Programming Assignment 2

Name: Mohaiminul Al Nahian
RIN: 662026703
Course no.: ECSE 6850

Introduction

In this programming assignment, our goal is to implement techniques to learn a deep neural network with two hidden layers for performing image digit classification. The MNIST dataset has been used as the training and testing dataset. The classifier will take an image of a hand-written numerical digit between 0 and 9 as input and classify it into one of 10 classes corresponding to digit 0 to 9 respectively. The neural network has been trained by minimizing the Negative Log Likelihood or cross-entropy loss function of the output nodes. We tune our parameters by minimizing the loss function over several epochs by gradient descent approach. It is computationally inefficient to calculate gradients of all training samples at a time. So, We have implemented Stochastic Gradient Descent technique. We have also applied L1 norm regularization of parameters to prevent the model from overfitting. In order to apply gradient descent, we apply backpropagation method, where the gradient of the output is first computed and the gradients pass to the previous layers and finally we update the parameters after each iteration
This report is organised in the following way. We will discuss the architecture of the model, hyper-parameters and experimental set-up, theoretical development of neural network with backpropagation etc. Then the experimental results and performance of the model is described. We also include an appendix with the codes used in this experimentation.

1 Section 1: Theoretical Development

1.1 Theory

For a given training dataset $D = \{x(m), y(m)\}, m = 1, 2, 3, 4....M$, where M is the total number of data, our goal is to classify each input x into one of K classes where $K > 2$. We design a neural network and train the model through stochastic gradient descent to learn the parameters Θ, where Θ =[W_1, W_{10}, W_2, W_{20}, ...] etc
To encode the label, we use 1-of-K encoding, where we use a Kx1 binary vector y, which contains a single 1 for element k (the correct class) and 0 elsewhere.
In order to learn the parameter Θ, we first define Negative Log-Likelihood Loss function. then we apply SGD to minimize the loss function. The loss function and process of gradient calculation with backpropagation to update parameters are shown below:

1.2 Loss Function

The negative conditional log likelihood loss function as the loss for a single training sample $(x[m], y[m])$, is defined as follows:

$$
\begin{aligned}
l(y[m], \hat{y}[m]) &= -\log p(y[m] \mid x[m]) \\
&= -\log \prod_{k=1}^{K} p(y[m][k] = 1 \mid x[m])^{y[m][k]} \\
&= -\sum_{k=1}^{K} y[m][k] * \log(p(y[m][k] = 1 \mid x[m])) \\
&= -\sum_{k=1}^{K} y[m][k] * \log(\hat{y}[m][k])
\end{aligned}
$$

We calculate the loss for all samples in an iteration and then sum them to get the total loss.

1.3 Neural Network Architecture

The neural network that we use has 784 input nodes, based on the size of the image. It has two hidden layers, each containing 100 nodes. Then the fourth layer is the output layer containing 10 nodes, corresponding to 10 classes of digits from 0 to 9. The activation function for the hidden layers is ReLU and for the output layer, it is softmax. The visual representation of the network is given below:

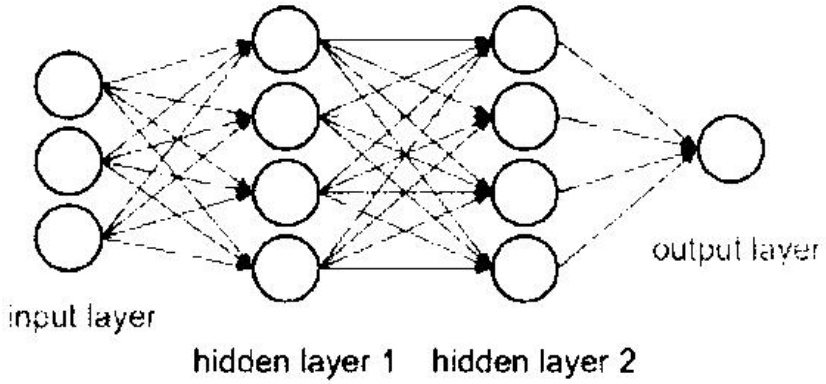


Figure 1: Neural Network Architecture

1.4 Activation Functions

We use ReLU activation function for the hidden layers and softmax activation for the output layers. These are described as follows:
ReLU:

$$\varphi(x) = \max(0, x)$$

Softmax:

$$\sigma_M\left(f_k(x)\right) = \frac{\exp\left(f_k(x)\right)}{\sum_{j=1}^{K}\exp\left(f_j(x)\right)}$$

1.5 Forward Pass

In forward propagation, we use the linear combination of the weight matrix W^1, input vector x[m] and bias vector W_0^1 to compute z^1. We compute H^1 by applying the ReLU activation function to z^1. Similarly, we compute z^2, H^2 and z^3. Finally, we apply softmax activation function to z^3 to get the output class probability given an input image $x[m]$. The equations are given below:

$$z^1 = \left(W^1\right)^T x[m] + W_0^1 \tag{1}$$
$$H^1[m] = \text{ReLU}\left(z^1\right) \tag{2}$$
$$z^2 = \left(W^2\right)^T H^1[m] + W_0^2 \tag{3}$$
$$H^2[m] = \text{ReLU}(z^2) \tag{4}$$
$$z^3 = \left(W^3\right)^T H^2[m] + W_0^3 \tag{5}$$
$$\hat{y}[m] = \sigma_M\left(z^3\right) \tag{6}$$

1.6 Backpropagation

At first, we compute the gradient at output layer $\nabla\hat{y}$. It is defined as,

$$\nabla\hat{y}[m] = \frac{\partial l(y[m], \hat{y}[m])}{\partial\hat{y}} = \begin{pmatrix} \frac{y[m][1]}{\hat{y}[m][1]} \\ \frac{y[m][2]}{\hat{y}[m][2]} \\ \vdots \\ \frac{y[m][K]}{\hat{y}[m][K]} \end{pmatrix}$$

Then, we compute the gradient of weights and output of each layer using the gradient chain rule successively. We use gradient chain rule to recursively compute the gradient of weight matrix at each layer. In reality, the gradient of output layer is passed to a previous layer to get the gradient of that layer. For layers, l = 3 to 0, where 0 is the input layer, we compute

$$\nabla W^l = \frac{\partial\hat{y}}{\partial W^l}\nabla\hat{y}$$
$$\nabla W_0^l = \frac{\partial\hat{y}}{\partial W_0^l}\nabla\hat{y}$$

Applying the gradient chain rule in the output layer vector, we get

$$\begin{aligned}
\nabla W_0^3[m] &= \frac{\partial z(m)}{\partial W_0^3}\frac{\partial \sigma_M(z(m))}{\partial z(m)}\nabla \hat{y}[m] \\
&= \frac{\partial \sigma_M(z(m))}{\partial z(m)}\nabla \hat{y}[m] \\
\nabla W^3[m] &= \frac{\partial z(m)}{\partial W^3}\frac{\partial \sigma_M(z(m))}{\partial z(m)}\nabla \hat{y}[m] \\
&= H^2\frac{\partial \sigma_M(z(m))}{\partial z(m)}\nabla \hat{y}[m] \\
\nabla H^2[m] &= \frac{\partial z(m)}{\partial H^2}\frac{\partial \sigma_M(z(m))}{\partial z(m)}\nabla \hat{y}[m] \\
&= W^3\frac{\partial \sigma_M(z(m))}{\partial z(m)}\nabla \hat{y}[m]
\end{aligned}$$

