MNIST Written Character Classification with a Multi-Layer Perceptron

MNIST is a dataset 70,000 images of handwritten numbers and their corresponding labels. The dataset comes pre-divided into a training set of 60,000 features and a test set of 10,000 features. The MNIST data set is significantly larger than the Iris Flower dataset, both in terms of its number of examples and its features. Another aspect is that because dataset works well with 10 output values, its useful to encode the output and target values into One Hot format. Additionally, this can be done as a pre-processing step.

Looking at the MNIST homepage, we can get an idea of the sorts of result values to be expected. Two comparable classifiers are listed, both being 2-layer Neural Networks. The first has 300 hidden neurons and a test error rate of 1.6%. The next has 1000 hidden neurons and a test error rate off 3.8%. With the same numbers of hidden neurons similar results should be expected.

Implementation

I found creating the MLP difficult, what follows is a record of how I made my Multi-Layer Perceptron.

Variance

This section concerns the implementation of `initWeight(obj,variance)`.

The key part of this function is the variance, which needs to be a random number in the interval positive variance to negative variance. I achieved this with some code from the MATLAB help for the rand() function. To test the code instead of passing `size(obj.hiddenWeights)` to the `rand()` I did:

```
test = -variance + (variance+variance) * rand(10000,1)
```

Then `max(test)` and `min(test)` which gave 0.9996 and -0.9996 respectively.

Forward Propagation

Forward propagation requires implementation of `compute_net_activation(obj, input)`

To test the function, I created a new script file called `DemoMLPtestcase.m`:

DemoMLPtestcase.m

```
% Create an MLP with 2 inputs, 2 hidden units, 1 output
m = MLP(2, 2, 1);
% Initialize weights in a range +/- 1
m.hiddenWeights = [ 6 0 -2 ; 2 -2 0 ];
m.outputWeights = [ -4 2 2 ];

%forward propagation
m.compute_output([1;0])

%backward propagation
m.adapt_to_target([1;0], 1, 0.5)
```

The script file mirrors the walkthrough in the Week 6 Lecture: MLP Training with Backpropagation lecture. The result of `m.compute_output([1;0])` is 0.4585, which when rounded is the same as the 0.46 on the lecture slides.

Workspace - MLP.compute_net_activation	
Name	Value
outputNet	-0.1665
output	0.4585
obj	1x1 MLP
input	[1;0]
i	2
hiddenNeuronsSize	2
hiddenNet	[4;2]
hidden	[0.9820;0.8808]
bias	1

As can be seen to the left, the Workspace values match those found on the lecture slides.

Key:

Variable Name	Lecture Notation Name
`input`	x_1 , x_2
`outputNet`	a (blue)
`output`	o (blue)
`hiddenNet`	a (green)
`hidden`	o (green)

Additionally, a for loop was included for the sigmoid activation part of the forward propagation so that the code would support an arbitrary number of neurons.

Backward Propagation

For the backwards propagation I took a similar approach as with forward propagation, I ran `DemoMLPtestcase.m` and again compared my values with the values in the lecture slides to confirm that the backwards propagation was working correctly.

Workspace - MLP.adapt_to_target	
Name	Value
bias	1
derivTwo	-0.1344
descent	[-3.9340,2.0592,2.0672]
hiddenBias	[0.9820;0.8808;1]
derivOne	[0.0093;-0.0291]

Workspace values that were not relevant were edited out.

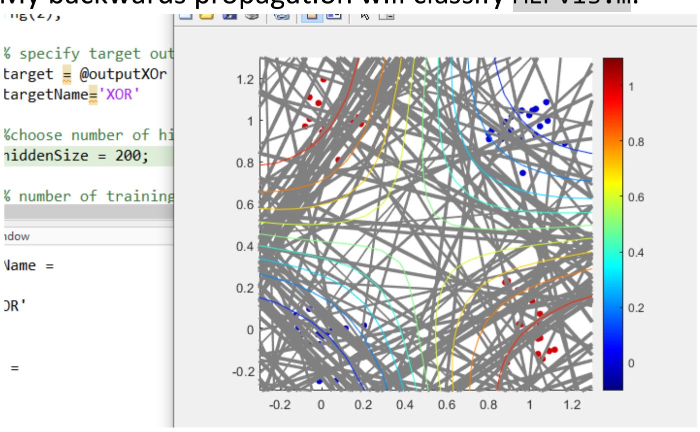
The actual values for the change in weight operations:

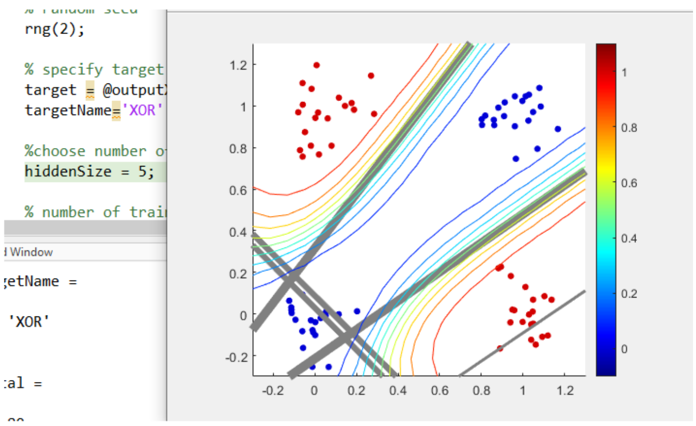
```
K>> [derivOne * inputBias]

ans =

    0.0093         0    0.0093
   -0.0291         0   -0.0291

K>> [derivTwo * hiddenBias']

ans =

   -0.1320   -0.1184   -0.1344
```

Key:

Variable Name	Lecture Notation Name
`bias`	1 (not given name on slides)
`derivTwo`	$d^{(2)}$
`descent` (blue part)	$= \left[\begin{pmatrix}-4\\2\\2\end{pmatrix} \cdot -0.13 \right]_{rows\ 1 \ldots N^1}$
`hiddenBias`	$v^{(2)}$
`derivOne`	$d^{(1)}$
(Not in Workspace) `[derivOne * inputBias]`	$\Delta W^{(1)}$
(Not in Workspace) `[derivTwo * hiddenBias']`	$\Delta W^{(2)}$

My backwards propagation will classify `MLPvis.m`:





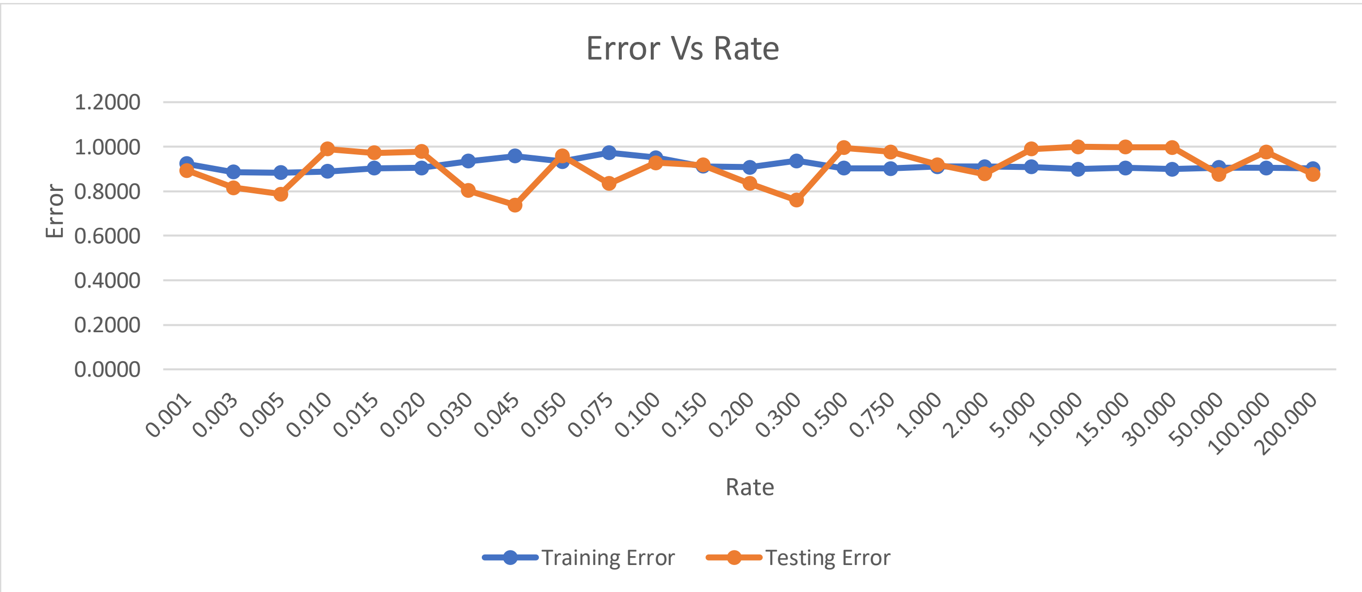
Experimentation

All results were gathered 3 times and averaged.

