

Deep Neural Networks

Neural Networks: ECE 5930

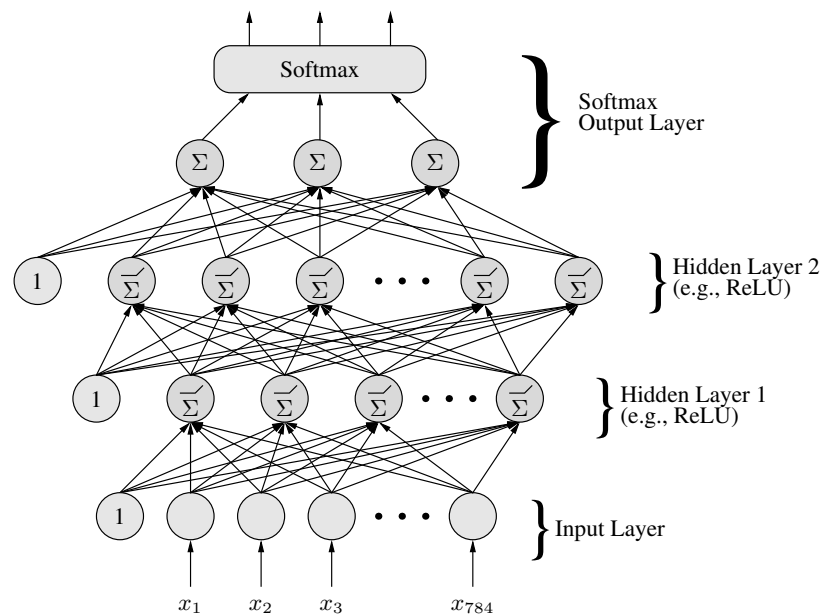


Figure: Two Hidden Layer Neural Network

Clint Ferrin
Utah State University
Mon Oct 23, 2017

Table of Contents

1	Summary	1
2	Program Description	1
3	Two-class Classifier	3
3.1	Increasing Network Complexity	4
3.2	Comparing a Neural Network to Other Classifiers	6
4	Ten-class Classifier	8
4.1	Increasing the Complexity of the MNIST Neural Network	9
5	Appendix	10
5.1	ten-class-classifier.py	10
5.2	two-class-classifier.py	16

List of Figures

1	10 Digits from the MNIST Data-set	1
2	Trained Output for 80% Training Data, 20% Testing Data with 0.0 Momentum	4
3	Trained Output for 80% Training Data, 20% Testing Data with 0.8 Momentum	4
4	Three Miss-classified Points	5
5	Trained Output for 80% Training Data, 20% Testing Data with 0.8 Momentum and an Additional Layer with 10 Neurons	5
6	Comparison of Two-class Classifier with 0.0 Momentum	7
7	Comparison of Two-class Classifier with 0.8 Momentum and an Additional Layer	7
8	Comparison of Differing Momentum with a Single Hidden Layer	8
9	Comparing the Mean Square Error with Momentum 0.0 and 0.8	9
10	Comparing the Mean Square Error with Momentum 0.0 and 0.8 with an Additional Layer	9

List of Tables

1	Binary Classifier Performance Comparison	6
2	Comparison of Bayes Optimal, 15-Nearest Neighbor, and Neural Network	7

Listings

1	Network Initialization	2
2	Back propagation	2
3	Creating 80% 20% Data	3
4	Output Accuracy on Test Data from Networks	5
5	Output Accuracy and Run-time on MNIST Test Data	9
6	Ten-class Classifier	10
7	Two-class Classifier	16

1 Summary

Neural Networks have applications in image recognition, data compression, and even stock market prediction. The basic concept behind Neural Networks is depicted on the main figure of the title page. This paper presents the basic structure for machine learning on classified data using a randomly generated data-set (2 classes), and the MNIST data-set (10 classes).

The MNIST data-set consists of 70,000 small images of digits 0-9 handwritten by high school students and employees of the US Census Bureau. Each image is 28×28 pixels so that when the image is vectorized it has a dimension 1×784 . The MNIST data-set is ideal for machine learning because of the variable nature of handwriting and the limited numbers of classes.

Ten numbers from the data-set can be seen in Figure 1.

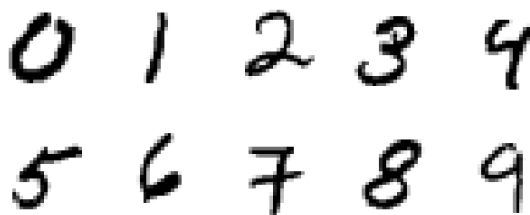


Figure 1: 10 Digits from the MNIST Data-set

The remainder of the paper will be dedicated to analyzing how effective Neural Networks are at correctly identifying different classes of data such as the MNIST as seen in Figure 1.

2 Program Description

The neural network class that I wrote in PYTHON can have any number of layers, neurons, and any type of activation function passed to it for an n dimensional input with k number of classes. The Network is initialized by specifying `num_inputs`, `num_outputs`, `batch_size`, and `epochs`.

In testing the different classes in this paper, the ReLU (Rectified Linear Unit) was used for the majority of the classification problems.

The desired data is read in, and any activation functions are defined for the different layers. As seen in the heading `# input layer`, the layers are created by passing the number of inputs that they will receive, the number of desired neurons, and the type of activation function. The final layer seen in the code snippet below does not have an activation function because by default softmax is run on the output of the network. In this way, the output option can easily be changed between the softmax and sigmoid functions.

Additional parameters exist for the initialization of the network, but they are optional parameters. Such variables include the momentum β , step size η , and regularization reg as seen in the initialization of the Neural Network in listing 1.

Listing 1: Network Initialization

```

1  # Clint Ferrin
2  # Oct 12, 2017
3  # Neural Network Classifier
4
5  import matplotlib.pyplot as plt
6  import numpy as np
7  import pickle
8  from tensorflow.examples.tutorials.mnist import input_data
9  import time
10
11 def main():
12     num_inputs = 784
13     num_outputs = 10
14     batch_size = 100
15     epochs = 10
16     mse_freq = 50
17
18     # open mnist data
19     X, Y, X_test, Y_test = get_mnist_train("./data")
20
21     # initialize activation functions
22     relu = activation_function(relu_func, relu_der)
23     sig = activation_function(sigmoid_func, sigmoid_der)
24     no_activation = activation_function(return_value, return_value)
25
26     num_neurons = 300
27     # two hidden layers
28     layers1 = [layer(num_inputs, num_neurons, relu)]
29     layers1.append(layer(num_neurons, 100, sig))
30     layers1.append(layer(100, num_outputs, no_activation))
31
32     # create neural network
33     network = NeuralNetwork(layers, eta=0.9, momentum=0.8, softmax=True)
34
35     # train network
36     network.train_network(X, Y, batch_size=batch_size,
37                           epochs=epochs, MSE_freq=mse_freq, reg=0.01)

```

For a full view of the different classes, such as the class `NeuralNetwork`, `layer` and `activation_function` used in the `NeuralNetwork` class, see Section 5.

The Training of the system is done using back propagation with gradient decent and mini-batches. Mini-batches are described in Section 4, but I will explain part of the back propagation in the code.

The code uses back propagation as seen in Listing 2. After forward propagation, the list of layers is reversed to traverse and solve using gradient decent. First the program finds the derivative of the difference squared to pass on to the last layers. Note that the softmax derivative has many forms, but my program had the most success using the form outlined on the website CS321n: Convolution Neural Networks for Visual Recognition.

Listing 2: Back propagation

```

1  def train_data(self, X, Y):
2      Yhat = self.forward_prop(X)
3      dE_dH = (Yhat-Y).T
4
5      # back propagation
6      if self.softmax is True:
7          self.layers[-1].output = np.ones(dE_dH.shape).T
8
9      for layer in self.layers[::-1]:
10         dE_dNet = layer.der(layer.output).T*dE_dH
11
12         # divide by number of samples in batch to regularize step
13         dE_dWeight = (np.dot(dE_dNet, layer.weight_der)) / \
14             layer.weight_der.shape[0]
15
16         # obtain multiplication to pass back to next layer

```

```

17         dE_dH = np.dot(layer.W[:,0:-1].T,dE_dNet) * self.reg
18
19         # update weight matrix with momentum
20         layer.W += -layer.momentum_matrix
21         layer.momentum_matrix = \
22             self.momentum * layer.momentum_matrix + \
23             self.eta * dE_dWeight
24
25         # create long entire list of errors to plot at specific frequency later
26         for indx,yhat in enumerate(Yhat):
27             self.error_array.append(sum((Y[indx]-yhat)*(Y[indx]-yhat)))

```

3 Two-class Classifier

The data set from `classasgntrain1.dat` is a grouping of data centered around 10 different means with a Gaussian Distribution for each class. I split the data into 80% training data and 20% testing data using the function seen in the listing below:

