Backpropagation with MNIST Dataset

DJ Wadhwa

Table of Contents

Introduction 2

Methods 3

- Weight and Bias Initialization 4
- Feedforward, Backpropagation, and Updating Weights and Biases 5
- Noise Induced Error and Test Set Accuracy 6

Results 8

- Mean Squared Error with 3 digits 8
- Error with respect to noise added to image 8
- Mean Squared Error with MNIST data with different epochs 9
- MNIST Test Set Accuracy for every Test Case 10

Conclusion 11

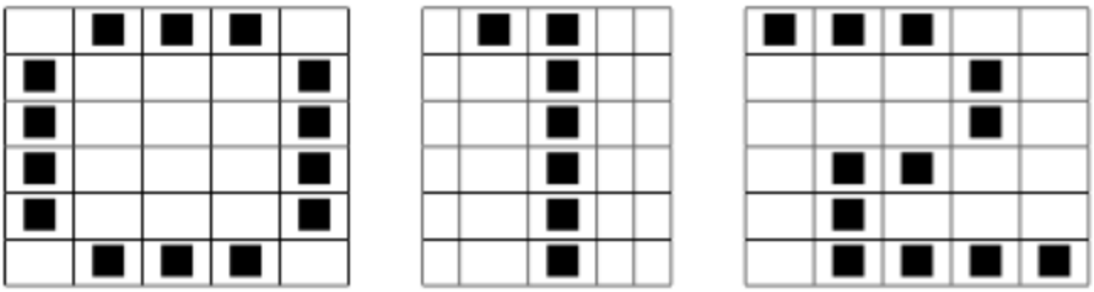
Appendix 12

- Weight and bias generation 12
- Backpropagation 12
- Add noise 13
- Calculate error from noise 14
- Graph all the results 14

Introduction

In performance learning by using ANN, there are many approaches to train a network. Previously, we discussed, demonstrated, and showed the result of training a single-layer network by using the perceptron learning rule. However, as the algorithm is not able to solve linearly separable classification problems, we are migrating to multilayer networks. In order to train these multilayer networks, we decided to use backpropagation.

Backpropagation is a general algorithm that can be used to train a multilayer network with non-linear transfer functions. The algorithm can be described as an approximate steepest descent algorithm, having its performance index equal to mean square error. Backpropagation uses the chain rule of calculus to calculate the derivatives, which explains the complexity that lies in the relationship between the network weights and error.



In this assignment, we are using the backpropagation algorithm to train a multilayer network (to be precise, a network with 4 layer total) to be able to solve pattern recognition problems which is similar to the ones that we did with Hebbian Learning.

The following picture is the pattern that we are trying to recognize through our trained multilayer ANN. Those digit patterns will be the inputs to our network. By using the previously-mentioned inputs, the network will have three outputs that should have similar output as the one described by t0, t1, and t2.

$$\mathbf{t}_0 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \quad \mathbf{t}_1 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \quad \mathbf{t}_2 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

By using the previous input patterns, we are trying to prove the accuracy of the backpropagation algorithm in solving pattern recognition related problems. Since we have done this same problem by using Hebbian algorithm in a single layer network (hard-lim transfer function), we want to investigate and compare the result with using multilayer network (log-sigmoid transfer function). Since the textbook focuses on the performance learning done by a network, experimenting backpropagation algorithm in a multilayer ANN will prove this as a feasible solution in any pattern classification and recognition problems (or in a more general area of discussion, ANN-related problems).

Methods

The neural network with backpropagation was built with an understanding of how a perceptron was built and computed. The key difference between a perceptron based network and a backpropagating network is that a perceptron is made of a single layer and adjusting the weight and bias would simply be a matter of following the perceptron

learning rule. While this works great for simple pattern recognition, a more complicated, multi-layered artificial neural network is needed for anything more complicated. This also draws some remembrance to an actual brain that consists of several neurons that are connected together to perform a particular function. In this case, the function of our program is rather simple: to train an ANN with handwritten Indo-Arabic numerals (0-9) that are stored as 28x28 pixel images in a dataset of 60,000 images from the MNIST database. In order to do so, we first have to build a network, test its effectiveness, and understand how to change our building parameters to make the program more modular.