Learning rate experiment

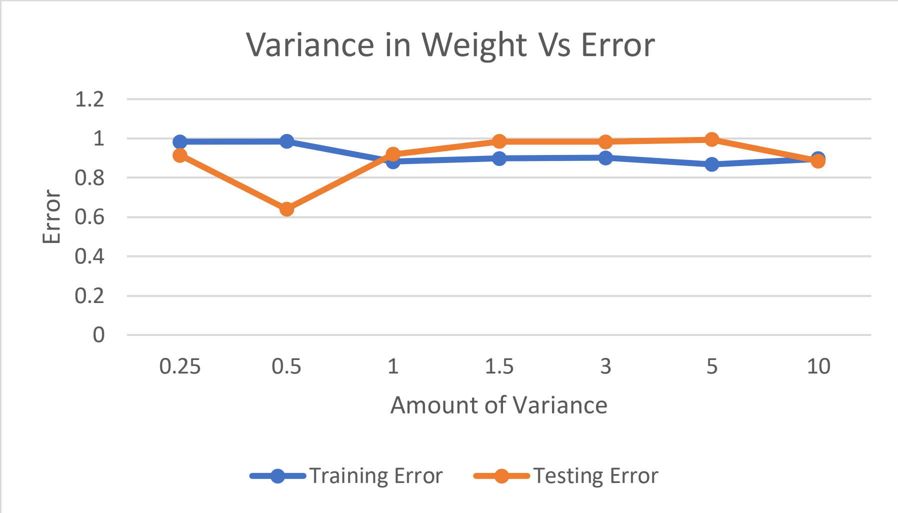
Learning Rate Experiment			
@10,000 epochs, 300 Neurons, 1.0 Weight, rng(2)			
Rate	Training Error	Testing Error	Time (seconds)
0.001	0.9238	0.8934	33.644
0.003	0.8862	0.8151	33.527
0.005	0.8830	0.7865	46.809
0.010	0.8893	0.9896	65.833
0.015	0.9032	0.9730	47.345
0.020	0.9049	0.9787	72.071
0.030	0.9347	0.8022	72.552
0.045	0.9586	0.7385	78.479
0.050	0.9342	0.9595	73.723
0.075	0.9730	0.8346	71.043
0.100	0.9502	0.9275	45.655
0.150	0.9119	0.9179	55.114
0.200	0.9083	0.8346	40.907
0.300	0.9359	0.7595	44.759
0.500	0.9030	0.9958	44.611
0.750	0.9017	0.9762	44.426
1.000	0.9108	0.9189	50.196
2.000	0.9107	0.8769	46.526
5.000	0.9087	0.9898	47.637
10.000	0.8996	1.0000	46.792
15.000	0.9046	0.9985	47.576

The first experiment was one aimed at finding an effective learning rate. The workspace cleared before each run.



Starting Weight experiment

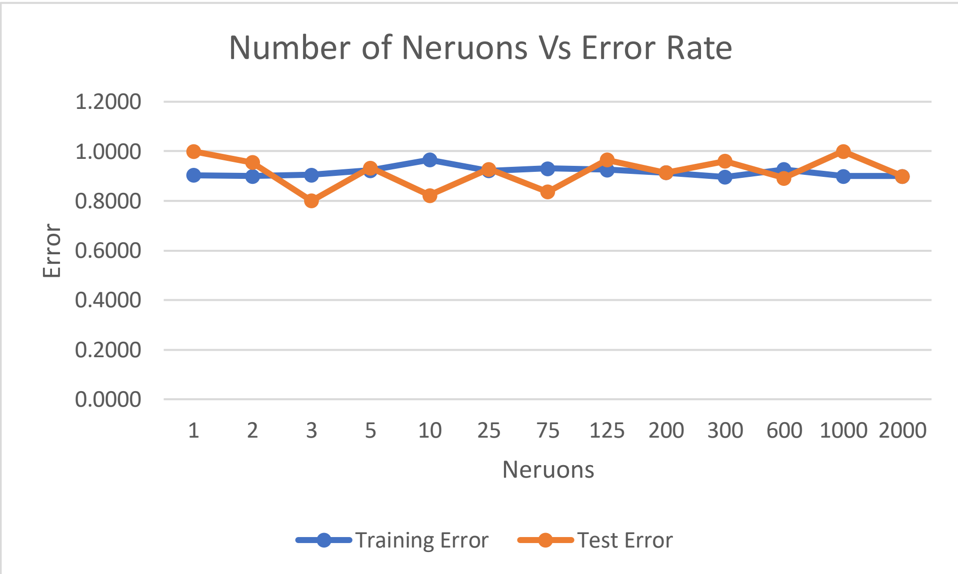
Initial Variance in Weight Experiment		
@100,000 epochs, 300 Neurons, 0.10 rate		
Weight	Training Error	Testing Error
0.25	0.9823	0.9148
0.5	0.9839	0.6406
1	0.8818	0.9184
1.5	0.8972	0.9837
3	0.9014	0.982
5	0.8677	0.9932



