Homework 5: MNIST Handwriting Recognition
AMATH 482: Computational Methods for Data Analysis, Winter 2017

Sean Lam
seanlam@uw.edu

March 16, 2017

Abstract

This report investigates the use of an artificial neural network for recognizing handwritten numbers from the Mixed National Institute of Standards and Technology (MNIST) database. Perceptrons were used to identify the digits. Different activation functions, different numbers of layers and neurons per layer, and different learning rates were explored.

I. Introduction and Overview

Artificial neural networks are fascinating mathematical models for machine learning. They are inspired by the brains of humans and animals: small processing units called neurons are connected together to form a complex neural network that is capable of learning and adaptation. The idea of a neural network dates back to the 1940s, but progress has been slow until the past decade, where there has been an explosion of available data to be able to train these neural networks on a much larger scale.

This project will solve one of the most basic machine learning problems: classifying handwritten single digit numbers. The dataset was provided by the Mixed National Institute of Standards and Technology (MNIST) and contains a labeled training set of 60,000 images and a labeled test set of 10,000 images. Perceptrons were 10×1 column vectors that identifies each of the digits. This project explored different activation functions, with different combinations of layers and neurons per layer, and different learning rates.

II. Theoretical Background

Deep neural networks (DNNs) are basically a huge optimization problem. To train a single layer neural network, we simply need to feed data ($\mathbf{X}$) through the input layer, and provide the labels for the data ($\mathbf{Y}$). Training a neural network is about solving the optimization problem $\mathbf{WX} = \mathbf{Y}$, where $\mathbf{W}$ are the weightings of neurons. Then, to classify new data, the neural network can receive a new input $\mathbf{X}$, and perform $\mathbf{WX} = \mathbf{Y}$ to solve for the most likely label ($\mathbf{Y}$) of input $\mathbf{X}$. Deep neural networks are fundamentally the same as neural networks except they have hidden layers in between the input and output layers, creating a more difficult optimization problem.

Figure 1 shows a visual representation of a 3-layer DNN. We have to go from one layer to another to get from the input to the output. Starting from the input layer X, we solve $\mathbf{W_1 X} = \mathbf{Y_1}$, where $\mathbf{Y_1}$ is the first hidden layer, then we solve $\mathbf{W_2 Y_1} = \mathbf{Y_2}$, where $\mathbf{Y_2}$ is the second hidden layer, then finally $\mathbf{W_3 Y_2} = \mathbf{Y_3}$, where $\mathbf{Y_3}$ is the output layer.

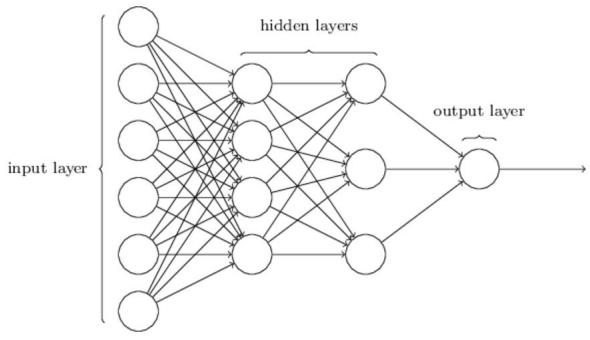

Figure 1: This is an example of a deep neural network with 3 layers (2 hidden layers and 1 output layer).

The optimization problem is solved using a method similar to Newton's method for finding zeros. It starts with an initial guess at the **W** matrices (weightings), then takes an input and solve for the output. It then calculates the output error, and then back propagates the error through every layer back to the input layer, and then changes the weightings based on the gradient descent of the error. Then the iterative process repeats for the desired number of epochs/iterations.

The least squares error is defined as the loss function

$$E = \frac{1}{2}\sum_j (\mathbf{y}_j - \mathbf{a}_j^L)^2 \tag{1}$$

where j are the neurons in the L layer (output layer), and y is the desired output, and a is the output activations.

Our goal is to create an algorithm that will minimize this error. We have to first understand how changing the weights in a network affects the error function.

The error δ_j^l of neuron j in layer l is defined as

$$\delta_j^l = \frac{\partial E}{\partial \mathbf{a}_j^l} \tag{2}$$

The error in the output layer δ_j^L can be written as

$$\delta_j^L = \frac{\partial E}{\partial \mathbf{a}_j^L} f'(\mathbf{a}_j^L) \tag{3}$$

where f' is the derivative of the activation function.

