CSI 873 Fall 2017 Midterm

Bruce Goldfeder

November 13, 2017

1 Code Artificial Neural Net

I coded the Artificial Neural Net (ANN) in python following the algorithm for stochastic gradient descent in TABLE 4.2 from the text on page 98. I implemented the python using looping structures for the main parts of the algorithm and matrix mathematics using the python numpy library. My results were were consistent with the graphs shown in the text on page 110. I provided the following set of parameters that are adjustable at the command line when invoking the application.

```
**Network Parameters**
input: number of input units
hidden: number of hidden units
output: number of output units

**Training Parameters**
train: number of training images to use
valid: number of validation images to use
test: number of test images to use
epochs: number of full training set run throughs

**Hyperparameters**
lrnrate: this is the factor for learning rate
momentum: this is the momentum term added to weight update
stop: This is the stopping criteria a threshold of percent drop in validation
error per iteration
```

```
**Future implementation**
rate_decay: used for annealing the learning rate using a 1/t decay using
            the mathematical form alpha=alpha0/a0+(a0+a0*rate_decay)
```

I also included a parameter that I will be adding in at a future date called ratedecay. I am hoping that this will have a limiting effect on training set overfitting after a large number of weight updates. I researched various methodologies to improve performance such as this annealing of the learning rate using a 1/T or a step methodology. The ANN algorithm is based directly from the Tom Mitchell text. Additional research for mitigating overfitting is from the Stanford Unsupervised Feature Learning Deep Learning web site ($http://deeplearning.stanford.edu/wiki/index.php/UFLDL\_Tutorial$)

I also researched various algorithms for the sigmoid unit and settled on the sigmoid presented in the text.

$$f(z) = \frac{1}{1 + \exp(-z)}$$

The other function that I explored was the tanh() function. This is a little tricky but easily overcome in that it has a range of [-1,1] while the sigmoid range is [0,1]. The tanh function is shown below:

$$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

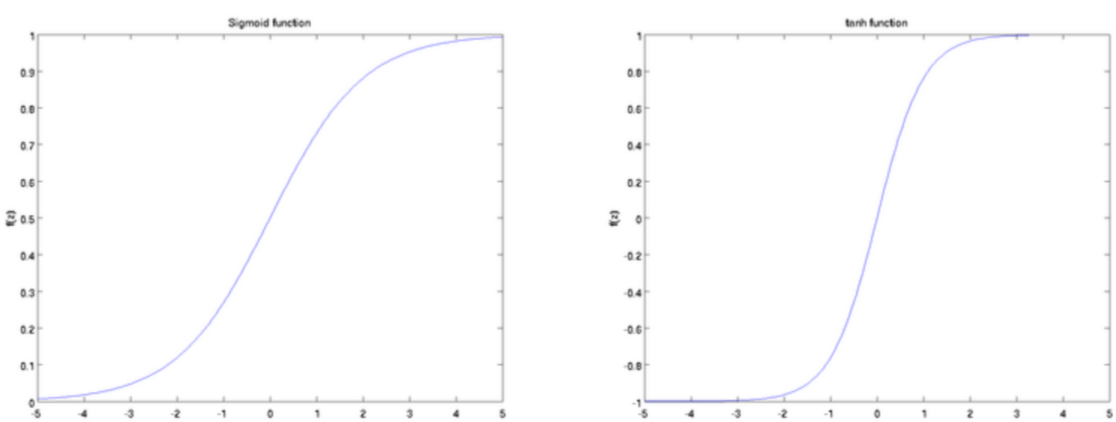


Figure 1: The Sigmoid and tanh functions

Both of these functions have easy to derive and easy to code derivatives:

Sigmoid Derivative

$$f'(z) = f(z)(1 - f(z))$$

tanh Derivative

$$f'(z) = 1 - (f(z))^2$$

For the midterm implementation, I leveraged the standard sigmoid function and derivative. I also coded a more robust version to make this function a bit more robust as I would have a tendency to run into overflow errors as the exponential term reached very large negative numbers. In later iterations of the code this was not needed as I had better control on the weights. The code that corrects these issues is shown below:

```
def sigmoid(z):
    try:
        if (z < 0):
            sig = 1.0 - 1.0/(1.0 + math.exp(z))
        else:
            sig = 1.0 / (1.0 + math.exp(-z))
    except OverflowError:
        if (z > 0):
            sig = 0.0000001
        else:
            sig = 1000000
    return sig
```

The code contained in the class NeuralNet contains four major functions matching the four steps defined in the text on Table 4-2 on page 98. The driver code reads in the data files, executes the methods first for training, then for validation, and finally for running the test set through for final results.

2 How to run the code

The python version is the current version of Anaconda which is Python 3.6.2. The code uses command line arguments that are optional and are passed into the program using the following flags:

An example to run trial number one is:

```
python csi873bgoldfeder.py -f C:/csi873/csi873midterm/data -e 30 -t 4000 -v 500
    -x 890
    -l 0.1 -m 0.1 -s 0.001
```