Weight and Bias Initialization

The first problem to tackle was to create initial weights and biases. We did so by first assigning random values to each element using the rand() function provided by MATLAB. This created a uniform distribution of random numbers between 0 and 1, and while this worked the algorithm took a very long time to converge. We then modified our weight generation function to use the randn() function, which would provide a normal distribution of random numbers between 0 and 1.

This was a slight improvement to the previous implementation, however, the initial weights and biases were still too large. We need a coefficient that would lower the value of randomly generated numbers. We first tried using 0.1 to reduce the range from 0 to 0.1, and while that did improve the network a bit it was still having issues with reducing the error quickly. I then turned to a paper from 2015 by Kaiming He, where it was suggested to initialize values of W with the coefficient of "$\sqrt{2}/\sqrt{n}$" where 'n' is the number of incoming connections coming into a given layer (https://arxiv.org/pdf/1502.01852.pdf, page 4). This significantly improved network

performance and required far fewer iterations to converge to a reasonable Error. The code for the weight and bias generation function is posted in the appendix.

Feedforward, Backpropagation, and Updating Weights and Biases

The weights (W), biases (b), neuron values (n), and activations (a) were stored in cell arrays. This allowed storing matrices of different lengths in the same variables. To calculate the neuron values for the first layer we used the classic perceptron equation W{1}*p+b{1}. The next few layers can be described as 'm' are computed as W{m}*a{m-1}+b{m}. The activation is then calculated by simply adding the logsig() function which is described as "1 / (1 + exp(-n))".

Following this, we need to calculate the sensitivities in each layer. This is done by first calculating the sensitivity in the last layer, which is done by finding the derivative of the error (t-a) and following the chain rule. The sensitivity for the last layer is calculated by using "S{m} = -2*diff_sig*(t-a{m})" where diff_sig is a Jacobian matrix of the partial derivative of a{m} with respect to each weight. This sensitivity is then propagated backward by using the equation "S{m} = diff_sig*W{m+1}'*S{m+1}" where 'W{m+1}'*S{m+1}' is the ratio of how much the sensitivity from the m+1 layer affects the sensitivity of the previous layer. The process is repeated till the sensitivity of each layer is propagated back to the first layer.

These sensitivities are then used to update the weights and biases for the next iteration. The equation used for this is "W{m} = W{m}-alpha*S{m}*a{m-1}'" where the alpha is the learning rate that is defined as a parameter to the backpropagation function. The sensitivity is multiplied by the activation of the previous layer, so the ratios as described above are then translated to real values. A similar equation is used to

calculate the new bias however it lacks “a{m-1}'” because the sensitivity for the bias is not a ratio but in fact the absolute value.

This marks one element from the training set. This needs to be repeated with every element of the training set. This draws parallels to the perceptron learning rule where each value of the training set is used to update the values of the weights and biases. Once the algorithm is done processing the error from every value of the training set it has completed one epoch. Many networks, including ours, need multiple epochs to perform well. The performance is measured by the Mean Squared Error (MSE) per epoch. If the MSE is decreasing, it indicates that the network is “learning”.

Noise Induced Error and Test Set Accuracy

Once the model was trained, we had to examine its accuracy. For part 1 of the problem, where we were given 3 digits (0, 1, and 2) and we added some noise by flipping pixels (0, 4, and 8) in the images. We reused he previously provided addNoise() function and made a modification where rather than the flipped bit being a -1 or 1, we changed it to 0 or 1. We then wrote the noise_error() function that tested the three different noise levels for each digit ten times and counted the number of times the digits from the network did not match the expected output. This gave us the error for each digit for each noise level. This and the MSE for this network were graphed by the graph_backprop() function.