Listing 3: Creating 80% 20% Data

```

1 class data_frame:
2     def __init__(self, data0, data1):
3         self.x0 = data0
4         self.x1 = data1
5         self.xtot = np.r_[self.x0,self.x1]
6         self.N0 = self.x0.shape[0]
7         self.N1 = self.x1.shape[0]
8         self.N = self.N0 + self.N1
9         self.xlim = [np.min(self.xtot[:,0]),np.max(self.xtot[:,0])]
10        self.ylim = [np.min(self.xtot[:,1]),np.max(self.xtot[:,1])]
11        class_x0 = np.c_[np.zeros([self.N0,1]),np.ones([self.N0,1])]
12        class_x1 = np.c_[np.ones([self.N1,1]),np.zeros([self.N1,1])]
13        self.class_tot = np.r_[class_x0,class_x1]
14        self.y = np.r_[np.ones([self.N0,1]),np.zeros([self.N1,1])]
15
16        # create a training set from the classasgntrain1.dat (80% and 20%)
17        self.train_x0 = data0[0:80]
18        self.train_x1 = data1[0:80]
19        self.train_tot = np.r_[data0[0:80],data1[0:80]]
20        self.train_class_tot = np.r_[self.class_tot[0:80],self.class_tot[100:180]]
21        self.test_data = np.r_[data0[80:100],data1[80:100]]
22        self.test_class_tot = np.r_[self.class_tot[80:100],self.class_tot[180:200]]
23
24    def get_classasgn_80_20():
25        data = np.loadtxt("./data/classasgntrain1.dat",dtype=float)
26        x0 = data[:,0:2]
27        x1 = data[:,2:4]
28        data = data_frame(x0,x1)
29        return data.train_tot,data.train_class_tot,data.test_data,data.test_class_tot

```

The network was trained using sigmoid functions, and it produced the output seen in Figure 2. Note that the step size was increased to 0.4, 0.7, and 0.9 with colors blue, red, and green respectively.

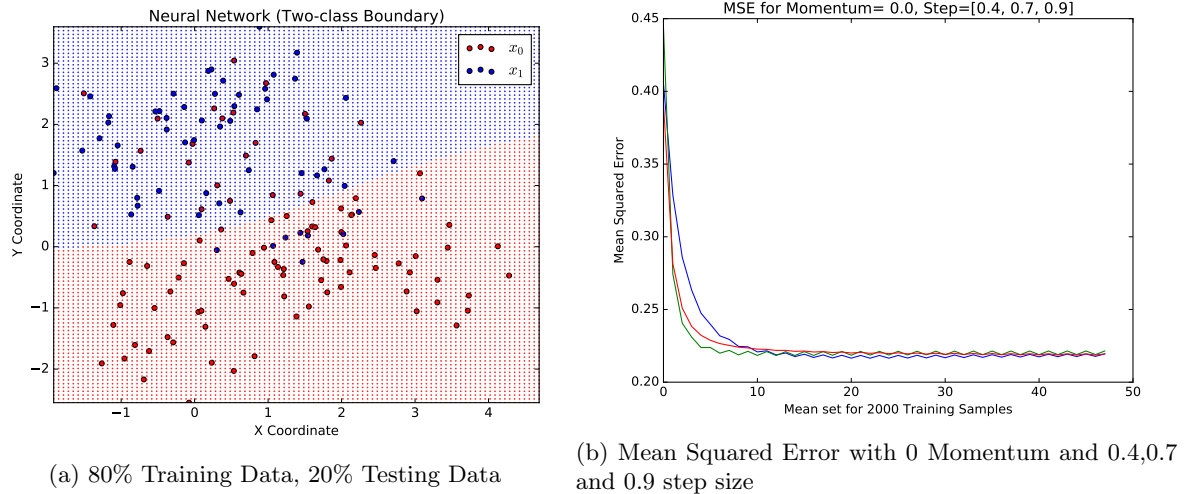


Figure 2: Trained Output for 80% Training Data, 20% Testing Data with 0.0 Momentum

I ran the same batch of data with 0.8 momentum and received the following results for the plot and MSE seen in Figure 3.

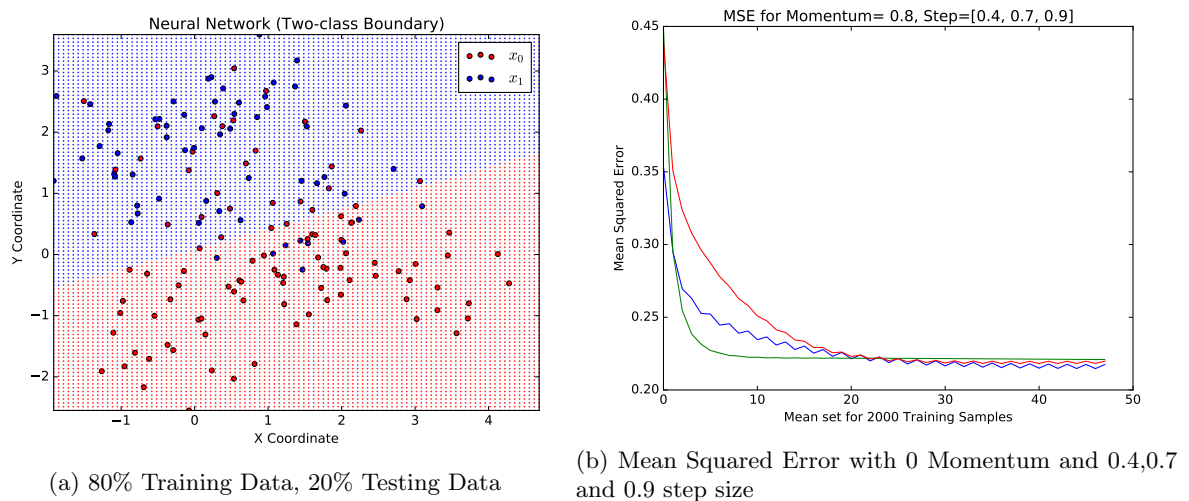


Figure 3: Trained Output for 80% Training Data, 20% Testing Data with 0.8 Momentum

3.1 Increasing Network Complexity

To increase the complexity, I introduced more neurons by making a layer that had 5 neurons connected to sigmoid functions, and 10 more neurons with a sigmoid functions that converged to a sigmoid output.

The increased complexity did not increase the accuracy in this case because the three points that were miss-classified seemed to be far from the other data as seen in Figure 4. It did increase the accuracy in the test data described in Section 3.2, and it did produce a new plot of MSE as seen below. Note that Figure 5 does not converge as fast as the other plots due to the increased complexity.

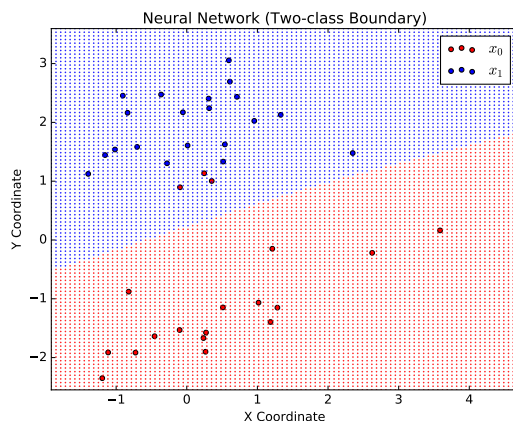
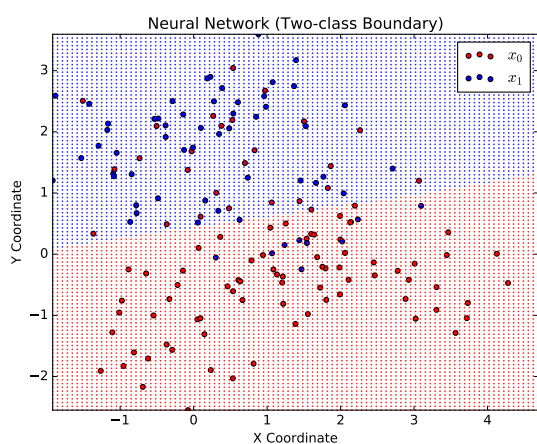
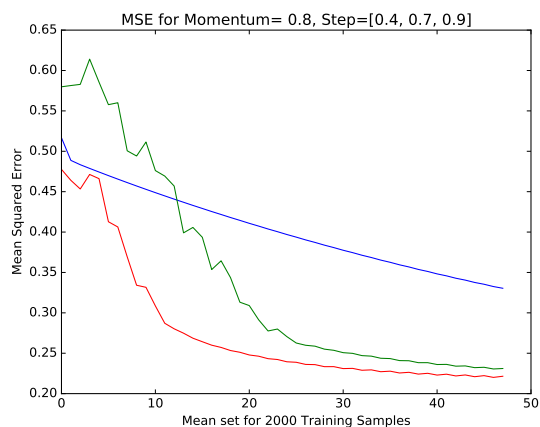


Figure 4: Three Miss-classified Points



(a) 80% Training Data, 20% Testing Data



(b) Mean Squared Error with 0 Momentum and 0.4, 0.7 and 0.9 step size

Figure 5: Trained Output for 80% Training Data, 20% Testing Data with 0.8 Momentum and an Additional Layer with 10 Neurons

The resulting listing from the program showed that it correctly classified the small batch of test data with only 3 mistakes with a 92.5% accuracy in each case because there were sufficient testing samples. The following listing shows how each layer performed with varying step sizes.