The activation function is the function of the activation of the neuron input from one layer to the next layer. The most basic activation function is linear, which would just be

$$\mathbf{a}^l = f(\mathbf{w}^l \mathbf{a}^{l-1}) = \mathbf{w}^l \mathbf{a}^{l-1} \tag{4}$$

However, in most cases we only want outputs from 0 to 1, so we use activation functions that constrain the outputs. Two examples of these functions are the sigmoidal function

$$\mathbf{a}^l = f(\mathbf{w}^l \mathbf{a}^{l-1}) = \frac{1}{1 + e^{-\mathbf{w}^l \mathbf{a}^{l-1}}} \tag{5}$$

and the hyperbolic tangent function

$$\mathbf{a}^l = f(\mathbf{w}^l \mathbf{a}^{l-1}) = tanh(\mathbf{w}^l \mathbf{a}^{l-1}) \tag{6}$$

$\frac{\partial E}{\partial \mathbf{a}_j^L}$ measures how fast the error function is changing as a function of the j^{th} output activation.

$f'(\mathbf{a}_j^l)$ measures how fast the activation function f is changing at $\mathbf{a}_j^l$.

Rewriting equation 3, which is in component form, to matrix form, is

$$\delta^L = \nabla_{\mathbf{a}} \mathbf{E} \cdot f'(\mathbf{a}^L) \tag{7}$$

The gradient of the error function $\nabla_{\mathbf{a}} \mathbf{E}$ is simply

$$\nabla_{\mathbf{a}} \mathbf{E} = \mathbf{a}^L - \mathbf{y} \tag{8}$$

Next, the equation for the error $\delta_{\mathbf{l}}$ in terms of the error in the next layer $\delta_{\mathbf{l+1}}$ is

$$\delta_l = ((\mathbf{w}^{l+1})^T \delta^{l+1}) \cdot f'(\mathbf{a}^l) \tag{9}$$

where w is the weight matrix for a given layer.

The last step is to change the weightings matrices by "taking a step" in the direction of the gradient descent of the error function with respect to the weightings.

The equation for the rate of change of the error function with respect to any weight in the network is

$$\frac{\partial E}{\partial \mathbf{w}^l} = \mathbf{a}^{l-1} \delta^l \tag{10}$$

where $\mathbf{a}^{l-1}$ is the activation of the neuron input to the weight $\mathbf{w}^l$, and δ^l is the error of the neuron output from the weight $\mathbf{w}^l$.

The step size is denoted as τ. The adjusted weightings are

Figure 2: These are eight sample images from the MNIST data set.

$$\mathbf{W}_l = \mathbf{W}_l - \tau(\mathbf{a^{l-1}}\delta^l) \tag{11}$$

Then this iterative process repeats until the weightings matrices converges to a solution.

III. Algorithm Implementation and Development

The first step is to load the files from the MNIST database using helper functions that Stanford has provided. The loaded matrices are the training set ($784 \times 60,000$), the training labels ($60,000 \times 1$), the test set ($784 \times 10,000$), and the test labels ($10,000 \times 1$). Figure 2 shows eight sample images from the MNIST data set.

The next step is the convert the labels into perceptrons for the neural network. The perceptrons will be 10×1 column vectors, that has a 1 on the (j+1)th row, where j is the label, and everything else 0. For example, a label of 3 will have the perceptron

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{12}$$

Then the initial weights are created, consisting of random numbers between -1 and 1. The size of the weight matrix depends on the size of the number of neurons in each layer. The first weight matrix will have dimensions $n \times 784$, where n is the number of neurons in the first layer, and 784 being the number of pixels in each image. The next weight matrices will have dimensions $n \times m$, where n is the number of neurons in the following layer, and m is the number of neurons in the previous layer.

Then we feed each image input forward into the neural network, producing the corresponding outputs for each layer using the activation function, which are equations 4, 5, or 6 depending on which activation function was used. Once the feedforward is complete, we back propagate to compute the errors in each layer using equation 7 for the output layer and equation 9 for the hidden layers. The next step is then to compute the gradient descent of the error function with respect to the weights in the network for each layer using equation 10. The last step of the algorithm is for the adjust the current weightings by taking a step in the gradient descent to get closer to the optimal weightings using equation 11. Then, we feed in the next image and repeat the process until the weightings are optimized. Because this algorithm is very computationally heavy, I will only be running this for at most three epochs (three runs through all the images).

IV. Computational Results

The first thing I computed was solving the $\mathbf{WX} = \mathbf{Y}$ problem using the pseudoinverse (least squares regression).

$$\mathbf{W} = \mathbf{Y_{train}}\mathbf{X_{train}^{\dagger}} \tag{13}$$

When I use these weightings I found from equation 13 to test my training set, I come out with a 100% accuracy, which makes sense because this overfits the training set. Cross validation is the most important thing for machine learning; it is important that the results work for all generic data, not just overfitted for the training set. Testing the same weightings on my testing set, I get 85% accuracy, which is high, but could be much better by using a deep neural network.