To measure the accuracy of the network against the MNIST dataset we had additional testing data to compare against. We did so by having four different test cases which were simply different in the number of epochs that were used to train the model. We began loading the test data similar to how we loaded the initial training data. The

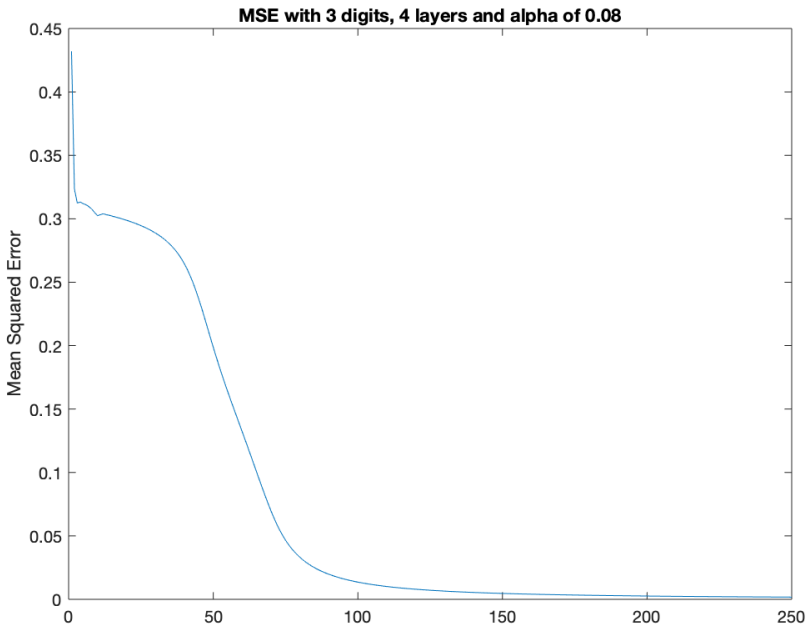
network then classified the test data and compared it against the correct labels for the test data. The total number of correct classifications were tallied and stored for each test case. This and the MSE for each test case were graphed by the graph_backprop() function.

The graphs are showcased under the Results section below, while the implementation is displayed in the Appendix.

Results

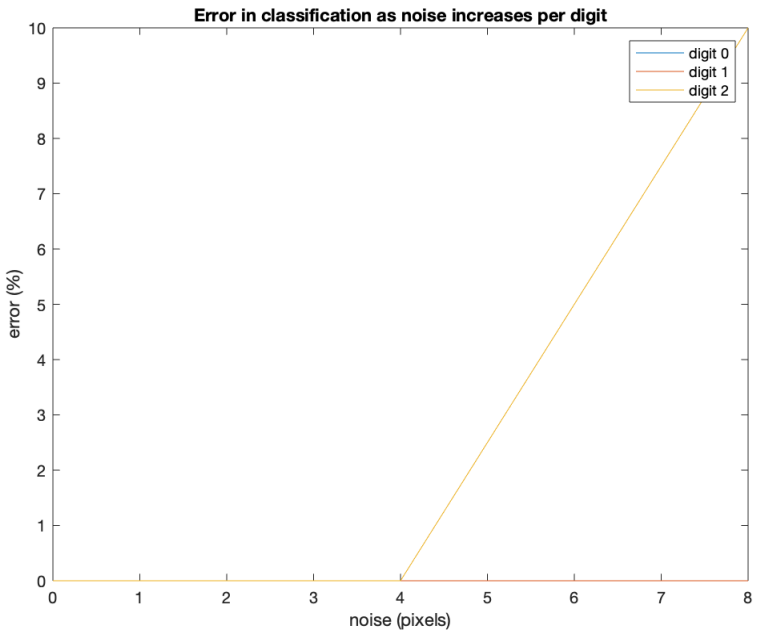
Mean Squared Error with 3 digits

As we can see the MSE of the network reduces as the number of the iterations increases. This level of performance is seen mostly due to the updating the weight initialization method.



