UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Department of Electrical, Electronic and Computer Engineering

EAI 320 - Intelligent systems

EAI 320 - Practical Assignment 6 Report

Author:
Mohamed Ameen Omar

Student number:
u16055323

April 26, 2018

Contents

List of Figures

1 Introduction

A Artificial Neural Network is a computer-based mathematical model of the structure and functioning of the human brain or more precisely of biological neural networks. Artificial Neural Networks are used as a supervised learning technique, wherein the Artificial Neural Network is given a set of inputs and a set of expected outcomes. Based off of this "Training" data, the model is able to form patterns and can therefore be used to predict the outcomes for unknown inputs.Artificial Neural Networks consists of an arbitrary set of layers, wherein each layer consists of Neurons (also known as units or nodes). Each Neuron in one layer is linked or connected to every other neuron in the following layer via a directed link [1]. Each directed link has numeric value known as it's weight associated with it. This weight determines the strength of the connection between the two neurons.

The first and last layers of a ANN are called the input layer and the output layer respectively. The input layer has at least as many neurons as the amount of inputs the ANN will receive and the output layer as the same amount of neurons as the number of outputs of the ANN.Between these two layers, we have what are called hidden layers. The amount of hidden layers in a Artificial Neural Network varies depending on the type of data the ANN will be used for. Although the lack of at least one hidden layer is acceptable for some basic problems, however, when dealing with more complex problems, the hidden layers are what give the ANN the power to learn and be successful at making predictions about unknown data.

An example of a ANN with a single hidden layer is shown in figure 1. The concept of an ANN is relatively simple to understand. The ANN is first "trained" with a set of inputs and outputs. Once a training stop-criterion such as amount of epochs(iterations through the training data set) or a specific error value is reached, training halts and the ANN will then be ready to accept unknown data inputs and output it's prediction.

There are many techniques through which a ANN is taught or trained, one of which that is extremely popular is known as Backpropagation, the details of which is discussed under Implementation and Methodology.

ANN's have a wide range of applications in AI including classification of images, recognizing handwritten numbers, self-driving cars and predicting the weather. However much research is still currently being done to optimize and improve ANN implementations.

2 Problem Definition

Students were tasked to implement the Backpropagation algorithm for a Artificial Neural Network with a single hidden layer. The ANN would take an arbitrary amount of inputs denoted by D, an arbitrary amount of hidden layer neurons denoted by H and an arbitrary amount of outputs denoted by C. The amount of inputs and amount of hidden neurons included the bias neuron for each of these layers. An example of the ANN is shown in figure 1.

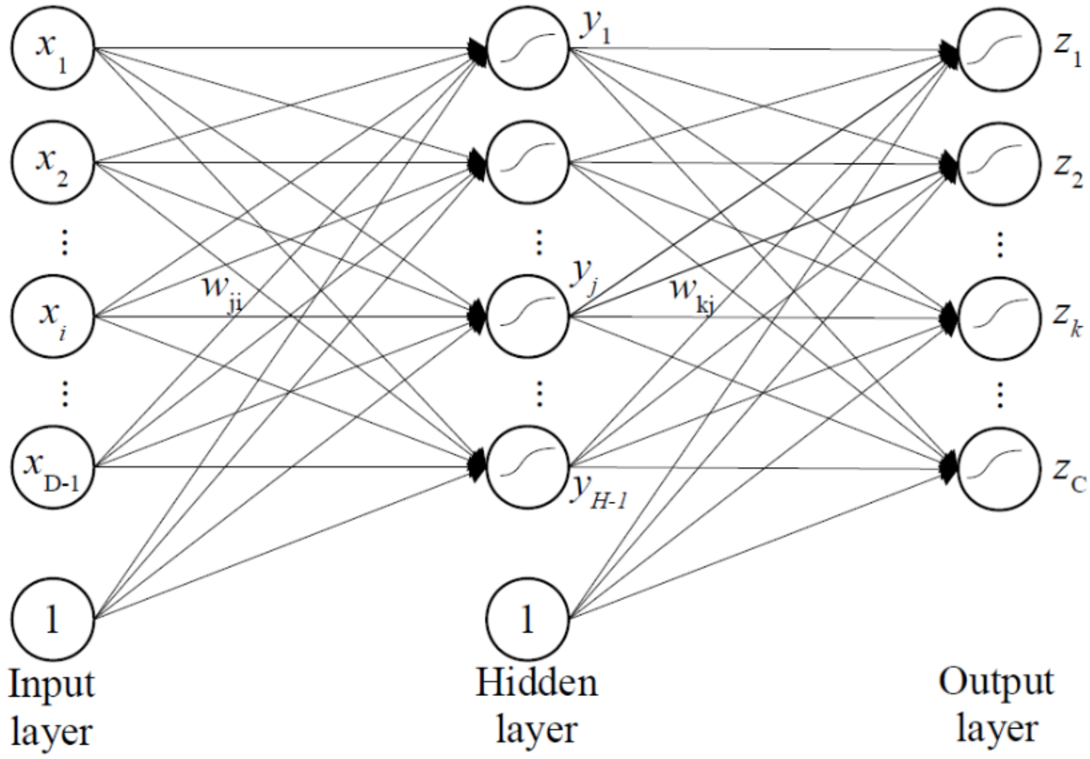


Figure 1: Example of the ANN provided in the practical specification

After implementation of their Backpropagation algorithm, students were further required to test their implementation on two problems given.
The first problem (Question 2) required students to demonstrate the ability of a ANN to model complex decision boundaries. Students applied their backpropagation algorithm implementation to model an arrow shaped decision boundary. An example of the expected output plot of the ANN is shown in figure 2. The training data was given to students in two *csv* files as the input data (named "q2inputs.csv") and the output or target data (named "q2targets.csv"). The ANN was to be trained with these files and thereafter a range of inputs between 0 and 1 were to be passed into the ANN, the output of which was plotted using the *matplotlib* Python library. The plotting code provided is shown in figure 3.

