Department of Electronic and Telecommunication Engineering

University of Moratuwa, Sri Lanka

EN2550 - Fundamentals of Image Processing and Machine Vision

ASSIGMENT 4

Submitted By

Nagasinghe K.R.Y 180411K

Submitted on

April 2, 2021

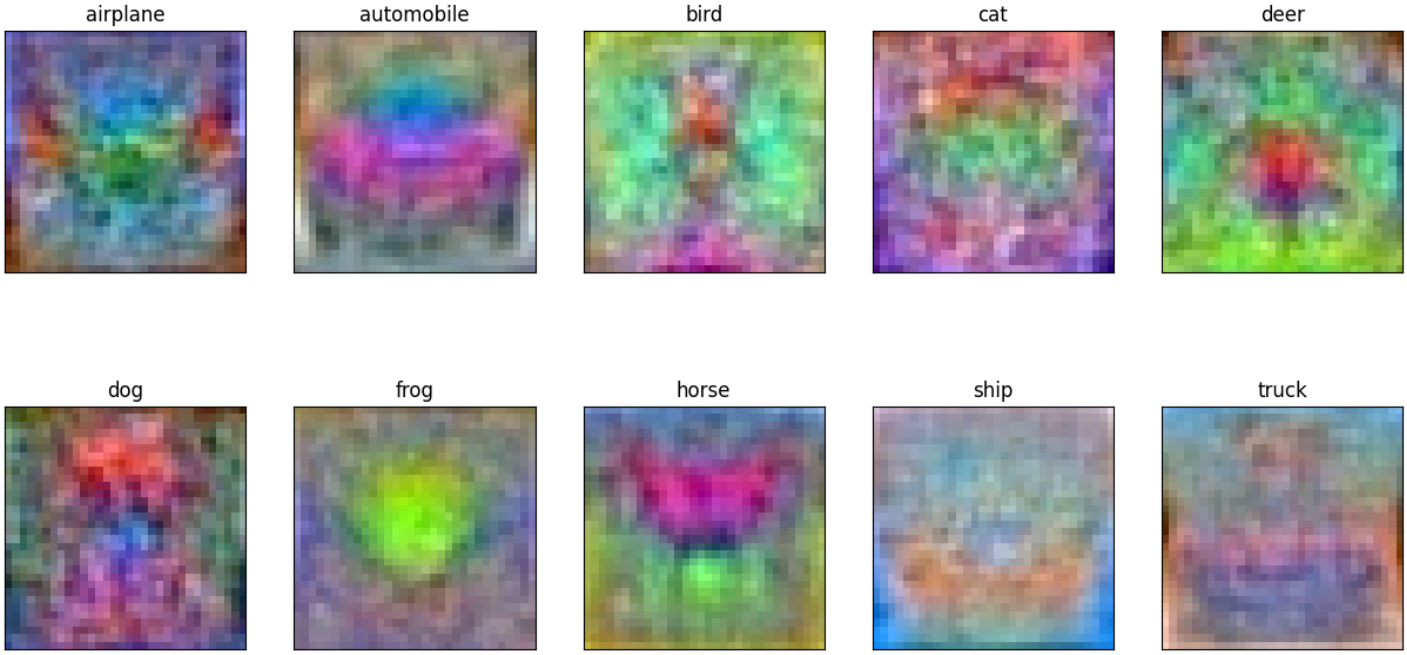
Full code: https://github.com/Ravindu-Yasas-Nagasinghe/EN2550-Computer-Vision-and-Image-Processing-Assigments

1)Linear Classification using gradient descent.

Here our data set is CIFAR-10. There are 10 different classes in this data set. I use tensorflow to import the data set to python. Our score function for the linear classifier is f (x) = W x +b, and the loss function is the mean sum of squared errors function. I run for 300 epochs as instructed in the assignment. The code for 1layer linear classifier using gradient descent is as follows.

