Mini Project

Mini-Project Team 4: Margaret Lanphere, Kier Fisher

Introduction

The final project for this course required us to put together all of our knowledge of multi-layer neural networks together to solve a complicated problem. This project was an extension of the previous backpropagation assignment, which involved the creation of a 2 layer network that identified handwritten digits. The MNIST Fashion Dataset for clothing images is much more challenging, since the difference between images within the same category is more pronounced. The more complicated nature of the dataset required us to further experiment with different configurations of neural network architecture.

Strategies we employed to maximize neural network accuracy included modifying the number of nodes in the hidden layer, the number of hidden layers and their size, different transfer functions in the output layer, and preprocessing the image data. Exploring the effect of these changes on the error rate of the network for a more challenging problem enabled us to gain a much greater understanding of how the design of a network affects its usability and effectiveness and the kinds of strategies that can be used to better train the network.

Methods

Using the Fashion-MNIST requires the creation of a more advanced neural network. The images that share labels have much more variance between themselves and are thus more difficult to quantify than the handwritten digit MNIST dataset.

Through our participation in the Kaggle competition, we utilized many different strategies to maximize the artificial neural network accuracy on the Fashion MNIST dataset. After the data was processed, the neural network architecture, transfer functions, and learning rate were varied and dataset augmentation was experimented with to try to improve network accuracy.

First, the dataset from the Kaggle competition website was processed for use with the previously written code for the handwriting MNIST dataset from the Backpropagation assignment. The first step was to partition the data into a training set and a test (or validation) set. The training set would be used to train the neural network's weights and biases and the test set would only be used to calculate the accuracy of the network. The initial split of the data was 70% training and 30% test, creating matrices of 42,000 and 18,000 columns, respectively. Then the IDs and labels were separated from the image pixel data and the image pixel data was scaled to values between 0 and 1. The labels were used to create a 10-by-42000 target matrix, where each column was 0s except for a lone 1 indicating the label of the corresponding image (like in previous assignments). The same process was repeated for the test portion.

The backpropagation algorithms developed for the previous assignment were improved during the work for this project. The matrix weights were originally random values between 0 and 0.1, but to increase the odds of covering the entire input space, they were converted to start between -0.1 and 0.1.

The first kind of neural network used was the 2 layer backpropagation algorithm with log-sigmoid transfer functions. Various hidden layer sizes were used to see how accuracy was impacted. The first results submitted to the Kaggle competition were from a 1-20-10 ANN with 800 iterations and an alpha value of 0.1. It had a Kaggle accuracy (public) score of 0.8297. There were 2 more days of competition, so it was just to test how to submit and compare Kaggle's accuracy value against the test dataset accuracy value. Those scores were found to be fairly close, less than 1 percent difference, but Kaggle's public score was always higher than our test accuracy score.

Then different versions of 3 layer neural networks were created and there was generally higher accuracy seen with these networks. The next Kaggle submission was submitted shortly before the next daily reset and had an accuracy of 0.86933 (Kaggle public leaderboard score). That network was a 3 layer ANN with 1-30-20-10 architecture, all log-sigmoid transfer functions, and ran for 1000 iterations with an alpha value of 0.1.

To try to further improve accuracy, the training data was augmented by adding "noisy" versions of existing images. A portion of the training set was duplicated, then run through a function that would alter, positively or negatively, the non-zero pixel values by a small percentage and kept within the bounds of 0 and 1. This set was then appended to the end of the training set, increasing the number of training images by 10,000 and 18,000. These sets were named “52kAug” and “60kAug” and were tested to see if they improved the network’s accuracy against the test dataset.

A 4 layer ANN was run to see if adding layers would improve the accuracy of the network for this problem. It was test 16 (labeled t16W1 in Table 1) with an architecture of 1-80-40-20-10 since it made logical sense to reduce the number of nodes by half with each layer. Adding layers was found to dramatically decrease performance against the test data (at least when using all log-sigmoid transfer functions), so that was not explored further.

Similar to the backpropagation assignment, the learning rate was also varied to see if larger or smaller values would improve the observed accuracy of the neural network. Most versions used 0.1 as the learning rate, but some were trained with either 0.075, 0.08, or 0.125 to see if that would improve the accuracy compared to the same test with a learning rate of 0.1.