Figure 3 shows the results of a one layer neural network, using a learning rate of 0.1. The linear activation function (Figure 3a) already reached above 80% by the 2000th image, and then just continued to oscillate throughout the three epochs. I think this is because we have such a huge training set, that all the weightings became very big, and as a result

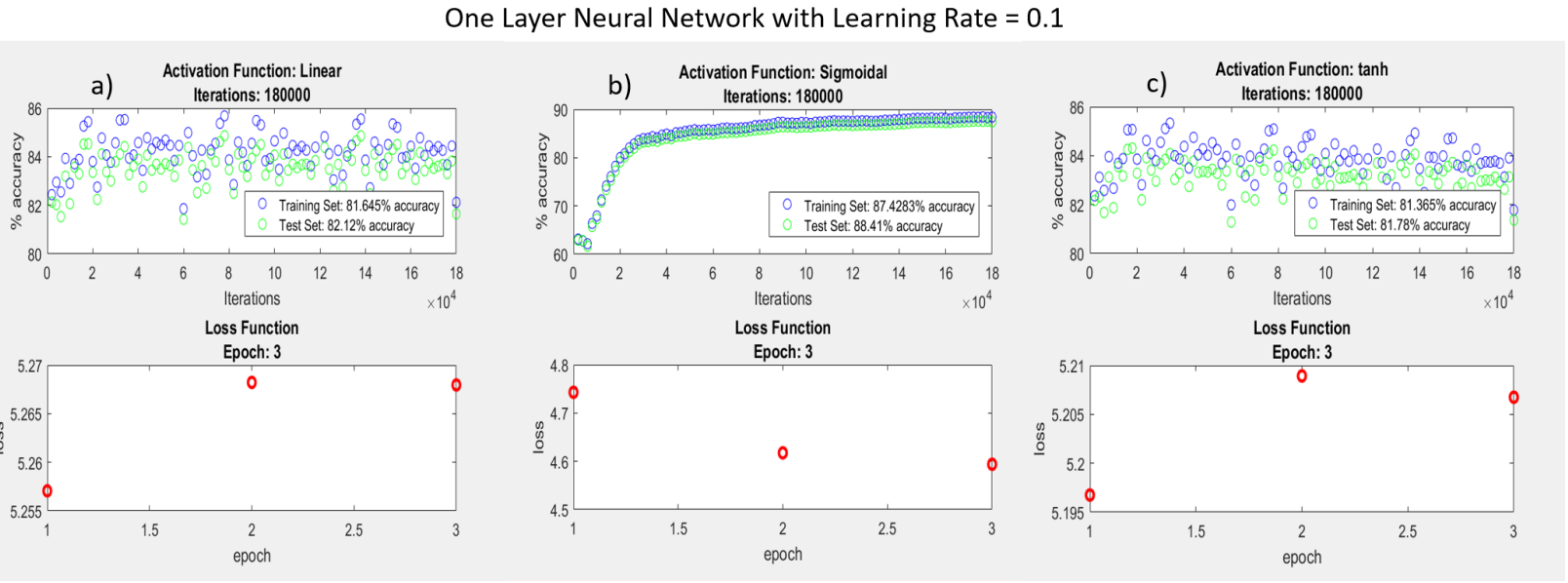

Figure 3: These are the results of the one layer neural network with a learning rate of 0.1. The top plots are the accuracies every 2000 iterations. The bottom plots are the losses per epoch. a) shows the results with a linear activation function. b) shows the results with a sigmoidal function. c) shows the results with a hyperbolic tangent function.

