A Feed-Forward Neural Network for Handwritten Digit Recognition

Wang Daming[†, a]

[a]University of Victoria
[b]ECE 503 Optimization for Machine Learning

Abstract This paper shows a small feed-forward neural network for Handwritten Digit Recognition on the MNIST dataset. The code is written in Matlab. I began with the process of loading and preprocessing the data, then convert the digit images into normalized vectors and perform transform class labels into one-hot encodings. The neural network consists of an input layer, a hidden layer, and an output layer. The input layer has 784 input nodes. The hidden layer uses the ReLU activation function, and the output layer uses the softmax function. Together, the model can accurately predict the digit of the handwritten images from 0 to 9. In order to optimize the model, I use forward and backward passes to compute the loss and gradients to apply mini-batch gradient descent for parameters. The model can achieve a high accuracy on both validation and test sets, which shows the effectiveness of the neural network and the optimization algorithm of our model. The result is shown in the format of a confusion matrix. The complete code is available in the appendix.

Keywords: MNIST; Neural Network; Handwritten Digit Recognition.

1. Introduction

Over the past few years, the field of machine learning has seen significant growth, especially in the area of deep learning. Moreover, most deep learning models are based on neural networks. Today, many breakthrough technologies in image classification, speech recognition, and natural language processing are using artificial neural networks (ANNs) (Schmidhuber, 2015; Goodfellow et al., 2016). This model is inspired by the biological neural networks of animal brains and has been applied to many uses like ChatGPT and other AI chat or generative tools. It is basically learning complex mappings from inputs to outputs, with a massive amount of data and appropriate training algorithms. Since Professor Lei Zhao introduced how the neural network works in the last few lectures, it made me wonder what we can do with a simple neural networking model and try to play with it. As for the database, I chose MNIST since most of the models used to process the MNIST dataset

Submitted on December 11, 2024.
[†]wdm1732418365@gmail.com

use logistic regression with a fundamental linear model used as a baseline for classification tasks (LeCun et al., 1998) or use Support Vector Machines with high-dimensional spaces (Cortes and Vapnik, 1995). However, I think it would be great to have a small neural network model implemented to solve the MNIST dataset, which would show a novel way to solve this very old dataset. The MNIST dataset is a collection of handwritten digits from zero to nine, normalized and centered in 28 x 28-pixel images with grayscale. This has become a very popular and standard test set for evaluating various machine learning algorithms and architectures (LeCun et al., 1998).

1.1 Problem Background

The goal of MNIST dataset classification is to map the given input image that contains a 28 x 28 pixel grayscale image of a handwritten digit from zero to nine numbers. This is a supervised learning problem. We now have a large amount of training data with labelled numbers correctly, and we want to find an algorithm to train the model so that it can correspondingly classify the unseen test data accurately. The condition of judging whether the result is accurate or not totally depends on the percentage of test images the model can correctly label. Over the years, many methods like classic k-Nearest Neighbours and SVM to state-of-the-art deep convolutional networks have been benchmarked on MNIST, making it a reliable dataset for algorithmic comparison (LeCun et al., 1998). Among all these methods, artificial neural networks remain an important method because of their flexibility and expressiveness. In this case, we choose to use the feed-forward neural networks, also known as multilayer perceptions, it is one of the earliest and most fundamental classes of neural networks (Rosenblatt, 1958; McCulloch and Pitts, 1943). A feed-forward network with a single hidden layer containing many hidden unites with some nonlinear activation functions like sigmoid or ReLU that can approximate any continuous function on compact subsets of $\mathbb{R}^n$ to arbitrary precision, also known as the Universal Approximation Theorem (Hornik et al., 1989; Cybenko, 1989). The feed-forward network has layers of neurons arranged sequentially. Then, each neuron applies a linear transformation with a weighted summation of inputs plus a bias term and then does a nonlinear activation function to complete the process (Bishop, 2006). This network can not only represent simple linear mappings but also complex functions because of the nonlinearities. Since we are dealing with MNIST, we only use a small network with one hidden layer, which is sufficient.

