Backpropagation

By: Group 13

CSS 485

Professor Michael Stiber, Ph D.

Team Member: Jun H Park, Thomas Hedrick

INTRODUCTION

The first part of this homework was to focus on creating a basic backpropagation learning algorithm that can be used in part two. The first part had us train a network to detect noisy numbers and try to pick the correct number out of 3 digits. The algorithms created in this part were used in part 2 on a much larger dataset. Part two involved using the MNIST dataset to train a network with backpropagation to detect what number was written when given a written number.

PART 1: SIMPLE CHARACTER RECOGNITION

Part 1 was to solve exercise 7.11 by using the Backpropagation method, which had been completed using Hebbian and Pseudo Inverse rules. The given input patterns were P_1 represents pixels represent 0 (In 30pixel). P_2 represents pixels represent 1 (In 30pixel). P_3 represents pixels represent 2 (In 30pixel). With the corresponding target output which represented with 1 and 0. At the end, group 13 added noise 0, 4, and 8 pixel on each digit and did that for 10 times just like exercise 7.11, then result out the accuracy. To accomplish the following problem, group 13 had processed the following method represented in Method Part 1 section.

METHOD PART 1

First of all, due to the given data, 30 inputs and 3 outputs. Group 13 was able to come up with 3 layer architecture presented in figure 1.0: ANN Architecture.

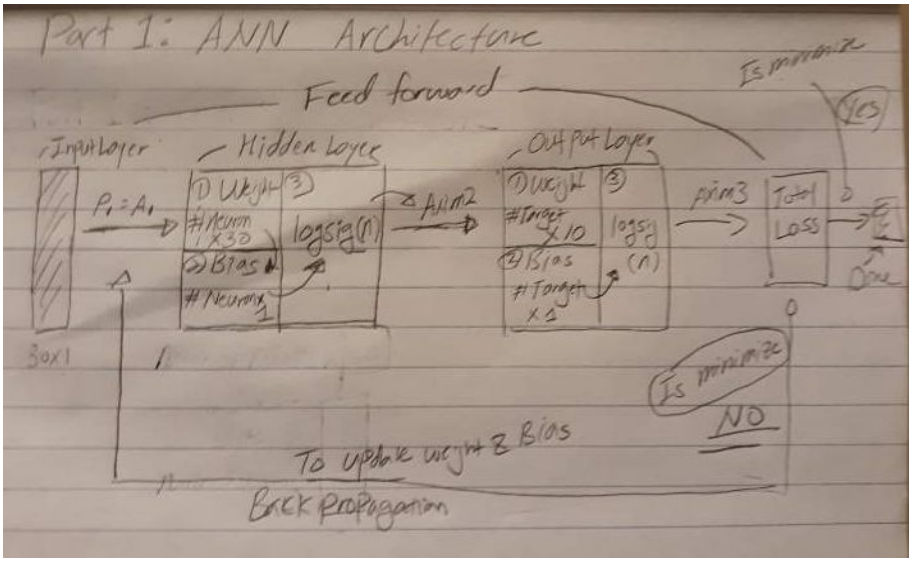


Figure 1.0-Part 1 ANN Architecture: This figure represents the 3 layer ANN architecture that was set up for part 1, with size of

- With input size is 30 x 1
- With target size is 3 x 1
- Hidden Layer Weight = Number of Neuron x input Size
- Hidden Layer Bias = Number of Neuron x 1
- Output Layer Weight = target size x Number of Neuron
- Output Layer Bias = target size x 1

and used logsig as a transfer function. We considered layer 0 as layer 1, therefore $P_1 = A_1$ in this report means $P_0 = A_0$

Once we set up the architecture, we decided to give some random small value for learning rate, which was 0.1 and number of neuron, which was 30.

Secondly, by using 30 neurons, we created matlab function called initialize, which presented in Appendix Section at the end of the report to create random weights and bias for both hidden layer and output layer.

Thirdly, by following the architecture, we used a feedforward method to obtain total loss or error, which was obtained by using the following equation.

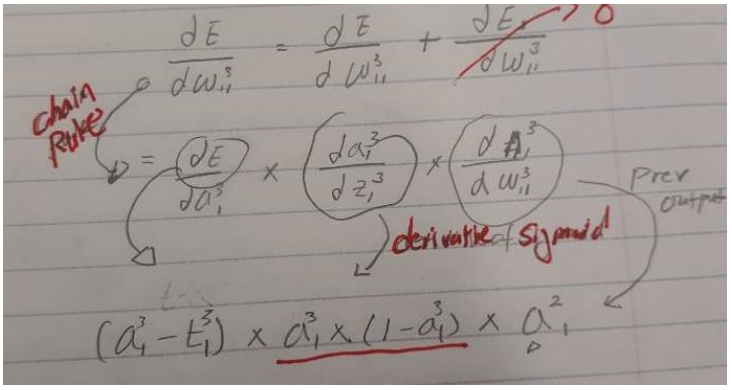
- n_1 = input
- $axion_1$ = input
- n_2 = (hidden layer weight * $axion_1$) + hidden layer bias
- $axion_2$ = logsig(n_2)
- n_3 = (output layer weight * axion2) + output layer bias
- $axion_3$ = logsig(n_3)
- totalError = target - axion3

The goal of feedforward on our ANN architecture is to obtain error value, however, this process has to be iterative to obtain minimum error value and if it is not minimum the weight and bias has to keep update to obtain minimum error, in other word, use gradient descent.

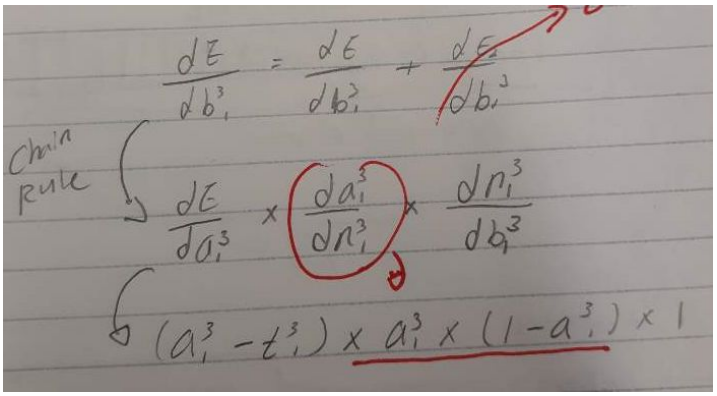
To achieve the goal, the following equation had been used, which can be obtained from the Appendix section in matlab code.