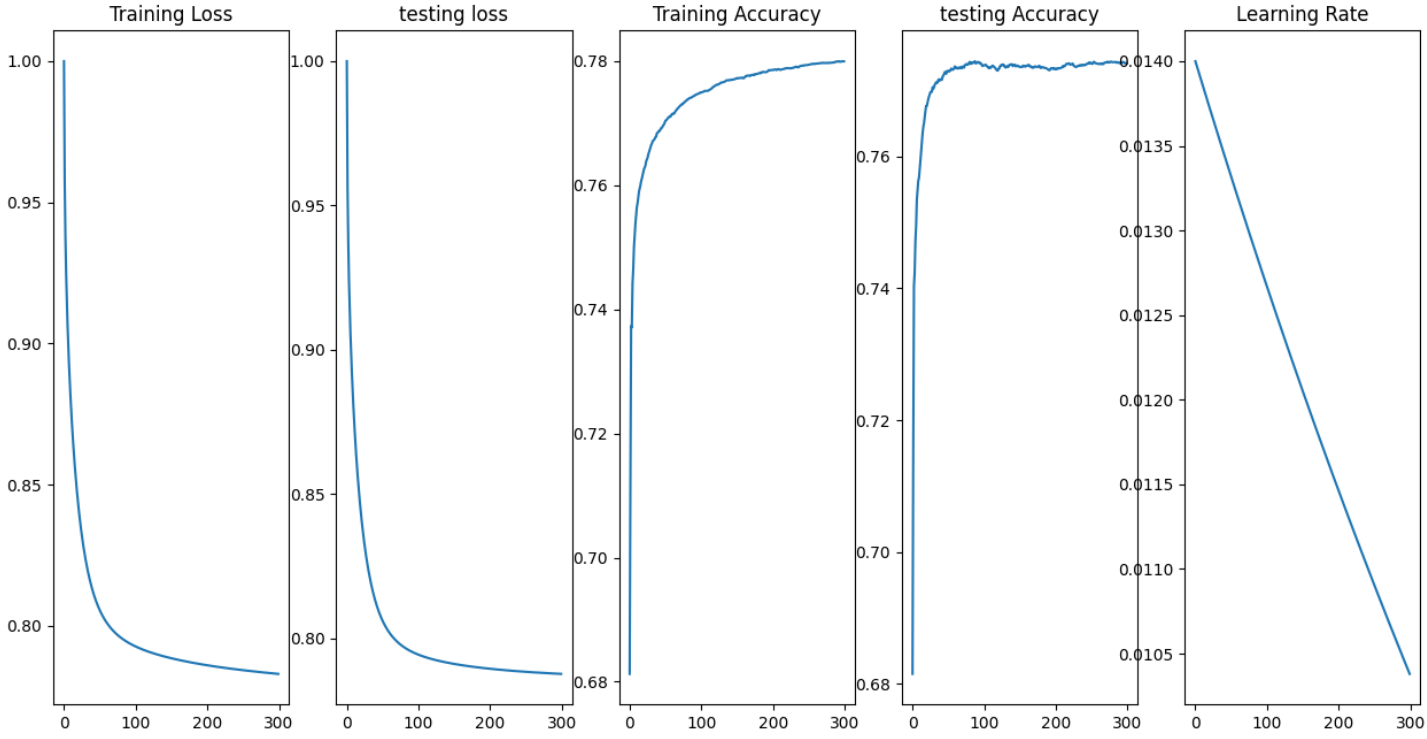
```
# Defining function for data preprocessing
def preprocessing(normalize,reshape):
    # Importing data set CIFAR-10
    (x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()

    K = len(np.unique(y_train)) # no.of Classes
    Ntr = x_train.shape[0] # Number of training data=50,000
    Nte = x_test.shape[0] # Number of testing data=10,000
    Din = 3072 # CIFAR10 = 3072 = 32x32x3

    if normalize:
        x_train, x_test = x_train / 255.0, x_test / 255.0# Normalize pixel values

    # Center the pixel values
    mean_image = np.mean(x_train, axis=0)
    x_train = x_train - mean_image
    x_test = x_test - mean_image
    y_train = tf.keras.utils.to_categorical(y_train, num_classes=K)
    y_test = tf.keras.utils.to_categorical(y_test, num_classes=K)

    #flatterning the input images
    if reshape:
        x_train = np.reshape(x_train,(Ntr,Din))
        x_test = np.reshape(x_test,(Nte,Din))
    # Changing the data types
    x_train = x_train.astype('float32')
    x_test = x_test.astype('float32')
    return x_train,y_train,x_test,y_test,K,Din,Ntr,Nte
# Defining linear Classifier function
def layer1LinearClassifier(x_train,y_train,x_test,y_test,K,Din,lr,lr_decay,reg,Ntr,Nte):
    batch_size = Ntr
    loss_history = []
    loss_history_testing = []
    train_acc_history = []
    val_acc_history = []
    lr_array=[]
    seed = 0
    rng = np.random.default_rng(seed=seed)

    # Initializing weight and bias arrays
    std=1e-5#standard deviation to generate random values for w1 and b1
    w1 = std*np.random.randn(Din, K)#weight matrix
    b1 = np.zeros(K)#k dimensional bias matrix

    for t in range(iterations):
        # To prevent over fitting we shuffle the training data set in order to randomize the training process.
        indices = np.arange(Ntr)
        rng.shuffle(indices)
        x=x_train[indices]
        y=y_train[indices]