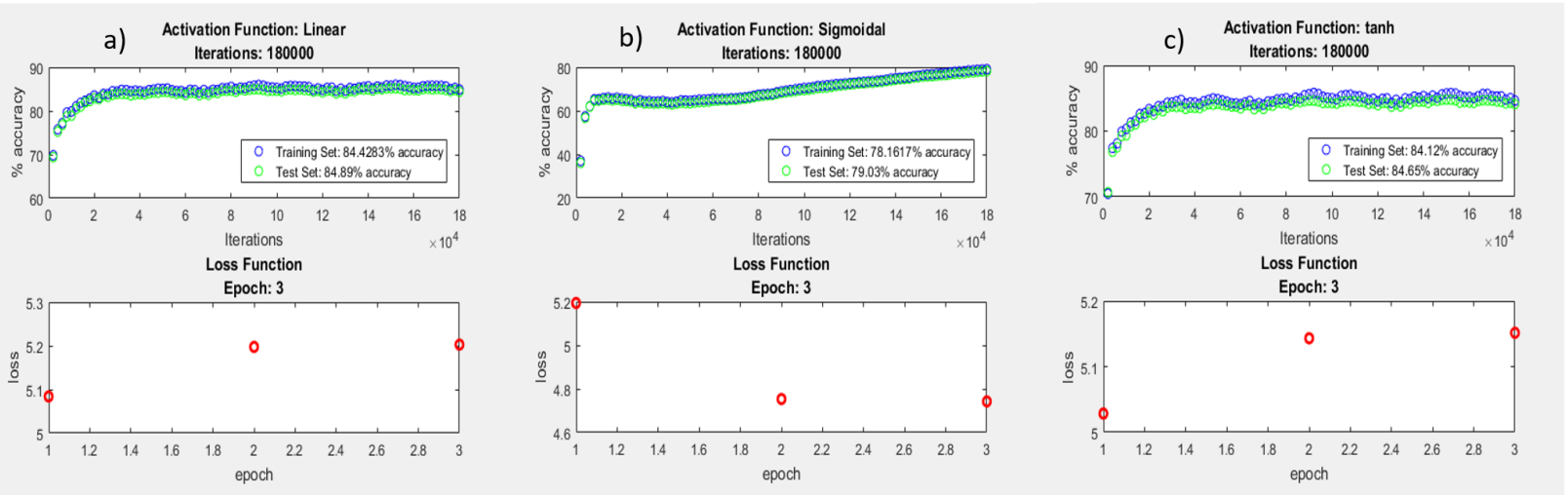

Figure 4: These are the results of the one layer neural network with a learning rate of 0.01. The top plots are the accuracies every 2000 iterations. The bottom plots are the losses per epoch. a) shows the results with a linear activation function. b) shows the results with a sigmoidal function. c) shows the results with a hyperbolic tangent function.

of that it keeps having to correct itself, (i.e. the first weighting might be 1000 so then the second has to be -1000 to make the output perceptron near a value of 1). The hyperbolic tangent function as shown on Figure 3c has a similar behavior, reaching a high accuracy really fast, and then oscillating around it. The loss functions are also similar, they are around 5.2 and barely change per epoch. The sigmoidal function, however, produced a very different result. It started off poorly, but continued to increase past the linear and tanh accuracies as it kept iterating. The loss function is actually decreasing for the sigmoidal function, and given more epochs, it will most likely continue to decrease. It can also be noticed that the test set accuracies are a little bit higher than the training set accuracies, which means we are not overfitting the data.

Figure 4 shows the results of a one layer neural network with a lower learning rate of 0.01. The results are similar to the neural network in Figure 3, except the sigmoidal function increased at a slower pace. Figure 5 shows the results of a one layer neural network with a higher learning rate of 1. These results are different than the other one layer neural

