Q3. Implementation of Softmax Regression on MNIST dataset

Loss without hyper-parameter tuning:

Epoch 10/100, Loss: 0.5976267656385753

Epoch 20/100, Loss: 0.5801805834018345

Epoch 30/100, Loss: 0.604945025894442

Epoch 40/100, Loss: 0.7263698474239005

Epoch 50/100, Loss: 0.46117563587772453

Epoch 60/100, Loss: 0.6523747249916084

Epoch 70/100, Loss: 0.6616549823005253

Epoch 80/100, Loss: 0.6711249654571032

Epoch 90/100, Loss: 0.7584290646446238

Epoch 100/100, Loss: 0.6277892482457132

Test accuracy (Without hyper-parameter tuning):

82.03%

Optimized Weights:

[[-1.01095657e-03 1.72226860e-03 -3.14322070e-03 ... -1.01704224e-04 -1.30116709e-03 5.61577682e-04]

[1.68932485e-03 -1.16440151e-03 1.81708667e-03 ... -6.37295810e-04 8.74646362e-04 -9.39516535e-04]

[2.40641900e-03 -7.89381142e-04 -6.16642124e-04 ... -8.59988148e-04 -3.42771496e-03 -3.63132269e-05]

...

[-1.44959986e-02 -8.72157205e-04 1.48505982e-02 ... -6.11254590e-03 -1.77291833e-02 -4.07800349e-03]

[-3.63297280e-03 7.61382295e-04 3.16438110e-03 ... -2.73088965e-03 -5.69135540e-03 -5.93574005e-04]

[2.22597047e-04 2.10225585e-04 1.31918947e-03 ... 3.12822703e-05 -4.22701497e-04 -5.65677567e-04]]

Optimized Biased:

[

```
[ 0.19340259 -0.21192837 -0.1375754   0.08207723 -0.89087443
2.01589893 0.38832067 -0.06408568 -0.45808546 -0.91715009]
```

]

Accuracy: 0.8318333333333333

Test accuracy after hyper-parameter:

82.13%

Optimized Hyper-parameters (Unregularised)

Optimized Hyper-paramter:{'num_epochs': 150, 'learning_rate': 0.01, 'mini_batch': 128}

Optimized Weights:[

[0.00035023 0.01359112 -0.01841283 ... -0.0165678 0.00277887 -0.01371037]

[-0.00465253 0.01316789 -0.00163234 ... -0.00988817 -0.00109378 -0.00858768]

[0.01008089 -0.00874048 0.00515294 ... -0.01190488 -0.00665652 0.00944155]

...

[-0.05286076 0.00222857 0.05693199 ... -0.01191811 -0.06718057 -0.00355527]

[-0.02552242 0.01890952 0.02114426 ... 0.00161787 -0.04066923 -0.01735235]

[-0.01023523 0.00578018 -0.0046532 ... 0.00276225 0.0050645 -0.00253758]

]

Optimized Biased:[

[0.5142275 -0.59402088 -0.05933569 0.38942373 -1.3933948 2.57389403

0.50889462 0.0232403 -0.49627366 -1.46665516]

]

Accuracy0.8554166666666667

Test accuracy **unregularised** after hyper-parameter: 83.97%

Importing numpy

In [3]:
```
import numpy as np
```

Loading the train and test data using **np.load**

Spliting the training data into ratio of 80:20 for validation set.

Randomizing the data for Stochastic Gradient Descent.

In [4]:
```
training_data = np.reshape(np.load("fashion_mnist_train_images.npy"), (-1, 2
training_labels = np.load("fashion_mnist_train_labels.npy")
testing_data = np.reshape(np.load("fashion_mnist_test_images.npy"), (-1, 28*
testing_labels= np.load("fashion_mnist_test_labels.npy")

num_datapoints = training_data.shape[0]

split_index = int(0.8*num_datapoints)

indices = np.arange(num_datapoints)
np.random.shuffle(indices)

train_indices = indices[:split_index]
val_indices = indices[split_index:]

X_train, X_val = training_data[train_indices], training_data[val_indices]
y_train, y_val = training_labels[train_indices], training_labels[val_indices
```

Based on hint 1, normalizing all the pixel values of the both training and validation data.