        # forward propagation
        y_pred=x.dot(w1)+b1
        y_pred_test=x_test.dot(w1)+b1
        val=y_pred_test.shape[0]
        # calculating loss using regularized loss function
        train_loss=(1/batch_size)*(np.square(y_pred-y)).sum()+reg*(np.sum(w1*w1))
        loss_history.append(train_loss)
        test_loss=(1/val)*(np.square(y_pred_test-y_test)).sum()+reg*(np.sum(w1*w1))
        loss_history_testing.append(test_loss)
        # calculating trainning and testing accuracies
        train_accuracy=1-(1/(10*batch_size))*(np.abs(np.argmax(y,axis=1)-np.argmax(y_pred,axis=1))).sum()
        train_acc_history.append(train_accuracy)

        test_accuracy=1-(1/(10*Nte))*(np.abs(np.argmax(y_test,axis=1)-np.argmax(y_pred_test,axis=1))).sum()
        val_acc_history.append(test_accuracy)

        if t%10 == 0:
            print('epoch %d/%d: train loss= %f || ,test loss= %f ||,train accuracy= %f ||, test accuracy= %f ||, learning rate= %f ||'
            % (t,iterations,train_loss,test_loss,train_accuracy,test_accuracy,lr))

        # Backward propagation
        dy_pred=(1./batch_size)*2.0*(y_pred-y)#partial deravative of L w.r.t y_pred
        dw1=x.T.dot(dy_pred)+reg*w1
        db1=dy_pred.sum(axis=0)


        w1-=lr*dw1#update weight matrix
        b1-=lr*db1#update bias matrix
        lr_array.append(lr)
        lr*=lr_decay#decaying the learning rate
    return w1,b1,loss_history,loss_history_testing,train_acc_history,val_acc_history,lr_array


#defining parameters
iterations = 300#gradient descent iterations
lr = 1.4e-2#learning rate
lr_decay= 0.999
reg = 5e-6 #lambda=regularization parameter
x_train,y_train,x_test,y_test,K,Din,Ntr,Nte=preprocessing(normalize=True,reshape=True)
 #Run the linear classifier
w1,b1,loss_history,loss_history_test,train_acc_history,val_acc_history,lr_array=layer1LinearClassifier(x_train,y_train,x_test,y_test,K,Din,lr,lr_decay,reg,Ntr,Nte)
```

