

Deep Neural Networks

Neural Networks: ECE 5930

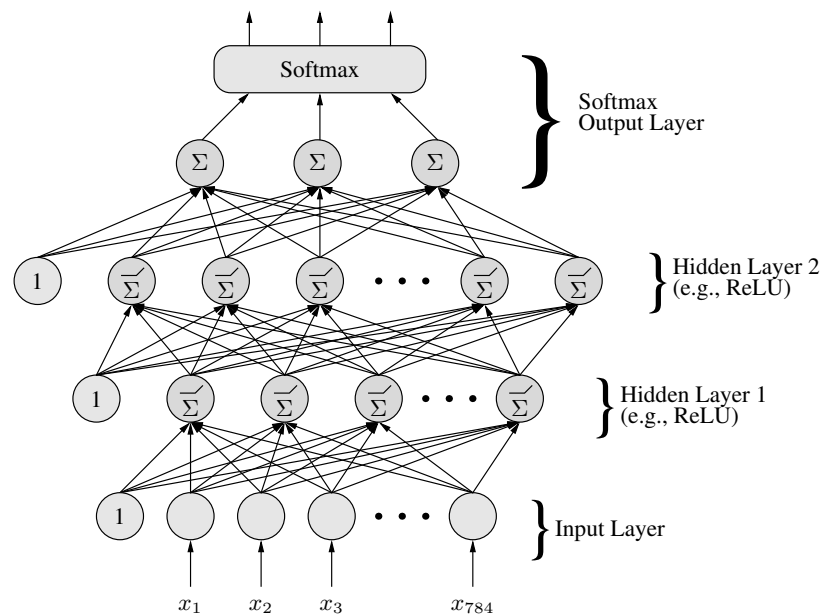


Figure: Two Hidden Layer Neural Network

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 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.

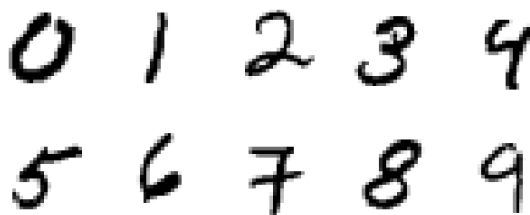


Figure 1: 10 Digits from the MNIST Data-set

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 would like to explain the back propagation portion of 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. They show that a simplified version of the derivative for the softmax that worked for my program.

Listing 2: Back propagation

```

1 def train_data(self, X, Y):
2     Yhat = self.forward_prop(X)
3     dE_dH = (Yhat - Y).T
4     iterlayers = iter(self.layers[::-1])
5
6     # back propagation
7     if self.softmax is True:
8         # divide by number of incoming batch size to regularize
9         dE_dWeight = (-np.dot(-dE_dH, self.layers[-1].weight_der) / \
10                        self.layers[-1].weight_der.shape[0])
11
12         # do not include the bias weights--not needed and will be updated later
13         dE_dH = np.dot(self.layers[-1].W[:, 0:-1].T, dE_dH) * self.reg
14         # Yhat.shape[0]

```

```

15         # update current weights with momentum
16         self.layers[-1].W += -self.eta*(dE_dWeight + \
17             self.momentum*self.layers[-1].momentum_matrix)
18         self.layers[-1].momentum_matrix = dE_dWeight
19
20         # skip the last layer if softmax
21         next(iterlayers)
22
23     for layer in iterlayers:
24         dE_dNet = layer.der(layer.output).T*dE_dH
25         dE_dWeight = (np.dot(dE_dNet, layer.weight_der)) / \
26             layer.weight_der.shape[0]
27
28         dE_dH = np.dot(layer.W[:,0:-1].T, dE_dNet) * self.reg
29             # Yhat.shape[0]
30
31         layer.W += -layer.momentum_matrix
32         layer.momentum_matrix = \
33             self.momentum * layer.momentum_matrix + \
34             self.eta * dE_dWeight
35
36     for indx, yhat in enumerate(Yhat):
37         self.error_array.append(sum((Y[indx]-yhat)*(Y[indx]-yhat)))
38

```

3 Two-class Classifier

The data set from `classasgntrain1.dat` is a grouping of data centered around 10 different points 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.

