EN2550 – Assignment 04

Name : Rathnayaka R.G.H.V.

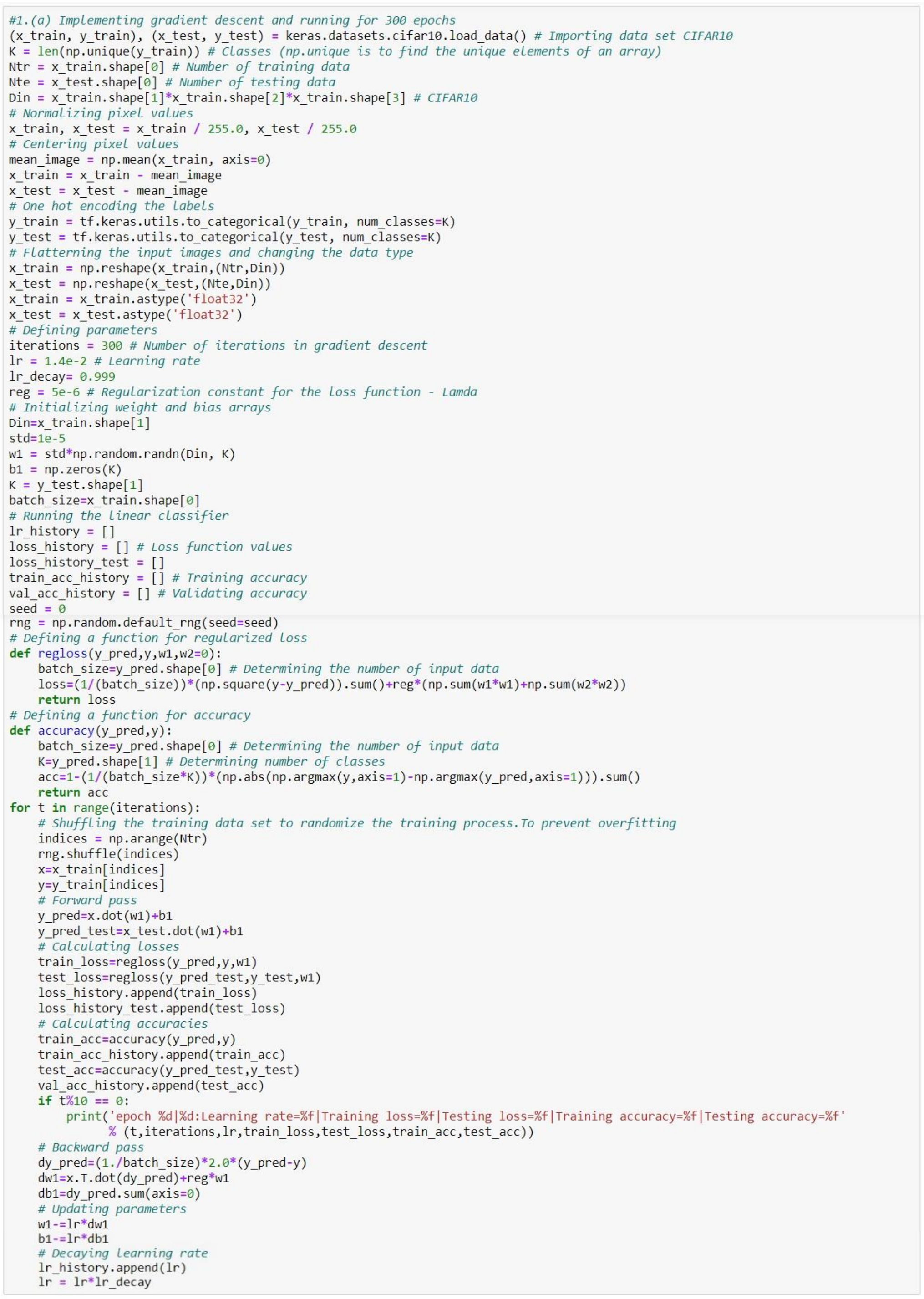
Index No : 180529E

01.Linear Classifier

CIFAR10 which contains 50000 32x32x3 training images and 10 different classes is used as the dataset here. Tensorflow is used to import the data set to python. For this linear classifier, f(x) = Wx+b is used as the score function. Loss function is taken as the mean sum of squared error function.

(a). Implementation of gradient descent and running for 300 epochs.

For the ease in coding two separate functions are used for loss function and accuracy function. Normalized difference between predicted label and true label with highest probability is used in calculating accuracy. The linear classifier is implemented as beside in the Figure (01) for 300 iterations by taking the initial learning rate 0.014.