Now, we need to compute the values of $\frac{\partial \sigma_M(z(m))}{\partial z(m)}$, as follows using matrix differentiation,

$$\frac{\partial \sigma_M(z(m))}{\partial z(m)} = \begin{bmatrix} \frac{\partial \sigma_M(z(m)[1])}{\partial z(m)[1]} & \cdots & \frac{\partial \sigma_M(z(m)[K])}{\partial z(m)[1]} \\ \frac{\partial \sigma_M(z(m)[1])}{\partial z(m)[2]} & \cdots & \frac{\partial \sigma_M(z(m)[K])}{\partial z(m)[2]} \\ \vdots & \ddots & \vdots \\ \frac{\partial \sigma_M(z(m)[1])}{\partial z(m)[K]} & \cdots & \frac{\partial \sigma_M(z(m)[K])}{\partial z(m)[K]} \end{bmatrix}$$

$$= \begin{bmatrix} \sigma_M(z(m)[1])\left(1-\sigma_M(z(m)[1])\right) & \cdots & -\sigma_M(z(m)[K])\sigma_M(z(m)[1]) \\ -\sigma_M(z(m)[1])\sigma_M(z(m)[2]) & \cdots & -\sigma_M(z(m)[K])\sigma_M(z(m)[2]) \\ \vdots & \ddots & \vdots \\ -\sigma_M(z(m)[1])\sigma_M(z(m)[K]) & \cdots & \sigma_M(z(m)[K])\left(1-\sigma_M(z(m)[K])\right) \end{bmatrix}$$

The gradients of the weight matrix in the hidden layer can be computed as:

$$\begin{aligned}
\nabla W_0^2 &= \frac{\partial H^2[m]}{\partial W_0^2}\nabla H^2[m] \\
&= \frac{\partial z(m)}{\partial W_0^2}\frac{\partial \varphi(z(m))}{\partial z(m)}\nabla H^2[m] \\
&= \frac{\partial \varphi(z(m))}{\partial z(m)}\nabla H^2[m] \\
\nabla W^2 &= \frac{\partial H^2[m]}{\partial W^l}\nabla H^2[m] \\
&= \frac{\partial z(m)}{\partial W^2}\frac{\partial \varphi(z(m))}{\partial z(m)}\nabla H^2[m] \\
\nabla H^1[m] &= \frac{\partial z(m)}{\partial H^1} \\
&= W^2\frac{\partial \varphi(z(m))}{\partial z(m)}\nabla H^2[m]
\end{aligned}$$

Let $\varphi(x)$ be the ReLU activation function. Here,

$$H^1[m] = \varphi(z[m]) = \max(0, \text{z[m]})$$

Here,

$$\frac{\partial \varphi(z[m])}{\partial z(m)} = \begin{bmatrix} \frac{\partial \varphi(z(m))[1])}{\partial z(m)[1]} & \cdots & \frac{\partial \varphi(z(m))[N2])}{\partial z(m)[1]} \\ \frac{\partial \varphi(z(m))[1])}{\partial z(m)[2]} & \cdots & \frac{\partial \varphi(z(m))[N2])}{\partial z(m)[2]} \\ \vdots & \ddots & \vdots \\ \frac{\partial \varphi(z(m))[1])}{\partial z(m)[N2]} & \cdots & \frac{\partial \varphi(z(m))[N2])}{\partial z(m)[N2]} \end{bmatrix}$$

where,

$$\frac{\partial \varphi(z(m))[i])}{\partial z(m)[j]} = \begin{cases} 1, \text{ if } i = j \text{ and } z[m][i] > 0 \\ 0, \text{ else} \end{cases}$$

Now, to compute $\frac{\partial z(m)}{\partial W^2}$ we can expand this 3D tensor of shape $\mathrm{N}_1 \times \mathrm{N}_2 \times \mathrm{N}_2$ and show that.

$$\frac{\partial z(m)[i]}{\partial W_j^2} = \frac{\partial \left(\left(W_i^2\right)^T H^1[m] + W_0^2[i] \right)}{\partial W_j^2} = \begin{cases} H^1[m], \text{ if } i = j \\ 0, \text{ otherwise} \end{cases}$$

In a similar way of gradient chain rule, we can go up to the input layer to compute ∇W^1 and ∇W_0^1

1.7 L1 Regularization

The gradient of L1 norm of W^l is given by

$$\nabla R_{L1}(W^l) = \begin{bmatrix} 0 & sign(W^l[1][j^*]) & \cdots & 0 \\ 0 & sign(W^l[2][j^*]) & \cdots & 0 \\ \vdots & \ddots & \vdots & \\ 0 & sign(W^l[N_{l-1}][j^*]) & \cdots & 0 \end{bmatrix}$$

Here, j^* is the index of the column vector with the largest L1 norm. so, $j^* = argmax(\sum_{i=1}^{N_{l-1}} |W^l[i][j]|)$. For regularization on W_0^l , we can use the vector regularization equations to derive its L1 norm and get the derivative

1.8 Stochastic Gradient Descent and weight update

When number of training sample is big, it is computationally inefficient to calculate gradients of loss function of all the samples. Rather, stochastic gradient descent can handle this situation without hampering the end result performance. Instead of all the samples, mini-batches D_b of 2 to 100 samples are selected in each iteration for loss and gradient calculation. Parameters are updated based on this gradient. In our case, 50 samples are chosen as mini-batch and SGD. The weight update equations are as follows:

$$W^l[t] = W^l[t-1] - \eta \times \left(\nabla W^l[t-1] + \lambda \times \nabla R_{L1}\left(W^l[t-1]\right)\right)$$
$$W_0^l[t] = W_0^l[t-1] - \eta \left(\nabla W_0^l[t-1] + \lambda \times \nabla R_{L1}(W_0^l[t-1])\right)$$

Here, η is the learning rate and λ is the regularization parameter

2 Section 2: Experimental Setup

2.1 Parameter initialization

Weights are initialized randomly from a normal distribution distribution with std 0.1 and biases are initialized as all 0.

2.2 Hyper-parameters and Training Set-up

The hyper-parameters are chosen through trial and error by running the codes for several epochs to see the trend of gradient descent. The learning rate η and regularization parameter λ are chosen to be 0.05 and 0.001 respectively. The batch size is 50. The training is performed for 30 epochs. In each epoch, all the training samples are used after being divided into several mini-batches with respect to batch size. In each iteration, a minibatch is chosen for SGD and the parameters are updated. An epoch ends when all the mini-batches have been used once in the training iterations

2.3 Saving the Learned Parameters

After running the training for 30 epochs, we get the updated weight and bias parameters. The weights after epoch 30 are converted from tensor to numpy array as per the instruction of the assignment and then saved using *pickle.dump* after the training finishes.

2.4 Coding Framework

Coding has been done in python with tensorflow v2.0. Other major libraries used are numpy and matplotlib.

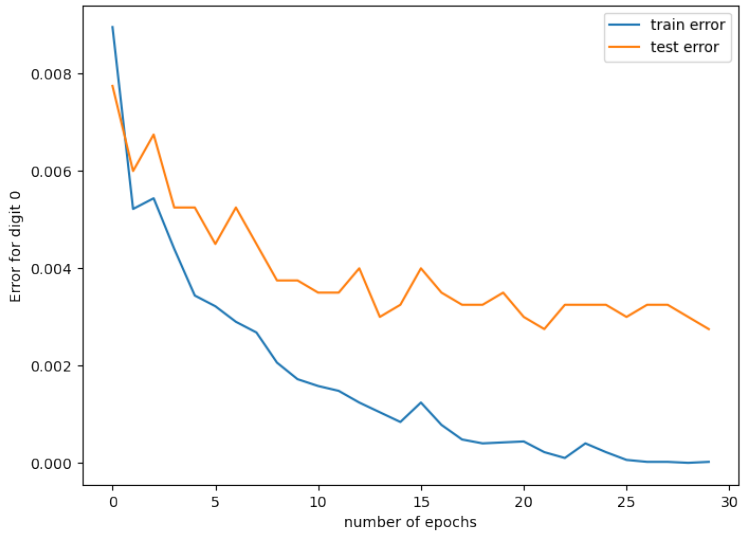
3 Section 3: Result Analysis

3.1 Classification Errors and Accuracies

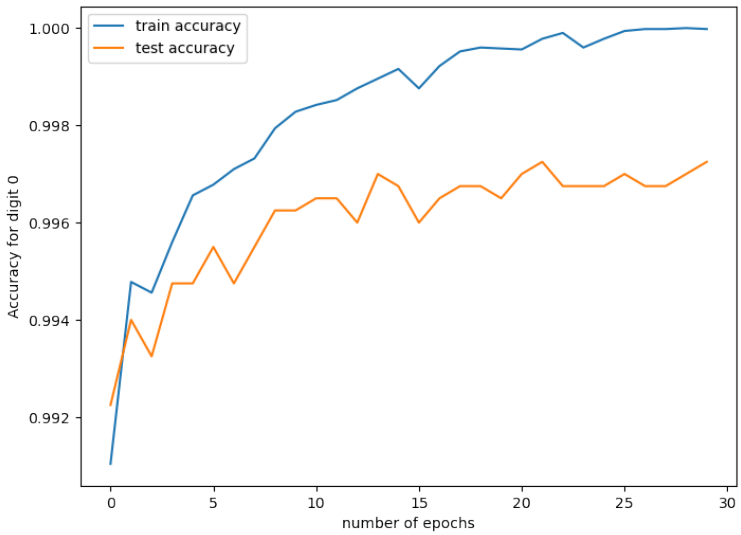
Using the best learned parameters we calculate the training errors, test error, training accuracy and test accuracy. The results are shown below:

item name	0	1	2	3	4	5	6	7	8	9	Average
Train Err	2e-5	1.8e-4	6e-5	4e-5	2e-5	2e-5	2e-5	1.6e-4	1.2e-4	1.2e-4	7.6e-5
Train Acc	0.99	0.99	0.98	0.998	0.99	0.99	0.998	0.99	0.99	0.99	0.99
Test Err	0.003	0.002	0.004	0.007	0.004	0.008	0.005	0.004	0.006	0.007	0.0049
Test Acc	0.997	0.998	0.996	0.993	0.996	0.993	0.995	0.995	0.994	0.992	0.995

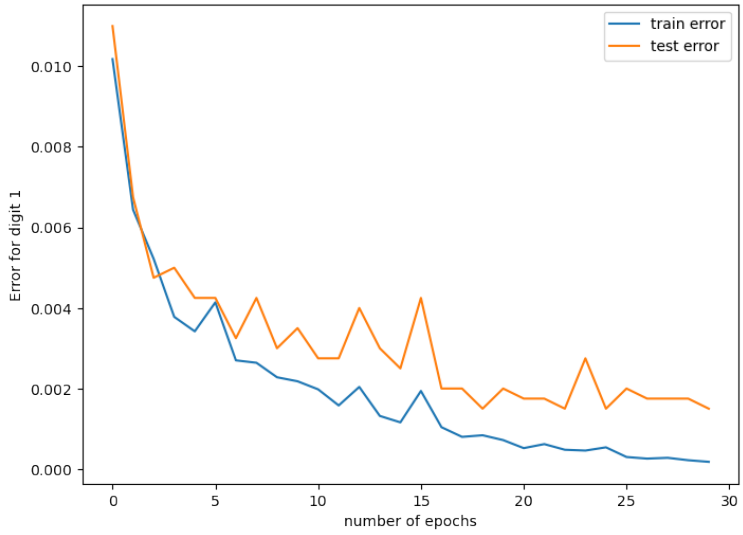
The overall training accuracy is 99.99%. Whereas, overall test accuracy is 99.51% and overall test error is 0.49%. The relevant plots are shown below