Lastly, changing the output layer’s transfer function was also implemented to try to maximize the neural network’s performance. Functions were created for 2 and 3 layer ANN with the output layer transfer function changed from log-sigmoid to softmax. The softmax function has the added weighting to the highest probability label and the de-weighting for the less probable labels, since all the values must sum to 1, so a higher accuracy rate was expected. The rectified linear unit (ReLU) transfer function was also experimented with, but initial tests showed low performance which may have been due to implementation error, so those were not explored further.

Results

The first successful neural network used a familiar architecture from the handwriting MNIST project, with a 1-20-10 network size as seen in Table 1 as version t3W1. Using normalized training and test data, this provided an accuracy of about 83%. Since it was close to the daily reset of the Kaggle submission period, it was submitted to see how the test results compared to the Kaggle results. The validation test results were about 90%, while the Kaggle score was approximately 83%. This indicated that there was likely significant overfit after running so many iterations with this small ANN size, so then the datasets were double-checked for the proper split and normalization and the function call was checked to make sure the right data was being used. Later versions didn’t have large differences between validation test performance and Kaggle score.

Table 1: Summary of Different ANN Versions and Accuracy

Version Name	Notes	Architecture	Iterations	Alpha	Accuracy	Kaggle Accuracy
t3W1...	Normalized data, 42k training set size	1-20-10	800	0.1	0.907 (likely overfit)	0.8297
t4W1...	Increased hidden layer size, increased iterations	1-40-10	1000	0.1	0.8613	
t5W1...	Added additional hidden layer	1-30-20-10	1000	0.1	0.8639	0.8693

t7W1...	Using the 52kAug data	1-30-20-10	1000	0.1	0.8582	
t8W1...	52kAug data, larger hidden layer	1-40-20-10	600	0.1	0.8657	
t9W1...	52kAug data, bigger hidden layers	1-50-25-10	600	0.1	0.8689	
t10W1...	52kAug data, 2 layer with larger hidden layer	1-100-10	600	0.1	0.8687	
t11W1...	60kAug data, 3 layer	1-25-15-10	800	0.125	0.8597	
t12W1...	60kAug data, 3 layer	1-25-15-10	600	0.08	0.8606	
t13W1...	60kAug data, 3 layer with larger layers	1-80-20-10	600	0.08	0.8692	0.874
t15W1...	60kAug data, 3 layers, increased alpha to 0.1 and weights now start at [-0.1, 0.1] instead of [0, 0.1]	1-80-20-10	500	0.1	0.8736	0.87766 (public score of last Kaggle entry)
t16W1...	4 layer ANN, 60kAug set	1-80-40-20-10	500	0.1	0.2013	
t18W1...	3 layer ANN, last layer with softmax, 60kAug	1-80-20-10	300	0.1	0.8736	
t19W1...	3 layer ANN, smaller alpha, sig-sig-softmax	1-80-20-10	300	0.08	0.8634	
t20W1...	2 layer ANN, sig-softmax, using 85/15 training/test split data (85Tr)	1-40-10	300	0.08	0.87	
t21W1...	3 layer ANN, sig-sig-softmax, 85Tr dataset	1-50-25-10	250	0.075	0.8683	
t22W1...	2 layer ANN, sig-sig, 42k dataset	1-100-10	600	0.1	0.8636	
t23W1...	2 layer ANN, sig-softmax, 42k dataset	1-100-10	600	0.1	0.8387	
t24W1...	2 layer ANN, sig-sig, 60kAug dataset	1-100-10	300	0.1	0.8723	

The next leap forward in accuracy was the increase in hidden layer nodes to 40, using a 1-40-10 architecture. This raised the observed accuracy with the validation test data to 86.1%. Then, a small increase in accuracy was found with the addition of another hidden layer, using an architecture of 1-30-20-10, raising the accuracy to about 86.4%. This ANN was the highest performer at the time, so it was used to generate the next set of Kaggle data and the public leaderboard score was 86.93%, which was nearly half a percent higher than the validation test.

Then the dataset was augmented with noisy versions, to expand the number of training images to reduce the effect of overfitting. There wasn't a very significant increase in accuracy observed from training on these additional images, as seen in Figure 1, but having additional data seemed to help a little. Generally, the ANN accuracy seemed to slightly increase with the addition of the noisy images to the training set.