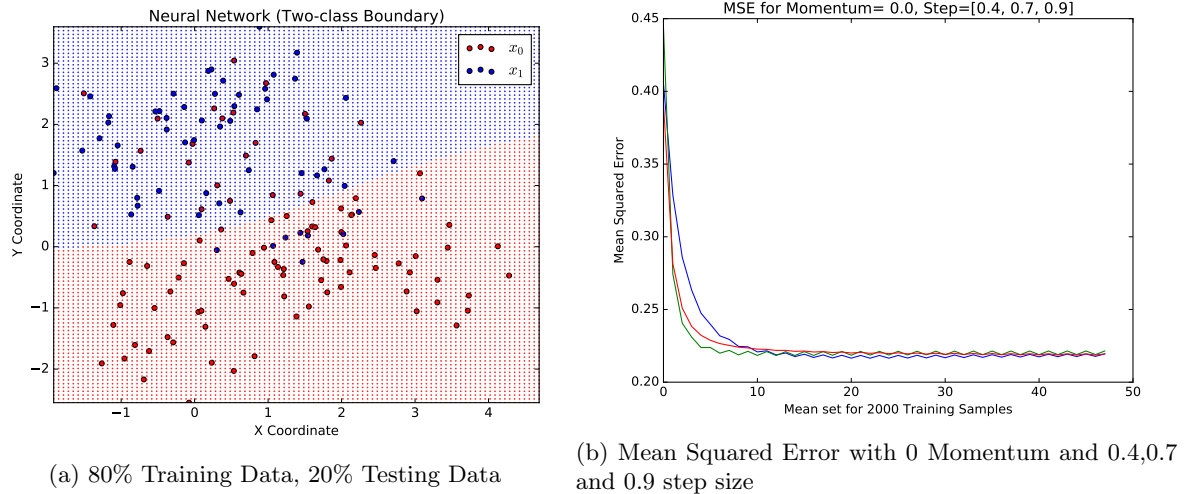


Figure 2: Trained Output for 80% Training Data, 20% Testing Data with 0.0 Momentum

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.

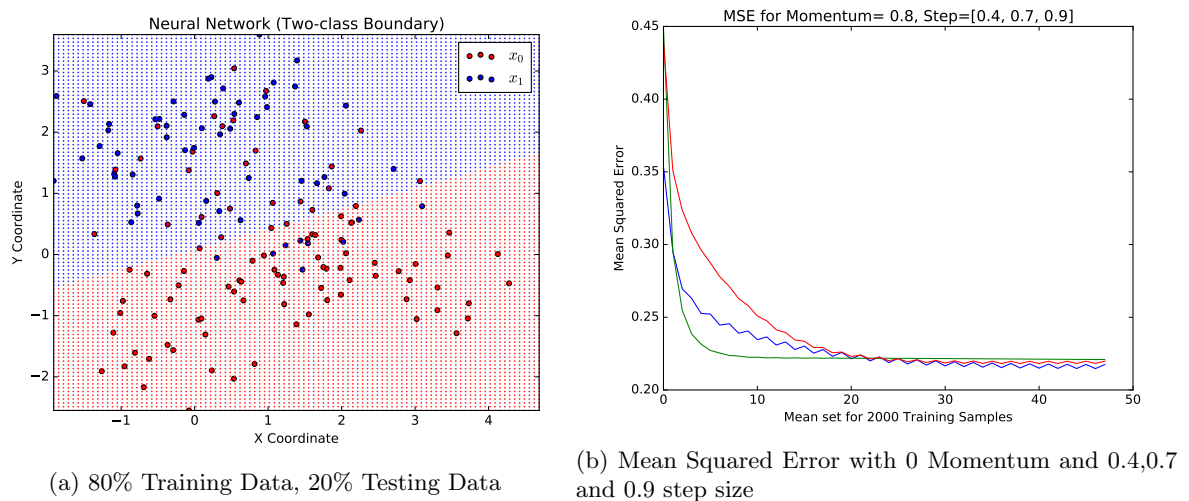


Figure 3: Trained Output for 80% Training Data, 20% Testing Data with 0.8 Momentum

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.

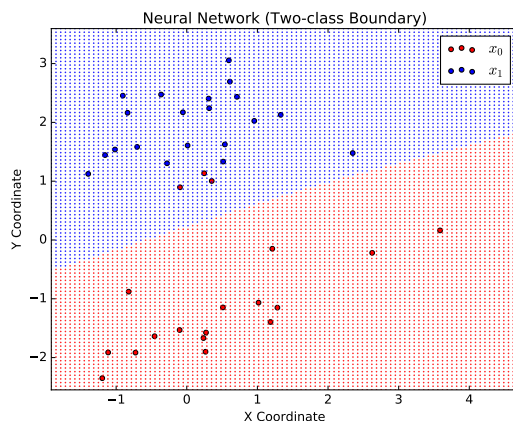
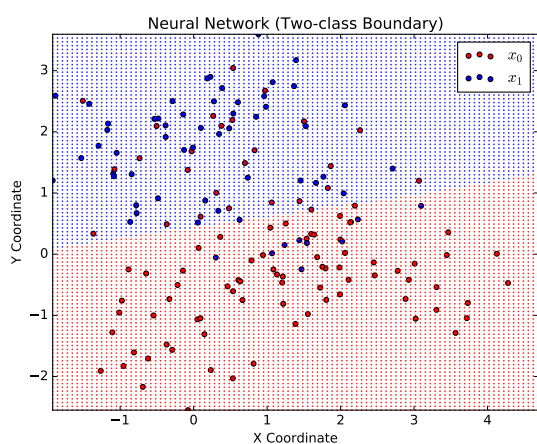
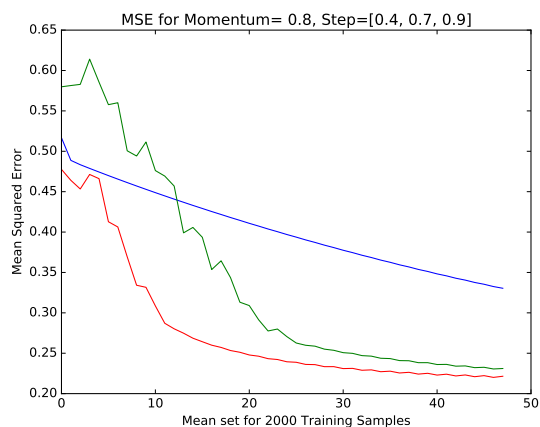


Figure 4: Three Miss-classified Points



(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.

