1. Brief Introduction

1.a MNIST

MNIST is a sizable collection of handwritten digits to test machine learning algorithms for the categorization of handwritten digits. The MNIST dataset, which is a collection of images from 0 to 9, has 70,000 handwritten digit images of size 28x28, with each pixel encoded as an integer from 1 (white) to 255 (black). 60,000 of MNIST dataset is used for training and 10,000 for testing.

1.b Challenging of MNIST

MNIST data set includes 70,000 samples and IRIS includes 150 samples. IRIS data set provides properties of the different species of IRIS in cm. For MNIST data set, images are stored as bytes, which means reading the data is more difficult. In addition, in order to get more training and testing results, the images need to preprocess.

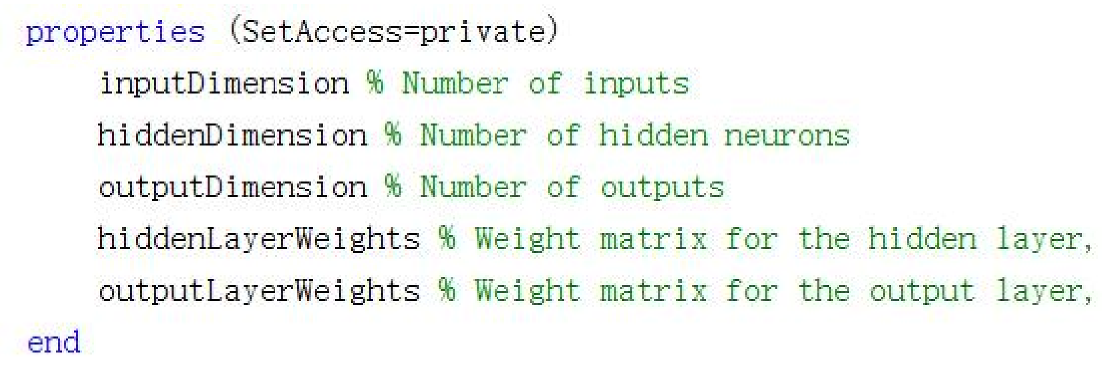
1.c Discuss key algorithm performances on the data set homepage

Softmax function is used to normalize the output activation level by mapping the output to values in the interval (0,1), assigning probabilities to possible outcomes where the sum of all element sums to one, and Softmax is also a good estimate of the probability that the class is the correct answer [1]. In the choice of activation, Softmax and other activations might be used in the hidden layer and output layer to compare the performance of different activation. the two-hidden layer fully connected multi-layer neural network sometimes shows better performance than one-hidden layer fully connected multi-layer neural network [2]. In the implement of multi-layer perception, the one hidden layer and two hidden layers might be implemented.

2. Implement and document a multi-layer perception and the backpropagation training algorithm

2.a.1 Implement and document the three-layer perception

Step 1: Define the properties of the three-layer perception

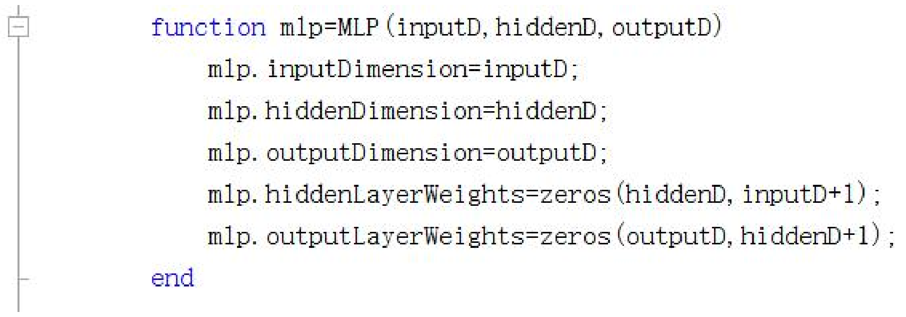


```
properties (SetAccess=private)
    inputDimension % Number of inputs
    hiddenDimension % Number of hidden neurons
    outputDimension % Number of outputs
    hiddenLayerWeights % Weight matrix for the hidden layer,
    outputLayerWeights % Weight matrix for the output layer,
end
```

Figure 1: the code of setting properties for the three-layer perception

For the three-layer perception, it includes the number of input layer neurons, hidden layer neurons, output layer neurons, and the weight of the hidden layer and the output layer.

Step 2: Define the function MLP(), which is used to initialize the three-layer perception.



```
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
```

Figure 2: the code of initialize the three-layer perception

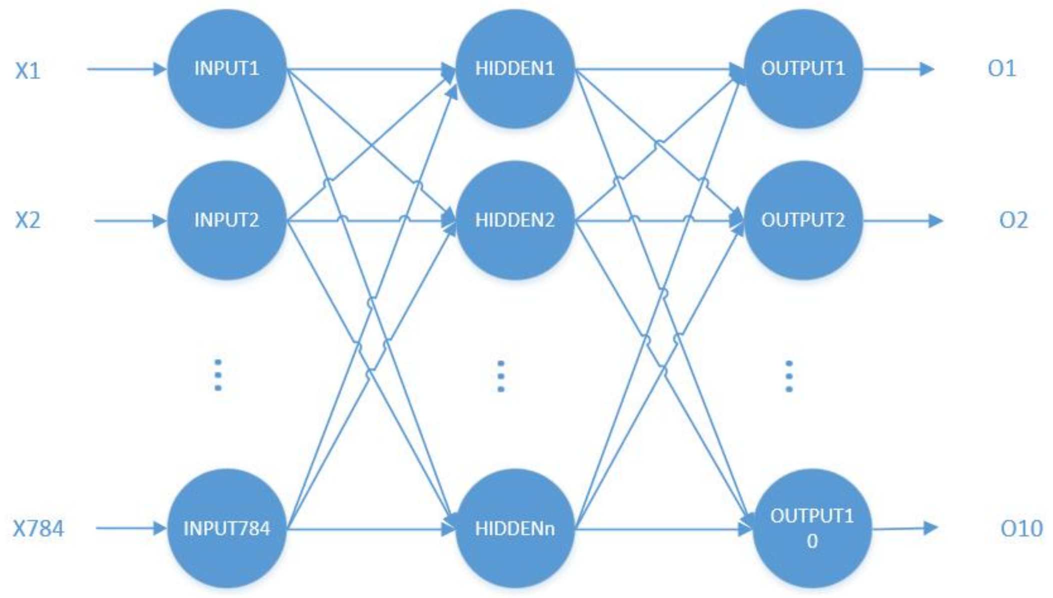
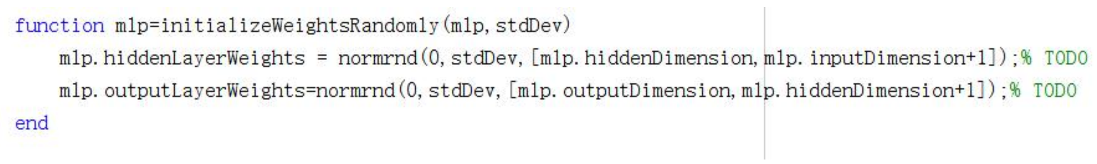


Figure 3: the diagram of the structure of three-layer perception

For this three-layer perception, the structure will follow the figure[3]. The first layer is the input layer, where the input is the pixels of the original image, whose dimension is 28*28. The second layer is the hidden layer whose dimension depends on the Customized input. The final layer is the output layer, considering the MNIST dataset includes ten labels, whose dimension will be set as 10. The weight matrix format for the hidden layer and output layer is set as *[current layer dimension, last layer dimension+1]*, Plus one because the bias including the weight matrixes.