1.2 Neural Networks and the Optimization Problem

The process of training the neural network is basically an optimization problem. Suppose we have a network with a set of parameters $\theta = \{W_1, b_1, W_2, b_2, \dots\}$, where W_i and b_i are the weights and bias terms of each layer. In the training procedure, the input will be put together with the corresponding true labels. Then, the network will predict and use a loss function to quantify the difference between the predictions and the true label. A good choice is a cross-entropy loss, which directly relates to the probabilistic framework underlying the softmax output layer (Goodfellow et al., 2016). This means if we let (X, Y) represent the dataset with X as a collection of input vectors and Y as the corresponding labelled vectors, we can then define the loss function $L(\theta$ that depends on the network's parameters. The goal of this training is to try to minimize the loss of function value as formula defined:

$$\min_{\theta} L(\theta) = \min_{\theta} \frac{1}{N} \sum_{i=1}^{N} \ell\big(f_\theta(x_i), y_i\big)$$

Here, we have $f_\theta(x_i)$ represents the feed-forward computation of the network on the input x_i. For the classification with a softmax output, we have $\ell(\hat{y}, y) = -\sum_{k=1}^{10} y_k \log(\hat{y}_k)$ here. $\hat{y}$ is the predicted probability distribution over the classes, and the y is the true class distribution.

Although neural network training is often high-dimensional and complex with some local minima and saddle points, etc, empirical results have shown that gradient-based methods are highly effective for neural network training procedure (Rumelhart et al., 1986; LeCun et al., 2012). The idea is to use computed gradients of the loss with respect to the parameters and take the error backward from the output layer to the input layer. Once we have the gradient, we can apply some optimization algorithms like mini-batch Gradient Descent. This method will update parameters using a subset or a batch of the training set at each iteration, and therefore improve the efficiency and often help generalization (Bottou, 2010). Furthermore, more advanced optimizers like Momentum, RMSProp, or Adam can be used to accelerate convergence and help escape suboptimal regions in the parameter space (Kingma and Ba, 2014).

1.3 Model Architecture

The method used is a relatively small feed-forward neural network architecture. The input layer has 784 units, the hidden layer has an order of 100 units, and the output layer has ten units, with one for each digit. In the hidden layer, a nonlinear activation function, the Rectified Linear Unit, is applied to enable the network to model nonlinear decision boundaries. Then, the output layer uses the softmax function to produce a probability distribution over the classes, making it suitable for a classification task (Goodfellow et al., 2016).

2. Problem Formulation as an Optimization Problem

In the process of recognition of handwritten digits using a feed-forward neural network, we need to find parameters that map input images to the correct class labels. For example, consider a dataset $D = \left\{ \left(x^{(i)}, y^{(i)} \right) \right\}_{i=1}^{N}$, where each $x^{(i)} \in \mathbb{R}^d$ is an input vector represent the image, and $y^{(i)} \in \{0, 1, \ldots, 9\}$ is the corresponding class label. Then, we can convert these labels into one-hot-encoded vectors $y^{(i)} \in \mathbb{R}^1 0$, where one of them should be 1, represents the correct class, and 0 for others (LeCun et al., 1998). The network can then be defined as $\theta = \{W_1, b_1, W_2, b_2\}$. W_1 is the weight matrix, and b_1 is the bias vector of the hidden layer. Similarly, W_2 is the weight matrix, and b_2 is the bias vector for the output layer.

When it comes to the computation steps for the feed-forward model, the hidden layer pre-activation has a formula of $z_1^{(i)} = W_1 x^{(i)} + b_1, \quad z_1^{(i)} \in \mathbb{R}^h$ where h is the number of hidden units. Then the ReLU is applied to the hidden layer with $a_1^{(i)} = \sigma\left(z_1^{(i)}\right) \in \mathbb{R}^h$ where $\sigma(z) = \max(z, 0)$ (Goodfellow et al., 2016). In the next step, similar to the hidden layer, the output layer pre-activation formula is $z_2^{(i)} = W_2 x^{(i)} + b_2, \quad z_2^{(i)} \in \mathbb{R}^k$ where k is the class amount 10. In order to get the probability distribution of the 10 classes, we can use the softmax function of:

$$\hat{y}_j^{(i)} = \frac{\exp\left(z_{2,j}^{(i)}\right)}{\sum_{m=1}^{k} \exp\left(z_{2,m}^{(i)}\right)}, \quad \text{for } j = 1, \ldots, k$$

The output $\hat{y}_j^{(i)}$ means the predicted probability that the input x^i belongs to the class $j-1$.