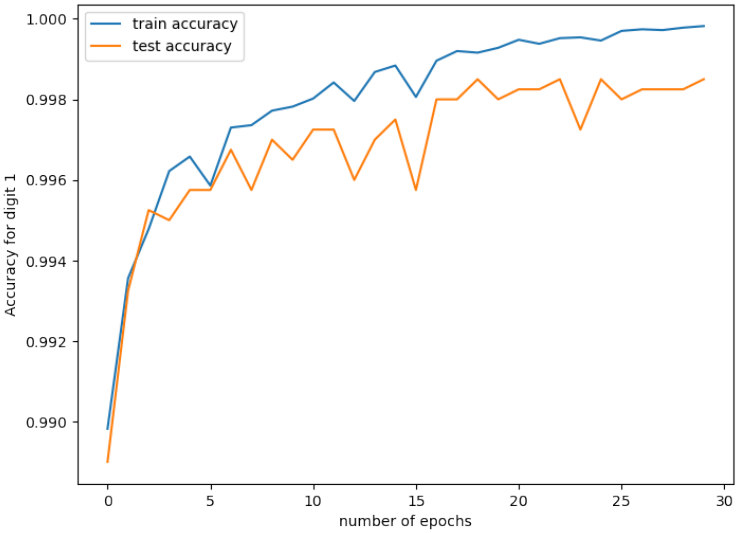
(a) Error for Digit 0



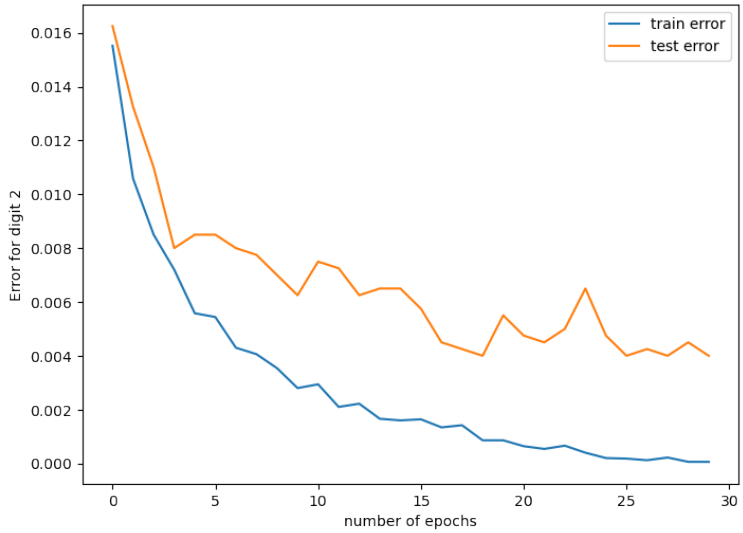
(b) Accuracy for Digit 0



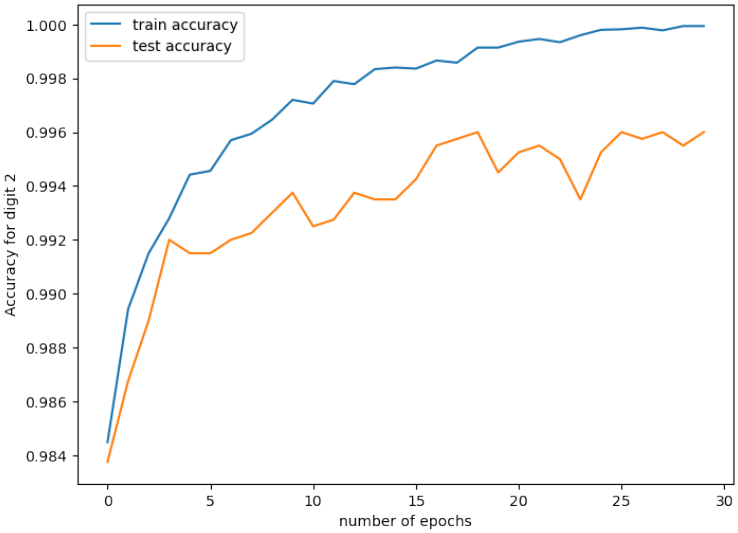
(a) Error for Digit 1



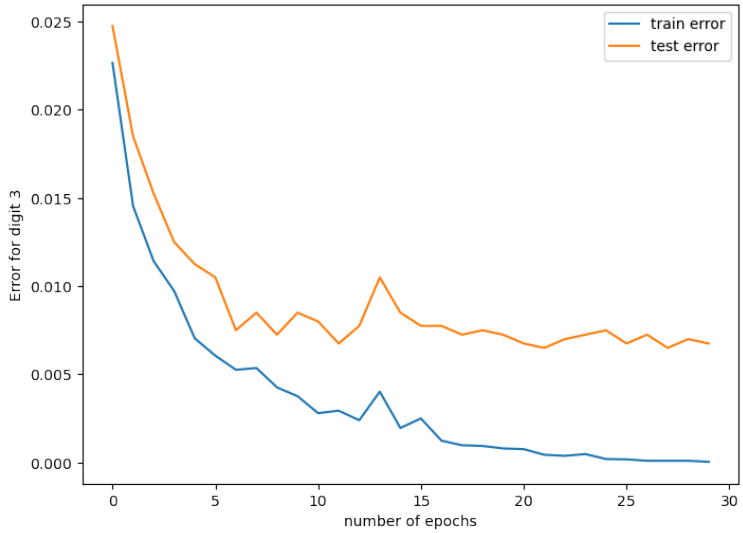
(b) Accuracy for Digit 1



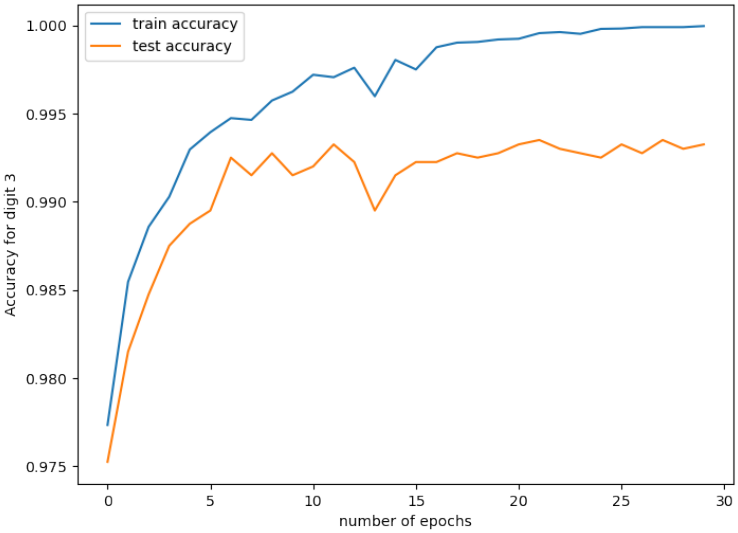
(a) Error for Digit 2



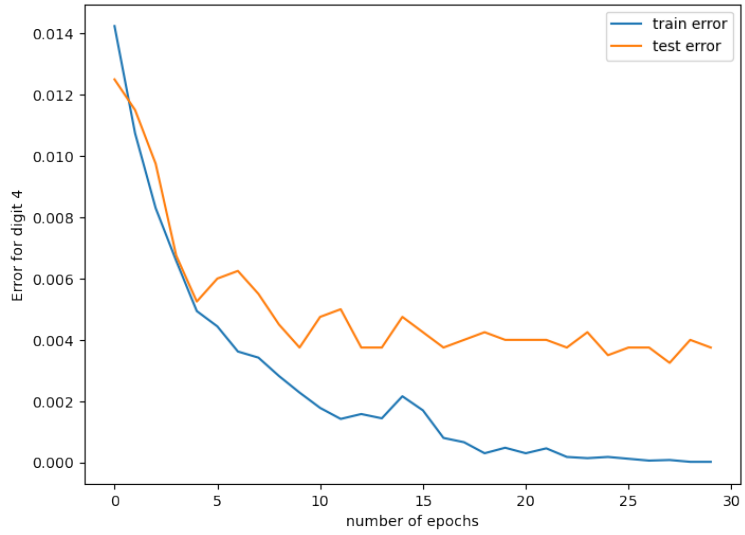
(b) Accuracy for Digit 2



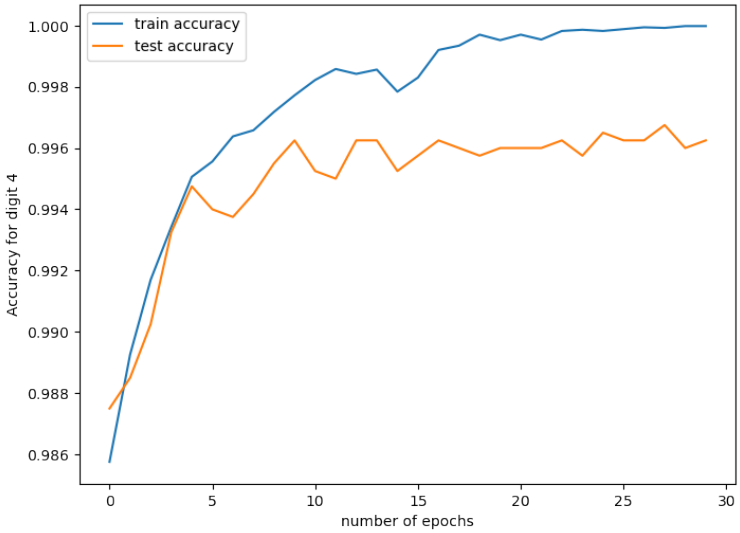
(a) Error for Digit 3



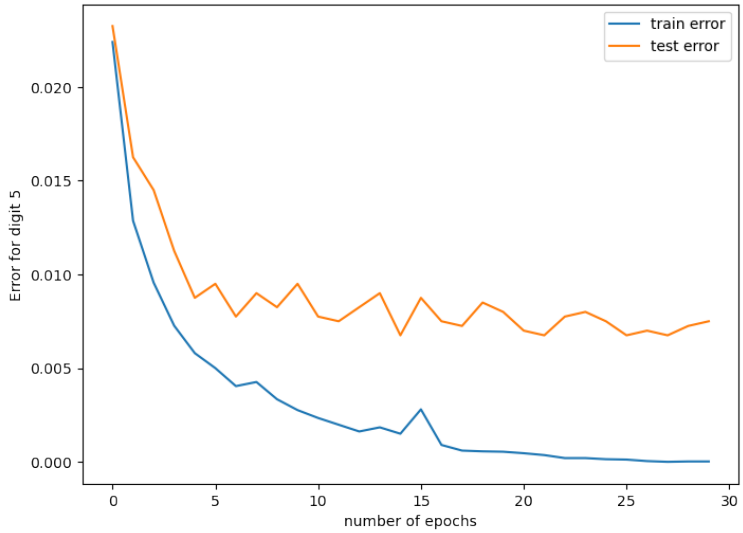
