COMP6011 Machine Learning

Coursework 2 MNIST Written Character Classification with a Multilayer Perceptron

[17011866, Computer Science]

1.

1). The database of Modified National Institute of Standards and Technology or MNIST database is the well-known collection of handwritten digits that are widely used for training in the machine learning purposes. It consists of 60,000 training images (high school students' handwritten sample digits of 0-9 that are non-identical to one another) with 10,000 test images which are normalised and centred in 28 by 28 pixels, totalling 784 pixels for each digit image. (LeCun, et al., 1998) The intensities of images are illustrated by greyscale ranging from black(0) to white(1). All of 60,000 images should be trained on the prototype, then test the output by using 10,000 test images to achieve the optimal accuracy of the literature. Also, LeCun mentioned that MNIST is ideal for "someone learning techniques and pattern recognition methods with minimal effort".

2). Iris data set or Fisher's Iris dataset is commonly used for data testing in machine learning which contains data set of three different species of iris flowers (Iris Setosa, Iris Virginica and Iris Versicolor) with 50 samples each totalling 150 samples. The four columns of the data measurement format are based on sepal width, sepal length, petal width, and petal length to discriminate amongst the species. K nearest neighbour(kNN) is one of the common supervised learning algorithms for this set which identify the floral type by the vote of majority class by the count of nearest neighbours.

However, with the MNIST dataset, the input data are converted from the formatted greyscale image of 784 pixels for each image whilst the Iris dataset has 4 columns of 150 samples only. If the user is to train only a few thousand sets of training images, the prototype wouldn't be able to recognise the patterns on the images properly and could result with massive error ratings. Thus, the prototype needs to be trained with the whole training set of images (60,000 images), requiring delicate and sophisticated algorithms such as multilayer perceptron.

3). The classification problems can be tackled by using variety of classifier algorithms such as Linear Classification, kNN, and Neural Nets or Multi-Layer Perceptron. One of the simple classification algorithms for supervised machine learning, kNN, measures the input data in X and output in Y column. As this algorithm functions by voting the nearest neighbour to classify the dataset, there are two major distance measurement methods, Euclidean and Manhattan theories where Euclidean is mainly used for iris dataset. Another key algorithm would be the Artificial Neural Network (ANN) which imitate how the human brain interact with data. The first ANN being invented was single-layer perceptron (linear classifier itself) cannot tackle the XOR problems. Hence, Multilayer Perceptron was introduced back then to tackle XOR problems and much more complicated data structures.

Based on the algorithm performances on the MNIST data set homepage, the error ratings across different classifiers are as follow:

- Linear classifier (1-Layer NN) with no pre-processing: 12%,
- Linear classifier with deskewing preprocessing: 8.4%
- kNN, Euclidean (L2) with no preprocessing: 5%.,
- Neural Nets (2-layer NN, 300 hidden units, mean square error) with no preprocessing: 4.7%,
- Neural Nets (2-layer NN, 800 HU, Cross-Entropy Loss) with no preprocessing: 1.6% respectively.

Thus, using multilayer perceptron with backpropagation for this experiment on MNIST dataset would presumably be having higher accuracy in terms of pattern recognition.

2.

1). I have implemented the multi-layer perceptron and the backpropagation training in Matlab as follow.

Feed forward propagation

Firstly, I have initialised the weight matrices using random function of MATLAB for both hidden layer and output layer weights. According to the vector scalar for perceptron, for each neuron, weights are multiply by the input and summed up before being applied by the step/activation function ($a = \sum_i w_i x_i$, eg. $a = w_{11}x_{11} + w_{22}x_{22}$). Thus, in the compute_forward_activation function, the hidden weights and input data are multiplied and summed up the result as a1. Then a1 is being applied by sigmoid activation function. The value of "hidden" (neuron value) is multiply by the output weights, summed as weighted sum of that neuron. Sigmoid activation function is applied using for the "output" (predicted data).

Backpropagation

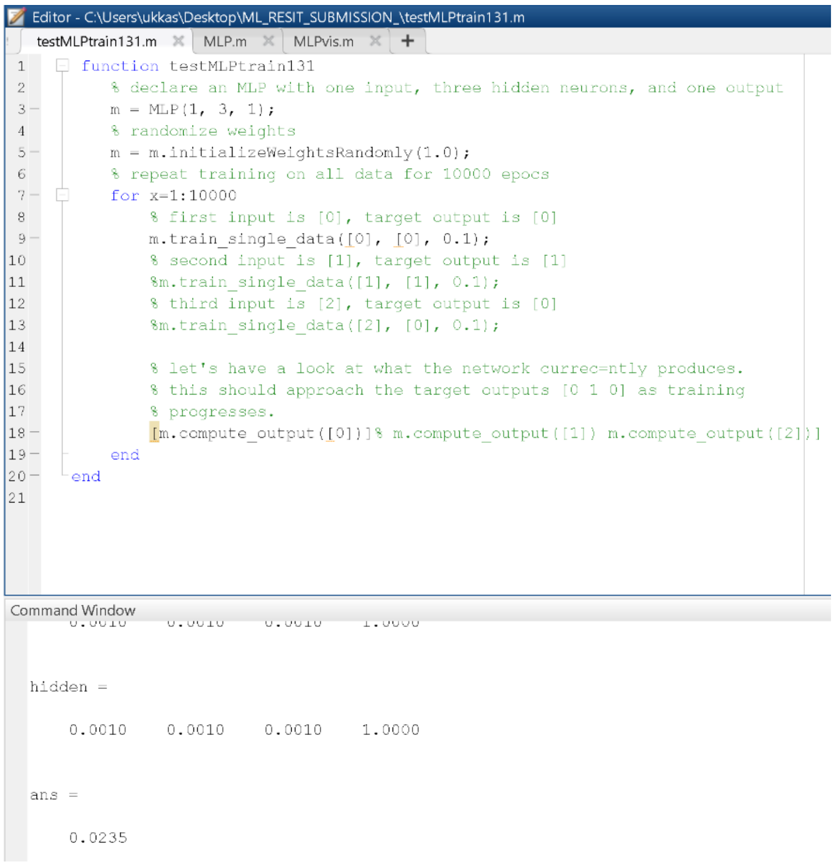
Error rate must be calculated first for us to do backpropagation. It can be calculated by deducting target data from the output data of each output layers. These error rates(e) are applied accordingly with the chain rules of partial derivatives at each respective node to achieve learning rate of output layer(delta). For delta, output, (1-output), error are multiplied as follow: delta = output.*(1-output).*e . For the delta1 of hidden neurons, delta1 = hidden.* (1-hidden).*e1