Step 4: initialize weight and bias

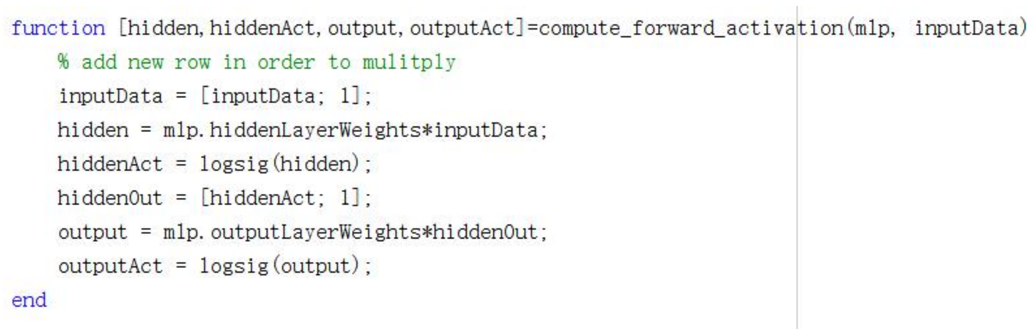


```
function mlp=initializeWeightsRandomly(mlp, stdDev)
    mlp.hiddenLayerWeights = normrnd(0, stdDev, [mlp.hiddenDimension, mlp.inputDimension+1]);% TODO
    mlp.outputLayerWeights=normrnd(0, stdDev, [mlp.outputDimension, mlp.hiddenDimension+1]);% TODO
end
```

Figure 4: the code of initialize the weight and bias

Initialize the weight matrixes of normally distributed random numbers of the form (hiddenDim)x(inputDim+1) and (outputDim)x(hiddenDim+1) are generated using the normand() function.

Step 4: Implement the forward propagation training algorithm



```
function [hidden, hiddenAct, output, outputAct]=compute_forward_activation(mlp,  inputData)
    % add new row in order to mulitply
    inputData = [inputData; 1];
    hidden = mlp.hiddenLayerWeights*inputData;
    hiddenAct = logsig(hidden);
    hiddenOut = [hiddenAct; 1];
    output = mlp.outputLayerWeights*hiddenOut;
    outputAct = logsig(output);
end
```

Figure 5: the code of forward propagation algorithm

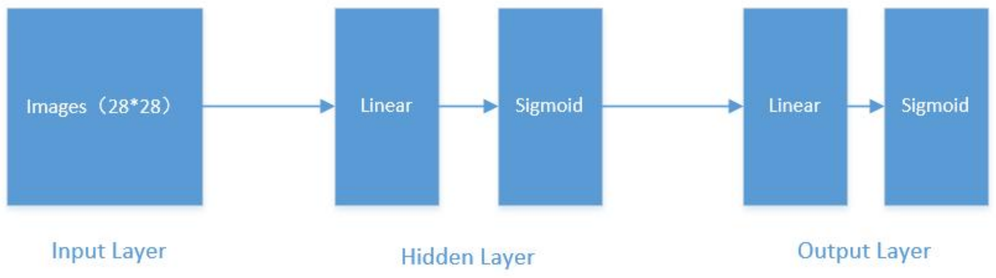


Figure 6: the specific structure of forward propagation algorithm

The forward propagation will follow the figure [5][6], between layers is divided into two parts: linear transformation and non-linear activation, following the below formula [1]:

$$net_l = out_{l-1} * W_l$$

$$out_l = activation(net_l)$$

Formula 1: the formula of forward propagation

l denotes the layer, W_l denotes the weight of the lth layer, out_{l-1} denotes the input to the connection, and net_l denotes the output of a linear transformation, out_l denotes the output of non-linear activation. The activation function Sigmoid is selected as the activation function for the hidden and output layers. Therefore, in the hidden layer and output layer, the process of forward propagation will follow below formula[2]:

$$net_{hi} = input_i * W_{hi}$$

$$out_{hi} = sigmoid(net_{hi})$$

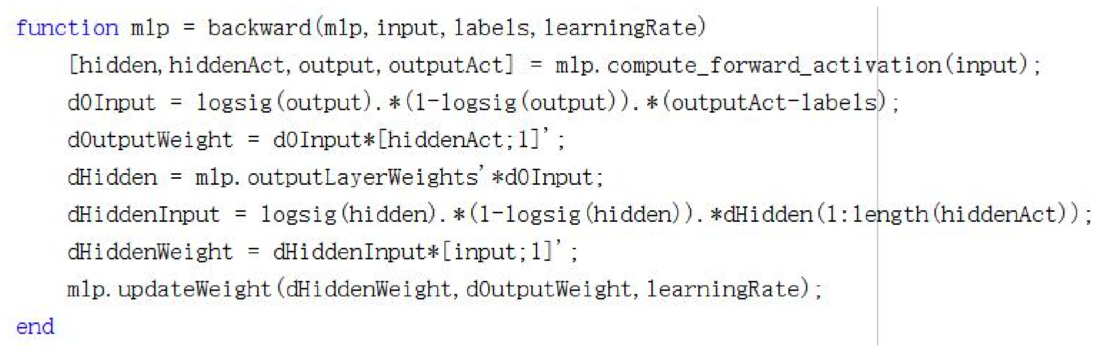
$$net_{Oi} = out_{hi} * W_{oi}$$

$$out_{oi} = sigmoid(net_{oi})$$

Formula 2: the specific formula of forward propagation

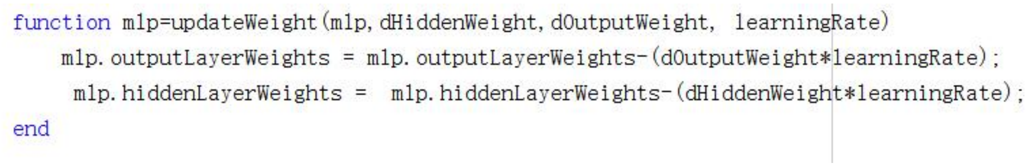
2.a.2 Implement and document the backpropagation training algorithm