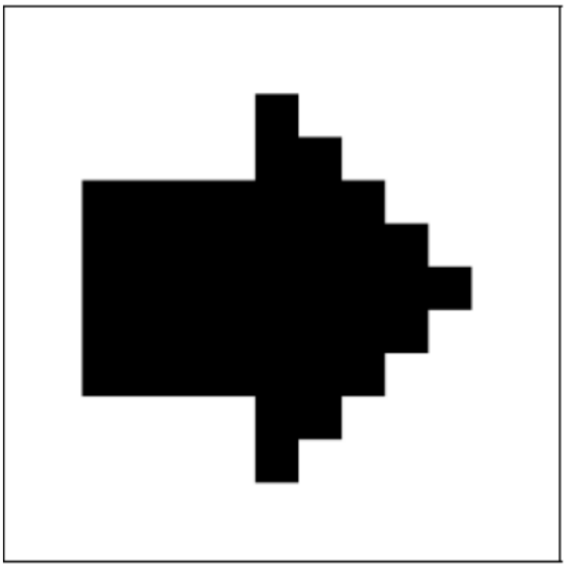


Figure 2: Example of the arrow plot output of the ANN for Question 2 in the practical specification

```
#Generate the surface
i = 0
j = 0
Z = np.zeros((21,21))
for x1 in np.arange(0,1,0.05):
    i = 0
    for x2 in np.arange(0,1,0.05):
        #Calculate the output of the neural network
        #given the input [1 x1 x2 ] and ANN weights, W1 and W2
        Z [i, j] = ffNeuralNet (np.array ([[1],[x1],[x2]] , W1, W2)
        i = i+1
    j = j+1
#Plot the surface
X = np.arange(0,21,1)
Y = np.arange(0,21,1)
X, Y = np.meshgrid(X, Y)
fig = plt.figure( )
ax = fig.gca(projection = '3d')
surf = ax.plotsurface(X,Y,Z, rstride = 1, cstride = 1, cmap=cm.
   coolwarm)
ax.setzlim(0,1)
```

Figure 3: Example of the ANN output plotting code for Question 2 provided in the practical specification

For the second problem (Question 3) students were required to apply the ANN to classify a wine data-set. The data-set consisted of 13 features. The ANN needed to be able to determine from which cultivar a wine sample originates based off of the training data provided. The inputs were the features and the outputs were the probability (between 0 and 1) of the culitvar from which the specific sample input originated. The training files provided were also in *csv* format.

3 Implementation and Methodology

The backpropogation algorithm is implemented as part of a Python class called *NeuralNetwork*. The class includes all helper functions needed to read the data as well as train the network. The program allows for user input so that the user can decide which question they would like to see the output of. Each Question is also implemented in separate helper functions to make the source code easier to read. The source code for the entire program can be found in Appendix A. D is the amount of inputs per sample into the ANN per sample, H is the amount of hidden neurons and C is the amount of output neurons per sample.

3.1 Neural Network Class

The Neural Network class implements a Artificial Neural Network. The class constructor takes in the Neural Network parameters such as *Number of epochs*, *Learning rate* and *Number of Hidden Neurons*. Based off of these parameters the Neural Network is built and trained.
The input neurons, output neurons and hidden neurons for each sample are stored in class member variables. The *fowardProp* and *backProp* member functions conduct the Forwardpropogation and back propagation respectively. Careful care has also been taken in order to ensure that the inputs are all normalized to the range of 1-0. The weights have also been initialized to a random value between $\frac{-1}{\sqrt{H}}$ and $\frac{1}{\sqrt{H}}$. This neural network also includes a bias neuron for both the input and output layers. The average squared error per epoch as well as the output error per sample ir recorded

3.2 Learning Algorithm

The learning algorithm employed in this ANN is the Backpropogation algorithm. The Backpropogation algorithm consists of two parts. Forward propagation and Backpropogation. The details for both are discussed below.

Forwardpropogation

During fowardpropogation, the inputs which is a Dx1 matrix is first transposed to make it a 1xD matrix. Then this input matrix is multiplied by the weights between the input layer and the hidden layer (stored in the *inputWeights* matrix). The result of this is the weighted sum for the hidden layer. This weighted sum is then passed into the activation function. In my implementation I have used the sigmoid function as the activation function. Thereafter the weighted sum for the output layer is computed by multiplying the hidden layer activation values with the weights between the hidden and output layers. Once again the activation function is used to compute the activation values for the output neurons. These activation values are the output of the ANN. This function is also used for getting the output of the ANN for other data.

Backpropogation

After fowardpropogation has executed the Backpropogation algorithm computes the cost function given in the practical specification. The algorithm makes use of the sigmoid-Prime member function which is the derivative of the sigmoid function. The weighted sums for the hidden and output layers that were stored by the Forwardpropogation algorithm are also used. Weight updates occur on a sample by sample basis, rather than once per epoch.

3.3 Training member function

The training member function is used to train the ANN. It extracts each sample passes it through *fowardProp* and *backProp* member functions for each epoch. At the end of an epoch the average squared error is computed and compared to the previous epoch, if the difference is within a certain range training stops. Training also terminates if the number of epochs declared to train has been reached.

3.4 User input

The user input prompts the user whether they would like to run Question 1 or Question 2. The user is also prompted for the ANN parameters they would like to use. Input validation has also been included.

4 Results

Figures 4 to 6 show the output for the user input functionality added to the program.

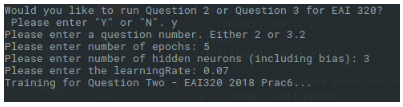


Figure 4: Output snippet showing an example of the user input

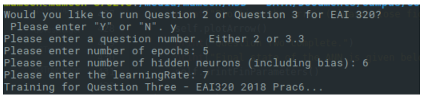


Figure 5: Output snippet showing an example of the user input

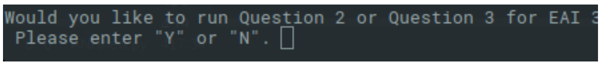


Figure 6: Output snippet showing an example of the user input

4.1 Question 2

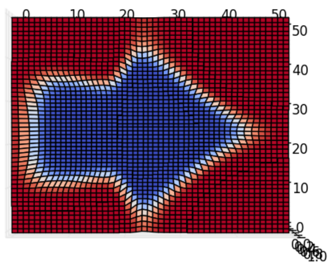


Figure 7: Plot of the ANN output for: 1000 epochs, 10 Hidden Neurons and a learning rate of 0.7

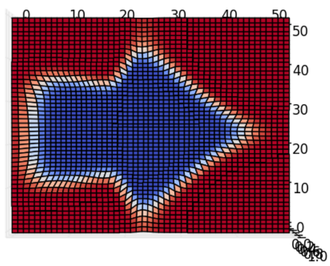


Figure 8: Plot of the ANN output for: 1000 epochs, 10 Hidden Neurons and a learning rate of 0.9

