Deep Neural Networks

Neural Networks: ECE 5930

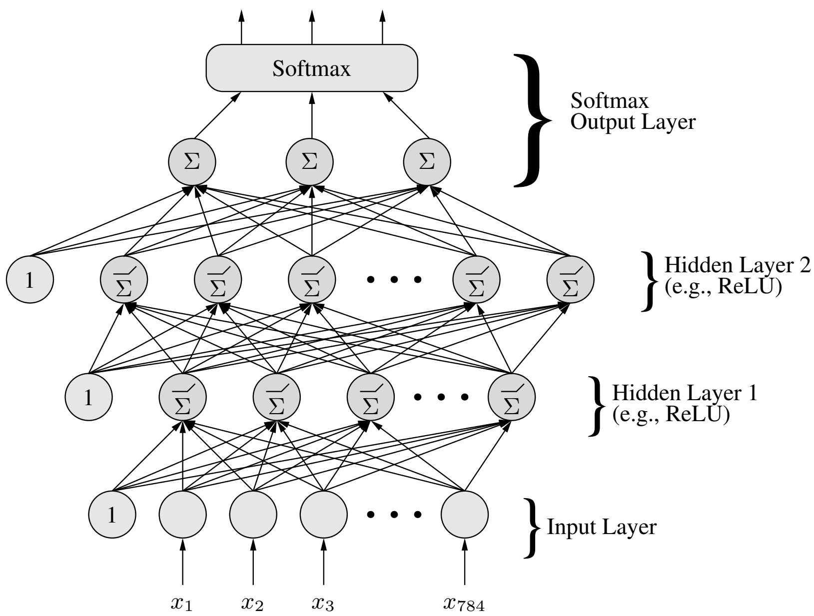


Figure: Two Hidden Layer Neural Network

Clint Ferrin
Utah State University
Mon Oct 23, 2017

Table of Contents

List of Figures

List of Tables

Listings

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 been seen in Figure 1.

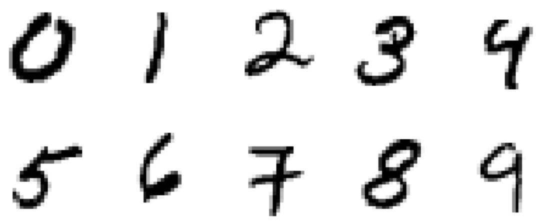


Figure 1: 10 Digits from the MNIST Data-set

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

```
# Clint Ferrin
# Oct 12, 2017
# Neural Network Classifier

import matplotlib.pyplot as plt
import numpy as np
import pickle
from   tensorflow.examples.tutorials.mnist import input_data
import time

def main():
    num_inputs = 784
    num_outputs= 10
    batch_size = 100
    epochs = 10
    mse_freq = 50

    # open mnist data
    X,Y,X_test,Y_test = get_mnist_train("./data")

    # initialize activation functions
    relu = activation_function(relu_func,relu_der)
    sig  = activation_function(sigmoid_func,sigmoid_der)
    no_activation = activation_function(return_value,return_value)

    num_neurons = 300
    # two hidden layers
    layers1 = [layer(num_inputs,num_neurons,relu)]
    layers1.append(layer(num_neurons,100,sig))
    layers1.append(layer(100,num_outputs,no_activation))

    # create neural network
    network = NeuralNetwork(layers,eta=0.9,momentum=0.8,softmax=True)

    # train network
    network.train_network(X,Y,batch_size=batch_size,
                          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

```
    def train_data(self, X, Y):
        Yhat = self.forward_prop(X)
        dE_dH = (Yhat-Y).T
        iterlayers = iter(self.layers[::-1])

        # back propagation
        if self.softmax is True:
            # divide by number of incoming batch size to regularize
            dE_dWeight = (-np.dot(-dE_dH,self.layers[-1].weight_der) / \
                          self.layers[-1].weight_der.shape[0])

            # do not include the bias weights--not needed and will be updated later
            dE_dH = np.dot(self.layers[-1].W[:,0:-1].T,dE_dH) * self.reg
                    # Yhat.shape[0]
```

```

            # update current weights with momentum
            self.layers[-1].W += -self.eta*(dE_dWeight + \
                    self.momentum*self.layers[-1].momentum_matrix)
            self.layers[-1].momentum_matrix = dE_dWeight

            # skip the last layer if softmax
            next(iterlayers)

        for layer in iterlayers:
            dE_dNet = layer.der(layer.output).T*dE_dH
            dE_dWeight = (np.dot(dE_dNet,layer.weight_der)) / \
                layer.weight_der.shape[0]

            dE_dH = np.dot(layer.W[:,0:-1].T,dE_dNet) * self.reg
                    # Yhat.shape[0]

            layer.W += -layer.momentum_matrix
            layer.momentum_matrix = \
                    self.momentum * layer.momentum_matrix + \
                    self.eta * dE_dWeight

        for indx,yhat in enumerate(Yhat):
            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 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:

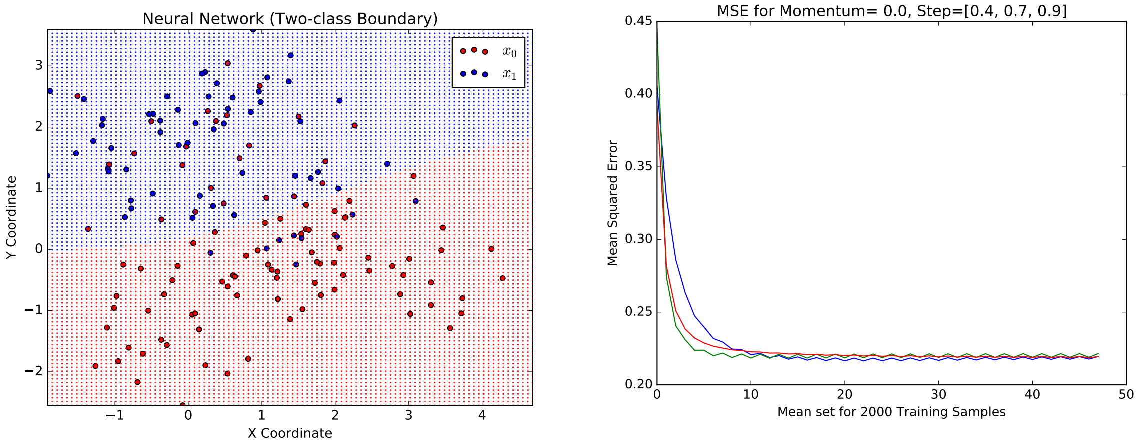
Listing 3: Creating 80% 20% Data

```
class data_frame:
    def __init__(self, data0, data1):
        self.x0 = data0
        self.x1 = data1
        self.xtot = np.r_[self.x0,self.x1]
        self.N0 = self.x0.shape[0]
        self.N1 = self.x1.shape[0]
        self.N = self.N0 + self.N1
        self.xlim = [np.min(self.xtot[:,0]),np.max(self.xtot[:,0])]
        self.ylim = [np.min(self.xtot[:,1]),np.max(self.xtot[:,1])]
        class_x0 = np.c_[np.zeros([self.N0,1]),np.ones([self.N0,1])]
        class_x1 = np.c_[np.ones([self.N1,1]),np.zeros([self.N1,1])]
        self.class_tot = np.r_[class_x0,class_x1]
        self.y = np.r_[np.ones([self.N0,1]),np.zeros([self.N1,1])]

        # create a training set from the classasgntrain1.dat (80% and 20%)
        self.train_x0 = data0[0:80]
        self.train_x1 = data1[0:80]
        self.train_tot = np.r_[data0[0:80],data1[0:80]]
        self.train_class_tot = np.r_[self.class_tot[0:80],self.class_tot[100:180]]
        self.test_data = np.r_[data0[80:100],data1[80:100]]
        self.test_class_tot = np.r_[self.class_tot[80:100],self.class_tot[180:200]]

def get_classasgn_80_20():
    data = np.loadtxt("./data/classasgntrain1.dat",dtype=float)
    x0 = data[:,0:2]
    x1 = data[:,2:4]
    data = data_frame(x0,x1)
    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.

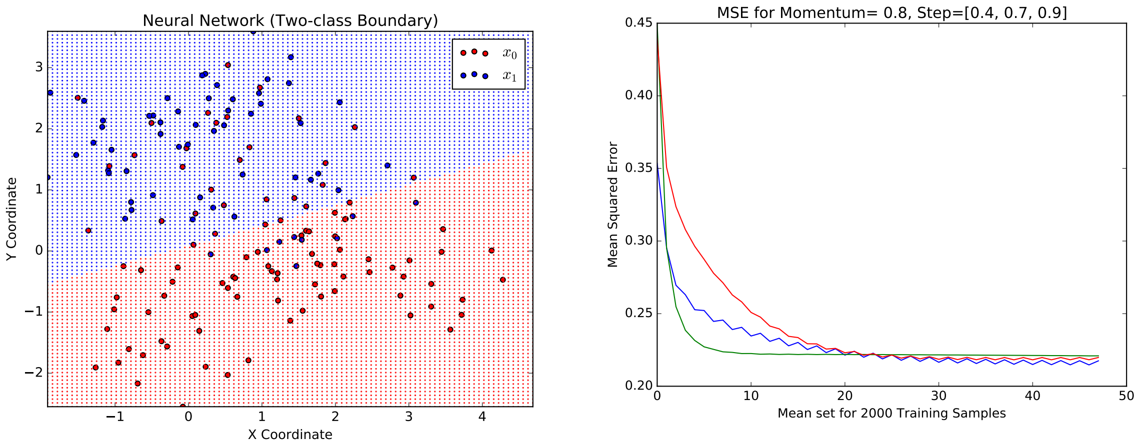


(a) 80% Training Data, 20% Testing Data

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

Figure 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.



(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 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.

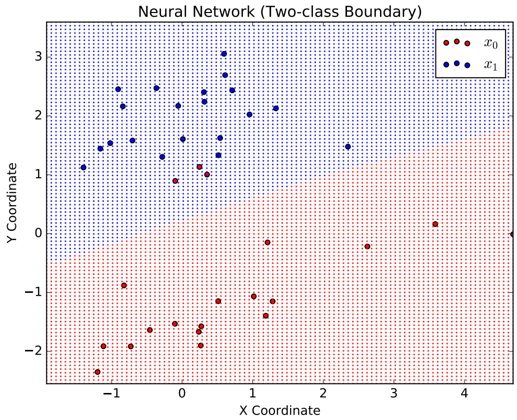
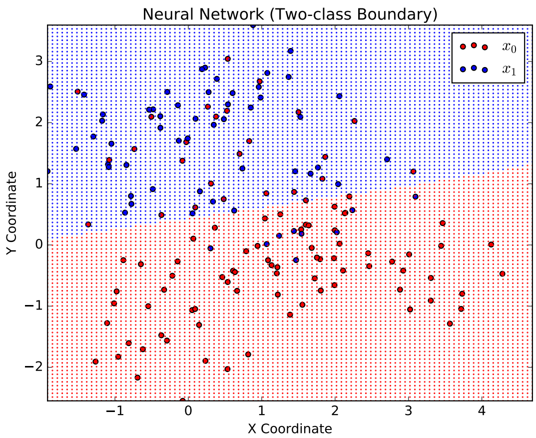
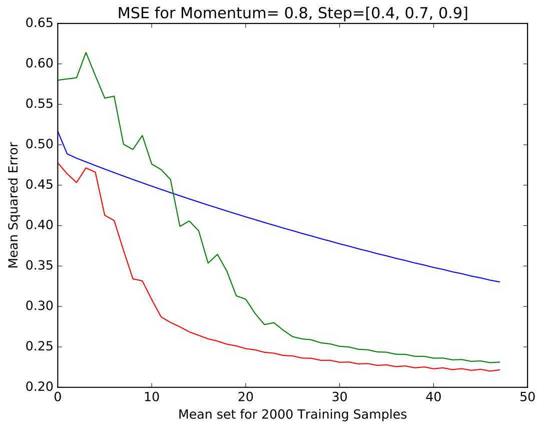


Figure 4: Three Miss-classified Points



(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

```
Two class layer with 1 hidden network (5 neurons). Epochs
mo-0.0-eta-0.4
Percent Correct: 92.5%
Run-time: 0.560807943344 seconds

mo-0.0-eta-0.7
Percent Correct: 92.5%
Run-time: 0.569748878479 seconds

mo-0.0-eta-0.9
Percent Correct: 92.5%
Run-time: 0.778621912003 seconds

mo-0.8-eta-0.4
Percent Correct: 92.5%
Run-time: 0.594919204712 seconds

```

```
mo-0.8-eta-0.7
Percent Correct: 92.5%
Run-time: 0.717699050903 seconds

mo-0.8-eta-0.9
Percent Correct: 92.5%
Run-time: 0.587964057922 seconds


Two class layer with 2 hidden networks (5 and 10 neurons respectively).
mo-0.0-eta-0.4
Percent Correct: 92.5%
Run-time: 0.685973882675 seconds