This experiments objective was to test different values for `initWeight` (`variance` in `MLP.m`).

Neuron number experiment

Number of Hidden Nerurons experiment		
@25000, 0.010 rate, 1 weight		
Neruons	Training Error	Test Error
1	0.9036	1.0000
2	0.9002	0.9554
3	0.9047	0.8003
5	0.9231	0.9340
10	0.9656	0.8227
25	0.9217	0.9280
75	0.9305	0.8365
125	0.9263	0.9663
200	0.9148	0.9139
300	0.8966	0.9603
600	0.9270	0.8919
1000	0.9000	1.0000
2000	0.9000	0.9000



The purpose of this test was to find the relationship between the number of hidden units and the error rate.

Pre-process targets experiment

Preprocessing Experiment	
@10000 epochs, 0.010 rate, 1 weight	
Non-processed Time(s	Preprocessed Time(s)
50.279	48.811
Change:	1.468

The purpose of this test was a look into how pre-processing the target data might speed up the time it takes to complete the tests.

Conclusion: Testing

Learning rate experiment

The number of epochs used was fairly arbitrary, then the (in my code the epochs only effect the training error). I used 300 neurons so my code would be somewhat comparable to the MNIST homepage datasets. The weight and RNG values are hold overs from `MLPvis.m`. I decided to do a test to determine an effective weight later and I used `rng(2)` because I decided that less variation between tests would be a net positive.

The values used were chosen for a variety of reasons: a wide selection of values was wanted (mostly because on the last coursework it was mentioned that I should use more values). I thought the values chosen would yield interesting results and finally the rate (roughly) goes up in an exponential curve because I knew at as the rate values got too high the results would be poor, so I minimised the number of high values.

From plotting that data, I found that the Error does not seem to change substantially for any given rate. As would be expected the Training Error is more consistent as there was more data to work with than in the testing set.

At times, the testing error would go below the training error, but it seems that these are outliers and not very significant. From here on I would use a rate of 0.010 as it was somewhat lower than the rest o the values while also not being excessively high.

Starting Weight experiment

I again had 300 Neurons but tuned up the number of epochs as the number of runs that needed to be done was smaller. Based on the last experiment I had a rate of 0.0100.

Observing the graph, the number of errors seems decrease until about 1 variance and then the difference is negligible. Based on these results I believe it would be reasonable to continue using a value of 1.

Neuron number experiment

This set of experiments was done with a weight of 1 and a rate of 0.010, based on the results of the previous experiments. The epochs were reduced from 100,000 to 25,000 because the later hidden neuron amounts would take long (the 2000 Neuron test took 799.711 seconds to complete).

The test error is erratic at the lower small hidden unit sizes and smooths out as the sizes increase.

The training error reduces slowly as the hidden neuron sizes increase to a peak of 300, which is in line with some of the data on the MNIST Homepage.

After 300 hidden units the error increases. With the 1000 and 2000 hidden neuron sizes the method I used to measure error lost precision.

Pre-process targets experiment

A shorter experiment, other than having only 10,000 epochs for time reasons the rest of the parameters are set how they have been as a result of the previous experiments.

To get MATLAB to not run through pre-processing code every run, I ran the code once then commented it out, making MATLAB use the already processed code contained in the workspace.

The results of the test show a half second increase – which while small on this set of data – it would translate to exponential savings as the number of epochs increases. A net benefit as long as storage is available.

Conclusion: Observations

Time

In the first experiment (Learning rate experiment) time was measured, for the sequent experiment time stop being measure. It was found that it was an inconsistent metric and not very useful. Things like the PC was hot from previous use, CPU time was being shared with other tasks, etc meant that the time to complete a test would change – it was hard to create test conditions in regard to time. However, for measuring the effectiveness of pre processing time was needed as a metric, so steps were taken to reduce the impact of the non-test conditions (for example leaving the PC alone while it carries out the tests).