Backpropagation training algorithm involves two parts, calculating the output layer error and the hidden layer error.



```
function mlp = backward(mlp, input, labels, learningRate)
    [hidden, hiddenAct, output, outputAct] = mlp.compute_forward_activation(input);
    dOInput = logsig(output).*(1-logsig(output)).*(outputAct-labels);
    dOutputWeight = dOInput*[hiddenAct;1]';
    dHidden = mlp.outputLayerWeights'*dOInput;
    dHiddenInput = logsig(hidden).*(1-logsig(hidden)).*dHidden(1:length(hiddenAct));
    dHiddenWeight = dHiddenInput*[input;1]';
    mlp.updateWeight(dHiddenWeight, dOutputWeight, learningRate);
end
```

Figure 7: the code of backward propagation



```
function mlp=updateWeight(mlp, dHiddenWeight, dOutputWeight,  learningRate)
    mlp.outputLayerWeights = mlp.outputLayerWeights-(dOutputWeight*learningRate);
     mlp.hiddenLayerWeights =  mlp.hiddenLayerWeights-(dHiddenWeight*learningRate);
end
```

Figure 8: the code of updating hidden layer weights and output layer weights

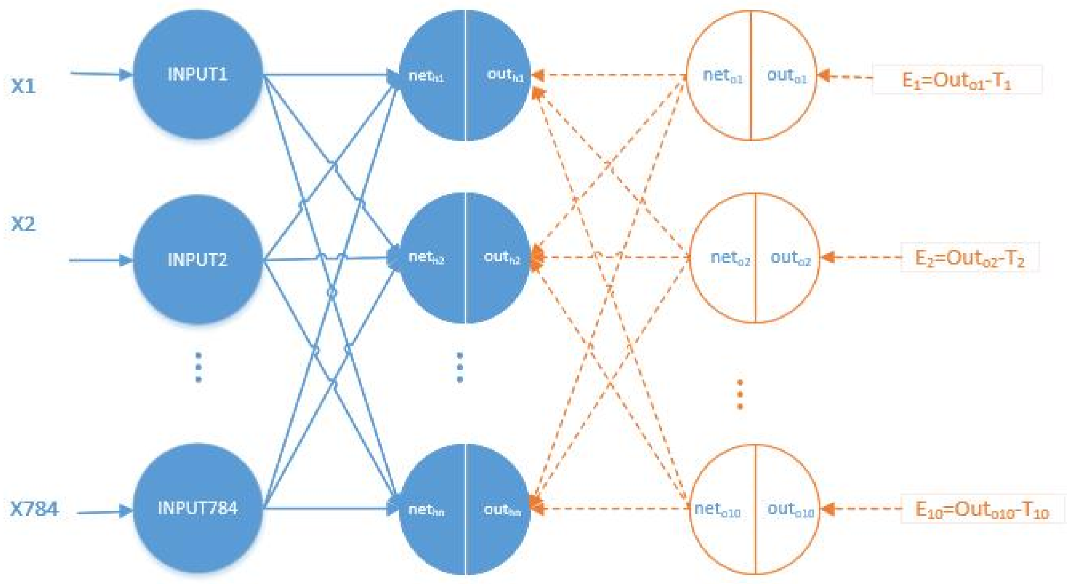


Figure 9: the diagram of output layer of backward propagation

$$\frac{\partial E_{total}}{\partial out_{oi}} = output_i - target_i$$

$$\frac{\partial sogmoid_i(out_{oi})}{\partial_{net_{oi}}} = out_{oi}(1 - out_{oi})$$

$$\frac{\partial net_{oi}}{\partial W_{oi}} = out_{hi}(i = 1,2,..10)$$

$$\frac{\partial E_{total}}{\partial W_{oi}} = \frac{\partial E_{total}}{\partial out_{oi}} * \frac{\partial sogmoid_i(out_{oi})}{\partial_{net_{oi}}} * \frac{\partial net_{oi}}{\partial W_{oi}}$$

$$W_{backoi} = W_{oi} - learingRate * \frac{\partial E_{total}}{\partial W_{oi}}$$

Formula 3: the formulas of output layer of backward propagation

E_{total} denotes the total difference between the actual output and the predicted output

W_{backoi} denotes the updated weights for the output layer

For the process of updating weights for the output layer, the overall steps following the figure[9], and the specific process follows the formula[3].

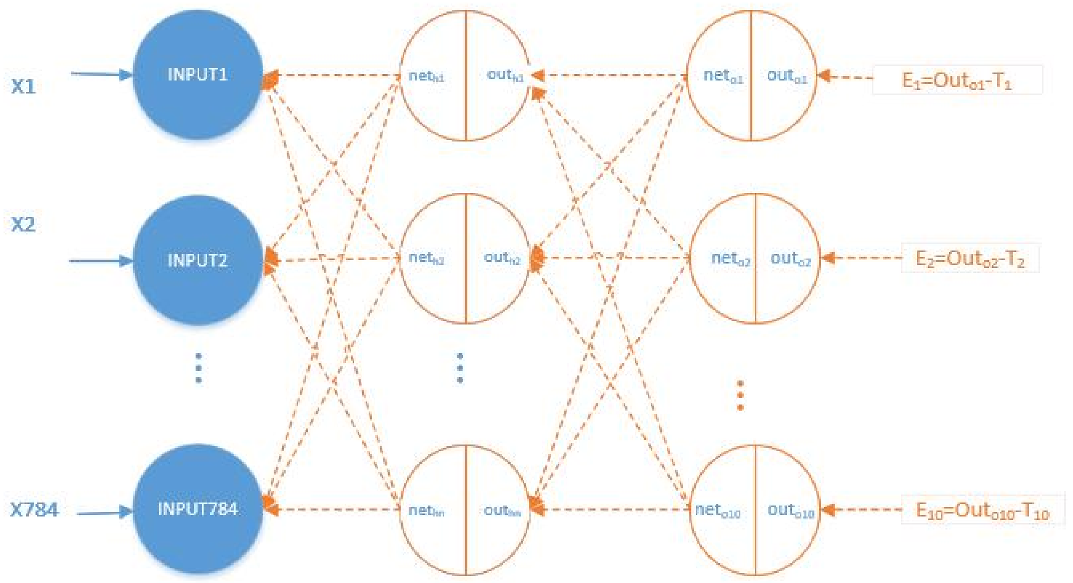


Figure 10: the diagram of hidden layer of backward propagation

$$\frac{\partial E_{total}}{\partial out_{hi}} = \frac{\partial E_1}{\partial out_{h1}} + \frac{\partial E_2}{\partial out_{h1}} + \dots + \frac{\partial E_{10}}{\partial out_{h1}}$$

$$\frac{\partial sogmoid_i(out_{hi})}{\partial_{net_{hi}}} = out_{hi}(1 - out_{hi})$$

$$\frac{\partial net_{hi}}{\partial W_{hi}} = input_i(i = 1,2,..10)$$

$$\frac{\partial E_{total}}{\partial W_{oi}} = \frac{\partial E_{total}}{\partial out_{hi}} * \frac{\partial sogmoid_i(out_{hi})}{\partial_{net_{hi}}} * \frac{\partial net_{hi}}{\partial W_{hi}}$$

$$W_{backhi} = W_{hi} - learingRate * \frac{\partial E_{total}}{\partial W_{hi}}$$

Formula 4: the formulas of hidden layer of backward propagation

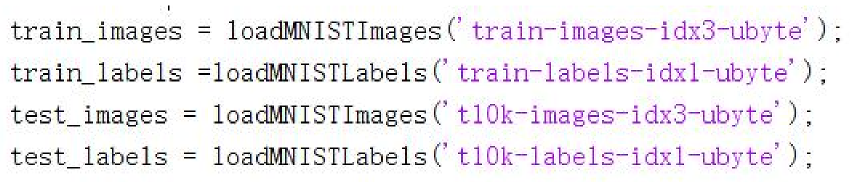
W_{backhi} denotes the updated weights for the hidden layer

For the process of updating weights for the output layer, the overall steps following the figure [10], and the specific process follows the formula [4].

