EN 2550: Assignment 04

Sandeepa H.K.C.A. 180564F

1.0 Linear Classifier

While training our neural network this is a common optimization algorithm. In this case, it allows us to adjust the learning parameters of the network such a way its output deviation is minimized.

To find output for given data sheet we choose linear classifier of a score function of $f(x) = Wx + b$ with a loss function of $L = (\frac{1}{N}) \sum (predicted\ output - actual\ output)^2 + R(w)$ where $R(w) = Regularization\ term ==> R(w) = \lambda \sum w^2$

```
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()
K = len(np.unique(y_train)) # Return the unique elements of a tratining output set and take it length as Classes
Ntr = x_train.shape[0] # number of training examples
Nte = x_test.shape[0] # number of testing examples
Din = 3072 # By CIFAR10 data set with 32 x 32 x 3 color images

# Normalize pixel values: Image data preprocessing
x_train, x_test = x_train / 255.0, x_test / 255.0
mean_image = np.mean(x_train, axis=0) # axis=0: mean of a column; Mean of each pixel
x_train = x_train - mean_image
x_test = x_test - mean_image

y_train = tf.keras.utils.to_categorical(y_train, num_classes=K)

#This function returns a matrix of binary values (either ‘1’ or ‘0’). It has number of rows equal to the length of
#the input vector and number of columns equal to the number of classes.

y_test = tf.keras.utils.to_categorical(y_test, num_classes=K)

#reshape/flatten the data
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')

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

This is preprocessing part of the code. Initially take the data from CIFAR10 and assign them to variables. (training and testing). Then normalization process is happening & create Y data as binary matrix and reshape(flatten) X data set because other wise we cannot use them in matrix multiplication. Since this is a 32x32x3 images it has 3072 pixels.

Then creating Weight matrix(w) and bias matrix(b) with correct size. This has not hidden layer, so weight matrix should have 10x3072 and b should have a size of 10. After selecting them initialize their values as randomly generated normal distribution with zero mean and given standard deviation. While running for given epochs, to avoid overfitting we should shuffle indices. Then create an output function as linear classifier for both training set and testing set. Loss is taken as the code. Take error and normalize it and subtract by 01 for take accuracy. All these values append to list. This is a forward propagation, and the system is not perfectly learning. So, we use backward propagation to learn our learning parameters for best accuracy. Take the partial derivations of loss w.r.t. learning parameters and use chain rule. Then apply them as the gradient descent algorithm says. ($w = w - \alpha\nabla w$) Then plot them.

```
batch_size = Ntr # for gradient descent optimization batch size is equal to number of training set
iterations = 300
# epochs
lr = 1.4e-2 # the learning rate alpha
lr_decay= 0.999
reg = 5e-6 #the regularization constant - lamda
```

```
std=1e-5 # standard deviation of normal distributaion
w1 = std*np.random.randn(Din, K) # initialize the w matrix with random values
b1 = np.zeros(K)

for t in range(iterations):
    batch_indices = np.random.choice(Ntr, batch_size)
    x = x_train[batch_indices]
    y = y_train[batch_indices]

    #forward
    y_pred = x.dot(w1) + b1
    y_pred_test = x_test.dot(w1) + b1

    loss = (1/batch_size)*(np.square(y - y_pred)).sum() + reg*(np.sum(w1*w1))
    loss_history.append(loss)

    # compute the accuracy as percentage
    training_acc = 100*(1 - (1/(batch_size*K))*(np.abs(np.argmax(y,axis=1) - np.argmax(y_pred,axis=1))).sum())
    testing_acc = 100*(1 - (1/(Nte*K))*(np.abs(np.argmax(y_test,axis=1) - np.argmax(y_pred_test,axis=1))).sum())
    train_acc_history.append(training_acc)
    val_acc_history.append(testing_acc)