- E(error) = $E_1 + E_2$, which error can be obtain target - a
- outputLayer Axion = sigmoid(outputLayer n)
- derivative of sigmoid = (sigmoid(n) * (1 - sigmoid(n))
- sigmoid(n) = a

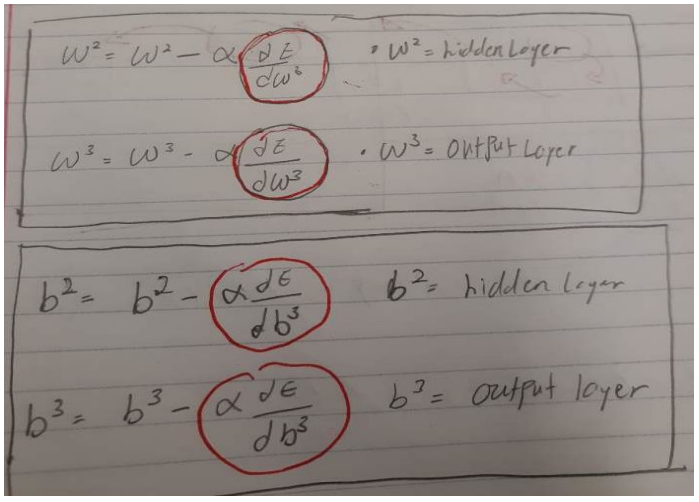
By using the equation above, we came up with the following equation.



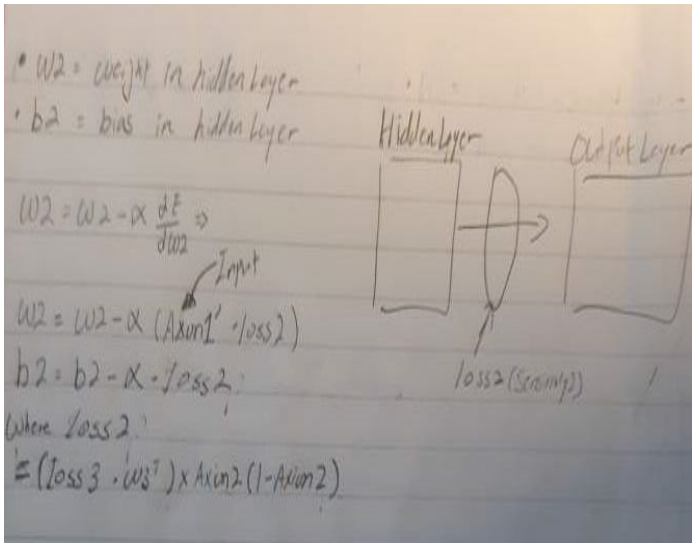
and for equation to get effect on E (error or loss) when bias change can be driven to math equation below.



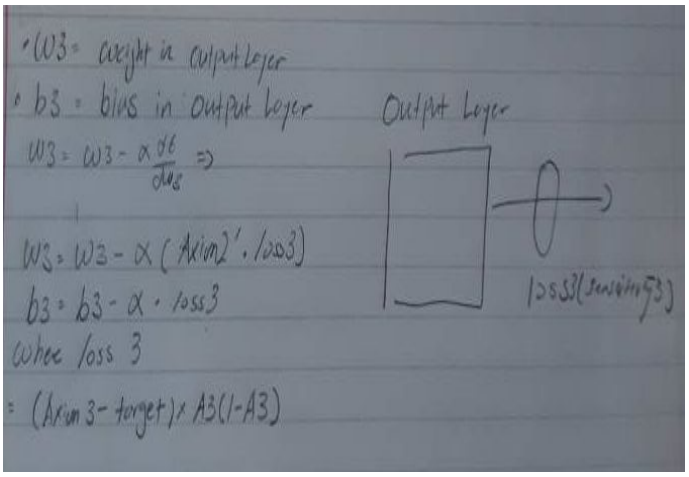
Then we used the formula above to update weight and bias, for instance, the equation of updating the weight can be represented with:



Finally, the equation for weight and bias in hidden layer can be represented with:



and equation for weight and bias in output layer can be represented with:



The following code is presented in the Appendix Section in "training" function.

By performing this equation iteratively, we were able to graph the mean square error for the ANN Architecture, which presented in figure 1.1- Training Weights and Hidden Layer.