2.b Build and code up systematically step by step and test

2.b.1 Build and code up systematically step by step to train module

Step1: Load MINST dataset

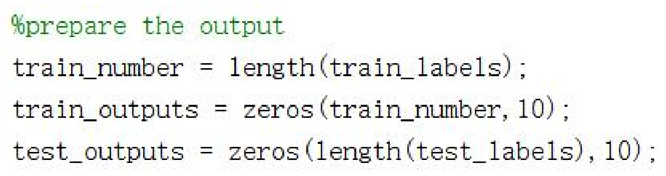


```
train_images = loadMNISTImages('train-images-idx3-ubyte');
train_labels =loadMNISTLabels('train-labels-idx1-ubyte');
test_images = loadMNISTImages('t10k-images-idx3-ubyte');
test_labels = loadMNISTLabels('t10k-labels-idx1-ubyte');
```

Figure 11: the code of load data

MNIST dataset will be loaded data using finished function. There are not normalizing the dataset because the format of image already has converted to double and rescale to [0,1] in *loadMNISTImages()* function.

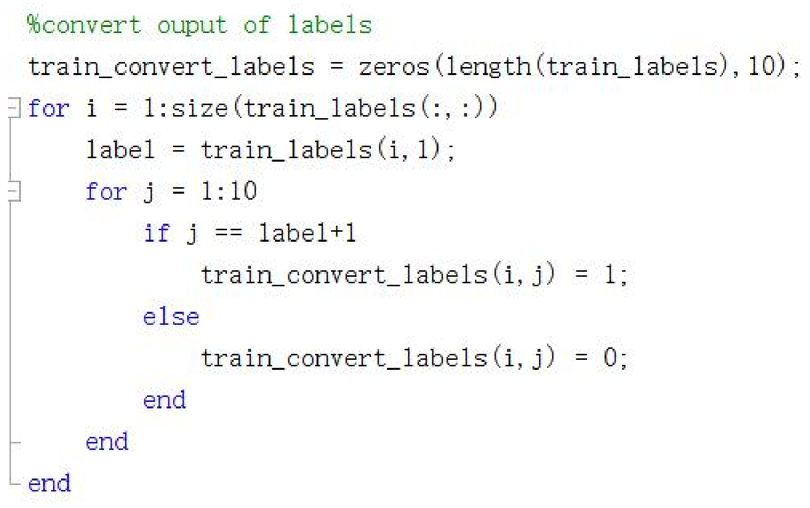
Step2: Create the matrixes to store the results of output



```
%prepare the output
train_number = length(train_labels);
train_outputs = zeros(train_number,10);
test_outputs = zeros(length(test_labels),10);
```

Figure 12: the code of creating matrixes for storing results

Step3: Change the format of training labels

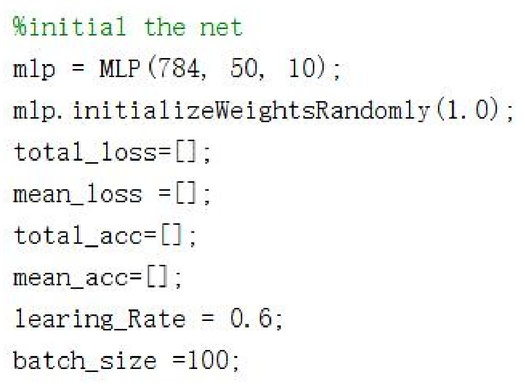


```
%convert ouput of labels
train_convert_labels = zeros(length(train_labels),10);
for i = 1:size(train_labels(:,:))
    label = train_labels(i,1);
    for j = 1:10
        if j == label+1
            train_convert_labels(i,j) = 1;
        else
            train_convert_labels(i,j) = 0;
        end
    end
end
```

Figure 13: the code of transforming the format of training labels

Considering that the dimension of the output layer is 10, the format of the training label will be transformed from *(trainLabelNumber)x1* to *(trainLabelNumber)x10*. The principle is to get the original training label, find the number of columns corresponding to the target in the transformed matrix and assign it to 1 (plus one because Matlab index starts at "1"), and assign the other columns to 0. For instance, the fourth column of the fifth row of the transformed matrix will be assigned to ‘’1’’ and the remaining columns are ‘’0’’ if the value of the fifth label in the training label is 3.

Step 4: initialize the net and parameters

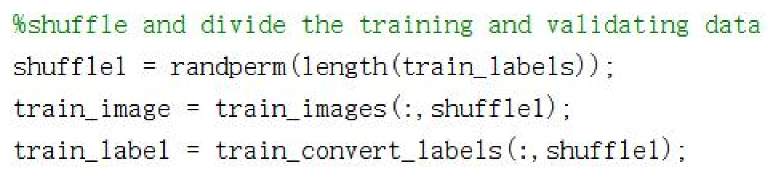


```
%initial the net
mlp = MLP(784, 50, 10);
mlp.initializeWeightsRandomly(1.0);
total_loss=[];
mean_loss =[];
total_acc=[];
mean_acc=[];
learing_Rate = 0.6;
batch_size =100;
```

Figure 14: the code of initializing the three-layer perception

Figure [14] shows the specific dimension of input, hidden and output layer of the three-layer perception will be created, and the hidden layer weights and output layer weights will be initialized randomly.