    if t%10 == 0:
        print("iteration %d / %d| loss %f| training accuracy %f| testing accuracy %f" % (t, iterations, loss, training_acc, testing_
    # Backward
    dy_pred = (1./batch_size)*2.0*(y_pred-y) # partial derivative w.r.t y_predicted
    dw1 = x.T.dot(dy_pred) + reg*w1
    db1 = dy_pred.sum(axis=0)
    #updating learning parameters
    w1 -= lr*dw1
    b1 -= lr*db1
    lr *= lr_decay
```

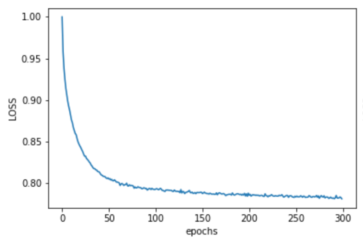


figure 01: training loss

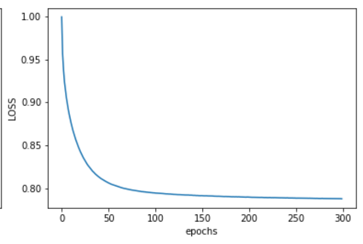


figure 02: testing loss

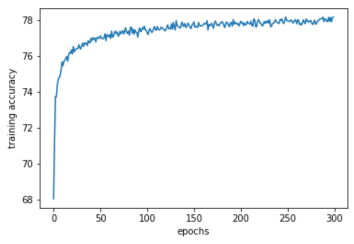


figure 03: training accuracy

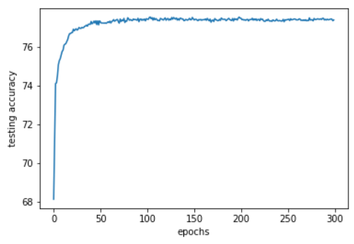


figure 04: testing accuracy

For initial learning rate 0.0140, training loss = 0.999999, testing loss = 0.999993, training accuracy = 68.065800, testing accuracy = 68.157000 and you can find my GitHub repository for full details.

After that I have to shoe my weight matrix as 10 images with labels using this part of the code.