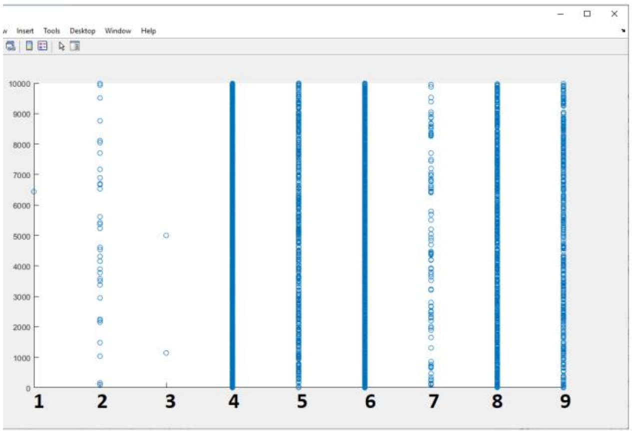
Epochs

Ideally, the number of Epochs (which on the used codes is the number of times the MLP back propagates on the training data) would have been higher but the CPU power to do those experiments in a timely manner was not available.

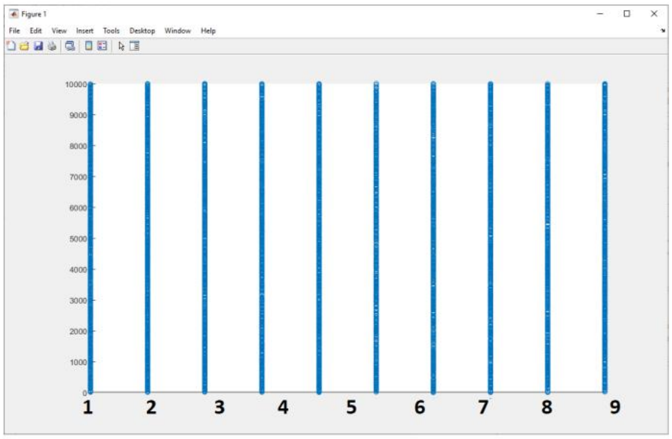
Conclusion: MNIST Homepage Results comparison

Clearly when comparing my results to similar MLPs on the MNIST page my MLP dramatically underperforms.

The 2-layer NN, 1000 hidden units has an error of 4.5, a comparable setup on my MLP is from the neuron number experiment gives and error of 0.9. The only clue I could find for what was going wrong this chart:



scatter(plotOutput,1:epochs)



scatter(plotTarget,1:epochs)

These are scatter plots made from all the values of my MLPs output on the left and the target values on the right – the points in a column, the more times that number appears. As might be expected the right picture, the data is uniform – there is an equal number (or a nearly equal number) of every type of number. On the other hand, my MLP only seems to want to classify 4s, 5s, 6s, 8s and 9s with 7s and 2s being classified much more rarely. Astoundingly my MLP seemly only classified 1s and 3s twice each *out of a set of 60,000!* It would be an interesting issue if it were not so destructive to my results.

I was not able to solve this issue nor able to secure assistance from anybody before the project needed to be handed in.

I looked online and found an K-Nearest Neighbour implementation with 97% correct classification, which is again far more successful than my own implementation

Code

Most visualisation was done in Excel, the visualisation done in MATLAB has the code appended at the bottom of the image