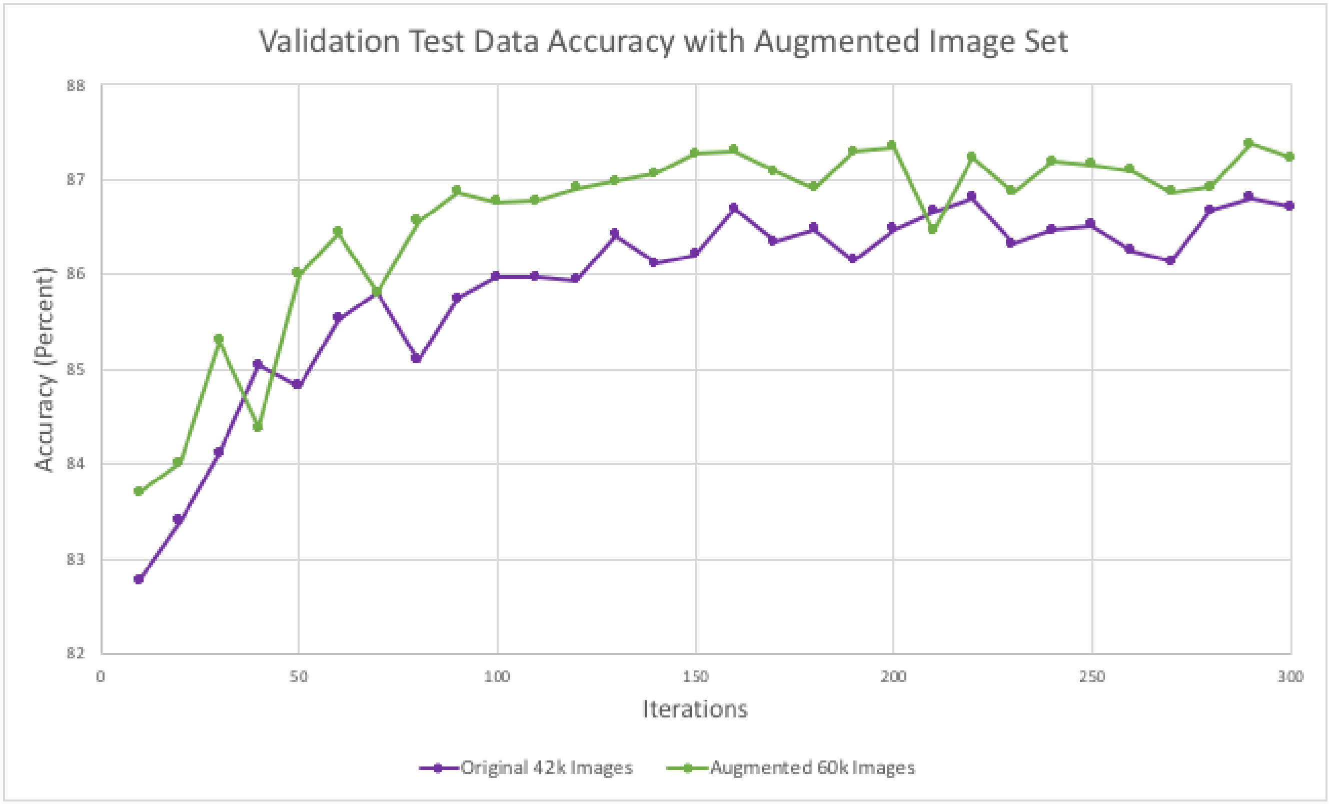

Figure 1: Comparing the changes in validation test set accuracy between the original 42,000 image training dataset and the augmented 60,000 image training set over time. Both used a 1-100-10 ANN architecture with a learning rate of 0.1 over at least 300 iterations, with validation tests every 10 iterations.

For the final Kaggle competition submission, a 1-80-20-10 ANN was used with the "60kAug" augmented training set for 500 iterations. It had 87.3% accuracy against the validation test dataset, which was the highest value recorded. The Kaggle public leaderboard score was 87.767%, which was a bit higher again. This ended up being our last submission since subsequent tests did not have significant improvement, and there was the danger of uploading data with a worse score. Upon the end of the competition, our final Kaggle private leaderboard score was lower than the public score, but the difference was less extreme for our group, causing us to switch places with another team to take 5th place with a private score of 87.185%.

