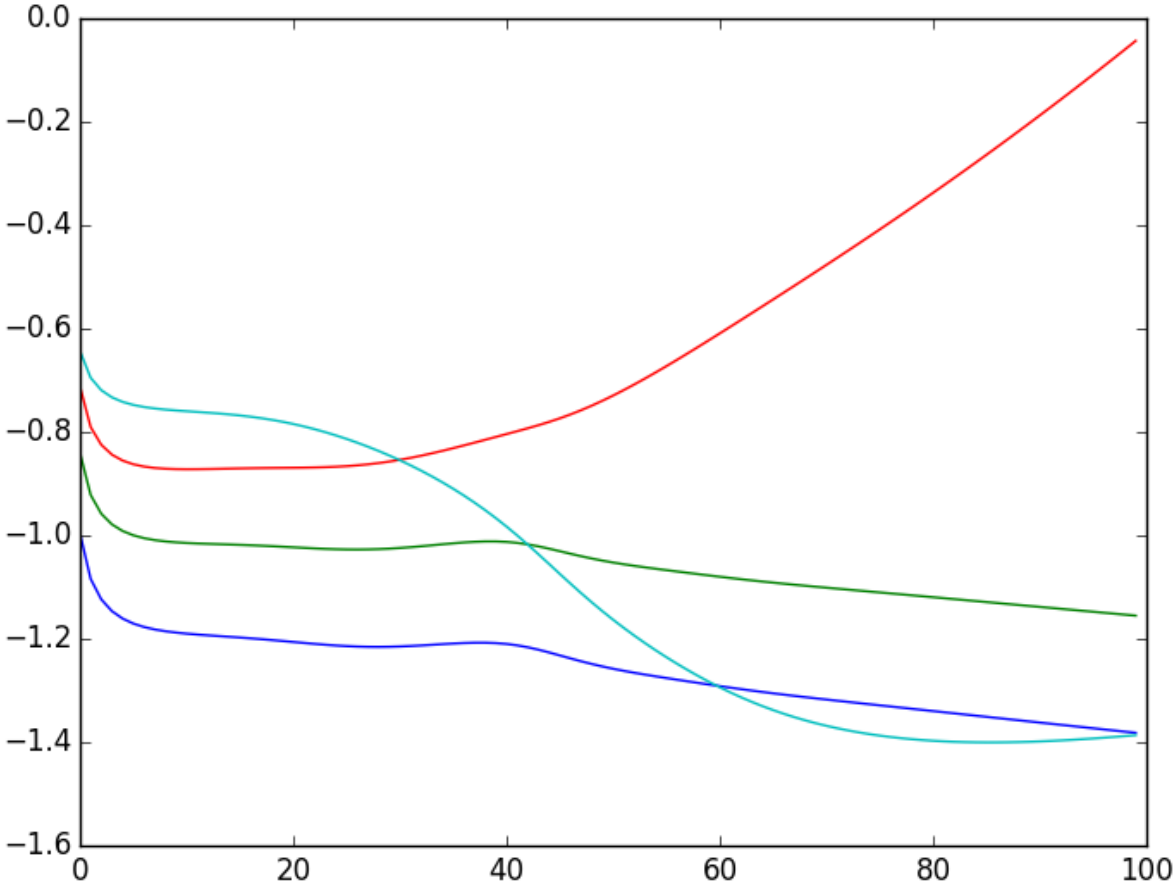

6
4
2
0
−2
−4
−6
0
20
40
60
80
100

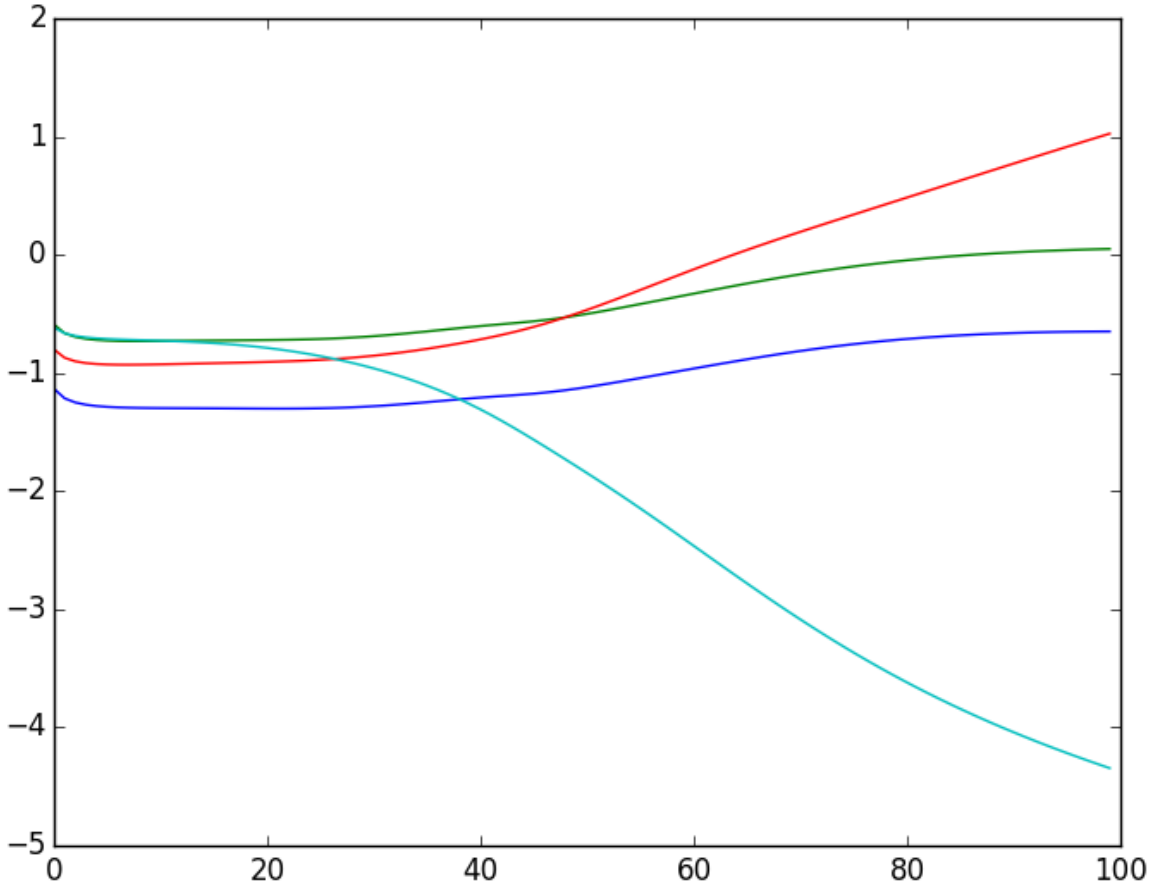