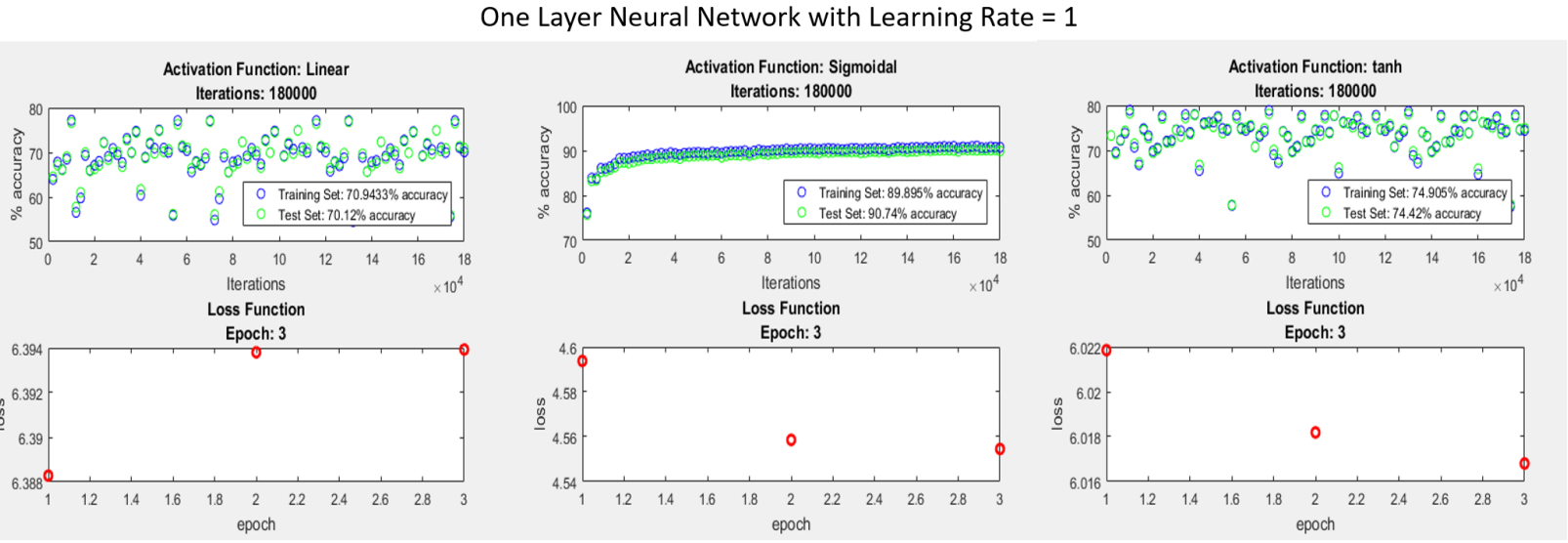

Figure 5: These are the results of the one layer neural network with a learning rate of 1. The top plots are the accuracies every 2000 iterations. The bottom plots are the losses per epoch. a) shows the results with a linear activation function. b) shows the results with a sigmoidal function. c) shows the results with a hyperbolic tangent function.

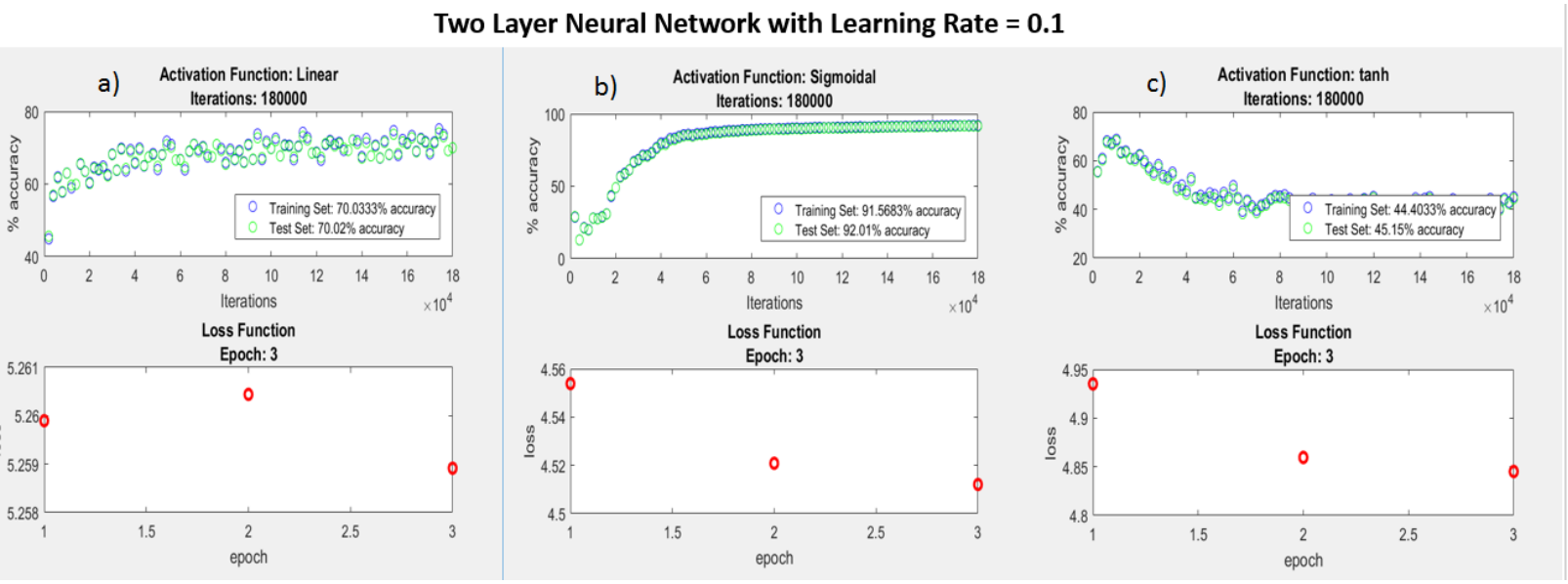

Figure 6: These are the results of the two layer neural network with a learning rate of 0.1. The hidden layer had 300 neurons. The top plots are the accuracies every 2000 iterations. The bottom plots are the losses per epoch. a) shows the results with a linear activation function. b) shows the results with a sigmoidal function. c) shows the results with a hyperbolic tangent function.

networks; the linear function was only able to get 70% accuracy, and the hyperbolic tangent function was only able to get 74% accuracy. However, the sigmoidal function was able to hit above 90% accuracy. with three epochs.