Normalizing by dividing each pixel value by 255.

In [5]:
```
# based on Hint1
X_train = X_train / 255.0
X_val = X_val / 255.0
```

Defining input_size and output_size.

Initializing hyperparameters for model such as **learning_rate, batch_size and num_epochs**.

In [6]:
```
input_size = 28*28
output_size = 10 # num_classes

# hyperparameters

learning_rate = 0.01
batch_size = 64
num_epochs = 100
```

```
alpha = 0.01 # L2 regularization constant
```

Initializing weight matrix and bias vector.

In [7]:
```
# initilize weights and bias
W = np.random.randn(input_size,output_size) * 0.01  # to avoid large initial
B = np.zeros((1,output_size))
```

Defining **Softmax**

In [8]:
```
def softmax(Z):
    # to avoid large exponent values, we will use the deviation idealogy of

    exponent_z = np.exp(Z - np.max(Z,axis=1,keepdims=True))

    prediction = exponent_z/(np.sum(exponent_z,axis=1,keepdims=True))

    return prediction
```

Defining **Log Loss**

In [9]:
```
def loss(y_label,y_pred,W,alpha):

    batch_s = y_label .shape[0]  # to divide for average loss over batch

    prob = -np.log(y_pred[range(batch_s),y_label])  # for each sample we sou

    loss = np.sum(prob) / batch_s

    reg_loss = alpha/2 * np.sum(np.square(W))

    batch_loss = loss + reg_loss

    return batch_loss
```

Calculating **Gradient**

In [10]:
```
def gradient(X_batch,Y_batch,W,alpha,pred,B):

    batch_s = X_batch.shape[0]  # here its 64

    predi = pred

    predi[range(batch_s),Y_batch] -=1 # subtract each predicted true class l

    predi /= batch_s    # we compute the average loss per sample

    weight_grad = np.dot(X_batch.T,predi) + alpha * W
    bias_grad = np.sum(predi, axis = 0, keepdims=True)


    # update weights and bias using gradient descent
    W -= learning_rate * weight_grad
```

```
    B -= learning_rate * bias_grad


    return W, B
```

In [22]:
```
num_tr_samples = X_train.shape[0]
tr_indices = np.arange(num_tr_samples)

for epoch in range(num_epochs):
    np.random.shuffle(tr_indices)

    for num in range(0,num_tr_samples,batch_size):

            batch_index = tr_indices[num: num + batch_size]


            x_batch = X_train[batch_index]
            y_batch = y_train[batch_index]

            Z = np.dot(x_batch,W) + B   # batcsizex10

            pred = softmax(Z)

            batch_loss = loss(y_batch,pred,W,alpha)

            # next we will update the weights and bias

            W,B = gradient(x_batch,y_batch,W,alpha,pred,B)

    if(epoch + 1) % 10 == 0:
        print(f'Epoch {epoch+1}/{num_epochs}, Loss: {batch_loss}')
```

```
Epoch 10/100, Loss: 0.5976267656385753
Epoch 20/100, Loss: 0.5801805834018345
Epoch 30/100, Loss: 0.604945025894442
Epoch 40/100, Loss: 0.7263698474239005
Epoch 50/100, Loss: 0.46117563587772453
Epoch 60/100, Loss: 0.6523747249916084
Epoch 70/100, Loss: 0.6616549823005253
Epoch 80/100, Loss: 0.6711249654571032
Epoch 90/100, Loss: 0.7584290646446238
Epoch 100/100, Loss: 0.6277892482457132
```

Accuracy for test data (Without hyper-parameter Tuning)