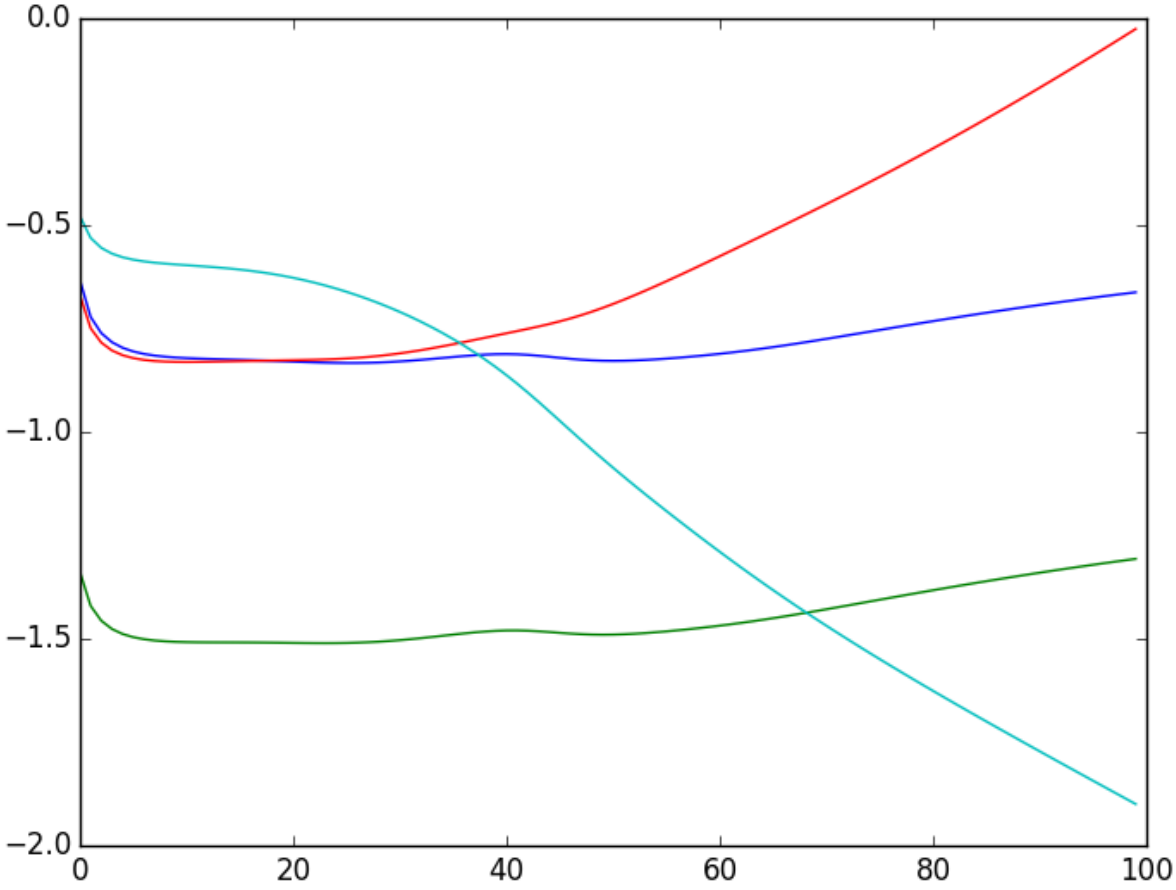
Now comes the weights from hidden to output layer:

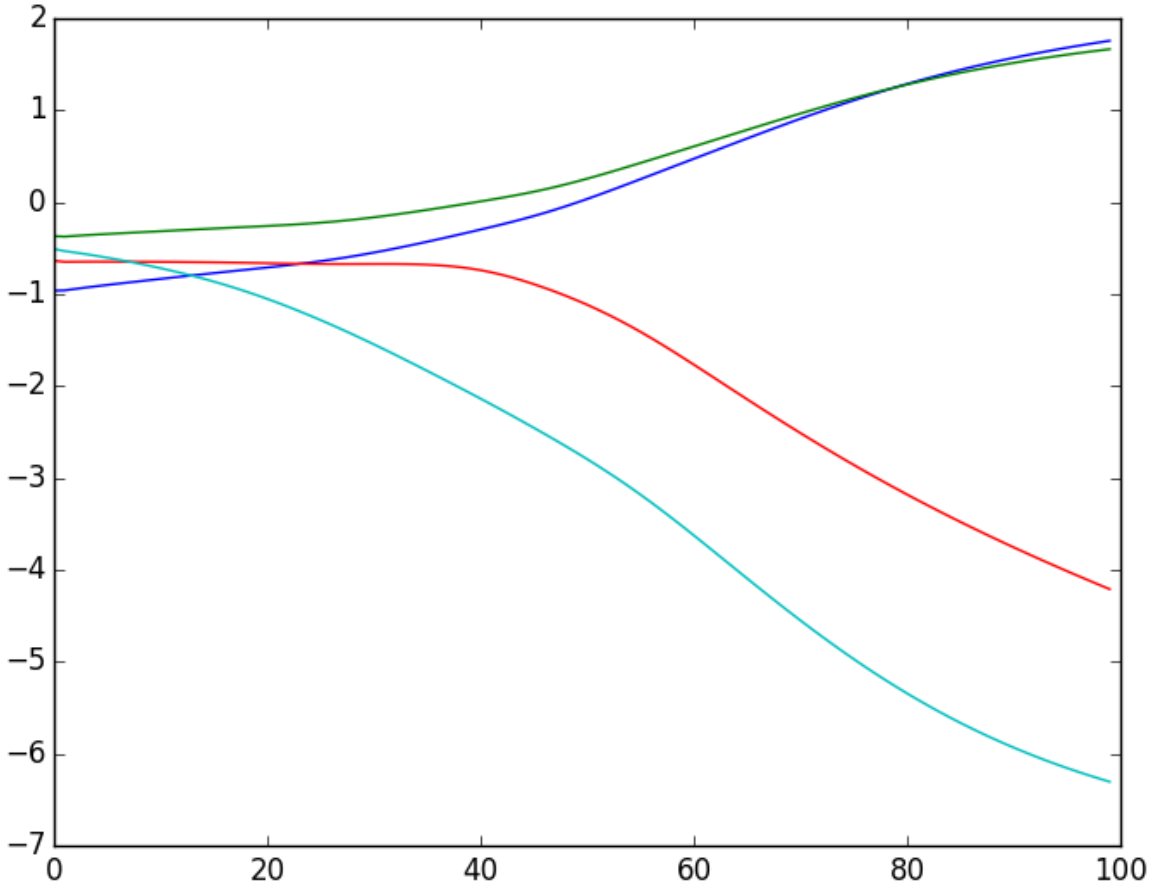

4
3
2
1
0
−1
−2
0
20
40
60
80
100

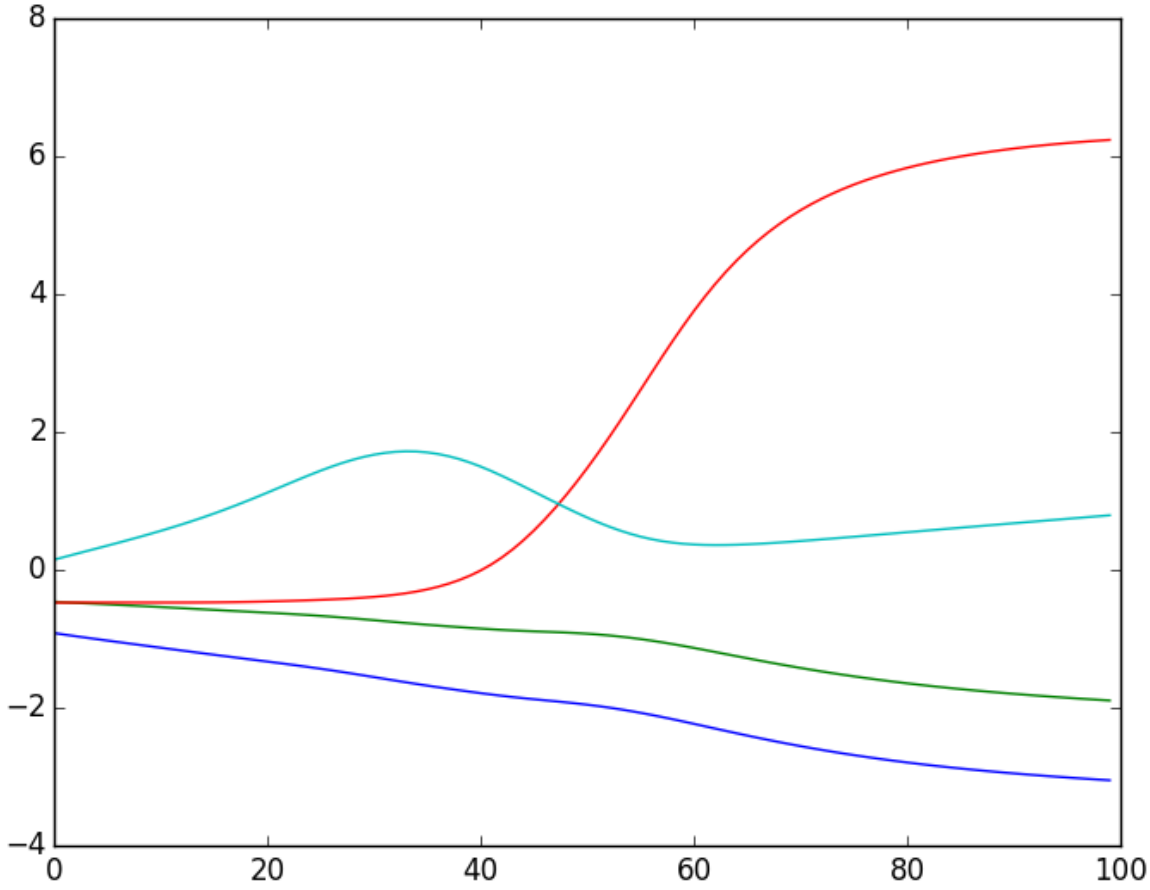

0.0
−0.5
−1.0
−1.5
−2.0
−2.5
0
20
40
60
80
100

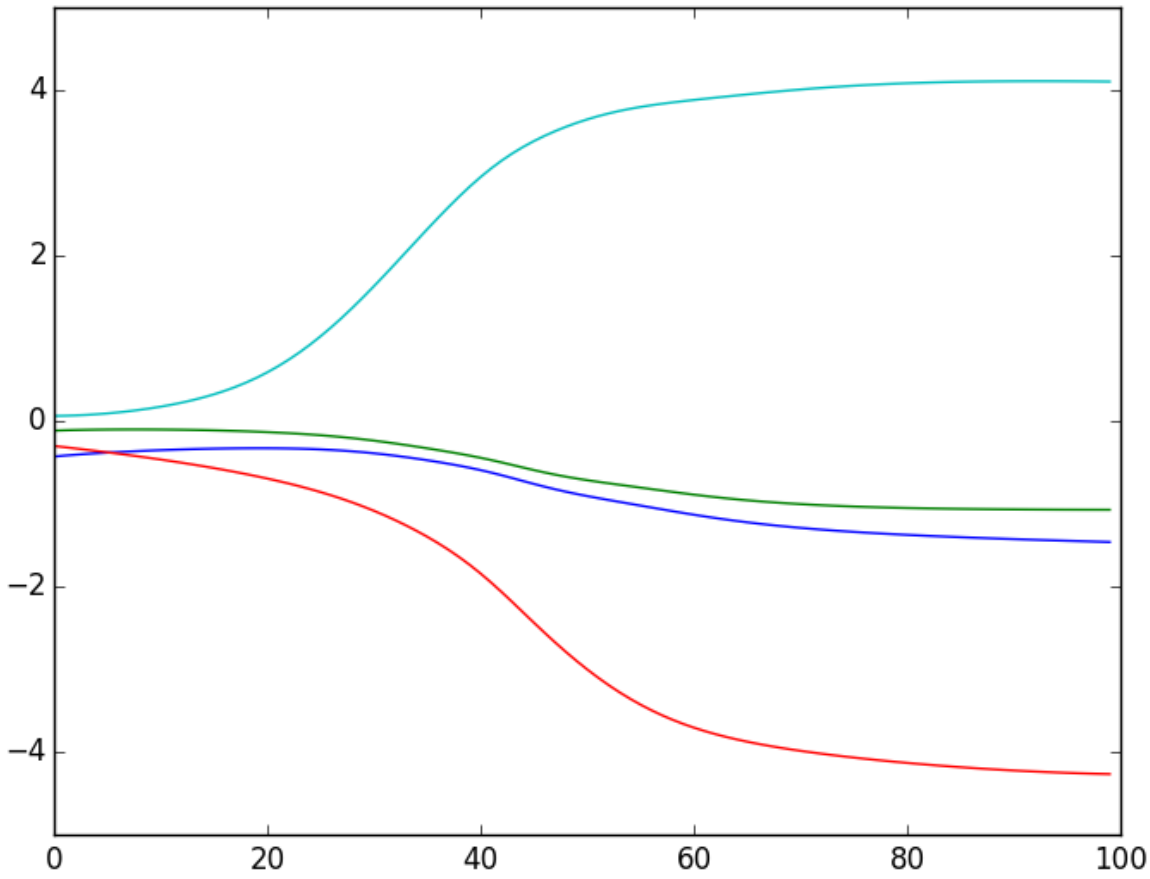

0.0
−0.2
−0.4
−0.6
−0.8
−1.0
−1.2
−1.4
−1.6
0
20
40
60
80
100

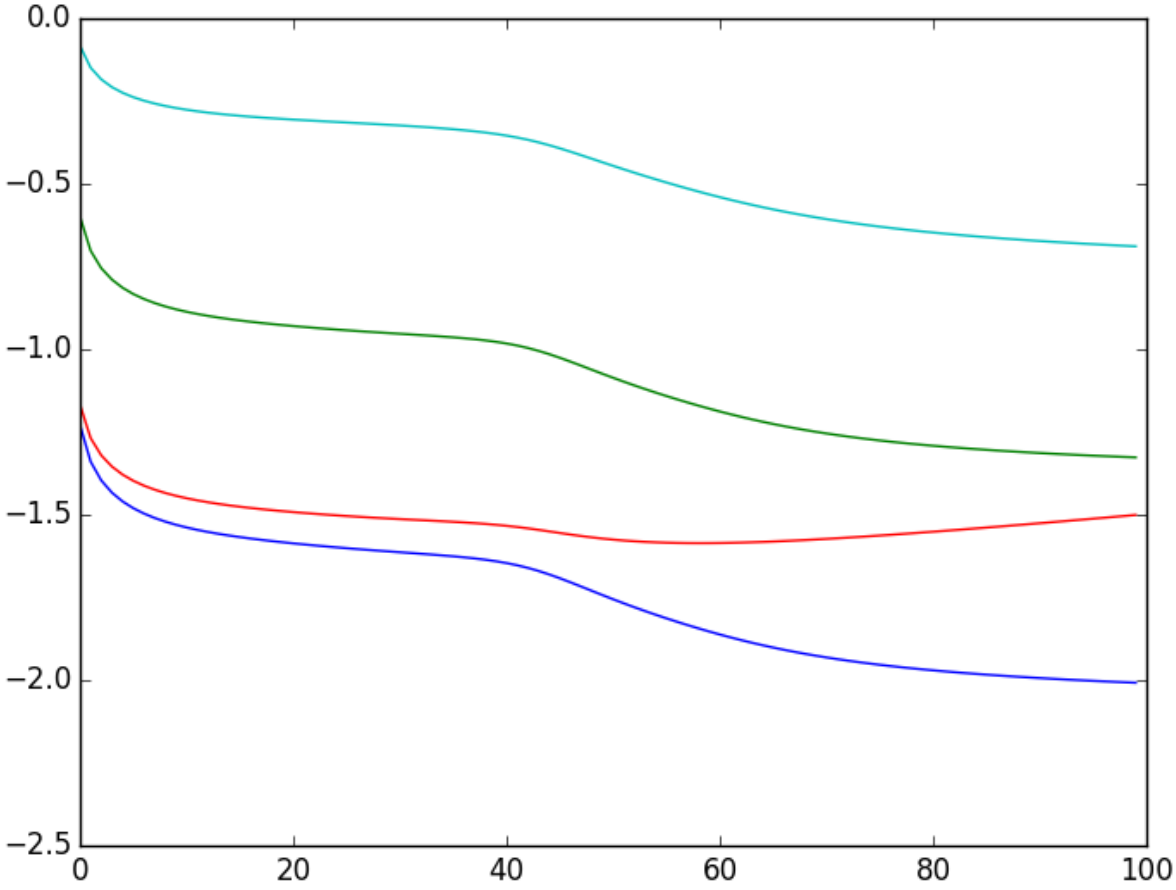

2
1
0
−1
−2
−3
−4
−5
0
20
40
60
80
100

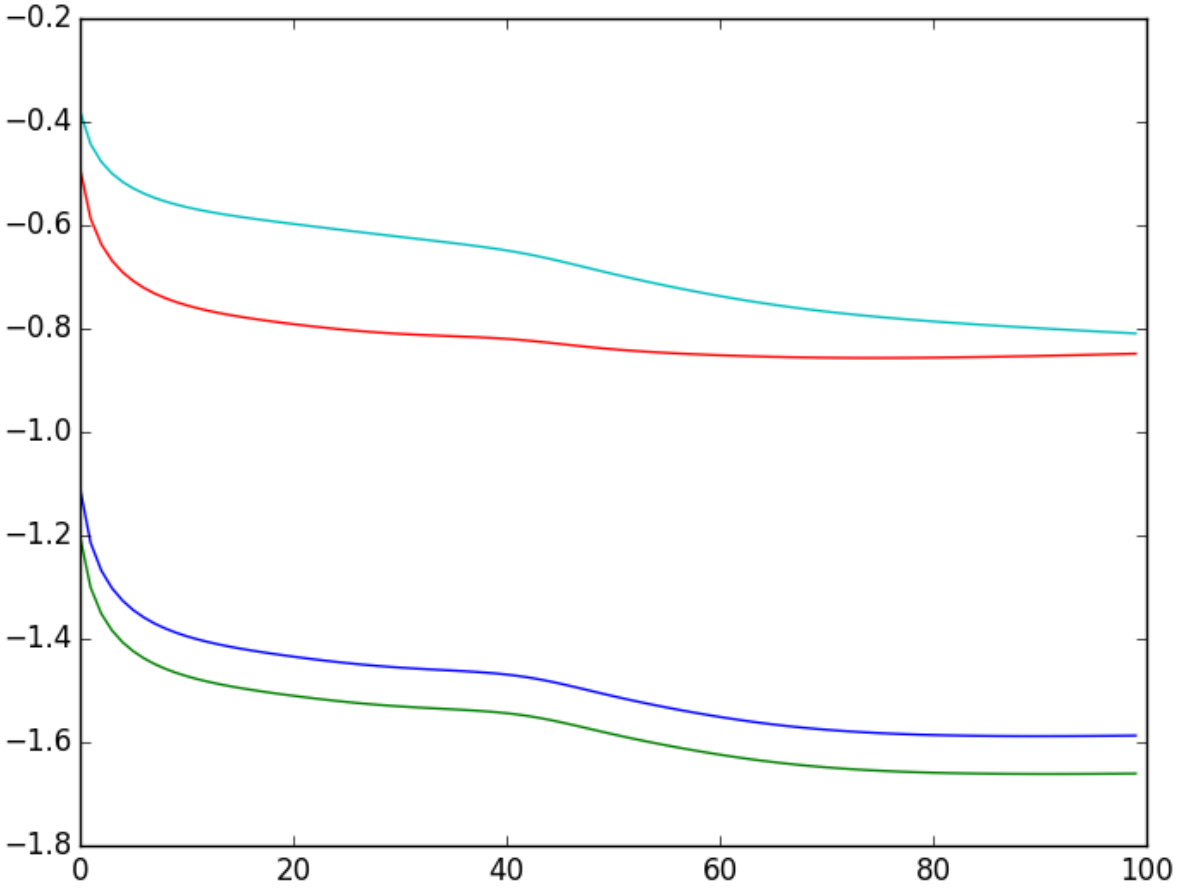

0.0
−0.5
−1.0
−1.5
−2.0
0
20
40
60
80
100


2
1
0
−1
−2
−3
−4
−5
−6
−7
0
20
40
60
80
100


8
6
4
2
0
−2
−4
0
20
40
60
80
100


4
2
0
−2
−4
0
20
40
60
80
100


0.0
−0.5
−1.0
−1.5
−2.0
−2.5
0
20
40
60
80
100



I note here that the weights are:

weight1:

```