(b) Accuracy for Digit 3



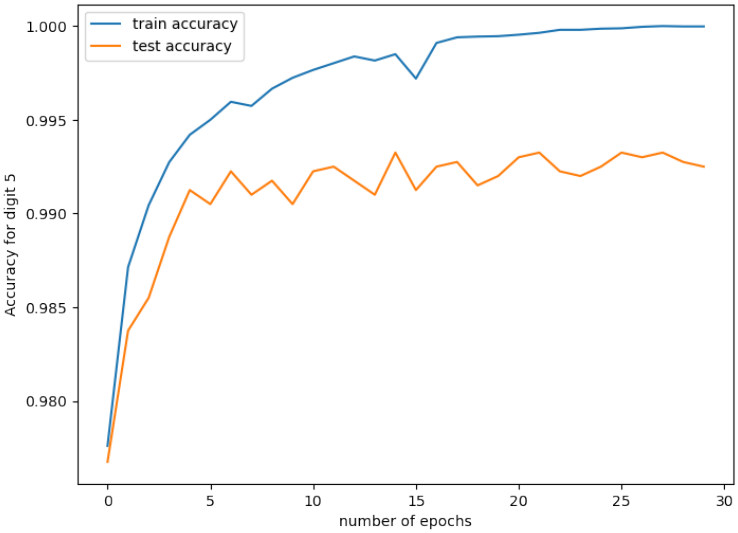
(a) Error for Digit 4



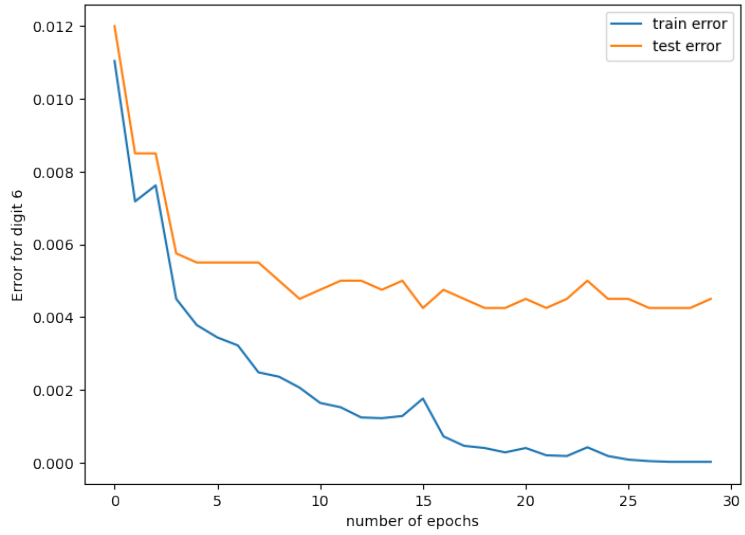
(b) Accuracy for Digit 4



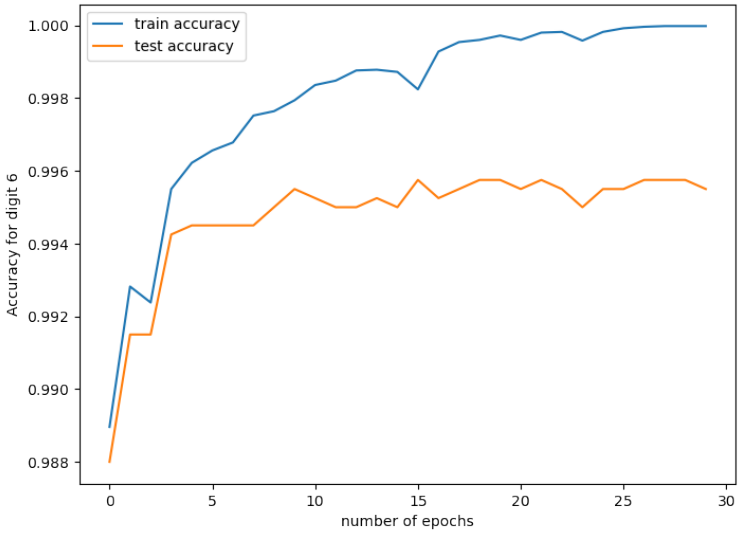
(a) Error for Digit 5



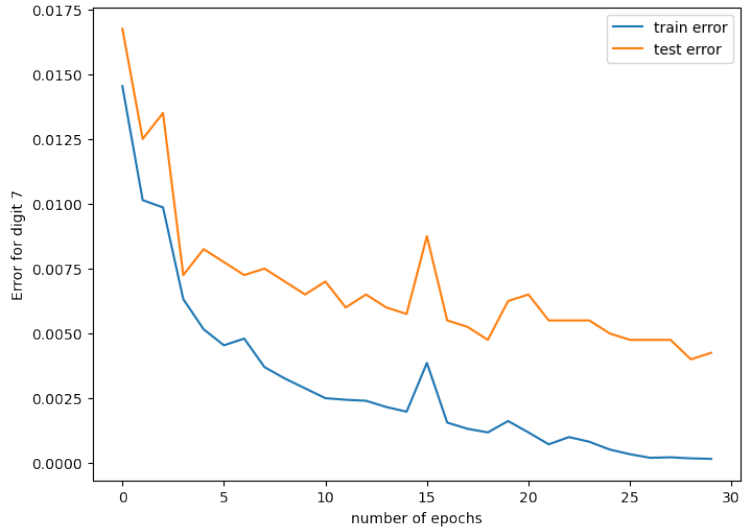
(b) Accuracy for Digit 5



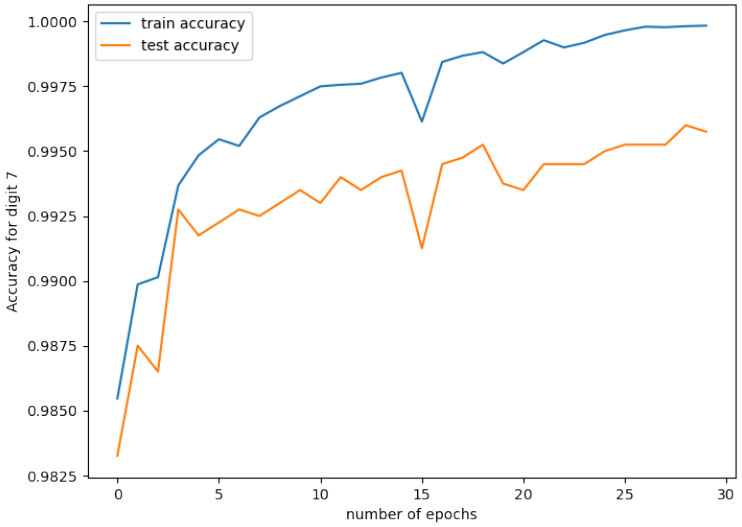
(a) Error for Digit 6



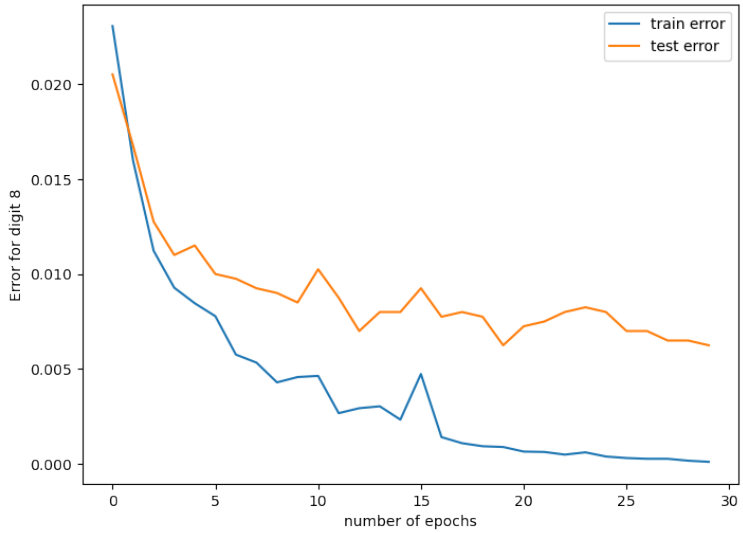
(b) Accuracy for Digit 6



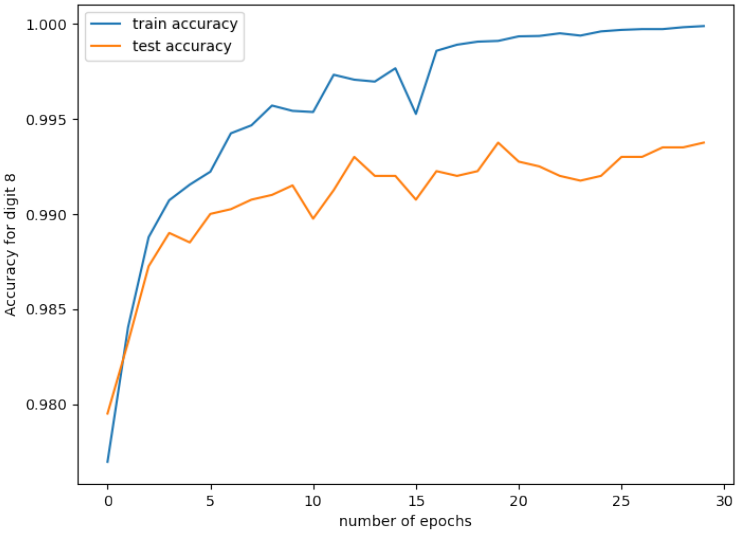
(a) Error for Digit 7