```
imgs_for_w1=[]
titles_of_imgs = ['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)
    imgs_for_w1.append(temp)
fig,ax = plt.subplots(2,5,figsize=(20,8))
# show resultant image as a type of 2 x 5
for i in range(2):
    for j in range(5):
        ax[i,j].imshow(imgs_for_w1[i*5+j], vmin=0, vmax=255)
        ax[i,j].set_title(titles_of_imgs[i*5+j])
plt.show()
fig.savefig('img_for_w1',transparent=True)
```

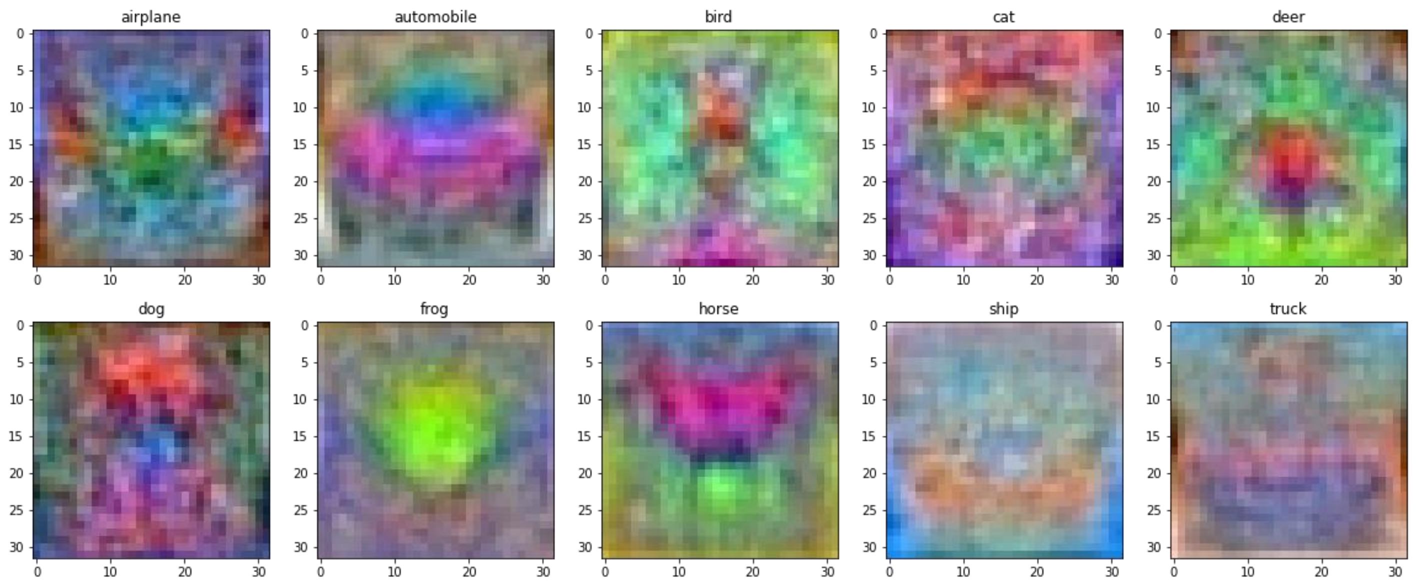


figure 05: W as 10 images

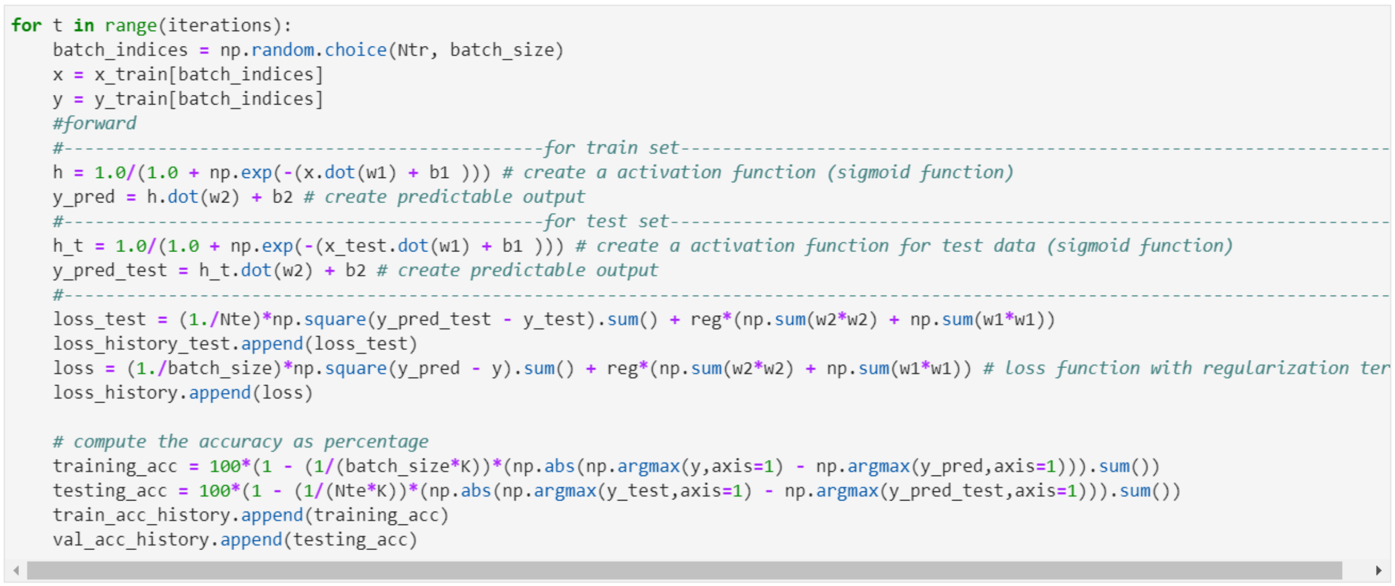
2.0 Two Layer Fully Connected Network

In this case there are two layers with 200 of hidden nodes. In this network there are two major parts. 1st one is activation and second one is output. For the activation there are lot of activation functions such as Sigmoid & ReLU. In this case Sigmoid function is used.

$$h(w_1, b_1) = 1/(1 + e^{-(w_1 x + b_1)})$$

$$y_{out} = hw_2 + b_2$$

To find forward propagation please follow the code below. It just going through above equations with help of numpy. It should compute for both training set and testing set. In backward propagation process there are many learnable parameters than previous one so partial derivative is computed as code using chain rule. Then update the parameters.