Listing 4: Output Accuracy on Test Data from Networks

```

1 Two class layer with 1 hidden network (5 neurons). Epochs
2 mo-0.0-eta-0.4
3 Percent Correct: 92.5%
4 Run-time: 0.560807943344 seconds
5
6 mo-0.0-eta-0.7
7 Percent Correct: 92.5%
8 Run-time: 0.569748878479 seconds
9
10 mo-0.0-eta-0.9
11 Percent Correct: 92.5%
12 Run-time: 0.778621912003 seconds
13
14 mo-0.8-eta-0.4
15 Percent Correct: 92.5%
16 Run-time: 0.594919204712 seconds
17

```

```

18 mo-0.8-eta-0.7
19 Percent Correct: 92.5%
20 Run-time: 0.717699050903 seconds
21
22 mo-0.8-eta-0.9
23 Percent Correct: 92.5%
24 Run-time: 0.587964057922 seconds
25
26
27 Two class layer with 2 hidden networks (5 and 10 neurons respectively).
28 mo-0.0-eta-0.4
29 Percent Correct: 92.5%
30 Run-time: 0.685973882675 seconds
31
32 mo-0.0-eta-0.7
33 Percent Correct: 92.5%
34 Run-time: 0.68302488327 seconds
35
36 mo-0.0-eta-0.9
37 Percent Correct: 92.5%
38 Run-time: 0.728396892548 seconds
39
40 mo-0.8-eta-0.4
41 Percent Correct: 92.5%
42 Run-time: 0.695672035217 seconds
43
44 mo-0.8-eta-0.7
45 Percent Correct: 92.5%
46 Run-time: 0.681930780411 seconds
47
48 mo-0.8-eta-0.9
49 Percent Correct: 92.5%
50 Run-time: 0.666880846024 seconds

```

3.2 Comparing a Neural Network to Other Classifiers

In previous processing, I found that other classification methods performed with following errors in percent as seen in Table 1. Note that the Bayes Optimal Classifier performed the best because it knew the true distribution of the data.

Method	train+run time	Errors in %	
		Training	Test
Linear Regression	1.23s	14.5	20.49
Quadratic Regression	1.70s	14.5	20.44
Linear Discriminant Analysis	2.49s	15.0	19.98
Quadratic Discriminant Analysis	3.26s	14.5	20.23
Logistic Regression	2.00s	14.0	20.00
1-Nearest Neighbor	35.02s	00.0	21.83
5-Nearest Neighbor	37.92s	12.0	20.29
15-Nearest Neighbor	36.47s	16.0	19.25
Bayes Naive	1.22s	14.0	20.04
Bayes Optimal Classifier	0.20s	14.0	19.14

Table 1: Binary Classifier Performance Comparison

To compare the Neural Network with the other classifiers, I used `classasgntrain1.dat` to train all data points from the data set and tested it on 20000 additional randomly generated data points to simulate the same test performed in the other linear classifiers. The results of this test can be seen in 0 momentum test in Figure 6. Again, blue corresponds to 0.4, red to 0.7, and green to 0.9.

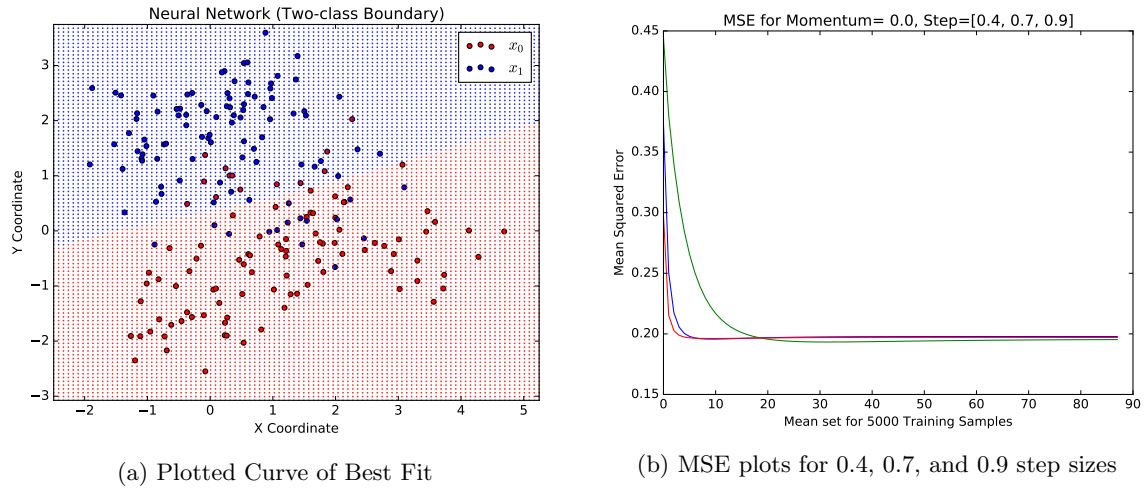


Figure 6: Comparison of Two-class Classifier with 0.0 Momentum

After adding the new complexity of a new layer in this case, the best result came from the Layer with 0.8 momentum and a step size of 0.9. The graph of the results and the corresponding MSE plot can be seen in Figure 7.

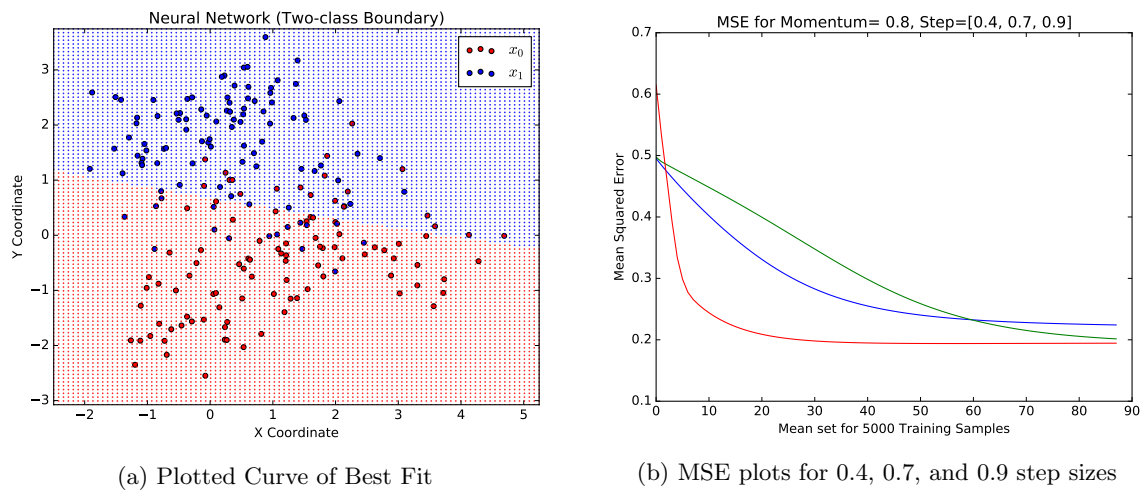


Figure 7: Comparison of Two-class Classifier with 0.8 Momentum and an Additional Layer

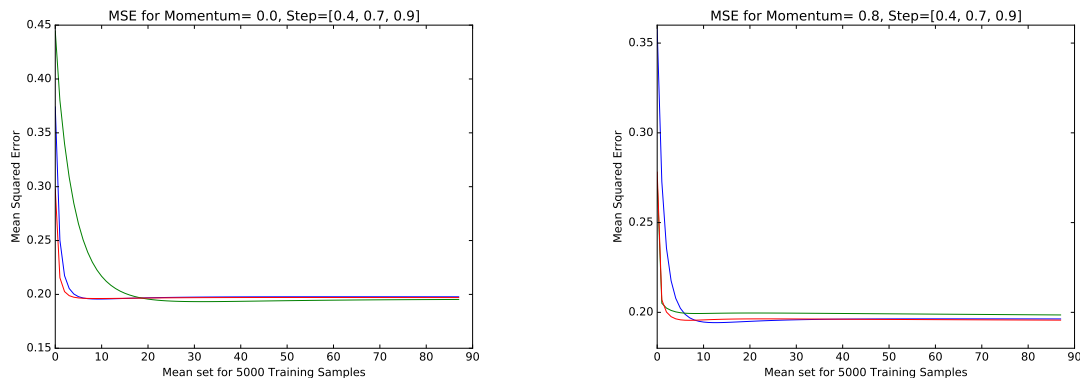
The Neural Network returned an error of 19.805%, which puts its results just behind the Bayes Optimal Classifier and the k-nearest neighbor approach. Because the 15-Nearest Neighbor is not practical with large datasets, and because a model for the Bayes Optimal Classifier is often impossible to find, the Neural Network is one of the most viable options to classify data in this data set.