To minimise the error rate, the weights are needed to be updated based on the gradient decent rules. To update output layer weight, we need hidden layer weight times d2, which can be derived by (output – targetOutputData) * (output * (1- output)). The result is then applied to the following code: mlp.outputLayerWeights = mlp.outputLayerWeights – learningRate * updateOutputLayerWeight.

To update the hidden layer weight, we need to multiply the d1 and input. d1 is derived from multiplication hidden neutron and (1-hidden neuron). The result is then applied as follows: mlp.hiddenLayerWeights = mlp.hiddenLayerWeights – learningRate * transposed updateHiddenLayerWeights.

3.

When running the testMLPtrain131, I have set only one parameter first and found the result as the following. Training single data of first input [0] to return [0], with learning rate of 0.1 is as follows:



Editor - C:\Users\ukkas\Desktop\ML_RESIT_SUBMISSION_\testMLPtrain131.m

testMLPtrain131.m | MLP.m | MLPvis.m

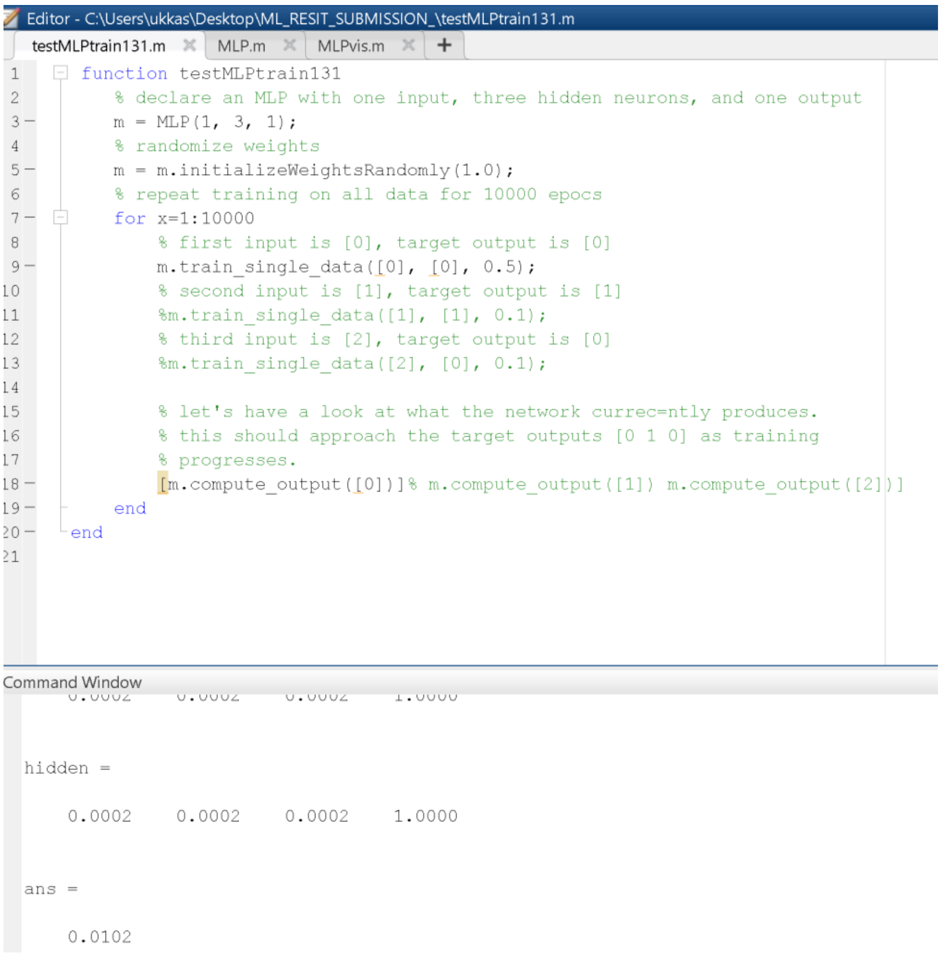
```
function testMLPtrain131
    % declare an MLP with one input, three hidden neurons, and one output
    m = MLP(1, 3, 1);
    % randomize weights
    m = m.initializeWeightsRandomly(1.0);
    % repeat training on all data for 10000 epocs
    for x=1:10000
        % first input is [0], target output is [0]
        m.train_single_data([0], [0], 0.1);
        % second input is [1], target output is [1]
        %m.train_single_data([1], [1], 0.1);
        % third input is [2], target output is [0]
        %m.train_single_data([2], [0], 0.1);

        % let's have a look at what the network currec=ntly produces.
        % this should approach the target outputs [0 1 0] as training
        % progresses.
        [m.compute_output([0])]% m.compute_output([1]) m.compute_output([2])]
    end
end
```