Table 1: Arguments for the program csi873bgoldfeder.py

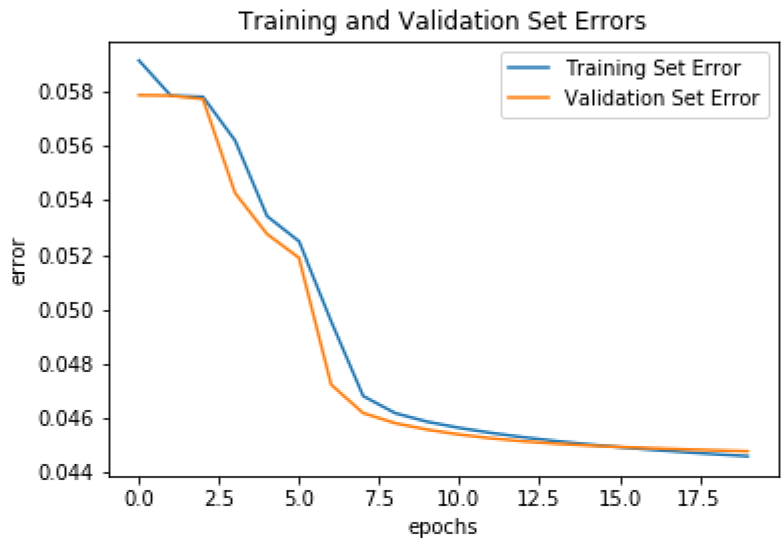
flag	**description**	**example**
-f	path to data	C:/csi873/data
-i	# hidden nodes	3
-e	# epochs to run	10
-t	# of training images per written number	100
-v	# validation images per written number	50
-x	# test images per written number	3
-l	learning rate	0.3
-m	momentum	0.3
-s	stop criteria	0.0005

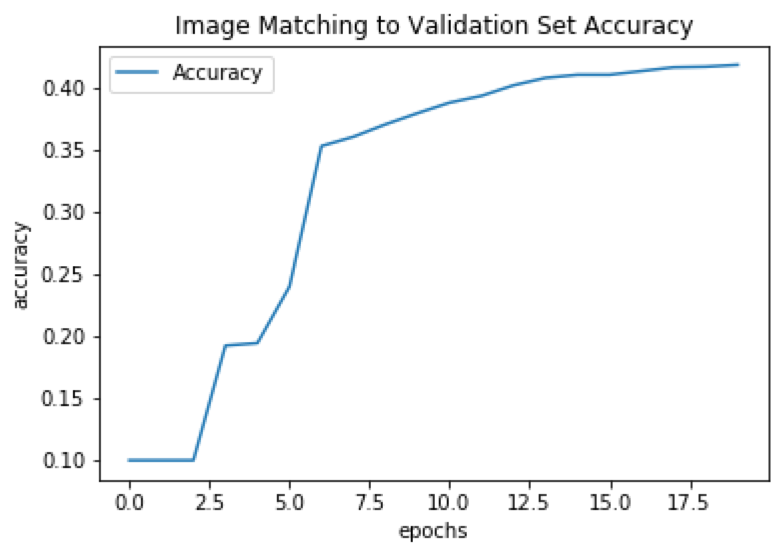
3 Successful Trial Run with limited data set

A successful trial run that indicated to me the ANN was running well is shown below using the following set of input parameters:

```
input      ==>        784
hidden     ==>          3
output     ==>         10
number of training images ==>       3000
number of validation images ==>       2000
number of test images ==>       2000
epochs     ==>         20
learn_rate ==>       0.10
momentum   ==>       0.10
```

Although the final accuracy rate was not the greatest, the behavior shown in both sets of output graphs validated that the ANN was iteratively reducing both the training set and the validation set sum of squared errors per epoch. The accuracy for this first trial came to 39.9% accuracy for the test set. The plots for the error and accuracy during training are shown below. Even with this small sample we are observation the positive reduction of training errors and iterative increase in accuracy of the validation set per epoch. I even noticed a slight start of overfitting in the error plot where the training error starting falling below the validation error.



