Programming Assignment 1

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

Introduction

In this programming assignment, our goal is to implement techniques to learn a multi-class logistic regressor 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 1 and 5 as input and classify it into one of 5 classes corresponding to digit 1 to 5 respectively. Logistic Regression is a probabilistic method for classification which is also called softmax classifier. The Negative Log Likelihood or cross-entropy loss function has been used as the loss function for this model. 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 L2 norm regularization of parameters to prevent the model from overfitting.
This report is comprised of three sections. In the first section, we will discuss the theoretical development of multiclass logistic regression model.The architecture of the model, hyper-parameters and experimental set-up is discussed in Section 2. In section 3, 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 discriminant function for each class $f_k(x) = W_k^T x + W_{k,0} = X^T \theta_k$,
where θ_k =$[W_k \ W_{k,0}]^T$, X=[x 1]^T and $\Theta = [\theta_1, \theta_2, ...\theta_K]$
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 gradient calculation is shown below:

1.2 Loss Function

The probability function by using a softmax function as

$$p(y(m) = k|x(m), \Theta) = \frac{exp(x^T(m)w_k + w_{k,0})}{\sum_{k'=1}^{K} exp(x^T(m)w_k' + w_{k',0})}$$

Here, $\theta_k = \begin{pmatrix} W_k \\ W_{k,0} \end{pmatrix}$ and $\Theta = [\theta_1, \theta_2....\theta_k]$

The likelihood function,

$$p(y(m)|x(m), \Theta) = \prod_{k=1}^{K} p(y(m) = k|x(m), \Theta)^{I(y(m)=k)}$$

The negative log-likelihood loss function is described below:

$$\begin{aligned}
L(D:\Theta) &= -\sum_{m=1}^{M} log[p(y(m)|x(m),\Theta)] \\
&= -\sum_{m=1}^{M} log\prod_{k=1}^{K} p(y(m)=k|x(m),\theta)^{I(y(m)=k})\\
&= -\sum_{m=1}^{M}\sum_{k=1}^{K} I(y(m)=k)log\frac{exp(x^T(m)w_k+w_{k,0})}{\sum_{k'=1}^{K} exp(x^T(m)w'_k+w'_{k,0})}\\
&= -\sum_{m=1}^{M}\sum_{k=1}^{K} I(y(m)=k)log\frac{exp(X^T(m)\theta_k)}{\sum exp(X^T(m)\theta_{k'})}\\
&= -\sum_{m=1}^{M}\sum_{k=1}^{K} I(y(m)=k)[(X^T(m)\theta_k)-log\sum_{k'=1}^{K} exp(X^T(m)\theta_{k'})]
\end{aligned}$$

1.3 Gradient Calculation

Now, The gradient of the given loss function can be formulated as follows:

$$\begin{aligned}
\nabla_\theta L(D:\Theta) &= \frac{\delta L(D:\Theta)}{\delta\Theta}\\
or, \nabla_\theta L(D:\Theta) &= [\frac{\delta L(D:\Theta)}{\delta\theta_1},\frac{\delta L(D:\Theta)}{\delta\theta_2}......\frac{\delta L(D:\Theta)}{\delta\theta_k}]
\end{aligned}$$

The gradient $\frac{\delta L(D:\Theta)}{\delta\theta_k}$ of each θ_k can be calculated from the Loss function above,

$$\begin{aligned}
\nabla_{\theta_k} L(D:\Theta) &= \frac{\delta L(D:\Theta)}{\delta\theta_k}\\
&= -\sum_{m=1}^{M}\{I(y(m)=k)-\frac{exp(X^T(m)\theta_k)}{\sum_{k'=1}^{K} exp(X^T(m)\theta_{k'}}\}X(m)
\end{aligned}$$

1.4 Regularization: L2 norm

We take the L2 norm of the parameters as a regularization function. We calculate the gradient of the L2 norm and add to the Loss function after scaling by a factor λ.