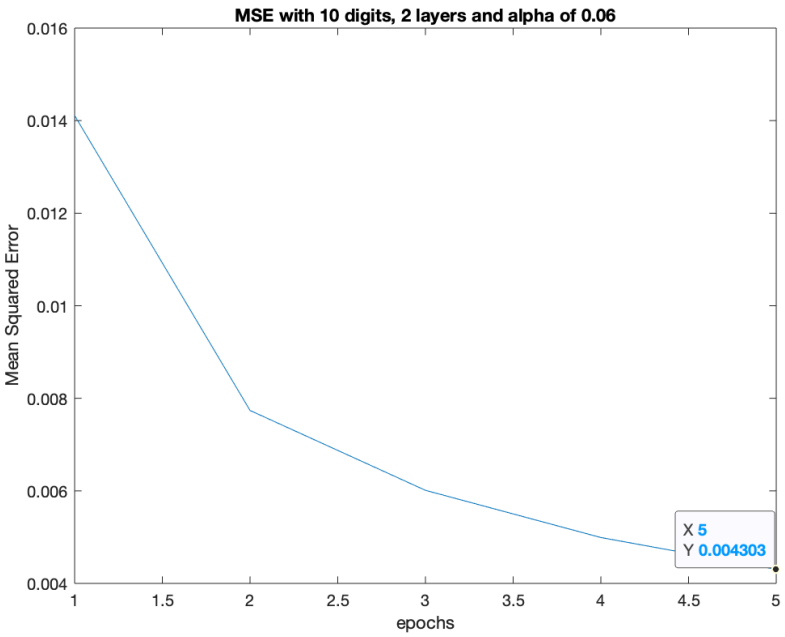
Error with respect to noise added to image

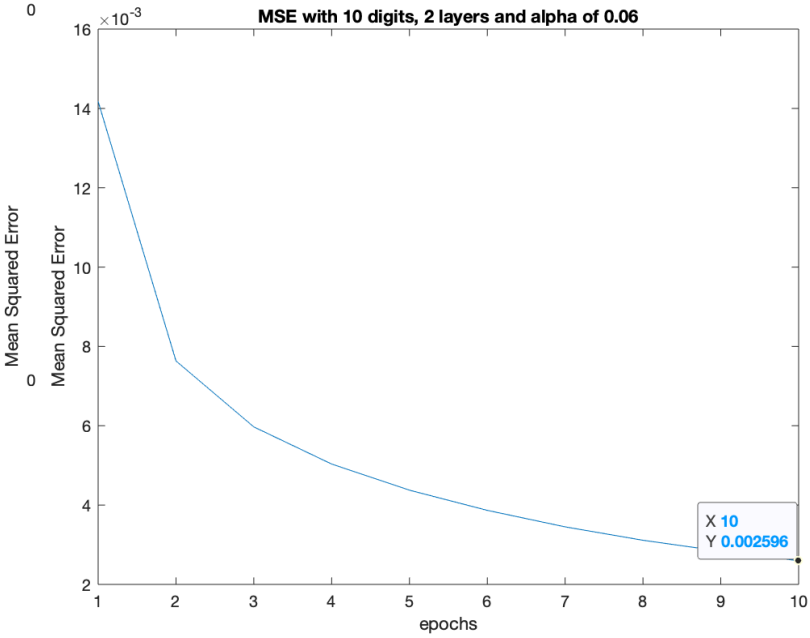
This graph shows that the error in classification using a neural network that employs backpropagation has an error of 10% for the most complicated digit (2) at noise of 8 digits, which is equates to 73% similarity to original data.