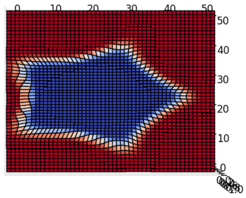


Figure 9: Plot of the ANN output for: 1000 epochs, 10 Hidden Neurons and a learning rate of 1

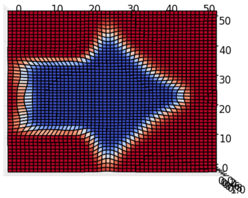


Figure 10: Plot of the ANN output for: 10000 epochs, 10 Hidden Neurons and a learning rate of 0.07

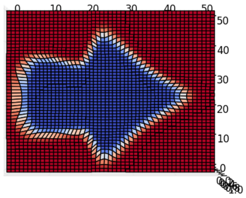


Figure 11: Plot of the ANN output for: 10000 epochs, 10 Hidden Neurons and a learning rate of 0.7

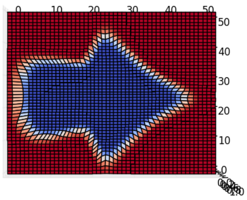


Figure 12: Plot of the ANN output for: 10000 epochs, 10 Hidden Neurons and a learning rate of 0.9

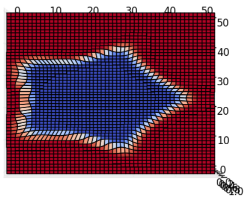


Figure 13: Plot of the ANN output for: 10000 epochs, 10 Hidden Neurons and a learning rate of 1

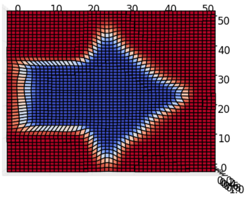


Figure 14: Plot of the ANN output for: 15000 epochs, 10 Hidden Neurons and a learning rate of 0.07

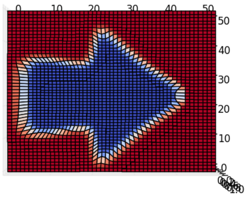


Figure 15: Plot of the ANN output for: 15000 epochs, 10 Hidden Neurons and a learning rate of 0.7

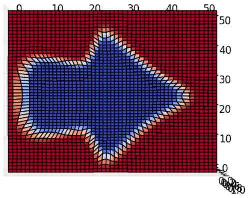


Figure 16: Plot of the ANN output for: 15000 epochs, 10 Hidden Neurons and a learning rate of 0.9

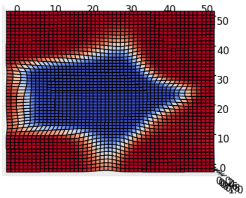


Figure 17: Plot of the ANN output for: 5000 epochs, 10 Hidden Neurons and a learning rate of 0.07

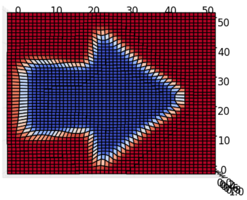


Figure 18: Plot of the ANN output for: 5000 epochs, 10 Hidden Neurons and a learning rate of 0.7

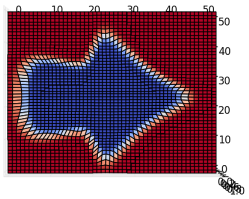


Figure 19: Plot of the ANN output for: 5000 epochs, 10 Hidden Neurons and a learning rate of 0.9

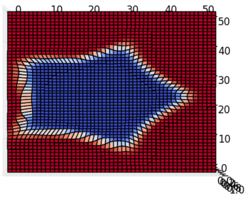


Figure 20: Plot of the ANN output for: 5000 epochs, 10 Hidden Neurons and a learning rate of 1

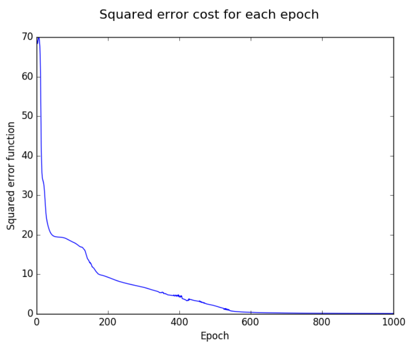
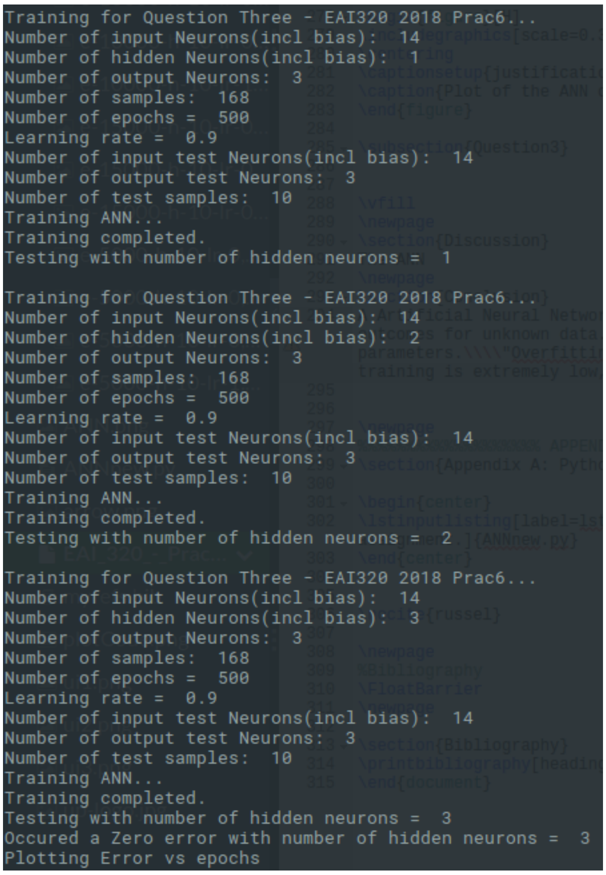


Figure 21: Plot of the Squared Error cost function for the data provided in Question 2 with 1000 epochs, 10 hidden neurons and a learning rate of 0.9