Command Window

```
    0.0010    0.0010    0.0010    1.0000


hidden =

    0.0010    0.0010    0.0010    1.0000


ans =

    0.0235
```

We can see that the answer “0.0235” is very near to the expected value [0]. However, when we changed the learning rate from 0.1 to 0.5 and 0.9 respectively, the results become “0.0102” and

“0.00765” which is relatively nearer to the expected value as in the following figures.

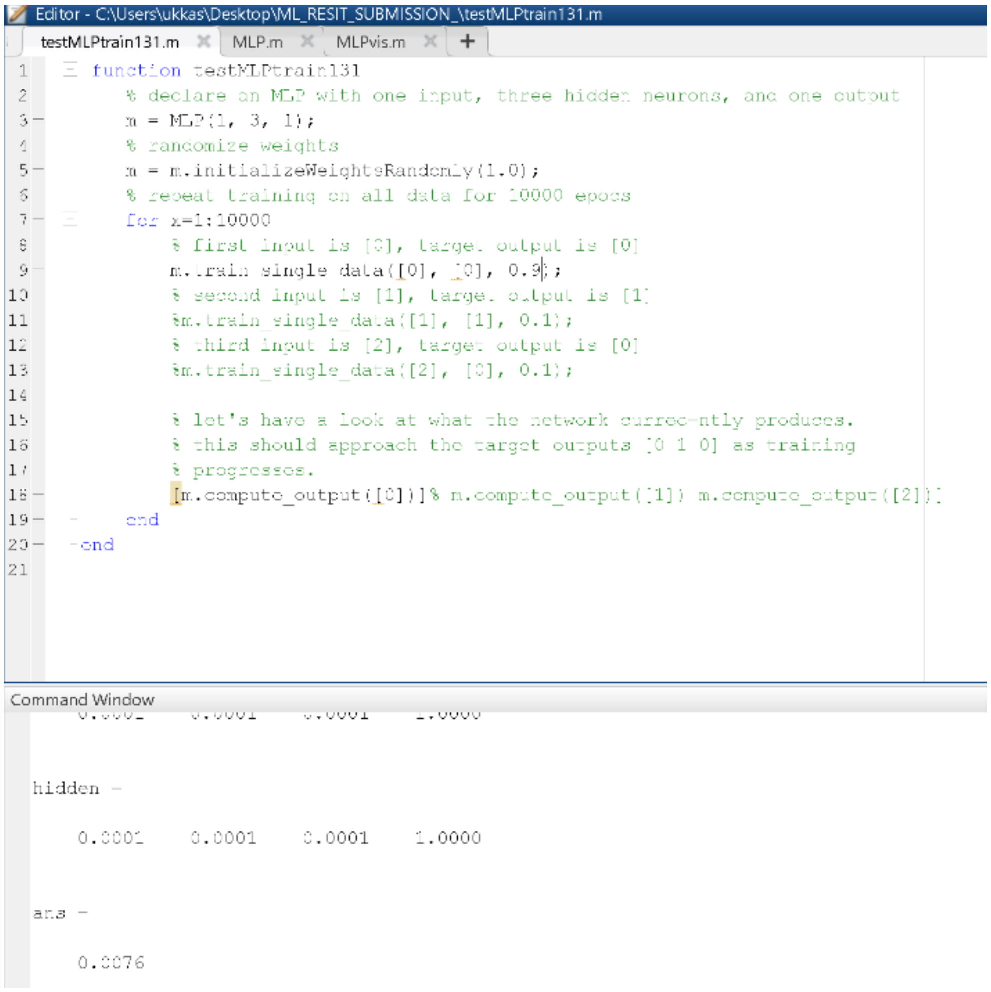

Editor - C:\Users\ukkas\Desktop\ML_RESIT_SUBMISSION_\testMLPtrain131.m

testMLPtrain131.m | MLP.m | MLPvis.m

```
function testMLPtrain131
    % declare an MLP with one input, three hidden neurons, and one output
    m = MLP(1, 3, 1);
    % randomize weights
    m = m.initializeWeightsRandomly(1.0);
    % repeat training on all data for 10000 epocs
    for x=1:10000
        % first input is [0], target output is [0]
        m.train_single_data([0], [0], 0.5);
        % second input is [1], target output is [1]
        %m.train_single_data([1], [1], 0.1);
        % third input is [2], target output is [0]
        %m.train_single_data([2], [0], 0.1);

        % let's have a look at what the network currec=ntly produces.
        % this should approach the target outputs [0 1 0] as training
        % progresses.
        [m.compute_output([0])]% m.compute_output([1]) m.compute_output([2])]
    end
end
```

Command Window

```
    0.0002    0.0002    0.0002    1.0000


hidden =

    0.0002    0.0002    0.0002    1.0000


ans =

    0.0102
```