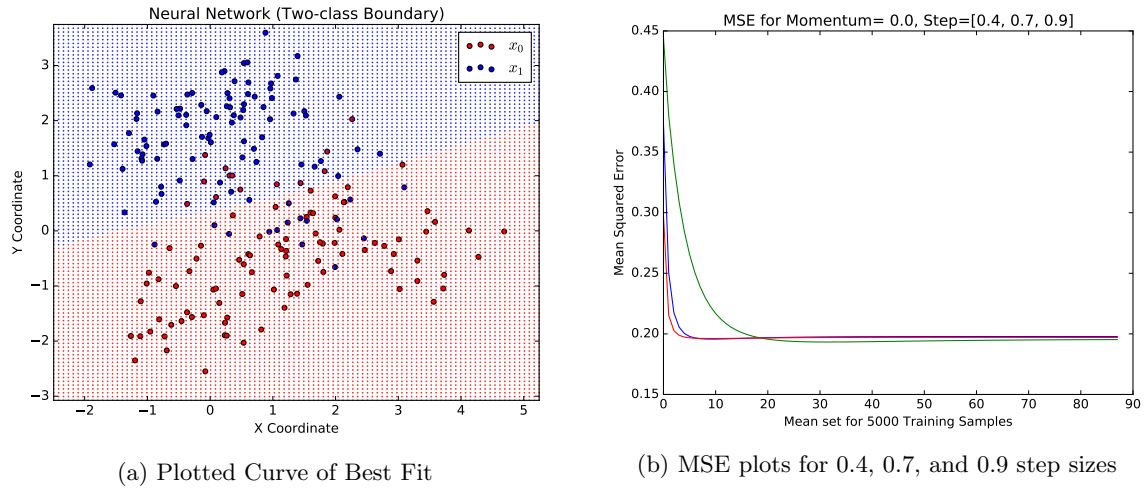


Figure 6: Comparison of Two-class Classifier with 0.0 Momentum

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.

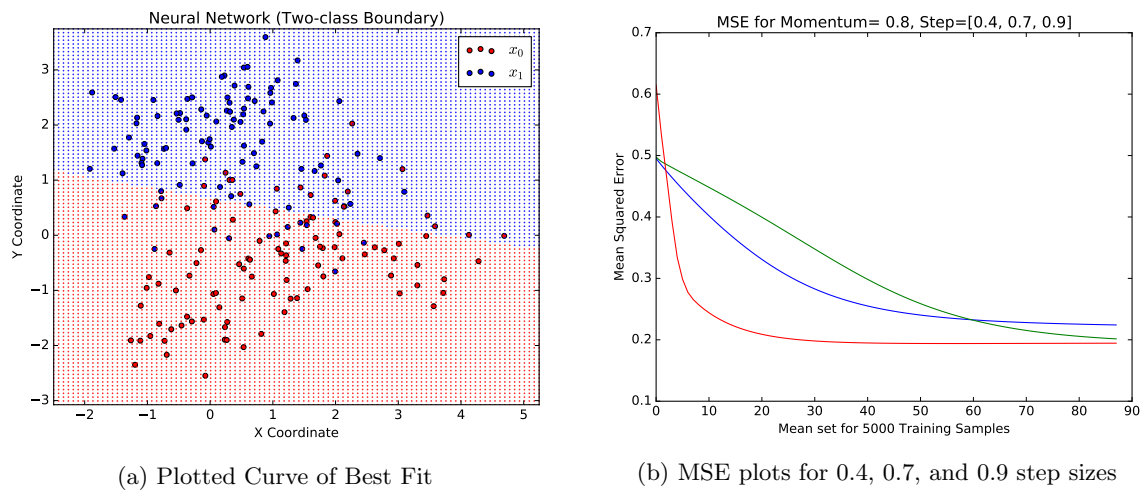


Figure 7: Comparison of Two-class Classifier with 0.8 Momentum and an Additional Layer

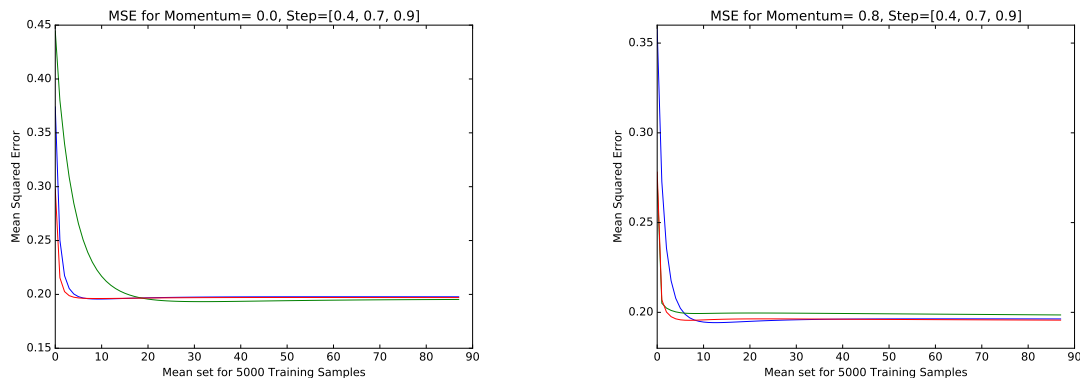
The Neural Network returned an error of 19.805%, which puts its results just behind the Bayes Optimal Classifier and the k-nearest neighbor approach. Because the 15-Nearest Neighbor is not practical with large datasets, and because a model for the Bayes Optimal Classifier is often impossible to find,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