mo-0.0-eta-0.7
Percent Correct: 92.5%
Run-time: 0.68302488327 seconds

mo-0.0-eta-0.9
Percent Correct: 92.5%
Run-time: 0.728396892548 seconds

mo-0.8-eta-0.4
Percent Correct: 92.5%
Run-time: 0.695672035217 seconds

mo-0.8-eta-0.7
Percent Correct: 92.5%
Run-time: 0.681930780411 seconds

mo-0.8-eta-0.9
Percent Correct: 92.5%
Run-time: 0.666880846024 seconds
```

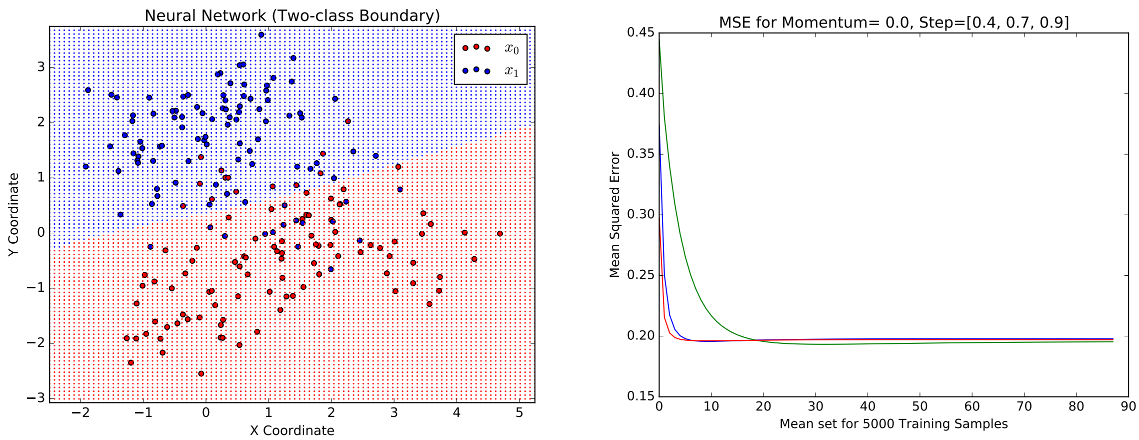
3.2 Comparing a Neural Network to Other Classifiers

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.

		Errors in %	
Method	train+run time	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.

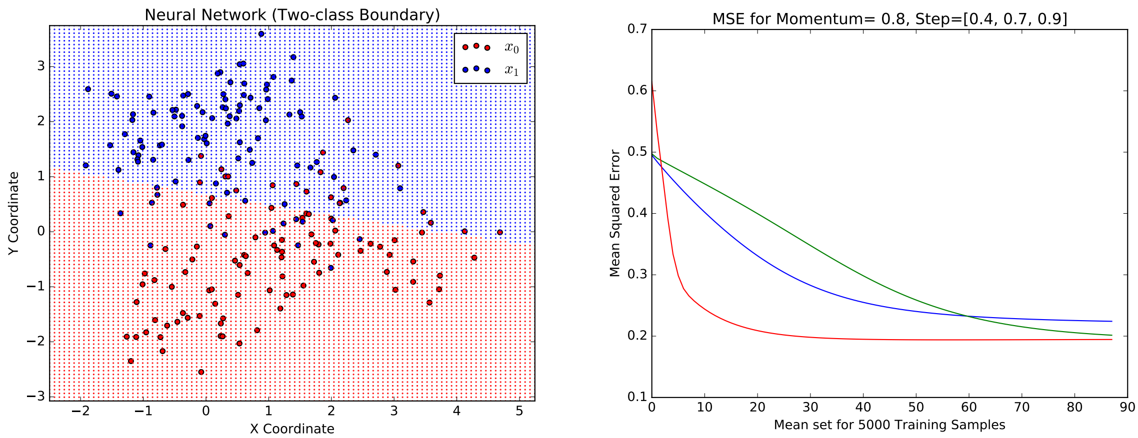


(a) Plotted Curve of Best Fit

(b) MSE plots for 0.4, 0.7, and 0.9 step sizes

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.



(a) Plotted Curve of Best Fit

(b) MSE plots for 0.4, 0.7, and 0.9 step sizes

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

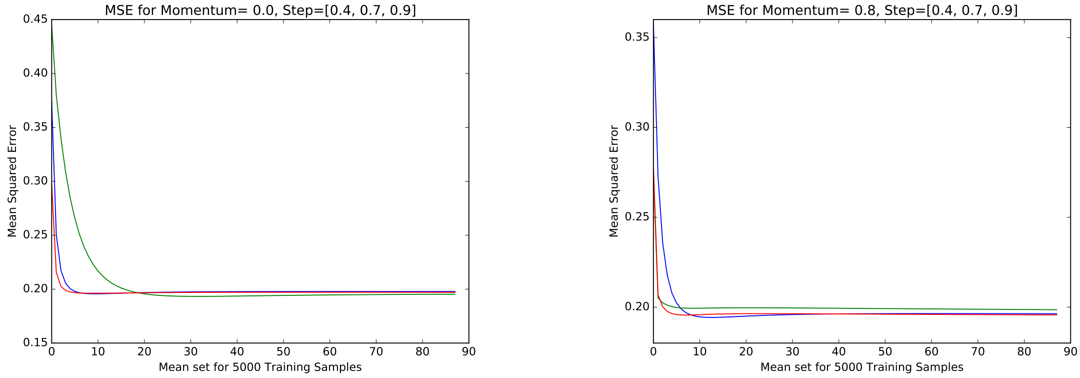
The Neural Network returned an error of 19.805%, which puts its results just behind the Bayes Optimal Classifier and the k-nearest neighbor approach. Because the 15-Nearest Neighbor is not practical with large datasets, and because a model for the Bayes Optimal Classifier is often impossible to find,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

		Errors in %	
Method	train+run time	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

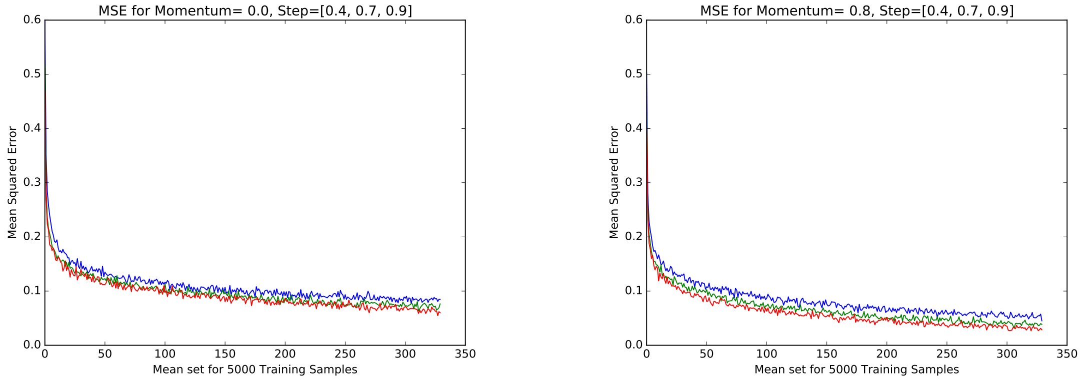
To test the MNIST data set, I created a network with 784 inputs, a hidden layer with 300 neurons, and an output of 10 classes connected to a softmax. The Mean Squared Error plot was set up to report the mean of every 50 iterations. As the assignment description asked, I used a `batch_size` of 100, and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

$$itrs = N/B \cdot epochs \tag{1}$$

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.



(a) MSE with 0 Momentum step size (b) MSE comparison of all weights

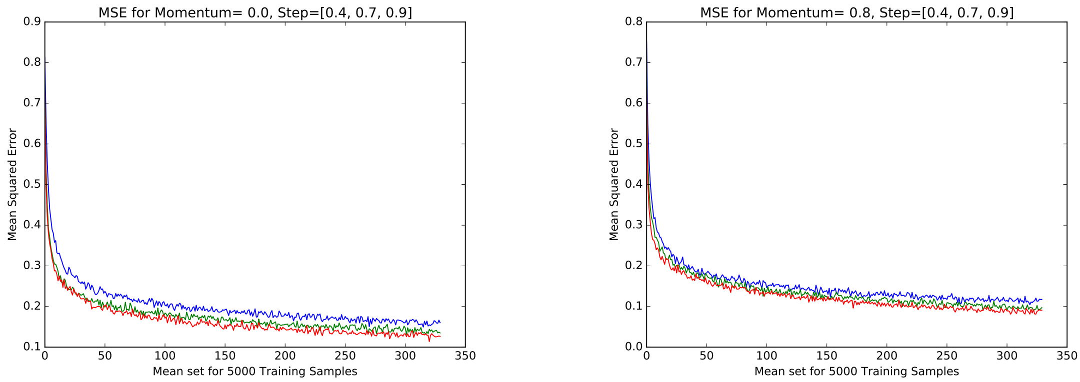
Figure 9: Comparing the Mean Square Error with Momentum 0.0 and 0.8

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.



(a) MSE with 0.0 momentum for a more complex NN (b) MSE with 0.8 momentum of a more complex NN

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