[[  1.47489876  -2.5760843   -3.0511345 ]
 [  0.92928698   1.51280543   0.32040281]
 [  0.43107114   1.03736064  -1.39691127]
 [  0.43427068   2.1810017   10.89102681]
 [  0.93946702   5.87792112  -1.10105519]
 [  1.30533365  -1.84003067  -0.44430932]
 [  0.77957584   0.74091862  -0.89458746]
 [  0.94392369   0.65787414  -0.93467616]
 [  1.32887916  -5.18711929   0.97513235]]
```

weight2:

```
[[-1.34283058 -1.75804513 -1.38186887 -0.64912913 -0.66223321  1.7567395
  -3.05225922 -1.46283067 -2.00783507 -1.58722903]
 [-1.93441049 -2.3005684  -1.15522497  0.04901775 -1.30674412  1.6650663
  -1.89516148 -1.07339441 -1.32672405 -1.66067473]
 [-0.9318083  -1.80285629 -0.04339187  1.0246678  -0.02545869 -4.21008171
   6.23713396 -4.26840735 -1.50077194 -0.84850747]
 [ 3.80087954 -0.69545278 -1.38639453 -4.34465927 -1.89964133 -6.30529267
   0.79157649  4.10322208 -0.68887813 -0.80941472]]
```

After the plot, I will use the training weights to estimate the test dataset. The test error is: 0.487674704055.

```
The error for test set is

0.487674704055

Process finished with exit code 0
```

Question 2

In the second question, I just use all the dataset instead of the 65% training dataset. This can be easily achieved, as I mentioned before, modify the train flag to false in the ann_regreesion_one function.