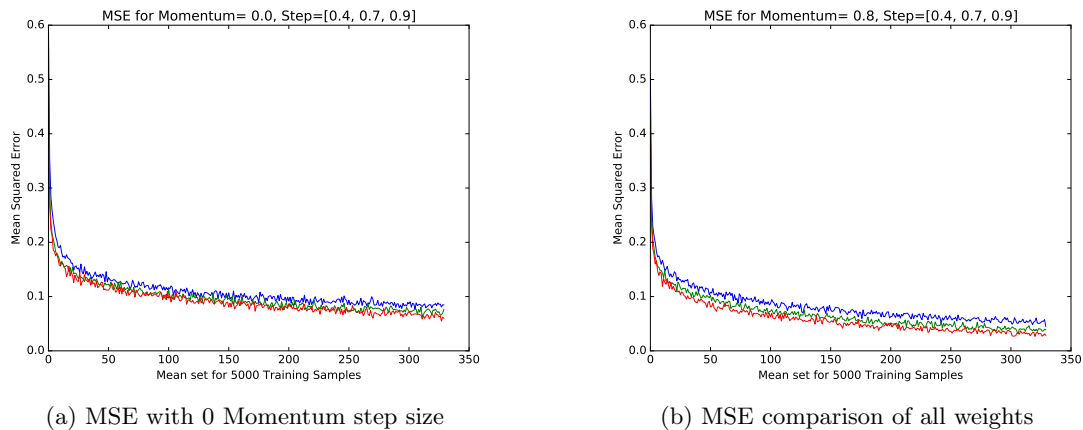


Figure 9: Comparing the Mean Square Error with Momentum 0.0 and 0.8

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 the program listing below.

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.

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.

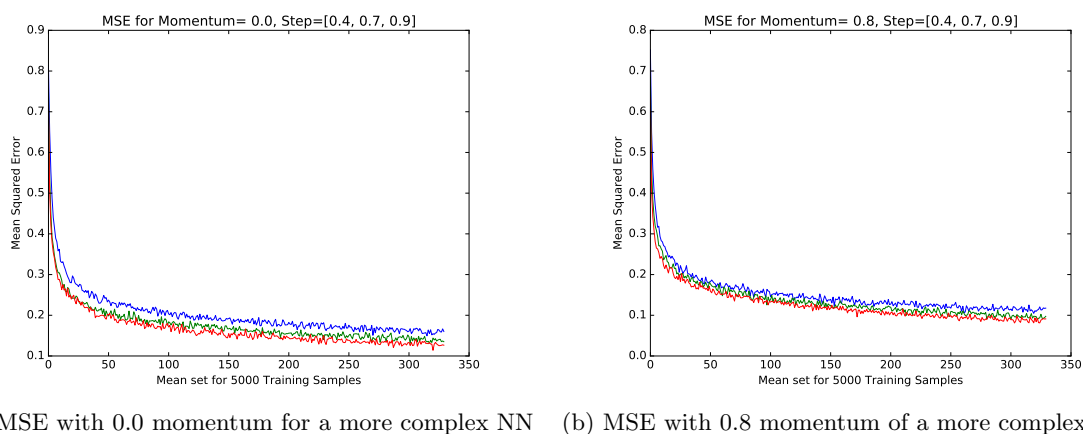


Figure 10: Comparing the Mean Square Error with Momentum 0.0 and 0.8 with an Additional Layer

For a detailed description of how long each network took in clock time (seconds) for a certain accuracy, see the listing below.

Listing 5: Output Accuracy 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 = 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 # 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,layers1]
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/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_test)
66             training_accuracy = network.validate_results(Yhat,Y_test)
67
68             # write statistics
69             file.write('mo-' + str(mom) + '-eta-' + str(step) + "\n")
70             file.write("Percent Correct: " + str(training_accuracy) + "%\n")
71             file.write("Run-time: " + str(end_time-start_time) + " seconds" + "\n\n")
72
73             # plot error
74             network.plot_error(index,mom,step)
75
76             # save combined error plot
77             plt.title("MSE for Momentum= " + str(mom) +
78                      ", Step=" + str(step_size))
79             plt.savefig('../report/media/mnist/ten-c-bat-' + str(batch_size) +
80                         '-mse-' + str(mse_freq) + '-lay-' + str(index) +
81                         '-mo-' + str(int(mom*10)) + '-eta-' + str(int(step*10)) +
82                         '.pdf',bbox_inches='tight')
83             plt.clf()
84
85 class NeuralNetwork:
86     def __init__(self, layers, softmax=True, momentum=0,
87                   eta=0.1, MSE_freq=50, reg=0.001):

```