Method	train+run time	Errors in %	
		Training	Test
Bayes Optimal Classifier	0.20s	14.0	19.14
15-Nearest Neighbor	36.47s	16.0	19.25
NN with 5 N sig, 10 N sig, $\beta = 0.8$, $\eta = 0.9$	2.78s	14.0	19.805

Table 2: Comparison of Bayes Optimal, 15-Nearest Neighbor, and Neural Network

It is also important to note that the momentum term has a significant effect on the speed at which the

Mean Squared Error drops. Figure 8 shows the dramatic speed difference that the momentum has on the convergence of the Mean Squared Error. Increasing the momentum to 0.8 did not have a significant effect on the percent of errors, but it did affect the number of iterations for convergence.



(a) MSE plots for 0.4, 0.7, and 0.9 step sizes with 0 momentum (b) MSE plots for 0.4, 0.7, and 0.9 step sizes with 0.8 momentum

Figure 8: Comparison of Differing Momentum with a Single Hidden Layer

4 Ten-class Classifier

To test the MNIST data set, I created a network with 784 inputs, a hidden layer with 300 neurons, and an output of 10 classes connected to a softmax. The Mean Squared Error plot was set up to report the mean of every 50 iterations. As the assignment description asked, I used a `batch-size` of 100, and plotted the MSE results.

The number of iterations using mini-batches where N is the total number of data samples and B is the size of your batch size is:

$$itrs = N/B \cdot epochs \quad (1)$$

I printed out the MSE every 50 iterations, as seen in Figure 10 for all of my programs, and I combined the MSE plots for incrementing step sizes for the same network with the same momentum. For the programs listed below, I ran my code for 30 epochs.

The output of the MSE for a single hidden layer with no momentum can now be seen in Figure 10. I graphed each increase of the step size with a new color. The blue line represents a step size of 0.4, the green line represents a step size of 0.7, and the red line represents a step size of 0.9. Note that the momentum increases the convergence of the MSE graph.

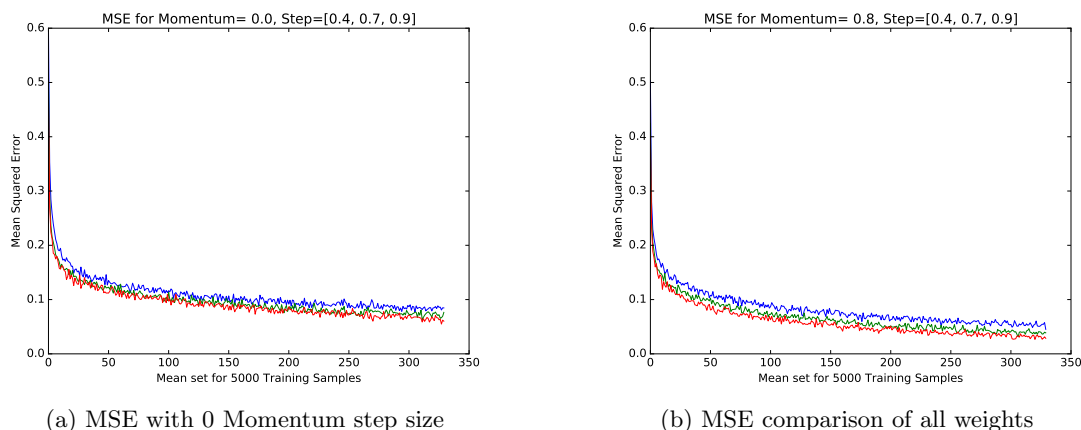


Figure 9: Comparing the Mean Square Error with Momentum 0.0 and 0.8

The two networks returned an accuracy on the test data of 94.82% (with 0 momentum and a step size of 0.9) and 96.89% (with 0.8 momentum and a step size of 0.9). The programs both took about 18 minutes to run as seen in Listing 5.

I tested several different iterations and epochs, and found that increasing the epochs to more than 60 did not have a significant effect on the output. For all of these tests, I used a total of 60 epochs.

4.1 Increasing the Complexity of the MNIST Neural Network

To hopefully increase the accuracy of the network, I created a network with two hidden layers. It has 300 neurons in the first hidden layer and 100 neurons in the second hidden layer. I tried different values of the step size parameter, plotting the MSE as a function of iteration as seen in Figure 10. I used a mini-batch of size 100 on the MNIST training data and tested it on the test data data from the MNIST database.

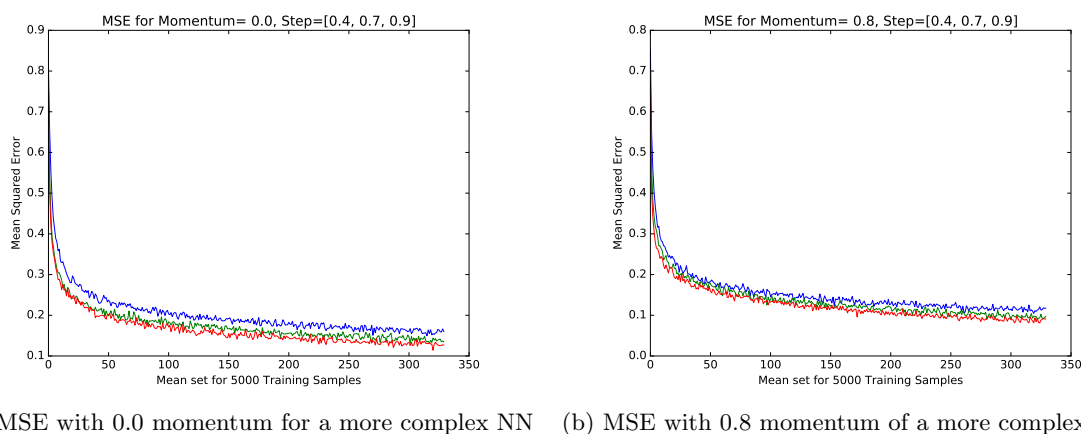


Figure 10: Comparing the Mean Square Error with Momentum 0.0 and 0.8 with an Additional Layer

For a detailed description of how long each network took in clock time (seconds) for a certain accuracy, see the listing below. Note that the increased complexity did not increase the complexity, but one reason is because I ran both for 60 epochs, and the extra complexity requires more time convergence.

Listing 5: Output Accuracy and Run-time on MNIST Test Data

```
1 Layer with 1 hidden network (300 neurons). Epochs
2 mo-0.0-eta-0.4
```

```

3 Percent Correct: 94.13%
4 Run-time: 1083.04311395 seconds
5
6 mo-0.0-eta-0.7
7 Percent Correct: 94.72%
8 Run-time: 1078.50167799 seconds
9
10 mo-0.0-eta-0.9
11 Percent Correct: 94.82%
12 Run-time: 1076.73657393 seconds
13
14 mo-0.8-eta-0.4
15 Percent Correct: 95.61%
16 Run-time: 1076.8610909 seconds
17
18 mo-0.8-eta-0.7
19 Percent Correct: 96.27%
20 Run-time: 1077.98965693 seconds
21
22 mo-0.8-eta-0.9
23 Percent Correct: 96.89%
24 Run-time: 1079.15961313 seconds
25
26
27 Layer with 2 hidden networks (300 and 100 neurons respectively).
28 mo-0.0-eta-0.4
29 Percent Correct: 89.74%
30 Run-time: 1220.183357 seconds
31
32 mo-0.0-eta-0.7
33 Percent Correct: 90.68%
34 Run-time: 1221.12571001 seconds
35
36 mo-0.0-eta-0.9
37 Percent Correct: 91.56%
38 Run-time: 1220.0601058 seconds
39
40 mo-0.8-eta-0.4
41 Percent Correct: 92.22%
42 Run-time: 1220.07663107 seconds
43
44 mo-0.8-eta-0.7
45 Percent Correct: 93.34%
46 Run-time: 1222.40497899 seconds
47
48 mo-0.8-eta-0.9
49 Percent Correct: 93.88%
50 Run-time: 1224.18042397 seconds

```

5 Appendix

5.1 ten-class-classifier.py

Listing 6: Ten-class Classifier

```

1 # Clint Ferrin
2 # Oct 12, 2017
3 # Neural Network Classifier
4
5 import matplotlib.pyplot as plt
6 import numpy as np
7 import pickle
8 from tensorflow.examples.tutorials.mnist import input_data
9 import time
10
11 def main():
12     num_inputs = 784
13     num_outputs = 10