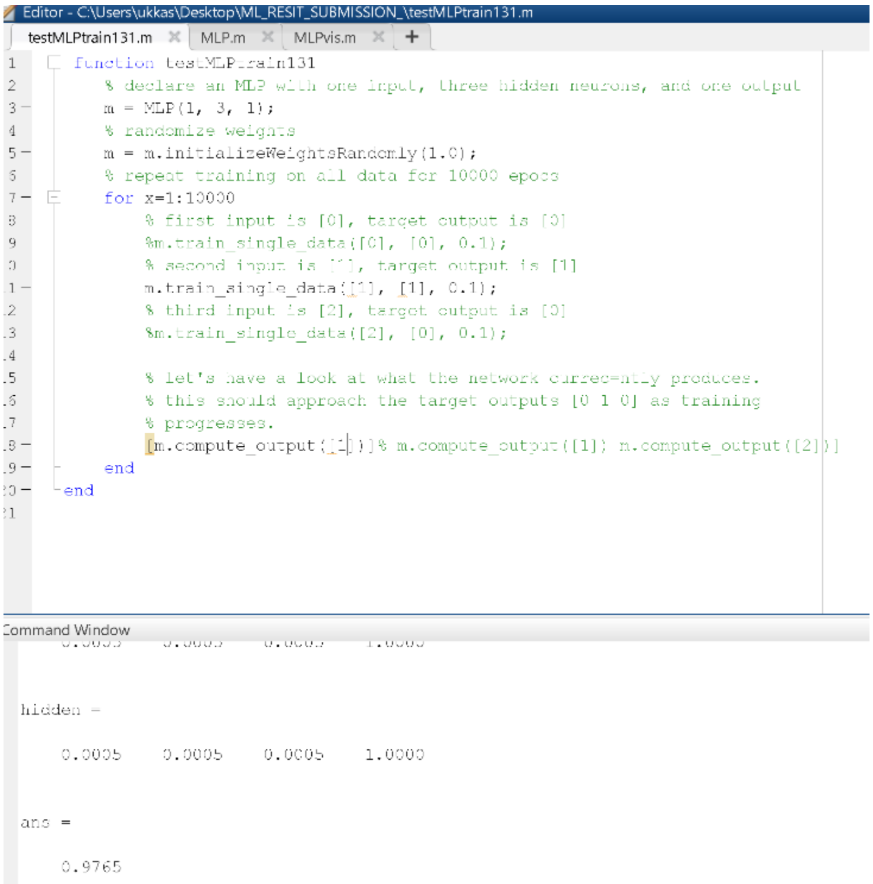

Editor - C:\Users\ukkas\Desktop\ML_RESIT_SUBMISSION_\testMLPtrain131.m

```
function testMLPtrain131
    % declare an MLP with one input, three hidden neurons, and one output
    m = MLP(1, 3, 1);
    % randomize weights
    m = m.initializeWeightsRandomly(1.0);
    % repeat training on all data for 10000 epocs
    for x=1:10000
        % first input is [0], target output is [0]
        m.train_single_data([0], [0], 0.9);
        % second input is [1], target output is [1]
        %m.train_single_data([1], [1], 0.1);
        % third input is [2], target output is [0]
        %m.train_single_data([2], [0], 0.1);

        % let's have a look at what the network currec-ntly produces.
        % this should approach the target outputs [0 1 0] as training
        % progresses.
        [m.compute_output([0])]% m.compute_output([1]) m.compute_output([2])]
    end
end
```

Command Window

```
    0.0001    0.0001    0.0001    1.0000


hidden =

    0.0001    0.0001    0.0001    1.0000


ans =

    0.0076
```

Also, for the single train data input of [1] to return value [1] with learning rate 0.1, 0.5, 0.9 are as follows:



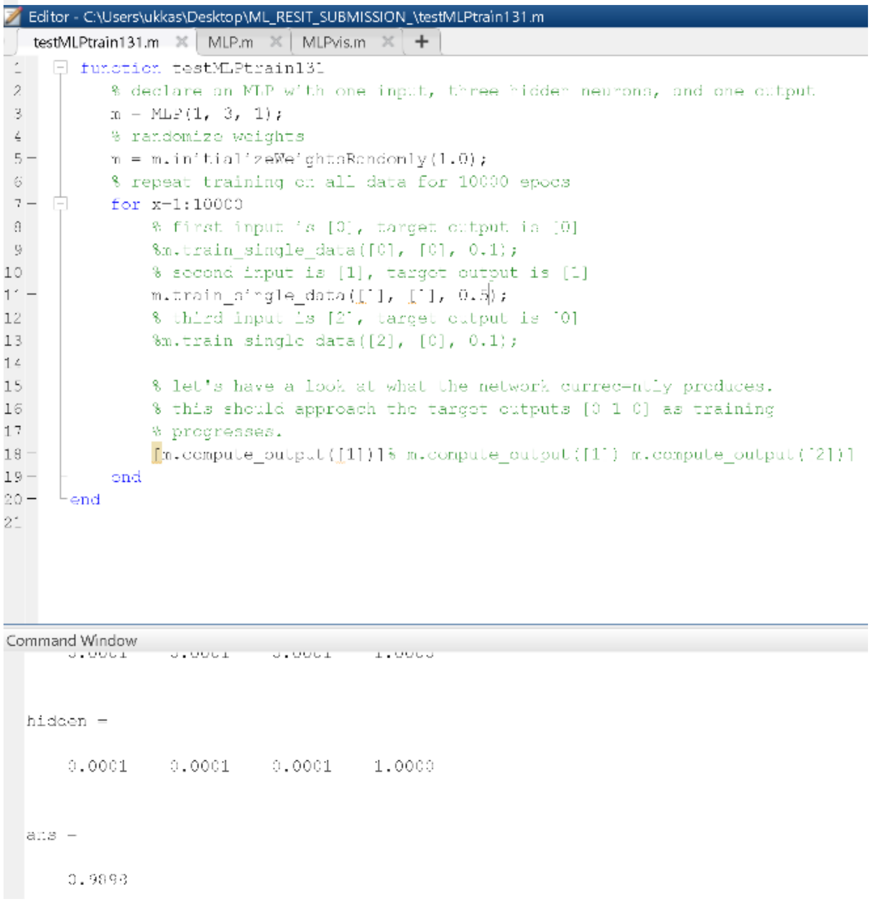
Editor - C:\Users\ukkas\Desktop\ML_RESIT_SUBMISSION_\testMLPtrain131.m