4.2 Question 3

The function trained the ANN with a variable amount of hidden neurons until there was a zero error. This condition was met when the amount of hidden neurons was equal to 3. The number of epochs and the learning rate was kept constant.



```
Training for Question Three - EAI320 2018 Prac6...
Number of input Neurons(incl bias):  14
Number of hidden Neurons(incl bias):  1
Number of output Neurons:  3
Number of samples:  168
Number of epochs =  500
Learning rate =  0.9
Number of input test Neurons(incl bias):  14
Number of output test Neurons:  3
Number of test samples:  10
Training ANN...
Training completed.
Testing with number of hidden neurons =  1

Training for Question Three - EAI320 2018 Prac6...
Number of input Neurons(incl bias):  14
Number of hidden Neurons(incl bias):  2
Number of output Neurons:  3
Number of samples:  168
Number of epochs =  500
Learning rate =  0.9
Number of input test Neurons(incl bias):  14
Number of output test Neurons:  3
Number of test samples:  10
Training ANN...
Training completed.
Testing with number of hidden neurons =  2

Training for Question Three - EAI320 2018 Prac6...
Number of input Neurons(incl bias):  14
Number of hidden Neurons(incl bias):  3
Number of output Neurons:  3
Number of samples:  168
Number of epochs =  500
Learning rate =  0.9
Number of input test Neurons(incl bias):  14
Number of output test Neurons:  3
Number of test samples:  10
Training ANN...
Training completed.
Testing with number of hidden neurons =  3
Occured a Zero error with number of hidden neurons =  3
Plotting Error vs epochs
```

Figure 22: Output snippet for the execution of Question 3

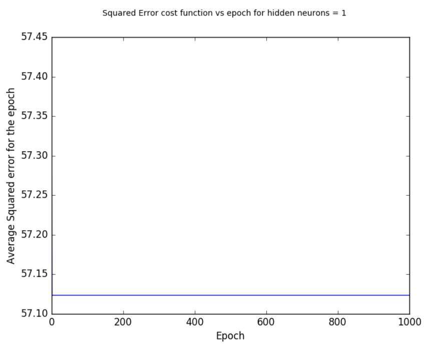


Figure 23: Plot of the Error vs Number of epochs for 1 hidden neuron and 1000 epochs

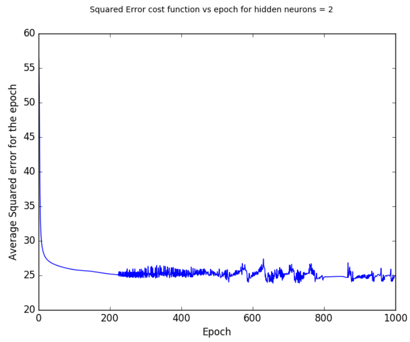


Figure 24: Plot of the Error vs Number of epochs for 2 hidden neurons and 1000 epochs

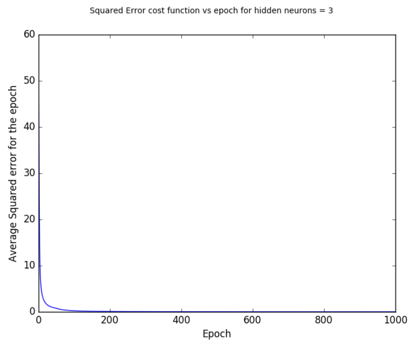


Figure 25: Plot of the Error vs Number of epochs for 3 hidden neurons and 1000 epochs

As we can see the error decays exponentially as the number of epochs increase. This error eventually converges to zero.

5 Discussion

"Overfitting" is a phenomenon wherein the ANN is over trained such that the error occurred for training is extremely low, but when unknown data is presented to the ANN, the error becomes very large. Factors that influence overfitting include the number of epochs (iterations through the training data) and the learning rate. If an extremely high learning rate is used, the ANN will quickly converge to an almost zero error, however if it is not stopped and continues to train, we will observe that the error will become extremely large when new unknown data is passed in. If a large number of epochs are run to train the ANN we will also overfitting taking place. A balance needs to be found between these parameters to avoid overfitting.

A huge problem faced when training a ANN with a large number of hidden neurons or a large number of epochs is the time needed for the ANN to train. Since complex matrix algebra, multiple function calls and huge amount of inputs, outputs and samples needs to be computed and processed, training a ANN does push the limits of the CPU. A lot of patience and time is needed to find the perfect balance between the time taken to train and the accuracy of the ANN.

Hidden Neurons play a very important role in the amount of epochs needed in order for ANN to converge to a zero or near zero error. From figures 7 to 10 we can see with 10 hidden neurons and about 1000 epochs the ANN has already been trained well enough to plot the arrow. From figure 21 we can see that the error with 10 hidden neurons quickly converges to zero and after about 700 epochs, we are over training. Therefore as we can see the amount of hidden neurons does play a huge role in the training of a ANN. A stop criterion can be implemented such that when the average error between two successive epochs is less than a threshold

6 Conclusion

A Artificial Neural Network is extremely powerful. It can learn complex patterns and can be used to predict the outcomes for unknown data. Careful care needs to be taken however when choosing the ANN parameters in order to find avoid overfitting and find a balance between accuracy and time taken to train. I was able to implement the algorithm and solve the problems posed during this practical assignment.