The result is:

```
The training error finally is

[ 0.30139998]


And the weight from input layer to hidden layer is

[[ 1.12312856  1.83819241 -5.20152711]
 [ 0.90850163 -0.58366865  2.16095429]
 [ 0.27704235  1.76491252  0.8995626 ]
 [ 0.68995951 -8.39525458  8.34058917]
 [ 0.72827571  4.90316898  4.49860979]
 [ 1.15609703  0.3255862  -2.34573739]
 [ 0.76935208  0.93227497  0.1800204 ]
 [ 0.62876936  1.68298978  2.01905257]
 [ 1.43756742 -4.40844522 -5.5040665 ]]


And the weight from hidden layer to output layer is

[[ 0.14473877 -1.89161456 -1.91198838 -1.97637565 -1.39433423 -0.35697903
  -4.67130972  0.34981622 -2.24884422 -1.6288231 ]
 [-0.63473401 -2.43887538 -1.69732626 -1.31702078 -2.01052486 -0.68818109
  -3.67396169  0.81870954 -1.57575424 -1.71318124]
 [-4.08521188 -1.80523957 -0.46684898  1.20378674  1.01123933  5.77149094
   3.51811301 -4.65729692 -1.56868599 -0.61341543]
 [ 2.12801216 -0.6657506  -0.15557277 -0.68176488 -0.47922277 -8.79039963
   6.86029547 -2.07155283 -0.4582872  -0.72880463]]

Process finished with exit code 0
```

This error is a little smaller than data with only 65% training. The next is weight1, and last one is weight2.

The activation function is:

$$a2 = sigmoid(weight1 * a1) = \frac{1}{1 + e^{-(weight1 * a1)}}$$

$$a3 = sigmoid(weight2 * a2) = \frac{1}{1 + e^{-(weight2 * a2)}}$$

we can substitute weight1, weight2, a1, a2 to these activation functions.

Question 3

I first provide the hand written snapshot:

Zhen Zhang.

2. In this question, I will provide both the hand-written calculations and the code results.

① hand-written:

First step is to initialize w_1 & w_2:

w_1:
$$\begin{pmatrix} 0.5989859 & 0.61570429 & 0.05917721 \\ 0.75031718 & 0.948\ldots095 & 0.53467906 \\ -0.19255602 & 0.752\ldots591 & 0.00731897 \\ 0.3282561 & 0.91760635 & 0.58836733 \\ 0.85519028 & 0.60467\ldots05 & 0.8\ldots57815 \\ 0.87948352 & 0.3209863\ldots & 0.12295477 \\ 0.72130334 & 0.44034709 & 0.12673571 \\ 0.58982365 & 0.03606831 & 0.20018212 \\ 0.78830119 & 0.01209676 & 0.3033456 \end{pmatrix}$$

w_2:
$$\begin{pmatrix} 0.02137596 & 0.3716\ldots458 & 0.4815\ldots628 & 0.7810\ldots4582 \\ 0.99748499 & 0.\ldots44448 & 0.44\ldots7296 & 0.9041\ldots0299 \\ 0.58202998 & 0.58454727 & 0.67\ldots7265 & 0.2337\ldots6869 \\ 0.29337628 & 0.6\ldots584613 & 0.448\ldots73726 & 0.1766\ldots7889 \\ 0.92894946 & 0.07\ldots94722 & 0.7043134 & 0.3910\ldots99112 \\ 0.50711942 & 0.97\ldots02332 & 0.6816\ldots4534 & 0.3205\ldots6465 \\ 0.45469223 & 0.7530\ldots7066 & 0.6971\ldots4853 & 0.8154\ldots7693 \\ 0.58787151 & 0.8061\ldots6316 & 0.6186\ldots0074 & 0.6135\ldots2579 \\ 0.26413764 & 0.7516\ldots6404 & 0.1509\ldots8073 & 0.7600\ldots0175 \\ 0.3052883 & 0.0800\ldots6098 & 0.7608\ldots0246 & 0.427\ldots6651 \end{pmatrix}$$

and the input $a_1 = (1, 0.69, 0.5, 0.49, 0.42, 0.5, 0, 0.49, 0.22)$.

$y = (0,0,0,0,0,0,1,0,0,0)$.

Next is the foward propagation part:

$z_2 = a_1 \cdot w_1 = (2.63518918, 2.53084912, 1.29185073)$

and $a_2 = \frac{1}{1+e^{-z_2}} = (0.93309224, 0.92627636, 0.78446028)$

after give it dimension 0: $a_2 = (1, 0.93309224, 0.92627636, 0.78446028)$

$z_3 = a_2 w_2 = (1.42704727, 2.34762673, 1.93274427, 1.49691172, 1.9552743, 2.29609194,$
$2.44283808, 2.39437804, 1.70152855, 1.42019219)$

and $a_3 = \frac{1}{1+e^{-z_3}} = (0.80644083, 0.9127454, 0.87366327, 0.81711424, 0.87602052,$
$0.90855286, 0.92003613, 0.9163976, 0.84573427, 0.80536854)$

Now backward feed:

$\delta_3 = -(y_0 - a_3) a_3 (1-a_3) = (-0.13588059, -0.07269217, -0.09643105, -0.12240838, -0.09514334,$
$-0.07548672, 0.00588291, -0.07020801, -0.11034111, -0.12624156)$

Then w_2 is $w_2 = w_2 - \delta_3 \cdot a_2$.

$w_2 =$

0.10087879	0.35990876	0.47003626	0.17117089
0.99021578	0.23766515	0.43893965	0.89840057
0.57238685	0.57554934	0.66354045	0.12762404
0.28119654	0.68445223	0.43742665	0.16709918
0.91943513	0.06306947	0.69555005	0.38363551
0.49957075	0.96403971	0.67465318	0.31464302
0.45552852	0.25361959	0.69769345	0.81593842
0.58085071	0.79861211	0.61209754	0.60801836
0.25310353	0.74134456	0.1407601	0.75134593
0.2926414	0.06828148	0.749109	0.41776195