testMLPtrain131.m | MLP.m | MLPvis.m

```
function testMLPtrain131
    % declare an MLP with one input, three hidden neurons, and one output
    m = MLP(1, 3, 1);
    % randomize weights
    m = m.initializeWeightsRandomly(1.0);
    % repeat training on all data for 10000 epocs
    for x=1:10000
        % first input is [0], target output is [0]
        %m.train_single_data([0], [0], 0.1);
        % second input is [1], target output is [1]
        m.train_single_data([1], [1], 0.1);
        % third input is [2], target output is [0]
        %m.train_single_data([2], [0], 0.1);

        % let's have a look at what the network currently produces.
        % this should approach the target outputs [0 1 0] as training
        % progresses.
        [m.compute_output([1])]% m.compute_output([1]) m.compute_output([2])]
    end
end
```

Command Window

```
hidden =

    0.0005    0.0005    0.0005    1.0000


ans =

    0.9765
```



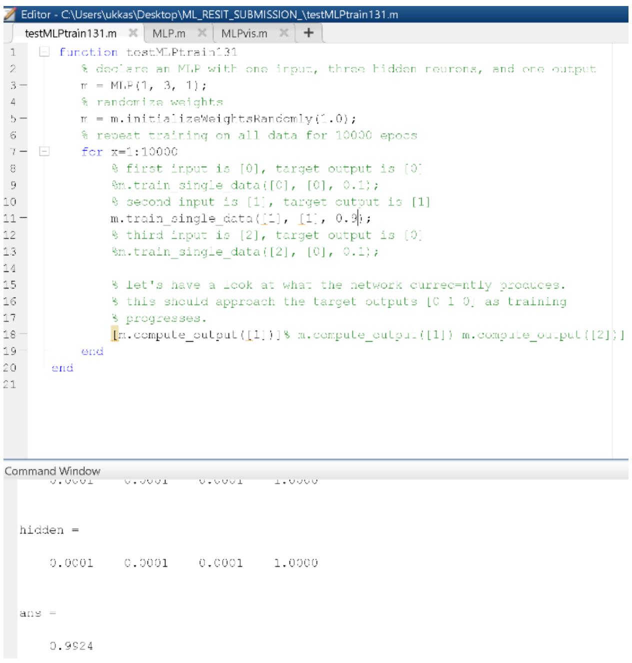
Editor - C:\Users\ukkas\Desktop\ML_RESIT_SUBMISSION_\testMLPtrain131.m

testMLPtrain131.m | MLP.m | MLPvis.m

```
function testMLPtrain131
    % declare an MLP with one input, three hidden neurons, and one output
    m = MLP(1, 3, 1);
    % randomize weights
    m = m.initializeWeightsRandomly(1.0);
    % repeat training on all data for 10000 epocs
    for x=1:10000
        % first input is [0], target output is [0]
        %m.train_single_data([0], [0], 0.1);
        % second input is [1], target output is [1]
        m.train_single_data([1], [1], 0.5);
        % third input is [2], target output is [0]
        %m.train single data([2], [0], 0.1);

        % let's have a look at what the network currently produces.
        % this should approach the target outputs [0 1 0] as training
        % progresses.
        [m.compute_output([1])]% m.compute_output([1]) m.compute_output([2])]
    end
end
```

Command Window

```
hidden =

    0.0001    0.0001    0.0001    1.0000


ans =

    0.9898
```