After that I plot the weight matrix W, as 10 images and plot the training loss, testing loss, training accuracy, testing accuracy and learning rate. W1 weight array is of the shape 3072 x 10. (Code for plotting is not included due to page constraints. Full code is available at the link on cover page).



Weight matrix as 10 images

Initial learning rate = 1.4×10^{-2}



Loss, testing loss, training accuracy, testing accuracy, learning rate of the linear classifier for 300 epochs.

After 300 epochs loss= 0.783117, test loss= 0.787730, train accuracy= 0.779958, test accuracy= 0.774120, learning rate= 0.010474.

2) 2 layer fully connected network

Here I use a two-layer dense network with H=200 hidden nodes. Code for this part is as follows.

```
#part 2
# function for two layer dense network
def layer_2(x_train,y_train,x_test,y_test,Din,lr,lr_decay,H,reg,K,Ntr,Nte):
    loss_history = []
    loss_history_test = []
    train_acc_history = []
    val_acc_history = []
    lr_array =[]
    seed = 0
    rng = np.random.default_rng(seed=seed)
    batch_size=Ntr

    std=1e-5
    #initializing weight and bias matrices for hidden layer
    w1 = std*np.random.randn(Din, H)
    b1 = np.zeros(H)
    #initializing weight and bias matrices for final layer
    w2 = std*np.random.randn(H, K)
    b2 = np.zeros(K)

    for t in range(iterations):
        indices = np.arange(Ntr)
        rng.shuffle(indices)# To avoid overfitting shuffle the training data set
        x=x_train[indices]
        y=y_train[indices]

        #forward propagation
        h=1/(1+np.exp(-(x.dot(w1)+b1)))
        h_test=1/(1+np.exp(-((x_test).dot(w1)+b1)))
        y_pred=h.dot(w2)+b2
        y_pred_test=h_test.dot(w2)+b2
        val=y_pred_test.shape[0]
        # calculating the training and testing loss
        training_loss=(1/batch_size)*(np.square(y_pred-y)).sum()+reg*(np.sum(w1*w1)+np.sum(w2*w2))
        loss_history.append(training_loss)
        testing_loss=(1/val)*(np.square(y_pred_test-y_test)).sum()+reg*(np.sum(w1*w1)+np.sum(w2*w2))
        loss_history_test.append(testing_loss)
       # calculating trainning and testing accuracies
       train_accuracy=1-(1/(10*batch_size))*(np.abs(np.argmax(y,axis=1)-np.argmax(y_pred,axis=1))).sum()
       train_acc_history.append(train_accuracy)

       test_accuracy=1-(1/(10*Nte))*(np.abs(np.argmax(y_test,axis=1)-np.argmax(y_pred_test,axis=1))).sum()
       val_acc_history.append(test_accuracy)
       # Print for every 10 iterations
       if t%10 == 0:
           print('epoch %d/%d: loss= %f || , test loss= %f ||, train accuracy= %f ||, test accuracy= %f ||, learning rate= %f ||'
           % (t,iterations,training_loss,testing_loss,train_accuracy,test_accuracy,lr))

       # Backward propagation
       #let's find the deravatives of the learnable parameters
       dy_pred=(1./batch_size)*2.0*(y_pred-y)#partial derivative of L w.r.t y_pred
       dw2=h.T.dot(dy_pred)+reg*w2
       db2=dy_pred.sum(axis=0)
       dh=dy_pred.dot(w2.T)
       dw1=x.T.dot(dh*h*(1-h))+reg*w1
       db1=(dh*h*(1-h)).sum(axis=0)
       #update weight matrices
       w1-=lr*dw1
       w2-=lr*dw2
       #update bias matrices
       b1-=lr*db1
       b2-=lr*db2
       lr_array.append(lr)
       lr*=lr_decay#decaying the learning rate
    return w1,b1,w2,b2,loss_history,loss_history_test,train_acc_history,val_acc_history,lr_array
x_train_2layer,y_train_2layer,x_test_2layer,y_test_2layer,K,Din,Ntr,Nte=preprocessing(normalize=False,reshape=True)
#Remove the normalization. Otherwise the model will not learn
iterations = 300#gradient descent iterations
lr = 1.4e-2#learning rate
lr_decay= 0.999
reg = 5e-6#lambda-regularization parameter
H=200 #hidden layer nodes
w1_2layer,b1_2layer,w2_2layer,b2_2layer,loss_history_2layer,loss_history_test_2layer,train_acc_history_2layer,val_acc_history_2layer,lr_array_2layer=layer_2(x_train_2layer,y_train_2layer,x_test_2layer,y_test_2layer,Din,lr,lr_decay,H,reg,K,Ntr,Nte)
```

Here gradients are computed in the direction from output to input layers and combined using chain rule.