```

```

14 batch_size = 100
15 epochs = 30
16 mse_freq = 50
17
18 # open mnist data
19 X,Y,X_test,Y_test = get_mnist_train("./data")
20
21 # initialize activation functions
22 relu = activation_function(relu_func,relu_der)
23 sig = activation_function(sigmoid_func,sigmoid_der)
24 no_activation = activation_function(return_value,return_value)
25
26 num_neurons = 300
27 # first layer tests
28 layers0 = [layer(num_inputs,num_neurons,relu)]
29 layers0.append(layer(num_neurons,num_outputs,no_activation))
30
31 # second layer tests
32 layers1 = [layer(num_inputs,300,relu)]
33 layers1.append(layer(300,100,relu))
34 layers1.append(layer(100,num_outputs,no_activation))
35
36 # set up test bench
37 layer_testbench = [layers0]
38 message = ["Layer with 1 hidden network (300 neurons). Epochs " + "\n",
39           "\nLayer with 2 hidden networks (300 and 100 neurons respectively).\n"]
40
41 momentum_values = [0.0,0.8]
42 step_size = [0.4,0.7,0.9]
43
44 file = open('../report/media/mnist/test/ten-long-class-network_statistics-bat-'
45             + str(batch_size) +
46             '-mse-' + str(mse_freq) + '.txt','w')
47
48 for index, layers in enumerate(layer_testbench):
49     file.write(message[index])
50     for mom in momentum_values:
51         for step in step_size:
52             print("Currently on layer " + str(index) + " momentum " + str(mom) + " step
53                   size " + str(step))
54
55             # create neural network
56             network = NeuralNetwork(layers,eta=step,momentum=mom)
57
58             # train network
59             start_time = time.time()
60             network.train_network(X,Y,batch_size=batch_size,
61                                   epochs=epochs,MSE_freq=mse_freq)
62             end_time = time.time()
63
64             # classify data
65             Yhat = network.classify_data(X)
66             Yhat_test = network.classify_data(X_test)
67             training_accuracy = network.validate_results(Yhat,Y)
68             training_accuracy = network.validate_results(Yhat_test,Y_test)
69
70             # write statistics
71             file.write('mo-' + str(mom) + '-eta-' + str(step) + "\n")
72             file.write("Percent Correct: " + str(training_accuracy) + "%\n")
73             file.write("Run-time: " + str(end_time-start_time) + " seconds" + "\n\n")
74
75             # plot error
76             network.plot_error(index,mom,step)
77
78             # save combined error plot
79             plt.title("MSE for Momentum= " + str(mom) +
80                      ", Step=" + str(step_size))
81             plt.savefig('../report/media/mnist/test/ten-c-bat-' + str(batch_size) +
82                         '-mse-' + str(mse_freq) + '-lay-' + str(index) +
83                         '-mo-' + str(int(mom*10)) + '-eta-' + str(int(step*10)) +
84                         '.pdf',bbox_inches='tight')
85             plt.clf()

```

```

86 class NeuralNetwork:
87     def __init__(self, layers, softmax=True, momentum=0,
88                 eta=0.1, MSE_freq=50, reg=0.001):
89         self.num_layers = len(layers)
90         self.num_outputs = layers[self.num_layers-1].num_neurons
91         self.error_array = []
92         self.error_plot = []
93         self.momentum = momentum
94         self.MSE_freq = MSE_freq
95         self.softmax = softmax
96         self.layers = layers
97         self.reg = reg
98         self.eta = eta
99         self.__set_GRV_starting_weights()
100
101     def train_network(self, X, Y, batch_size=100, epochs=100, MSE_freq=50):
102         self.MSE_freq = MSE_freq * batch_size
103         print("Training Data...")
104
105         # definition of iterations with mini-batch = N/B*epochs
106         itrs_per_epoch = int(np.ceil(X.shape[0]/float(batch_size)))
107         total_itrs = itrs_per_epoch * epochs
108
109         # print out 100 samples to gauge speed of program
110         if total_itrs > 5000:
111             print_frequency = total_itrs/100
112
113         # if iterations are few, print out 10
114         else:
115             print_frequency = total_itrs/10
116             if print_frequency is 0:
117                 print_frequency += 1 # to avoid modulo by zero
118
119         completed_epocs = 0
120         for i in range(total_itrs):
121             # randomly select samples from input data for batch
122             batch = np.random.randint(0,X.shape[0],batch_size)
123             self.train_data(X[batch],Y[batch])
124             if i%itrs_per_epoch is 0:
125                 print("Epoch %d. MSE: %f"%(completed_epocs,
126                     np.mean(self.error_array[-self.MSE_freq:])))
127                 completed_epocs += 1
128
129             if i%print_frequency is 0:
130                 print("Iteration %d MSE: %f"%(i+1,
131                     np.mean(self.error_array[-self.MSE_freq:])))
132
133
134         # create error plot
135         print("Final MSE: %f"%(np.mean(self.error_array[-self.MSE_freq:])))
136
137         # reverse order of list and split into even parts sizeof=MSE_freq
138         plot = self.error_array[::-1]
139         for i in range(0,len(plot),self.MSE_freq):
140             self.error_plot.append(np.mean(plot[i:i+self.MSE_freq]))
141         self.error_plot = self.error_plot[::-1]
142
143     def train_data(self, X, Y):
144         Yhat = self.forward_prop(X)
145         dE_dH = (Yhat-Y).T
146
147         # back propagation
148         if self.softmax is True:
149             self.layers[-1].output = np.ones(dE_dH.shape).T
150
151         for layer in self.layers[::-1]:
152             dE_dNet = layer.der(layer.output).T*dE_dH
153
154             # divide by number of samples in batch to regularize step
155             dE_dWeight = (np.dot(dE_dNet,layer.weight_der)) / \
156                 layer.weight_der.shape[0]
157
158             # obtain multiplication to pass back to next layer

```

```

159         dE_dH = np.dot(layer.W[:,0:-1].T,dE_dNet) * self.reg
160
161         # update weight matrix with momentum
162         layer.W += -layer.momentum_matrix
163         layer.momentum_matrix = \
164             self.momentum * layer.momentum_matrix + \
165             self.eta * dE_dWeight
166
167         # create long entire list of errors to plot at specific frequency later
168         for indx,yhat in enumerate(Yhat):
169             self.error_array.append(sum((Y[indx]-yhat)*(Y[indx]-yhat)))
170
171     def forward_prop(self, X):
172         prev_out = X
173         for layer in self.layers:
174             prev_out = np.c_[prev_out,np.ones([prev_out.shape[0],1])]
175             prev_out = layer.forward(prev_out)
176
177         if self.softmax is True:
178             self.layers[-1].output = self.stable_softmax(self.layers[-1].net)
179
180         return self.layers[-1].output
181
182     def classify_data(self, X):
183         Yhat = self.forward_prop(X)
184         class_type = np.argmax(Yhat,axis=1)
185         # returns list instead of matrix
186         return class_type
187
188     def stable_softmax(self, X):
189         exp_norm = np.exp(X - np.max(X))
190         return exp_norm / np.sum(exp_norm, axis=1).reshape((-1,1))
191
192     def validate_results(self, Yhat, Y):
193         Yhat_enc = (np.arange(Y.shape[1]) == Yhat[:, None]).astype(float)
194         num_err = np.sum(abs(Yhat_enc - Y))/2
195         training_accuracy = (len(Yhat)-num_err)/len(Yhat)*100
196         print("%d Mistakes. Training Accuracy: %.2f%%"%(int(num_err),training_accuracy))
197         return training_accuracy
198
199     def plot_error(self,index,momentum,eta):
200         plt.plot(range(len(self.error_plot)), self.error_plot)
201         plt.xlabel("Mean set for %d Training Samples"%(self.MSE_freq))
202         plt.ylabel("Mean Squared Error")
203
204     def write_network_values(self, filename):
205         pickle.dump(self, open(filename, "w"))
206         print("Network written to: %s" %(filename))
207
208     def __set_GRV_starting_weights(self):
209         # find number of outputs at each layer
210         for i in range(self.num_layers-2):
211             self.layers[i].num_outputs = self.layers[i+1].num_neurons
212         self.layers[-1].num_outputs = self.num_outputs
213
214         for layer in self.layers:
215             sigma = np.sqrt(float(2) / (layer.num_inputs + layer.num_neurons))
216             layer.W = np.random.normal(0,sigma,layer.W.shape)
217
218 class layer:
219     def __init__(self,num_inputs,num_neurons, activation):
220         self.W = np.random.uniform(0,1,[num_neurons,num_inputs+1])
221         self.momentum_matrix = np.zeros([num_neurons,num_inputs+1])
222         self.num_neurons = num_neurons
223         self.num_inputs = num_inputs
224         self.activation = activation
225         self.num_outputs = None
226         self.weight_der = None
227         self.net = None
228         self.output = None
229
230     def forward(self, X):
231         self.weight_der = X