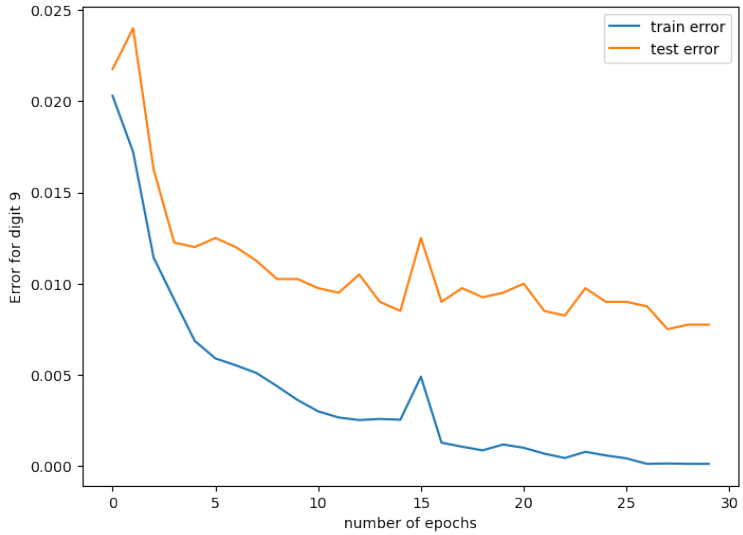
(b) Accuracy for Digit 7



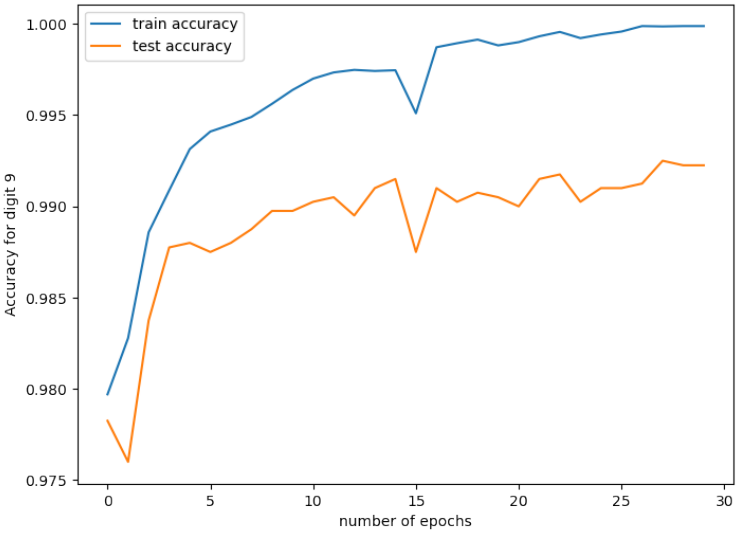
(a) Error for Digit 8



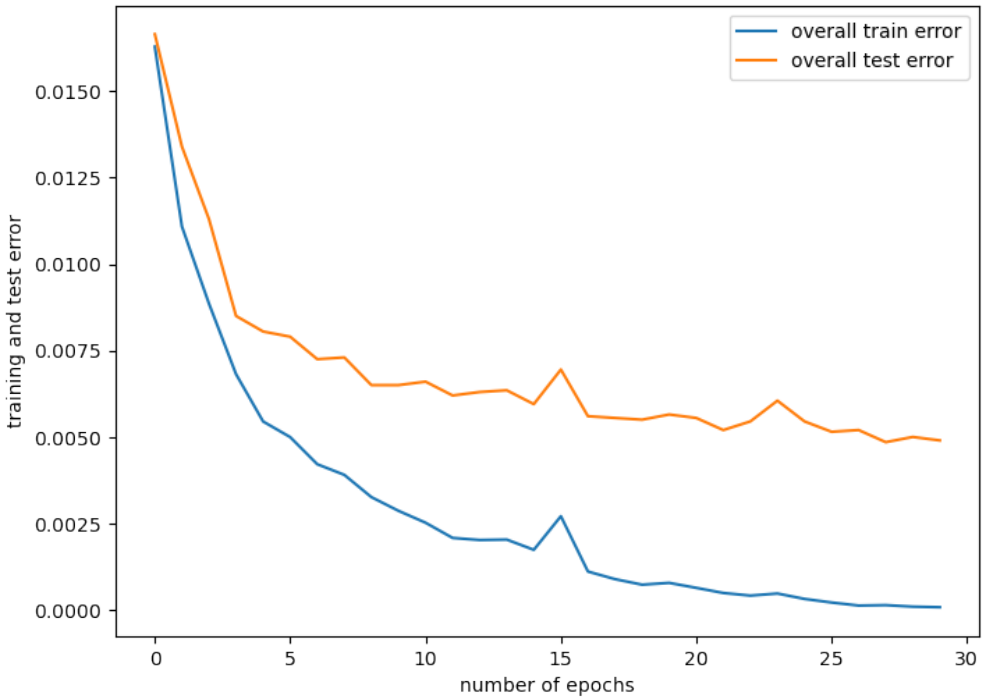
(b) Accuracy for Digit 8



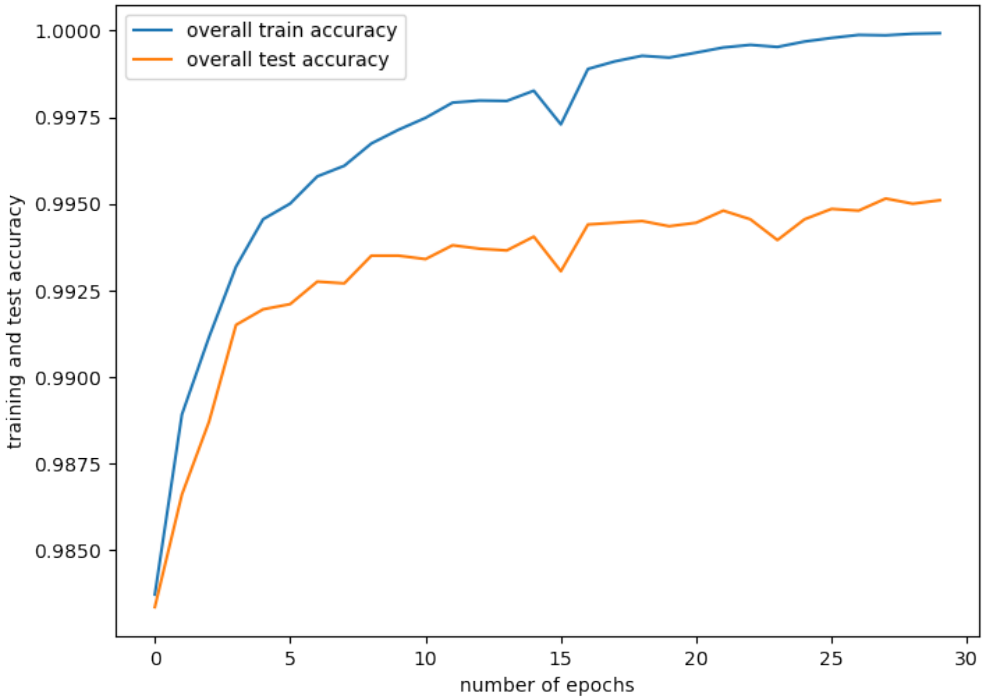
(a) Error for Digit 9



(b) Accuracy for Digit 9



(a) Overall Error for all 10 Digits



(b) Overall Accuracy for all 10 Digits

3.2 Average Train & Test Loss Over Epochs

The average train and test loss over epochs are shown in the following figure:

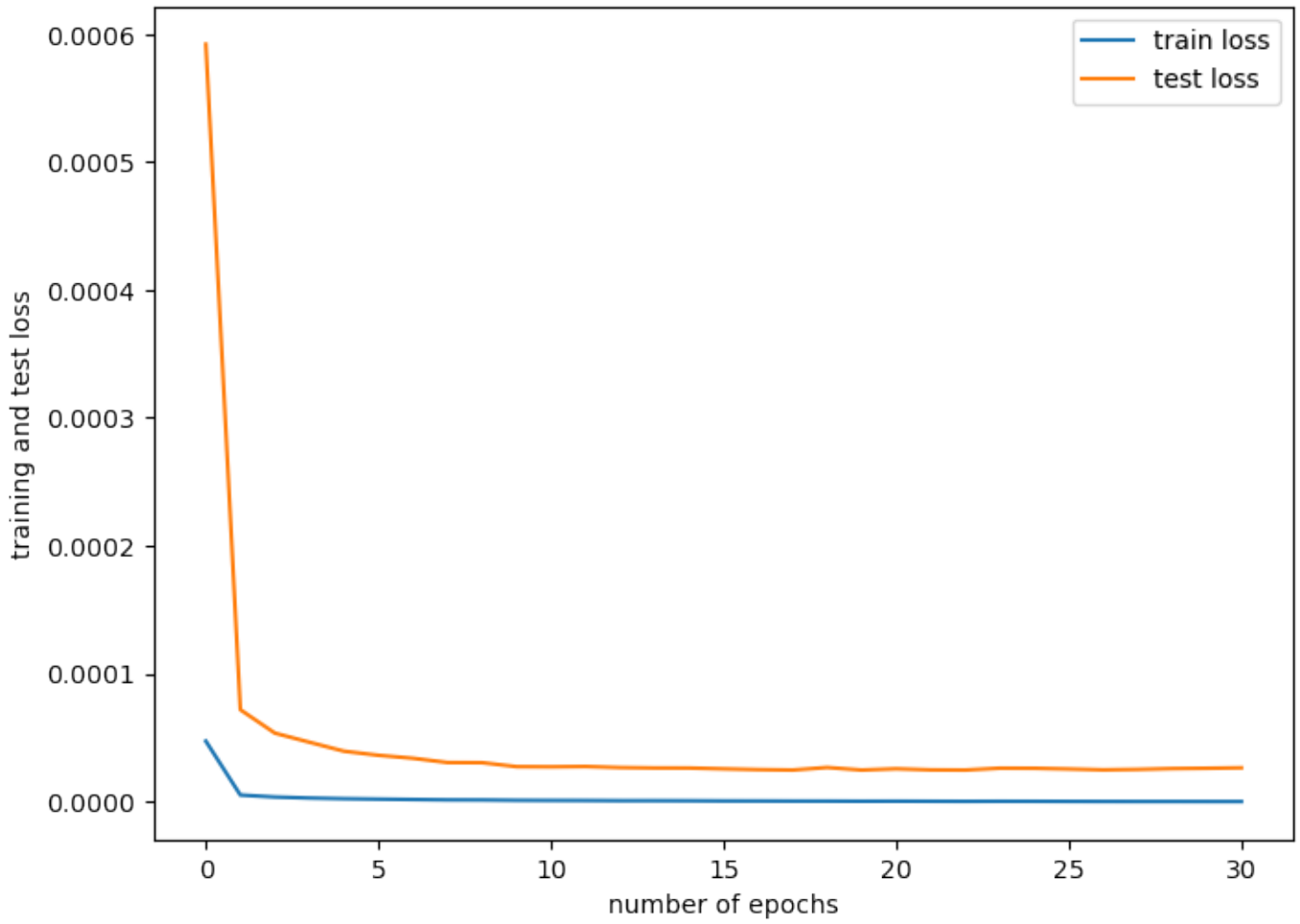


Figure 13: Average Training and Testing Loss over Epochs

APPENDIX: CODES

```
# %%
import tensorflow as tf
import os
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np
import random
import pickle
from tqdm import tqdm

# %%
#function for 1 hot label encoding
```

```
def label_encoding(label):

    y = np.zeros([10,len(label)])

    for i in range(len(label)):
        y[int(label[i]),i] = 1

    return y

#function for computing classification performance
def performance_metrics(y_true,y_pred_ind):
    y_pred=np.zeros(y_true.shape)
    for i in range(len(y_pred_ind)):
        y_pred[i,y_pred_ind[i]]=1

    errors=np.zeros(y_pred.shape[1])
    accuracies=np.zeros(y_pred.shape[1])
    for i in range(y_pred.shape[1]):
        errors[i]=np.sum(y_pred[:,i]!=y_true[:,i])/y_true.shape[0]
        accuracies[i]=1-errors[i]
    return accuracies,errors



#function for loading data
def load_data(f_loc, im_size):
    f_list = os.listdir(f_loc)
    x_data = np.zeros([len(f_list),im_size])

    for i in range(len(f_list)):
        im = mpimg.imread(f_loc + '/' + f_list[i])
        im = np.reshape(im,[1,im_size])
        x_data[i:i+1,0:im_size] = im

    x_data = x_data/255    #normalization
    x_data = np.float32(x_data)
    x_data = x_data.transpose()

    return x_data



# %%
#loading Train data
train_dir = r'train_data'
```

```
im_size = 784
train_data = load_data(train_dir, im_size)

#loading Test data
test_dir = r'test_data'
test_data = load_data(test_dir, im_size)




#loading and encoding Train Labels
path_to_label = r'labels'
tr_labels = np.loadtxt(path_to_label+'/'+'train_label.txt')
train_label = label_encoding(tr_labels)
train_label = np.float32(train_label)

#loading and encoding Test Labels
te_labels = np.loadtxt(path_to_label+'/'+'test_label.txt')
test_label = label_encoding(te_labels)
test_label = np.float32(test_label)

print("Train data shape:",train_data.shape)
print("Train label shape:",train_label.shape)
print("Test data shape:",test_data.shape)
print("Test label shape:",test_label.shape)

# %%
Y=np.transpose(train_label)

print(Y.shape)

X=np.transpose(train_data)

print(X.shape)

X_test=np.transpose(test_data)
print(X_test.shape)
Y_test=np.transpose(test_label)
print(Y_test.shape)

# %%


def regularized_gradient(w,mode=''): #Compute regularization gradient
    c=1e-6 #To prevent division by zero
```

```
    j=np.argmax(np.sum(np.abs(w),0)) #Getting Max L1 column
    b=np.zeros(w.shape)

    if mode=='matrix':

        b[:,j]=np.round((w[:,j]+c)/(abs(w[:,j])+c)) #regularization gradient of matrix p

    elif mode=='vector':
        b=np.round((w+c)/(abs(w)+c)) #regularization gradient of vector parameters

    return b

def forward_propagation(X,Y,W1,W2,W3,b1,b2,b3,mode=0): #Forward Propagation for 2 hidden
    Z1 = tf.matmul(X,W1) + b1
    A1 = tf.nn.relu(Z1)
    Z2 = tf.matmul(A1,W2) + b2
    A2 = tf.nn.relu(Z2)
    Z3 = tf.matmul(A2,W3) + b3
    A3 = tf.nn.softmax(Z3)
    if mode==0:
        return Z1,Z2,Z3,A1,A2,A3
    else:
        return A3


def weight_update(x,Y,W1,W2,W3,b1,b2,b3,learning_rate,lam=1e-4,mode=0): #Function for we
    z1,z2,z3,h1,h2,out = forward_propagation(x,Y,W1,W2,W3,b1,b2,b3)
    #cost = -tf.reduce_mean(tf.reduce_sum(Y*tf.math.log(A3),axis=1))
    loss = - tf.reduce_mean(tf.reduce_sum(tf.multiply(Y,tf.math.log(out)), axis=1))

    if mode==1:
        return loss

    ### backpropagation
    dy = -tf.divide(Y, out)
    a= tf.expand_dims(out * dy, 1)
    b=tf.eye(k, batch_shape=[batch_size])
    c=tf.expand_dims(out,1)
    dz3 = tf.squeeze(tf.matmul(a, b - c))
    a=tf.matmul(tf.expand_dims(h2,-1), tf.expand_dims(dz3, 1))
    dw3 = tf.reduce_mean(a, axis=0)
    db3 = tf.reduce_mean(dz3, axis=0)
    dh2 = tf.matmul(dz3, tf.transpose(w3))
    a=tf.matmul(tf.expand_dims(dh2, 1), tf.linalg.diag(tf.cast(z2 > 0, tf.float32)))
    dz2 = tf.squeeze(a)
```

```
    a=tf.matmul(tf.expand_dims(h1,-1), tf.expand_dims(dz2, 1))
    dw2 = tf.reduce_mean(a, axis=0)
    db2 = tf.reduce_mean(dz2, axis=0)
    dh1 = tf.matmul(dz2, tf.transpose(w2))
    a=tf.matmul(tf.expand_dims(dh1, 1), tf.linalg.diag(tf.cast(z1 > 0, tf.float32)))
    dz1 = tf.squeeze(a)
    a=tf.matmul(tf.expand_dims(x,-1), tf.expand_dims(dz1, 1))
    dw1 = tf.reduce_mean(a, axis=0)
    db1= tf.reduce_mean(dz1, axis=0)

    ### L1 regularization Gradient calculation
    dw3R = regularized_gradient(w3,'matrix')
    dw2R = regularized_gradient(w2,'matrix')
    dw1R = regularized_gradient(w1,'matrix')
    db3R = regularized_gradient(b3,'vector')
    db2R = regularized_gradient(b2,'vector')
    db1R = regularized_gradient(b1,'vector')


    #update weights
    W1 = W1 - learning_rate*(dw1+lam*dw1R)
    W2 = W2 - learning_rate*(dw2+lam*dw2R)
    W3 = W3 - learning_rate*(dw3+lam*dw3R)
    b1 = b1 - learning_rate*(db1+lam*db1R)
    b2 = b2 - learning_rate*(db2+lam*db2R)
    b3 = b3 - learning_rate*(db3+lam*db3R)

    return W1,W2,W3,b1,b2,b3,loss

# %%
n_0 =784
num_classes = 10

n_1=100
n_2=100
k = Y.shape[1] #10
lr=0.05
lam=1e-4
batch_size=50
epoch=30
num_batches=int(len(X)/batch_size)
print(num_batches)
```

```
# %%
### Weight Initialization
tf.random.set_seed(0)
w1=tf.random.normal(    [n_0,n_1], mean=0.0, stddev=0.1, dtype=tf.dtypes.float32, seed=N