Step 5: Shuffle the training dataset

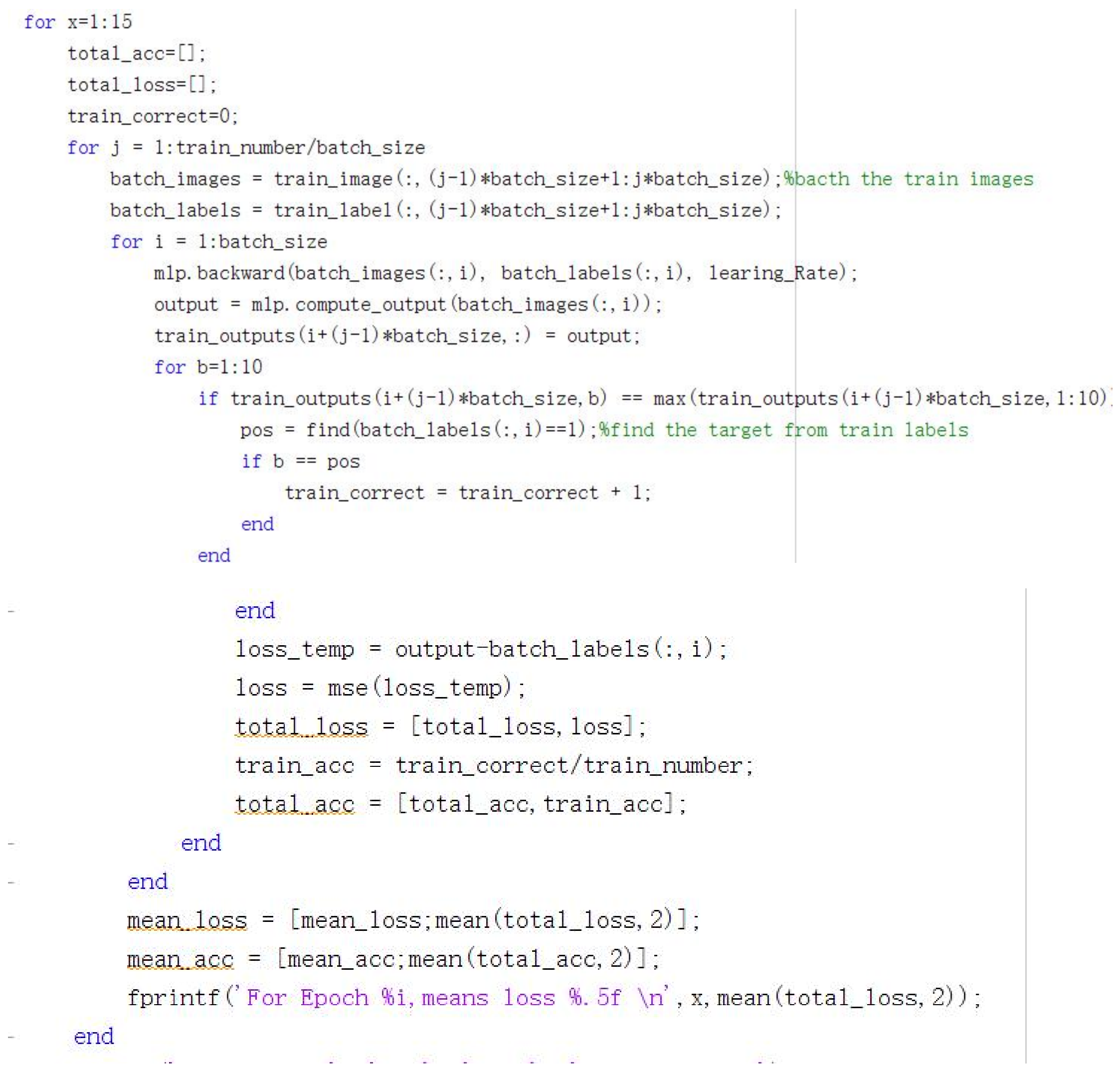


```
%shuffle and divide the training and validating data
shuffle1 = randperm(length(train_labels));
train_image = train_images(:,shuffle1);
train_label = train_convert_labels(:,shuffle1);
```

Figure 15: the code of shuffle the training labels and images

The reason for shuffling the MNIST dataset is that the arrangement of the MNIST dataset after shuffle will have a certain randomness, so that the next sample is equally likely to be of any type of data when read in sequence. In addition, Shuffle prevents overfitting and allows the model to learn more correct features. Therefore, *randperm()* function will generate 60,000 random integers in the range 1 to 60,000 so that the training dataset is shuffled following the random integers.

Step 6: Train the three-layer perception

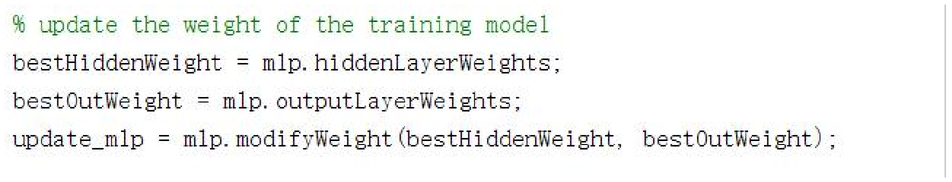


```
for x=1:15
    total_acc=[];
    total_loss=[];
    train_correct=0;
    for j = 1:train_number/batch_size
        batch_images = train_image(:, (j-1)*batch_size+1:j*batch_size);%bacth the train images
        batch_labels = train_label(:, (j-1)*batch_size+1:j*batch_size);
        for i = 1:batch_size
            mlp.backward(batch_images(:,i), batch_labels(:,i), learing_Rate);
            output = mlp.compute_output(batch_images(:,i));
            train_outputs(i+(j-1)*batch_size,:) = output;
            for b=1:10
                if train_outputs(i+(j-1)*batch_size,b) == max(train_outputs(i+(j-1)*batch_size,1:10)
                    pos = find(batch_labels(:,i)==1);%find the target from train labels
                    if b == pos
                        train_correct = train_correct + 1;
                    end
                end
            end
            loss_temp = output-batch_labels(:,i);
            loss = mse(loss_temp);
            total_loss = [total_loss,loss];
            train_acc = train_correct/train_number;
            total_acc = [total_acc,train_acc];
        end
    end
    mean_loss = [mean_loss;mean(total_loss,2)];
    mean_acc = [mean_acc;mean(total_acc,2)];
    fprintf('For Epoch %i,means loss %.5f \n',x,mean(total_loss,2));
end
```

Figure 16: the process of training module

The process of training module can be divided into the following steps. Firstly, considering the MNIST dataset includes 60000 samples, the training dataset will be divided into batches, and use one batch of samples each time to train to update the weights. Meanwhile, batching dataset can reduce the pressure of machine and coverage faster. Secondly, for each batch of data, the module will implement the forward propagation and backward propagation, and each output will be stored to matrixes. Thirdly, the training accuracy of each batch will be calculated. The process of calculating training accuracy is finding the predicted output and the target output, and then compare the predicted and target output whether are same. Fourthly, the method of calculating loss will be chosen MSE. For each loss and training accuracy, it will be store to array. Once one epoch finished, the average loss and training accuracy will be calculated and store.