$\delta_2 = \left(\sum_{j=1}^{10} w_{2j} \cdot \delta_{3j}\right) \cdot (1-a_2) \cdot a_2 = (-0.02638086, -0.03240595, -0.07412227)$

and $w_1 = w_1 - \delta_2 \cdot a_1 =$

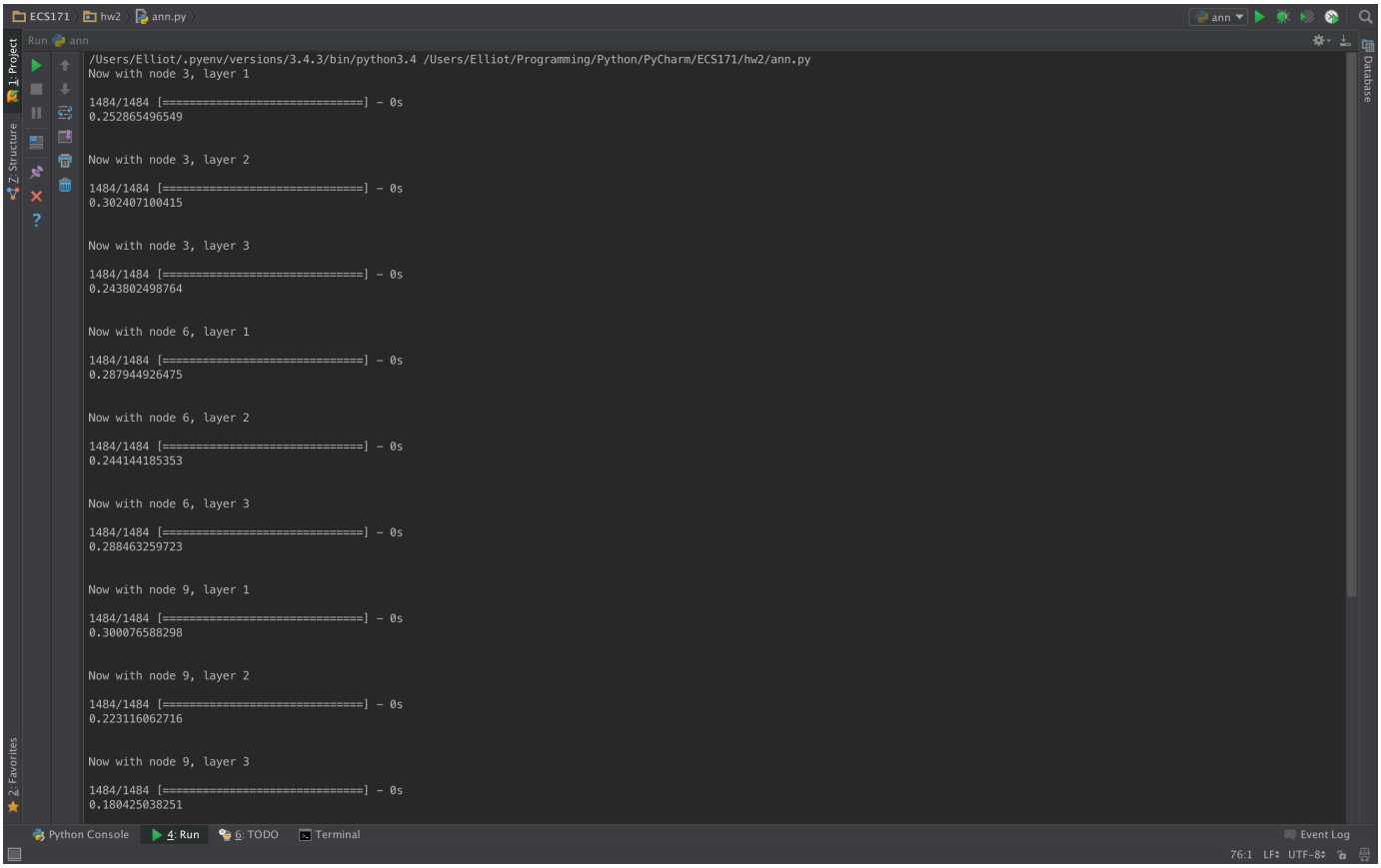
0.59634726	0.61746869	0.05176474
0.74849691	0.94597349	0.52956459
0.19123698	0.75130562	0.00361283
0.32896344	0.91601846	0.58473532
0.85408229	0.6033111	0.8194405
0.87816448	0.31936608	0.11924864
0.72130334	0.44034409	0.12673571
0.58853099	0.03448042	0.1965501
0.78772081	0.01138383	0.30171497