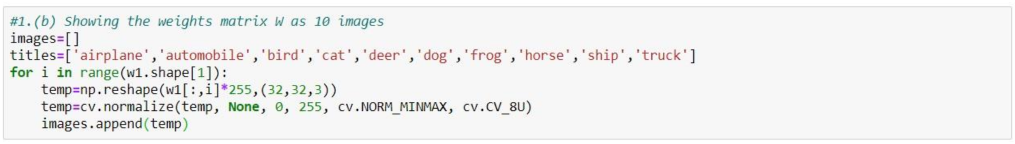


```
#1.(a) Implementing gradient descent and running for 300 epochs
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data() # Importing data set CIFAR10
K = len(np.unique(y_train)) # Classes (np.unique is to find the unique elements of an array)
Ntr = x_train.shape[0] # Number of training data
Nte = x_test.shape[0] # Number of testing data
Din = x_train.shape[1]*x_train.shape[2]*x_train.shape[3] # CIFAR10
# Normalizing pixel values
x_train, x_test = x_train / 255.0, x_test / 255.0
# Centering pixel values
mean_image = np.mean(x_train, axis=0)
x_train = x_train - mean_image
x_test = x_test - mean_image
# One hot encoding the labels
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 and changing the data type
x_train = np.reshape(x_train,(Ntr,Din))
x_test = np.reshape(x_test,(Nte,Din))
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
# Defining parameters
iterations = 300 # Number of iterations in gradient descent
lr = 1.4e-2 # Learning rate
lr_decay= 0.999
reg = 5e-6 # Regularization constant for the loss function - Lamda
# Initializing weight and bias arrays
Din=x_train.shape[1]
std=1e-5
w1 = std*np.random.randn(Din, K)
b1 = np.zeros(K)
K = y_test.shape[1]
batch_size=x_train.shape[0]
# Running the linear classifier
lr_history = []
loss_history = [] # Loss function values
loss_history_test = []
train_acc_history = [] # Training accuracy
val_acc_history = [] # Validating accuracy
seed = 0
rng = np.random.default_rng(seed=seed)
# Defining a function for regularized loss
def regloss(y_pred,y,w1,w2=0):
    batch_size=y_pred.shape[0] # Determining the number of input data
    loss=(1/(batch_size))*(np.square(y-y_pred)).sum()+reg*(np.sum(w1*w1)+np.sum(w2*w2))
    return loss
# Defining a function for accuracy
def accuracy(y_pred,y):
    batch_size=y_pred.shape[0] # Determining the number of input data
    K=y_pred.shape[1] # Determining number of classes
    acc=1-(1/(batch_size*K))*(np.abs(np.argmax(y,axis=1)-np.argmax(y_pred,axis=1))).sum()
    return acc
for t in range(iterations):
    # Shuffling the training data set to randomize the training process.To prevent overfitting
    indices = np.arange(Ntr)
    rng.shuffle(indices)
    x=x_train[indices]
    y=y_train[indices]
    # Forward pass
    y_pred=x.dot(w1)+b1
    y_pred_test=x_test.dot(w1)+b1
    # Calculating losses
    train_loss=regloss(y_pred,y,w1)
    test_loss=regloss(y_pred_test,y_test,w1)
    loss_history.append(train_loss)
    loss_history_test.append(test_loss)
    # Calculating accuracies
    train_acc=accuracy(y_pred,y)
    train_acc_history.append(train_acc)
    test_acc=accuracy(y_pred_test,y_test)
    val_acc_history.append(test_acc)
    if t%10 == 0:
        print('epoch %d|%d:Learning rate=%f|Training loss=%f|Testing loss=%f|Training accuracy=%f|Testing accuracy=%f'
              % (t,iterations,lr,train_loss,test_loss,train_acc,test_acc))
    # Backward pass
    dy_pred=(1./batch_size)*2.0*(y_pred-y)
    dw1=x.T.dot(dy_pred)+reg*w1
    db1=dy_pred.sum(axis=0)
    # Updating parameters
    w1-=lr*dw1
    b1-=lr*db1
    # Decaying learning rate
    lr_history.append(lr)
    lr = lr*lr_decay
```

Figure (01) - Linear Classification using Gradient Descent

(b).Weight matrix W as 10 images.

Weight array w1, is in the shape of 3072x10 where 3072 represents the length of the flattened input images while 10 represents the number of classes. Therefore, there is a respective class in CIFAR10 data set for each node of the 10 node linear layer. Therefore, each node should able to calculate a score to an image by comparing it with the corresponding class at the end of training. As it is taken the vector product between the input image and the node, node should be a similar image to label of its class. We can identify a similarity by reshaping the each column of the weight array back to an image. The key components in weight matrix implementation is given under *Figure (02)* while the 10 images are given below in *Figure (03)*.



```
#1.(b) Showing the weights matrix W as 10 images
images=[]
titles=['airplane','automobile','bird','cat','deer','dog','frog','horse','ship','truck']
for i in range(w1.shape[1]):
    temp=np.reshape(w1[:,i]*255,(32,32,3))
    temp=cv.normalize(temp, None, 0, 255, cv.NORM_MINMAX, cv.CV_8U)
    images.append(temp)
```