```
Layer with 1 hidden network (300 neurons). Epochs
mo-0.0-eta-0.4
Percent Correct: 94.13%
```

```
Run-time: 1083.04311395 seconds

mo-0.0-eta-0.7
Percent Correct: 94.72%
Run-time: 1078.50167799 seconds

mo-0.0-eta-0.9
Percent Correct: 94.82%
Run-time: 1076.73657393 seconds

mo-0.8-eta-0.4
Percent Correct: 95.61%
Run-time: 1076.8610909 seconds

mo-0.8-eta-0.7
Percent Correct: 96.27%
Run-time: 1077.98965693 seconds

mo-0.8-eta-0.9
Percent Correct: 96.89%
Run-time: 1079.15961313 seconds


Layer with 2 hidden networks (300 and 100 neurons respectively).
mo-0.0-eta-0.4
Percent Correct: 89.74%
Run-time: 1220.183357 seconds

mo-0.0-eta-0.7
Percent Correct: 90.68%
Run-time: 1221.12571001 seconds

mo-0.0-eta-0.9
Percent Correct: 91.56%
Run-time: 1220.0601058 seconds

mo-0.8-eta-0.4
Percent Correct: 92.22%
Run-time: 1220.07663107 seconds

mo-0.8-eta-0.7
Percent Correct: 93.34%
Run-time: 1222.40497899 seconds

mo-0.8-eta-0.9
Percent Correct: 93.88%
Run-time: 1224.18042397 seconds
```

5 Appendix

5.1 ten-class-classifier.py

Listing 6: Ten-class Classifier

```python
# Clint Ferrin
# Oct 12, 2017
# Neural Network Classifier

import matplotlib.pyplot as plt
import numpy as np
import pickle
from   tensorflow.examples.tutorials.mnist import input_data
import time

def main():
    num_inputs = 784
    num_outputs= 10
    batch_size = 100
```

```
    epochs = 10
    mse_freq = 50

    # open mnist data
    X,Y,X_test,Y_test = get_mnist_train("./data")

    # initialize activation functions
    relu = activation_function(relu_func,relu_der)
    sig  = activation_function(sigmoid_func,sigmoid_der)
    no_activation = activation_function(return_value,return_value)

    num_neurons = 300
    # first layer tests
    layers0 = [layer(num_inputs,num_neurons,relu)]
    layers0.append(layer(num_neurons,num_outputs,no_activation))

    # second layer tests
    layers1 = [layer(num_inputs,300,relu)]
    layers1.append(layer(300,100,relu))
    layers1.append(layer(100,num_outputs,no_activation))

    # set up test bench
    layer_testbench = [layers0,layers1]
    message = ["Layer with 1 hidden network (300 neurons). Epochs " +  "\n",
                "\nLayer with 2 hidden networks (300 and 100 neurons respectively).\n"]

    momentum_values = [0.0,0.8]
    step_size = [0.4,0.7,0.9]

    file = open('../report/media/mnist/ten-long-class-network_statistics-bat-'
            + str(batch_size) +
            '-mse-' + str(mse_freq) + '.txt',"w")

    for index, layers in enumerate(layer_testbench):
        file.write(message[index])
        for mom in momentum_values:
            for step in step_size:
                print("Currently on layer " + str(index) + " momentum " + str(mom) + " step size " + str(step))

                # create neural network
                network = NeuralNetwork(layers,eta=step,momentum=mom)

                # train network
                start_time = time.time()
                network.train_network(X,Y,batch_size=batch_size,
                                      epochs=epochs,MSE_freq=mse_freq)
                end_time = time.time()

                # classify data
                Yhat = network.classify_data(X_test)
                training_accuracy = network.validate_results(Yhat,Y_test)

                # write statistics
                file.write('mo-' + str(mom) + '-eta-' + str(step) + "\n")
                file.write("Percent Correct: " + str(training_accuracy) + "%\n")
                file.write("Run-time: " + str(end_time-start_time) +" seconds" + "\n\n")

                # plot error
                network.plot_error(index,mom,step)

            # save combined error plot
            plt.title("MSE for Momentum= " + str(mom) +
                      ", Step=" + str(step_size))
            plt.savefig('../report/media/mnist/ten-c-bat-' + str(batch_size) +
                      '-mse-' + str(mse_freq) + '-lay-' + str(index) +
                      '-mo-' + str(int(mom*10)) + '-eta-' + str(int(step*10)) +
                      '.pdf',bbox_inches='tight')
            plt.clf()

class NeuralNetwork:
    def __init__(self, layers, softmax=True, momentum=0,
                 eta=0.1, MSE_freq=50, reg=0.001):
```

```
        self.num_layers  = len(layers)
        self.num_outputs = layers[self.num_layers-1].num_neurons
        self.error_array = []
        self.error_plot  = []
        self.momentum = momentum
        self.MSE_freq = MSE_freq
        self.softmax= softmax
        self.layers = layers
        self.reg = reg
        self.eta = eta
        self.__set_GRV_starting_weights()

    def train_network(self, X, Y, batch_size=100, epochs=100, MSE_freq=50):
        self.MSE_freq = MSE_freq * batch_size
        print("Training Data...")

        # definition of iterations with mini-batch = N/B*epochs
        itrs_per_epoch = int(np.ceil(X.shape[0]/float(batch_size)))
        total_itrs = itrs_per_epoch * epochs

        # print out 100 samples to gauge speed of program
        if total_itrs > 5000:
            print_frequency = total_itrs/100

        # if iterations are few, print out 10
        else:
            print_frequency = total_itrs/10
            if print_frequency is 0:
                print_frequency += 1 # to avoid modulo by zero

        completed_epocs = 0
        for i in range(total_itrs):
            # randomly select samples from input data for batch
            batch = np.random.randint(0,X.shape[0],batch_size)
            self.train_data(X[batch],Y[batch])
            if i%itrs_per_epoch is 0:
                print("Epoch %d. MSE: %f"%(completed_epocs,
                    np.mean(self.error_array[-self.MSE_freq:])))
                completed_epocs += 1

            if i%print_frequency is 0:
                print("Iteration %d MSE: %f"%(i+1,
                    np.mean(self.error_array[-self.MSE_freq:])))