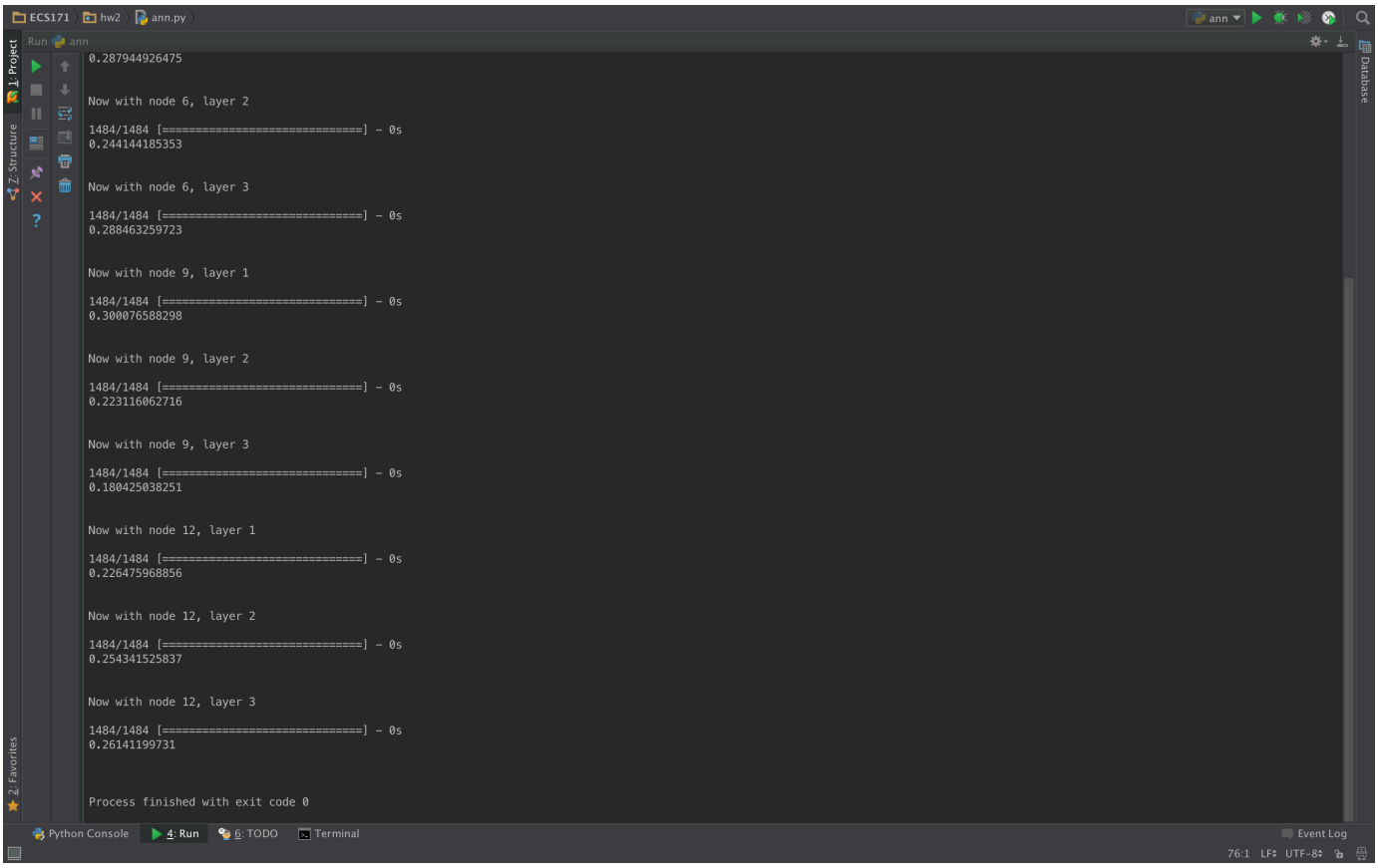
And here is the algorithm output in main_three():

```
The first weight is

[[ 0.5963472   0.61246369  0.05176494]
 [ 0.74849691  0.94597349  0.52956459]
 [ 0.19123698  0.75130562  0.00361283]
 [ 0.32696344  0.91601846  0.58473532]
 [ 0.85408229  0.603311    0.819465  ]
 [ 0.87816448  0.31936608  0.11924864]
 [ 0.72130334  0.44034709  0.12673571]
 [ 0.58853099  0.03448042  0.1965501 ]
 [ 0.78772081  0.01138383  0.30171497]]


The second weight is

[[ 0.0087879   0.99021578  0.57238685  0.28116544  0.91943513  0.49957075
   0.45528052  0.58085071  0.25310353  0.29266414]
 [ 0.35990876  0.23766515  0.57554934  0.6844523   0.06306947  0.96403971
   0.75361959  0.79961211  0.74134456  0.06828148]
 [ 0.47003626  0.43893965  0.66354045  0.43742665  0.6955005   0.67465318
   0.69769345  0.61209754  0.1407601   0.749109  ]
 [ 0.77117099  0.89840057  0.22620404  0.16709998  0.38363551  0.31464302
   0.81593842  0.60801836  0.75134593  0.41776195]]


The output here is

[[ 0.80644083  0.9127454   0.87366327  0.81711424  0.87602052  0.90855286
   0.92003613  0.9163976   0.84573427  0.80536854]]

Process finished with exit code 0
```

Question 4

In this problem, I use the package of Keras in python. It provides a simple and user-friendly interface, which depends on numpy and theano. The function multiple_layer_node is a closure that implements this function by providing node, and its inner function will provide the layer. Here I will first show the snapshot and then give a summary table:





Now summary them into a table:

layer/node	3	6	9	12
1	0.253	0.288	0.300	0.226
2	0.302	0.244	0.223	0.254
3	0.244	0.288	0.18	0.261

From the table, we can see that in layer 2 or 3, with the increase of node number, the error rate first decreases then increases, this may be a sign of overfit when increasing. The node 9 is best in layer 2 or layer 3. Also, the error of layer 3 is the best compared with layer 1 or 2, meaning it is a little accurate when adding layers, but also attention should be paid to overfit.

Node 9 with layer 3 is the best model to choose.

Question 5

Here by calling the main_five() function, it calls the multiple_layer_node by supplying the best model (node 9, layer 3) found in the previous problem, and the result is:

```
1484/1484 [==============================] - 0s
0.311492978059


1/1 [==============================] - 0s
[[ 0.46714997  0.66111314  0.5212877   0.48663355  0.47958467  0.42406736
   0.43033514  0.74806002  0.72312129  0.60518595]]

Process finished with exit code 0
```

Since 0.74806002 is the largest compared to others, so it belongs to the eighth class, which is 'NUC'.

Question 6

Let me talk about the method I used in question 1 - 5 when handling the response variable (y).

There are ten classes for the response value, I encode them to ten 0-1 value sets. It means, for example, if one observation belongs to the seventh class, the response

variable after encoding will be (0,0,0,0,0,0,1,0,0,0). The predicted value with the largest value will be the classified class. For example, if the output is (0.1,0.1,0.1,0.9,0.1,0.1,0.1,0.1,0.1,0.1), the class will be the fourth one. I have also considered to use 0-9 as the response value, but it is not a good idea, since this method mis-consider the ratio effect and interval effect. They are basic principles in statistics, which makes this approach invalid.

Also mention here, the RSS is the measure of error:

$$RSS = \frac{1}{2}\sum_{j=1}^{n}\sum_{i=1}^{m}(y_{ij} - a_{ij})^2$$

When measuring uncertainty, since the output is a value between 0 and 1, so the larger the value, the more certain it should belong to the class, and the less the value, the less certain it should belong to that class.