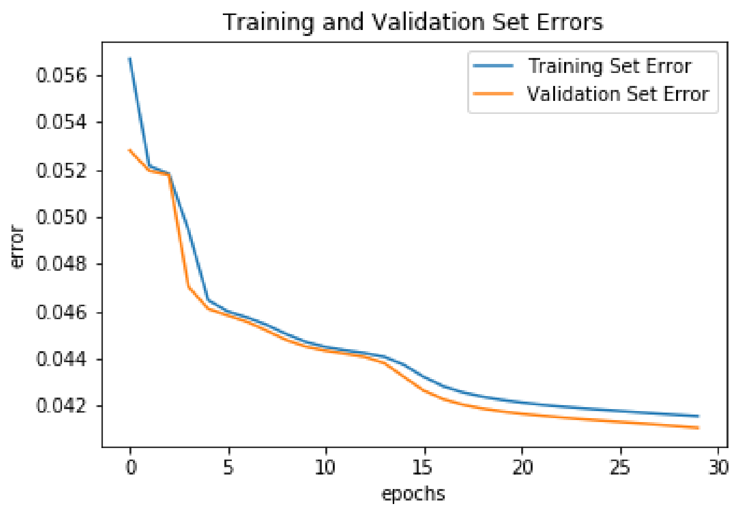


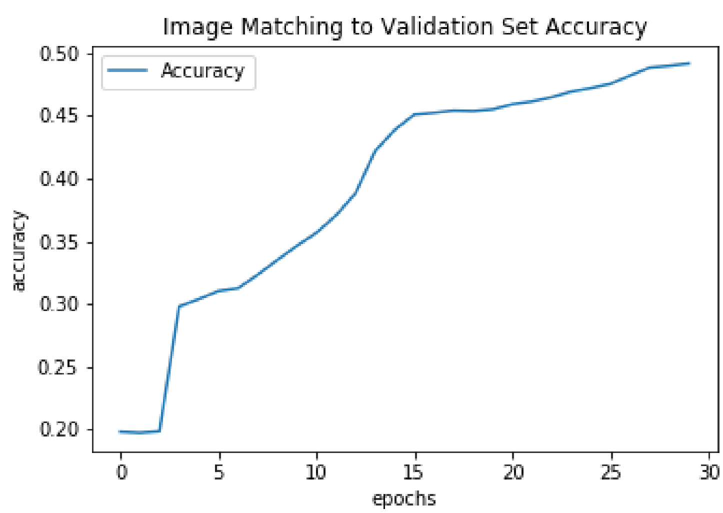
4 Trial One - 3 hidden nodes

To make this and later trials, for 2 and 3 nodes respectively, I increased the training sample size to 4000, the validation set size to 500, and the test set size to 890. I increased the number of epochs to 30. In addition, the initial runs went very quickly to overfitting, to counter this I lowered the learning rate to 0.03. This is due to the fact that the number of input images increased by a factor of 10, I inferred that the impact of these parameters would need to be mitigated proportionately. The output is as follows:

```
input      ==>        784
```

```
hidden      ==>          3
output      ==>         10
number of training images ==>      40000
number of validation images ==>       5000
number of test images ==>       8900
epochs      ==>         30
learn_rate ==>       0.03
momentum    ==>       0.10
Final Test results of  4301  out of  8900  accuracy is  0.4832584269662921
```





This provided an improvement over the smaller data set providing 48.3% accuracy over the test set. Lowering the learning rate had the positive effect of mitigating the overfitting, but I observe that I can raise it a little or I can add epochs of training.

4.a 95% Confidence Interval

The result set was 4301 correct responses out of 8900 images tested. The 95% confidence interval can be calculated using the formulas from chapter 5:

$$\sigma_{errors(h)} = \sqrt{\frac{p(1-p)}{n}}$$

4301 correct responses out of 8900 = .4833

$$\sigma_{errors(h)} = \sqrt{\frac{.4833(.5167)}{8900}} = .005297$$

Applying the formula

$$\mu \pm z_N \sigma; 1.96 \times .005297 = .01038$$

giving .4833 ± .01038; a 95% confidence interval of [.4729,.4937]

5 Trial Two - Two hidden nodes

```
csi873bgoldfeder.py -f C:/csi873/csi873midterm/data -i 2 -e 30 -t 4000 -v 500 -x 890
-l 0.05 -m 0.1
```

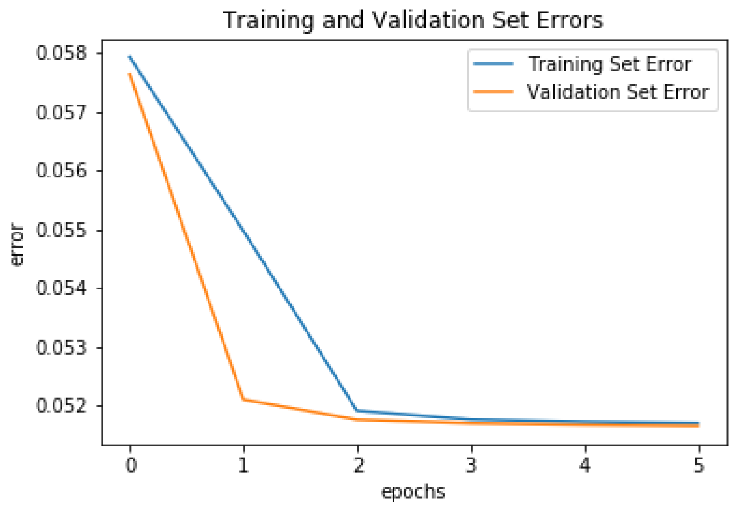
In this trial the number of hidden nodes was reduced from three to two hidden nodes. I also bumped up the learning rate from .03 to .05 to see if that accelerates the learning rate and I can observe the cross-over of the training and the validation errors indicating overfitting.
The parameters used are shown below:

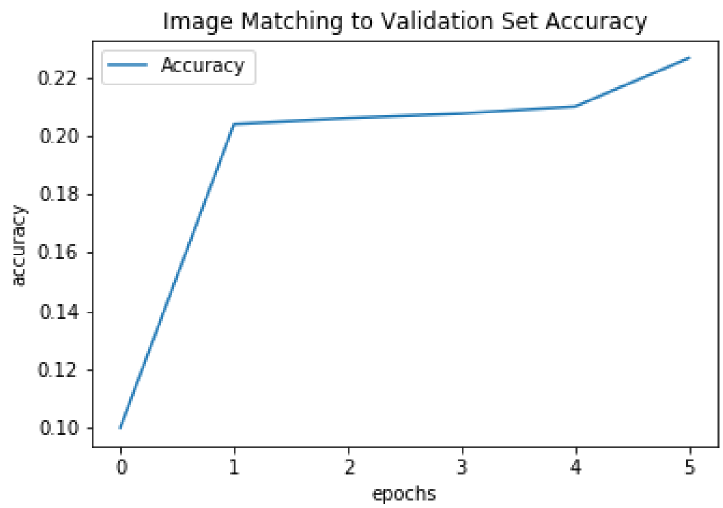
```
input      ==>        784
hidden     ==>          2
output     ==>         10
number of training images ==>      40000
number of validation images ==>      5000
number of test images ==>      8900
epochs     ==>         30
learn_rate ==>       0.05
```