Editor - C:\Users\ukkas\Desktop\ML_RESIT_SUBMISSION_\testMLPtrain131.m

testMLPtrain131.m | MLP.m | MLPvis.m

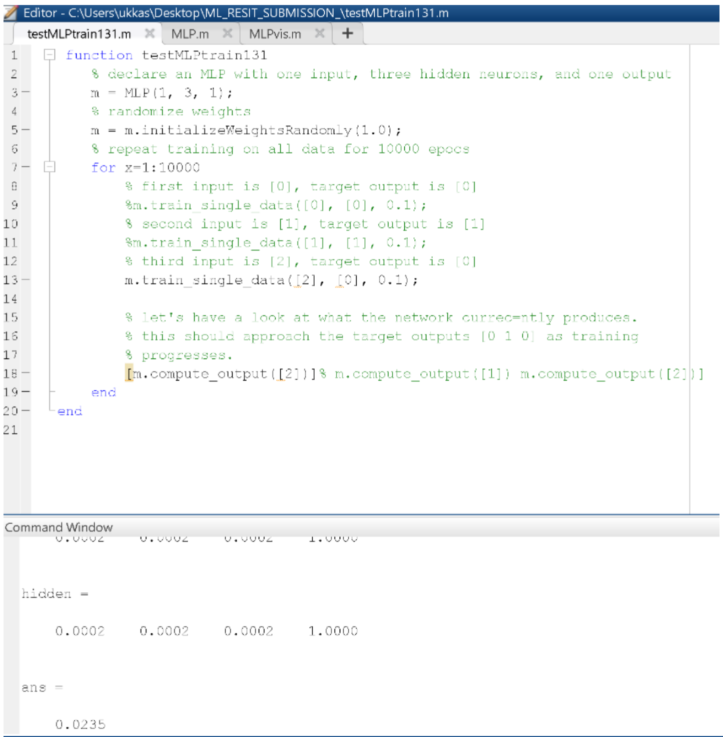
```matlab
function testMLPtrain131
    % declare an MLP with one input, three hidden neurons, and one output
    m = MLP(1, 3, 1);
    % randomize weights
    m = m.initializeWeightsRandomly(1.0);
    % repeat training on all data for 10000 epocs
    for x=1:10000
        % first input is [0], target output is [0]
        %m.train_single_data([0], [0], 0.1);
        % second input is [1], target output is [1]
        m.train_single_data([1], [1], 0.9);
        % third input is [2], target output is [0]
        %m.train_single_data([2], [0], 0.1);

        % let's have a look at what the network currec=ntly produces.
        % this should approach the target outputs [0 1 0] as training
        % progresses.
        [m.compute_output([1])]% m.compute_output([1]) m.compute_output([2])]
    end
end
```

Command Window

```
    0.0001    0.0001    0.0001    1.0000


hidden =

    0.0001    0.0001    0.0001    1.0000


ans =

    0.9924
```

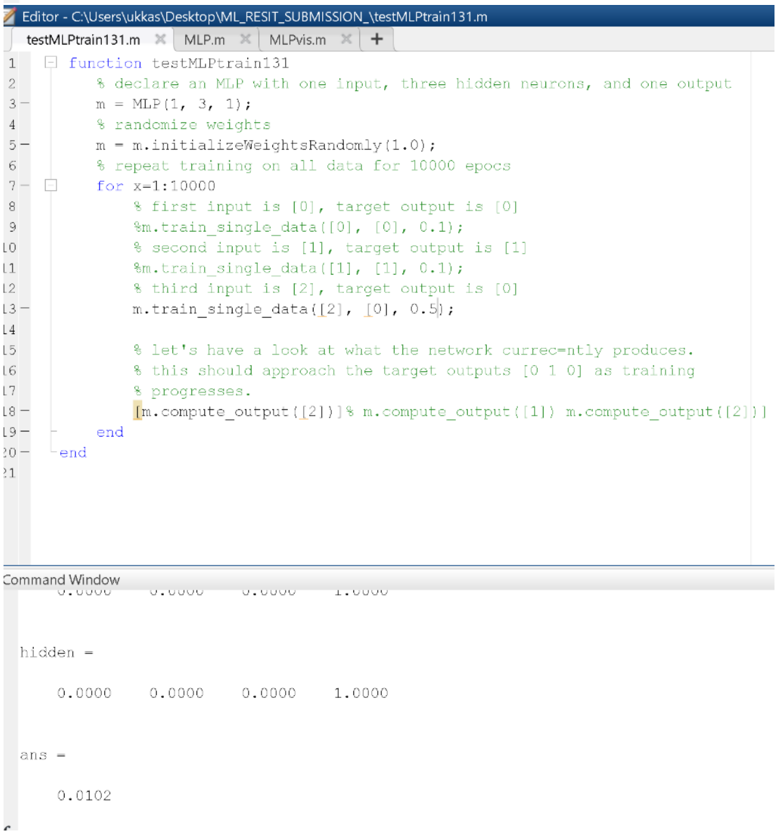
The answer at learning rate 0.1 is “0.9765”, learning rate 0.5 is “0.9898”, learning rate 0.9 is “0.9924”

Once again for the single train data input of [2] to return [0] is carried out at learning rates of 0.1, 0.5, 0.9 as follows:


Editor - C:\Users\ukkas\Desktop\ML_RESIT_SUBMISSION_\testMLPtrain131.m
testMLPtrain131.m
MLP.m
MLPvis.m
Command Window