Figure 6 shows the results of a two layer neural network with a learning rate of 0.1, and 300 neurons in the hidden layer. Figure 6a are the results for the linear activation function; it actually has worse accuracy than the one layer neural networks. However, from the trend of the accuracy plot, it seems like the accuracy was still steadily increasing when three epochs were reached. The loss function is pretty much the same for all three epochs for the linear function. Figure 6b are the results for the sigmoidal activation function. Like the one layer network, the sigmoidal function gives the most accurate results. It started off poorly, but continued to increase to over 90% over three epochs. The loss function did not change as dramatically as the one layer network, which means that the sigmoidal function was able to reach a high accuracy before the first epoch was complete, and the second and third epochs still increased the accuracy, but very slowly. The tanh function results are shown on Figure 6c. The results are interesting because the accuracy was less than

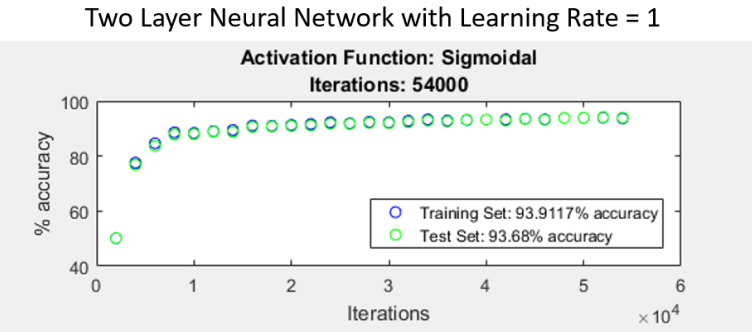

Figure 7: These are the results of the two layer neural network with a learning rate of 0.1. The hidden layer had 300 neurons. The top plots are the accuracies every 2000 iterations. The bottom plots are the losses per epoch. a) shows the results with a linear activation function. b) shows the results with a sigmoidal function. c) shows the results with a hyperbolic tangent function.

50%, but the loss function was still lower than the linear function, which had a 70% accuracy. In the beginning, the tanh function was increasing until about 10,000 iterations, when it started to decay all the way down to around 45%. I do not know exactly why this happened, but it possibly reached a local minimum on the gradient descent and got stuck down there, because the hyperbolic tangent function is very flat around its edges.

Since the results from the one layer neural network showed that the sigmoidal function with a learning rate of 1 was the most accurate, I decided to run those same parameters but on a two layer neural network, with 300 neurons in the hidden layer. The result, shown in Figure 6, was a 93.68% accuracy on the test set from only 54,000 iterations, before it even reached one full epoch. It is also interesting how only after 50,000 iterations, the training accuracy started becoming higher than the test accuracy, which means it started to overfit.

V. Summary and Conclusions

In this report, I investigated the use of a deep neural networks to solve a common problem in machine learning: handwriting recognition. I experimented with one layer and two layer networks, different activation functions: sigmoidal, hyperbolic tangent, and linear, and different learning rates: 0.01, 0.1, and 1. These neural networks ran for a maximum of three epochs. The accuracies for the one layer network were pretty consistent, ranging from 70% to 90%, but are not sufficiently accurate enough. The sigmoidal function worked well with a higher learning rate, but the linear and tanh functions worked better with lower learning rates. The two layer neural networks performed worse with the linear function and especially the tanh function, which probably got stuck at a local minimum, but performed average with the sigmoidal function at a learning rate of 0.1. The last test with the two layer sigmoidal function produced the highest accuracy, at 93.68% at only 54,000 iterations (90% of an epoch).

Handwriting recognition is only the most basic of machine learning problems. The same concepts and algorithms implemented in this project can be used for greater projects in machine learning such as computer vision or natural language processing. Machine learning is a relatively new field; even professionals do not completely understand how deep learning works, and it is still really slow because it takes a massive training set in order for the accuracies to be respectable, whereas our brains are able to recognize things much faster and with less training data. Once deep learning algorithms are able to be computed in real-time and still be robust, it will open up a whole new spectrum of possibilities in every technical field.