`momentum    ==>        0.10`

`stopping criteria ==>    0.00050`

By limiting the number of hidden units down to two, the output seemed to become limited due to the lack of expression ability from just the two hidden units. This trial ended very early by hitting the stopping criteria on the 5th iteration. Understandably the accuracy of 22.49% was about half of the prior trial using three hidden units. The graphs are shown below:





5.a 95% Confidence Interval

The result set was 2002 correct responses out of 8900 images tested. The 95% confidence interval can be calculated using the formulas from chapter 5:

$$\sigma_{error_s(h)} = \sqrt{\frac{p(1-p)}{n}}$$

2002 correct responses out of 8900 = .2249

$$\sigma_{error_s(h)} = \sqrt{\frac{..2249(.7751)}{8900}} = .004426$$

Applying the formula

$$\mu \pm z_N\sigma; 1.96 \times .004426 = .008675$$

giving .2249 ± .008675; a 95% confidence interval of [.2163,.2336]

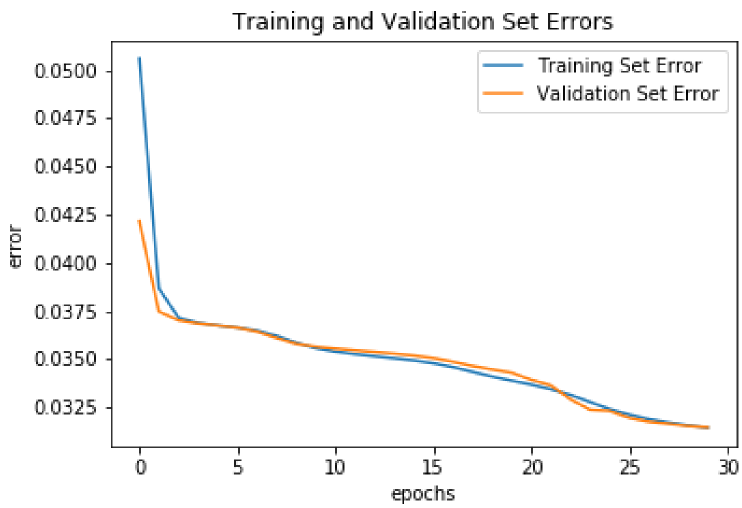
6 Trial 3: Four Hidden Units

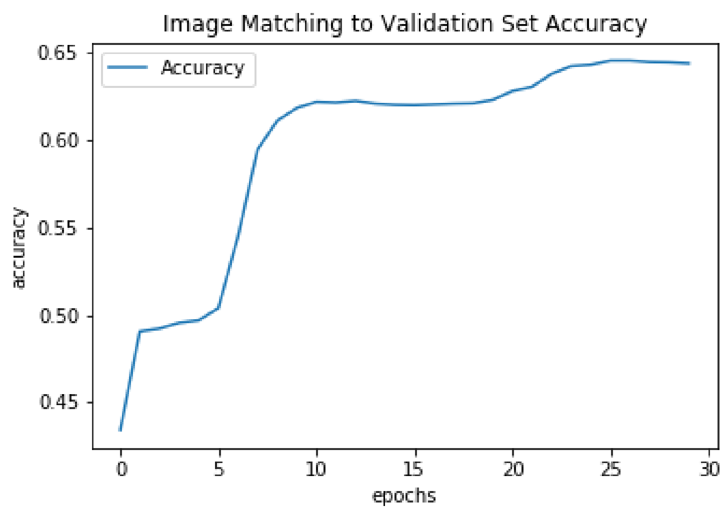
```
csi873bgoldfeder.py -f C:/csi873/csi873midterm/data -i 4 -e 30 -t 4000 -v 500 -x 890
-l 0.05 -m 0.1
```

For this last trial the number of hidden units is increased to four. I left all other parameters the same to include the stopping criteria of .05% drop in Validation Error rate (this is the default setting).
The following parameters were used:

```
input      ==>        784
hidden     ==>          4
output     ==>         10
number of training images ==>      40000
number of validation images ==>       5000
number of test images ==>      8900
epochs     ==>         30
learn_rate ==>       0.05
momentum   ==>       0.10
stopping criteria ==>    0.00050
```

The addition of two additional hidden nodes from the previous trial nearly tripled the test accuracy. An interesting observation is that the stopping criteria did not get triggered using the threshold percentage drop of .05% decrease in validation error. A higher learning rate could impact this or an increase in the number of epochs. From the graphs below it appears that additional accuracy could have been gained with either of these approaches. The momentum appears to have popped the validation set accuracy in two locations on the graph where major leaps in accuracy occurred leading to the final more gradual climb at the end. The graphs are shown below:





6.a 95% Confidence Interval

The result set was 5689 correct responses out of 8900 images tested. The 95% confidence interval can be calculated using the formulas from chapter 5:

$$\sigma_{error_s(h)} = \sqrt{\frac{p(1-p)}{n}}$$

5689 correct responses out of 8900 = .6392

$$\sigma_{error_s(h)} = \sqrt{\frac{.6392(.7751)}{8900}} = .005090$$

Applying the formula

$$\mu \pm z_N \sigma; 1.96 \times .005090 = .009976$$

giving .6392 ± .009976; a 95% confidence interval of [.6292,.6492]

```python
# -*- coding: utf-8 -*-
"""
Created on Wed Oct 18 18:29:16 2017

@author:   Bruce Goldfeder
           CSI 873
           Fall 2017
           Midterm
"""

import os,sys
import math
import numpy as np
import matplotlib.pyplot as plt

def sigmoid2(z):
    try:
        if (z < 0):
            sig = 1.0 - 1.0/(1.0 + math.exp(z))
        else:
            sig = 1.0 / (1.0 + math.exp(-z))
    except OverflowError:
        if (z > 0):
            sig = 0.01
        else:
            sig = 100
    return sig

def sigmoid(z):
    sig = 1.0 / (1.0 + math.exp(-z))
    return sig

def ReadInFiles(path,trnORtst):
    # This reads in all the files from a directory filtering on what the file
    # starts with
    fullData = []
    fnames = os.listdir(path)
    for fname in fnames:
        if fname.startswith(trnORtst):
            print (fname)
            data = np.loadtxt(path + "\\" + fname)
            fullData.append(data)
    #numFiles = len (fullData)
    #print(numFiles)

    return fullData

```

```
def ReadInOneList(fullData,maxRows):
    # This function combines all of the data into one array for ease of use
    # It contains a capping ability to configure how many results to use
    allData = []
    numFiles = len (fullData)
    for j in range (numFiles):
        # allows for smaller data set sizes
        numRows = len (fullData[j])
        #print('numrows,maxrows ',numRows,maxRows)
        if (maxRows < numRows):
            numRows = maxRows

        for k in range(numRows):
            allData.append(fullData[j][k])
    return np.asarray(allData)

def ReadInValidList(fullData,start,end):
    # This function combines all of the data into one array for ease of use
    # It contains a capping ability to configure how many results to use
    allData = []
    numFiles = len (fullData)
    for j in range (numFiles):
        # allows for smaller data set sizes
        numRows = len (fullData[j])
        if numRows < start + end :
            print('Starting at ' + str(start) + ' there are not ' + str(end) + \
                  ' images in the data set. Please retry')
            sys.exit()
        for k in range(start,end):
            allData.append(fullData[j][k])
    return np.asarray(allData)


def HeatMap(numberIn):
    #heat map to show numbers
    plt.matshow(numberIn.reshape(28,28))
    plt.colorbar()
    plt.show()

class NeuralNet(object):

    def __init__(self, input, hidden, output, inNum, valNum, tstNum, epochs, lrn_rate,
        momentum,stop):
        """
        **Network Parameters**
        input: number of input units
        hidden: number of hidden units
```

```
        output: number of output units

        **Training Parameters**
        inNum: number of images to run
        epochs: number of full training set run throughs

        **Hyperparameters**
        lrn_rate: this is the factor for learning rate
        momentum: this is the momentum term added to weight update
        rate_decay: used for annealing the learning rate using a 1/t decay using
                   the mathematical form alpha=alpha0/a0+(a0+a0*rate_decay)

        """
        # Training Parameters
        self.inNum = inNum
        self.valNum = valNum
        self.tstNum = tstNum
        self.epochs = epochs

        # Hyperparameters
        self.lrn_rate = lrn_rate
        self.momentum = momentum

        # initialize arrays
        self.input = input # add 1 for bias node
        self.hidden = hidden # This bias node add causes trouble
        self.output = output

        # set up arrays for the outputs of the nodes
        self.ai = np.ones(self.input) # removes the threshold input
        #print('ai shape ',self.ai.shape)
        self.ah = np.ones(self.hidden) # removes the threshold input
        self.ao = np.ones(self.output)

        # create random weights between -0.05 and 0.05 as per text on pg. 98
        # plus one for the bias/threshold unit
        self.wHidThresh = np.random.uniform(-0.05, 0.05, size = (self.hidden))
        self.wOutThresh = np.random.uniform(-0.05, 0.05, size = (self.output))

        self.wi = np.random.uniform(-0.05, 0.05, size = (self.input, self.hidden))
        self.wo = np.random.normal(-0.05, 0.05, size = (self.hidden, self.output))

        # temporary arrays to hold the numbers to be updated each iteration
        self.ci = np.zeros((self.input, self.hidden))
        self.co = np.zeros((self.hidden, self.output))

        # Error array to capture the output array for each training output unit
```

```
        self.outUnitErr = np.zeros((self.inNum,self.output))

        # Error array to capture the output array for each training output unit
        self.outValErr = np.zeros((self.valNum,self.output))

        # Error array to capture the output array for each training output unit
        self.outTstErr = np.zeros((self.tstNum,self.output))

        # String to append to files to identify experiment parameters
        self.expName = 'in' + str(self.input) + 'hi' + str(self.hidden) + \
                'out' + str(self.output) + 'lr' + str(self.lrn_rate) + \
                'mo' + str(self.momentum) + 'trN' + str(self.inNum) + \
                'vN' + str(self.valNum) + \
                'tsN' + str(self.tstNum) + 'ep' + str(self.epochs)

        # Set the stopping criteria which is the percent drop in Validation
        # from the current iteration to the previous
        self.stop = stop

        print("name " + self.expName)

    def print_params(self):

        print ('%-10s ==> %10d' % ('input', self.input))
        print ('%-10s ==> %10d' % ('hidden', self.hidden))
        print ('%-10s ==> %10d' % ('output', self.output))
        print ('%-10s ==> %10d' % ('number of training images', self.inNum))
        print ('%-10s ==> %10d' % ('number of validation images', self.valNum))
        print ('%-10s ==> %10d' % ('number of test images', self.tstNum))
        print ('%-10s ==> %10d' % ('epochs', self.epochs))
        print ('%-10s ==> %10.2f' % ('learn_rate', self.lrn_rate))
        print ('%-10s ==> %10.2f' % ('momentum', self.momentum))
        print ('%-10s ==> %10.5f' % ('stopping criteria', self.stop))

    def makeTargetArray(self,answer):
        tk = int(answer)
        #print('tk is ', tk)
        # Make the trained output 0.1 for 0 and 0.9 for 1 as per the text
        # from the first paragraph on the top of page 115
        tkArray = np.add(np.zeros(10),0.1)
        tkArray[tk]= 0.9
        #print("answer array is ",tkArray)

        return tkArray

    def feedForward(self,image,answer):