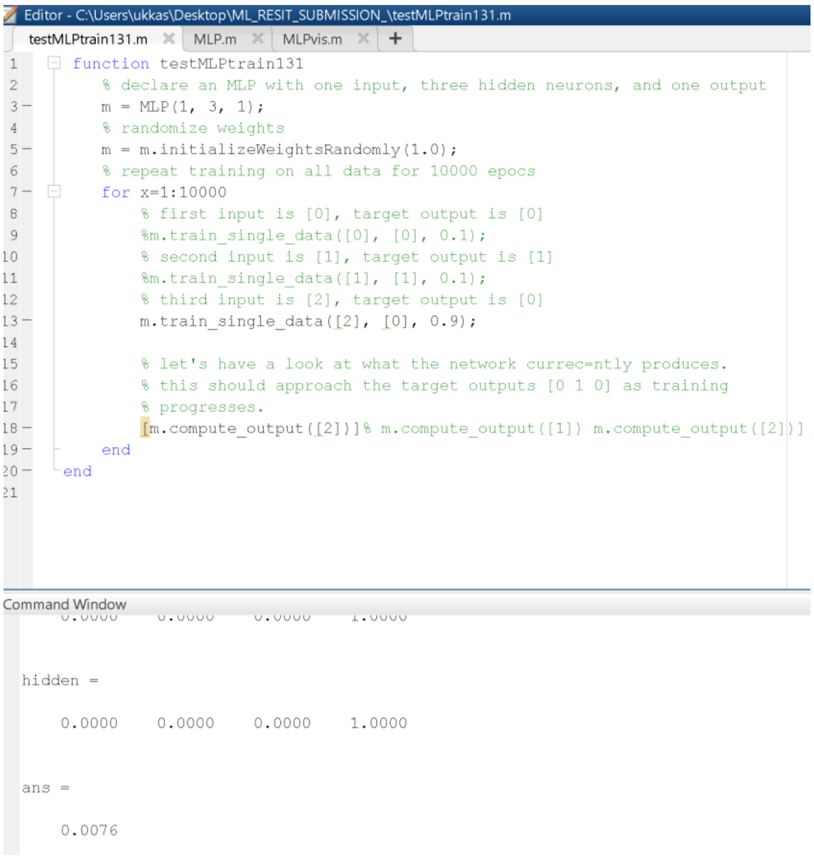
```
function testMLPtrain131
    % declare an MLP with one input, three hidden neurons, and one output
    m = MLP(1, 3, 1);
    % randomize weights
    m = m.initializeWeightsRandomly(1.0);
    % repeat training on all data for 10000 epocs
    for x=1:10000
        % first input is [0], target output is [0]
        %m.train_single_data([0], [0], 0.1);
        % second input is [1], target output is [1]
        %m.train_single_data([1], [1], 0.1);
        % third input is [2], target output is [0]
        m.train_single_data([2], [0], 0.1);

        % let's have a look at what the network currec=ntly produces.
        % this should approach the target outputs [0 1 0] as training
        % progresses.
        [m.compute_output([2])]% m.compute_output([1]) m.compute_output([2])]
    end
end
```

```
    0.0002    0.0002    0.0002    1.0000


hidden =

    0.0002    0.0002    0.0002    1.0000


ans =

    0.0235
```


Editor - C:\Users\ukkas\Desktop\ML_RESIT_SUBMISSION_\testMLPtrain131.m
testMLPtrain131.m
MLP.m
MLPvis.m
Command Window

```
function testMLPtrain131
    % declare an MLP with one input, three hidden neurons, and one output
    m = MLP(1, 3, 1);
    % randomize weights
    m = m.initializeWeightsRandomly(1.0);
    % repeat training on all data for 10000 epocs
    for x=1:10000
        % first input is [0], target output is [0]
        %m.train_single_data([0], [0], 0.1);
        % second input is [1], target output is [1]
        %m.train_single_data([1], [1], 0.1);
        % third input is [2], target output is [0]
        m.train_single_data([2], [0], 0.5);

        % let's have a look at what the network currec=ntly produces.
        % this should approach the target outputs [0 1 0] as training
        % progresses.
        [m.compute_output([2])]% m.compute_output([1]) m.compute_output([2])]
    end
end
```

```
    0.0000    0.0000    0.0000    1.0000


hidden =

    0.0000    0.0000    0.0000    1.0000


ans =

    0.0102
```


Editor - C:\Users\ukkas\Desktop\ML_RESIT_SUBMISSION_\testMLPtrain131.m
testMLPtrain131.m
MLP.m
MLPvis.m
Command Window

```
function testMLPtrain131
    % declare an MLP with one input, three hidden neurons, and one output
    m = MLP(1, 3, 1);
    % randomize weights
    m = m.initializeWeightsRandomly(1.0);
    % repeat training on all data for 10000 epocs
    for x=1:10000
        % first input is [0], target output is [0]
        %m.train_single_data([0], [0], 0.1);
        % second input is [1], target output is [1]
        %m.train_single_data([1], [1], 0.1);
        % third input is [2], target output is [0]
        m.train_single_data([2], [0], 0.9);

        % let's have a look at what the network currec=ntly produces.
        % this should approach the target outputs [0 1 0] as training
        % progresses.
        [m.compute_output([2])]% m.compute_output([1]) m.compute_output([2])]
    end
end
```

```
    0.0000    0.0000    0.0000    1.0000


hidden =

    0.0000    0.0000    0.0000    1.0000


ans =

    0.0076
```

The answers for single train data input [2] to return [0] at learning rate 0.1 is “0.0235”, 0.5 is “0.0102”, and 0.9 is “0.0076” respectively. Hence, we can see that the higher the learning rate gets, the results are getting nearer to the expected value.

Error rating can be calculated as “error rate = target data – output data”. For example, error rating for single train data input [1] returning [1] for all three learning rates can be calculated as :

[(1-0.9765) + (1-0.9898) + (1-0.9924)] / 3

= [0.0235 + 0.0102 + 0.0076]/3

= 0.0137

5.

Results documented for other algorithms as well as MLP configurations on the data set homepage are variable depending how many neural nets and how many hidden units are carried out. The test error rate of 2 neural nets using 300 hidden units using deskewing pre-processing at 1.6 %. (LeCun, et al., 1998) My main limitations are that requiring more knowledge of handling datasets using Matlab because I successfully obtain the answer very near to the expected result whilst training for single data only. However, I could have achieved better results for training multiple data at the same time could have been obtained with further knowledge.