$$\begin{aligned}
R(\Theta) &= \lambda\Theta^T\Theta\\
or, \nabla_\theta R(\Theta) &= 2\lambda\Theta
\end{aligned}$$

1.5 Parameter Update

The formula for parameter update is:

$$\Theta^i = \Theta^{i-1} - \eta[\nabla_\theta L(D : \Theta) + \lambda \nabla_\theta R(\Theta)]$$

1.6 Stochastic Gradient Descent

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, 100 samples are chosen as mini-batch and SGD.The gradient of SGD is calculated as follows:

$$\frac{\delta L(D, \Theta)}{\delta \Theta} = \frac{1}{S} \sum_{x(m), y(m) \epsilon D_b} \frac{\delta l(x(m), y(m), \Theta)}{\delta \Theta}$$

where $l(x(m), y(m), \theta)$ is the loss calculated for m-th sample and S is the batch size.

2 Section 2: Experimental Setup

2.1 Weight initialization

Weights are initialized randomly from a Uniform distribution between 0 and 0.1.

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 0.0001 and 0.1 respectively. The batch size is 100. The training is performed for 1000 epochs. In each epoch, all the training samples are used after being divided into several mini-batches.

2.3 Saving the Learned Parameters

The best parameters are chosen based on the lowest average test error for all 5 digits. It is seen that, the test error reaches its minimum on **epoch number 536**. After that, the test error no longer decreases up to the maximum epoch of 1000. So, it is evident that the model converges on epoch 536. The weights during epoch 536 is saved using *pickle.dump* after the training finishes. It should be noted that **The dimension of the saved weight is 785x5**

2.4 Coding Framework

Coding has been done in python where all the matrix manipulations have been done using the *numpy* module.

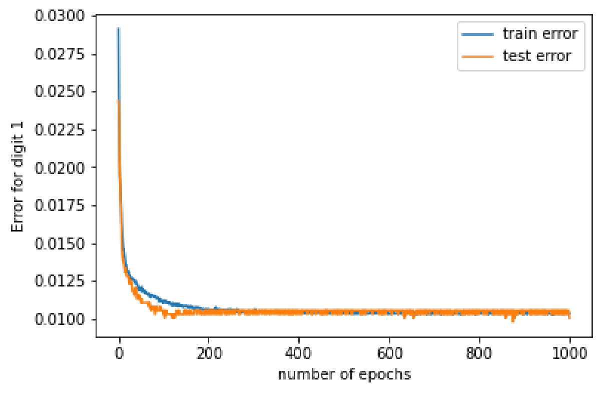
3 Section 3: Result Analysis

3.1 Classification Errors and Accuracies

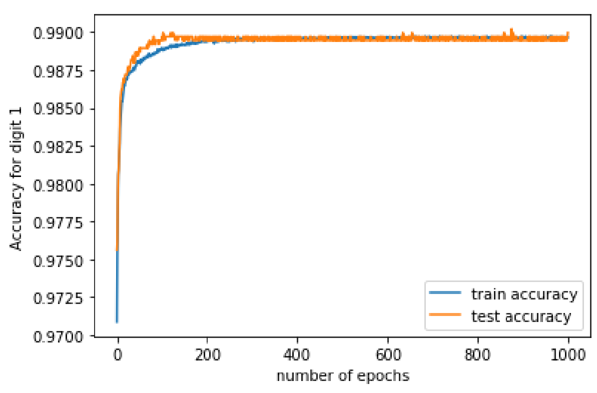
Using the best learned parameters we calculate the training errors, test error, training accuracy and test accuracy. The results are shown below:

Item name	Digit 1	Digit 2	Digit 3	Digit 4	Digit 5	Average for All the Digits
Training Error	0.0104	0.0232	0.0266	0.0104	0.0235	0.0188
Training Accuracy	0.9896	0.9767	0.9733	0.9895	0.9765	0.9811
Test Error	0.0102	0.0210	0.0296	0.0123	0.0281	0.0202
Test Accuracy	0.9897	0.9789	0.9703	0.9876	0.9718	0.9797