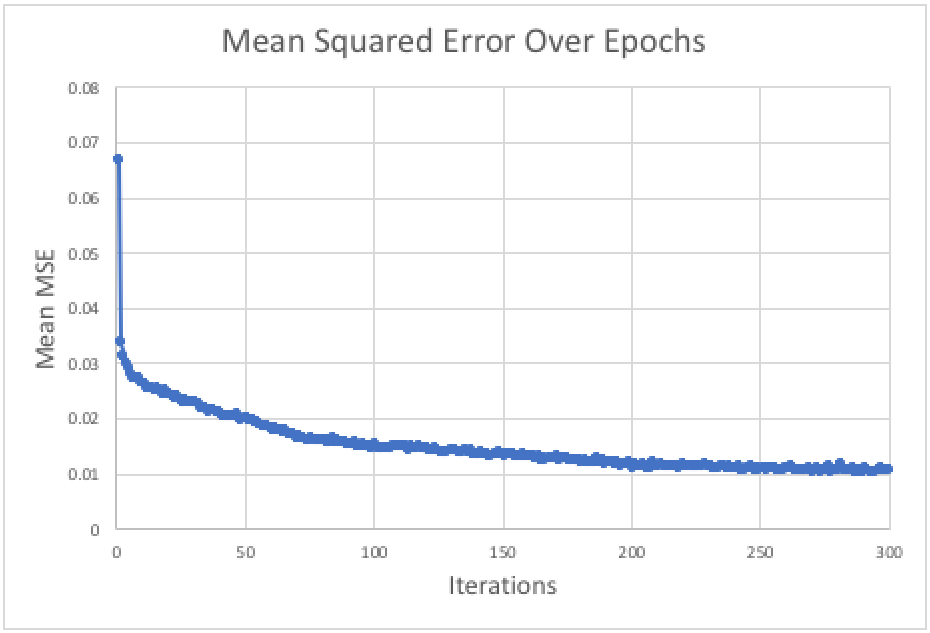

Figure 2: Comparing the changes in MSE over time for test 24. This graph is from data collected in t24W1 using an ANN with architecture 1-100-10 and log-sigmoid transfer functions over 300 iterations with an alpha value of 0.1.

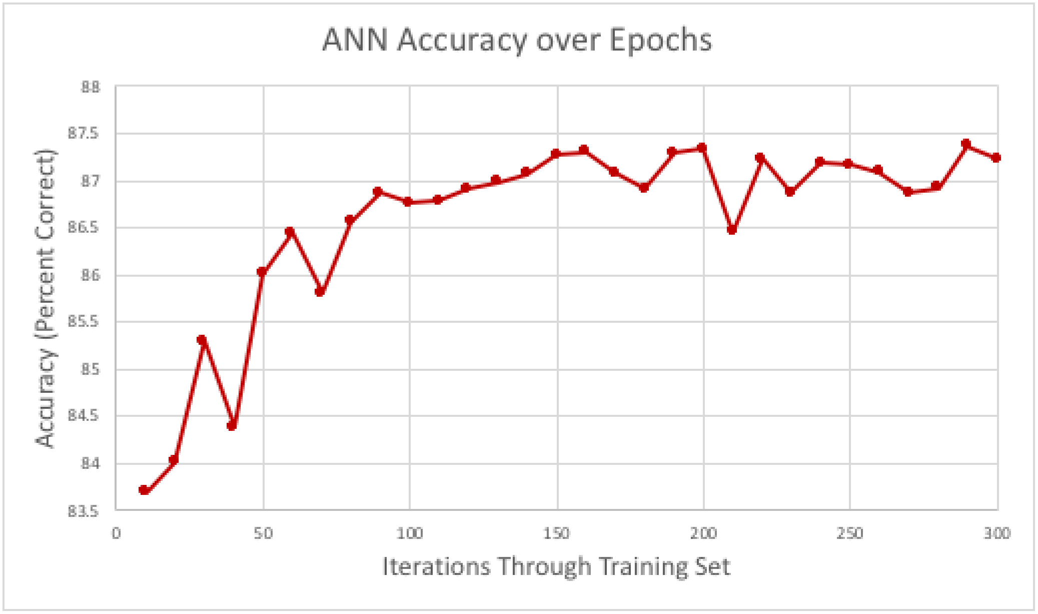

Figure 3: Comparing the changes in validation test accuracy over time for test 24. This graph is from data collected in t24W1 using an ANN with architecture 1-100-10 and log-sigmoid transfer functions over 300 iterations with an alpha value of 0.1.