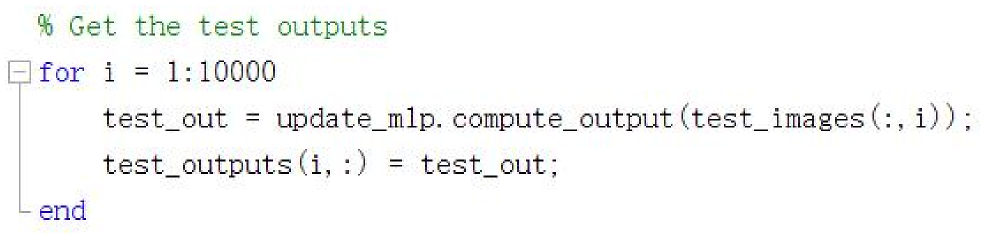
Step 7: Store the updated hidden layer weight and output layer weight



```
% update the weight of the training model
bestHiddenWeight = mlp.hiddenLayerWeights;
bestOutWeight = mlp.outputLayerWeights;
update_mlp = mlp.modifyWeight(bestHiddenWeight, bestOutWeight);
```

Figure 17: the code of updating the trained hidden layer weights and output layer weights

Step 8: Get the test outputs

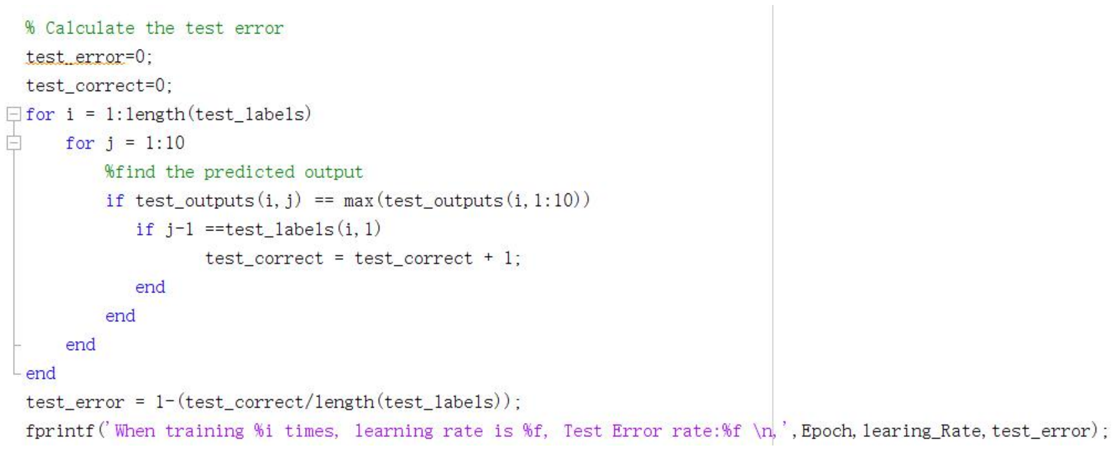


```
% Get the test outputs
for i = 1:10000
    test_out = update_mlp.compute_output(test_images(:, i));
    test_outputs(i, :) = test_out;
end
```

Figure 18: the code of obtaining test output

Using the updated module to obtain the test outputs and store to array

Step 9: Calculate the test accuracy



```
% Calculate the test error
test_error=0;
test_correct=0;
for i = 1:length(test_labels)
    for j = 1:10
        %find the predicted output
        if test_outputs(i, j) == max(test_outputs(i, 1:10))
           if j-1 ==test_labels(i, 1)
                  test_correct = test_correct + 1;
           end
        end
    end
end
test_error = 1-(test_correct/length(test_labels));
fprintf('When training %i times, learning rate is %f, Test Error rate:%f \n',Epoch, learing_Rate, test_error);
```

Figure 19: the code of calculating the test error

The method of calculating the test accuracy is finding the predicted outputs and target outputs and then comparing the results whether are same.

3.Realize and describe an experiment in Matlab

The third part will report the most important experimental results, including the parameter settings of three-layer perception

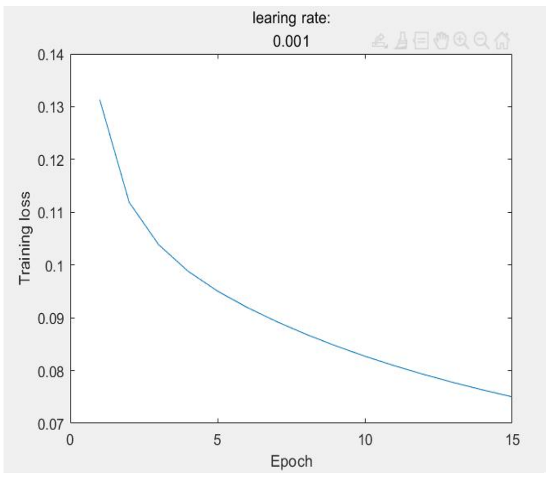
3.a Effect of parameters

3.a.1 Effect of learning rate

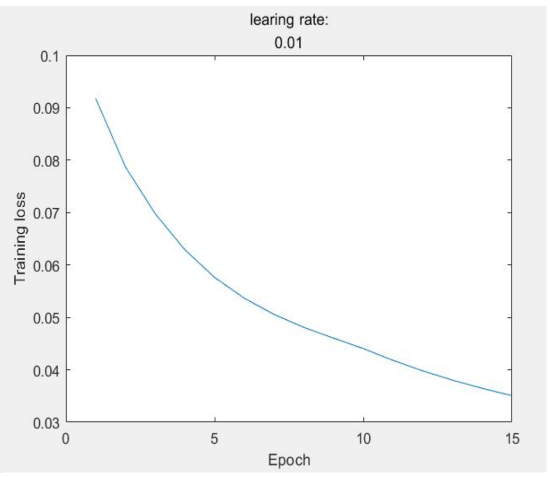
Considering the different learning rate shows the different training effect on MNIST dataset, learning rate will be set to 0.001, 0.01, 0.5, 1 and 1.5 respectively to train model and test data so that find the range of learning rates which show more excellent performance on the designed model.

Learning Rate (Epoch 15, Hidden Dimension 50 Batch Size 100)	**Training Accuracy**	**Testing Accuracy**
0.001	41.96%	56.56%
0.01	82.50%	82.27%
0.5	97.80%	95.74%
1	96.51%	95.21%
1.5	94.38%	93.36%

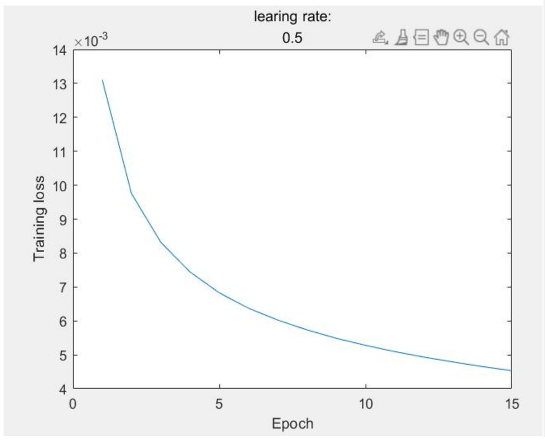
Table 1 : the comparison of training accuracy and testing accuracy on different learning rate