When it comes to the evaluation of predictions, we can use the cross-entropy loss function. For (x^i, y^i), where y^i is one-hot encoded, the function is given by $\ell\left(\hat{y}^{(i)}, y^{(i)}\right) = -\sum_{j=1}^{k} y_j^{(i)} \log\left(\hat{y}_j^{(i)}\right)$ (Goodfellow et al., 2016; Bishop, 2006). Since there's only one correct class classification result, the function can be simplified to $\ell\left(\hat{y}^{(i)}, y^{(i)}\right) = -\log\left(\hat{y}_{j*}^{(i)}\right)$ where $j*$ is the index of the correct class. This will allow the model to continue to assign a high probability to the correct class. Since we have a single loss function, we can then calculate the entire training dataset of N examples using:

$$\min_{\theta} L(\theta) = \frac{1}{N}\sum_{i=1}^{N} \ell\left(\hat{y}^{(i)}, y^{(i)}\right) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{k} y_j^{(i)} \log\left(\hat{y}_j^{(i)}\right)$$

In order to solve this $\min_{\theta}$, we can use gradient-based optimization methods. We can then perform updates using variants of SGD $\theta \leftarrow \theta - \alpha \nabla_{\theta} L(\theta)$, where $\alpha > 0$ is the learning rate. By changing this learning rate, the result will also be different. Too small or too large values of learning rate will all cause a huge decrease in the accuracy result. By doing so, mini-batches of data often can improve computational efficiency and can help the model generalize better (LeCun et al., 1998; Goodfellow et al., 2016). In general, these are the formulation steps of the feed-forward neural network for MNIST classification as a high-dimensional unconstrained optimization problem.

3. Solution Methods

For the detailed steps in training the feed-forward neural network and solving the optimization problem $\min_{\theta}$, we applied gradient-based optimization techniques, including forward pass, backward pass and gradient aggregation procedure. Since the direct computation of partial derivatives will become very complicated in neural networks involving the composition of multiple linear transformations and nonlinear activation functions, backpropagation is introduced by (Rumelhart et al., 1986). This method applies the chain rule in a structured manner. In more detail, this algorithm processes the network from the output layer back toward the input layer. Therefore, intermediate computations will be reused to reduce exponential complexity. As for each step:

1. Forward pass: This is the step to compute the network's output and loss for the given input batch.

2. Backward pass: This is used to compute the gradients of the loss. Then, propagate these gradient results back to each preceding layer using the chain rule to compute gradients with respect to each layer's input and parameters.

3. Grdient aggregation: This is processed to get the full gradient $\nabla_{\theta} L(\theta)$ for the current batch.

After we get the gradients in the above steps, we can then decide on updating the parameters θ. We choose the mini-batch gradient descent (Bottou, 2010) that computes gradients on smaller batches of the data to increase the speed and reduce resource usage on these kinds of large datasets. Moreover, this method can provide a beneficial noise that helps escape suboptimal local minima and avoid overfitting (Hardt et al., 2016; Keskar et al., 2017). Therefore, one single SGD iteration can be then calculated by $\theta \leftarrow \theta - \alpha \nabla_\theta L_{\text{batch}}(\theta)$ where α is the learning rate and $L_{\text{batch}}(\theta)$ is the loss computed on a small batch of the training data. Finally, after each training epoch, we evaluate performance on a validation set and testing sets. And also calculate the confusion matrix as previous lab experiments done similarly.

In summary, this procedure should provide an effective and accurate result for neural network training. However, some future works can also be done by using adaptive optimizers like Adam (Kingma and Ba, 2014) to compute the moving averages of first and second moments of gradients and use these statistics to adapt the parameter-wise learning rates so that it can find a perfect learning rate value that maximizes the accuracy. Moreover, applying regularization techniques like L2 Regularization that adds a penalty term $\lambda \|W\|_2^2$ can further improve generalization (Goodfellow et al., 2016).