Data.m (main MNIST experimentation file)

```
% random seed
%rng(2);
clear all

% Change the filenames if you've saved the files under different names
% On some platforms, the files might be saved as
% train-images.idx3-ubyte / train-labels.idx1-ubyte
trainImages = loadMNISTImages('train-images-idx3-ubyte');
trainLabels = loadMNISTLabels('train-labels-idx1-ubyte');

% % We are using display_network from the autoencoder code
% display_network(images(:,1:100)); % Show the first 100 images
% disp(labels(1:10));

mlpInput = 784; % Number of inputs, maybe make it size of images
mlpHiddenNeurons = 300; % Number of hidden neurons
mlpOutput = 10; % Number of outputs

multi = MLP(mlpInput, mlpHiddenNeurons, mlpOutput);
multi = multi.initWeight(1);

%Images are the input values, Labels are the target values!

epochs = 10000; %Number of times to repeat
imagesSize = size(trainImages,2);
rate = 0.015;
output = [];
trainError = 0;
index = 1;
target = zeros(mlpOutput,1);

for t = 1:epochs
    target = zeros(mlpOutput,1);
    index = randi([1 imagesSize]); %Picks a random number within the size of images second
dimension
    input = trainImages(:, index);
    target(trainLabels(index) + 1) = 1; % One hot encoding, '+ 1' because MATLAB doesn't
start form zeroth element

    multi.adapt_to_target(input, target, rate); %Backward prop, time to learn!

    output = multi.compute_output(input); %Forward prop (on only the input)

    %Calulate Errors
    [Max,targetDecode] = max(target); % Decodes the One Hot back to numbers
    [Max,outputDecode] = max(output); % Max isn't actually needed here

    if targetDecode == outputDecode
         else
             trainError = trainError + 1;
    end

    plotOutput(t) = outputDecode;
    plotTarget(t) = targetDecode;
end

trainError = trainError / epochs;

%Classifies the whole set
```

```
%Load the testing data
testImages = loadMNISTImages('t10k-images-idx3-ubyte');
testLabels = loadMNISTLabels('t10k-labels-idx1-ubyte');

testSize = size(testImages, 2);
testError = 0;

for i = 1:testSize
    target = zeros(mlpOutput,1);
    input = testImages(:, i);
    target(testLabels(i) + 1) = 1;

    output = multi.compute_output(input);

    %Calulate Errors
    [Max,targetDecode] = max(target);
    [Max,outputDecode] = max(output);

    if targetDecode == outputDecode
        else
            testError = testError + 1;
    end

end

testError = testError / testSize;

[trainError, testError, mlpHiddenNeurons]
```

DataPrePro.m (the same as *Data.m* but with changes to allow for pre-processing the targets)

```
% random seed
%rng(2);

% Change the filenames if you've saved the files under different names
% On some platforms, the files might be saved as
% train-images.idx3-ubyte / train-labels.idx1-ubyte
trainImages = loadMNISTImages('train-images-idx3-ubyte');
trainLabels = loadMNISTLabels('train-labels-idx1-ubyte');

% % We are using display_network from the autoencoder code
% display_network(images(:,1:100)); % Show the first 100 images
% disp(labels(1:10));

mlpInput = 784; % Number of inputs, maybe make it size of images
mlpHiddenNeurons = 300; % Number of hidden neurons
mlpOutput = 10; % Number of outputs

multi = MLP(mlpInput, mlpHiddenNeurons, mlpOutput);
multi = multi.initWeight(1);

%Images are the input values, Labels are the target values!

epochs = 10000; %Number of times to repeat
imagesSize = size(trainImages,2);
%preProcessTarget = zeros(10, imagesSize);
rate = 0.015;
output = [];
trainError = 0;
index = 1;

%Preproccesing training targets
```

```
for n = 1: imagesSize
    preProcessTarget(trainLabels(n) + 1, n) = 1;
end;

for t = 1:epochs
    index = randi([1 imagesSize]); %Picks a random number within the size of images second
dimension
    input = trainImages(:, index);
    target = preProcessTarget(:,index); % setting target

    multi.adapt_to_target(input, target, rate); %Backward prop, time to learn!

    output = multi.compute_output(input); %Forward prop (on only the input)

    %Calulate Errors
    [Max,targetDecode] = max(target); % Decodes the One Hot back to numbers
    [Max,outputDecode] = max(output); % Max isn't actually needed here

    if targetDecode == outputDecode
         else
             trainError = trainError + 1;
    end

    plotOutput(t) = outputDecode;
    plotTarget(t) = targetDecode;
end

trainError = trainError / epochs;

%Classifies the whole set
%Load the testing data
testImages = loadMNISTImages('t10k-images-idx3-ubyte');
testLabels = loadMNISTLabels('t10k-labels-idx1-ubyte');

testSize = size(testImages, 2);
testError = 0;
preProcessTarget = zeros(10, imagesSize);


%Preproccesing testing targets
for n = 1: testSize
    preProcessTarget(trainLabels(n) + 1, n) = 1;
end

for i = 1:testSize
    input = testImages(:, i);
    target = preProcessTarget(:,i);

    output = multi.compute_output(input);

    %Calulate Errors
    [Max,targetDecode] = max(target);
    [Max,outputDecode] = max(output);

    if targetDecode == outputDecode
         else
             testError = testError + 1;
    end
end

testError = testError / testSize;

[trainError, testError, mlpHiddenNeurons]
```