In [20]:
```
def test(X):
    Z = np.dot(X, W) + B
    A = softmax(Z)
    return np.argmax(A, axis=1)

X_test = testing_data / 255.0
y_test = testing_labels
y_pred = test(X_test)
accuracy = np.mean(y_pred == y_test)
print(f'Test accuracy: {accuracy * 100:.2f}%')
```

```
Test accuracy: 82.03%
```

Hyperparameter Tuning

In [11]:
```
import numpy as np
```

In [12]:
```
training_data = np.reshape(np.load("fashion_mnist_train_images.npy"), (-1, 2
training_labels = np.load("fashion_mnist_train_labels.npy")
testing_data = np.reshape(np.load("fashion_mnist_test_images.npy"), (-1, 28*
testing_labels= np.load("fashion_mnist_test_labels.npy")

num_datapoints = training_data.shape[0]

split_index = int(0.8*num_datapoints)

indices = np.arange(num_datapoints)
np.random.shuffle(indices)

train_indices = indices[:split_index]
val_indices = indices[split_index:]

X_train, X_val = training_data[train_indices], training_data[val_indices]
y_train, y_val = training_labels[train_indices], training_labels[val_indices


# based on Hint1

X_train = X_train / 255.0
X_val = X_val / 255.0



input_size = 28*28
output_size = 10 # num_classes
```

In [13]:
```
def softmax(Z):
    # to avoid large exponent values, we will use the deviation idealogy of

    exponent_z = np.exp(Z - np.max(Z,axis=1,keepdims=True))

    prediction = exponent_z/(np.sum(exponent_z,axis=1,keepdims=True))

    return prediction
```

In [14]:
```
def loss(y_label,y_pred,W,alpha):

    batch_s = y_label .shape[0]  # to divide for average loss over batch

    prob = -np.log(y_pred[range(batch_s),y_label])  # for each sample we sou

    loss = np.sum(prob) / batch_s

    reg_loss = alpha/2 * np.sum(np.square(W))
```

```python
    batch_loss = loss + reg_loss

    return batch_loss
```

In [15]:
```python
def gradient(X_batch,Y_batch,W,alpha,pred,B,learning_rate):

    batch_s = X_batch.shape[0]  # here its 64

    predi = pred

    predi[range(batch_s),Y_batch] -=1 # subtract each predicted true class l

    predi /= batch_s    # we compute the average loss per sample

    weight_grad = np.dot(X_batch.T,predi) + alpha * W
    bias_grad = np.sum(predi, axis = 0, keepdims=True)


    # update weights and bias using gradient descent
    W -= learning_rate * weight_grad
    B -= learning_rate * bias_grad

    return W, B
```

In [18]:
```python
def train_softmax(X_train,y_train,num_epochs,batch_size,learning_rate,alpha)

    num_tr_samples = X_train.shape[0]
    tr_indices = np.arange(num_tr_samples)

    # initialize weight
    W = np.random.randn(input_size,output_size) * 0.01  # to avoid large ini
    B = np.zeros((1,output_size))

    for epoch in range(num_epochs):
        np.random.shuffle(tr_indices)

        for num in range(0,num_tr_samples,batch_size):

                batch_index = tr_indices[num: num + batch_size]


                x_batch = X_train[batch_index]
                y_batch = y_train[batch_index]

                Z = np.dot(x_batch,W) + B   # batcsizex10

                pred = softmax(Z)

                batch_loss = loss(y_batch,pred,W,alpha)

                # next we will update the weights and bias

                W,B = gradient(x_batch,y_batch,W,alpha,pred,B,learning_rate)
```

```
        # print(f'Epoch {epoch+1}/{num_epochs}, Loss: {batch_loss}')

    return W,B
```

In this we have taken 3 different values of each hyper-parameter such as **learning_rate**, **mini_batches_size**, **num_epochs_testing** and two different values **alpha**.

