**Checklist:**

Completed:

Pseudocode **X**

Well-documented **X**

Code works **X (For some scenarios there was no learning)**

Learning parameters: **X**

inputs, targets, nodeLayers, learning rate,

epochs, batchSize

Termination conditions: max epochs, correct classification

Mini-batch SGD **X**

Outputs as requested (iris, MNIST, and XOR) **X**

Incomplete – All the requirement are fulfilled. Accuracies for mnist data are not acceptable.

Not sure: Reason behind some of the inaccuracies in results are not know.

**Pseudocode:**

1. Initialize the weights and biases with standard random values with mean 0 and standard deviation 1
2. Number of column in bias for each of the layer should be equal to mini batch size.
3. For epoch starting from 1 to number of epochs:
   1. Generate an input matrix with batch size number of columns
   2. Generate a target matrix with batch size number of columns
   3. Using this input and target matrix calculate the change in weights and biases using backpropagation algorithm:
      1. Calculate z and activation vectors for each layer and save them in an array
      2. Calculate output error δ values using the last activation record and last z values using the formula

δx,l = ∇aCxΘ σ’(zx,l)

* + 1. Backpropagate the error using the formula

δx,l = ((wl+1)*T*δx,l+1 Θ σ’(zx,l)

* + 1. Calculate the new weight and biases for each layer

wl →wl – (η/m) δx,l(ax,l-1)*T*

bl →bl – (η/m) δx,l

* 1. Calculating Accuracies: For each input column:
     1. Calculate the activation value using the new weights and biases
     2. If the target value and output activation is same increase the count of accurate identified output
     3. Calculate Squared Error for each record. At the end of for loop, calculate MSE
  2. Print the results for each epoch

**Instructions:**

Code is written in octave 4.0.3 version. Load the all the files in the same folder including the data files.

In the submission folder there are three files:

1. xorScript.m – For xor.csv
2. irisScript.m - For iris.csv
3. mnistScript.m – For mnist data

Open any of these file in the octave editor and change the value of any of the below parameters:

1. layerVector – row vector showing number of neurons in each layer including input and output layer
2. newEpoch – number of epochs to run
3. batchSize – mini batch size
4. eta – value learning rate

After making changes save the file and open the octave command window and run the following command:

**>> source <scriptname>**

Script can also be executed from the editor window using the run button.

If you are running for a different dataset then call the following function in the command window:

**start(input,target,layerVector,numEpoch,batchSize,eta);**

where layerVector is row vector.

**Description:**

The code is well documented and contains all the information to understand the logic flow. There are two main files start.m and backpropagation.m which contains the core logic.

Start.m:

Line 24-33 : Weight and Bias Initializations

Line 65: Call to back propagation algorithm

Line 73-94: Accuracy calculations

Line 96: Plotting Results

backpropagation.m

Line 40-49 : Calculation of activation values

Line 60: Output error calculation

Line 65-75: Calculation of δ values at each layer

Line 78-94: Calculation and updating of weight and biases

There are other supporting files:

1. sigmoid.m => For calculating value of sigmoid function
2. sigmoid\_prime.m => For calculating derivative of sigmoid function
3. cost\_derivative => Calculating cost derivative
4. Globalvariables => Declares all the global variables used in the application

Other script files are used to run the application with specific data set:

1. xorScript.m
2. irisScript.m
3. mnistScrip.m

**Code:**

1. **xorScript.m – For xor.csv data file**

##

#author: Vaibhav Sharma

#Date: 10/08/2016

#Description: Script file for xor.csv file

##

clear;

clc;

load xor.csv;

xor = xor';

input = xor(1:2,:);

target = xor(3,:);

layerVector = [2,2,3,1];

numEpoch = 10;

batchSize = 1;

eta = 0.1;

start(input,target,layerVector,numEpoch,batchSize,eta);

1. **irisScrip.m – For iris.csv data file**

##

#author: Vaibhav Sharma

#Date: 10/08/2016

#Description: Script file for iris.csv file

##

clear;

clc;

load iris.csv;

iris = iris';

input = iris(1:4,:);

target = iris(5:7,:);

layerVector = [4,20,3];

numEpoch = 100;

batchSize = 10;

eta = 0.1;

start(input,target,layerVector,numEpoch,batchSize,eta);