DemoMLPtestcase.m (Follows the lecture slides, used to see if my propagations worked)

```
% Create an MLP with 2 inputs, 2 hidden units, 1 output
m = MLP(2, 2, 1);
% Initialize weights in a range +/- 1
m.hiddenWeights = [ 6 0 -2 ; 2 -2 0 ];
m.outputWeights = [ -4 2 2 ];

%forward propagation
m.compute_output([1;0])

%backward propagation
m.adapt_to_target([1;0], 1, 0.5)
```

MPL.m (*filled out*)

```
% A Multi-layer perceptron class
classdef MLP < handle
    % Member data
    properties (SetAccess=private) %change back to private
        inputDim % Number of inputs
        hiddenDim % Number of hidden neurons
        outputDim % Number of outputs

        hiddenWeights % Weight matrix for the hidden layer, format (hiddenDim)x(inputDim+1)
to include bias terms
        outputWeights % Weight matrix for the output layer, format (outputDim)x(hiddenDim+1)
to include bias terms
    end

    methods
        % Constructor: Initialize to given dimensions and set all weights
        % zero.
        function obj=MLP(inputD,hiddenD,outputD)
            obj.inputDim=inputD;
            obj.hiddenDim=hiddenD;
            obj.outputDim=outputD;
            obj.hiddenWeights=zeros(hiddenD,inputD+1);
            obj.outputWeights=zeros(outputD,hiddenD+1);
        end

        % TODO Implement a randomized initialization of the weight
        % matrices.
        % Use the 'variance' parameter to control the spread of initial
        % values.
        function obj=initWeight(obj,variance)
            % Note: 'obj' here takes the role of 'this' (Java/C++) or
            % 'self' (Python), refering to the object instance this member
            % function is run on.

            %obj.hiddenWeights=% TODO
            %obj.outputWeights=% TODO

            %Assign weights withing a range defined by range "variance"
            hidden = -variance + (variance+variance) * rand(size(obj.hiddenWeights));
            output = -variance + (variance+variance) * rand(size(obj.outputWeights));
            %"" from the "rand( -variance + (variance+variance)*)" documentation
            obj.hiddenWeights = hidden;
            obj.outputWeights = output;

%               test = -variance + (variance+variance) * rand(10000,1);
```

```
%             max(test);
%             min(test);
        end

        % TODO Implement the forward-propagation of values algorithm in
        % this method.
        % hiddenNet ~ net activation of the hidden-layer neurons
        % hidden ~ output of the hidden-layer neurons
        % outputNet ~ net activation of the output-layer neurons
        % output ~ output of the output-layer neurons
        % Note: the return value is automatically fit into a array
        % containing the above four elements
        function [hiddenNet,hidden,outputNet,output]=compute_net_activation(obj, input)
            %hiddenNet = % TODO
            %hidden = % TODO
            %outputNet = % TODO
            %output = % TODO

            bias = 1;%Should maybe be a global constant

            hiddenNeuronsSize = size(obj.hiddenWeights,1);
            hiddenNeuronsSize = hiddenNeuronsSize(1);

            %hiddenNet = % TODO
            hiddenNet = obj.hiddenWeights * [input;bias];

            %hidden = % TODO
            %   use activation func on data (sigmoid)
            %   Consider making a new function
            %   consider turning into for loop (to see if more neurons)

            %For 2 neurons only, sigmoid
%             hidden = [ 1/(1 + exp(-hiddenNet(1))) ; 1/(1 + exp(-hiddenNet(2)))];

            %For any number of neurons, sigmoid
            hidden = [];


            for i = 1:hiddenNeuronsSize
                hidden = [hidden; 1/(1 + exp(-hiddenNet(i)))];
            end

            %outputNet = % TODO
            outputNet = obj.outputWeights * [hidden;bias];

            %output = % TODO
            output = 1 / (1 + exp(-outputNet));
            output = output';
        end

        function hidden = sig(hiddenNet)
            hidden = 1/(1 + exp(-hiddenNet(i)));
        end

        % This function calls the forward propagation and extracts only the
        % overall output. It does not have to be altered.
        function output=compute_output(obj,input)
            [hN,h,oN,output] = obj.compute_net_activation(input);
        end

        % TODO Implement the backward-propagation of errors (learning) algorithm in
        % this method.
        function obj=adapt_to_target(obj,input,target,rate)
```

```
            [hN,h,oN,o] = obj.compute_net_activation(input);
            % TODO

            bias = 1;
            hiddenBias = [h;bias];
            inputBias = [input;bias]'; %Transpose is here

            derivTwo = [o - target] .* (o .* (1 - o));
            descent = obj.outputWeights - rate * [derivTwo * hiddenBias']; %Gradient Descent
Update

            %use updated weights to do deriv
            obj.outputWeights = descent; %This took me so long to figure out omg

            descent(:,end) = []; %Gets rid of the last value, 'end' gives the last value in
the martix!
            derivOne = [descent' * derivTwo] .* (h .* (1 - h));

            obj.hiddenWeights = obj.hiddenWeights - rate * [derivOne * inputBias]; %Gradient
Descent Update

        end
    end
end
```