Mean Squared Error with MNIST data with different epochs

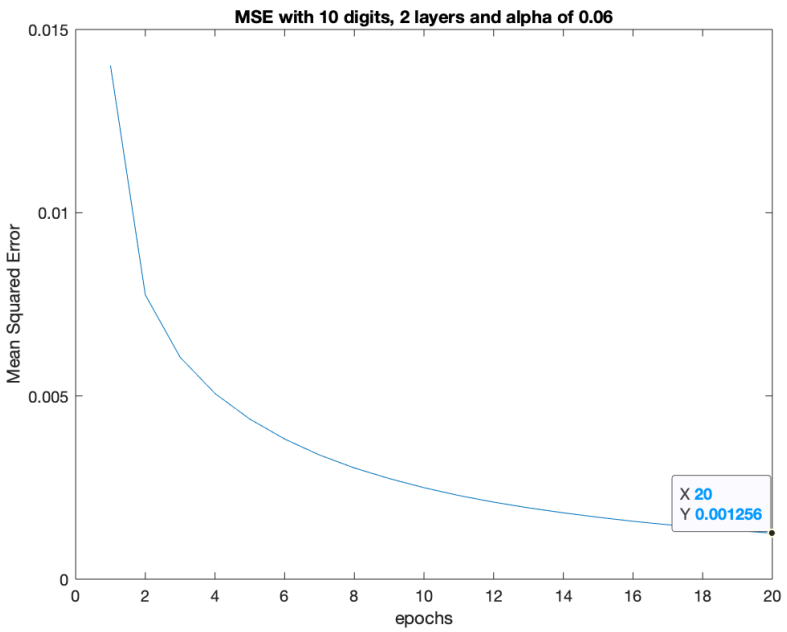
As we can see, as the number of epochs increases the MSE decreases. This is due to increasing the number of epochs a neural network is trained for improves its performance. That said there is a point of diminishing returns where increasing iterations will no longer decrease MSE or the relation may be asymptotic. At that point to improve the performance, the network design needs to be changed (i.e. number of neurons and layers).