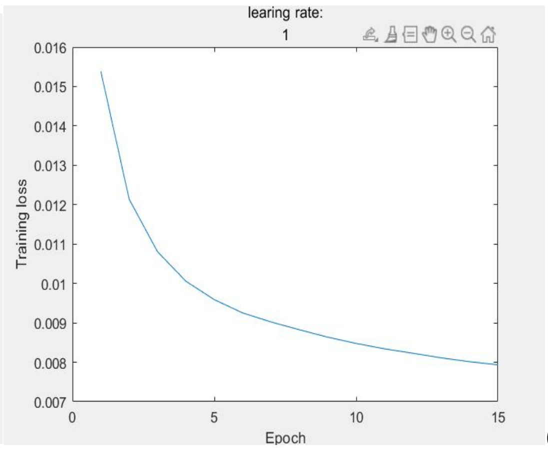


(a) Learning rate: 0.001

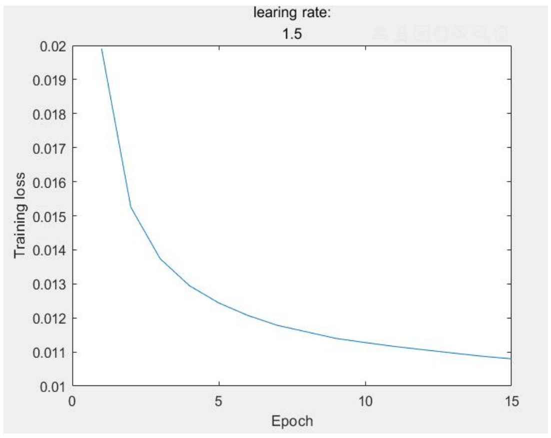


(b) Learning rate: 0.01





(c) Learning rate: 0.5 (d) Learning rate: 1



(e) Learning rate: 1.5

Figure 20: loss of different learning rates

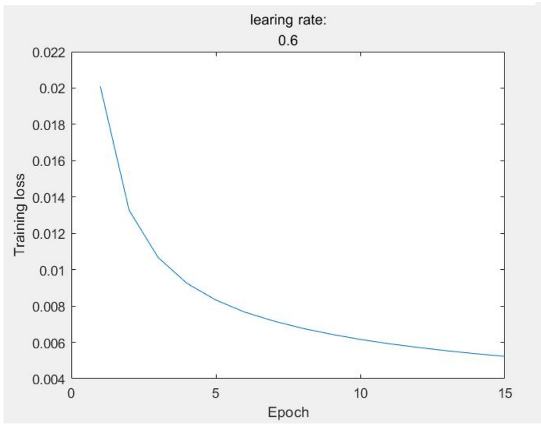
According to table [1] and figure [20], it can be obtained that the loss reduces slowly if the value of learning rate reaches to 0.001 and 0.01 respectively.

Meanwhile, it obtains comparable or even better training, testing accuracy loss performances than others when learning rate reaches to 0.5, and 1.5. Therefore, the more excellent learning rate will be found around 0.5, 1, 1.5.

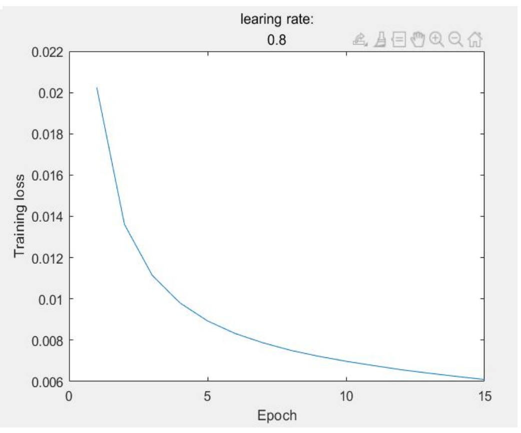
Learning Rate (Epoch 15, Hidden Neurons 50 Batch Size 100)	**Training Accuracy**	**Test Accuracy**
0.6	97.86%	96.30%
0.8	97.53%	94.89%

1.2	96.66%	95.45%
1.4	95.06%	93.83%
1.7	90.92%	90.61%

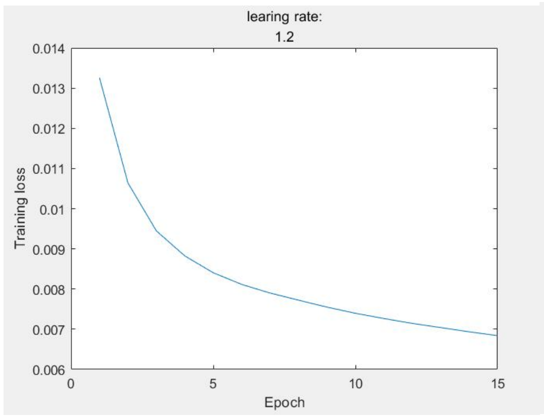
Table 2: The comparison of training accuracy and testing accuracy on different learning rate



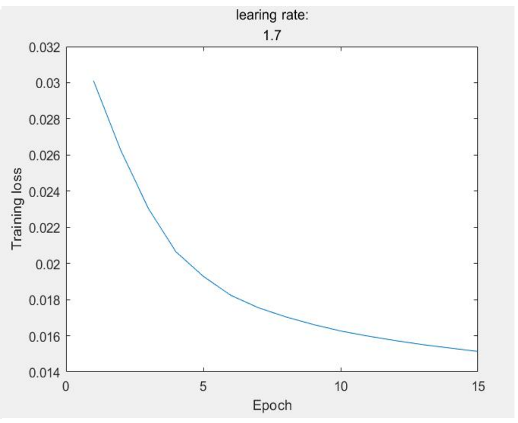
(a) Learning rate: 0.6



(b) Learning rate: 0.8



(c) Learning rate: 1.2



(d) Learning rate: 1.7

Figure 21: the loss on different learning rate

According to table [2] and figure [21], the loss of the model is small but the training accuracy and testing accuracy starts to reduce when the learning rates increase to 1.4 and more than 1.4, which indicates that the model is underfitting. Therefore, the learning rate reaches to 1.4 or higher does not show an excellent learning rate. When learning rate is set as 0.6, from the change of the loss curve, it can be observed that the model converges positively, and the training accuracy and test accuracy show excellent performance. Thus, the learning rate will be taken as 0.6, which is the one successfully parameter.

Based on the experiment of learning rate, for all the next experiments, unless mentioned, otherwise learning rate will be set to 0.6 to explore other parameters.

3.b Effect of hidden neurons

For reducing the range of excellent hidden neurons, hidden neurons will be set to 10, 50, 200 respectively to train model and test data so that find the range of hidden neurons which show more excellent performance on the designed model.

Hidden Neurons (Epoch 15 Batch Size 100)	**Training Accuracy**	**Testing Accuracy**
10	92.61%	90.47%
50	97.86%	96.30%
200	70.30%	68.50%

Table 3: The comparison of training accuracy and testing accuracy on different hidden neurons

According to table [3], it can be concluded that if the number of hidden neurons is set as 200, the training accuracy and testing accuracy shows the most negative performance. This is because there are many hidden neurons so that the module overfits and cannot generalize. In addition, few neurons produce underfits so that the little features can be learned from the three-layer perception while the hidden neurons are set as 10. Therefore, the size of hidden neurons will be set more than 10 and around 50.