7 Appendix A: Python Code

```
import numpy
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
import matplotlib.pyplot as plt
numpy.random.seed(15) #for consistency?

##if the userInput paramter is False, the program willl build a NN with the data passed into the constructor
class NeuralNetwork:
    def __init__(self, userInput = True, learningRate = 0.9, epochs = 1000, numHidden = 10, inputFile = "q3TrainInputs.csv", outputFile = "q3TrainTargets.csv",testInputFile = "q3TestInputs.csv", testOutputFile = "q3TestTargets.csv"):
            self.numInputNeurons = 0 #number of input Neurons including bias per sample - extracted from inputFile
            self.numHiddenNeurons = numHidden #number of hidden Neurons including bias per sample -- must be sepcified
            self.numOutputNeurons = 0 #number of input Neurons per sample -- extracted from output file
            self.inputNeurons = None #matrix of all the input Neurons incluing bias for all samples DxN
            self.outputNeurons = None #matrix of all the outputNeurons for all samples CxN for every sample
            self.hiddenNeurons = None #matrix of all the hidden Neurons including bias HxN for every sample - changes after every epoch
            self.estimatedOutput = None #EstimatedOutut values for the given samples CxN for every sample -changes after every epoch
            self.numSamples = 0#number of samples in the training data -- extracted from input and output files
            self.epochs = epochs #number of epochs for training -- specified
            self.Z2 = None #h-1 x N ---weighted sums from input to hidden for each hidden(excl hidden bias)
            self.Z3 = None #CxN --weighted sums from hidden to output
            self.error = None #CxN ---Error for every output for every sample - target-estimated (changes after epoch)
            self.squaredErrorSample = None #1xN -- the squared error for every output per sample (changes after epoch)
            self.avSquaredError = None #epoch x 1 --- it is the average squared error per epoch (sum of all of the sample squared erros divide numSamples)
            self.learningRate = learningRate#learning rate -- must be specified
            self.inputWeights = None#DxH-1 --random with a range -1/sqrt(numHiddenNeurons) to 1/sqrt(numHiddenNeurons)
            self.hiddenWeights = None #HxC -- 1/sqrt(numHiddenNeurons) to 1/sqrt(numHiddenNeurons)
            self.inputFile = inputFile #csv for input training data --- must be specified -------- myTestIn.csv , q2inputs.csv
            self.outputFile = outputFile #csv for output training data --- must be sepcified ----------- myTestOut.csv, q2targets.csv
            self.userInput = userInput#whether to take user input or use specified constructor paramters
            self.epochStop = 0
            ########testing
```

```python
        self.testInputFile = testInputFile #csv for testing -- inputs
        self.testOutputFile = testOutputFile #csv for testing -- outputs
        self.inputTestNeurons = None #whenTesting
        self.outputTestNeurons = None
        self.TestsquaredErrorSample = None #1xN -- the squared error for every output per sample (changes after epoch)
        self.numInputTestNeurons = 0 #number of input Neurons including bias per sample - extracted from inputFile
        self.numOutputTestNeurons = 0 #number of input Neurons per sample -- extracted from output file
        self.numTestSamples = 0 #number of samples in test data
        self.estimatedTestOutput = None
        self.initNN()

        #q3TestInputs.csv  q3TestTargets.csv
    def userInputFunc(self):
        if(self.userInput is False):
            return
        question = input("Would you like to run Question 2 or Question 3 for EAI 320?\n Please enter \"Y\" or \"N\". ")
        while(question.upper() != "Y" and question.upper() != "N"):
            print("You entered \"",question,"\" which is invalid, please enter a valid character")
            question = input("Would you like to run Question 2 or Question 3 for EAI 320?\n Please enter \"Y\" or \"N\". ")
        if(question == "Y" or question == "y"):
            number  = input("Please enter a question number. Either 2 or 3.")
            while(number == "" or number == " "):
                print("An error occured.")
                number  = input("Please enter a question number. Either 2 or 3.")
            while(float(number) != 2 and float(number) != 3):
                print(float(number))
                print("You entered \"",number,"\" which is invalid, please enter a valid number")
                number  = input("Please enter a question number. Either 2 or 3.")
            ######get parameters
            epoch = input("Please enter number of epochs: ")
            while(epoch == "" or epoch == " "):
                print("An error occured.")
                epoch = input("Please enter number of epochs: ")
            self.epochs = int(epoch)
            hidden = input("Please enter number of hidden neurons (including bias): ")
            while(hidden == "" or hidden == " "):
                print("An error occured.")
                hidden = input("Please enter number of hidden neurons (including bias): ")
            self.numHiddenNeurons = int(hidden)
            lr = input("Please enter the learningRate: ")
            while(lr == "" or lr == " "):
                print("An error occured.")
                lr = input("Please enter the learningRate: ")
            self.learningRate = float(lr)
```

```
            if(float(number) == 2):
                self.questionTwo()
                return
            if(float(number) == 3):
                self.question3()
                return
        else:
            print("Functionality not built yet. Program ending")
            return
    #initizes the initial state of the Neural Network
    def initNN(self):
        print("This is a Neural Network implementation with a single hidden layer.")
        if(self.userInput is False):
            print("Constructing the Neural Network with the data pass into the constructor")
            self.initAll()
            self.printIniParameters()
            print("Training...")
            self.train()
            print("End Training")
            print("This is the state of the Neural Network")
            self.printFinParameters()
            print("Program will terminate now...")
            return
        else:
            self.userInputFunc()

    #initializes all member data for training
    #weights, inputs, outputs, etc
    def initAll(self):
        self.initInput()
        self.initHidden()
        self.initOutput()
        self.initWeights()
        self.initZ()
        self.initError()
        if(self.testInputFile != ""):
            self.initTestInput()
            self.initTestOutput()

    #trains the network with the given input file and output file training data
    #runs for the number of epochs(1 = all samples in training data set)
    def train(self):
        for epoch in range(0,self.epochs):
            for sample in range(0,self.numSamples):