        # create error plot
        print("Final MSE: %f"%(np.mean(self.error_array[-self.MSE_freq:])))

        # reverse order of list and split into even parts sizeof=MSE_freq
        plot = self.error_array[::-1]
        for i in range(0,len(plot),self.MSE_freq):
            self.error_plot.append(np.mean(plot[i:i+self.MSE_freq]))
        self.error_plot = self.error_plot[::-1]

    def train_data(self, X, Y):
        Yhat = self.forward_prop(X)
        dE_dH = (Yhat-Y).T
        iterlayers = iter(self.layers[::-1])

        # back propagation
        if self.softmax is True:
            # divide by number of incoming batch size to regularize
            dE_dWeight = (-np.dot(-dE_dH,self.layers[-1].weight_der) / \
                          self.layers[-1].weight_der.shape[0])

            # do not include the bias weights--not needed and will be updated later
            dE_dH = np.dot(self.layers[-1].W[:,0:-1].T,dE_dH) * self.reg
                    # Yhat.shape[0]

            # update current weights with momentum
            self.layers[-1].W += -self.eta*(dE_dWeight + \
                    self.momentum*self.layers[-1].momentum_matrix)
            self.layers[-1].momentum_matrix = dE_dWeight
```

```

            # skip the last layer if softmax
            next(iterlayers)

        for layer in iterlayers:
            dE_dNet = layer.der(layer.output).T*dE_dH
            dE_dWeight = (np.dot(dE_dNet,layer.weight_der)) / \
                layer.weight_der.shape[0]

            dE_dH = np.dot(layer.W[:,0:-1].T,dE_dNet) * self.reg
                    # Yhat.shape[0]

            layer.W += -layer.momentum_matrix
            layer.momentum_matrix = \
                    self.momentum * layer.momentum_matrix + \
                    self.eta * dE_dWeight

        for indx,yhat in enumerate(Yhat):
            self.error_array.append(sum((Y[indx]-yhat)*(Y[indx]-yhat)))

    def forward_prop(self, X):
        prev_out = X
        for layer in self.layers:
            prev_out = np.c_[prev_out,np.ones([prev_out.shape[0],1])]
            prev_out = layer.forward(prev_out)

        if self.softmax is True:
            self.layers[-1].output = self.stable_softmax(self.layers[-1].net)

        return self.layers[-1].output

    def classify_data(self, X):
        Yhat = self.forward_prop(X)
        class_type = np.argmax(Yhat,axis=1)
        # returns list instead of matrix
        return class_type

    def stable_softmax(self, X):
        exp_norm = np.exp(X - np.max(X))
        return exp_norm / np.sum(exp_norm, axis=1).reshape((-1,1))

    def validate_results(self, Yhat, Y):
        Yhat_enc = (np.arange(Y.shape[1]) == Yhat[:, None]).astype(float)
        num_err = np.sum(abs(Yhat_enc - Y))/2
        training_accuracy = (len(Yhat)-num_err)/len(Yhat)*100
        print("%d Mistakes. Training Accuracy: %.2f%%"%(int(num_err),training_accuracy))
        return training_accuracy

    def plot_error(self,index,momentum,eta):
        plt.plot(range(len(self.error_plot)), self.error_plot)
        plt.xlabel("Mean set for %d Training Samples"%(self.MSE_freq))
        plt.ylabel("Mean Squared Error")

    def write_network_values(self, filename):
        pickle.dump(self, open(filename, "we"))
        print("Network written to: %s" %(filename))

    def __set_GRV_starting_weights(self):
        # find number of outputs at each layer
        for i in range(self.num_layers-2):
            self.layers[i].num_outputs = self.layers[i+1].num_neurons
        self.layers[-1].num_outputs = self.num_outputs

        for layer in self.layers:
            sigma = np.sqrt(float(2) / (layer.num_inputs + layer.num_outputs))
            layer.W = np.random.normal(0,sigma,layer.W.shape)

class layer:
    def __init__(self,num_inputs,num_neurons, activation):
        self.W = np.random.uniform(0,1,[num_neurons,num_inputs+1])
        self.momentum_matrix = np.zeros([num_neurons,num_inputs+1])
        self.num_neurons = num_neurons
        self.num_inputs  = num_inputs
```

```
        self.activation  = activation
        self.num_outputs = None
        self.weight_der  = None
        self.net     = None
        self.output = None

    def forward(self, X):
        self.weight_der = X
        self.net = np.dot(X, self.W.T)
        self.output = self.activation.function(self.net)
        return self.output

    def der(self, X):
        return self.activation.derivative(X)

    def set_initial_conditions(self):
        print("test")

class activation_function:
    def __init__(self,function,derivative):
        self.function = function
        self.derivative = derivative

    def function(self,x):
        return self.function(x)

    def derivative(self,x):
        return self.derivative(x)

def print_digits(X,ordered,m,n):
    f, ax = plt.subplots(m,n)
    ordered = get_ordered(X);
    for i in range(m):
        for j in range(n):
            ordered[i*n+j] = ordered[i*n+j].reshape(28,28)
            ax[i][j].imshow(ordered[i*n+j], cmap = plt.cm.binary, interpolation="nearest")
            ax[i][j].axis("off")
    plt.show()

def sigmoid_func(x):
    return 1/(1+np.exp(-x))

def sigmoid_der(x):
    return (x*(1-x))

def relu_func(X):
    return np.maximum(0,X)

def relu_der(X):
    X[X<0]=0
    return X

def return_value(X):
    return X

def gendata2(class_type,N):
    m0 = np.array(
         [[-0.132,0.320,1.672,2.230,1.217,-0.819,3.629,0.8210,1.808, 0.1700],
          [-0.711,-1.726,0.139,1.151,-0.373,-1.573,-0.243,-0.5220,-0.511,0.5330]])

    m1 = np.array(
          [[-1.169,0.813,-0.859,-0.608,-0.832,2.015,0.173,1.432,0.743,1.0328],
          [ 2.065,2.441,0.247,1.806,1.286,0.928,1.923,0.1299,1.847,-0.052]])

    x = np.array([[],[]])
    for i in range(N):
        idx = np.random.randint(10)
        if class_type == 0:
            m = m0[:,idx]
        elif class_type == 1:
            m = m1[:,idx]
        else:
            print("not a proper classifier")
```

```
            return 0
        x = np.c_[x, [[m[0]],[m[1]]] + np.random.randn(2,1)/np.sqrt(5)]
    return x.T

def get_ordered_digits(X_train):
    ordered = [
            X_train[7] , # 0
            X_train[4] , # 1
            X_train[16], # 2
            X_train[1] , # 3
            X_train[2] , # 4
            X_train[27], # 5
            X_train[3] , # 6
            X_train[14], # 7
            X_train[5] , # 8
            X_train[8] , # 9
            ]
    return ordered

def get_moon_class_data():
    data = np.loadtxt("./data/classasgntrain1.dat",dtype=float)
    x0 = data[:,0:2]
    x1 = data[:,2:4]
    data = data_frame(x0,x1)
    return data.xtot,data.class_tot

def get_moon_gendata():
    x0 = gendata2(0,10000)
    x1 = gendata2(1,10000)
    data = data_frame(x0,x1)
    return data.xtot, data.class_tot

class data_frame:
    def __init__(self, data0, data1):
        self.x0 = data0
        self.x1 = data1
        self.xtot = np.r_[self.x0,self.x1]
        self.N0 = self.x0.shape[0]
        self.N1 = self.x1.shape[0]
        self.N = self.N0 + self.N1
        self.xlim = [np.min(self.xtot[:,0]),np.max(self.xtot[:,0])]
        self.ylim = [np.min(self.xtot[:,1]),np.max(self.xtot[:,1])]
        class_x0 = np.c_[np.zeros([self.N0,1]),np.ones([self.N0,1])]
        class_x1 = np.c_[np.ones([self.N1,1]),np.zeros([self.N1,1])]
        self.class_tot = np.r_[class_x0,class_x1]
        self.y = np.r_[np.ones([self.N0,1]),np.zeros([self.N1,1])]