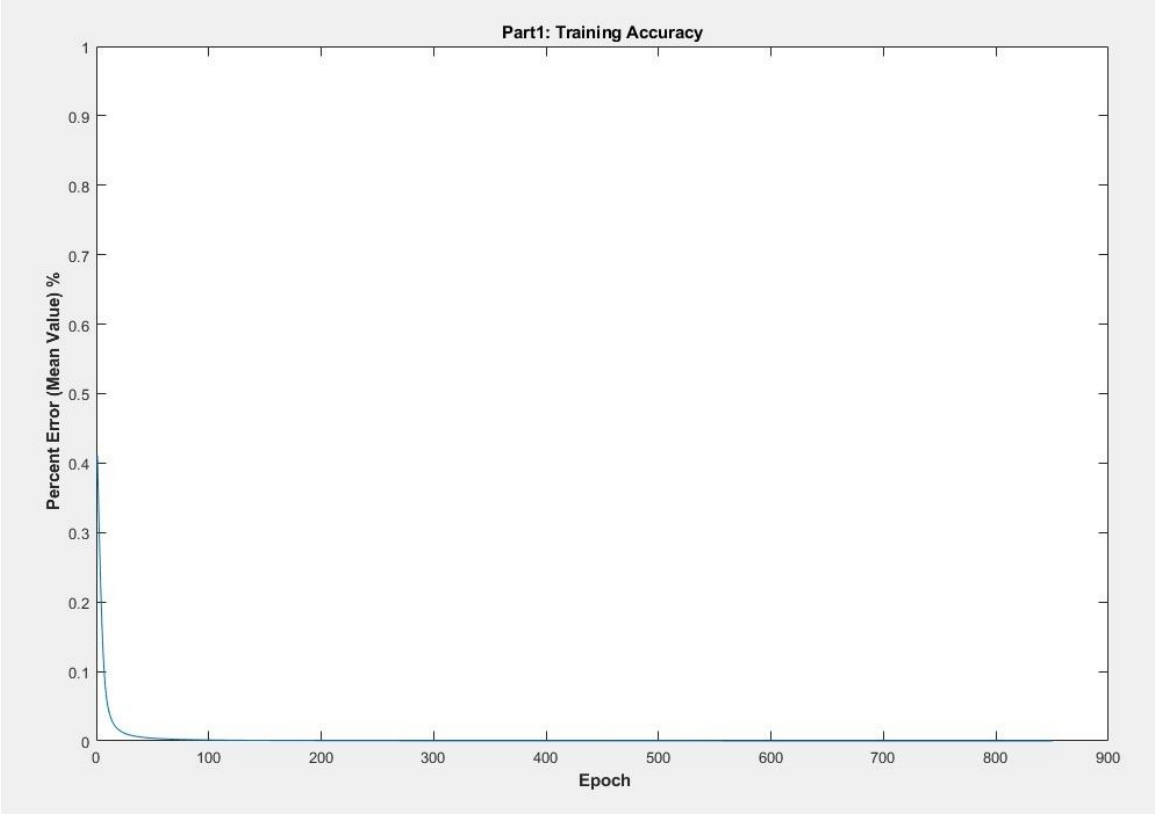


Figure 1.1-Training Weights of Hidden Layer: This figure represents the percent error that the network displays during 850 runthrough or epochs. The percent error is the mean square error that is calculated after every epoch.

Once The network was trained it was then tested on digits that have had noise added to them. The number of pixels changed in each digit was 0, 4, and 8 pixels. 30 images were created for

each test and then compared to the targets to see if the network could correctly identify the digit.

Results part 1

The results of part one can be seen in figure 1.1 with how well the network did with 0, 4, 8 pixels changed. With 0 pixels the network did perfectly but as the number of pixels changed increased the network performed less. The network has shown the ability to learn and still be able to correctly identify some noisy digits even with 8 pixels changed.

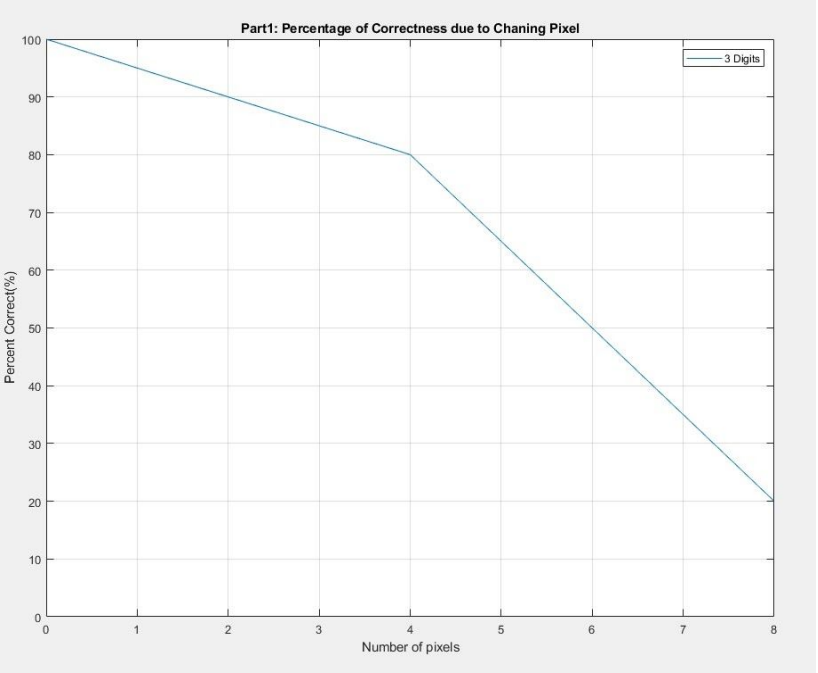


Figure 1.2-Correctness of network against pixels changed: This figure represents how well a network based off of 3 digits does with 0, 4, and 8 pixels changed. The y axis is the percent correct out of 30 images for each test.

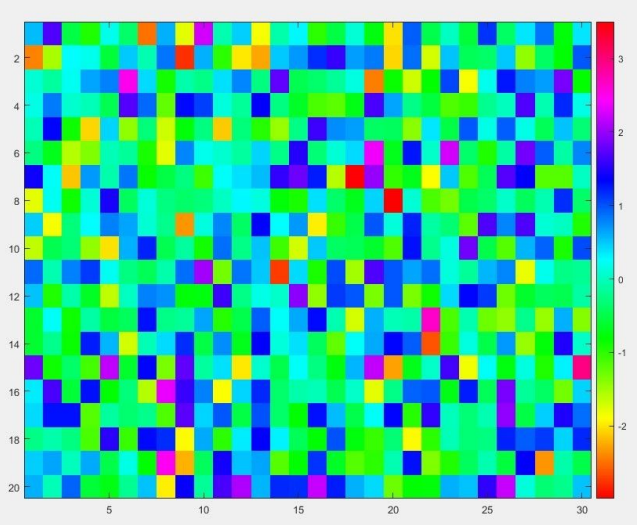


Figure 1.3-Hidden Layer Weight Matrix: This figure represents the distributions of the weights in the hidden layer of the network.

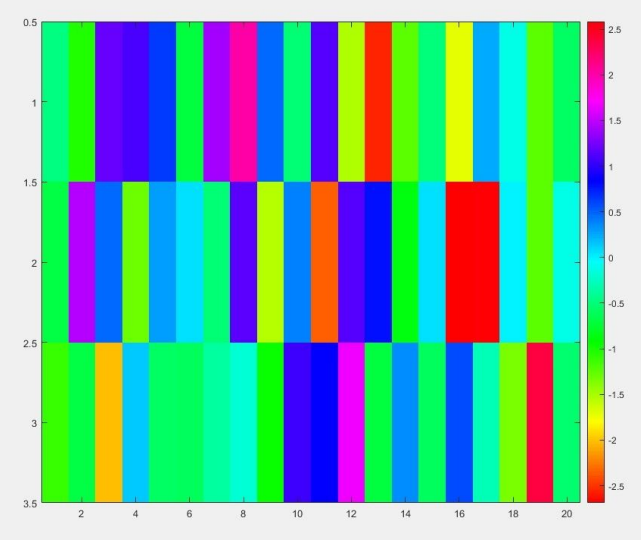


Figure 1.4-Output Layer Weight Matrix: This figure represents the distributions of the weights in the output layer of the network.