4. Computer Simulations and Numerical Results

The dataset contains 60,000 training images and 10,000 test images, with each image being a 28 x 28-pixel grayscale representing a handwritten digit from 0 to 9, which means an input vector of 784 dimensions. The data preprocess normalizes the pixel values into [0, 1], and the labels are converted into one-hot-encoded vectors of length 10. Then, the neural network takes the input layer of size 784, a hidden layer with 100 ReLU-activated units, and outputs 10 softmax units. After that, mini-batch gradient descent with a batch size of 128 and a learning rate of 0.87 will be used to train for several epochs. The reason for choosing 0.87 is that after trying some learning rates manually, 0.1, 0.5, 0.8, and 0.9, we tested out that 0.87 would give a higher accuracy result. Finally, the cross-entropy loss was used as the objective function. Gradients were computed via backpropagation, and parameters were updated with vanilla gradient descent using mini-batches. The final output result shows:

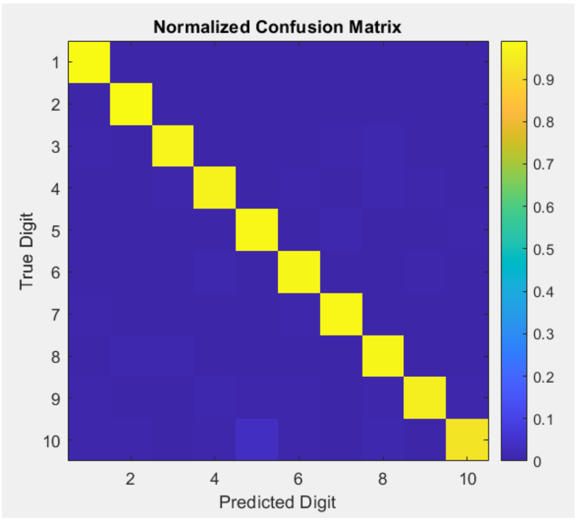
```
Epoch 1: Train Loss = 0.1412 | Val Loss = 0.1168 | Val
    Acc = 96.40%
Epoch 2: Train Loss = 0.1053 | Val Loss = 0.0995 | Val
    Acc = 97.18%
Epoch 3: Train Loss = 0.0941 | Val Loss = 0.0977 | Val
    Acc = 97.06%
Epoch 4: Train Loss = 0.0567 | Val Loss = 0.0797 | Val
    Acc = 97.42%
Epoch 5: Train Loss = 0.0577 | Val Loss = 0.0808 | Val
    Acc = 97.60%
```

The average training loss, validation loss, and validation accuracy were reported in each epoch of the training process. The improvements in accuracy show that the model is learning to generalize. Noticeably, by the fifth epoch, the validation accuracy has reached 97.60%. This shows the model works great on the validation sets and, ideally, would work great on the test sets. It turns out that the final test accuracy is:

```
Test Loss = 0.0945 | Test Accuracy = 97.26%
```

The result of 97.26% is quite competitive. It shows that a basic feed-forward neural network can produce accurate results on the MNIST dataset. As we look deeper at the model performance using the confusion matrix, it shows as below:

```
Confusion Matrix:
      968        0        3        1        1        1        2        1        3        0
        0     1124        3        0        0        2        3        1        2        0
        6        3     1003        1        3        0        4        9        3        0
        0        0        7      980        0        7        1        8        4        3
        0        1        1        0      964        0        9        2        1        4
        1        1        0        8        2      872        3        1        4        0
        4        3        1        1        3        4      939        1        2        0
        1        8        8        2        1        0        0     1005        1        2
        4        3        3        8        7        5        3        5      932        4
        4        6        0        6       38        6        0       10        0      939
```



It appears that most misclassification is relatively sparse. Most of the digits can be correctly classified, but some digits are still often confused with others. For example, number 4 and 9 have similar shapes and indeed show 38 misclassification cases, which is higher than others. This also happens on number 7 and 9 with 10 errors. Therefore, some more work should be done, as mentioned in the previous section, to further improve the liability and ability to distinguish similar shape numbers to achieve better accuracy.

References

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*, Springer.

Bottou, L. (2010). Large-scale machine learning with stochastic gradient descent, *Proceedings of the COMPSTAT'2010*, Springer, pp. 177–186.