```

```
        #print (input,len(inputs))
        """
        The feedforward algorithm propogates the input forward through the network
        It calculates the net by summing from one to n the weights (including bias
        w0) times the value sigma term for each node in the hidden and output
        layers. This is then used as the negative exponent value in the sigmoid
        function
            net = Sigma[i=0,n]wi*xi
            ah = sigmoid(net) = 1/(1 + e**-net)
        """
        #self.ai[0] = 1.0 # This is the threshold for every hidden unit
        self.ai = image # remember to account for actual value in 0th index

        # hidden activations

        for j in range(self.hidden):
            sum = self.wHidThresh[j]
            for i in range(self.input):
                prodAW = self.ai[i] * self.wi[i][j]
                sum += prodAW
            #print ('sum is ',sum)
            self.ah[j] = sigmoid(sum) #what about using tanh here?

        # output activations
        for k in range(self.output):
            sum = self.wOutThresh[k]
            for j in range(self.hidden):
                prodAWO = self.ah[j] * self.wo[j][k]
                sum += prodAWO
            self.ao[k] = sigmoid(sum)

    def calcDeltaKO(self,answer):
        """
        Calculates the delta_k for each output unit using the formula from the text
        delta_k = ouput_k * (1-ouput_k)*(target_k - output_k)
        """
        tkArray= self.makeTargetArray(answer)

        delta_k = np.zeros(self.output)

        # Use the derivative of sigmoid times actual - observed
        for y in range(self.output):
            delta_k[y] = self.ao[y] * (1-self.ao[y]) * (tkArray[y] - self.ao[y])
            #print ('tkarray[y] ',tkArray[y],' ao[y] ',self.ao[y],' sub ',tkArray[y] - self.
                ao[y])


```

```
        return delta_k

    def calcDeltaKH(self,d_k):
        """
        Calculates the delta_h for each hidden unit using the forumula from the text
        delta_h = output_h * (1-ouput_h) * Sum(weight_kh * delta_ko)
        ah is the output of hidden units
        w_kh is the weight of hidden units or the array in main 'wi'
        d_k is the delta_k returned from the function 'calcDeltaKO' for output nodes
        """

        # Use derivative term for sigmoid applied to the hidden outputs
        delta_h1 = np.zeros(self.hidden) # first part of equation
        delta_h = np.zeros(self.hidden) # output of equation
        # corrected - switched rows and columns
        for hid in range(self.hidden):
            sum = 0.0  # question should this be the threshold value?
            delta_h1[hid] = self.ah[hid] * (1 - self.ah[hid])
            #print('y is ',y)
            # This calculates the second part of equation summing over this hidden
            # node the product of the weight_kh*delta_k of the output node
            for out in range(self.output):
                #print('x is ',x)

                # the product of the weight_kh*delta_k of the output node
                delta_h2 = self.wo[hid,out] * d_k[out]
                # Sum up all weight*dk for all output this hidden node touches
                sum += delta_h2
                #print(sum)

            # Calculate the derivative times the sum of the w_kh * delta_k
            dh = delta_h1[hid] * sum
            #print('delta h is ',dh)
            delta_h[hid] = dh

        #print('delta_h is: ',delta_h,' shape is ',delta_h.shape)
        return delta_h

    def updateWeights(self,answer,d_ko,d_kh):
        """
        The weights are updated using the forumula from the text
        w_ji <-- w_ji + Delta(w_ji)

        where
        Delta(w_ji) = n*delta_j*x_ji
        """
        # update the weights connecting hidden to output
```

```python
        # the co array represents the n-1 or prior iteration delta value
        for j in range(self.hidden):
            for k in range(self.output):
                delta = self.lrn_rate * d_ko[k] * self.ah[j] + (self.momentum * self.co[j][k])
                #print('delta is ',delta)
                self.wo[j][k] += delta
                self.co[j][k] = delta

        # update the weights connecting input to hidden
        for i in range(self.input):   # add in w0 threshold term
            for j in range(self.hidden): # add in w0 threshold term
                delta = self.lrn_rate * d_kh[j] * self.ai[i] + (self.momentum * self.ci[i][j])
                self.wi[i][j] += delta
                self.ci[i][j] = delta

    # to save time and complexity these are 3 different functions
    #TODO refactor to one function
    def calculateError(self,num,answer):

        tkArray = self.makeTargetArray(answer)
        for out in range(self.output):
            self.outUnitErr[num][out] = (tkArray[out] - self.ao[out])**2

    def calculateValErr(self,num,answer):

        tkArray = self.makeTargetArray(answer)
        for out in range(self.output):
            self.outValErr[num][out] = (tkArray[out] - self.ao[out])**2

    def calculateTstErr(self,num,answer):

        tkArray = self.makeTargetArray(answer)
        for out in range(self.output):
            self.outTstErr[num][out] = (tkArray[out] - self.ao[out])**2

    def plotErrorperNum(self):

        plt.imshow(self.outUnitErr[:,:])
        plt.xlabel('iterations')
        plt.ylabel('error')
        plt.title('Sum of squared errors for each output unite')
        plt.grid(True)
        plt.savefig("test.png")
        plt.show()