                self.fowardProp(sample)
                self.backProp(sample)
            self.avSquaredError[epoch][0] = self.getAvSquaredError(epoch)
            if(epoch != 0):
                if( numpy.abs(self.avSquaredError[epoch][0] - self.avSquaredError[epoch-1][0]) <= 0.0000000002):
                    print("Stop Criterion reach for a squaredError <= 0.0000000002 on", epoch)
                    self.epochStop = epoch
```

```
                    return
                print("END OF EPOCH , ", epoch+1)
        print("The average squared error for all epochs are shown below")
        temp = "Please note however that cells with a value of exactly \"0\" are iterations skipped"
        temp = temp + "due to the error boundary being reached before total number of epochs being completed"
        print(temp)
        if(self.epochStop == 0):
            self.epochStop = self.epochs
        print(self.avSquaredError)

    #returns the average squared error for the current epoch
    def getAvSquaredError(self,epoch):
        temp = 0
        for x in range(0,len(self.squaredErrorSample[0])):
            temp = temp + self.squaredErrorSample[0][x]
        temp = (temp/self.numSamples)
        return temp

    #conducts a foward prop
    # if testing:
        #isTrain = False, input = Dx1 input data(if not normalized to (0,1) it will be)
        #will store all foward prop paramters in the sample = 0 index
        #will return a Cx1 array of outputs
        #input must include the bias value of 1 -- does no error checking for it
        #only a single
    # if Training:
    #   isTrain = True, inputs already normalized
    #   conducts a foward propgation with the sample(from 0) passed in as a parameter
    #   will compute the error and the squaredError for the sample
    #will tranpose all input data to make it a 1xD ie = [x,y,z] instead of [[x],[y],[z]]
    def fowardProp(self, sample, isTrain = True, input = None):
        #if it is a test
        if(isTrain is False):
            #sample = 0
            tempInput = input.transpose() #make a 1xD
            #if not normalized -> normalize:
            if(self.findMax(tempInput) > 1):
                tempInput = self.normalizeArray(tempInput)
            #tempInput = self.normalizeArray(input)
            #print("Conducting foward propogation for testCase")
        else:
            #print("Conducting foward propogation for sample at index", sample)
            tempInput = (self.inputNeurons[:,[sample]]).transpose() #make a 1xD
        weightedSum = numpy.matmul(tempInput,self.inputWeights)
        for x in range(0,self.numHiddenNeurons-1):
            self.Z2[x][sample] = weightedSum[0][x]
        for x in range(0, self.numHiddenNeurons-1):
            self.hiddenNeurons[x][sample] = self.sigmoidFunction(self.Z2[x][sample])
```

```
        tempHidden = (self.hiddenNeurons[:,[sample]]).transpose() #make 1xH
        weightedSum = numpy.matmul(tempHidden, self.hiddenWeights)
        for x in range(0,self.numOutputNeurons):
            self.Z3[x][sample] = weightedSum[0][x]
        #print("Z3")
        #print(self.Z3)
        for x in range(0, self.numOutputNeurons):
            self.estimatedOutput[x][sample] = self.sigmoidFunction(self.Z3[x][sample])
        #print("Estimated Output")
        #print(self.estimatedOutput)
        if(isTrain is False): #don't compute the error since it is not needed
            return (self.estimatedOutput[:,[sample]])
        #compute the error (target - output) for the current sample for each output
        for x in range(0, self.numOutputNeurons):
            self.error[x][sample] = self.outputNeurons[x][sample] - self.estimatedOutput[x][sample]
        #compute the squared error for the current sample (get square for each error for each output)
        # sum them and multiply by 0.5 - store for the current sample
        temp = 0
        for x in range(0,self.numOutputNeurons):
            temp += ( self.error[x][sample] *  self.error[x][sample] ) #take the square of each error for the current sample and sum
        temp = temp*0.5
        self.squaredErrorSample[0][sample] = temp #store squared error for the current sample
        return None
    #conducts a back Propogation to update the weights
    #called after foward prop for same @sample -> the sample index we backProping for
    #updates the weights
    def backProp(self,sample):
        #print("backProp")
       # Z3s = self.Z3[:,[sample]] #Cx1
        error = self.error[:,[sample]] #cx1
        Z3Prime = numpy.zeros(self.numOutputNeurons) #C
        for x in range(0,self.numOutputNeurons):
            Z3Prime[x] = self.sigmoidPrime(self.Z3[x][sample])
        Z3Prime = numpy.diag(Z3Prime) #CxC ---------Z3 prime matrix
        outputDelta = numpy.matmul(Z3Prime,error) #Cx1
        hiddenUpdate = numpy.matmul(self.hiddenNeurons[:,[sample]],outputDelta.transpose()) #HxC -- Actual update values
        hiddenUpdate = hiddenUpdate * self.learningRate
        ############ WE have hidden to output layer update #################
        #print("HIDDEN TO OUTPUT WEIGHT update")
        #print(hiddenUpdate)
        ####now compute input to hidden layer update ##########################
        tempHiddenWeight = self.hiddenWeights[:len(self.hiddenWeights)-1,:] #H-1 xC
        temp = numpy.matmul(tempHiddenWeight,outputDelta) # (h-1)xC times cx1 = (h-1)x1
```

```
        Z2Prime = numpy.zeros(self.numHiddenNeurons-1) # h-1 x 1
        #pass into sigmoid prime
        for x in range(0,self.numHiddenNeurons-1):
            Z2Prime[x] = self.sigmoidPrime(self.Z2[x][sample])
        Z2Prime = numpy.diag(Z2Prime)#make it a diagonal
        inputDelta = numpy.matmul(Z2Prime, temp) #h-1 x 1
    #       print(inputDelta.shape)
        inputUpdate = numpy.matmul(self.inputNeurons[:,[sample]], numpy.transpose(inputDelta)) #Dxh-1
        inputUpdate = inputUpdate * self.learningRate
        #print("Input weight update")
        #print(inputUpdate)
        ##update the weights
        self.hiddenWeights = self.hiddenWeights + hiddenUpdate
        self.inputWeights = self.inputWeights + inputUpdate
        #print("UPDATED hidden WEIGHT FOR PENIS, " ,sample)
        #print(self.hiddenWeights)
    #    print("UPDATED INPUT WEIGHT FOR PENIS, " ,sample)
    #    print(self.inputWeights )


    #imports the data given in a csv into a numpy array
    #the rows represent the inputs while the columns represent the sample
    #returns a reference to the data matrix
    def readData(self,filename):
        return numpy.genfromtxt(filename,delimiter=',')

    #function that initializes the input neurons array
    #reads in the csv into a numpy two dimensional array
    #normalizes the data
    #adds the bias nodes with a value of 1 - as the last input for each sample
    #rows represent inputs for ALL samples
    #columns represent the samples
    def initInput(self):
        self.inputNeurons = self.readData(self.inputFile) #reads data into a array
        self.inputNeurons = self.normalizeArray(self.inputNeurons)