MLPvis.m (*just incase*)

```
clear all
close all

% random seed
rng(2);

% specify target output function (represented by function pointers here)
target = @outputXOr
targetName='XOR'

%choose number of hidden neurons
hiddenSize = 4;

% number of training steps between two plot renderings
speedUp = 1000;

record=0; % set to 1 to record video
% WARNING: these videos are /uncompressed/ at first and VERY LARGE
if record
    mov(1:1)=struct('cdata',[],'colormap',[]);
    frame=1;
    title=['mlp-' targetName];
    writerObj = VideoWriter([title '.avi'], 'Uncompressed AVI');
    open(writerObj);
end

% create training data
Neach = 20;
Ntotal = 4*Neach
% 0/1 values
X = [repmat([0;0],1,Neach) repmat([0;1],1,Neach) repmat([1;0],1,Neach)
repmat([1;1],1,Neach)];
% plus noise
X = X + randn(2, Ntotal)*0.1;
%... plus constant feature:
XF = [X; repmat([1],1,Ntotal)];
```

```
% outputs
Y = repmat(0, 1, Ntotal);
for i=1:Ntotal
    Y(i) = target(X(:,i));
end

%show data (color chosen by labels in Y)
scatter(X(1,:), X(2,:), 25, Y, 'filled')

colormap('jet');
colorbar();

pbaspect([1 1 1]) % quadratic aspect ratio;

hold on;

% initialize MLP
m = MLP(2, hiddenSize, 1);
m = m.initWeight(1.0);

% initialize hidden neuron visualization
for i=1:hiddenSize
    points = perceptronBoundary(m.hiddenWeights(i,1:3), 0);
    strength = m.outputWeights(1,i);
    boundary(i) = plot(points(1,:), points(2,:), 'LineWidth',2, 'Color', [0.5 0.5 0.5]);
end

% initialize output visualization (countours genereted through a grid of input values)
meshSize = 20;
as = linspace(-0.3,1.3,meshSize);
bs = linspace(-0.3,1.3,meshSize);
[A, B] = meshgrid(as,bs);
E = zeros(meshSize,meshSize);
for i=1:meshSize
    for j=1:meshSize
        E(i,j) = m.compute_output([A(i,j); B(i,j)]);
    end
end
[colorMat, outputContour] = contour(A,B,E);
caxis([-0.1 1.1]);

% enforce axes limits
xlim([-0.3 1.3]);
ylim([-0.3 1.3]);

% initialize marker indicating current data
marker = scatter([], [], 100, 's', 'filled');

hold off;

for t = 1:10000
    for i=1:speedUp
        % choose random sample from data
        index = randi([1 Ntotal], 1, 1);

        % evaluate MLP's output (fwd prop)
        yest = m.compute_output(X(:,index));

        % perform learning step (back prop)
        m.adapt_to_target(X(:,index), Y(index), 0.05);
    end

    % update visualizations
    for i=1:hiddenSize
```

```
        points = perceptronBoundary(m.hiddenWeights(i,1:3), 0);
        strength = m.outputWeights(1,i);
        strength = 1+2*sqrt(abs(strength));
        if strength>15
            strength = 15;
        end
        set(boundary(i), 'XData', points(1,:), 'YData', points(2,:), 'LineWidth',strength);
    end
    for i=1:meshSize
        for j=1:meshSize
            E(i,j) = m.compute_output([A(i,j); B(i,j)]);
        end
    end
    set(outputContour, 'ZData', E);

    drawnow;

    % keep frame for video
    if record
        mov(frame)=getframe(gcf);
        writeVideo(writerObj,mov(frame));
        frame=frame+1;
    end
end

% store video
if record
    close(writerObj);
end
```

[END]