- Input layer to hidden layer

$$h(W_1,\ b_1) = \frac{1}{1 + \exp\left(-W_1 x - b_1\right)}$$

- Hidden layer to the output layer

$$y\_pred(W_2, b_2) = W_2 h + b_2$$

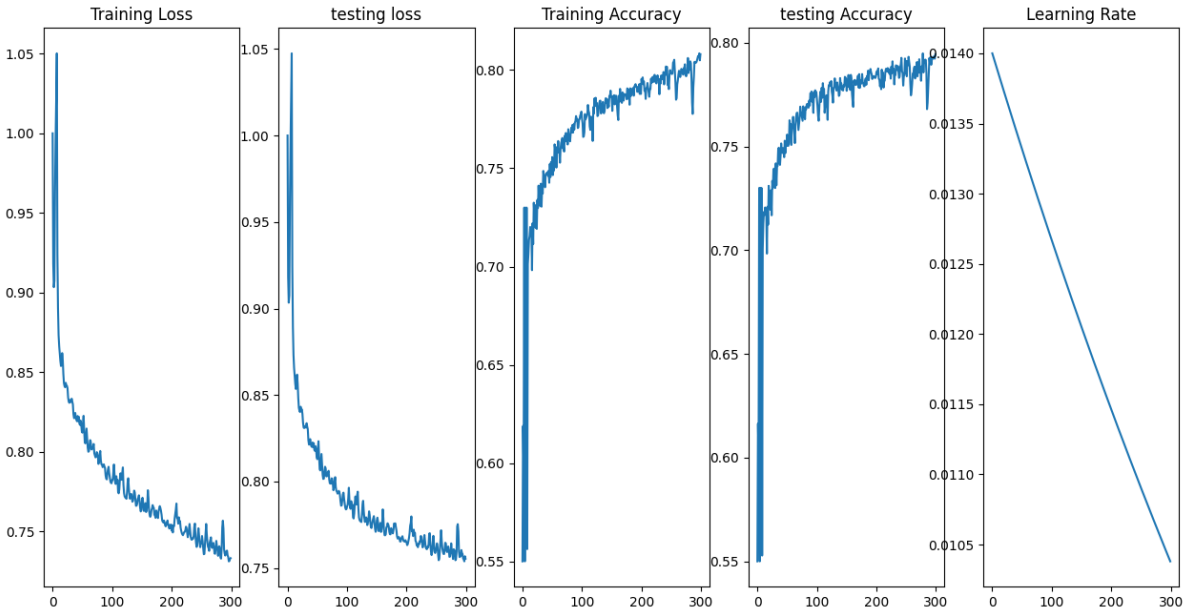
- Total number of learnable parameters in the network = (200 x 3072+200) + (10 x 200+10) = 616,610

We preprocess the input data set without pixel normalization to avoid underfitting.

As we can see from the below results when the number of iterations increase the loss decreases and accuracy increases. But the rate of the loss decreasing and accuracy increasing reduces with the iterations.

As we can see from the results, when we use 2 layer fully connected network instead of single layer as in part 1, we can reduce the train and test loss and increase training and testing accuracy. So, if we increase the number of layers further, we can achieve more and more accuracy and reduce loss.

Initial learning rate = 1.4 x $a0^{-2}$



Loss, testing loss, training accuracy, testing accuracy, learning rate of the 2 layer fully connected network for 300 epochs.

After 300 epochs, loss= 0.734748, test loss= 0.757078, train accuracy= 0.803568, test accuracy= 0.790080 and learning rate= 0.010474.

3) Stochastic gradient descent with a batch size of 500.

Here instead of using 50,000 samples I only uses 500 samples. The code for this part is as follows. (The full code is available in the link at cover page. Code part for plotting the graphs and calling the function is not included).

```python
#part 3
# Function for two layer dense network with stochastic gradient descent
def mini_batching(x_train,y_train,x_test,y_test,Din,lr,lr_decay,H,reg,K,Ntr,Nte):
    loss_history = []
    loss_history_test = []
    train_acc_history = []
    val_acc_history = []
    lr_array =[]
    seed = 0
    rng = np.random.default_rng(seed=seed)
    batch_size=500 #make batch size =500 for stochastic gradient descent

    std=1e-5
    #initializing weight and bias matrices for hidden layer
    w1 = std*np.random.randn(Din, H)
    b1 = np.zeros(H)
    #initializing weight and bias matrices for final layer
    w2 = std*np.random.randn(H, K)
    b2 = np.zeros(K)

    for t in range(iterations):
        training_loss = 0#making training loss, testing loss =0 for next epoch
        testing_loss=0
        train_accuracy=0#making training accuracy, testing accuracy =0 for next epoch
        test_accuracy=0
        for begin in range(0,Ntr,batch_size):#running 100 groups for each epoch
            indices = np.arange(Ntr)
            indices=indices[begin:begin+batch_size]#taking only 500 samples
            rng.shuffle(indices)# To avoid overfitting shuffle the training data set
            x=x_train[indices]
            y=y_train[indices]
            #forward propagation
            h=1/(1+np.exp(-(x.dot(w1)+b1)))
            y_pred=h.dot(w2)+b2
            h_test=1/(1+np.exp(-((x_test).dot(w1)+b1)))
            y_pred_test=h_test.dot(w2)+b2
            val=y_pred_test.shape[0]
```