```

87     self.num_layers = len(layers)
88     self.num_outputs = layers[self.num_layers-1].num_neurons
89     self.error_array = []
90     self.error_plot = []
91     self.momentum = momentum
92     self.MSE_freq = MSE_freq
93     self.softmax= softmax
94     self.layers = layers
95     self.reg = reg
96     self.eta = eta
97     self.__set_GRV_starting_weights()
98
99     def train_network(self, X, Y, batch_size=100, epochs=100, MSE_freq=50):
100         self.MSE_freq = MSE_freq * batch_size
101         print("Training Data...")
102
103         # definition of iterations with mini-batch = N/B*epochs
104         itrs_per_epoch = int(np.ceil(X.shape[0]/float(batch_size)))
105         total_itrs = itrs_per_epoch * epochs
106
107         # print out 100 samples to gauge speed of program
108         if total_itrs > 5000:
109             print_frequency = total_itrs/100
110
111         # if iterations are few, print out 10
112         else:
113             print_frequency = total_itrs/10
114             if print_frequency is 0:
115                 print_frequency += 1 # to avoid modulo by zero
116
117         completed_epocs = 0
118         for i in range(total_itrs):
119             # randomly select samples from input data for batch
120             batch = np.random.randint(0,X.shape[0],batch_size)
121             self.train_data(X[batch],Y[batch])
122             if i%itrs_per_epoch is 0:
123                 print("Epoch %d. MSE: %f"%(completed_epocs,
124                     np.mean(self.error_array[-self.MSE_freq:])))
125                 completed_epocs += 1
126
127             if i%print_frequency is 0:
128                 print("Iteration %d MSE: %f"%(i+1,
129                     np.mean(self.error_array[-self.MSE_freq:])))
130
131
132         # create error plot
133         print("Final MSE: %f"%(np.mean(self.error_array[-self.MSE_freq:])))
134
135         # reverse order of list and split into even parts sizeof=MSE_freq
136         plot = self.error_array[::-1]
137         for i in range(0,len(plot),self.MSE_freq):
138             self.error_plot.append(np.mean(plot[i:i+self.MSE_freq]))
139         self.error_plot = self.error_plot[::-1]
140
141     def train_data(self, X, Y):
142         Yhat = self.forward_prop(X)
143         dE_dH = (Yhat-Y).T
144         iterlayers = iter(self.layers[::-1])
145
146         # back propagation
147         if self.softmax is True:
148             # divide by number of incoming batch size to regularize
149             dE_dWeight = (-np.dot(-dE_dH,self.layers[-1].weight_der) / \
150                 self.layers[-1].weight_der.shape[0])
151
152             # do not include the bias weights--not needed and will be updated later
153             dE_dH = np.dot(self.layers[-1].W[:,0:-1].T,dE_dH) * self.reg
154             # Yhat.shape[0]
155
156             # update current weights with momentum
157             self.layers[-1].W += -self.eta*(dE_dWeight + \
158                 self.momentum*self.layers[-1].momentum_matrix)
159             self.layers[-1].momentum_matrix = dE_dWeight

```

```

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

```

```

233     self.activation = activation
234     self.num_outputs = None
235     self.weight_der = None
236     self.net = None
237     self.output = None
238
239     def forward(self, X):
240         self.weight_der = X
241         self.net = np.dot(X, self.W.T)
242         self.output = self.activation.function(self.net)
243         return self.output
244
245     def der(self, X):
246         return self.activation.derivative(X)
247
248     def set_initial_conditions(self):
249         print("test")
250
251 class activation_function:
252     def __init__(self, function, derivative):
253         self.function = function
254         self.derivative = derivative
255
256     def function(self, x):
257         return self.function(x)
258
259     def derivative(self, x):
260         return self.derivative(x)
261
262 def print_digits(X, ordered, m, n):
263     f, ax = plt.subplots(m, n)
264     ordered = get_ordered(X);
265     for i in range(m):
266         for j in range(n):
267             ordered[i*n+j] = ordered[i*n+j].reshape(28,28)
268             ax[i][j].imshow(ordered[i*n+j], cmap = plt.cm.binary, interpolation="nearest")
269             ax[i][j].axis("off")
270     plt.show()
271
272 def sigmoid_func(x):
273     return 1/(1+np.exp(-x))
274
275 def sigmoid_der(x):
276     return (x*(1-x))
277
278 def relu_func(X):
279     return np.maximum(0,X)
280
281 def relu_der(X):
282     X[X<0]=0
283     return X
284
285 def return_value(X):
286     return X
287
288 def gendata2(class_type,N):
289     m0 = np.array(
290         [[-0.132,0.320,1.672,2.230,1.217,-0.819,3.629,0.8210,1.808, 0.1700],
291          [-0.711,-1.726,0.139,1.151,-0.373,-1.573,-0.243,-0.5220,-0.511,0.5330]])
292
293     m1 = np.array(
294         [[-1.169,0.813,-0.859,-0.608,-0.832,2.015,0.173,1.432,0.743,1.0328],
295          [ 2.065,2.441,0.247,1.806,1.286,0.928,1.923,0.1299,1.847,-0.052]])
296
297     x = np.array([[[]],[[]])
298     for i in range(N):
299         idx = np.random.randint(10)
300         if class_type == 0:
301             m = m0[:,idx]
302         elif class_type == 1:
303             m = m1[:,idx]
304         else:
305             print("not a proper classifier")

```