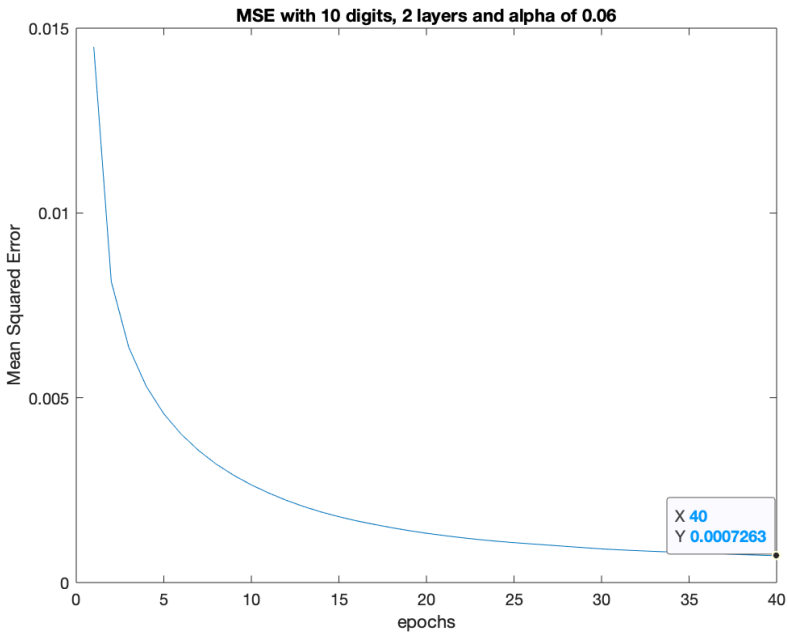


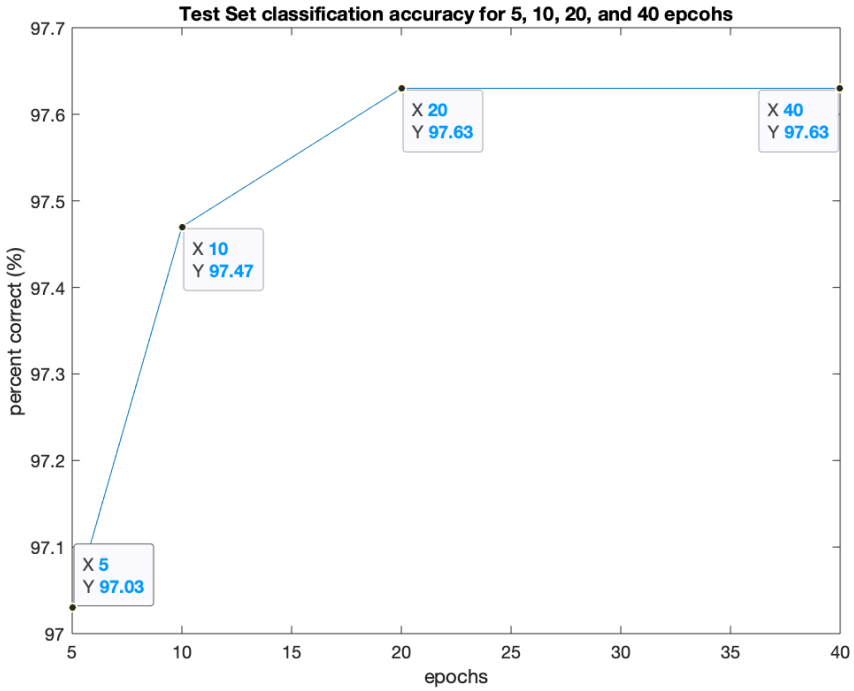


MNIST Test Set Accuracy for every Test Case

The aforementioned analysis that increase the number of epochs does not necessarily improve performance of network can be reflected below as the accuracy of network trained for 20 neurons







Conclusion

By understanding the results, we can conclude two things. Firstly, in comparison to the Hebbian Rule, backpropagation is a much better Associator. This can be seen in the error caused by adding noise to the images, where in using the Hebbian rule had a very high error rate given an image with noise of 6 pixels. This can be contrasted to the results of ANN using backpropagation where the error rate was 10% given an image with noise of 8 pixels. Another conclusion we can deduce is giving a neural network more time usually increases the performance, however after a certain point the performance may no longer increase most likely due to descending into a local minimum. This thus requires a change in the design of the network.

Appendix

Weight and bias generation

```
function [W,b] = generate_weight_bias(row,col)
W = randn(row, col)*sqrt(2)/(row);
b = zeros(row, 1);
end
```

Backpropagation

```
function [W,b,mse] = backprop(p, t, L, s, alpha, epochs, output)
% p = input, t = target, L = number of layers, s = number of neurons
%alpha = learning_rate, epochs = # of epochs to propagate,
%output = number_of_neurons in the output layer

[input_length, input_samples] = size(p);

% store all matricies
W = cell(L,1); %weight
b = cell(L,1); %bias
n = cell(L,1); %output before transfer function
a = cell(L,1); %output for neuron
S = cell(L,1); %sensitivity

%generate Weights and biases
for m = 1:L
    if (m == 1)
        %calculate the weight, bias, n for first hidden layer
        [W{m},b{m}] = generate_weight_bias(s,input_length);
    elseif (m == L)
        % calculate the weight, bias, n for the last hidden layer
        [W{m},b{m}] = generate_weight_bias(output,s);
    else
        %calculate the weights, and biases between hidden layers
        [W{m},b{m}] = generate_weight_bias(s,s);
    end
end

%calculate and propagate sensitivites backwards
x = 1;
mse = zeros(epochs,1);
while x < epochs+1
    %feedforward
    label = zeros(output,1);
    mse(x) = 0;
    for i = 1:input_samples
        for m = 1:L
```

```
            if (m == 1)
                n{m} = W{m}*p(:,i)+b{m};
            else
                n{m} = W{m}*a{m-1}+b{m};
            end
            a{m} = logsig(n{m}); %calculate the output for each layer
        end
        %convert expected output to vector
        label(t(i)+1) = 1;

        %propagate sensitivities backwards
        for m = L:-1:1
            diff_sig = diag((1-a{m}).*a{m});
            if (m == L)
                mse(x) = mse(x) + sum((label-a{m}).^2)/output;
                S{m} = -2*diff_sig*(label-a{m});
            else
                S{m} = diff_sig*W{m+1}'*S{m+1};
            end
        end

        label (t(i)+1) = 0;
        %update weights and biases
        for m = 1:L
            if (m == 1)
                W{m} = W{m}-alpha*S{m}*p(:,i)';
            else
                W{m} = W{m}-alpha*S{m}*a{m-1}';
            end
            b{m} = b{m}-alpha*S{m};
        end
    end
    mse(x) = mse(x)/input_samples;
    x=x+1;
end
end
```