        # create a training set from the classasgntrain1.dat (80% and 20%)
        self.train_x0 = data0[0:80]
        self.train_x1 = data1[0:80]
        self.train_tot = np.r_[data0[0:80],data1[0:80]]
        self.train_class_tot = np.r_[self.class_tot[0:80],self.class_tot[100:180]]
        self.test_data = np.r_[data0[80:100],data1[80:100]]
        self.test_class_tot = np.r_[self.class_tot[80:100],self.class_tot[180:200]]

def get_classasgn_80_20():
    data = np.loadtxt("./data/classasgntrain1.dat",dtype=float)
    x0 = data[:,0:2]
    x1 = data[:,2:4]
    data = data_frame(x0,x1)
    return data.train_tot,data.train_class_tot,data.test_data,data.test_class_tot

def get_mnist_train(file_path):
    mnist = input_data.read_data_sets(file_path)
    X = mnist.train.images
    y = mnist.train.labels.astype("int")
    Y = (np.arange(np.max(y) + 1) == y[:, None]).astype(float)
    X_test = mnist.test.images
    y_test = mnist.test.labels.astype("int")
    Y_test = (np.arange(np.max(y_test) + 1) == y_test[:, None]).astype(float)
    return X,Y,X_test,Y_test

def plot_data(x0,x1):
```

```
    xtot = np.r_[x0,x1]
    xlim = [np.min(xtot[:,0]),np.max(xtot[:,0])]
    ylim = [np.min(xtot[:,1]),np.max(xtot[:,1])]

    fig = plt.figure() # make handle to save plot
    plt.scatter(x0[:,0],x0[:,1],c='red',label='$x_0$')
    plt.scatter(x1[:,0],x1[:,1],c='blue',label='$x_1$')
    plt.xlabel('X Coordinate')
    plt.ylabel('Y Coordinate')
    plt.title("Neural Network (Two-class Boundary)")
    plt.legend()

def plot_boundaries(xlim, ylim, equation):
    xp1 = np.linspace(xlim[0],xlim[1], num=100)
    yp1 = np.linspace(ylim[0],ylim[1], num=100)

    red_pts = np.array([[],[]])
    blue_pts= np.array([[],[]])
    for x in xp1:
        for y in yp1:
            point = np.array([x,y]).reshape(1,2)
            prob = equation(point)
            if prob == 0:
                blue_pts = np.c_[blue_pts,[x,y]]
            else:
                red_pts = np.c_[red_pts,[x,y]]

    plt.scatter(blue_pts[0,:],blue_pts[1,:],color='blue',s=0.25)
    plt.scatter(red_pts[0,:],red_pts[1,:],color='red',s=0.25)
    plt.xlim(xlim)
    plt.ylim(ylim)

if __name__ == '__main__':
  main()
```

5.2 two-class-classifier.py

Listing 7: Two-class Classifier

```
# Clint Ferrin
# Oct 12, 2017
# Neural Network Classifier

import matplotlib.pyplot as plt
import numpy as np
import pickle
from   tensorflow.examples.tutorials.mnist import input_data
import time

def main():
    num_inputs = 2
    num_outputs= 2
    batch_size = 1 # not used. All data used
    epochs = 2000
    mse_freq = 200

    # open mnist data
    X,Y,X_test,Y_test = get_classasgn_80_20()
    # Y = Y[:,1].reshape(-1,1)
    # Y_test = Y_test[:,1].reshape(-1,1)

    # initialize activation functions
    relu = activation_function(relu_func,relu_der)
    sig  = activation_function(sigmoid_func,sigmoid_der)
    no_activation = activation_function(return_value,return_value)

    # first layer tests
    layers0 = [layer(num_inputs,5,sig)]
    layers0.append(layer(5,num_outputs,sig))

```

```
    layers1 = [layer(num_inputs,5,sig)]
    layers1.append(layer(5,10,sig))
    layers1.append(layer(5,num_outputs,sig))

    layer_testbench = [layers0,layers1]

    message = ["Two class layer with 1 hidden network (5 neurons). Epochs " +  "\n",
               "\nTwo class layer with 2 hidden networks (5 and 10 neurons respectively).\n"]

    momentum_values = [0.0,0.8]
    step_size = [0.4,0.7,0.9]

    file = open('../report/media/two-class-80-20/test/two-class-net-80-20-statistics-bat-' +
        str(batch_size) + '-mse-' + str(mse_freq) + '.txt',"w")

    for index, layers in enumerate(layer_testbench):
        file.write(message[index])
        plt.clf()
        for mom in momentum_values:
            for step in step_size:
                print("Currently on layer " + str(index) + " momentum " + str(mom) + " step
                    size " + str(step))

                # create neural network
                network = NeuralNetwork(layers,eta=step,momentum=mom,softmax=False)

                # train network
                start_time = time.time()
                network.train_network(X,Y,batch_size=batch_size,
                                      epochs=epochs,MSE_freq=mse_freq)
                end_time = time.time()

                # classify data
                Yhat = network.classify_data(X_test)
                training_accuracy = network.validate_results(Yhat,Y_test)

                file.write('mo-' + str(mom) + '-eta-' + str(step) + "\n")
                file.write("Percent Correct: " + str(training_accuracy) + "%\n")
                file.write("Run-time: " + str(end_time-start_time) +" seconds" + "\n\n")

                # plot data points and graph boundaries
                plt.figure(1)
                plt.clf()
                plot_data(X_test[0:20],X_test[20:40])

                xtot = np.r_[X,X_test]
                xlim = [np.min(xtot[:,0]),np.max(xtot[:,0])]
                ylim = [np.min(xtot[:,1]),np.max(xtot[:,1])]

                plot_boundaries(xlim,ylim,network.classify_data)
                plt.savefig('../report/media/two-class-80-20/test/two-c-net-80-20-bat-' + str(
                    batch_size) +
                        '-mse-' + str(mse_freq) + '-lay-' + str(index) +
                        '-mo-' + str(int(mom*10)) + '-eta-' + str(int(step*10)) +
                        '.pdf',bbox_inches='tight')
                plt.show()
                plt.clf()

                plt.figure(2)
                # plot error and graph boundaries
                network.plot_error(index,mom,step)

            # save combined error plots
            plt.title("MSE for Momentum= " + str(mom) +
                      ", Step=" + str(step_size))
            plt.savefig('../report/media/two-class-80-20/two-c-error-80-20-bat-' + str(
                batch_size) +
                      '-mse-' + str(mse_freq) + '-lay-' + str(index) +
                      '-mo-' + str(int(mom*10)) + '-eta-' + str(int(step*10)) +
                      '.pdf',bbox_inches='tight')
            plt.show()
            plt.clf()
```