```

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

```

```

379     xtot = np.r_[x0,x1]
380     xlim = [np.min(xtot[:,0]),np.max(xtot[:,0])]
381     ylim = [np.min(xtot[:,1]),np.max(xtot[:,1])]
382
383     fig = plt.figure() # make handle to save plot
384     plt.scatter(x0[:,0],x0[:,1],c='red',label='$x_0$')
385     plt.scatter(x1[:,0],x1[:,1],c='blue',label='$x_1$')
386     plt.xlabel('X Coordinate')
387     plt.ylabel('Y Coordinate')
388     plt.title("Neural Network (Two-class Boundary)")
389     plt.legend()
390
391 def plot_boundaries(xlim, ylim, equation):
392     xpl = np.linspace(xlim[0],xlim[1], num=100)
393     ypl = np.linspace(ylim[0],ylim[1], num=100)
394
395     red_pts = np.array([[[]],[[]])
396     blue_pts= np.array([[[]],[[]])
397     for x in xpl:
398         for y in ypl:
399             point = np.array([x,y]).reshape(1,2)
400             prob = equation(point)
401             if prob == 0:
402                 blue_pts = np.c_[blue_pts,[x,y]]
403             else:
404                 red_pts = np.c_[red_pts,[x,y]]
405
406     plt.scatter(blue_pts[0,:],blue_pts[1:],color='blue',s=0.25)
407     plt.scatter(red_pts[0,:],red_pts[1:],color='red',s=0.25)
408     plt.xlim(xlim)
409     plt.ylim(ylim)
410
411 if __name__ == '__main__':
412     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_classasgn_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
41 momentum_values = [0.0,0.8]
42 step_size = [0.4,0.7,0.9]
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             xt看 = 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()

```

```

101 class NeuralNetwork:
102     def __init__(self, layers, softmax=True, momentum=0,
103                 eta=0.1, MSE_freq=50, reg=0.001):
104         self.num_layers = len(layers)
105         self.num_outputs = layers[self.num_layers-1].num_neurons
106         self.error_array = []
107         self.error_plot = []
108         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         # iters_per_epoch = int(np.ceil(X.shape[0]/float(batch_size)))
147         total_iters = epochs
148
149         if total_iters > 5000:
150             print_frequency = total_iters/100
151         else:
152             print_frequency = total_iters/10
153             if print_frequency is 0:
154                 print_frequency += 1
155
156         completed_epocs = 0
157         for i in range(total_iters):
158             # batch = np.random.randint(0, X.shape[0], batch_size)
159             # self.train_data(X[batch], Y[batch])
160             self.train_data(X, Y)
161             if i%print_frequency is 0:
162                 print("Iteration %d MSE: %f"%(i+1, np.mean(self.error_array[-self.MSE_freq:])))
163
164             # create error plot
165             print("Final MSE: %f"%(np.mean(self.error_array[-self.MSE_freq:])))
166
167             plot = self.error_array[::-1]
168             for i in range(0, len(plot), self.MSE_freq):
169                 self.error_plot.append(np.mean(plot[i:i+self.MSE_freq]))
170             self.error_plot = self.error_plot[::-1]
171
172     def train_data(self, X, Y):

```

```

173     Yhat = self.forward_prop(X)
174     dE_dH = (Yhat-Y).T
175     iterlayers = iter(self.layers[:-1])
176
177     # back propagation
178     if self.softmax is True:
179         # divide by number of incoming batch size to regularize
180         dE_dWeight = (-np.dot(-dE_dH,self.layers[-1].weight_der) / \
181                        self.layers[-1].weight_der.shape[0])
182
183         # do not include the bias weights--not needed and will be updated later
184         dE_dH = np.dot(self.layers[-1].W[:,0:-1].T,dE_dH)
185
186         # update current weights with momentum
187         self.layers[-1].W += -self.eta*(dE_dWeight + \
188                                         self.momentum*self.layers[-1].momentum_matrix)
189         self.layers[-1].momentum_matrix = dE_dWeight
190
191         # skip the last layer if softmax
192         next(iterlayers)
193
194     for layer in iterlayers:
195         dE_dNet = layer.der(layer.output).T*dE_dH
196         dE_dWeight = (np.dot(dE_dNet,layer.weight_der)) / \
197                       layer.weight_der.shape[0]
198
199         dE_dH = np.dot(layer.W[:,0:-1].T,dE_dNet)
200
201         layer.W += -layer.momentum_matrix
202         layer.momentum_matrix = \
203             self.momentum * layer.momentum_matrix + \
204             self.eta * dE_dWeight
205
206     for indx,yhat in enumerate(Yhat):
207         self.error_array.append(sum((Y[indx]-yhat)*(Y[indx]-yhat)))
208
209     def stable_softmax(self, X):
210         exp_norm = np.exp(X - np.max(X))
211         return exp_norm / np.sum(exp_norm, axis=1).reshape((-1,1))
212
213     def plot_error(self,index,momentum,eta):
214         plt.plot(range(len(self.error_plot)), self.error_plot)
215         plt.xlabel("Mean set for %d Training Samples"%(self.MSE_freq))
216         plt.ylabel("Mean Squared Error")
217
218     def write_network_values(self, filename):
219         pickle.dump(self, open(filename, "w"))
220         print("Network written to: %s" %(filename))
221
222     def validate_results(self, Yhat, Y):
223         Yhat_enc = (np.arange(Y.shape[1]) == Yhat[:, None]).astype(float)
224         num_err = np.sum(abs(Yhat_enc - Y))/2
225         training_accuracy = (len(Yhat)-num_err)/len(Yhat)*100
226         print("%d Mistakes. Training Accuracy: %.2f%%"%(int(num_err),training_accuracy))
227         return training_accuracy
228
229     def set_initial_conditions(self):
230         # self.layers[0].W[0,:] = [0.15,0.2,0.35]
231         # self.layers[0].W[1,:] = [0.25,0.3,0.35]
232         # self.layers[0].W[2,:] = [0.25,0.3,0.35]
233
234         self.layers[0].W[0,:] = [0.1,0.1,0.01]
235         self.layers[0].W[1,:] = [0.2,0.2,0.1 ]
236         self.layers[0].W[2,:] = [0.3,0.3,0.1 ]
237
238 class layer:
239     def __init__(self,num_inputs,num_neurons, activation):
240         self.num_neurons = num_neurons
241         self.num_inputs = num_inputs
242         self.num_outputs = None
243         self.weight_der = None
244         self.activation = activation
245         self.net = None

```