Appendix A: MATLAB functions used and brief implementation explanation

loadMNISTimages.m - helper function from Stanford to load MNIST images
loadMNISTlabels.m - helper function from Stanford for load MNIST labels
pinv(A) - calculates the pseudoinverse of the given matrix A
rand(x,y) - creates a matrix of size $x \times y$ of random numbers from 0 to 1
sigmf(A, [b c]) - performs the sigmoid function on matrix A, with shape and position parameters b and c
tanh(A) - performs the hyperbolic tangent function on matrix A

Appendix B: MATLAB codes

Main.m

```
clear all; clc; close all;
train_set = loadMNISTImages('train-images.idx3-ubyte');
train_label = loadMNISTLabels('train-labels.idx1-ubyte');
test_set = loadMNISTImages('t10k-images.idx3-ubyte');
test_label = loadMNISTLabels('t10k-labels.idx1-ubyte');


%% 1 layer
close all;
for j = 1:3
    figure(j)
    numLayers = 1;
    numNeurons = [10];
    tau = 1;
    max_epochs = 3;
    % activation: 1 = sigmoidal, 2 = tanh, 3 = linear
    activation = j;
    [accuracy, error] = neocognitron(train_set, train_label, test_set, test_label, ...
        numLayers, numNeurons, tau, max_epochs, activation);
end

%% 2 layer
close all;
for j = 1:3
    figure(j)
    numLayers = 2;
    numNeurons = [300 10];
    tau = 1;
    max_epochs = 3;
    % activation: 1 = sigmoidal, 2 = tanh, 3 = linear
    activation = j;
    [accuracy, error] = neocognitron(train_set, train_label, test_set, test_label, ...
        numLayers, numNeurons, tau, max_epochs, activation);
end
```

neocognitron.m

```
function [accuracy,error] = neocognitron(train_set, train_label, test_set, test_label, numLayers, numNeurons
% The most awesome deep neural network algorithm
% Inputs:
% train_set: training set
% train_label: training labels
% test_set: test set
% test_label: test labels
% numLayers: number of layers for deep neural network
% numNeurons: row vector of number of neurons for the corresponding layer
% tau: learning rate
% max_epoch: how many epochs to run
% activation: which activation function to use
% 1 = sigmoidal, 2 = tanh

% Outputs:
% accuracy when tested on test set

    % Initialize Data
    [pixels, n_train] = size(train_set); %training size and pixel count
```

```
[~, n_test] = size(test_set); %testing size
%perceptrons (index 1 is 0, index 10 is 9)
train_perceptron = makePerceptron(train_label);
test_perceptron = makePerceptron(test_label);
%random weights -1 to 1
A = cell(1,numLayers);
for j = 1:numLayers
    if (j == 1)
       A{j} = 2 * rand(numNeurons(j), pixels) -1;
    else
       A{j} = 2 * rand(numNeurons(j),numNeurons(j-1))-1;
    end
end
output = cell(1,numLayers);
delta = cell(1,numLayers);
activationFunction = {'Sigmoidal', 'tanh', 'Linear'};
%figure(1)
% TRAINING
for j = 1:max_epochs
   loss = zeros(1, n_train);
   disp([num2str(j), ' epoch']);
   for k = 1:n_train
        input = train_set(:,k); %initial input
        if (activation == 1)
            %forward
            output{1} = sigmf(A{1} * input,[0.01 0]); %first layer
            for l = 2:numLayers %all other layers
                output{l} = sigmf(A{l} * output{l-1},[0.01 0]);
            end
            %back propagation (calculate errors)
            delta{end} = (output{end} - train_perceptron(:,k)).*output{end}.*(1-output{end});
            for l = numLayers-1:-1:1
                delta{l} = (A{l+1}' * delta{l+1}).*output{l}.*(1-output{l});
            end
        elseif (activation == 2)
            output{1} = tanh(0.01 * A{1} * input);
            for l = 2:numLayers
                output{l} = tanh(0.01 * A{l} * output{l-1});
            end
            delta{end} = (output{end}-train_perceptron(:,k)).*(1-output{end}.^2);
            for l = numLayers-1:-1:1
               delta{l} = (1-output{l}.^2).*(A{l+1}'*delta{l+1});
            end
        elseif (activation == 3)
            output{1} = 0.01 * A{1} * input;
            for l = 2:numLayers
               output{l} = 0.01 * A{l} * output{l-1};
            end
            delta{end} = (output{end} - train_perceptron(:,k));
            for l = numLayers-1:-1:1
                delta{l} = A{l+1}'*delta{l+1};
            end
        end
        A{1} = A{1} - tau * (delta{1}*input');
        for l = 2:numLayers
           A{l} = A{l} - tau * (delta{l}*output{l-1}');
        end
        %INTERNAL TESTING
```

```
            loss(k) = 0.5 * (sum(abs(output{end} - test_perceptron(k))))^2;

            if (mod(k,2000) == 0)
                iteration = (j-1)*n_train+k;
                [accuracytest, errortest] = crossValidate(test_set, test_perceptron...
                    ,numLayers, A, output);
                subplot(2,1,1)
                plot(iteration,accuracytest,'bo')
                hold on
                [accuracytrain, errortrain] = crossValidate(train_set, train_perceptron...
                    ,numLayers, A, output);
                plot(iteration, accuracytrain, 'go')
                hold on
                title({['Activation Function: ', activationFunction{activation}];...
                    ['Iterations: ', num2str(iteration)]})
                legend(['Training Set: ', num2str(accuracytrain), '% accuracy'],...
                   ['Test Set: ', num2str(accuracytest),'% accuracy'],'Location','southeast')
                xlabel('Iterations')
                ylabel('% accuracy')
                pause(0.001)
            end
        end
        E = sum(loss) / n_train;
        subplot(2,1,2)
        plot(j,E,'r-o','Linewidth',[2])
        title({['Loss Function'];['Epoch: ', num2str(j)]})
        xlabel('epoch')
        ylabel('loss')
        hold on
    end

    % TESTING
    [accuracy, error] = crossValidate(test_set, test_perceptron...
        ,numLayers, A, output);
end
```