Add noise

```
function pvec = addNoise(pvec, num)
% ADDNOISE Add noise to "binary" vecto
%   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 pixels
pvec(inds(1:num)) = uint8(~pvec(inds(1:num)));
```

Calculate error from noise

```
function error = noise_error(W,b,p,t)

a = cell(length(W),1);
error = zeros (3,length(t));

for i = 1:3
    for j = length(t)
        correct = 0;
        for k = 1:10
            for h = 1:length(W) %calculation of a
                if (h == 1)
                    noised_image = addNoise(p(:,j),i*4-4);
                    a{h} = logsig(W{h}*noised_image+b{h});
                else
                    a{h} = logsig(W{h}*a{h-1}+b{h});
                end
            end
            [m,ind] = max(a{end});
            if (ind == t(j)+1)
                correct = correct +1;
            end
        end
        error(i,j) = (10-correct)*100/10;
    end
end
end
```

Graph all the results

```
function graph_backprop()

p = [0,1,1,1,0,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1,0,1,1,1,0; %0
    0,1,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1,0,0; %1
    1,1,1,0,0,0,0,0,1,0,0,0,0,1,0,0,1,1,0,0,0,1,0,0,0,0,1,1,1,1]'; %2

% t is read as t0=[1,0,0], t1=[0,1,0], t2 = [0,0,1] by the backprop
% algorithm, however it is stated like this for the sake of brevity
t = [0;1;2];

% number of iterations for three digits
epochs = 3000;
output = 3;
```

```
layers = 4;
neurons = 100;
learning_rate = 0.08;
[W,b,mse] = backprop(p,t,layers,neurons,learning_rate,epochs,output);
error = noise_error(W,b,p,t);
figure;
plot1 = plot([1:epochs],mse);
xlabel('epochs');
ylabel('Mean Squared Error');
name = ['MSE with ' num2str(output) ' digits, ' num2str(layers) ' layers and
alpha of ' num2str(learning_rate)];
title (name);
figure;
plot1 = plot([0,4,8],error);
xlabel('noise (pixels)');
ylabel('error (%)');
title ('Error in classification as noise increases per digit');
legend([plot1],["digit 0","digit 1","digit 2"]);

p = loadMNISTImages("train-images.idx3-ubyte");
t = loadMNISTLabels("train-labels.idx1-ubyte");

test_inputs = loadMNISTImages("t10k-images.idx3-ubyte");
test_labels = loadMNISTLabels("t10k-labels.idx1-ubyte");

progress = zeros(4,1);
counter = 1;
for j = [5,10,20,40]
    epochs = j;
    output = 10;
    layers = 2;
    neurons = 100;
    learning_rate = 0.06;
    [W,b,mse2] = backprop(p,t,layers,neurons,learning_rate,epochs,output);

    figure;
    plot1 = plot([1:epochs],mse2);
    xlabel('epochs');
    ylabel('Mean Squared Error');
    name = ['MSE with ' num2str(output) ' digits, ' num2str(layers) ' layers
and alpha of ' num2str(learning_rate)];
    title (name);

    x = 0;
    for i = 1:length(test_inputs)
        [m,ind] = max(logsig(W{2}*logsig(W{1}*test_inputs(:,i)+b{1})+b{2}));
        if(test_labels(i)+1 == ind)
            x = x+1;
        end
    end
    progress(counter) = x*100/length(test_inputs);
    counter=counter+1;
end
```

```
figure;
plot1 = plot([5,10,20,40],progress);
xlabel('epochs');
ylabel('percent correct (%)');
title ('Test Set classification accuracy for 5, 10, 20, and 40 epcohs');

end
```

Helper function to read MNIST data from http://ufldl.stanford.edu/wiki/resources/mnistHelper.zip