Something that became apparent during these various ANN versions was that while MSE was steadily decreasing over time due to the backpropagation algorithm, the accuracy of the network's weights and biases increased only to a point. This point seemed to be under 200 iterations, where the validation test accuracy would then seemingly randomly increase and decrease in small amounts. It didn't seem to converge on a particular point, so very long-running training sessions of hundreds or a thousand iterations was not more beneficial than running shorter training sessions of only a few hundred iterations. As seen in Figures 2 and 3, which were collected from the same ANN, the MSE continues to slowly decrease while the observed accuracy peaks around 160 iterations and then fluctuates significantly.

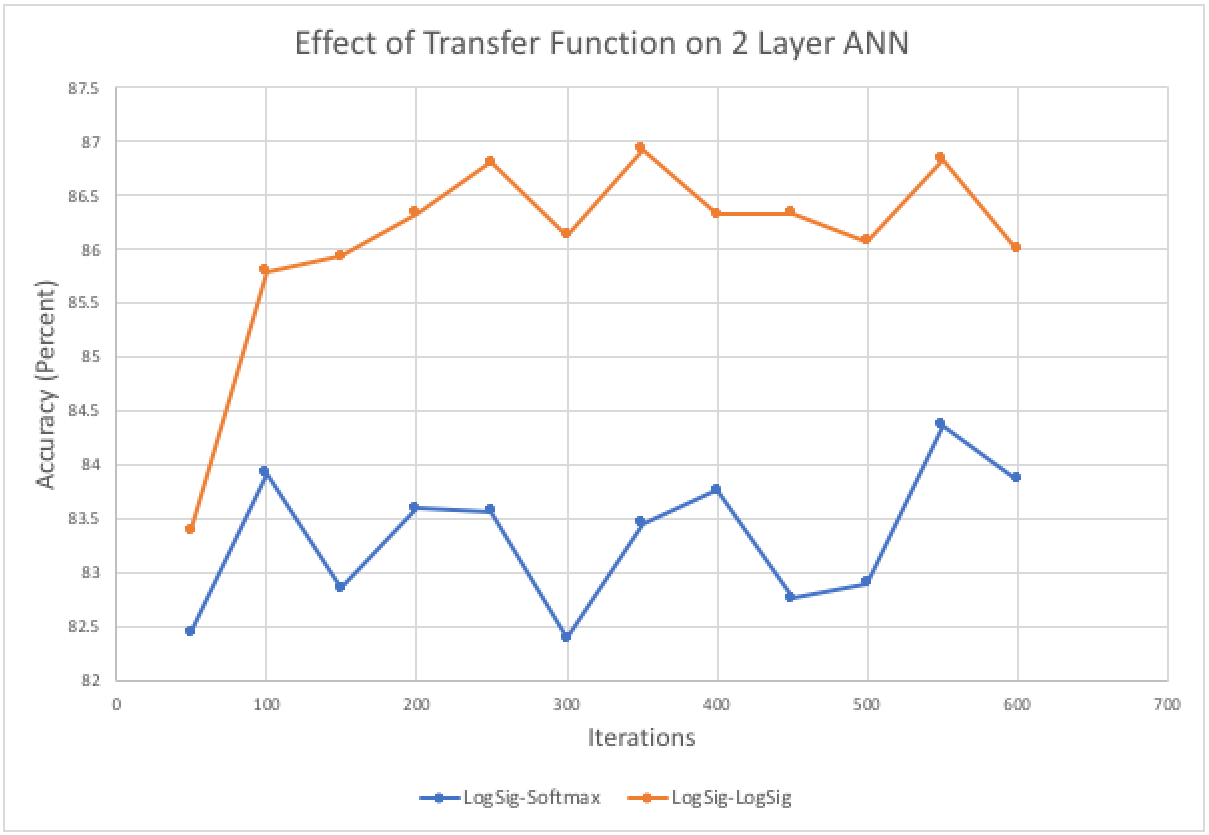

Figure 4: Using two ANN with the same 1-100-10 architecture, original 42k image training set, 0.1 learning rate, and 600 iterations duration, we compared the accuracy between an ANN with a log-sigmoid hidden layer transfer function and softmax output layer transfer function with one that had log-sigmoid for both layers.