For each iteration a combination of different values have been taken.

Therefore in total 54 different combinations have been taken.

For each of the combination loss has been calculated.

The combination with least loss have been considered as the optimized values for hyperparameters.

In [38]:
```
# Lets tune our hyperparameters

def validation(X_train,y_train,X_val,y_val):

    learning_rates = [1e-4,1e-3,1e-2]
    mini_batch_sizes = [32, 64,128]
    num_epochs_testing = [50, 100,150]
    alpha = [1e-2,1e-1]

    best_accuracy = 0  # setting mse to positive infinity to ensure the firs
    best_hyperparams = {}   # dictionary to store the three HP parameters
    best_weights, best_bias = None, None

    for rate in learning_rates:
        for a in alpha:
            for batch in mini_batch_sizes:
                for epoch in num_epochs_testing:

                    weights, bias = train_softmax(X_train,y_train,epoch,batc



                    Z = np.dot(X_val, weights) + bias
                    A = softmax(Z)
                    y_pred = np.argmax(A, axis=1)

                    accuracy = np.mean(y_pred == y_val)

                    # print(f"Num_Epoch {epoch}, Batch_size {batch}, Learnin

                    if accuracy > best_accuracy:
                        best_accuracy = accuracy
                        best_hyperparameters = {'num_epochs': epoch,'learnin
                        best_weights,best_bias = weights,bias
```

```
    return best_hyperparameters,best_weights,best_bias,best_accuracy
```

The optimized combination of hyper-parameter are:

In [39]:
```
best_hyp,best_weights,best_bias,best_acc= validation(X_train,y_train,X_val,y

print("Optimized Hyper-paramter:" + str(best_hyp))
print("Optimized Weights:" + str(best_weights))
print("Optimized Biased:" + str(best_bias))
print("Accuracy" + str(best_acc))
```

```
Optimized Hyper-paramter:{'num_epochs': 50, 'learning_rate': 0.01, 'mini_bat
ch': 128, 'Alpha': {0.01}}
Optimized Weights:[[-1.01095657e-03  1.72226860e-03 -3.14322070e-03 ... -1.0
1704224e-04
  -1.30116709e-03  5.61577682e-04]
 [ 1.68932485e-03 -1.16440151e-03  1.81708667e-03 ... -6.37295810e-04
   8.74646362e-04 -9.39516535e-04]
 [ 2.40641900e-03 -7.89381142e-04 -6.16642124e-04 ... -8.59988148e-04
  -3.42771496e-03 -3.63132269e-05]
 ...
 [-1.44959986e-02 -8.72157205e-04  1.48505982e-02 ... -6.11254590e-03
  -1.77291833e-02 -4.07800349e-03]
 [-3.63297280e-03  7.61382295e-04  3.16438110e-03 ... -2.73088965e-03
  -5.69135540e-03 -5.93574005e-04]
 [ 2.22597047e-04  2.10225585e-04  1.31918947e-03 ...  3.12822703e-05
  -4.22701497e-04 -5.65677567e-04]]
Optimized Biased:[[ 0.19340259 -0.21192837 -0.1375754   0.08207723 -0.890874
43  2.01589893
   0.38832067 -0.06408568 -0.45808546 -0.91715009]]
Accuracy0.8318333333333333
```

In [ ]:
```
np.save("best_model_weights3.npy", best_weights)
np.save("best_model_bias3.npy", best_bias)
np.save("best_model_hyperparameter3",best_hyp)
np.save("best_model_accuracy",best_acc)
```

In [41]:
```
test_weights = np.load("best_model_weights3.npy")
test_bias = np.load("best_model_bias3.npy")
```

In [44]:
```
def test(X,W,B):
    Z = np.dot(X, W) + B
    A = softmax(Z)
    return np.argmax(A, axis=1)

X_test = testing_data / 255.0
y_test = testing_labels
y_pred = test(X_test,test_weights,test_bias)
accuracy = np.mean(y_pred == y_test)

print(f'Test accuracyafter hyper-parameter: {accuracy * 100:.2f}%')
```

```
Test accuracyafter hyper-parameter: 82.13%
```