Cortes, C. and Vapnik, V. (1995). Support-vector networks, *Machine Learning* **20**(3): 273–297.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function, *Mathematics of Control, Signals and Systems* **2**(4): 303–314.

Goodfellow, I., Bengio, Y. and Courville, A. (2016). *Deep Learning*, MIT Press.

Hardt, M., Recht, B. and Singer, Y. (2016). Train faster, generalize better: Stability of stochastic gradient descent, *International Conference on Machine Learning (ICML)*, pp. 1225–1234.

Hornik, K., Stinchcombe, M. and White, H. (1989). Multilayer feedforward networks are universal approximators, *Neural Networks* **2**(5): 359–366.

Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M. and Tang, P. T. P. (2017). On large-batch training for deep learning: Generalization gap and sharp minima, *International Conference on Learning Representations (ICLR)* .

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization, *arXiv preprint arXiv:1412.6980* .

LeCun, Y., Bottou, L., Bengio, Y. and Haffner, P. (1998). Gradient-based learning applied to document recognition, *Proceedings of the IEEE* **86**(11): 2278–2324.

LeCun, Y., Bottou, L., Orr, G. B. and Müller, K.-R. (2012). Efficient backprop, pp. 9–48.

McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity, *Bulletin of Mathematical Biophysics* **5**: 115–133.

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain, *Psychological Review* **65**(6): 386–408.

Rumelhart, D. E., Hinton, G. E. and Williams, R. J. (1986). Learning representations by back-propagating errors, *Nature* **323**(6088): 533–536.

Schmidhuber, J. (2015). Deep learning in neural networks: An overview, *Neural Networks* **61**: 85–117.

A. Matlab Code

The complete Matlab Code used in this report is listed below:

loadMNISTImages.m

```matlab
% Load MNIST images from a IDX3-ubyte file
function images = loadMNISTImages(filename)
    % open file with big-endian format
    file = fopen(filename,'r','b');
    % use first 4 bytes to chekc the file type
    if fread(file,1,'int32') ~= 2051
        error('Invalid magic number in %s', filename);
    end
    numImages = fread(file,1,'int32');
    numRows = fread(file,1,'int32');
    numCols = fread(file,1,'int32');
    fprintf('Loading %d images of size %d x %d from %s\n', numImages, numRows, numCols, filename);
    % each pixel is stored as an unsigned byte
    images = fread(file,inf,'unsigned char');
    images = reshape(images, numRows*numCols, numImages);
    images = double(images)/255;
    fclose(file);
    fprintf('Successfully loaded %d images from %s.\n', numImages, filename);
end
```

loadMNISTLabels.m

```matlab
% Load MNIST labels from a IDX1-ubyte file
function labels = loadMNISTLabels(filename)
    fid = fopen(filename,'r','b');
    if fread(fid,1,'int32') ~= 2049
        error('Invalid magic number in %s', filename);
    end
    % read the next 4 bytes
    numLabels = fread(fid,1,'int32');
    labels = fread(fid,inf,'unsigned char');
    fclose(fid);
    fprintf('Loading %d labels from %s\n', numLabels, filename);
end
```

foward_pass.m

```matlab
function [Y_hat, cache] = forward_pass(X, W1, b1, W2, b2
    )
    % X: input
    % 784 * N

    % linear combination for hidden layer neurons
    % 100 * N
    Z1 = W1 * X + b1;

    % ReLU activation for hidden layer
    A1 = max(Z1, 0);

    % linear combination for output layer neurons
    % 10 * N
    Z2 = W2 * A1 + b2;

    % softmax function that convert Z2 to probabilities
    exps = exp(Z2 - max(Z2, [], 1));
    Y_hat = exps./sum(exps, 1);

    % store Z1, A1, Z2 for backward_pass function
    cache.Z1 = Z1;
    cache.A1 = A1;
    cache.Z2 = Z2;
end
```