Bibliography

LeCun, Y., Cortes, C. & Burges, C. J., 1998. *The MNIST Database of handwritten digits.* [Online]
Available at: http://yann.lecun.com/exdb/mnist/
[Accessed 12 10 2021].

Source Code

MLP.m

```
% Note: this file merely specifies the MLP class. It is not meant to be
% executed as a stand-alone script. The MLP needs to be instantiated and
% then used elsewhere, see e.g. 'testMLP131train.m'.

% A Multi-layer perceptron class
classdef MLP < handle
    % Member data
    properties (SetAccess=private)
        inputDimension % Number of inputs
        hiddenDimension % Number of hidden neurons
        outputDimension % Number of outputs

        hiddenLayerWeights % Weight matrix for the hidden layer, format
(hiddenDim)x(inputDim+1) to include bias terms
        outputLayerWeights % 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.
        % inputD ~ dimensionality of input vectors
        % hiddenD ~ number of neurons (dimensionality) in the hidden layer
        % outputD ~ number of neurons (dimensionality) in the output layer
        function mlp=MLP(inputD,hiddenD,outputD)
            mlp.inputDimension=inputD;
            mlp.hiddenDimension=hiddenD;
            mlp.outputDimension=outputD;
            mlp.hiddenLayerWeights=zeros(hiddenD,inputD+1);
            mlp.outputLayerWeights=zeros(outputD,hiddenD+1);
        end

        % TODO Implement a randomized initialization of the weight
        % matrices.
        % Use the 'stdDev' parameter to control the spread of initial
        % values.
        function mlp=initializeWeightsRandomly(mlp,~)
            % Note: 'mlp' here takes the role of 'this' (Java/C++) or
            % 'self' (Python), refering to the object instance this member
            % function is run on.
            mlp.hiddenLayerWeights= 2 * rand(mlp.hiddenDimension,mlp.inputDimension+ 1) - 1;
            mlp.outputLayerWeights= 2 * rand(mlp.outputDimension,mlp.hiddenDimension+1)-1;
        end

        % TODO Implement the forward-propagation of values algorithm in
        % this method
        %
        % inputData ~ a vector of data representing a single input to the
        % network in column format. It's dimension must fit the input
        % dimension specified in the contructor.
        %
        % hidden ~ output of the hidden-layer neurons
        % output ~ output of the output-layer neurons
        %
        % Note: the return value is automatically fit into a array
        % containing the above two elements
        function [hidden,output]=compute_forward_activation(mlp, inputData)

              bias = 1;
              v1 = [
                  inputData,
                  bias
                  ];

              a1 = sum((v1' .*  mlp.hiddenLayerWeights)');
              hiddenNeurons = Sigmoid(a1);
              hidden = [hiddenNeurons, 1]
              a2 = hidden * mlp.outputLayerWeights';
              output = Sigmoid(a2);

        end


        % This function calls the forward propagation and extracts only the
        % overall output. It does not have to be altered.
        function output=compute_output(mlp,input)
            [~,output] = mlp.compute_forward_activation(input);
        end


        % TODO Implement the backward-propagation of errors (learning) algorithm in
        % this method.
        %
        % This method implements MLP learning by means on backpropagation
        % of errors on a single data point.
        %
        % inputData ~  a vector of data representing a single input to the
        %   network in column format.
        % targetOutputData ~ a vector of data representing the
        %   desired/correct output the network should generate for the given
        %   input (this is the supervision signal for learning)
        % learningRate ~ step width for gradient descent
        %
        % This method is expected to update mlp.hiddenLayerWeights and
        % mlp.outputLayerWeights.
        function mlp=train_single_data(mlp, inputData, targetOutputData, learningRate)

                     [h,o] = mlp.compute_forward_activation(inputData);

                     d2= (o-targetOutputData) * ( o * (1-o));
                     updateOutputLayerWeight= h * d2;


                     d1 = h(1,1:mlp.hiddenDimension) .* (1- h(1,1:mlp.hiddenDimension)) ;

                     input=[
                         inputData,
                         1
                         ];
                     updateHiddenLayerWeights = d1 .* input;

                     mlp.outputLayerWeights =  mlp.outputLayerWeights - learningRate *
updateOutputLayerWeight;
                     mlp.hiddenLayerWeights = mlp.hiddenLayerWeights - learningRate *
updateHiddenLayerWeights';






              end
        end
end
```

Sigmoid.m

```
function y = Sigmoid(x)
    y = 1 ./(1 + exp(-x));
end
```

testMLPtrain131.m

```
function testMLPtrain131
    % declare an MLP with one input, three hidden neurons, and one output
    m = MLP(1, 3, 1);
    % randomize weights
    m = m.initializeWeightsRandomly(1.0);
    % repeat training on all data for 10000 epocs
    for x=1:10000
```

```
        % first input is [0], target output is [0]
        %m.train_single_data([0], [0], 0.1);
        % second input is [1], target output is [1]
        %m.train_single_data([1], [1], 0.1);
        % third input is [2], target output is [0]
        m.train_single_data([2], [0], 0.9);

        % let's have a look at what the network currec=ntly produces.
        % this should approach the target outputs [0 1 0] as training
        % progresses.
        [m.compute_output([2])]% m.compute_output([1]) m.compute_output([2])]
    end
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
```

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
```