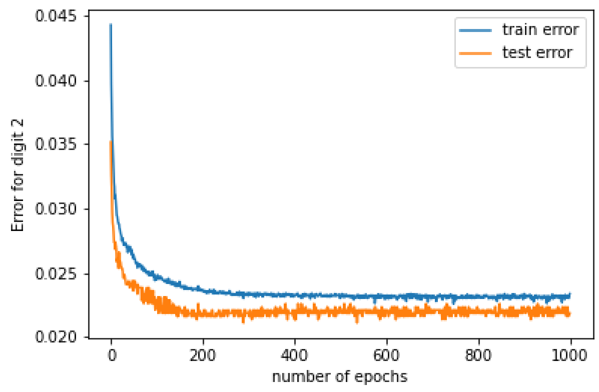
So, the overall training accuracy is 98.11% and overall training error is 1.88% . Whereas, overall test accuracy is 97.97% and overall test error is 2.02%. The relevant plots are shown below



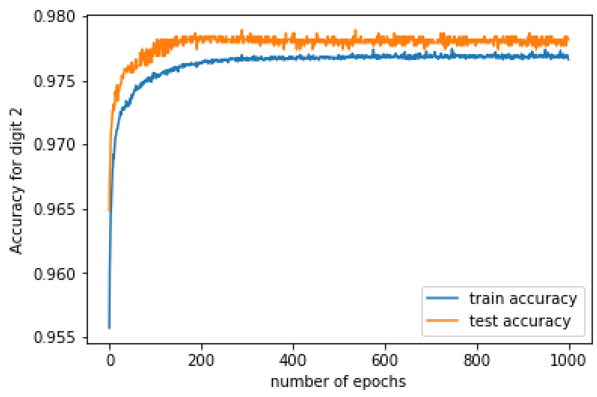
(a) Error for Digit 1



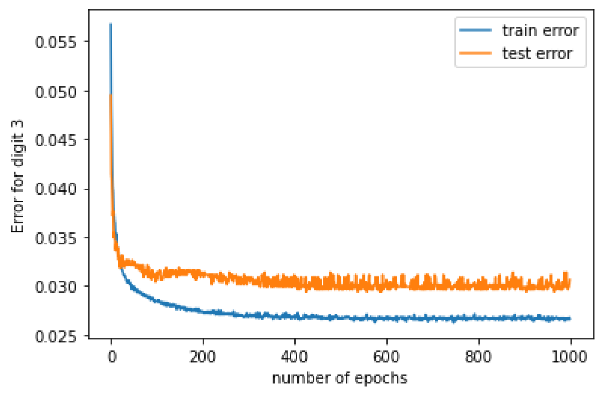
(b) Accuracy for Digit 1



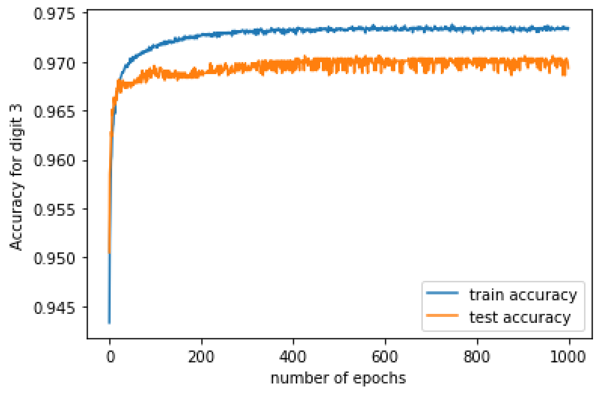
(a) Error for Digit 2



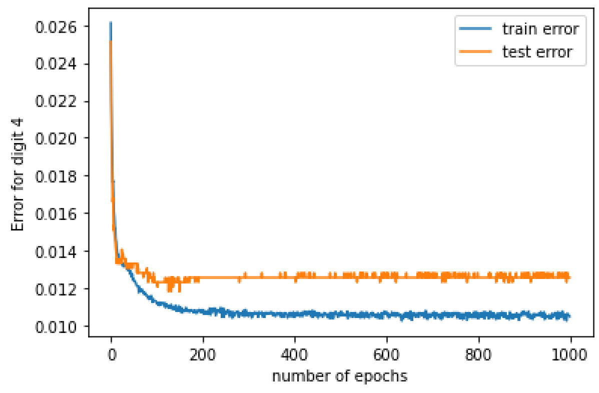
(b) Accuracy for Digit 2



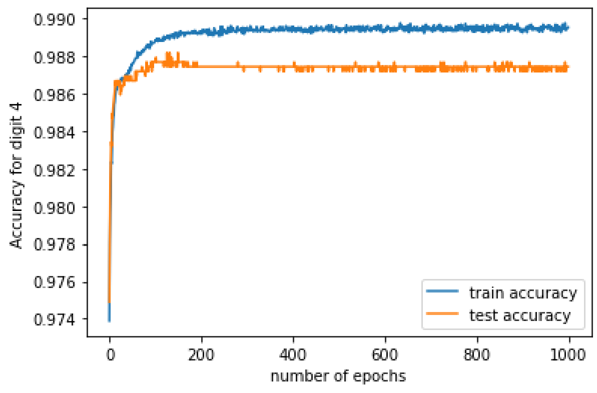
(a) Error for Digit 3