PART 2: HANDWRITING RECOGNITION WITH THE MNIST DATABASE

Part 2 was to train the handwritten number by using train.csv file by using backpropagation and test the trained weight and bias by using test.csv file to measure the accuracy of the trained weight and bias. To accomplish the following problem, group 13 had processed the following method represented in Method Part 2 section.

METHOD PART 2

First of all, before we started to design the ANN architecture, which is similar architecture to Part1, but different input size, target size, number of neuron, and learning rate we had to understand the train.csv file. Since the train.csv file contains 60000 rows with 785 columns. The first column is representing the value that 784 elements represents. 784 elements are in grayscale values rgb(0-255) and output was 0 to 9. Therefore, by using the following

information, we came up with the following network architecture, which presented in figure 2.0: ANN Architecture for MNIST dataset.

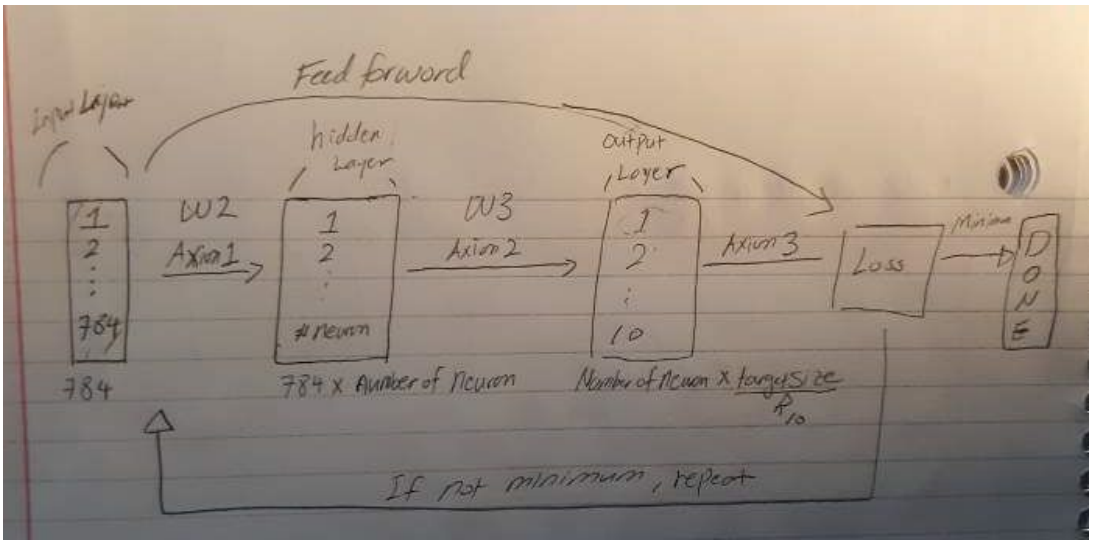


Figure 2.0-ANN Architecture for MNIST dataset: This figure represents the 3 layer ANN architecture that was set up for part 2, with size of

- With input size is 1 x 784, because there are 784 elements and group 13 will loop it through 60000 times, to insert into function.
- With target size is 1 x 10, because it's 0 to 9
- Hidden Layer Weight = 784 x Number of Neuron
- Hidden Layer Bias = 1 x Number of Neuron
- Output Layer Weight = Number of Neuron x target, (10) because target 0 - 9
- Output Layer Bias = 1 x 10

and used logsig as a transfer function. Group 13 had considered layer 0 as layer 1, therefore $P_1 = A_1$ in this report means $P_0 = A_0$

Secondly, we created a function to create randomize weight and bias based on the size we displayed in figure 2.0. However, unlike Part1, we decided to normalize the weight and bias for both hidden layer and output layer. To initialize, we used xavier initialization, which we found during our neural network exploration. We used this to initialize the random weight in hidden and output layer more efficiently, which is displayed in matlab code in Appendix section MatLab Code 2.0: Backpropagation, initialize function.

Thirdly, after initialize data into weight and bias for both hidden layer and output layer, group we performed a feedforward to obtain total error, which is exactly same procedure and logic proposed in Part 1, therefore please refer part 1 method with math calculation, but for part2, it is looping or training epoch * row (60,000) times to update for weights and bias for both hidden layer and output layer.

Each input and target data that goes into the loop has to be normalized, which was another explore that we explored. Since, within 784 data, it is between 0 to 255, therefore, by dividing into 255.0 and multiply 0.99 and add 0.01, all the inputs are now prevent 0~1. The output has to be changed since we are using one-hot encoding, to achieve set all targets (1 x

10) to 0.01 and change the target value's index to 0.99. For example, if target value is 5, then it will be represent like the following

0.01	0.01	0.01	0.01	0.99	0.01	0.01	0.01	0.01	0.01

Finally, we used test.csv to test the trained weight and bias and recorded its performance.

Results part 2

The final results can be seen in figure 2.2 when using the test set with the final average being 94.3%. The network was able to learn through the training set to be able to correctly identify numbers when given the training set.

We decided to explore how normalizing the input and weights would affect how long it took the network to learn. In figure 2.0 it shows the training of the network before normalization where it went from about 4.7% total loss to 3.3 % total loss with 4 epochs. However when we did normalize the inputs and weights as seen in figure 2.1 the total loss was about 1.7% within only one epoch. While they are both converging when we normalized the inputs and weights we were able to converge much faster leading to us only having to run 1 epoch as opposed to many when it was not normalized.