backward_pass.m

```matlab
function [loss, dW1, db1, dW2, db2] = backward_pass(X, Y
    , W1, b1, W2, b2)
    % get the predicted probabilities and Z1, A1, Z2
    from forward_pass
    [Y_hat, cache] = forward_pass(X, W1, b1, W2, b2);

    % average the loss and gradients over all samples
    N = size(X,2);

    % compute cross-entropy loss
    loss = -sum(sum(Y .* log(Y_hat+1e-12)))/N;

    % gradient of loss of Z2 for output layer
    % 10 * N
    dZ2 = (Y_hat - Y);
    % gradient of W2
    % 10 * 100
    dW2 = (dZ2 * cache.A1')/size(X,2);
    % 10 * 1
    db2 = mean(dZ2,2);
    % 100 * N
    dA1 = W2' * dZ2;
    % ReLU derivative
    dZ1 = dA1 .* (cache.A1>0);
    % 100 * 784
    dW1 = (dZ1 * X')/size(X,2);
    % 100 * 1
    db1 = mean(dZ1,2);
end
```

compute_loss.m

```
function [loss, acc] = compute_loss(X, Y, W1, b1, W2, b2)
    [Y_hat, ~] = forward_pass(X, W1, b1, W2, b2);
    N = size(X,2);
    loss = -sum(sum(Y.*log(Y_hat+1e-12)))/N;
    [~, pred] = max(Y_hat, [], 1);
    [~, labels] = max(Y, [], 1);
    acc = mean(pred == labels);
end
```

project.m

```
% load training and test data
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');

% convert labels to one-hot encoding:
numClasses = 10;
Y_train = zeros(numClasses, length(train_labels));
for i = 1:length(train_labels)
    Y_train(train_labels(i)+1, i) = 1;
end
Y_test = zeros(numClasses, length(test_labels));
for i = 1:length(test_labels)
    Y_test(test_labels(i)+1, i) = 1;
end
```