Figure 4 shows the effect of changing a 2 layer ANN to having softmax as the transfer function for the output layer. This shows that it was not effective to implement that transfer function with such a small network. The 3 layer ANN with softmax transfer function for the output layer had a higher accuracy rate of over 87%. Since the 3 layer 1-80-20-10 ANN with log-sigmoid transfer functions for all 3 layers and the 3 layer 1-80-20-10 ANN with 2 log-sigmoid transfer functions and softmax for the output layer had identical final accuracy values of 87.36%, we did not observe a noticeable increase in accuracy from implementing the softmax transfer function.

Conclusions

This project involved the application of our increasing knowledge of neural network architecture in order to solve a difficult classification problem. We expanded upon the backpropagation algorithm from the previous assignment with more experimentation with the specifics of its implementation details to try to maximize accuracy on this more challenging dataset.

During the process of diving into the guts of a neural network, we learned just how complicated the process of designing a successful network can be. The huge amount of possible combinations of implementation details was intimidating at first. Iteratively changing the values of each parameter showed us that each one could have a major effect on the error rate of the network. Considering that each run through of the network took up to 80 minutes meant that we had to think carefully about how each change could impact the results. In the future it would be fulfilling to learn how to deduce which parameters could affect the biggest change in the results, as the effect on the network was never easy to anticipate. If we had more time, we would have explored implementing convolutional neural network strategies to further draw out the differences in the image data for better classification performance.

One surprising thing that we found when testing different numbers of hidden layers was its effect on the error rate. With different network architecture and other parameter variations, similar results were obtained between 2 and 3 layer networks, but adding a fourth layer dramatically lowered performance even with a long training time. Looking back at the lecture on convolutional neural networks it now makes sense why the multi-layer network did not perform as well, since more layers require more extensive implementation details to correctly classify inputs. Our (relatively) simple network with 2 to 3 layers was better suited to the simpler alterations of the backpropagation algorithm.

One of the more difficult aspects of this project was writing extensible matlab code and trying to save the best results from different ANN variations. In order to test different network architectures, the code had to be structured to accept a wide range of possible user inputs. Different numbers of inputs, hidden layers, nodes per layer, and learning rates all had to be accounted for. The key to keeping track of all of the changing parameters was writing descriptive variable names and isolating important pieces of the network into their own functions for easy debugging. This was a challenging and interesting project with a fun competition component that greatly expanded our knowledge of neural network training.