Hidden Neurons (Epoch 15 Batch Size 100)	**Training Accuracy**	**Testing Accuracy**
20	96.24%	93.78%
30	97.21%	94.82%
40	97.28%	95.21%
60	88.89%	87.26%

Table 4: The comparison of training accuracy and testing accuracy on different hidden neurons

According to table [4], if hidden layer includes 50 neurons, it will perform more positive result than other number of hidden neurons.

Based on the experiment of hidden neurons, for all the next experiments, unless mentioned, otherwise hidden neurons will be set to 50 to explore other parameters.

3.c Effect of batch size

Based on the above experiment, the excellent batch size will be explored around 100.

Batch Size (Epoch 15)	Training Accuracy	Testing Accuracy
50	97.78%	95.81%
100	97.86%	96.30%
200	88.48%	87.54%

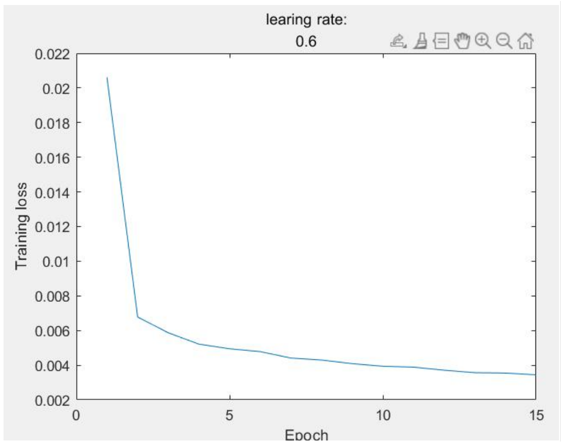
Table 5: The comparison of training accuracy and testing accuracy on different batch size

According to table [5], when batch size reaches to 100, it shows better performances.

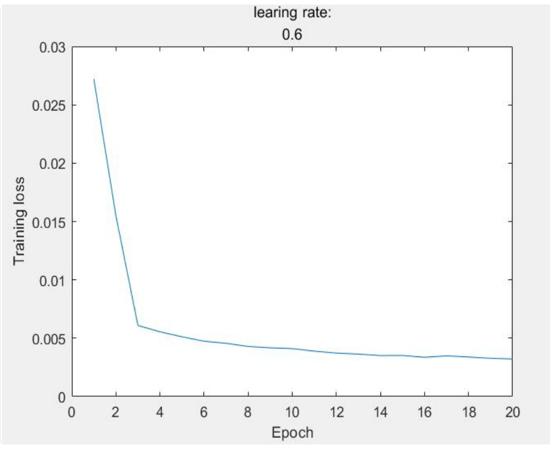
3.d Effect of epoch

Epoch	Training Accuracy	Test Accuracy
15	97.86%	96.30%
20	97.99%	95.85%
30	98.03%	96.14%

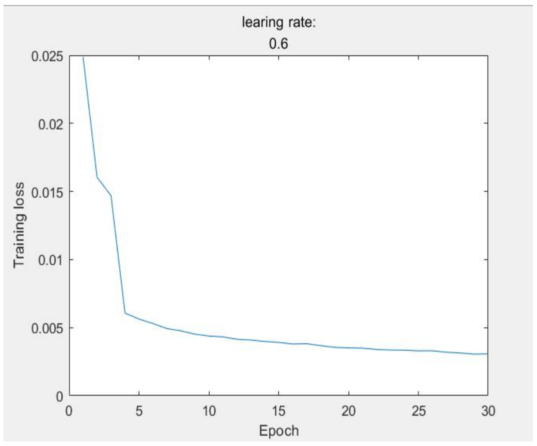
Table 6: The comparison of training accuracy and testing accuracy on different epoch



(a) Epoch: 15



(b) Epoch: 20



(c) Epoch:30

Figure 22: the loss on different epochs

According to table [6] and figure [22], it can be observed that if epoch equals 15, the testing accuracy reach the highest than other epochs.

4. Conclusion

4.a Results

In conclusion, after modifying each parameter and observing the results of loss, training accuracy and testing accuracy, the most excellent value of each parameter shown in table [7].

Learning Rate	Hidden Neurons	Batch Size	Epoch	Training Accuracy	Testing Accuracy
0.6	50	100	15	97.86%	96.30%

Table 7: the most excellent value of each parameter and training accuracy and testing accuracy

Compared with the above three-layer perception, the baseline linear classifier is used to implement, which means each output unit receives a weighted sum from the input unit's pixel values. The categories of input numbers are represented by the output and largest units [3]. The three-layer perception includes activations of non-linear, which has the ability to introduce nonlinear factors to the neurons, allowing the three-layer perception to approximate any nonlinear function at will. The original image is randomly distorted using the affine transform to generate additional images, and the network is then further trained on manually segmented characters from the censored images, and then inserted into the check reading system and trained globally [4]. For the process of before training module, the original images only are be divided by 255, not randomly distorted.

4.b Current limitations and further strategies

For structure of multi-layer perception, it only includes one hidden layer. Compared with two or more hidden layers, one hidden layer might not fit better. Therefore, in the future, the size of hidden layer will be increased to two or more in order to explore the effect of different size of hidden layer.

For activation, the above experiments do not implement and compare the effects of tanh and Softmax activation functions on this network. In future work, the tanh activation function will be implemented, and the Softmax activation function will be acted together with cross entropy loss on the output layer to explore the performances.

For the strategy of training and testing module, it has not validation set. In the future, the validation set will be set to receive feedback through the effect of the validation set, according to the effect to explore whether it is necessary to terminate the current model training, change the parameters and then train in order to finally get the optimal model.

Reference

[1] O. Matan, R. K. Kiang, C. E. Stenard, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, L. D. Jackel and Y. LeCun: Handwritten character recognition using neural network architectures, Proc. of the 4th US Postal Service Advanced Technology Conference, Washington D.C., November 1990,

[2] Y. LeCun, L. Bottou, Y. Bengio and P. Haffner: Gradient-Based Learning Applied to Document Recognition, Intelligent Signal Processing, 306-351, IEEE Press, 2001,

[3] Y. LeCun, L. D. Jackel, L. Bottou, C. Cortes, J. S. Denker, H. Drucker, I. Guyon, U. A. Muller, E. Sackinger, P. Simard and V. Vapnik: Learning Algorithms For Classification: A Comparison On Handwritten Digit Recognition, in Oh, J. H. and Kwon, C. and Cho, S. (Eds), Neural Networks: The Statistical Mechanics Perspective, 261-276, World Scientific, 1995

[4] L. Bottou, Y. LeCun and Y. Bengio: Global Training of Document Processing Systems using Graph Transformer Networks, Proc. of Computer Vision and Pattern Recognition, 490-494, IEEE, Puerto-Rico, 1997