Figure (02) - Showing the weight matrix W as 10 images

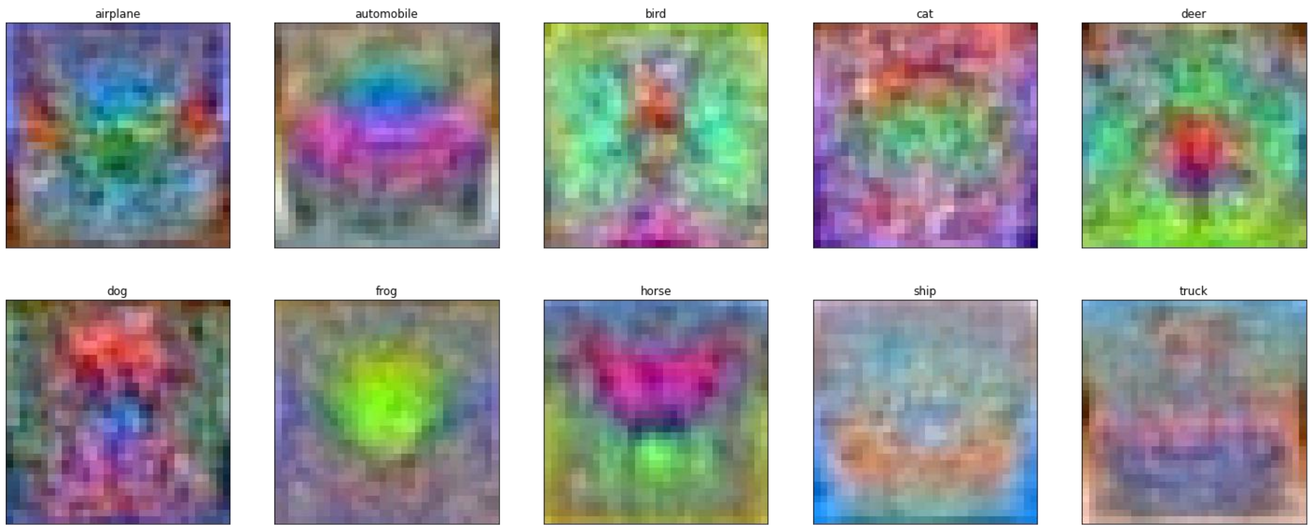


Figure (03) – Weight Matrix W as 10 images

(c). Learning rate, training loss, testing loss, training accuracy and testing accuracy variation with number of epochs.

Initial learning rate = 0.014

Focusing on the variation of losses and accuracies of both training data set and testing data set, it is clearly noticed that there is a huge slope at the beginning and a gentle slope at the last part. The reason for this is having less number of nodes and layers. As a conclusion we can come to a decision that the linear classifier is neither overfitting nor underfitting. The variations are clearly depicted in *Figure (04).*

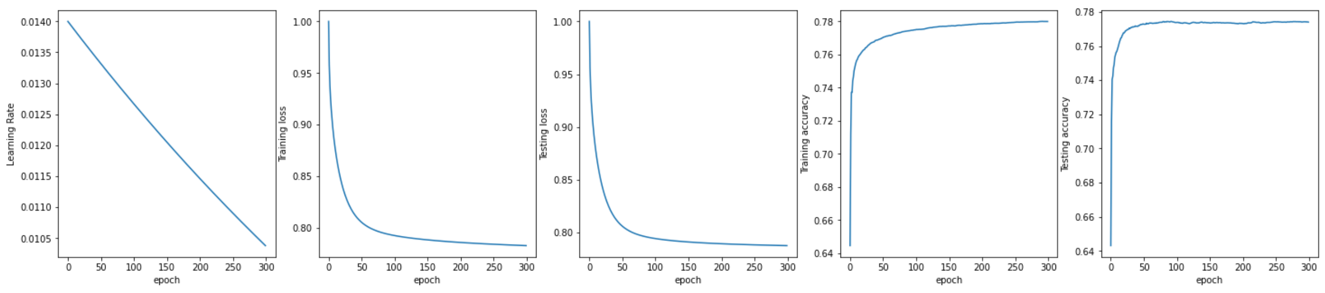


Figure (04) – variation of learning rate, training loss, testing loss, training accuracy and testing accuracy with the number of epochs.

The results after 300 iterations in linear classification using gradient descent are,

```
epoch 290|300: Learning rate=0.010474|Training loss=0.783118|Testing loss=0.787731
|Training accuracy=0.779980|Testing accuracy=0.774130
```

02.Two layer fully connected network

(a). Implementation of gradient descent and running for 300 epochs.