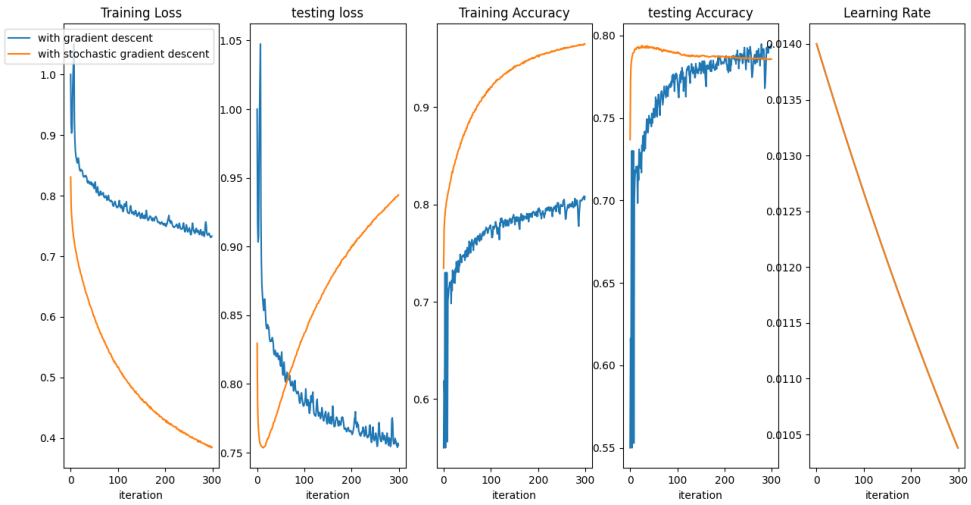
Part i

```python
            # calculating the training and testing loss for each mini batch
            mini_training_loss=(1/batch_size)*(np.square(y_pred-y)).sum()+reg*(np.sum(w1*w1)+np.sum(w2*w2))
            mini_testing_loss=(1/val)*(np.square(y_pred_test-y_test)).sum()+reg*(np.sum(w1*w1)+np.sum(w2*w2))
            training_loss+= mini_training_loss#updating training loss for each epoch
            testing_loss+= mini_testing_loss#updating testing loss for each epoch

            # calculating trainning and testing accuracies for each mini batch
            mini_train_accuracy=1-(1/(10*batch_size))*(np.abs(np.argmax(y,axis=1)-np.argmax(y_pred,axis=1))).sum()
            mini_test_accuracy=1-(1/(10*Nte))*(np.abs(np.argmax(y_test,axis=1)-np.argmax(y_pred_test,axis=1))).sum()
            train_accuracy+=mini_train_accuracy#updating training accuracy for each epoch
            test_accuracy+=mini_test_accuracy#updating testing accuracy for each epoch

            # Backward propagation
            #let's find the deravatives of the learnable parameters
            dy_pred=(1./batch_size)*2.0*(y_pred-y)#partial deravative of L w.r.t y_pred
            dw2=h.T.dot(dy_pred)+reg*w2
            db2=dy_pred.sum(axis=0)
            dh=dy_pred.dot(w2.T)
            dw1=x.T.dot(dh*h*(1-h))+reg*w1
            db1=(dh*h*(1-h)).sum(axis=0)
            #update weight matrices
            w1-=lr*dw1
            w2-=lr*dw2
            #update bias matrices
            b1-=lr*db1
            b2-=lr*db2
        #taking average of 100 groups to find accuracy
        train_accuracy=train_accuracy/(Ntr/batch_size)
        test_accuracy=(test_accuracy/(Ntr/batch_size))
        #taking average of 100 groups to find loss
        training_loss=training_loss/(Ntr/batch_size)
        testing_loss=testing_loss/(Ntr/batch_size)

        loss_history.append(training_loss)
        loss_history_test.append(testing_loss)
        train_acc_history.append(train_accuracy)
        val_acc_history.append(test_accuracy)
        lr_array.append(lr)
        lr*=lr_decay#decaying the learning rate
        # Print for every 10 iterations
        if t%1 == 0:
            print('epoch %d/%d: loss= %f || , test loss= %f ||, train accuracy= %f ||, test accuracy= %f ||, learning rate= %f ||'
            % (t,iterations,training_loss,testing_loss,train_accuracy,test_accuracy,lr))
    return w1,b1,w2,b2,loss_history,loss_history_test,train_acc_history,val_acc_history,lr_array
```