```

```

232     self.net = np.dot(X, self.W.T)
233     self.output = self.activation.function(self.net)
234     return self.output
235
236     def der(self, X):
237         return self.activation.derivative(X)
238
239     def set_initial_conditions(self):
240         print("test")
241
242 class activation_function:
243     def __init__(self, function, derivative):
244         self.function = function
245         self.derivative = derivative
246
247     def function(self, x):
248         return self.function(x)
249
250     def derivative(self, x):
251         return self.derivative(x)
252
253 def print_digits(X, ordered, m, n):
254     f, ax = plt.subplots(m, n)
255     ordered = get_ordered(X);
256     for i in range(m):
257         for j in range(n):
258             ordered[i*n+j] = ordered[i*n+j].reshape(28,28)
259             ax[i][j].imshow(ordered[i*n+j], cmap = plt.cm.binary, interpolation="nearest")
260             ax[i][j].axis("off")
261     plt.show()
262
263 def sigmoid_func(x):
264     return 1/(1+np.exp(-x))
265
266 def sigmoid_der(x):
267     return (x*(1-x))
268
269 def relu_func(X):
270     return np.maximum(0,X)
271
272 def relu_der(X):
273     X[X<0]=0
274     return X
275
276 def return_value(X):
277     return X
278
279 def gendata2(class_type, N):
280     m0 = np.array(
281         [[-0.132, 0.320, 1.672, 2.230, 1.217, -0.819, 3.629, 0.8210, 1.808, 0.1700],
282          [-0.711, -1.726, 0.139, 1.151, -0.373, -1.573, -0.243, -0.5220, -0.511, 0.5330]])
283
284     m1 = np.array(
285         [[-1.169, 0.813, -0.859, -0.608, -0.832, 2.015, 0.173, 1.432, 0.743, 1.0328],
286          [ 2.065, 2.441, 0.247, 1.806, 1.286, 0.928, 1.923, 0.1299, 1.847, -0.052]])
287
288     x = np.array([[[]], [[]]])
289     for i in range(N):
290         idx = np.random.randint(10)
291         if class_type == 0:
292             m = m0[:, idx]
293         elif class_type == 1:
294             m = m1[:, idx]
295         else:
296             print("not a proper classifier")
297             return 0
298         x = np.c_[x, [[m[0]], [m[1]]] + np.random.randn(2,1)/np.sqrt(5)]
299     return x.T
300
301 def get_ordered_digits(X_train):
302     ordered = [
303         X_train[7] , # 0
304         X_train[4] , # 1

```



```

305         X_train[16], # 2
306         X_train[1] , # 3
307         X_train[2] , # 4
308         X_train[27], # 5
309         X_train[3] , # 6
310         X_train[14], # 7
311         X_train[5] , # 8
312         X_train[8] , # 9
313     ]
314     return ordered
315
316 def get_moon_class_data():
317     data = np.loadtxt("./data/classasgntrain1.dat", dtype=float)
318     x0 = data[:,0:2]
319     x1 = data[:,2:4]
320     data = data_frame(x0,x1)
321     return data.xtot,data.class_tot
322
323 def get_moon_gendata():
324     x0 = gendata2(0,10000)
325     x1 = gendata2(1,10000)
326     data = data_frame(x0,x1)
327     return data.xtot, data.class_tot
328
329 class data_frame:
330     def __init__(self, data0, data1):
331         self.x0 = data0
332         self.x1 = data1
333         self.xtot = np.r_[self.x0,self.x1]
334         self.N0 = self.x0.shape[0]
335         self.N1 = self.x1.shape[0]
336         self.N = self.N0 + self.N1
337         self.xlim = [np.min(self.xtot[:,0]),np.max(self.xtot[:,0])]
338         self.ylim = [np.min(self.xtot[:,1]),np.max(self.xtot[:,1])]
339         class_x0 = np.c_[np.zeros([self.N0,1]),np.ones([self.N0,1])]
340         class_x1 = np.c_[np.ones([self.N1,1]),np.zeros([self.N1,1])]
341         self.class_tot = np.r_[class_x0,class_x1]
342         self.y = np.r_[np.ones([self.N0,1]),np.zeros([self.N1,1])]
343
344         # create a training set from the classasgntrain1.dat (80% and 20%)
345         self.train_x0 = data0[0:80]
346         self.train_x1 = data1[0:80]
347         self.train_tot = np.r_[data0[0:80],data1[0:80]]
348         self.train_class_tot = np.r_[self.class_tot[0:80],self.class_tot[100:180]]
349         self.test_data = np.r_[data0[80:100],data1[80:100]]
350         self.test_class_tot = np.r_[self.class_tot[80:100],self.class_tot[180:200]]
351
352 def get_classasgn_80_20():
353     data = np.loadtxt("./data/classasgntrain1.dat", dtype=float)
354     x0 = data[:,0:2]
355     x1 = data[:,2:4]
356     data = data_frame(x0,x1)
357     return data.train_tot,data.train_class_tot,data.test_data,data.test_class_tot
358
359 def get_mnist_train(file_path):
360     mnist = input_data.read_data_sets(file_path)
361     X = mnist.train.images
362     y = mnist.train.labels.astype("int")
363     Y = (np.arange(np.max(y) + 1) == y[:, None]).astype(float)
364     X_test = mnist.test.images
365     y_test = mnist.test.labels.astype("int")
366     Y_test = (np.arange(np.max(y_test) + 1) == y_test[:, None]).astype(float)
367     return X,Y,X_test,Y_test
368
369 def plot_data(x0,x1):
370     xtot = np.r_[x0,x1]
371     xlim = [np.min(xtot[:,0]),np.max(xtot[:,0])]
372     ylim = [np.min(xtot[:,1]),np.max(xtot[:,1])]
373
374     fig = plt.figure() # make handle to save plot
375     plt.scatter(x0[:,0],x0[:,1],c='red',label='$x_0$')
376     plt.scatter(x1[:,0],x1[:,1],c='blue',label='$x_1$')
377     plt.xlabel('X Coordinate')

```

```

378 plt.ylabel('Y Coordinate')
379 plt.title("Neural Network (Two-class Boundary)")
380 plt.legend()
381
382 def plot_boundaries(xlim, ylim, equation):
383     xpl = np.linspace(xlim[0],xlim[1], num=100)
384     ypl = np.linspace(ylim[0],ylim[1], num=100)
385
386     red_pts = np.array([],[])
387     blue_pts= np.array([],[])
388     for x in xpl:
389         for y in ypl:
390             point = np.array([x,y]).reshape(1,2)
391             prob = equation(point)
392             if prob == 0:
393                 blue_pts = np.c_[blue_pts,[x,y]]
394             else:
395                 red_pts = np.c_[red_pts,[x,y]]
396
397     plt.scatter(blue_pts[0,:],blue_pts[1:],color='blue',s=0.25)
398     plt.scatter(red_pts[0,:],red_pts[1:],color='red',s=0.25)
399     plt.xlim(xlim)
400     plt.ylim(ylim)
401
402 if __name__ == '__main__':
403     main()

```

5.2 two-class-classifier.py

Listing 7: Two-class Classifier

```

1 # Clint Ferrin
2 # Oct 12, 2017
3 # Neural Network Classifier
4
5 import matplotlib.pyplot as plt
6 import numpy as np
7 import pickle
8 from tensorflow.examples.tutorials.mnist import input_data
9 import time
10
11 def main():
12     num_inputs = 2
13     num_outputs= 2
14     batch_size = 1 # not used. All data used
15     epochs = 2000
16     mse_freq = 200
17
18     # open mnist data
19     X,Y,X_test,Y_test = get_classesgn_80_20()
20     # Y = Y[:,1].reshape(-1,1)
21     # Y_test = Y_test[:,1].reshape(-1,1)
22
23     # initialize activation functions
24     relu = activation_function(relu_func,relu_der)
25     sig = activation_function(sigmoid_func,sigmoid_der)
26     no_activation = activation_function(return_value,return_value)
27
28     # first layer tests
29     layers0 = [layer(num_inputs,5,sig)]
30     layers0.append(layer(5,num_outputs,sig))
31
32     layers1 = [layer(num_inputs,5,sig)]
33     layers1.append(layer(5,10,sig))
34     layers1.append(layer(5,num_outputs,sig))
35
36     layer_testbench = [layers0,layers1]
37
38     message = ["Two class layer with 1 hidden network (5 neurons). Epochs " + "\n",
39               "\nTwo class layer with 2 hidden networks (5 and 10 neurons respectively).\n"]