```

246     self.W = np.random.uniform(0,1,[num_neurons,num_inputs+1])
247     self.momentum_matrix = np.zeros([num_neurons,num_inputs+1])
248     self.output = None
249
250
251     def forward(self, X):
252         self.weight_der = X
253         self.net = np.dot(X, self.W.T)
254         self.output = self.activation.function(self.net)
255         return self.output
256
257     def der(self, X):
258         return self.activation.derivative(X)
259
260     def set_initial_conditions(self):
261         print("test")
262
263 class activation_function:
264     def __init__(self,function,derivative):
265         self.function = function
266         self.derivative = derivative
267
268     def function(self,x):
269         return self.function(x)
270
271     def derivative(self,x):
272         return self.derivative(x)
273
274 def get_moon_class_data():
275     data = np.loadtxt("./data/classasgntrain1.dat",dtype=float)
276     x0 = data[:,0:2]
277     x1 = data[:,2:4]
278     data = data_frame(x0,x1)
279     return data.xtot,data.class_tot
280
281 def get_moon_gendata():
282     x0 = gendata2(0,10000)
283     x1 = gendata2(1,10000)
284     data = data_frame(x0,x1)
285     return data.xtot, data.class_tot
286
287 def get_classasgn_80_20():
288     data = np.loadtxt("./data/classasgntrain1.dat",dtype=float)
289     x0 = data[:,0:2]
290     x1 = data[:,2:4]
291     data = data_frame(x0,x1)
292     return data.train_tot,data.train_class_tot,data.test_data,data.test_class_tot
293
294 class data_frame:
295     def __init__(self, data0, data1):
296         self.x0 = data0
297         self.x1 = data1
298         self.xtot = np.r_[self.x0,self.x1]
299         self.N0 = self.x0.shape[0]
300         self.N1 = self.x1.shape[0]
301         self.N = self.N0 + self.N1
302         self.xlim = [np.min(self.xtot[:,0]),np.max(self.xtot[:,0])]
303         self.ylim = [np.min(self.xtot[:,1]),np.max(self.xtot[:,1])]
304         class_x0 = np.c_[np.zeros([self.N0,1]),np.ones([self.N0,1])]
305         class_x1 = np.c_[np.ones([self.N1,1]),np.zeros([self.N1,1])]
306         self.class_tot = np.r_[class_x0,class_x1]
307         self.y = np.r_[np.ones([self.N0,1]),np.zeros([self.N1,1])]
308
309         # create a training set from the classasgntrain1.dat
310         self.train_x0 = data0[0:80]
311         self.train_x1 = data1[0:80]
312         self.train_tot = np.r_[data0[0:80],data1[0:80]]
313         self.train_class_tot = np.r_[self.class_tot[0:80],self.class_tot[100:180]]
314         self.test_data = np.r_[data0[80:100],data1[80:100]]
315         self.test_class_tot = np.r_[self.class_tot[80:100],self.class_tot[180:200]]
316
317 def plot_data(x0,x1):
318     xt看 = np.r_[x0,x1]

```