```
class NeuralNetwork:
    def __init__(self, layers, softmax=True, momentum=0,
                 eta=0.1, MSE_freq=50, reg=0.001):
        self.num_layers  = len(layers)
        self.num_outputs = layers[self.num_layers-1].num_neurons
        self.error_array = []
        self.error_plot  = []
        self.momentum = momentum
        self.MSE_freq = MSE_freq
        self.softmax= softmax
        self.layers = layers
        self.reg = reg
        self.eta = eta
        self.__set_GRV_starting_weights()

    def __set_GRV_starting_weights(self):
        for i in range(self.num_layers-2):
            self.layers[i].num_outputs = self.layers[i+1].num_neurons
        self.layers[-1].num_outputs = self.num_outputs

        for layer in self.layers:
            sigma = np.sqrt(float(2) / (layer.num_inputs + layer.num_neurons))
            layer.W = np.random.normal(0,sigma,layer.W.shape)

    def forward_prop(self, X):
        prev_out = X
        for layer in self.layers:
            prev_out = np.c_[prev_out,np.ones([prev_out.shape[0],1])]
            prev_out = layer.forward(prev_out)

        if self.softmax is True:
            self.layers[-1].output = self.stable_softmax(self.layers[-1].net)

        return self.layers[-1].output

    def classify_data(self, X):
        Yhat = self.forward_prop(X)
        class_type = np.argmax(Yhat,axis=1)
        return class_type

    def train_network(self, X, Y, batch_size=100, epochs=100, MSE_freq=50):
        self.MSE_freq = MSE_freq * batch_size
        print("Training Data...")

        # definition of iterations with mini-batch = N/B*epochs
        # itrs_per_epoch = int(np.ceil(X.shape[0]/float(batch_size)))
        total_itrs = epochs

        if total_itrs > 5000:
            print_frequency = total_itrs/100
        else:
            print_frequency = total_itrs/10
            if print_frequency is 0:
                print_frequency += 1

        completed_epocs = 0
        for i in range(total_itrs):
            # batch = np.random.randint(0,X.shape[0],batch_size)
            # self.train_data(X[batch],Y[batch])
            self.train_data(X,Y)
            if i%print_frequency is 0:
                print("Iteration %d MSE: %f"%(i+1, np.mean(self.error_array[-self.MSE_freq:]))
                    )

        # create error plot
        print("Final MSE: %f"%(np.mean(self.error_array[-self.MSE_freq:])))

        plot = self.error_array[::-1]
        for i in range(0,len(plot),self.MSE_freq):
            self.error_plot.append(np.mean(plot[i:i+self.MSE_freq]))
        self.error_plot = self.error_plot[::-1]

    def train_data(self, X, Y):
```

```
        Yhat = self.forward_prop(X)
        dE_dH = (Yhat-Y).T
        iterlayers = iter(self.layers[::-1])

        # back propagation
        if self.softmax is True:
            # divide by number of incoming batch size to regularize
            dE_dWeight = (-np.dot(-dE_dH,self.layers[-1].weight_der) / \
                          self.layers[-1].weight_der.shape[0])

            # do not include the bias weights--not needed and will be updated later
            dE_dH = np.dot(self.layers[-1].W[:,0:-1].T,dE_dH)

            # update current weights with momentum
            self.layers[-1].W += -self.eta*(dE_dWeight + \
                    self.momentum*self.layers[-1].momentum_matrix)
            self.layers[-1].momentum_matrix = dE_dWeight

            # skip the last layer if softmax
            next(iterlayers)

        for layer in iterlayers:
            dE_dNet = layer.der(layer.output).T*dE_dH
            dE_dWeight = (np.dot(dE_dNet,layer.weight_der)) / \
                layer.weight_der.shape[0]

            dE_dH = np.dot(layer.W[:,0:-1].T,dE_dNet)

            layer.W += -layer.momentum_matrix
            layer.momentum_matrix = \
                    self.momentum * layer.momentum_matrix + \
                    self.eta * dE_dWeight

        for indx,yhat in enumerate(Yhat):
            self.error_array.append(sum((Y[indx]-yhat)*(Y[indx]-yhat)))

    def stable_softmax(self, X):
        exp_norm = np.exp(X - np.max(X))
        return exp_norm / np.sum(exp_norm, axis=1).reshape((-1,1))

    def plot_error(self,index,momentum,eta):
        plt.plot(range(len(self.error_plot)), self.error_plot)
        plt.xlabel("Mean set for %d Training Samples"%(self.MSE_freq))
        plt.ylabel("Mean Squared Error")

    def write_network_values(self, filename):
        pickle.dump(self, open(filename, "we"))
        print("Network written to: %s" %(filename))

    def validate_results(self, Yhat, Y):
        Yhat_enc = (np.arange(Y.shape[1]) == Yhat[:, None]).astype(float)
        num_err = np.sum(abs(Yhat_enc - Y))/2
        training_accuracy = (len(Yhat)-num_err)/len(Yhat)*100
        print("%d Mistakes. Training Accuracy: %.2f%%"%(int(num_err),training_accuracy))
        return training_accuracy

    def set_initial_conditions(self):
        # self.layers[0].W[0,:] = [0.15,0.2,0.35]
        # self.layers[0].W[1,:] = [0.25,0.3,0.35]
        # self.layers[0].W[2,:] = [0.25,0.3,0.35]

        self.layers[0].W[0,:] = [0.1,0.1,0.01]
        self.layers[0].W[1,:] = [0.2,0.2,0.1 ]
        self.layers[0].W[2,:] = [0.3,0.3,0.1 ]

class layer:
    def __init__(self,num_inputs,num_neurons, activation):
        self.num_neurons = num_neurons
        self.num_inputs = num_inputs
        self.num_outputs = None
        self.weight_der = None
        self.activation = activation
        self.net = None
```

```
        self.W = np.random.uniform(0,1,[num_neurons,num_inputs+1])
        self.momentum_matrix = np.zeros([num_neurons,num_inputs+1])
        self.output = None


    def forward(self, X):
        self.weight_der = X
        self.net = np.dot(X, self.W.T)
        self.output = self.activation.function(self.net)
        return self.output

    def der(self, X):
        return self.activation.derivative(X)

    def set_initial_conditions(self):
        print("test")

class activation_function:
    def __init__(self,function,derivative):
        self.function = function
        self.derivative = derivative

    def function(self,x):
        return self.function(x)

    def derivative(self,x):
        return self.derivative(x)

def get_moon_class_data():
    data = np.loadtxt("./data/classasgntrain1.dat",dtype=float)
    x0 = data[:,0:2]
    x1 = data[:,2:4]
    data = data_frame(x0,x1)
    return data.xtot,data.class_tot

def get_moon_gendata():
    x0 = gendata2(0,10000)
    x1 = gendata2(1,10000)
    data = data_frame(x0,x1)
    return data.xtot, data.class_tot

def get_classasgn_80_20():
    data = np.loadtxt("./data/classasgntrain1.dat",dtype=float)
    x0 = data[:,0:2]
    x1 = data[:,2:4]
    data = data_frame(x0,x1)
    return data.train_tot,data.train_class_tot,data.test_data,data.test_class_tot

class data_frame:
    def __init__(self, data0, data1):
        self.x0 = data0
        self.x1 = data1
        self.xtot = np.r_[self.x0,self.x1]
        self.N0 = self.x0.shape[0]
        self.N1 = self.x1.shape[0]
        self.N = self.N0 + self.N1
        self.xlim = [np.min(self.xtot[:,0]),np.max(self.xtot[:,0])]
        self.ylim = [np.min(self.xtot[:,1]),np.max(self.xtot[:,1])]
        class_x0 = np.c_[np.zeros([self.N0,1]),np.ones([self.N0,1])]
        class_x1 = np.c_[np.ones([self.N1,1]),np.zeros([self.N1,1])]
        self.class_tot = np.r_[class_x0,class_x1]
        self.y = np.r_[np.ones([self.N0,1]),np.zeros([self.N1,1])]