1. **mnistScript.m – For mnist Data File**

##

#author: Vaibhav Sharma

#Date: 10/08/2016

#Description: Script file for mnistTrn.mat file

##

clear;

clc;

load mnistTrn.mat

layerVector = [784,40,30,10]

numEpoch = 1000

batchSize = 10

eta = 2.0

start(trn,trnAns,layerVector,numEpoch,batchSize,eta)

1. **Globalvariables.m – Lists all the globals declared in application**

#{

author: Vaibhav Sharma

Date: 10/08/2016

Description: This file contains all the global variables used in the applicaiton

#}

global weights # Stores weigths for the network

global biases # Stores biases for the network

global numberOfLayers #Stores number of layers in the network

global attained100Accuracy # Tells Application that 100% Accuracy is attained

1. **start.m – Contains the main function where processing starts**

##

#author: Vaibhav Sharma

#Date: 10/08/2016

#Description

#Arguments:

# input > a matrix with a column for each example, and a row for each input feature

# target > a matrix with a column for each example, and a row for each output feature

# nodeLayers > a vector with the number of nodes in each layer (including the input and output layers).

# numEpochs > (scalar) desired number of epochs to run

# batchSize > (scalar) number of instances in a mini-batch

# eta > (scalar) learning rate

#Output: Prints the output accuracy

##

function start(input,target,nodeLayers,numEpochs,batchSize,eta)

# Global varaibles that will be shared across multiple functions

global weights

global biases

global numberOfLayers

global attained100Accuracy

# Initializing Global Varaibles

numberOfLayers = columns(nodeLayers);

weights = cell(1,numberOfLayers);

biases = cell(1,numberOfLayers);

attained100Accuracy = 0;

inputRows = rows(input);

inputCols = columns(input);

targetRows = rows(target);

targetColumns = columns(target);

##

#Initialing weigths and biases with values with mean 0

#and standard devaiation 1

#Leaving the first index space blank. Starting from 2nd index

##

#stdnormal\_rnd

for i = 2: (numberOfLayers)

weights(i) = randn(nodeLayers(i),nodeLayers(i-1));

biases(i) = randn(nodeLayers(i),1);

endfor

for i = 2: (numberOfLayers)

b = biases{1,i};

b = repmat(b,1,batchSize);

biases(i) = b;

endfor

accuracies = ones(1,numEpochs);

mseV = zeros(1,numEpochs);

for i = 1:numEpochs

##

#Randomly shuffling to remove any bias in the input data

# Not done in this version

#inputTemp = input'

#inputTemp = inputTemp(randperm(inputCols),:)

#input = inputTemp'

##

for j = 1:batchSize:inputCols

a = input(:,[j:j+batchSize-1]);

t = target(:,[j:j+batchSize-1]);

backpropogation(a,t,batchSize,eta);

endfor

if (attained100Accuracy == 1)

break;

endif

#Calculating Accuracy

correctCount = 0;

mseSum = 0 ;

for k = 1:inputCols

a = input(:,[k:k]);

t = target(:,[k:k]);

for l = 2:numberOfLayers

w = weights{1,l};

b = biases{1,l};

z = z = w \* a + b;

a = sigmoid(z);

endfor

a = round(a(1));

res = isequal(a,t);

if res == 1

correctCount++;

endif

diff = t - a;

diff = diff .\* diff;

mseSum = mseSum + sum(diff);

endfor

MSE = mseSum /(2 \* inputCols);

acc = correctCount/inputCols;

printf("Epoch %d: MSE: %f Correct: %d \ %d Accuracy: %f \n",i,MSE,correctCount,inputCols,acc);

fflush(stdout);

accuracies(i) = correctCount/inputCols;

endfor

plot(i= 1:numEpochs,accuracies(i));

endfunction

1. **backpropogation.m – Implements Backpropogation Algorithm**

##

#author: Vaibhav Sharma

#Date: 10/08/2016

#Description: Implements Backpropogation Algorithm

#Arguments:

# a > input activation

# t > target values for activation

# batchsize > Size of mini batch

# eta > learning rate

#Output:

##

function backpropogation(a,t,batchsize,eta)

#Global Variables - Initialized in start.m

global weights

global biases

global numberOfLayers

global attained100Accuracy

# For storing activation record at each layer

activations = cell(1,numberOfLayers);