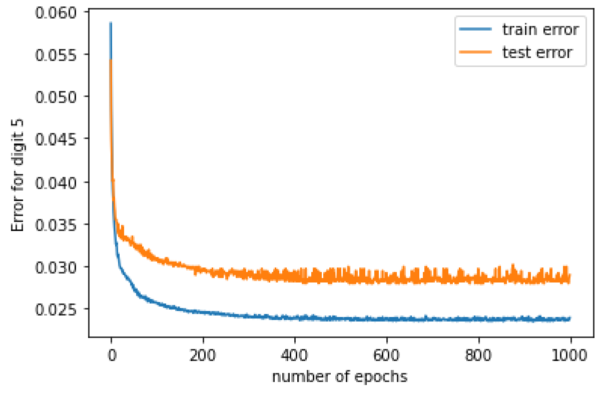
(b) Accuracy for Digit 3



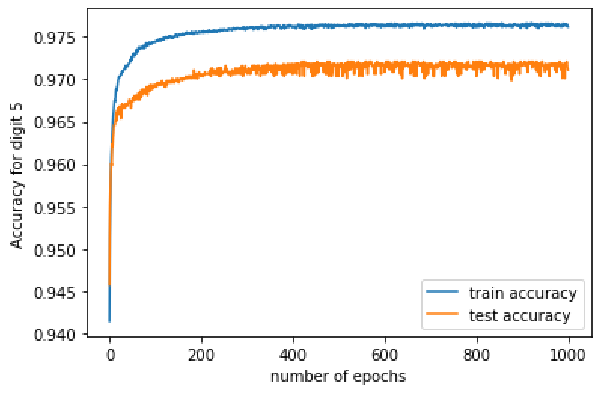
(a) Error for Digit 4



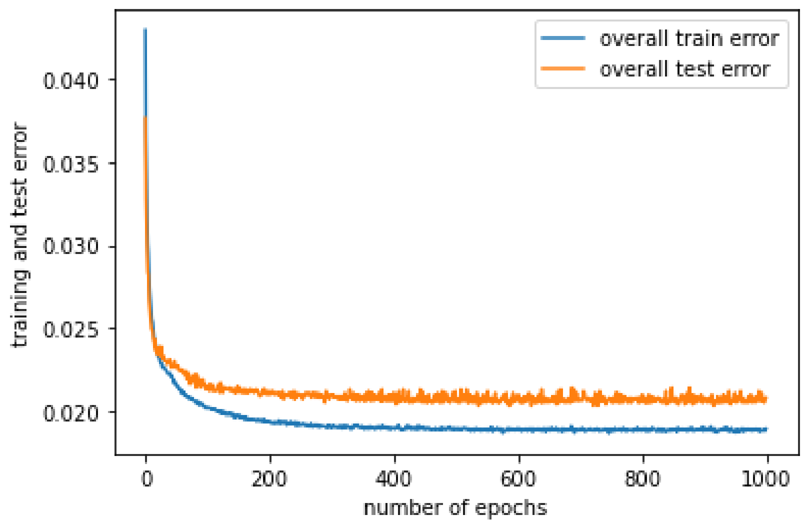
(b) Accuracy for Digit 4



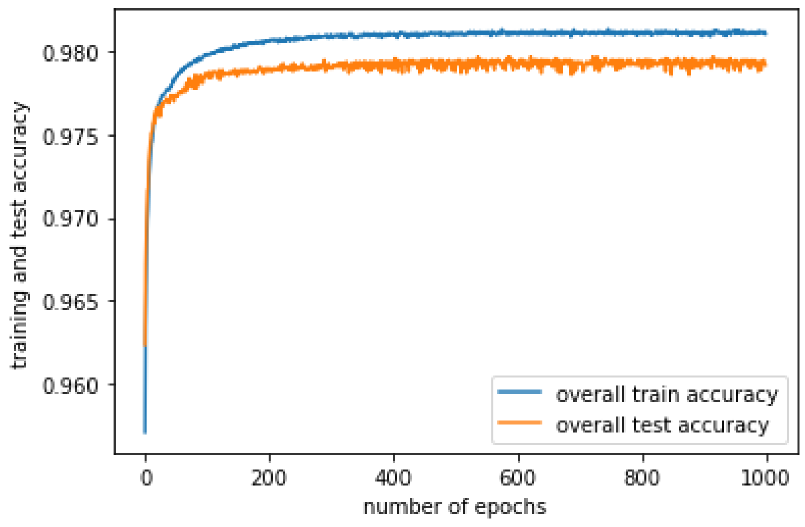
(a) Error for Digit 5



(b) Accuracy for Digit 5



(a) Overall Error for all 5 Digits



(b) Overall Accuracy for all 5 Digits

3.2 Average Train & Test Loss Over Epochs

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

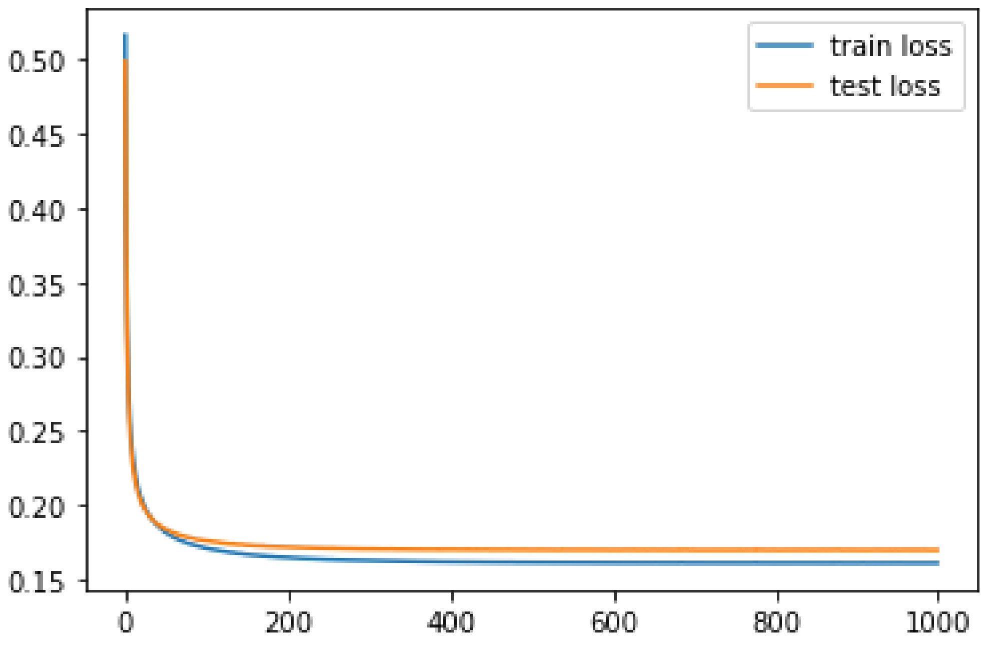


Figure 7: Average Training and Testing Loss over Epochs