        #it is two dimensional
        if(len(self.inputNeurons.shape)>1):
            bias = numpy.ones(len(self.inputNeurons[0]))
        else:
            if(self.inputNeurons.shape == ()):
                 bias = numpy.ones(1)
            else:
                bias = numpy.ones(len(self.inputNeurons))
        self.inputNeurons = numpy.vstack([self.inputNeurons,bias])
        if(len(self.inputNeurons.shape)>1):
            self.numInputNeurons = (self.inputNeurons.shape)[0]
            self.numSamples = len(self.inputNeurons[0])


    #initializes the Output neurons array
    def initOutput(self):
        self.outputNeurons = self.readData(self.outputFile)
        #then the output is not single dimensonal - we have more than one output
```

```
        if(len(self.outputNeurons.shape)>1):
            self.numOutputNeurons = (self.outputNeurons.shape)[0]
        else:
            self.numOutputNeurons = 1
            self.outputNeurons = self.outputNeurons.reshape(self.numOutputNeurons,self.numSamples)
        self.estimatedOutput = numpy.zeros((self.numOutputNeurons,self.numSamples))

    #initiliaze Z's
    def initZ(self):
        self.Z2 = numpy.zeros((self.numHiddenNeurons-1,self.numSamples))
        self.Z3 = numpy.zeros((self.numOutputNeurons, self.numSamples))

    #initialize the oZ2riginal weights of the neural network
    def initWeights(self):
        root = numpy.sqrt(self.numHiddenNeurons)
        self.inputWeights = numpy.random.uniform(low=(-1/root), high=(1/root), size=(self.numInputNeurons,self.numHiddenNeurons-1))
        self.hiddenWeights = numpy.random.uniform(low=(-1/root), high=(1/root), size=(self.numHiddenNeurons,self.numOutputNeurons))
        return

    #init Hidden neuron array
    def initHidden(self):
        self.hiddenNeurons = numpy.zeros((self.numHiddenNeurons,self.numSamples))
        for x in range(0,self.numSamples):
            self.hiddenNeurons[self.numHiddenNeurons-1][x] = 1

    def initError(self):
        self.error = numpy.zeros((self.numOutputNeurons,self.numSamples))
        self.squaredErrorSample = numpy.zeros((1,self.numSamples))
        self.avSquaredError = numpy.zeros((self.epochs, 1))
        if(self.testInputFile != ""):
            self.TestsquaredErrorSample =  numpy.zeros((1,self.numTestSamples))

    #normalizes all values in the array passed in
    def normalizeArray(self,temp):
        maximum = self.findMax(temp)
        minimum = self.findMin(temp)
        if(maximum == minimum):
            minimum = 0
        #if input data is not normalized, normalize
        if( (maximum <=1) and (minimum >=0)):
            return temp
        else:
            if(len(temp.shape) == 1):
                for x in range(0,len(temp)):
                    temp[x] = self.normalize(minimum,maximum,temp[x])
            else:
                for x in range(0,len(temp)):
                    for y in range(0,len(temp[0])):
                        temp[x][y] = self.normalize(minimum,maximum,temp[x][y])
            return temp
```

```
    #returns the maximum value in a 2D numpy array
    def findMax(self,array):
        return numpy.max(array)
    #returns the minimum values in a 2D numpy array
    def findMin(self, array):
        return numpy.min(array)
    #function to normalize our dataoutpuavSquaredErrort
    #takes in the minimum and maximum values of the data set
    #takes in the val to normalize
    #normalizes to a range between 0 and 1
    #returns the normalized value
    def normalize(self,minVal,maxVal,val):
        temp = (val-minVal)*(1-0)
        temp = temp/(maxVal-minVal)
        return temp
    #sigmoid activation function
    def sigmoidFunction(self,val):
        return ( 1 / ( 1+numpy.exp(-1*val) ) )
    #https://math.stackexchange.com/questions/78575/derivative-of-sigmoid-function-sigma-x-frac11e-x
    #http://www.ai.mit.edu/courses/6.892/lecture8-html/sld015.htm
    def sigmoidPrime(self,val):
        temp = self.sigmoidFunction(val)
        return(temp*(1-temp))
    #tanh activation function
    def tanhFunction(self,val):
        return numpy.tanh(val)
    def printInputNeurons(self):
        print(self.inputNeurons)
    def printHiddenNeurons(self):
        print(self.hiddenNeurons)
    def printOutputNeurons(self):
        print(self.outputNeurons)
    #for Question two
    #increased the resolution from 20 to 50 -- saves the ouput plot to a file
    def plotArrow(self):
        res = 50
        # Generate the surface
        i = 0
        j = 0
        Z = numpy.zeros((res+1,res+1))
        for x1 in numpy.arange (0,1,1/res):
            i = 0
            for x2 in numpy.arange (0,1,1/res):
                # Calculate the output of the neural network
                # given the input [1 x1 x2 ] and ANN weights , W1 and W2
                #forward prop returns the output of foward prop
                Z[i,j] = self.fowardProp(0,False,numpy.array([[x1],[x2],[1]]))
                i = i +1
            j = j +1
        #print(Z)
        # Plot the surface
        X = numpy.arange(0,res+1,1)
        Y = numpy.arange(0,res+1,1)
```

```
        X  , Y = numpy.meshgrid(X,Y)
        fig = plt.figure()
        ax = fig.gca(projection = '3d')
        surf = ax.plot_surface(X,Y,Z ,rstride = 1,cstride = 1 , cmap = plt
.cm.coolwarm)
        ax.set_zlim(0,1)
        name1 = 'e-' + str(self.epochs) + '-h-' + str(self.
numHiddenNeurons) + '-lr-' + str(self.learningRate)
        name = 'Q1PLOTS/' + name1 + '.png'
        ax.view_init(elev=90, azim=270)
        #plt.savefig(name) #uncomment to save
        #print("Plot saved as ", name)
        plt.show()   #uncomment to show the output

    #Does question 2
    def questionTwo(self):
        self.inputFile = "q2inputs.csv"
        self.outputFile = "q2targets.csv"
        self.testInputFile = ""
        self.testOutputFile = ""
        print("Training for Question Two - EAI320 2018 Prac6...")
        print()
        self.initAll()
        self.printIniParameters()
        print("Training ANN...")
        self.train()
        print("Training completed.")
        print()
        print("The average squared error for each epoch is given below:")
        for x in range(0,self.epochStop):
            temp = "Epoch" + str(x+1) + ":" + str(self.avSquaredError[x
][0])
            print(temp)
        print()
        print("Plotting the arrow (Please close figure window when done)")
        self.plotArrow()
        print("Question Two complete.")
        avError = self.avSquaredError.flatten() *self.numSamples
        print()
        print("Plotting Squared Error cost function per epoch")
        plt.xlabel("Epoch")
        plt.ylabel("Squared error function")
        plt.plot(avError)
        plt.suptitle('Squared error cost for each epoch', fontsize=16)
        plt.show()
        self.printFinParameters()

    #does question3
    #increament number of hidden neurons from 1 until a squared error
    def question3(self):
        count = 0 #amount of hidden neurons 0 = 1
        for hidden in range(1,20000):
            error = False
            self.numHiddenNeurons = hidden