```

```
    def plotErrList(self,errTrn,errTst):
        plt.figure()
        plt.ylabel('error')
        plt.xlabel('epochs')
        ax = plt.subplot(111)
        ax.plot(errTrn,label='Training Set Error')
        ax.plot(errTst,label='Validation Set Error')

        plt.title('Training and Validation Set Errors')
        ax.legend()
        plt.savefig('pics/errPlots_' + self.expName + '.png')
        plt.show()

    def plotAccList(self,accList):
        plt.figure()
        plt.ylabel('accuracy')
        plt.xlabel('epochs')
        ax = plt.subplot(111)
        ax.plot(accList,label='Accuracy')

        plt.title('Image Matching to Validation Set Accuracy')
        ax.legend()
        plt.savefig('pics/accPlot_' + self.expName + '.png')
        plt.show()

def driver(dpath,inNodes,outNodes,hidNodes,epochs,trnNum,valNum,tstNum,lrnRate,momentum,stop
    ):

    # Read in the Training data first
    dataset = ReadInFiles(dpath,'train')
    my_data = ReadInOneList(dataset,trnNum)

    # Convert the 0-255 to 0 through 1 values in data
    my_data[:,1:] /= 255.0
    #HeatMap(my_data[40,1:])

    # randomize the rows for better training
    np.random.shuffle(my_data)
    inNum,cols = my_data.shape
    just_img_data = my_data[:,1:]
    answer = my_data[:,0]

    # Create the Validation data
    my_valid = ReadInValidList(dataset,tstNum,tstNum+valNum)
    my_valid[:,1:] /= 255.0
    valNum,valCols = my_valid.shape
    #print('val num is ',valNum)
```

```
    just_valid_data = my_valid[:,1:]
    answerValImg = my_valid[:,0]
    #print('array of answerws to follow')
    #print(answerValImg)

    # Read in the test data
    #dpath2 = os.getcwd()+'\data3'
    dataset2 = ReadInFiles(dpath,'test')
    my_test = ReadInOneList(dataset2,tstNum)

    tstNum,cols = my_test.shape
    #print('num rows ',tstNum)

    # Convert the 0-255 to 0 through 1 values in data
    my_test[:,1:] /= 255.0

    just_test_data = my_test[:,1:]
    answerImg = my_test[:,0]

    myNet = NeuralNet(inNodes, hidNodes, outNodes, inNum, valNum, tstNum, epochs,lrnRate,
       momentum,stop)
    myNet.print_params()

    trnErrorList = []
    trnValErrList = []
    tstErrList = []
    accList = []

    # Iterate over the number of epochs of data to run
    for eps in range(myNet.epochs):
        print("Training epoch ",eps)
        # Iterate over the range of total images for training
        for imgNum in range(inNum):
            myNet.feedForward(just_img_data[imgNum,:],answer[imgNum])

            # Calculate the error term deltaKO for each output unit
            deltaKO = myNet.calcDeltaKO(answer[imgNum])
            #print('deltaKO is: ',deltaKO,' shape is ',deltaKO.shape)

            # Calculate the error term deltaKH for each hidden unit
            deltaKH = myNet.calcDeltaKH(deltaKO)
            #print ('delta h is ',deltaKH)

            # Update the weights for output and hidden weight sets
            myNet.updateWeights(answer[imgNum],deltaKO,deltaKH)

            # Calculate the error per image per output unit
```

```
            myNet.calculateError(imgNum,answer[imgNum])

        # Output the training set error
        errPerEpoch = np.sum(myNet.outUnitErr,dtype='float')
        trnErrorList.append(errPerEpoch/(inNum*10.0))
        print("Total Training Error for epoch ",eps," is ",errPerEpoch/(inNum*10.0))

        accuracyList = []

        # Run this epochs trained model against the Validation Set of data
        for imgNum in range(valNum):

            myNet.feedForward(just_valid_data[imgNum,:],answerValImg[imgNum])

            valAnswer = myNet.ao.argmax(axis=0)
            #print('Val Answer is ',valAnswer, ' image answer is ',answerValImg[imgNum])
            if (valAnswer - answerValImg[imgNum] == 0):
                accuracyList.append(1)
            else:
                accuracyList.append(0)

            # Calculate the error for the validation images per output unit
            myNet.calculateValErr(imgNum,answerValImg[imgNum])

        # Output the Validation set error
        errValEpoch = np.sum(myNet.outValErr,dtype='float')
        trnValErrList.append(errValEpoch/(valNum*10.0)) # for the ten digits
        print("Total Validation Error for epoch ",eps," is ",errValEpoch/(valNum*10.0))

        # Output the Validation set accuracy
        right = sum(accuracyList)
        total = len(accuracyList)
        print('Results of ',right,' out of ',total,' accuracy is ',right/total)
        accList.append(right/total)

        # for every epoch iteration need to save off weights
        weights = [myNet.wHidThresh,myNet.wOutThresh,myNet.wi,myNet.wo]
        np.savez('output/weightEpoch_' + str(eps) + '_' + myNet.expName + '.npz', \
                                        wHidThresh=weights[0], \
                                        wOutThresh=weights[1], \
                                        wi=weights[2], \
                                        wo=weights[3])

        # Check for the stopping criteria on Validation Test Set
        criteriaMet = False
        if len(trnValErrList) > 1:
            currErr = trnValErrList[-1]