3.3 Saved Parameters Visualization

The Saved Parameter are visually shown below:

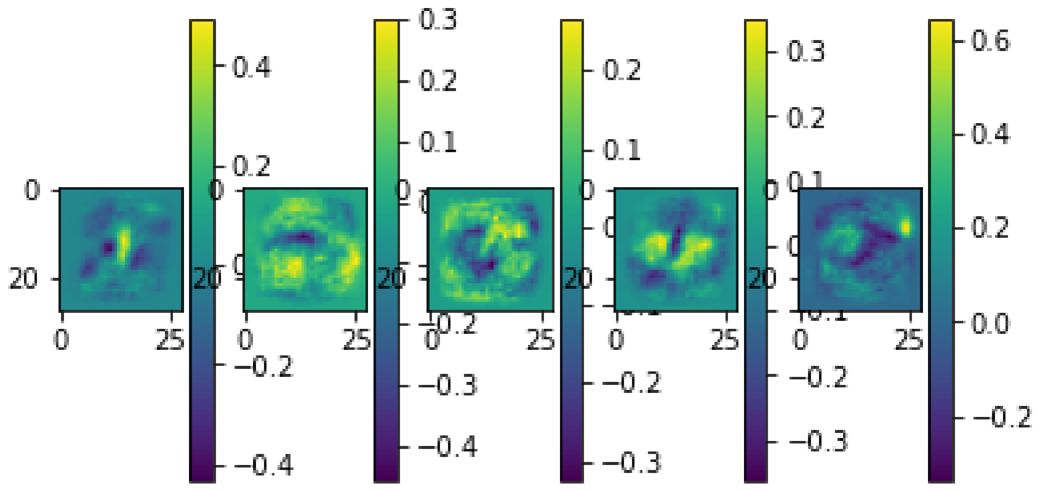


Figure 8: Saved Parameters for all 5 Digits $[W_1, W_2, W_3, W_4, W_5]$

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

# %%


######## Change it according to python script and data location ########
cwd=os.path.realpath(__file__)
script_name=os.path.basename(cwd)
cwd=cwd.replace(script_name,'')
print('Current Working Directory: ',cwd)
#os.chdir(r'C:\\Users\\nahia\Google Drive (nahian.buet11@gmail.com)\\Spring-22 Drive Fol
os.chdir(cwd) #Change it to your own directory




# %%

#function for 1 hot label encoding
def label_encoding(label):

    y = np.zeros([5,len(label)])

    for i in range(len(label)):
        y[int(label[i]-1),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.ones([len(f_list),im_size+1])

    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)

#Function for parameter update
def param_update(X,Y,W):
    A=np.exp(X@W)
    Z=A/np.sum(A,axis=1,keepdims=True)
    B=np.log(Z)
    Loss=-np.sum(B*Y) #Total loss over mini-batch