            print("Training for Question Three - EAI320 2018 Prac6...")
            self.initAll() #intilializes all arrays
```

```
            print("Number of input Neurons(incl bias): ", self.
    numInputNeurons)
            print("Number of hidden Neurons(incl bias): ", self.
    numHiddenNeurons)
            print("Number of output Neurons: ", self.numOutputNeurons)
            print("Number of samples: ", self.numSamples)
            print("Number of epochs = ", self.epochs)
            print("Learning rate = ", self.learningRate)
            print("Number of input test Neurons(incl bias): ", self.
    numInputTestNeurons)
            print("Number of output test Neurons: ", self.
    numOutputTestNeurons)
            print("Number of test samples: ", self.numTestSamples)
            print("Training ANN...")
            self.train() #trains with data passed in
            print("Training completed.")
            print("Testing with number of hidden neurons = " , self.
    numHiddenNeurons)
            #fowardProp
            #foward propoagte sample by sample and return a cx1 array of
    outputs from the NN
            #store in testEstimated output
            for sample in range(0,self.numTestSamples):
                self.estimatedTestOutput[:,[sample]] = self.fowardProp(
    sample, False,self.inputTestNeurons[:,[sample]]) #False because not
    training
            #now have the estimate Output from the NN for every sample in
    the input test
            #compare by rounding
            for sample in range(0,self.numTestSamples):
                for out in range(0,self.numOutputTestNeurons):
                    if(not(self.outputTestNeurons[out][sample] == numpy.
    round(self.estimatedTestOutput[out][sample]))):
                        error = True
                        break
            if(error == False):
                break
            count = count+1
            print()
        print("Occured a Zero error with number of hidden neurons = ",
    self.numHiddenNeurons)
        print("Plotting Error vs epochs")
        for x in range(1,self.numHiddenNeurons+1):
            self.epochs = 1000
            self.numHiddenNeurons = x
            self.initAll() #intilializes all arrays
            self.train()
            avError = self.avSquaredError.flatten() *self.numSamples
            plt.xlabel("Epoch")
            plt.ylabel("Average Squared error for the epoch")
            plt.plot(avError)
            temp = "Squared Error cost function vs epoch for hidden
    neurons = " + str(self.numHiddenNeurons)
            plt.suptitle(temp, fontsize=10)
            plt.show()

    #prints the initial parameters of the NN
```

```
    def printIniParameters(self):
        print("Input Neurons (All training data - incl bias)")
        self.printInputNeurons()
        print("Hidden Neurons (incl bias) - initial ")
        self.printHiddenNeurons()
        print("Target Output Neurons (All training data)")
        self.printOutputNeurons() #targets
        print("Estimated output neurons - initial")
        print(self.estimatedOutput)
        print("First Layer Weights (Input to hidden) - initial")
        print(self.inputWeights)
        print("Second Layer Weights (Hidden to output) - initial")
        print(self.hiddenWeights)
        print("Errors (target - output) -- Initial")
        print(self.error)
        print("Squared errors per sample -- Initial")
        print(self.squaredErrorSample)
        print("Average Squared Error per Epoch -- Initial")
        print(self.avSquaredError)
        print("_______________________")
        print("Number of input Neurons(incl bias): ", self.numInputNeurons
)
        print("Number of hidden Neurons(incl bias): ", self.
numHiddenNeurons)
        print("Number of output Neurons: ", self.numOutputNeurons)
        print("Number of samples: ", self.numSamples)
        print("Number of epochs = ", self.epochs)
        print("Learning rate = ", self.learningRate)

    #prints the parameters of the NN after training
    def printFinParameters(self):
        print("Input Neurons (All training data - incl bias)")
        self.printInputNeurons()
        print("Hidden Neurons (incl bias) - after training(Last epoch)")
        self.printHiddenNeurons()
        print("Target Output Neurons (All training data)")
        self.printOutputNeurons() #targets
        print("Estimated output neurons - after training(Last epoch)")
        print(self.estimatedOutput)
        print("First Layer Weights (Input to hidden) -  After Training")
        print(self.inputWeights)
        print("Second Layer Weights (Hidden to output) - After Training")
        print(self.hiddenWeights)
        print("Errors (target - output) -- After Training(last epoch)")
        print(self.error)
        print("Squared errors per sample -- After Training (last epoch)")
        print(self.squaredErrorSample)
        print("Average Squared Error per Epoch -- After Training")
        print(self.avSquaredError)
        print("_______________________")
        print("Number of input Neurons(incl bias): ", self.numInputNeurons
)
        print("Number of hidden Neurons(incl bias): ", self.
numHiddenNeurons)
        print("Number of output Neurons: ", self.numOutputNeurons)
        print("Number of samples: ", self.numSamples)
        print("Number of epochs = ", self.epochs)
```

```
            print("Learning rate = ", self.learningRate)

    #initializes the input test array
    def initTestInput(self):
        self.inputTestNeurons = self.readData(self.testInputFile) #reads data into a array
        self.inputTestNeurons = self.normalizeArray(self.inputTestNeurons)
        #it is two dimensional
        if(len(self.inputTestNeurons.shape)>1):
            bias = numpy.ones(len(self.inputTestNeurons[0]))
        else:
            if(self.inputTestNeurons.shape == ()):
                 bias = numpy.ones(1)
            else:
                bias = numpy.ones(len(self.inputTestNeurons))
        self.inputTestNeurons = numpy.vstack([self.inputTestNeurons,bias])
        if(len(self.inputTestNeurons.shape)>1):
            self.numInputTestNeurons = (self.inputTestNeurons.shape)[0]
            self.numTestSamples = len(self.inputTestNeurons[0])

    #initializes the output test array
    def initTestOutput(self):
        self.outputTestNeurons = self.readData(self.testOutputFile)
        #then the output is not single dimensonal - we have more than one output
        if(len(self.outputTestNeurons.shape)>1):
            self.numOutputTestNeurons = (self.outputTestNeurons.shape)[0]
        else:
            self.numOutputTestNeurons = 1
            self.outputTestNeurons = self.outputTestNeurons.reshape(self.numOutputTestNeurons,self.numTestSamples)
        self.estimatedTestOutput = numpy.zeros((self.numOutputTestNeurons,self.numTestSamples))





test = NeuralNetwork(True)
```

Listing 1: Full program source code for this practical assignment.

8 Bibliography

[1] S. Russel and P. Norvig, *Artificial intelligence : A modern approach*, Third. Pearson, 2010.