w2=tf.random.normal(    [n_1,n_2], mean=0.0, stddev=0.1, dtype=tf.dtypes.float32, seed=N

w3=tf.random.normal(   [n_2,k], mean=0.0, stddev=0.1, dtype=tf.dtypes.float32, seed=None

b1=tf.zeros(    [n_1], dtype=tf.dtypes.float32, name=None)

b2=tf.zeros(    [n_2], dtype=tf.dtypes.float32, name=None)

b3=tf.zeros(    [k], dtype=tf.dtypes.float32, name=None)


# %%
train_errors=[]
train_accuracy=[]
test_errors=[]
test_accuracy=[]
train_losses=[]
test_losses=[]


tr_loss=weight_update(X,Y,w1,w2,w3,b1,b2,b3,lr,mode=1)
te_loss=weight_update(X_test,Y_test,w1,w2,w3,b1,b2,b3,lr,mode=1)
train_losses.append(tr_loss)
test_losses.append(te_loss)


#TRAINING WITH SGD for EPOCHS

for i in tqdm(range(epoch)):
    index=list(range(X.shape[0]))
    random.shuffle(index) #shuffle the index to get random batches for iterations in eac

    #weight update over mini-batches
    for j in range(num_batches): #Runs code over all batches in one epoch
        a=index[j*batch_size:(j+1)*batch_size]
        x=X[a,:]

        y=Y[a,:]
```

```
        w1,w2,w3,b1,b2,b3,cost=weight_update(x,y,w1,w2,w3,b1,b2,b3,lr,lam,mode=0)

    tr_loss=weight_update(X,Y,w1,w2,w3,b1,b2,b3,lr,mode=1)
    te_loss=weight_update(X_test,Y_test,w1,w2,w3,b1,b2,b3,lr,mode=1)
    train_losses.append(tr_loss)
    test_losses.append(te_loss)




    a3=forward_propagation(X,Y,w1,w2,w3,b1,b2,b3,mode=1)
    train_acc,train_error=performance_metrics(Y,np.argmax(a3,axis=1))


    a3test=forward_propagation(X_test,Y_test,w1,w2,w3,b1,b2,b3,mode=1)
    test_acc,test_error=performance_metrics(Y_test,np.argmax(a3test,axis=1))


    train_errors.append(train_error)
    train_accuracy.append(train_acc)
    test_errors.append(test_error)
    test_accuracy.append(test_acc)

# %%
#VISUALIZATION
train_errors=np.array(train_errors)
test_errors=np.array(test_errors)
train_accuracy=np.array(train_accuracy)
test_accuracy=np.array(test_accuracy)

for i in range(num_classes):
    plt.figure(figsize=(8, 6), dpi=100)
    plt.plot(train_errors[:,i], label = "train error")
    plt.plot(test_errors[:,i], label = "test error")
    plt.legend()
    plt.xlabel('number of epochs')
    plt.ylabel('Error for digit ' + str(i))
    plt.show()

    plt.figure(figsize=(8, 6), dpi=100)
    plt.plot(train_accuracy[:,i], label = "train accuracy")
    plt.plot(test_accuracy[:,i], label = "test accuracy")
    plt.legend()
    plt.xlabel('number of epochs')
```

```
    plt.ylabel('Accuracy for digit ' + str(i))
    plt.show()

# %%
avg_er_train = np.sum(train_errors, 1)/num_classes
avg_er_test = np.sum(test_errors, 1)/num_classes
avg_acc_train = np.sum(train_accuracy, 1)/num_classes
avg_acc_test = np.sum(test_accuracy, 1)/num_classes

plt.figure(figsize=(8, 6), dpi=100)
plt.plot(avg_er_train, label = "overall train error")
plt.plot(avg_er_test, label ="overall test error")
plt.legend()
plt.xlabel('number of epochs')
plt.ylabel('training and test error')
plt.show()

plt.figure(figsize=(8, 6), dpi=100)
plt.plot(avg_acc_train, label = "overall train accuracy")
plt.plot(avg_acc_test, label ="overall test accuracy")
plt.legend()
plt.xlabel('number of epochs')
plt.ylabel('training and test accuracy')
plt.show()

# %%
#Average loss per epoch
train_losses=np.array(train_losses)
test_losses=np.array(test_losses)
train_losses_avg=train_losses/len(tr_labels)
test_losses_avg=test_losses/len(te_labels)


#Plot Losses
plt.figure(figsize=(8, 6), dpi=100)
plt.plot(train_losses_avg, label = "train loss")
plt.plot(test_losses_avg, label = "test loss")
plt.xlabel('number of epochs')
plt.ylabel('training and test loss')
plt.legend()
plt.show()

# %%
#Checking and getting performance metrics
a3L=forward_propagation(X,Y,w1,w2,w3,b1,b2,b3,mode=1)
```

```
train_accL,train_errorL=performance_metrics(Y,np.argmax(a3L,axis=1))


a3testL=forward_propagation(X_test,Y_test,w1,w2,w3,b1,b2,b3,mode=1)
test_accL,test_errorL=performance_metrics(Y_test,np.argmax(a3testL,axis=1))


print("Training Error: ",train_errorL)
print("Test Error: ",test_errorL)
print("Training Accuracy: ",train_accL)
print("Test Accuracy: ",test_accL)

print('Average tr err',np.mean(train_errorL))
print('Average te err',np.mean(test_errorL))
print('Average tr acc',np.mean(train_accL))
print('Average te acc',np.mean(test_accL))

# %%
#savig the parameters as numpy arrays

theta=[w1.numpy(),b1.numpy(),w2.numpy(),b2.numpy(),w3.numpy(),b3.numpy()]
filehandler = open("nn_parameters.txt","wb")
pickle.dump(theta, filehandler, protocol=2)
filehandler.close()

# %%

#Verification that the parameters were saved correctly
file=open("nn_parameters.txt","rb")
theta_load=pickle.load(file)
file.close()
w1l=theta_load[0]
b1l=theta_load[1]
w2l=theta_load[2]
b2l=theta_load[3]
w3l=theta_load[4]
b3l=theta_load[5]

a3L=forward_propagation(X,Y,w1l,w2l,w3l,b1l,b2l,b3l,mode=1)
train_accL,train_errorL=performance_metrics(Y,np.argmax(a3L,axis=1))


a3testL=forward_propagation(X_test,Y_test,w1l,w2l,w3l,b1l,b2l,b3l,mode=1)
test_accL,test_errorL=performance_metrics(Y_test,np.argmax(a3testL,axis=1))
```

```
print("Training Error: ",train_errorL)
print("Test Error: ",test_errorL)
print("Training Accuracy: ",train_accL)
print("Test Accuracy: ",test_accL)

print('Average tr err',np.mean(train_errorL))
print('Average te err',np.mean(test_errorL))
print('Average tr acc',np.mean(train_accL))
print('Average te acc',np.mean(test_accL))
```