```

319     xlim = [np.min(xtot[:,0]),np.max(xtot[:,0])]
320     ylim = [np.min(xtot[:,1]),np.max(xtot[:,1])]
321
322     fig = plt.figure() # make handle to save plot
323     plt.scatter(x0[:,0],x0[:,1],c='red',label='$x_0$')
324     plt.scatter(x1[:,0],x1[:,1],c='blue',label='$x_1$')
325     plt.xlabel('X Coordinate')
326     plt.ylabel('Y Coordinate')
327     plt.title("Neural Network (Two-class Boundary)")
328     plt.legend()
329
330 def plot_boundaries(xlim, ylim, equation):
331     xpl = np.linspace(xlim[0],xlim[1], num=100)
332     ypl = np.linspace(ylim[0],ylim[1], num=100)
333
334     red_pts = np.array([[[]],[[]]])
335     blue_pts= np.array([[[]],[[]]])
336     for x in xpl:
337         for y in ypl:
338             point = np.array([x,y]).reshape(1,2)
339             prob = equation(point)
340             if prob == 0:
341                 blue_pts = np.c_[blue_pts,[x,y]]
342             else:
343                 red_pts = np.c_[red_pts,[x,y]]
344
345     plt.scatter(blue_pts[0,:],blue_pts[1,:],color='blue',s=0.25)
346     plt.scatter(red_pts[0,:],red_pts[1,:],color='red',s=0.25)
347     plt.xlim(xlim)
348     plt.ylim(ylim)
349
350 def sigmoid_func(x):
351     return 1/(1+np.exp(-x))
352
353 def sigmoid_der(x):
354     return (x*(1-x))
355
356 def return_value(X):
357     return X
358
359 def relu_func(X):
360     return np.maximum(0,X)
361
362 def relu_der(X):
363     X[X<0]=0
364     return X
365
366 def softmax_func(x):
367     exps = np.exp(x)
368     return exps / np.sum(exps)
369
370 def gendata2(class_type,N):
371     m0 = np.array(
372         [[-0.132,0.320,1.672,2.230,1.217,-0.819,3.629,0.8210,1.808, 0.1700],
373          [-0.711,-1.726,0.139,1.151,-0.373,-1.573,-0.243,-0.5220,-0.511,0.5330]])
374
375     m1 = np.array(
376         [[-1.169,0.813,-0.859,-0.608,-0.832,2.015,0.173,1.432,0.743,1.0328],
377          [ 2.065,2.441,0.247,1.806,1.286,0.928,1.923,0.1299,1.847,-0.052]])
378
379     x = np.array([[[]],[[]]])
380     for i in range(N):
381         idx = np.random.randint(10)
382         if class_type == 0:
383             m = m0[:,idx]
384         elif class_type == 1:
385             m = m1[:,idx]
386         else:
387             print("not a proper classifier")
388             return 0
389     x = np.c_[x, [[m[0]],[m[1]]] + np.random.randn(2,1)/np.sqrt(5)]
390     return x.T
391

```

```

392 def get_ordered_digits(X_train):
393     ordered = [
394         X_train[7] , # 0
395         X_train[4] , # 1
396         X_train[16], # 2
397         X_train[1] , # 3
398         X_train[2] , # 4
399         X_train[27], # 5
400         X_train[3] , # 6
401         X_train[14], # 7
402         X_train[5] , # 8
403         X_train[8] , # 9
404     ]
405     return ordered
406
407 def print_digits(X,ordered,m,n):
408     f, ax = plt.subplots(m,n)
409     ordered = get_ordered(X);
410     for i in range(m):
411         for j in range(n):
412             ordered[i*n+j] = ordered[i*n+j].reshape(28,28)
413             ax[i][j].imshow(ordered[i*n+j], cmap = plt.cm.binary, interpolation="nearest")
414             ax[i][j].axis("off")
415
416     plt.show()
417
418 def get_mnist_train(file_path):
419     mnist = input_data.read_data_sets(file_path)
420     X = mnist.train.images
421     y = mnist.train.labels.astype("int")
422     Y = (np.arange(np.max(y) + 1) == y[:, None]).astype(float)
423     X_test = mnist.test.images
424     y_test = mnist.test.labels.astype("int")
425     Y_test = (np.arange(np.max(y_test) + 1) == y_test[:, None]).astype(float)
426     return X,Y,X_test,Y_test
427
428 def get_2_class_data():
429     X = np.array([[0.05, 0.1],
430                  [0.07, 0.1],
431                  [0.05, 0.1],
432                  [0.05, 0.1],
433                  [0.05, 0.1]])
434
435     Y = np.array([[0.01, 0.99],
436                  [0.01, 0.99],
437                  [0.01, 0.99],
438                  [0.01, 0.99],
439                  [0.01, 0.99]])
440     return X,Y
441
442 def get_3_class_data():
443     X = np.array([[0.05, 0.1],
444                  [0.07, 0.3],
445                  [0.09, 0.5],
446                  [0.05, 0.1]])
447
448     Y = np.array([[1, 0, 0],
449                  [0, 1, 0],
450                  [0, 0, 1],
451                  [1, 0, 0]])
452     return X,Y
453
454 def get_spiral_class_data():
455     np.random.seed(0)
456     N = 100 # number of points per class
457     D = 2 # dimensionality
458     K = 3 # number of classes
459     X = np.zeros((N*K,D))
460     y = np.zeros(N*K, dtype='uint8')
461     for j in xrange(K):
462         ix = range(N*j,N*(j+1))
463         r = np.linspace(0.0,1,N) # radius
464         t = np.linspace(j*4,(j+1)*4,N) + np.random.randn(N)*0.2 # theta

```



```
465     X[ix] = np.c_[r*np.sin(t), r*np.cos(t)]
466     y[ix] = j
467     # fig = plt.figure()
468     # plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap=plt.cm.Spectral)
469     # plt.xlim([-1,1])
470     # plt.ylim([-1,1])
471     Y = (np.arange(np.max(y) + 1) == y[:, None]).astype(float)
472     return X,Y
473
474 if __name__ == '__main__':
475     main()
```