        # create a training set from the classasgntrain1.dat
        self.train_x0 = data0[0:80]
        self.train_x1 = data1[0:80]
        self.train_tot = np.r_[data0[0:80],data1[0:80]]
        self.train_class_tot = np.r_[self.class_tot[0:80],self.class_tot[100:180]]
        self.test_data = np.r_[data0[80:100],data1[80:100]]
        self.test_class_tot = np.r_[self.class_tot[80:100],self.class_tot[180:200]]

def plot_data(x0,x1):
    xtot = np.r_[x0,x1]
```

```python
    xlim = [np.min(xtot[:,0]),np.max(xtot[:,0])]
    ylim = [np.min(xtot[:,1]),np.max(xtot[:,1])]

    fig = plt.figure() # make handle to save plot
    plt.scatter(x0[:,0],x0[:,1],c='red',label='$x_0$')
    plt.scatter(x1[:,0],x1[:,1],c='blue',label='$x_1$')
    plt.xlabel('X Coordinate')
    plt.ylabel('Y Coordinate')
    plt.title("Neural Network (Two-class Boundary)")
    plt.legend()

def plot_boundaries(xlim, ylim, equation):
    xp1 = np.linspace(xlim[0],xlim[1], num=100)
    yp1 = np.linspace(ylim[0],ylim[1], num=100)

    red_pts = np.array([[],[]])
    blue_pts= np.array([[],[]])
    for x in xp1:
        for y in yp1:
            point = np.array([x,y]).reshape(1,2)
            prob = equation(point)
            if prob == 0:
                blue_pts = np.c_[blue_pts,[x,y]]
            else:
                red_pts = np.c_[red_pts,[x,y]]

    plt.scatter(blue_pts[0,:],blue_pts[1,:],color='blue',s=0.25)
    plt.scatter(red_pts[0,:],red_pts[1,:],color='red',s=0.25)
    plt.xlim(xlim)
    plt.ylim(ylim)

def sigmoid_func(x):
    return 1/(1+np.exp(-x))

def sigmoid_der(x):
    return (x*(1-x))

def return_value(X):
    return X

def relu_func(X):
    return np.maximum(0,X)

def relu_der(X):
    X[X<0]=0
    return X

def softmax_func(x):
    exps = np.exp(x)
    return exps / np.sum(exps)

def gendata2(class_type,N):
    m0 = np.array(
         [[-0.132,0.320,1.672,2.230,1.217,-0.819,3.629,0.8210,1.808, 0.1700],
          [-0.711,-1.726,0.139,1.151,-0.373,-1.573,-0.243,-0.5220,-0.511,0.5330]])

    m1 = np.array(
          [[-1.169,0.813,-0.859,-0.608,-0.832,2.015,0.173,1.432,0.743,1.0328],
          [ 2.065,2.441,0.247,1.806,1.286,0.928,1.923,0.1299,1.847,-0.052]])

    x = np.array([[],[]])
    for i in range(N):
        idx = np.random.randint(10)
        if class_type == 0:
            m = m0[:,idx]
        elif class_type == 1:
            m = m1[:,idx]
        else:
            print("not a proper classifier")
            return 0
        x = np.c_[x, [[m[0]],[m[1]]] + np.random.randn(2,1)/np.sqrt(5)]
    return x.T

```

```
def get_ordered_digits(X_train):
    ordered = [
            X_train[7] , # 0
            X_train[4] , # 1
            X_train[16], # 2
            X_train[1] , # 3
            X_train[2] , # 4
            X_train[27], # 5
            X_train[3] , # 6
            X_train[14], # 7
            X_train[5] , # 8
            X_train[8] , # 9
            ]
    return ordered

def print_digits(X,ordered,m,n):
    f, ax = plt.subplots(m,n)
    ordered = get_ordered(X);
    for i in range(m):
        for j in range(n):
            ordered[i*n+j] = ordered[i*n+j].reshape(28,28)
            ax[i][j].imshow(ordered[i*n+j], cmap = plt.cm.binary, interpolation="nearest")
            ax[i][j].axis("off")

    plt.show()

def get_mnist_train(file_path):
    mnist = input_data.read_data_sets(file_path)
    X = mnist.train.images
    y = mnist.train.labels.astype("int")
    Y = (np.arange(np.max(y) + 1) == y[:, None]).astype(float)
    X_test = mnist.test.images
    y_test = mnist.test.labels.astype("int")
    Y_test = (np.arange(np.max(y_test) + 1) == y_test[:, None]).astype(float)
    return X,Y,X_test,Y_test

def get_2_class_data():
    X = np.array([[0.05, 0.1],
                  [0.07, 0.1],
                  [0.05, 0.1],
                  [0.05, 0.1],
                  [0.05, 0.1]])

    Y = np.array([[0.01, 0.99],
                  [0.01, 0.99],
                  [0.01, 0.99],
                  [0.01, 0.99],
                  [0.01, 0.99]])
    return X,Y

def get_3_class_data():
    X = np.array([[0.05, 0.1],
                  [0.07, 0.3],
                  [0.09, 0.5],
                  [0.05, 0.1]])

    Y = np.array([[1, 0, 0],
                  [0, 1, 0],
                  [0, 0, 1],
                  [1, 0, 0]])
    return X,Y

def get_sprial_class_data():
    np.random.seed(0)
    N = 100 # number of points per class
    D = 2 # dimensionality
    K = 3 # number of classes
    X = np.zeros((N*K,D))
    y = np.zeros(N*K, dtype='uint8')
    for j in xrange(K):
        ix = range(N*j,N*(j+1))
        r = np.linspace(0.0,1,N)  # radius
        t = np.linspace(j*4,(j+1)*4,N) + np.random.randn(N)*0.2 # theta
```

```
            X[ix] = np.c_[r*np.sin(t), r*np.cos(t)]
            y[ix] = j
        # fig = plt.figure()
        # plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap=plt.cm.Spectral)
        # plt.xlim([-1,1])
        # plt.ylim([-1,1])
        Y = (np.arange(np.max(y) + 1) == y[:, None]).astype(float)
        return X,Y

if __name__ == '__main__':
    main()
```