```

```

40 momentum_values = [0.0,0.8]
41 step_size = [0.4,0.7,0.9]
42
43
44 file = open('../report/media/two-class-80-20/test/two-class-net-80-20-statistics-bat-' +
45             str(batch_size) + '-mse-' + str(mse_freq) + '.txt','w')
46
47 for index, layers in enumerate(layer_testbench):
48     file.write(message[index])
49     plt.clf()
50     for mom in momentum_values:
51         for step in step_size:
52             print("Currently on layer " + str(index) + " momentum " + str(mom) + " step
53                   size " + str(step))
54
55             # create neural network
56             network = NeuralNetwork(layers,eta=step,momentum=mom,softmax=False)
57
58             # train network
59             start_time = time.time()
60             network.train_network(X,Y,batch_size=batch_size,
61                                  epochs=epochs,MSE_freq=mse_freq)
62             end_time = time.time()
63
64             # classify data
65             Yhat = network.classify_data(X_test)
66             training_accuracy = network.validate_results(Yhat,Y_test)
67
68             file.write('mo-' + str(mom) + '-eta-' + str(step) + "\n")
69             file.write("Percent Correct: " + str(training_accuracy) + "%\n")
70             file.write("Run-time: " + str(end_time-start_time) + " seconds" + "\n\n")
71
72             # plot data points and graph boundaries
73             plt.figure(1)
74             plt.clf()
75             plot_data(X_test[0:20],X_test[20:40])
76
77             xtot = np.r_[X,X_test]
78             xlim = [np.min(xtot[:,0]),np.max(xtot[:,0])]
79             ylim = [np.min(xtot[:,1]),np.max(xtot[:,1])]
80
81             plot_boundaries(xlim,ylim,network.classify_data)
82             plt.savefig('../report/media/two-class-80-20/test/two-c-net-80-20-bat-' + str(
83                 batch_size) +
84                 '-mse-' + str(mse_freq) + '-lay-' + str(index) +
85                 '-mo-' + str(int(mom*10)) + '-eta-' + str(int(step*10)) +
86                 '.pdf',bbox_inches='tight')
87             plt.show()
88             plt.clf()
89
90             plt.figure(2)
91             # plot error and graph boundaries
92             network.plot_error(index,mom,step)
93
94             # save combined error plots
95             plt.title("MSE for Momentum= " + str(mom) +
96                      ", Step=" + str(step_size))
97             plt.savefig('../report/media/two-class-80-20/two-c-error-80-20-bat-' + str(
98                 batch_size) +
99                 '-mse-' + str(mse_freq) + '-lay-' + str(index) +
100                 '-mo-' + str(int(mom*10)) + '-eta-' + str(int(step*10)) +
101                 '.pdf',bbox_inches='tight')
102             plt.show()
103             plt.clf()
104
105 class NeuralNetwork:
106     def __init__(self, layers, softmax=True, momentum=0,
107                  eta=0.1, MSE_freq=50, reg=0.001):
108         self.num_layers = len(layers)
109         self.num_outputs = layers[self.num_layers-1].num_neurons
110         self.error_array = []
111         self.error_plot = []
112         self.momentum = momentum

```

```

109     self.MSE_freq = MSE_freq
110     self.softmax= softmax
111     self.layers = layers
112     self.reg = reg
113     self.eta = eta
114     self.__set_GRV_starting_weights()
115
116     def __set_GRV_starting_weights(self):
117         for i in range(self.num_layers-2):
118             self.layers[i].num_outputs = self.layers[i+1].num_neurons
119         self.layers[-1].num_outputs = self.num_outputs
120
121         for layer in self.layers:
122             sigma = np.sqrt(float(2) / (layer.num_inputs + layer.num_neurons))
123             layer.W = np.random.normal(0,sigma,layer.W.shape)
124
125     def forward_prop(self, X):
126         prev_out = X
127         for layer in self.layers:
128             prev_out = np.c_[prev_out,np.ones([prev_out.shape[0],1])]
129             prev_out = layer.forward(prev_out)
130
131         if self.softmax is True:
132             self.layers[-1].output = self.stable_softmax(self.layers[-1].net)
133
134         return self.layers[-1].output
135
136     def classify_data(self, X):
137         Yhat = self.forward_prop(X)
138         class_type = np.argmax(Yhat,axis=1)
139         return class_type
140
141     def train_network(self, X, Y, batch_size=100, epochs=100, MSE_freq=50):
142         self.MSE_freq = MSE_freq * batch_size
143         print("Training Data...")
144
145         # definition of iterations with mini-batch = N/B*epochs
146         # itrs_per_epoch = int(np.ceil(X.shape[0]/float(batch_size)))
147         total_itrs = epochs
148
149         if total_itrs > 5000:
150             print_frequency = total_itrs/100
151         else:
152             print_frequency = total_itrs/10
153             if print_frequency is 0:
154                 print_frequency += 1
155
156         completed_epocs = 0
157         for i in range(total_itrs):
158             batch = np.random.randint(0,X.shape[0],batch_size)
159             self.train_data(X[batch],Y[batch])
160
161             self.train_data(X,Y)
162             if i%print_frequency is 0:
163                 print("Iteration %d MSE: %f"%(i+1, np.mean(self.error_array[-self.MSE_freq:])))
164
165         # create error plot
166         print("Final MSE: %f"%(np.mean(self.error_array[-self.MSE_freq:])))
167
168         plot = self.error_array[::-1]
169         for i in range(0,len(plot),self.MSE_freq):
170             self.error_plot.append(np.mean(plot[i:i+self.MSE_freq]))
171         self.error_plot = self.error_plot[::-1]
172
173     def train_data(self, X, Y):
174         Yhat = self.forward_prop(X)
175         dE_dH = (Yhat-Y).T
176         iterlayers = iter(self.layers[::-1])
177
178         # back propagation
179         if self.softmax is True:
180             # divide by number of incoming batch size to regularize

```

```

181         dE_dWeight = (-np.dot(-dE_dH,self.layers[-1].weight_der) / \
182                         self.layers[-1].weight_der.shape[0])
183
184         # do not include the bias weights--not needed and will be updated later
185         dE_dH = np.dot(self.layers[-1].W[:,0:-1].T,dE_dH)
186
187         # update current weights with momentum
188         self.layers[-1].W += -self.eta*(dE_dWeight + \
189                                     self.momentum*self.layers[-1].momentum_matrix)
190
191         self.layers[-1].momentum_matrix = dE_dWeight
192
193         # skip the last layer if softmax
194         next(iterlayers)
195
196     for layer in iterlayers:
197         dE_dNet = layer.der(layer.output).T*dE_dH
198         dE_dWeight = (np.dot(dE_dNet,layer.weight_der)) / \
199                     layer.weight_der.shape[0]
200
201         dE_dH = np.dot(layer.W[:,0:-1].T,dE_dNet)
202
203         layer.W += -layer.momentum_matrix
204         layer.momentum_matrix = \
205             self.momentum * layer.momentum_matrix + \
206             self.eta * dE_dWeight
207
208     for indx,yhat in enumerate(Yhat):
209         self.error_array.append(sum((Y[indx]-yhat)*(Y[indx]-yhat)))
210
211     def stable_softmax(self, X):
212         exp_norm = np.exp(X - np.max(X))
213         return exp_norm / np.sum(exp_norm, axis=1).reshape((-1,1))
214
215     def plot_error(self,index,momentum,eta):
216         plt.plot(range(len(self.error_plot)), self.error_plot)
217         plt.xlabel("Mean set for %d Training Samples"%(self.MSE_freq))
218         plt.ylabel("Mean Squared Error")
219
220     def write_network_values(self, filename):
221         pickle.dump(self, open(filename, "w"))
222         print("Network written to: %s" %(filename))
223
224     def validate_results(self, Yhat, Y):
225         Yhat_enc = (np.arange(Y.shape[1]) == Yhat[:, None]).astype(float)
226         num_err = np.sum(abs(Yhat_enc - Y))/2
227         training_accuracy = (len(Yhat)-num_err)/len(Yhat)*100
228         print("%d Mistakes. Training Accuracy: %.2f%%"%(int(num_err),training_accuracy))
229         return training_accuracy
230
231     def set_initial_conditions(self):
232         # self.layers[0].W[0,:] = [0.15,0.2,0.35]
233         # self.layers[0].W[1,:] = [0.25,0.3,0.35]
234         # self.layers[0].W[2,:] = [0.25,0.3,0.35]
235
236         self.layers[0].W[0,:] = [0.1,0.1,0.01]
237         self.layers[0].W[1,:] = [0.2,0.2,0.1 ]
238         self.layers[0].W[2,:] = [0.3,0.3,0.1 ]
239
240     class layer:
241         def __init__(self,num_inputs,num_neurons, activation):
242             self.num_neurons = num_neurons
243             self.num_inputs = num_inputs
244             self.num_outputs = None
245             self.weight_der = None
246             self.activation = activation
247             self.net = None
248             self.W = np.random.uniform(0,1,[num_neurons,num_inputs+1])
249             self.momentum_matrix = np.zeros([num_neurons,num_inputs+1])
250             self.output = None
251
252         def forward(self, X):