    Q=Y-Z
    grad=-X.T@Q #Gradient of loss wrt W over mini-batch

    W_new=W-lr*(grad+2*lam*W) #Update of W with regularization
    return Loss,W_new



# %%
lr=0.0001 #learning rate
lam=0.1 #Regularization parameter
W=np.random.uniform(low=0.0, high=0.1, size=(X.shape[1],Y.shape[1]))#small weights initi
batch_size=100

epoch=1000 #max number of epochs
```

```
num_batch=int(len(X)/batch_size) #Total number of batches
print('total number of batches:',num_batch)


best_error=100000 #Arbitrary large number, will be replaced by the lowest error during i
best_W=np.zeros((X.shape[1],Y.shape[1])) #Arbitrary zero weights, will be replaced by th

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



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_batch): #Runs code over all batches in one epoch
        a=index[j*batch_size:(j+1)*batch_size]
        x=X[a,:]
        y=Y[a,:]

        loss,W=param_update(x,y,W)

    tr_loss,_=param_update(X,Y,W)
    te_loss,_=param_update(X_test,Y_test,W)
    train_losses.append(tr_loss)
    test_losses.append(te_loss)
    train_acc,train_error=performance_metrics(Y,np.argmax(X@W,axis=1))
    test_acc, test_error=performance_metrics(Y_test,np.argmax(X_test@W,axis=1))


    train_errors.append(train_error)
    train_accuracy.append(train_acc)
    test_errors.append(test_error)
    test_accuracy.append(test_acc)
    if np.mean(test_error)<best_error:
        best_error=np.mean(test_error)
        best_W=W

        best_epoch=i
```

```
# %%
print('Best Epoch: ',best_epoch)
print('Best Average Test Error: ',best_error)
print('Best errors for each digit:', test_errors[best_epoch])

# %%
best_test_acc, best_test_error=performance_metrics(Y_test,np.argmax(X_test@best_W,axis=1
print('Best Test Accuracy: ',best_test_acc)
print('Best Test Error: ',best_test_error)
print('best average test accuracy: ',np.mean(best_test_acc))
print('best average test error: ',np.mean(best_test_error))

# %%
best_train_acc, best_train_error=performance_metrics(Y,np.argmax(X@best_W,axis=1))
print('Best Train Accuracy: ',best_train_acc)
print('Best Train Error: ',best_train_error)
print('best average train accuracy: ',np.mean(best_train_acc))
print('best average train error: ',np.mean(best_train_error))

# %%
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(5):
    plt.figure()
    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+1))
    plt.show()

    plt.figure()
    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+1))
    plt.show()

# %%
avg_er_train = np.sum(train_errors, 1)/5
```

```
avg_er_test = np.sum(test_errors, 1)/5
avg_acc_train = np.sum(train_accuracy, 1)/5
avg_acc_test = np.sum(test_accuracy, 1)/5

plt.figure()
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()
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()

# %%
#saving weights in a text file
#Shape of W is (785,5)
filehandler = open("multiclass_parameters.txt","wb")
pickle.dump(best_W, filehandler)
filehandler.close()

# %%
#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 Loss
plt.figure()
plt.plot(train_losses_avg, label = "train loss")
plt.plot(test_losses_avg, label = "test loss")
plt.legend()
plt.show()

# %%
#Plotting the learned weights
```

```
for i in range(5):
    image = best_W[0:784,i]
    image= image.reshape(28,28)
    plt.subplot(1,5,i+1)
    plt.imshow(image)
    plt.colorbar()
plt.show()

# %%
tf.test.is_gpu_available(cuda_only=False, min_cuda_compute_capability=None)

# %%
#Load the saved parameter files for checking everything is working fine
file=open("multiclass_parameters.txt","rb")
W_load=pickle.load(file)
print(W_load.shape)
file.close()

# %%
#Plotting the learned weights after loading
for i in range(5):
    image = W_load[0:784,i]
    image= image.reshape(28,28)
    plt.subplot(1,5,i+1)
    plt.imshow(image)
    plt.colorbar()
plt.show()
```