In a two layer fully connected network gradients are computed from output layer to input layer direction. After that they are combined using chain rule. Gradient from input layer to hidden layer h (w_1, b_1) is calculates as:

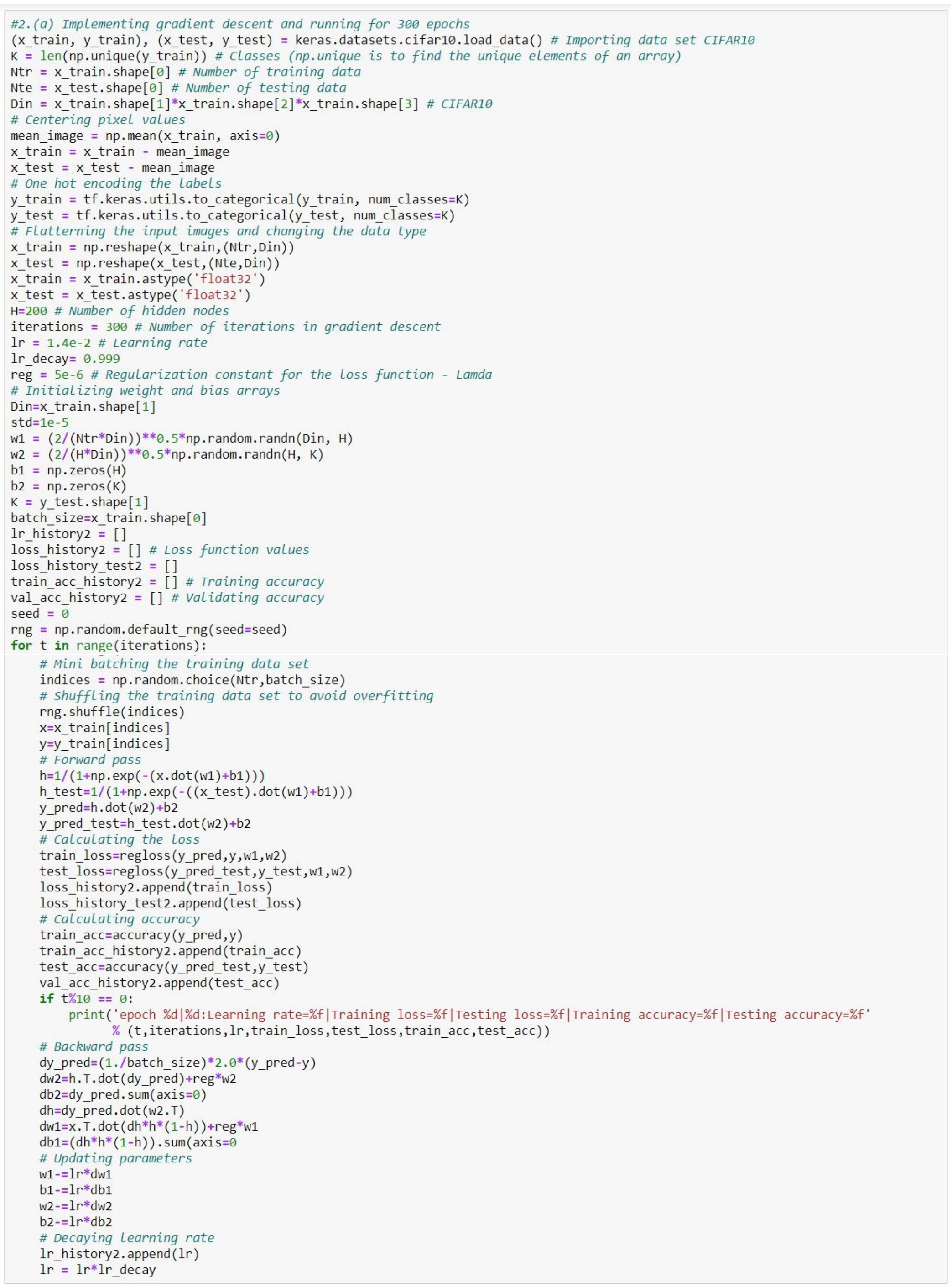
$$h(W_1,\ b_1) = \frac{1}{1+\exp\left(-W_1x - b_1\right)}$$

Gradient from hidden layer to the output layer y_pred (W_2, b_2) is calculated as:

$$y\_pred(W_2, b_2) = \ W_2h + b_2$$

To avoid underfitting, the input data set is preprocessed without pixel normalization. So, normalization part in the question number 1 has been omitted. Instead of that sigmoid function is used to normalize the score of each of the hidden layer.