```

```python
            prevErr = trnValErrList[-2]
            diffErrRatio = (prevErr - currErr) / prevErr
            print("{0:.4f}%".format(100.0 * diffErrRatio))
            if (diffErrRatio < myNet.stop):
                print("Stopping criteria of ",str(stop)," is more than ",str(diffErrRatio))
                criteriaMet = True

        if criteriaMet:
            break

    # Need to run the Test set of data
    # First find the optimal set of weights from Validation
    # Find the epoch with the lowest validation set error and then
    # set the weights in the ANN to those weights for testing
    # This should only not be the last set if there are very large stopping
    # criteria which would make the error start to go back up due to
    # overtraining for example
    npValErr = np.asarray(trnValErrList)
    minEpoch = npValErr.argmin(axis=0)
    print("The epoch with lowest validation error is ",str(minEpoch))
    optWt = np.load('output/weightEpoch_' + str(minEpoch) + '_' + myNet.expName + '.npz')
    myNet.wHidThresh = optWt['wHidThresh']
    myNet.wOutThresh = optWt['wOutThresh']
    myNet.wi = optWt['wi']
    myNet.wo = optWt['wo']
    optWt.close()
    #print('Weight arrays wHidThresh', myNet.wHidThresh,'\n' \
    #    'wOutThresh ',myNet.wOutThresh, '\n' \
    #    'shape of wi ',myNet.wi.shape, '\n' \
    #    'shape of wo ',myNet.wo.shape)

    # Then run the test images through using the optimal weights
    tstAccList = []
    for imgNum in range(tstNum):

        myNet.feedForward(just_test_data[imgNum,:],answerImg[imgNum])

        tstAnswer = myNet.ao.argmax(axis=0)
        #print('Val Answer is ',valAnswer, ' image answer is ',answerValImg[imgNum])
        if (tstAnswer - answerImg[imgNum] == 0):
            tstAccList.append(1)
        else:
            tstAccList.append(0)

    # Output the Test set error
    errTstEpoch = np.sum(myNet.outTstErr,dtype='float')
    tstErrList.append(errTstEpoch/(tstNum*10.0)) # for the ten digits
```

```
    print("Final Testing Error is ",errTstEpoch/(tstNum*10.0))

    # Output the Validation set accuracy
    right = sum(tstAccList)
    total = len(tstAccList)
    testAccuracy = right/total
    print('Final Test results of ',right,' out of ',total,' accuracy is ',testAccuracy)


    # Plot output and save plot data to file
    myNet.plotErrList(trnErrorList,trnValErrList)
    myNet.plotAccList(accList)

    np.savez('output/plotData_' + myNet.expName + '.npz', \
                          trnErrorList=trnErrorList, \
                          trnValErrList=trnValErrList, \
                          accList=accList, \
                          testAccuracy=testAccuracy)

if __name__ == "__main__":

    # Parse commjand line options filename, epsilon, and maximum iterations
    from optparse import OptionParser

    parser = OptionParser()
    parser.add_option("-f", "--file", dest="filepath", help="Folder path for data")
    parser.add_option("-i", "--hid", dest="hidNodes", help="Number of Hidden Nodes")
    parser.add_option("-e", "--epochs", dest="epochs", help="Number of Epochs")
    parser.add_option("-t", "--train", dest="trnNum", help="Number of Training Images per Number")
    parser.add_option("-v", "--valid", dest="valNum", help="Number of Validation Images per Number")
    parser.add_option("-x", "--test", dest="tstNum", help="Number of Test Images per Number")
    parser.add_option("-l", "--learn", dest="lrnRate", help="Number of Test Images per Number")
    parser.add_option("-m", "--momentum", dest="momentum", help="Number of Test Images per Number")
    parser.add_option("-s", "--stop", dest="stop", help="Validation Stopping Criteria Percentage")


    options, args = parser.parse_args()

    if not options.filepath :
        print("Used default of data" )
        filepath = os.getcwd()+'\data'
```

```
    else: filepath = options.filepath

    if not options.hidNodes :
        print("Used default hidden nodes of 3" )
        hidNodes = 3
    else: hidNodes = int(options.hidNodes)

    if not options.epochs :
        print("Used default epochs = 30" )
        epochs = 30
    else: epochs = int(options.epochs)

    if not options.trnNum :
        print("Used default trnNum = 1000" )
        trnNum = 4500
    else: trnNum = int(options.trnNum)

    if not options.valNum :
        print("Used default valNum = 500" )
        valNum = 500
    else: valNum = int(options.valNum)

    if not options.tstNum :
        print("Used default tstNum = 500" )
        tstNum = 890
    else: tstNum = int(options.tstNum)

    if not options.lrnRate :
        print("Used default lrnRate = 0.5" )
        lrnRate = 0.5
    else: lrnRate = float(options.lrnRate)

    if not options.momentum :
        print("Used default momentum = 0.5" )
        momentum = 0.5
    else: momentum = float(options.momentum)

    if not options.stop :
        print("Used default stop = 0.05%" )
        stop = 0.0005
    else: stop = float(options.stop)

    inNodes = 784
    outNodes = 10

    driver(filepath,inNodes,outNodes,hidNodes,epochs,trnNum,valNum,tstNum,lrnRate,momentum,
        stop)
```