In [19]:
```
# Lets tune our hyperparameters  - unregularised

def validation(X_train,y_train,X_val,y_val):

    learning_rates = [1e-4,1e-3,1e-2]
    mini_batch_sizes = [32, 64,128]
    num_epochs_testing = [50, 100,150]
    #alpha = [1e-2,1e-1]

    best_accuracy = 0  # setting mse to positive infinity to ensure the firs
    best_hyperparams = {}   # dictionary to store the three HP parameters
    best_weights, best_bias = None, None

    for rate in learning_rates:
            for batch in mini_batch_sizes:
                for epoch in num_epochs_testing:

                    weights, bias = train_softmax(X_train,y_train,epoch,batc



                    Z = np.dot(X_val, weights) + bias
                    A = softmax(Z)
                    y_pred = np.argmax(A, axis=1)

                    accuracy = np.mean(y_pred == y_val)

                    # print(f"Num_Epoch {epoch}, Batch_size {batch}, Learnin

                    if accuracy > best_accuracy:
                        best_accuracy_un = accuracy
                        best_hyperparameters_un = {'num_epochs': epoch,'lear
                        best_weights_un,best_bias_un = weights,bias

    return best_hyperparameters_un,best_weights_un,best_bias_un,best_accurac
```

In [20]:
```
best_hyp_un,best_weights_un,best_bias_un,best_acc_un= validation(X_train,y_t

print("Optimized Hyper-paramter:" + str(best_hyp_un))
print("Optimized Weights:" + str(best_weights_un))
print("Optimized Biased:" + str(best_bias_un))
print("Accuracy" + str(best_acc_un))
```

```
Optimized Hyper-paramter:{'num_epochs': 150, 'learning_rate': 0.01, 'mini_ba
tch': 128}
Optimized Weights:[[ 0.00035023  0.01359112 -0.01841283 ... -0.0165678   0.0
0277887
  -0.01371037]
 [-0.00465253  0.01316789 -0.00163234 ... -0.00988817 -0.00109378
  -0.00858768]
 [ 0.01008089 -0.00874048  0.00515294 ... -0.01190488 -0.00665652
   0.00944155]
 ...
 [-0.05286076  0.00222857  0.05693199 ... -0.01191811 -0.06718057
  -0.00355527]
 [-0.02552242  0.01890952  0.02114426 ...  0.00161787 -0.04066923
  -0.01735235]
 [-0.01023523  0.00578018 -0.0046532  ...  0.00276225  0.0050645
  -0.00253758]]
Optimized Biased:[[ 0.5142275  -0.59402088 -0.05933569  0.38942373 -1.393394
8   2.57389403
   0.50889462  0.0232403  -0.49627366 -1.46665516]]
Accuracy0.8554166666666667
```

In [22]:
```
np.save("best_model_weights_un.npy", best_weights_un)
np.save("best_model_bias_un.npy", best_bias_un)
np.save("best_model_hyperparameter_un.npy",best_hyp_un)
np.save("best_model_accuracy_un.npy",best_acc_un)
```

In [24]:
```
test_weights_un = np.load("best_model_weights_un.npy")
test_bias_un = np.load("best_model_bias_un.npy")
```

In [26]:
```
def test(X,W,B):
    Z = np.dot(X, W) + B
    A = softmax(Z)
    return np.argmax(A, axis=1)

X_test = testing_data / 255.0
y_test = testing_labels
y_pred = test(X_test,test_weights_un,test_bias_un)
accuracy = np.mean(y_pred == y_test)

print(f'Test accuracy unregularised after hyper-parameter: {accuracy * 100:.
```

```
Test accuracy unregularised after hyper-parameter: 83.97%
```