The total number of learnable parameters in the network = (200 x 3072 + 200) + (10 x 200 + 10) = 616610



```
#2.(a) Implementing gradient descent and running for 300 epochs
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data() # Importing data set CIFAR10
K = len(np.unique(y_train)) # Classes (np.unique is to find the unique elements of an array)
Ntr = x_train.shape[0] # Number of training data
Nte = x_test.shape[0] # Number of testing data
Din = x_train.shape[1]*x_train.shape[2]*x_train.shape[3] # CIFAR10
# Centering pixel values
mean_image = np.mean(x_train, axis=0)
x_train = x_train - mean_image
x_test = x_test - mean_image
# One hot encoding the labels
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 and changing the data type
x_train = np.reshape(x_train,(Ntr,Din))
x_test = np.reshape(x_test,(Nte,Din))
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
H=200 # Number of hidden nodes
iterations = 300 # Number of iterations in gradient descent
lr = 1.4e-2 # Learning rate
lr_decay= 0.999
reg = 5e-6 # Regularization constant for the loss function - Lamda
# Initializing weight and bias arrays
Din=x_train.shape[1]
std=1e-5
w1 = (2/(Ntr*Din))**0.5*np.random.randn(Din, H)
w2 = (2/(H*Din))**0.5*np.random.randn(H, K)
b1 = np.zeros(H)
b2 = np.zeros(K)
K = y_test.shape[1]
batch_size=x_train.shape[0]
lr_history2 = []
loss_history2 = [] # Loss function values
loss_history_test2 = []
train_acc_history2 = [] # Training accuracy
val_acc_history2 = [] # Validating accuracy
seed = 0
rng = np.random.default_rng(seed=seed)
for t in range(iterations):
    # Mini batching the training data set
    indices = np.random.choice(Ntr,batch_size)
    # Shuffling the training data set to avoid overfitting
    rng.shuffle(indices)
    x=x_train[indices]
    y=y_train[indices]
    # Forward pass
    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
    # Calculating the loss
    train_loss=regloss(y_pred,y,w1,w2)
    test_loss=regloss(y_pred_test,y_test,w1,w2)
    loss_history2.append(train_loss)
    loss_history_test2.append(test_loss)
    # Calculating accuracy
    train_acc=accuracy(y_pred,y)
    train_acc_history2.append(train_acc)
    test_acc=accuracy(y_pred_test,y_test)
    val_acc_history2.append(test_acc)
    if t%10 == 0:
        print('epoch %d|%d:Learning rate=%f|Training loss=%f|Testing loss=%f|Training accuracy=%f|Testing accuracy=%f'
              % (t,iterations,lr,train_loss,test_loss,train_acc,test_acc))
    # Backward pass
    dy_pred=(1./batch_size)*2.0*(y_pred-y)
    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
    # Updating parameters
    w1-=lr*dw1
    b1-=lr*db1
    w2-=lr*dw2
    b2-=lr*db2
    # Decaying learning rate
    lr_history2.append(lr)
    lr = lr*lr_decay
```

Figure (05) – Gradient Descent for a two layer fully connected network

The code beside shown in Figure (05) shows the gradient descent implementation of two layer fully connected network.

<u>(b). Learning rate, training loss, testing loss, training accuracy and testing accuracy variation with number of epochs.</u>

Initial learning rate = 0.014

The Figure (06) shows the variation of each parameter with the number of iterations. Results show that with the increase of number of iterations both training and the testing losses decrease while both training and testing accuracies increase. But the rate of loss decreasing and accuracy decreasing reduces with the increase of number of iterations. From the results it is clear that two layer network has a reduced training and testing losses when compares it with a single layer network. It is also clear that the training and the testing accuracies in two layer fully connected network is comparatively higher than that of one layer network. So, it is very clear that increasing the number of layers paves the way for reducing losses and gaining a higher accuracy. Plots shows oscillations just because of the increase in number of hidden layers and nodes.

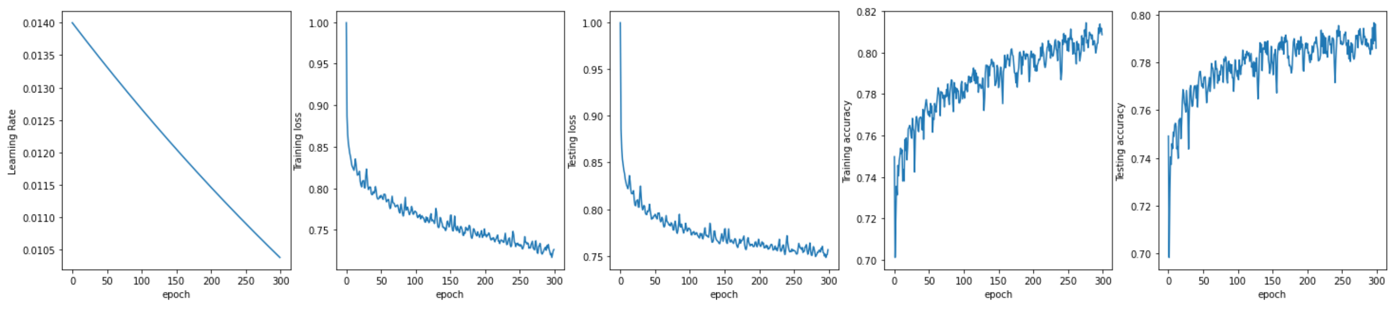


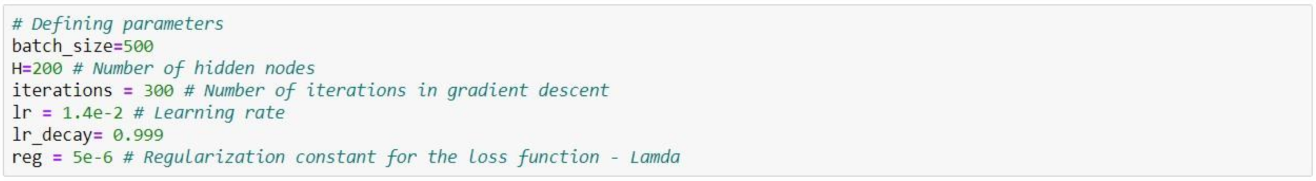
Figure (06) - variation of learning rate, training loss, testing loss, training accuracy and testing accuracy with the number of epochs.