Part ii

(I have attached the code in two parts as part i and part ii due to space restrictions.)

Here we compute the gradients off the loss function w.r.t to the functions constructed only using 500 samples. It is advantageous to use stochastic gradient descent instead of gradient descent because we can avoid being stuck at a local minimum instead of global minimum when finding loss. By using SGD, we can compute errors and updates weights much faster as batch size is low. SGD often converges much faster compared to GD.



Loss, testing loss, training accuracy, testing accuracy, learning rate of the 2 layer fully connected network using gradient descent and stochastic gradient descent for 300 iterations.

After 300 iterations for stochastic gradient descent with batch size =500, loss= 0.385273, test loss= 0.937635, train accuracy= 0.964924, test accuracy= 0.785701 and learning rate= 0.010370.

By comparing the plots from part2(GD) and SGD, we can state that accuracy has been increased significantly when using SGD instead of GD. By observing the training loss plot we can state that SGD has reached the convergence much faster than GD and has much low loss than GD. (Here in testing loss graph with SGD, the testing loss has reduced and then increased. But we can get a continuously reducing graph if we reduce the learning rate, but then iteration time will increase.)

4) CNN

Here I preprocess data without normalization of pixels and without reshaping the images. As seen from the results, we can clearly state that CNN is overfitting. Here I used sgd optimizer with momentum. The code is as follows.

```
#part 4
from tensorflow.keras import layers,models,optimizers
x_train_CNN,y_train_CNN,x_test_CNN,y_test_CNN,K,Din,Ntr,Nte=preprocessing(normalize=True,reshape=False)
model = models.Sequential()
#layers
model.add(layers.Conv2D(32,(3,3),activation='relu',input_shape=(32,32,3)))
model.add(layers.MaxPool2D((2,2)))
model.add(layers.Conv2D(64,(3,3),activation='relu'))
model.add(layers.MaxPool2D((2,2)))
model.add(layers.Conv2D(64,(3,3),activation='relu'))
model.add(layers.MaxPool2D((2,2)))
model.add(layers.Flatten())
model.add(layers.Dense(64,activation='relu'))
model.add(layers.Dense(10))

sgd = keras.optimizers.SGD(lr=1.4e-2, momentum=0.9, decay=1e-6)#lr =learning rate, decay =learning rate decay
model.compile(optimizer=sgd,loss=tf.keras.losses.CategoricalCrossentropy(from_logits=True),metrics=["accuracy"])
print(model.summary())
history=model.fit(x_train_CNN,y_train_CNN,epochs=50,batch_size=50,validation_data=(x_test_CNN,y_test_CNN))

print(model.optimizer.get_config())
plt.plot(history.history['loss'],label='training loss')#plotting the graphs
plt.xlabel('epoch')
plt.legend(loc='lower right')
plt.xlim([0,50])
plt.show()
```

(In plotting code part, only code for plotting loss is included)

Here I used,

learning rate = 0.014

momentum = 0.9.

learning rate decay =1e-6

There are 73,418 total learnable parameters in this network.

After 50 epochs loss: 0.2491, accuracy: 0.9164, testing loss: 1.7655, testing accuracy: 0.6914