```

```

254     self.weight_der = X
255     self.net = np.dot(X, self.W.T)
256     self.output = self.activation.function(self.net)
257     return self.output
258
259     def der(self, X):
260         return self.activation.derivative(X)
261
262     def set_initial_conditions(self):
263         print("test")
264
265 class activation_function:
266     def __init__(self, function, derivative):
267         self.function = function
268         self.derivative = derivative
269
270     def function(self, x):
271         return self.function(x)
272
273     def derivative(self, x):
274         return self.derivative(x)
275
276 def get_moon_class_data():
277     data = np.loadtxt("./data/classasgntrain1.dat", dtype=float)
278     x0 = data[:, 0:2]
279     x1 = data[:, 2:4]
280     data = data_frame(x0, x1)
281     return data.xtot, data.class_tot
282
283 def get_moon_gendata():
284     x0 = gendata2(0, 10000)
285     x1 = gendata2(1, 10000)
286     data = data_frame(x0, x1)
287     return data.xtot, data.class_tot
288
289 def get_classasgn_80_20():
290     data = np.loadtxt("./data/classasgntrain1.dat", dtype=float)
291     x0 = data[:, 0:2]
292     x1 = data[:, 2:4]
293     data = data_frame(x0, x1)
294     return data.train_tot, data.train_class_tot, data.test_data, data.test_class_tot
295
296 class data_frame:
297     def __init__(self, data0, data1):
298         self.x0 = data0
299         self.x1 = data1
300         self.xtot = np.r_[self.x0, self.x1]
301         self.N0 = self.x0.shape[0]
302         self.N1 = self.x1.shape[0]
303         self.N = self.N0 + self.N1
304         self.xlim = [np.min(self.xtot[:, 0]), np.max(self.xtot[:, 0])]
305         self.ylim = [np.min(self.xtot[:, 1]), np.max(self.xtot[:, 1])]
306         class_x0 = np.c_[np.zeros([self.N0, 1]), np.ones([self.N0, 1])]
307         class_x1 = np.c_[np.ones([self.N1, 1]), np.zeros([self.N1, 1])]
308         self.class_tot = np.r_[class_x0, class_x1]
309         self.y = np.r_[np.ones([self.N0, 1]), np.zeros([self.N1, 1])]
310
311         # create a training set from the classasgntrain1.dat
312         self.train_x0 = data0[0:80]
313         self.train_x1 = data1[0:80]
314         self.train_tot = np.r_[data0[0:80], data1[0:80]]
315         self.train_class_tot = np.r_[self.class_tot[0:80], self.class_tot[100:180]]
316         self.test_data = np.r_[data0[80:100], data1[80:100]]
317         self.test_class_tot = np.r_[self.class_tot[80:100], self.class_tot[180:200]]
318
319 def plot_data(x0, x1):
320     xtot = np.r_[x0, x1]
321     xlim = [np.min(xtot[:, 0]), np.max(xtot[:, 0])]
322     ylim = [np.min(xtot[:, 1]), np.max(xtot[:, 1])]
323
324     fig = plt.figure() # make handle to save plot
325     plt.scatter(x0[:, 0], x0[:, 1], c='red', label='$x_0$')
326     plt.scatter(x1[:, 0], x1[:, 1], c='blue', label='$x_1$')

```

```

327 plt.xlabel('X Coordinate')
328 plt.ylabel('Y Coordinate')
329 plt.title("Neural Network (Two-class Boundary)")
330 plt.legend()
331
332 def plot_boundaries(xlim, ylim, equation):
333     xpl = np.linspace(xlim[0],xlim[1], num=100)
334     ypl = np.linspace(ylim[0],ylim[1], num=100)
335
336     red_pts = np.array([],[])
337     blue_pts= np.array([],[])
338     for x in xpl:
339         for y in ypl:
340             point = np.array([x,y]).reshape(1,2)
341             prob = equation(point)
342             if prob == 0:
343                 blue_pts = np.c_[blue_pts,[x,y]]
344             else:
345                 red_pts = np.c_[red_pts,[x,y]]
346
347     plt.scatter(blue_pts[0,:],blue_pts[1,:],color='blue',s=0.25)
348     plt.scatter(red_pts[0,:],red_pts[1,:],color='red',s=0.25)
349     plt.xlim(xlim)
350     plt.ylim(ylim)
351
352 def sigmoid_func(x):
353     return 1/(1+np.exp(-x))
354
355 def sigmoid_der(x):
356     return (x*(1-x))
357
358 def return_value(X):
359     return X
360
361 def relu_func(X):
362     return np.maximum(0,X)
363
364 def relu_der(X):
365     X[X<0]=0
366     return X
367
368 def softmax_func(x):
369     exps = np.exp(x)
370     return exps / np.sum(exps)
371
372 def gendata2(class_type,N):
373     m0 = np.array(
374         [[-0.132,0.320,1.672,2.230,1.217,-0.819,3.629,0.8210,1.808, 0.1700],
375          [-0.711,-1.726,0.139,1.151,-0.373,-1.573,-0.243,-0.5220,-0.511,0.5330]])
376
377     m1 = np.array(
378         [[-1.169,0.813,-0.859,-0.608,-0.832,2.015,0.173,1.432,0.743,1.0328],
379          [ 2.065,2.441,0.247,1.806,1.286,0.928,1.923,0.1299,1.847,-0.052]])
380
381     x = np.array([],[])
382     for i in range(N):
383         idx = np.random.randint(10)
384         if class_type == 0:
385             m = m0[:,idx]
386         elif class_type == 1:
387             m = m1[:,idx]
388         else:
389             print("not a proper classifier")
390             return 0
391         x = np.c_[x, [[m[0]], [m[1]]] + np.random.randn(2,1)/np.sqrt(5)]
392     return x.T
393
394 def get_ordered_digits(X_train):
395     ordered = [
396         X_train[7] , # 0
397         X_train[4] , # 1
398         X_train[16], # 2
399         X_train[1] , # 3

```

```

400         X_train[2] , # 4
401         X_train[27], # 5
402         X_train[3] , # 6
403         X_train[14], # 7
404         X_train[5] , # 8
405         X_train[8] , # 9
406     ]
407     return ordered
408
409 def print_digits(X,ordered,m,n):
410     f, ax = plt.subplots(m,n)
411     ordered = get_ordered(X);
412     for i in range(m):
413         for j in range(n):
414             ordered[i*n+j] = ordered[i*n+j].reshape(28,28)
415             ax[i][j].imshow(ordered[i*n+j], cmap = plt.cm.binary, interpolation="nearest")
416             ax[i][j].axis("off")
417
418     plt.show()
419
420 def get_mnist_train(file_path):
421     mnist = input_data.read_data_sets(file_path)
422     X = mnist.train.images
423     y = mnist.train.labels.astype("int")
424     Y = (np.arange(np.max(y) + 1) == y[:, None]).astype(float)
425     X_test = mnist.test.images
426     y_test = mnist.test.labels.astype("int")
427     Y_test = (np.arange(np.max(y_test) + 1) == y_test[:, None]).astype(float)
428     return X,Y,X_test,Y_test
429
430 def get_2_class_data():
431     X = np.array([[0.05, 0.1],
432                  [0.07, 0.1],
433                  [0.05, 0.1],
434                  [0.05, 0.1],
435                  [0.05, 0.1]])
436
437     Y = np.array([[0.01, 0.99],
438                  [0.01, 0.99],
439                  [0.01, 0.99],
440                  [0.01, 0.99],
441                  [0.01, 0.99]])
442     return X,Y
443
444 def get_3_class_data():
445     X = np.array([[0.05, 0.1],
446                  [0.07, 0.3],
447                  [0.09, 0.5],
448                  [0.05, 0.1]])
449
450     Y = np.array([[1, 0, 0],
451                  [0, 1, 0],
452                  [0, 0, 1],
453                  [1, 0, 0]])
454     return X,Y
455
456 def get_spiral_class_data():
457     np.random.seed(0)
458     N = 100 # number of points per class
459     D = 2 # dimensionality
460     K = 3 # number of classes
461     X = np.zeros((N*K,D))
462     y = np.zeros(N*K, dtype='uint8')
463     for j in xrange(K):
464         ix = range(N*j,N*(j+1))
465         r = np.linspace(0.0,1,N) # radius
466         t = np.linspace(j*4,(j+1)*4,N) + np.random.randn(N)*0.2 # theta
467         X[ix] = np.c_[r*np.sin(t), r*np.cos(t)]
468         y[ix] = j
469     # fig = plt.figure()
470     # plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap=plt.cm.Spectral)
471     # plt.xlim([-1,1])
472     # plt.ylim([-1,1])

```



```
473     Y = (np.arange(np.max(y) + 1) == y[:, None]).astype(float)
474     return X, Y
475
476 if __name__ == '__main__':
477     main()
```