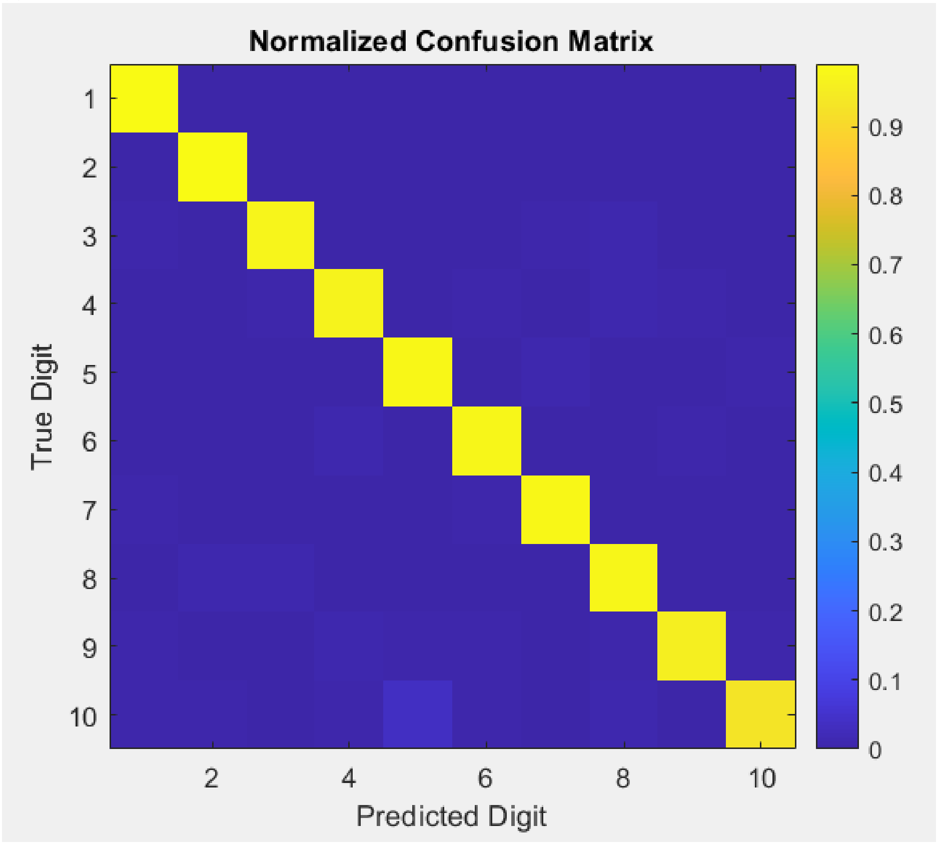
```

X_train = train_images; % 784 * 60000
X_test = test_images; % 784 * 10000

% create a validation set from training data to verify
    the correctness on
% training sets to have a basic idea of accuracy.
val_size = 5000;
X_val = X_train(:, end-val_size+1:end);
Y_val = Y_train(:, end-val_size+1:end);
X_train = X_train(:, 1:end-val_size);
Y_train = Y_train(:, 1:end-val_size);

% initialize Neural Network Parameters
% We have 28x28 = 784
inputSize = 784;
hiddenSize = 100;
outputSize = 10;

% make sure the random numbers generated are the same
    every time
rng('default');
% W1 is a matrix of size hiddenSize * inputSize
% initialize some small random values and scaled by 0.01
W1 = 0.01*randn(hiddenSize, inputSize);
b1 = zeros(hiddenSize, 1);
W2 = 0.01*randn(outputSize, hiddenSize);
b2 = zeros(outputSize, 1);

% Mini-Batch Gradient Descent Training
numEpochs = 5;
batchSize = 128;
% change the learning_rate
learning_rate = 0.87;
fprintf('Learning Rate is %f \n', learning_rate);
numTrain = size(X_train, 2);

for epoch = 1:numEpochs
    % shuffle training data
    idx = randperm(numTrain);
    X_train = X_train(:, idx);
    Y_train = Y_train(:, idx);

    % loop over mini-batches
    for start_i = 1:batchSize:numTrain
        end_i = min(start_i+batchSize-1, numTrain);
        X_batch = X_train(:, start_i:end_i);
        Y_batch = Y_train(:, start_i:end_i);
        [loss, dW1, db1, dW2, db2] = backward_pass(
    X_batch, Y_batch, W1, b1, W2, b2);
        W1 = W1 - learning_rate * dW1;
        b1 = b1 - learning_rate * db1;
        W2 = W2 - learning_rate * dW2;
        b2 = b2 - learning_rate * db2;
    end

    % compute loss on training and validation set
    [train_loss, ~] = compute_loss(X_train, Y_train, W1,
     b1, W2, b2);
    [val_loss, val_acc] = compute_loss(X_val, Y_val, W1,
     b1, W2, b2);
    fprintf('Epoch %d: Train Loss = %.4f | Val Loss =
    %.4f | Val Acc = %.2f%%\n', ...
        epoch, train_loss, val_loss, val_acc*100);
end

% evaluate on test sets
[test_loss, test_acc] = compute_loss(X_test, Y_test, W1,
     b1, W2, b2);
fprintf('Test Loss = %.4f | Test Accuracy = %.2f%%\n',
    test_loss, test_acc*100);

% test set confusion matrix
[Y_hat_test, ~] = forward_pass(X_test, W1, b1, W2, b2);
[~, pred_test] = max(Y_hat_test, [], 1);
[~, true_test] = max(Y_test, [], 1);
confMat = confusionmat(true_test, pred_test);
disp('Confusion Matrix:');
disp(confMat);

% normalize the confusion matrix
confMatNorm = bsxfun(@rdivide, confMat, sum(confMat,2));
imagesc(confMatNorm);
colorbar;
title('Normalized Confusion Matrix');
xlabel('Predicted Digit');
ylabel('True Digit');
axis square;
```

Figure A1
Confusion Matrix

(a) Confusion Matrix with numbers

```
Confusion Matrix:
         968           0           3           1           1           1           2           1           3           0
           0        1124           3           0           0           2           3           1           2           0
           6           3        1003           1           3           0           4           9           3           0
           0           0           7         980           0           7           1           8           4           3
           0           1           1           0         964           0           9           2           1           4
           1           1           0           8           2         872           3           1           4           0
           4           3           1           1           3           4         939           1           2           0
           1           8           8           2           1           0           0        1005           1           2
           4           3           3           8           7           5           3           5         932           4
           4           6           0           6          38           6           0          10           0         939
```

(b) Normalized Confusion Matrix


Normalized Confusion Matrix
True Digit
Predicted Digit