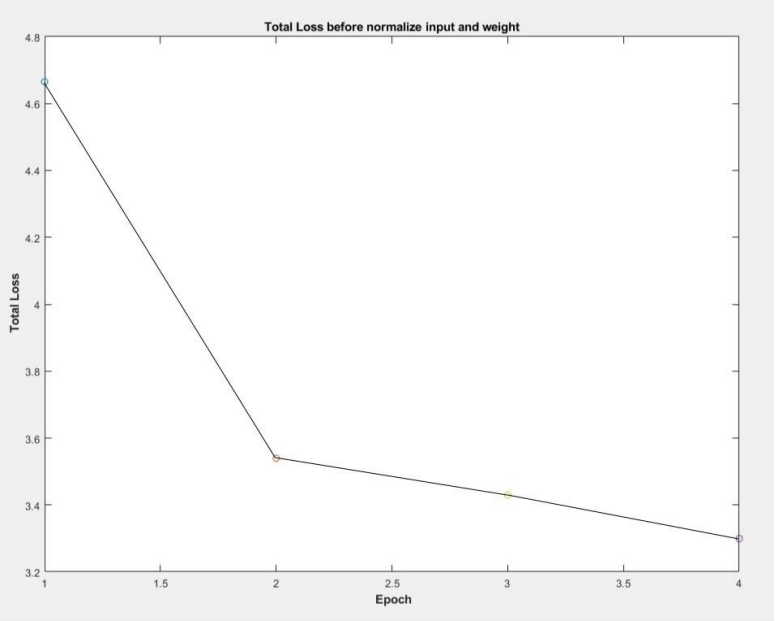


Figure 2.0-Training Speed Before Normalization: This figure shows the total loss that the network generated over 4 epochs and before normalization. The y axis is how much the loss was generated over an epoch. While the x axis is the number of epochs.

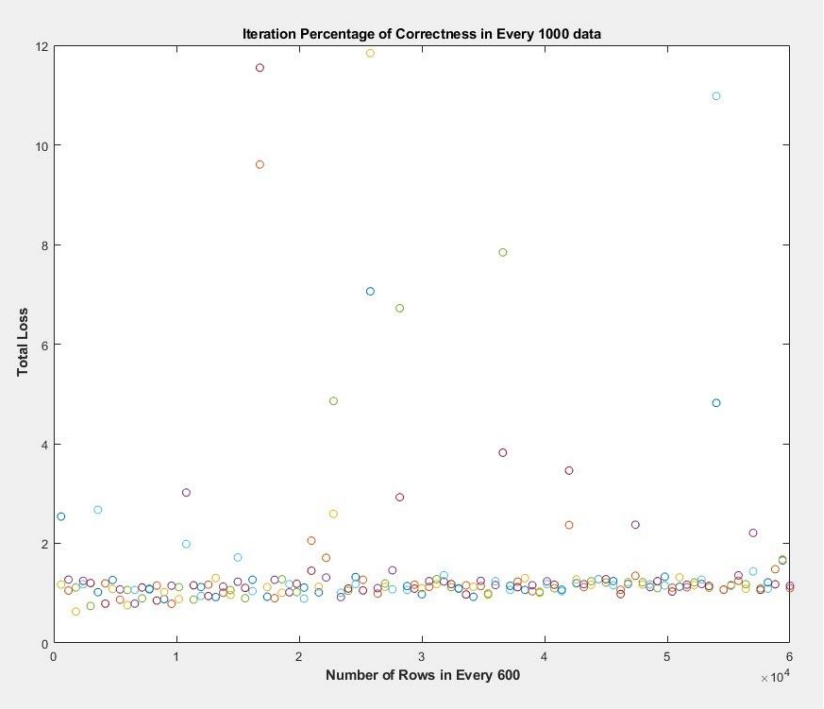


Figure 2.1-Training Speed with Normalization: This figure shows the total loss that the network generated in one epoch with weights and inputs normalized. The y axis is how much loss was generated by 600 inputs. The x axis are the number of inputs grouped in 600.

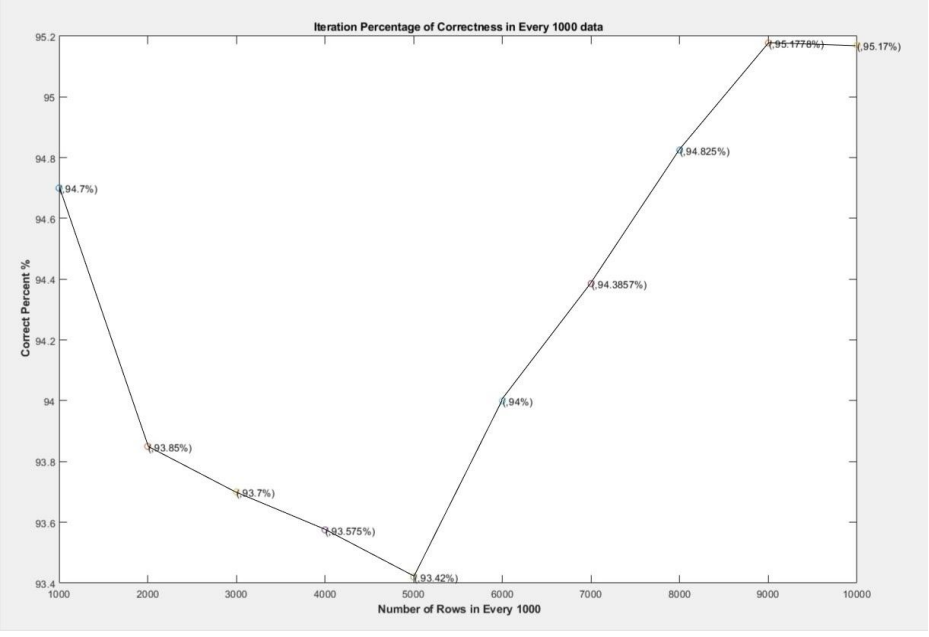


Figure 2.2-Average Percentage on test data: This figure represents how well the network did on the test set. Every 1000 data points the average correctness was calculated and plotted.The y axis is the percentage that the network got correct. The x axis are 1000 test set data points put together.