# For storing z values at each layer

# zVals(1) is left uninitialized

zVals = cell(1,numberOfLayers);

# For storing delta values at each layer

# deltaVals(1) is left uninitialized

deltaVals = cell(1,numberOfLayers);

#weightsPd = cell(1,numberOfLayers);

#biasesPd = cell(1,numberOfLayers);

activations(1) = a;

# Calculation z and activation values for each layer starting from 2

for i = 2:numberOfLayers

w = weights{1,i};

b = biases{1,i};

z = w \* a + b;

aTemp = sigmoid(z);

zVals(i) = z;

activations(i) = aTemp;

a = aTemp;

endfor

aLast = activations{1,numberOfLayers}; # Getting the final output

aLast = round(aLast); # Rounding the vector

zLast = zVals{1,numberOfLayers}; # Getting the last z value

diffLast = cost\_derivative(aLast,t);

#if(any(diffLast) == 0)

# attained100Accuracy = 1;

# return

#endif

deltaLast = diffLast .\* sigmoid\_prime(zLast);

deltaVals(numberOfLayers) = deltaLast;

# Caluclating delta values

# deltaVals(1) is left uninitialized

for i = numberOfLayers-1:-1:2

weightT = weights{1,i+1}';

deltaV = deltaVals{1,i+1};

zVal = zVals{1,i};

product = weightT \* deltaV;

sigmaPrime = sigmoid\_prime(zVal);

deltaVals(i) = product .\* sigmaPrime;

endfor

# Updating biases

for i = 2: numberOfLayers

bPrev = biases{1,i}; # Old Weight for the layer

change = (eta/batchsize) \* deltaVals{1,i};

biases(i) = bPrev - change;

endfor

# Updating weights

for i = 2: numberOfLayers

dVal = deltaVals{1,i}; # Delta value for the layer

aVal = activations{1,i-1}'; # Transpose of activation of the previous layer

product = dVal \* aVal; # Matrix Multiplication

change = (eta/batchsize) \* product; # Change for the layer

wPrev = weights{1,i}; # Old Weight for the layer

wNew = wPrev - change; # Calculating New Weight

weights(i) = wNew; # Updating new weights

endfor

endfunction

1. **cost\_derivative.m – For Calculating cost derivative**

##

#author: Vaibhav Sharma

#Date: 10/08/2016

#Description: Calculates partial derivatives

#Output: Returns the value of partial derivatives

##

function [pdRet] = cost\_derivative(out\_activations,y)

pdRet = out\_activations - y;

endfunction

1. **sigmoid.m – For calculating sigmoid function**

##

#author: Vaibhav Sharma

#Date: 10/08/2016

#Description: Calculates the sigma value of the incoming vector z

#Output: Returns the value of sigmoid neuron

##

function sigRet = sigmoid(z)

sigRet = 1.0 ./ (1.0 +exp(-z));

endfunction

1. **sigmoid\_prime.m – For calculating derivative of sigmoid function**

##

#author: Vaibhav Sharma

#Date: 10/08/2016

#Description: Calculates the derivative sigmoid function

#Output: Returns the value of derivative of sigmoid function

##

function [derVal] = sigmoid\_prime(z)

derVal = sigmoid(z) .\* (1 - sigmoid(z));

endfunction

**Analysis:**

Algorithm learned well for the xor data. Learning was slow for both iris and mnist data. For the testing done, there was almost zero recognitions. Though the weight and biases were changing with every mini batch but the network was not able to recognize the target data accurately.

Also, in some of the cases we can see overfitting in the output.

**Output:**

xor.csv

Run 1: epoch = 10, hidden nodes =2 , batchSize = 4, eta = 0.1



Run 2: epoch = 10, hidden nodes = 2, batchSize = 1, eta = 0.1



Run 3: epoch = 20, hidden nodes = [3 2], batchSize = 1, eta = 0.1



Run 4: epoch = 50, hidden nodes = 5, batchSize = 1, eta = 0.1



iris.csv

Run 1: epoch = 100, hidden nodes =20 , batchSize = 10, eta = 0.1



Mnist.mat

Run 1: epoch = 30, hidden nodes =30 , batchSize = 10, eta = 3.0