loadMNISTimages.m

```
function images = loadMNISTImages(filename)
%loadMNISTImages returns a 28x28x[number of MNIST images] matrix containing
%the raw MNIST images

fp = fopen(filename, 'rb');
assert(fp ~= -1, ['Could not open ', filename, '']);

magic = fread(fp, 1, 'int32', 0, 'ieee-be');
assert(magic == 2051, ['Bad magic number in ', filename, '']);

numImages = fread(fp, 1, 'int32', 0, 'ieee-be');
numRows = fread(fp, 1, 'int32', 0, 'ieee-be');
numCols = fread(fp, 1, 'int32', 0, 'ieee-be');

images = fread(fp, inf, 'unsigned char');
images = reshape(images, numCols, numRows, numImages);
images = permute(images,[2 1 3]);

fclose(fp);

% Reshape to #pixels x #examples
```

```
images = reshape(images, size(images, 1) * size(images, 2), size(images, 3));
% Convert to double and rescale to [0,1]
images = double(images) / 255;

end
```

loadMNISTlabels.m

```
function labels = loadMNISTLabels(filename)
%loadMNISTLabels returns a [number of MNIST images]x1 matrix containing
%the labels for the MNIST images

fp = fopen(filename, 'rb');
assert(fp ~= -1, ['Could not open ', filename, '']);

magic = fread(fp, 1, 'int32', 0, 'ieee-be');
assert(magic == 2049, ['Bad magic number in ', filename, '']);

numLabels = fread(fp, 1, 'int32', 0, 'ieee-be');

labels = fread(fp, inf, 'unsigned char');

assert(size(labels,1) == numLabels, 'Mismatch in label count');

fclose(fp);

end
```

makePerceptron.m

```
function perceptron = makePerceptron(labels)
% Inputs:
% training/testing labels

%Outputs:
% respective perceptron
    n = length(labels);
    perceptron = zeros(10, n);
    for j = 1:n
       label = labels(j);
       perceptron(label + 1, j) = 1;
    end
end
```

crossValidate.m

```
function [accuracy, error] = crossValidate(test_set, test_perceptron, numLayers, A, output)
    correct = 0;
    n_test = length(test_perceptron);
    for m = 1:n_test
        input = test_set(:,m);
        output{1} = sigmf(A{1} * input,[0.01 0]); %first layer
        for l = 2:numLayers %all other layers
            output{l} = sigmf(A{2} * output{l-1},[0.01 0]);
        end
        [~,testResult] = max(output{end});
        [~,label] = max(test_perceptron(:,m));
        if (testResult == label)
            correct = correct + 1;
        end
```

```
    end
    accuracy = correct/n_test * 100;
    error = 100 - accuracy;
end
```

pinvSolve.m

```
% Using AX = Y
xdagger = pinv(train_set);
train_perceptron = makePerceptron(train_label);
A = train_perceptron * xdagger;

%training data
yTrain = A * train_set;
%test data
yTest = A * test_set;
test_perceptron = makePerceptron(test_label);

%% training percentages
wrong = 0;
[~,numLabels] = size(train_label);
for j = 1:numLabels
   [~,correctLabel] = max(train_perceptron(:,j));
   [~,guessLabel] = max(yTrain(:,j));
   if (correctLabel ~= guessLabel)
       wrong = wrong + 1;
   end
end
accuracy = (1 - wrong / numLabels) * 100

%% testing percentages
wrong = 0;
[~,numLabels] = size(test_perceptron);
for j = 1:numLabels
   [~,correctLabel] = max(test_perceptron(:,j));
   [~,guessLabel] = max(yTest(:,j));
   if (correctLabel ~= guessLabel)
       wrong = wrong + 1;
   end
end
accuracy = (1 - wrong / numLabels) * 100
```