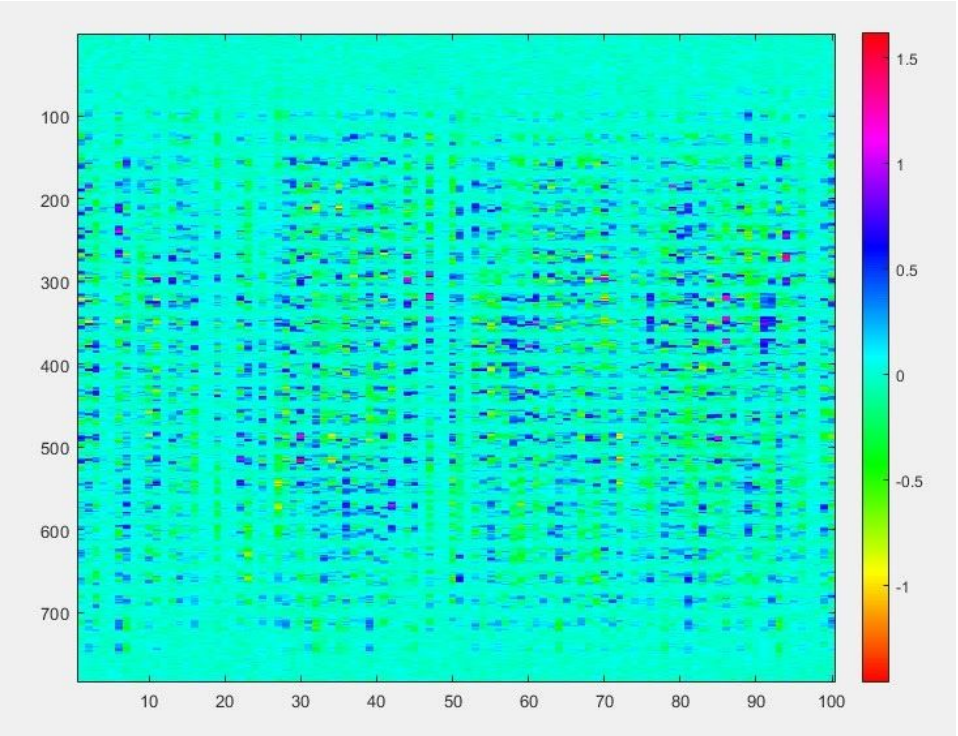


Figure 2.3-Hidden Layer Weight Matrix: This figure represents the distributions of the weights in the hidden layer of the network.

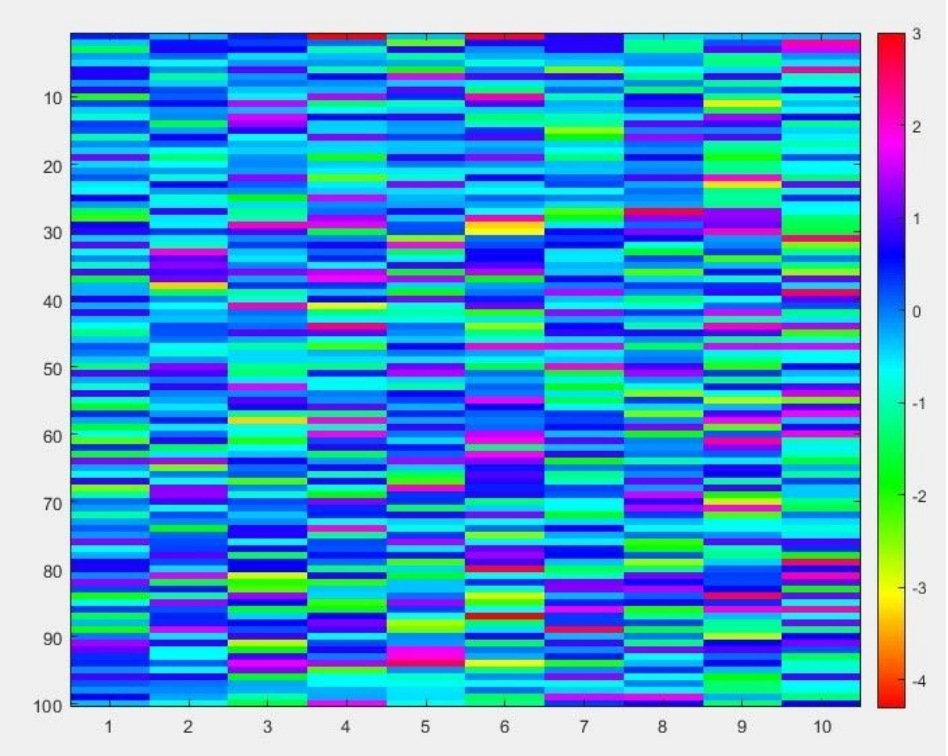


Figure 2.4-Output Layer Weight Matrix: This figure represents the distributions of the weights in the output layer of the network.

CONCLUSION

Backpropagation is very useful at training a network and then using the network to identify new inputs. We were able to train a large network on the Mnist dataset and then able to test it with the Mnist training set. We found that our network was able to correctly identify numbers with 94.3%. We were able to get this number higher about 97% when we ran the network through 100 epochs, however that took a long time to run.

We found that normalizing the weights and inputs are super important for wanting to train a network faster. When we normalized we were able to reduce the total loss immediately whereas with it not normalized it would still converge but take a longer time. This shows that preprocessing is super important to large networks with large variations in numbers to reduce the time of convergence.

In the future it might be interesting to explore running the network in mini-batches to help us reduce the amount of time we spend in an epoch. This is because the longest part of our implementation was waiting on the training due to the size of the epochs. We were worried about it being too big to make an array in MatLabs so in the future with larger dataset we could split it into min-batches so that we don't strain our computers as much.