Appendix

1. MATLAB Code of Backpropagation Functions

3 Layer with Softmax Transfer Function on Output Layer

```
function [W1, b1, W2, b2, W3, b3, mseValues, accValues] = bP3Lsoftmax(trainInputs, trainTargets,
learningRate, iterations, testInputs, testTargets, testLabels)
%   ------- BACKPROPAGATION FUNCTION -------
%
%   INPUTS:
%   trainInputs = input vector for training set
%   trainTargets = target vector for training set
%   learning rate = alpha learning rate
%   iterations = set value for maximum iterations
%   architecture = parameter not implemented (for different architectures)

%   transfer function definitions (HARDCODED):
%   layer1 function: log-sigmoid
%   layer2 function: log-sigmoid
%   layer3 function: softmax()

%   OUTPUTS:
%   W1 = updated weight matrix for layer 1
%   b1 = updated bias vector for layer 1
%   W2 = updated weight matrix for layer 2
%   b2 = updated bias vector for layer 2
%   W3 = updated weight matrix for layer 3
%   b3 = updated bias vector for layer 3
%   mseValues = vector of average MSE for each epoch
%   accValues = vector holding periodic validation test accuracy results

% HIDDEN LAYER SIZE HARDCODED HERE
hiddenLayer1 = 80;
hiddenLayer2 = 20;


% get size of target space to determine output layer size
[targRows targCols] = size(trainTargets);
outputRows = targRows;

%   Initial Weights and Biases created using small random values
[trainRows trainCols] = size(trainInputs);

% ---- Layer 1 ----- %
W1 = zeros(hiddenLayer1, trainRows); %creates empty weight matrix
b1 = rand(hiddenLayer1, 1);
for m = 1:hiddenLayer1
        for n = 1:trainRows
        x = randi(2);
        if(x == 2)
```

```
             x = -1;
             end
             W1(m,n) = x * rand(1)/10;
    end
end

% ---- Layer 2 ----- %
%[w1rows w1cols] = size(W1);
W2 = zeros(hiddenLayer2, hiddenLayer1);
b2 = rand(hiddenLayer2, 1);

for m = 1:hiddenLayer2
             for n = 1:hiddenLayer1
             x = randi(2);
        if(x == 2)
             x = -1;
        end
        W2(m,n) = x * rand(1)/10;
        %b2(m) = rand(1)/10;
             end
end

% ---- Layer 3 ----- %
%[w2rows w2cols] = size(W2);
W3 = zeros(outputRows, hiddenLayer2);
b3 = rand(outputRows, 1);

for m = 1:outputRows
             for n = 1:hiddenLayer2
             x = randi(2);
        if(x == 2)
              x = -1;
        end
        W3(m,n) = x * rand(1)/10;
        %b2(m) = rand(1)/10;
             end
end


alpha = learningRate;
mseIter = 1;       % just a start value for the while loop
iters = 1;      % iteration counter
mseValues = zeros(iterations, 1);   %initialize MSE vector
accValues = zeros(max((iterations/50),1), 1);

% outside loop controls the training iterations
while( mseIter > 0.005 && iters < iterations + 1)

             mseIter = 0; % cumulative MSE variable for training iteration

             % for each input and target in the training sets:
             for passes = 1:trainCols

        % get input and target from their matrices
        input = trainInputs(:,passes);
```

```
target = trainTargets(:,passes);

%------ Now Propagate Forwards ------%
a1 = logSigmoid((W1 * input) + b1);

a2 = logSigmoid((W2 * a1) + b2);

a3 = softmax((W3 * a2) + b3);

error = target - a3; % error vector
mseIter = mseIter + mse(target, a3); % add this pair's MSE

%------ Now Calculate Sensitivities and Backpropagate ------%

F2n3 = zeros(outputRows, outputRows);
for i = 1:outputRows
        for j = 1:outputRows
        if i == j
        F2n3(i,j) = (1 - a3(i,1))*(a3(i,1));
        else
          F2n3(i,j) = -a3(i,1) * a3(j,1);
        end
        end
end


% create and populate the f2(n2) derivative matrix
F2n2 = zeros(hiddenLayer2, hiddenLayer2);
for i = 1:hiddenLayer2
        for j = 1:hiddenLayer2
        if i == j
        F2n2(i,j) = (1 - a2(i,1))*(a2(i,1));
        end
        end
end

% create and populate the f1(n1) derivative matrix
F2n1 = zeros(hiddenLayer1, hiddenLayer1);
      [f1rows, f1cols] = size(F2n1);
for i = 1:f1rows
        for j = 1:f1cols
        if i == j
        F2n1(i,j) = (1 - a1(i,1))*(a1(i,1));
        end
        end
end

% calculate first sensitivity s^M
s3 = -2 * F2n3 * error;  % creates vector sized outputRows x 1

% calculate first sensitivity s^M-1
s2 = F2n2 * W3' * s3;  % creates vector sized outputRows x 1

% calculate next layer's sensitivity s^M-2
s1 = F2n1 * W2' * s2; %creates vector hiddenLayer x 1 sized
```

```
        %------ Update Weights and Biases ------%

        W3 = W3 - (alpha * s3 * a2');
        b3 = b3 - (alpha * s3);

        W2 = W2 - (alpha * s2 * a1');
        b2 = b2 - (alpha * s2);

        W1 = W1 - (alpha * s1 * input');
        b1 = b1 - (alpha * s1);
             end
             mseIter = mseIter/targCols; % get the average for the epoch
             mseValues(iters, 1) = mseIter; % save it to output array
             iters = iters + 1; % update iters for while loop control
             x = mod(iters,50);
             if(x == 0)
        accValues(iters/50,1) = determineAccuracy((validTest3LaySM(testInputs, testTargets, W1, b1, W2, b2,
W3, b3)), testLabels)
             end
end

end
```

Additional Mini Project Code (such as other backpropagation algorithm versions, helper functions, etc.) can be found at my github repository:
https://github.com/margaret3991/neural-networks-MATLAB/tree/master/miniproject