```python
for t in range(iterations):
    batch_indices = np.random.choice(Ntr, batch_size)
    x = x_train[batch_indices]
    y = y_train[batch_indices]
    #forward
    #----------------------------------------------for train set--------------------------------------------------------------
    h = 1.0/(1.0 + np.exp(-(x.dot(w1) + b1 ))) # create a activation function (sigmoid function)
    y_pred = h.dot(w2) + b2 # create predictable output
    #---------------------------------------------for test set---------------------------------------------------------------
    h_t = 1.0/(1.0 + np.exp(-(x_test.dot(w1) + b1 ))) # create a activation function for test data (sigmoid function)
    y_pred_test = h_t.dot(w2) + b2 # create predictable output
    #-----------------------------------------------------------------------------------------------------------------------------
    loss_test = (1./Nte)*np.square(y_pred_test - y_test).sum() + reg*(np.sum(w2*w2) + np.sum(w1*w1))
    loss_history_test.append(loss_test)
    loss = (1./batch_size)*np.square(y_pred - y).sum() + reg*(np.sum(w2*w2) + np.sum(w1*w1)) # loss function with regularization ter
    loss_history.append(loss)

    # compute the accuracy as percentage
    training_acc = 100*(1 - (1/(batch_size*K))*(np.abs(np.argmax(y,axis=1) - np.argmax(y_pred,axis=1))).sum())
    testing_acc = 100*(1 - (1/(Nte*K))*(np.abs(np.argmax(y_test,axis=1) - np.argmax(y_pred_test,axis=1))).sum())
    train_acc_history.append(training_acc)
    val_acc_history.append(testing_acc)
```

When finding accuracy and loss you should choose correct size for data set for training and testing and also remember they are arrays with 10 labels.

```
    #backward
    dy_pred = 1./batch_size*2.0*(y_pred - y) # partial derivatives w.r.t y_predicted
    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)
```

For initial learning rate 0.0140, training loss = 1.000003, testing loss = 1.000003, training accuracy = 75.025000, testing accuracy = 75.000000 and you can find my GitHub repository for full details.

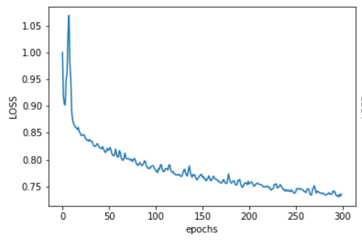


figure 06: training loss

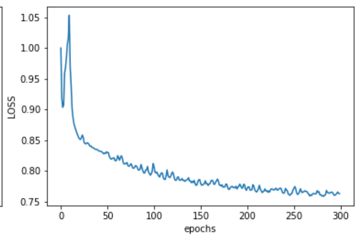


figure 07: testing loss

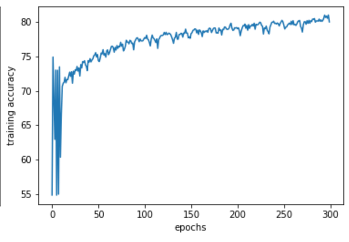


figure 08: training accuracy

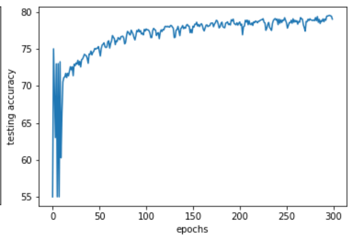


figure 09: testing accuracy

3.0 Stochastic Gradient Descent

Gradient descent algorithms have been modified to construct stochastic gradient descent algorithms. Instead of using all of the observations, stochastic gradient descent measures the gradient using a random small subset of them. This method will minimize computation time in some cases. In this case we used technique called minibatches.

a) The set of observations is randomly divided into minibatches by stochastic gradient descent.
b) The gradient is computed, and the vector is transferred for each minibatch.
c) When all minibatches have been used, the iteration, or epoch, is said to be complete, and the next one begins.