APPENDIX

MatLab Code 1.0: Backpropagation Part 1

```
format compact
% creating input and output variable
input0 = [0 1 1 1 1 0 1 0 0 0 0 1 1 0 0 0 0 1 1 0 0 0 0 1 0 1 1 1 1 0]';
input1 = [0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0]';
input2 = [1 0 0 0 0 0 0 1 0 0 1 1 1 1 0 0 1 0 1 0 1 1 0 0 1 0 0 0 0 0 1]';
target0 = [1 0 0]';
target1 = [0 1 0]';
target2 = [0 0 1]';
% set up input and output
input = [input0 input1 input2];
target = [target0 target1 target2];
NEURON = 20;
learningRate = 0.45;
% create randome hidden layer weight and bias
[hiddenWe, hiddenBi, outputWe, outputBi, learningR] = initialize(input, target, NEURON,
learningRate);
% initialize the variable
hiddenWeight = hiddenWe;
hiddenBias = hiddenBi;
outputWeight = outputWe;
outputBias = outputBi;
learningRate = learningR;

% ==========Start=======================
dispGraph = 1;
```

```
[outputW, outputB, hiddenW, hiddenB] = training(input, target, hiddenWeight,
hiddenBias, outputWeight, outputBias, learningRate, dispGraph);
myHiddenW = hiddenW;
myHiddenB = hiddenB;
myOutputW = outputW;
myOutputB = outputB;
predictedValue = predict(input, hiddenW, hiddenB, outputW, outputB);
% display the weight using the fig
figure()
imagesc(myOutputW);
colormap(hsv);
colorbar;

figure()
imagesc(myHiddenW);
colormap(hsv);
colorbar;
% end of display weight

% =======Calling function for Changing pixel ================
graph = [0 0 0];

for j = 1:3
   graph(j) = doCalculateWithWeight(input, ((j-1)*4), myHiddenW, myHiddenB,
myOutputW, myOutputB, target);
end

figure()
xAxis = [0,4,8];
pixels0 = plot(xAxis, graph());
title('Part1: Percentage of Correctness due to Chaning Pixel')
xlabel('Number of pixels'), ylabel('Percent Correct(%)'), M1 = "3 Digits";
hold on
legend([pixels0;], [M1]);
grid on, axis([0 8 0 100]);

% ============End=========================
function count = doCalculateWithWeight(inputPx, numberOfPixelChange, myHiddenW,
myHiddenB, myOutputW, myOutputB, target)
   [row, col] = size(inputPx);
   tempPxho = [];
   totalCorrect = 0;
   for i = 1:10
      for k = 1:col
         eachDigit = inputPx(:,k,:);
         length(eachDigit);
         % fprintf("Changing Now\n")
```

```
        tempPxho(:,k,:) = addNoise(eachDigit, numberOfPixelChange);
      end
      % axion1, p0 == a0
      axion1 = tempPxho;
      % axion2
      hiddenN2 = myHiddenW * axion1 + myHiddenB;
      hiddenAxion2 = logsig(hiddenN2);
      % axion3
      outputN3 = myOutputW * hiddenAxion2 + myOutputB;
      outputAxion3 = logsig(outputN3);
      %  ================ Error guy ==================
      tempError = 0;
      [aM,aI] = max(outputAxion3);
      [tM,tI] = max(target);
      if tI == aI
        totalCorrect = totalCorrect + 1;
      end
      tempPxho = inputPx;
   % End of Error uy
   end
   totalCorrect = totalCorrect / 10;

   count = totalCorrect*100;

end
% ==============Function Call==============
% ==============Adding Noise to input ============================
% ============Add Noise Function==============
function pvec = addNoise(pvec, num)
   % ADDNOISE Add noise to "binary" vector
   %   pvec pattern vector (-1 and 1)
   %   num  number of elements to flip randomly
   % Handle special case where there's no noise
   if num == 0
      return;
   end
   % first, generate a random permutation of all indices into pvec
   inds = randperm(length(pvec));
   % then, use the first n elements to flip
   pvec(inds(1:num)) = -pvec(inds(1:num));
end
% ==============End of Noise to input=========

% ================= End of Testing Chagning ====================
function [hiddenWeight, hiddenBias, outputWeight, outputBias, learningRate] =
initialize(input, target, neuron, learningR)
   inputSize = length(input);
   targetSize = length(target);
```

```
    % initalize hidden layer Weight and Bias
    hiddenWeight = randn(neuron, inputSize);
    hiddenBias = rand(neuron, 1);
    % initialize output layer Weight and Bias
    outputWeight = randn(targetSize, neuron);
    outputBias = rand(targetSize, 1);
    learningRate = learningR;
end

function [axion1, axion2, axion3, totalError] = feedforward(input, target, hiddenW,
hiddenB, outputW, outputB)
    % to prevent log go to infinte
    % input p0 == a0
    % I just decided to n1 == p0
    n1 = input;
    axion1 = input;
    % hidden layer
    n2 = (hiddenW * axion1) + hiddenB;
    axion2 = logsig(n2);
    % output layer
    n3 = (outputW * axion2) + outputB;
    axion3 = logsig(n3);
    totalError = target - axion3;
end

function [outputW, outputB, hiddenW, hiddenB] = training(input, target, hiddenW,
hiddenB, outputW, outputB, learningRate, dispGraph)
    myInput = input;

    % my testing for normalize the input and output
    target1 = [0.99 0.01 0.01]';
    target2 = [0.01 0.99 0.01]';
    target3 = [0.01 0.01 0.99]';
    myTarget = [target1, target2, target3];

    % end of my testing

    myLearningRate = learningRate;
    % putting errors to Error Vector
    epoch = 900;
    disp("ErrorSize")
    ErrorVector = zeros(1,epoch);
    if dispGraph == 1
        figure()
    end
    [row, col] = size(target)
    % ==========EPOCH ==============
    for j = 1:epoch
```

```
        tempError = 0;
        for i = 1:col
            % normalize the input data
            myTestInput = ((input(:,i) / 1.0) * 0.99) + 0.01;
            [axion1, axion2, axion3, totalError] = feedforward(input(:,i), myTarget(:,i),
hiddenW, hiddenB, outputW, outputB);

            loss3 = (axion3 - myTarget(:,i)) .* axion3 .* (1 - axion3);
            outputW = outputW - learningRate * (loss3 * axion2');
            outputB = outputB - learningRate * loss3;
            loss2 = (outputW' * loss3) .* axion2 .* (1 - axion2);
            hiddenW = hiddenW - learningRate * (axion1' .* loss2);
            hiddenB = hiddenB - learningRate * loss2;

            tempError = tempError + totalError' * totalError;
        end
        tempError = tempError / (col*col);
        ErrorVector(j) = tempError;
        if dispGraph == 1
              plot(ErrorVector(1:j));
              ylim([0,1]);
              drawnow();
              title('Part1: Training Accuracy')
              xlabel(["\bf Epoch"])
              ylabel(["\bf Percent Error (Mean Value) %"])
         end
    end
end

% ========== predict function ========================
% ========== use w and b that is corrected by training
function predicted_num = predict(inputData, hiddenW, hiddenB, outputW, outputB)
    predictN2 = hiddenW * inputData + hiddenB;
    predictAxion2 = logsig(predictN2);
    predictN3 = outputW * predictAxion2 + outputB;
    disp("predict")
    predictAxion3 = logsig(predictN3);
    predictedValue = [];
    [row, col] = size(predictAxion3);
    predicted_num = predictAxion3
end

% ==========Accuracy=====================
function count = accuracy(axion3 ,targetOuput)
     [aM,aI] = max(axion3);
     [tM,tI] = max(targetOuput);
      count = tI == aI;
end
```