The results after 300 iterations in two layer fully connected network using gradient descent are,

```
epoch 290|300: Learning rate=0.010474|Training loss=0.728806|Testing loss=0.757132
|Training accuracy=0.799808|Testing accuracy=0.785240
```

03. Stochastic gradient descent (SGD) with a batch size of 500

Instead of using 50000 samples here, 500 samples are selected randomly. The only modification done to the code segment above are depicted in the *Figure (07)* below.



```
# Defining parameters
batch_size=500
H=200 # Number of hidden nodes
iterations = 300 # Number of iterations in gradient descent
lr = 1.4e-2 # Learning rate
lr_decay= 0.999
reg = 5e-6 # Regularization constant for the loss function - Lamda
```

Figure (07) – Modifications done to code in Figure (05)

(a). Learning rate, Training loss, testing loss, training accuracy and testing accuracy variation with number of epochs.

Figure (08) below will show the variation of learning rate, training loss, testing loss, training accuracy and testing accuracy with number of epochs. Here, gradients are calculated with respective to only a batch size of 500. It is very advantageous of using stochastic gradient descent than normal gradient descent because, it is able to avoid being stuck at a local minimum instead of global minimum when finding loss. As the batch size is low, stochastic gradient descent can compute errors and update weights much faster than gradient descent.

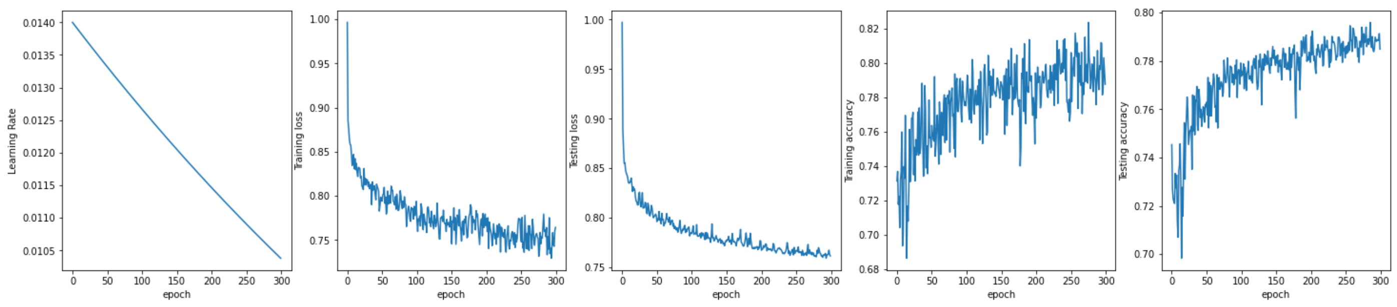


Figure (08) - variation of learning rate, training loss, testing loss, training accuracy and testing accuracy with the number of epochs

The results after 300 iterations using stochastic gradient descent are,

```
epoch 290|300: Learning rate=0.010474|Training loss=0.741685|Testing loss=0.76
1266|Training accuracy=0.794200|Testing accuracy=0.791420
```

(b). Comparison of results

The *Figure (09)* below will clearly depict the variation of learning rate, training loss, testing loss, training accuracy and testing accuracy with the number of epochs with and without mini batching.