```
for t in range(iterations):
    for start in range(0, Ntr, batch_size):
        batch_indices = np.random.choice(Ntr, batch_size)
        x = x_train[batch_indices]
        y = y_train[batch_indices]
```

Run two for loops. First one for epochs and second one for randomly selected batches. After one epoch finished the losses and accuracies are counted. This may take time but quickly get higher accuracy value shown in my GitHub repository.

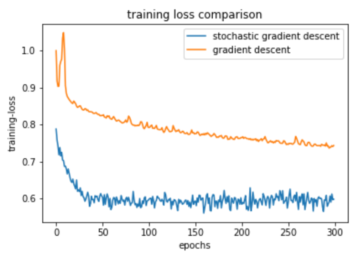


figure 07: training loss

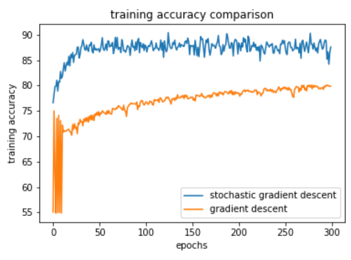


figure 08: training accuracy

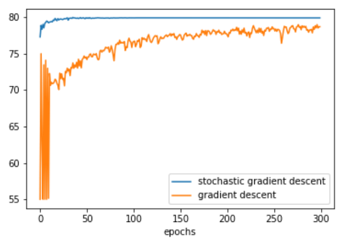


figure 09: testing accuracy

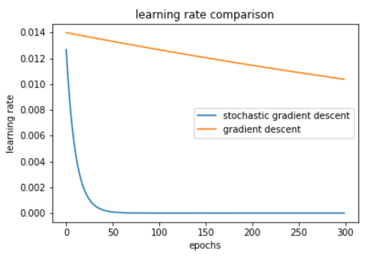


figure 10: learning rate

Compare to gradient descent this has high accuracy and low loss and quickly become high accurate network.

4.0 Construct a CNN

Building a convolutional neural network (CNN) is a great way to use deep learning to identify pictures. The Keras Python library makes building a CNN very easy. Use a 3x3 filter matrix with 2x2 max pooling. That means when convolute it takes max value in the window. After preprocessing the data, the model is build.

```
model = Sequential()
# convolution layer & maxpooling layer with 32 nodes
model.add(Conv2D(32, (3, 3), activation = "relu", input_shape=(32,32,3)))
model.add(MaxPooling2D(pool_size=(2, 2)))
# convolution layer & maxpooling layer with 64 nodes
model.add(Conv2D(64, (3, 3), activation = "relu"))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, (3, 3), activation = "relu"))
model.add(MaxPooling2D(pool_size=(2, 2)))
# Flatten serves as a connection between the convolution and dense layers.
model.add(Flatten())
# output layer
model.add(Dense(64, activation = "relu"))
model.add(Dense(10))
```

As this shows model.add() helps to add layers with activation function of ReLU. After addi ng convolutional and max pooling layers it must be flattened to connect to the Dense(output) layer. 64 & 32 are nodes for corresponding layers. After computing ***model.summary()*** function returns the details. According to results no. of learnable parameters = 73,418.

```
model.compile(optimizer='adam',loss=tf.keras.losses.CategoricalCrossentropy(from_logits=True),metrics=["accuracy"])
history=model.fit(x_train,y_train,epochs=50,batch_size=50,validation_data=(x_test,y_test),)
```

Then compile and fit the model. For optimizer parameter we used 'adam' and other as abov e code. Then train the model for given data. After compiling training loss = 0.1001, testing loss = 2. 1620, training accuracy = 0.9642, testing accuracy = 0.70 and you can find my GitHub repository for full details.

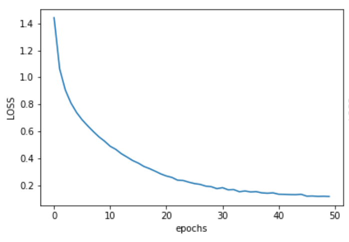


figure 11: training loss

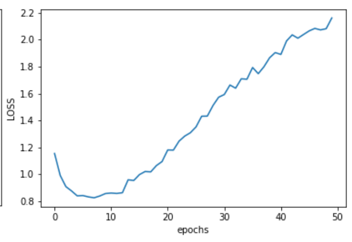


figure 12: testing loss

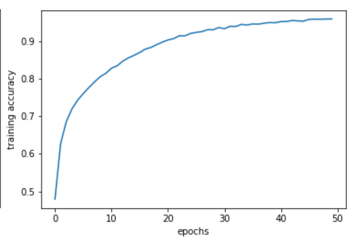


figure 13: training accuracy

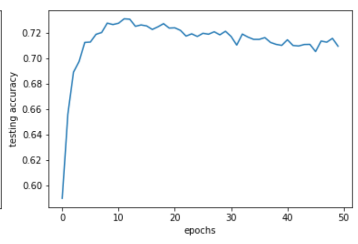


figure 14 testing accuracy