MatLab Code 2.0: Backpropagation Part 2

```
format compact
% input data is 0 to 255
testData = csvread('mnist_train.csv');
[row, col] = size(testData);

% setting up the basic value
inputNode = 784; % 28 * 28
hiddenNode = 100; % num of neuron
outputNode = 10;
learning_rate = 0.3;
epochs = 1;

% ============ initialize the variables ==============
[hiddenW, hiddenB, outputW, outputB, learningRate] = initialize(inputNode, outputNode,
hiddenNode, learning_rate);
hiddenWeight = hiddenW;
hiddenBias = hiddenB;
outputWeight = outputW;
outputBias = outputB;
learningRate = learningRate;
% ============ end of initalize ========================

% ========== Start to call training function ==============
[outputW, outputB, hiddenW, hiddenB] = training(hiddenWeight, hiddenBias,
outputWeight, outputBias, learningRate, testData);

% ========== Weight and Bias to Used ===============
finalHiddenW = hiddenW;
finalHiddenB = hiddenB;
finalOutputW = outputW;
finalOutputB = outputB;

% ========= open test csv file ==============
finalTestData = csvread('mnist_test.csv');
[testDataRow, testDataCol] = size(finalTestData);

anoterVari = 0;
graphChecker = 0;
for iterateTest = 1:testDataRow
    % normalize the input
    myInput = ((finalTestData(iterateTest,2:testDataCol) / 255.0) * 0.99) + 0.01;
    myTarget = finalTestData(iterateTest, 1:1);
    predictedValue = predict(myInput, finalHiddenW, finalHiddenB, finalOutputW,
finalOutputB);
```

```
    returnPercentage = accuracy(predictedValue, myTarget);
    anoterVari = anoterVari + returnPercentage;
end
anoterVari;

finalErrorPercent = (anoterVari / testDataRow) * 100

% professor's function to display weight figure
% figure()
% imagesc(outputW);
% colormap(hsv);
% colorbar;

% figure()
% imagesc(finalHiddenW);
% colormap(hsv);
% colorbar;

% ================= System Call =======================
% ================= initial function
function [hiddenWeight, hiddenBias, outputWeight, outputBias, learningRate] =
initialize(input, output ,neuron, learningR)
    % initalize hidden layer Weight and Bias
    hiddenWeight = randn(input, neuron) / sqrt(input / 2);
    hiddenBias = rand(1,neuron);
    % initialize output layer Weight and Bias
    outputWeight = randn(neuron, output) / sqrt(neuron / 2);
    outputBias = rand(1, output);
    learningRate = learningR;
end

function [axion1, axion2, axion3, totalError] = feedforward(input, target, hiddenW,
hiddenB, outputW, outputB)
    delta = 1e-7;
    % input p0 == a0
    % I just decided to n1 == p0
    n1 = input;
    axion1 = input;
    % hidden layer
    n2 = (axion1 * hiddenW) + hiddenB;
    axion2 = logsig(n2);
    % output layer
    n3 = (axion2 * outputW) + outputB;
    axion3 = logsig(n3);
    %  ============== made change on summing total error ==============
    totalLoss = sum(target .* log(axion3 + delta) + (1 - target) .* log((1-axion3) + delta));
    totalError = -totalLoss;
end
```

```
% erased input and target parameter
function [outputW, outputB, hiddenW, hiddenB] = training(hiddenW, hiddenB, outputW,
outputB, learningRate, setTrainingData, dispGraph)
   [totalRow, totalCol] = size(setTrainingData)
   % input data = 784 and target is 10
   %trainingInput = input;
   %trainingTarget= target;
   % need to put these inside the loop
   % totalRow = 60000
   dataForGraph = 0;
   epoch = 2
   for j = 1:epoch
      for i = 1:totalRow
         % normalize the target output for hot-one encoding
         target_data = zeros(1, 10) + 0.01;
         target_data(setTrainingData(i,1)+1) = 0.99;
         % myTestInput is the 784 to put inside feedforword as input
         myTestInput = ((setTrainingData(i,2:totalCol) / 255.0) * 0.99) + 0.01;
         [axion1, axion2, axion3, totalError] = feedforward(myTestInput, target_data,
hiddenW, hiddenB, outputW, outputB);
         totalLoss = totalError;
         %
==========================================================================
=
         loss3 = (axion3 - target_data) .* axion3 .* (1 - axion3);
         outputW = outputW - learningRate * (axion2' * loss3);
         outputB = outputB - learningRate * loss3;

         loss2 = (loss3 * outputW') .* axion2 .* (1 - axion2);
         hiddenW = hiddenW - learningRate * (axion1' *loss2);
         hiddenB = hiddenB - learningRate * loss2;
         % ============= Below b and if statement graph for percentage
         % of correctness
         b = mod(i, 600);
         if b == 0
            %disp("Count" + i + " totalError:")
            totalLoss;
            dataForGraph = (totalLoss);
            plot(i,dataForGraph, '-o')
            title('Iteration Percentage of Correctness in Every 1000 data')
            xlabel(["\bf Number of Rows in Every 600"])
            ylabel(["\bf Total Loss"])
            hold on
         end
      end
      %plot(j,totalLoss, '-o')
      %title('Total Loss before normalize input and weight')
```

```
        %xlabel(["\bf Epoch"])
        %ylabel(["\bf Total Loss"])
        %hold on
    end
end

function predicted_num = predict(inputData, hiddenW, hiddenB, outputW, outputB)
    predictN2 = inputData * hiddenW + hiddenB;
    predictAxion2 = logsig(predictN2);
    predictN3 = predictAxion2 * outputW + outputB;
    %disp("predict")
    predictAxion3 = logsig(predictN3);
    predicted_num  = predictAxion3;
end

% returns the matched data compare to the first col of the
function myAccuracy = accuracy(predictedValue, firstColofTestData)
    [aM, aI] = max(predictedValue);
    valuePredicted = aI - 1;
    %disp("Display Predicted Value")
    valuePredicted;
    %disp("Display answer value")
    firstColofTestData;
    if valuePredicted == firstColofTestData
        myAccuracy = 1;
    else
        myAccuracy = 0;
    end
end
```