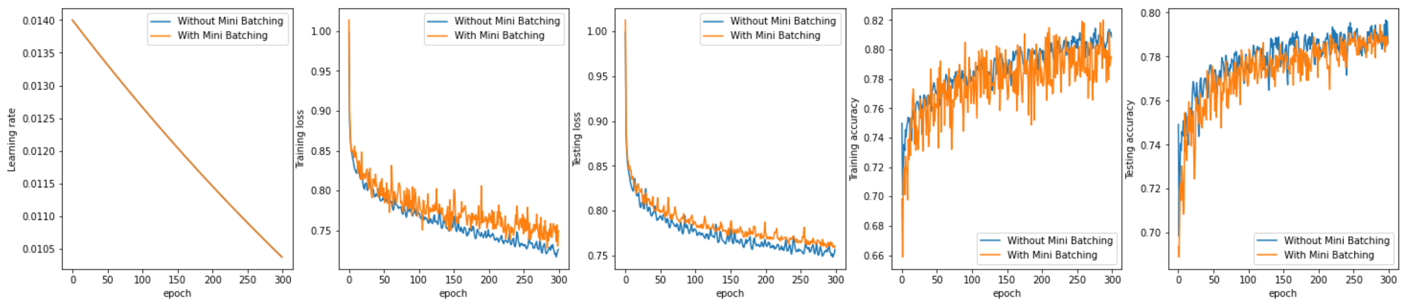


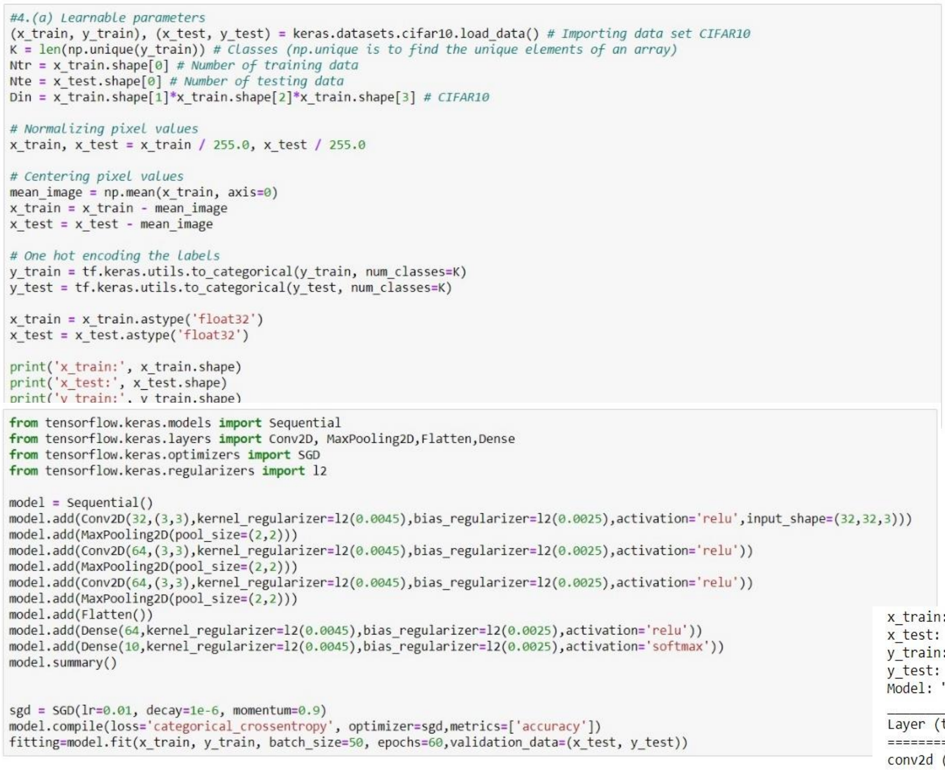
Figure (09) - variation of learning rate, training loss, testing loss, training accuracy and testing accuracy with the number of epochs with and without mini batching

Going deep inside to the plots it shows that there are huge oscillations in both loss and accuracy than the initial one. This will happen mainly because of the random selection of data from mini batches. Algorithm functions very fast as mentioned in the above section. This is a good tradeoff due to the sufficient similarity with and without mini batching although there are oscillations. Accuracy with mini batching is somewhat higher than the earlier stage. From the plots we can determine that with mini batching convergence can be taken much faster than earlier also. Loss function in GSD is much noisier than GD. This is the highlighting disadvantage in stochastic gradient descent.

04. CNN

In constructing a CNN, preprocessing is done without pixel normalization and image reshaping. The code is given beside under *Figure (10).*

Here, SDG is used with batch size 50 and CategoricalCrossentropy as the loss. Results of this code shows the followings.



```
#4.(a) Learnable parameters
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data() # Importing data set CIFAR10
K = len(np.unique(y_train)) # Classes (np.unique is to find the unique elements of an array)
Ntr = x_train.shape[0] # Number of training data
Nte = x_test.shape[0] # Number of testing data
Din = x_train.shape[1]*x_train.shape[2]*x_train.shape[3] # CIFAR10

# Normalizing pixel values
x_train, x_test = x_train / 255.0, x_test / 255.0

# Centering pixel values
mean_image = np.mean(x_train, axis=0)
x_train = x_train - mean_image
x_test = x_test - mean_image

# One hot encoding the labels
y_train = tf.keras.utils.to_categorical(y_train, num_classes=K)
y_test = tf.keras.utils.to_categorical(y_test, num_classes=K)

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')

print('x_train:', x_train.shape)
print('x_test:', x_test.shape)
print('y_train:', y_train.shape)
```

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D,Flatten,Dense
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.regularizers import l2

model = Sequential()
model.add(Conv2D(32,(3,3),kernel_regularizer=l2(0.0045),bias_regularizer=l2(0.0025),activation='relu',input_shape=(32,32,3)))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Conv2D(64,(3,3),kernel_regularizer=l2(0.0045),bias_regularizer=l2(0.0025),activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Conv2D(64,(3,3),kernel_regularizer=l2(0.0045),bias_regularizer=l2(0.0025),activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Flatten())
model.add(Dense(64,kernel_regularizer=l2(0.0045),bias_regularizer=l2(0.0025),activation='relu'))
model.add(Dense(10,kernel_regularizer=l2(0.0045),bias_regularizer=l2(0.0025),activation='softmax'))
model.summary()


sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9)
model.compile(loss='categorical_crossentropy', optimizer=sgd,metrics=['accuracy'])
fitting=model.fit(x_train, y_train, batch_size=50, epochs=60,validation_data=(x_test, y_test))
```

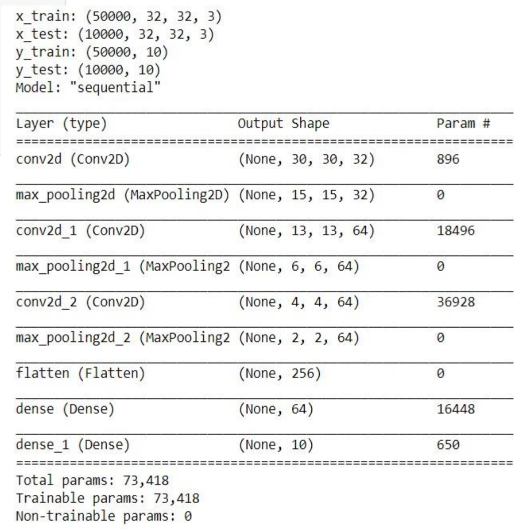
Figure (10) – CNN using Keras.models.Squential

<u>(a). Learnable parameters</u>

Details regarding learnable parameters in this network are available in *Figure (11)* below.

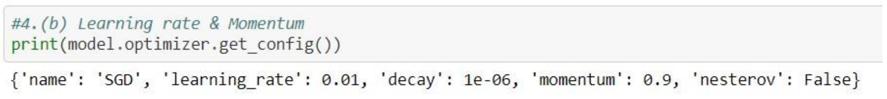
```
Total params: 73,418
```

Figure (11) – Learnable parameters in CNN



```
x_train: (50000, 32, 32, 3)
x_test: (10000, 32, 32, 3)
y_train: (50000, 10)
y_test: (10000, 10)
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 30, 30, 32)        896
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 15, 15, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 13, 13, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 6, 6, 64)          0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 4, 4, 64)          36928
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 2, 2, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 256)               0
_________________________________________________________________
dense (Dense)                (None, 64)                16448
_________________________________________________________________
dense_1 (Dense)              (None, 10)                650
=================================================================
Total params: 73,418
Trainable params: 73,418
Non-trainable params: 0
```

<u>(b). Learning rate and momentum</u>



```
#4.(b) Learning rate & Momentum
print(model.optimizer.get_config())
```

```
{'name': 'SGD', 'learning_rate': 0.01, 'decay': 1e-06, 'momentum': 0.9, 'nesterov': False}
```

Figure (12) – Learning rate and Momentum

<u>(c). Training loss, testing loss, training accuracy and testing accuracy variation with number of epochs.</u>

The variation of training loss, testing loss, training accuracy and testing accuracy with the number of epochs in this CNN is given under *Figure (13)* below. From then we can clearly point out that CNN is overfitting.

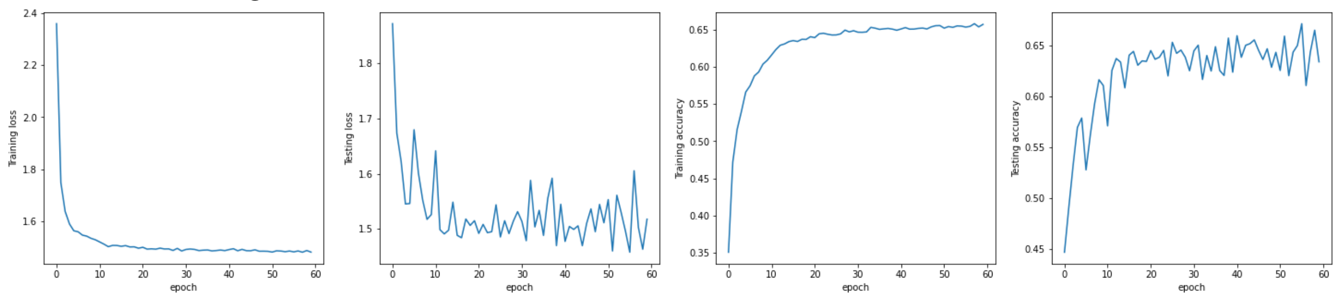


Figure (13) - variation of training loss, testing loss, training accuracy and testing accuracy with the number of epochs

The results after 60 iterations in CNN are,

```
Epoch 60/60
1000/1000 [==============================] - 69s 69ms/step - loss: 1.4810 - accura
cy: 0.6588 - val_loss: 1.5177 - val_accuracy: 0.6337
```

The complete code for this assignment is available at : https://github.com/hiruna-vidumina/EN2550---Fundamentals-of-Image-Processing-and-Machine-Vision/blob/main/Assignment%204/a04.ipynb

-The enD-