Deep Learning Assignment 1

Implementation of a Deep Neural Network

Khadija Sitabkhan - 20236001

Design of the code:

1. We Have defined all the functions that are commonly used globally so that we dont repeat the same code again
2. 3 Classes are defined:
   - Logistic_Regression: This is the simple logistic regression with no hidden Layers.
   - myNN_WithHidden: This class includes the hidden layer of the neural network. There is 1 hidden layer defined with number of nodes can be changed by changing the self.hidden parameter
   - MyNN_RMSProp : This is the enhancement with RMS Propagation added. In RMSProp we calculate the squared gradient to adjust the weights and bias
3. Load the Cifar Libraries : To load images of 'deer' & 'Frog' in the given data set and classify using the Neural net code designed earlier

Importing Libraries

- **Numpy** : Used for large arrays, dot product calculation, exponential calculation(sigmoid)
- **sklearn** : Used to call the Train test split function to split the data set randomly into training and testing data
- **pandas** : Used to call the read_csv function in order to read data from the moons400.csv and blobs250.csv files
- **random** : Used to call random numbers to select data using in the Stochastic gradient step

In [1]:

```
#Importing libraries

import numpy as np
from sklearn.preprocessing import StandardScaler

from sklearn.model_selection import train_test_split

import pandas as pd
import random
```

Standard Scalar Function:

- The aim is to distribute the data so that its standard deviation equals to **1**
- The formula used is `x[i]=(x[i]-mean(x))/std(x)` where x[i] is the value present in the ith row, mean(x) is the mean of the entire column and std(x) is the standard deviation of the data in the entire column.
- Alot of machine learning algorithms perform best when the data is normally distributed. Here we change the values of the data in the data set but the distribution(range) remains the same, so that the data can converge faster.[1]

In [2]:

```
def Scalar_function(X):
    samples , attribs =  X.shape
    for j in range(attribs):
        for i in range(samples):
            X[i, j] = (X[i, j] - np.mean(X[:][j])) / np.std(X[:][j])
    return X
```

Sigmoid Function

Sigmoid is an activation function that is used in our neural network and logistic regression implementation. The weighted sum and bias when added together gives us a value and the actiavtion function decides whether that value will activate the neuron or not. Hence the name activation function
Sigmoid is widely used in binary classification where we can assign 0 and 1 to either of the classes[4].
There are upto 6 different activation functions:

1. Linear function
2. Sigmoid
3. Tanh
4. ReLU
5. Leaky Relu
6. Softmax

In [3]:

```
def sigmoid_numpy(x):
    return (1/(1+np.exp(-x)))
```

Accuracy Function

We measure the accuracy in terms of how much percentage of out predictions was correct. 2 Numpy arrays are passed to the function

1. The test set output that we already have (Ground Truth)
2. The predicted output

In [4]:

```
def accuracy(y_test,y_predicted):
    score=0
    for i in range(0, len(y_predicted)-1):
        if (y_predicted[i]==y_test[i]):
            score=score+1
    score = score / len(y_test)
    print(f"score is : {round(score*100,2)}%")
```

Cost Function

This cost function is used to determine the cost the algorithm has to pay to determine the predicted value. We have calculated the cost using the formula below. Since we are using stochastic gradient descent the value of N always remains 1 Hence no summation and mean required. We take log as it is strictly monotonically increasing[2]

In [5]:

```
def cost_function(y_true,y_predicted):
    epsilon = 1e-15
    y_predicted_new = max(y_predicted,epsilon)
    y_predicted_new = [min(y_predicted_new,1-epsilon)]
    y_predicted_new = np.array(y_predicted_new)
    return (-(y_true*np.log(y_predicted_new)+(1-y_true)*np.log(1-y_predicted_new)))
```

Logistic Regression

Despite the name contains regression, Logistic Regression is a classification algorithm. We assign weights to the different classes or features in order to determine the probability of the output class.
It is binary logistic regression if we need to determine between two classes.
It is multilinear classification If we have more than 2 classes.

The main formula used is

`z(x) = b + w1.x1 + w2.x2+ ... + wn.xn`

$$z(x) = b + \sum_{i=0}^{N} wi.xi$$

where wi is the weight and xi is the value assigned with the i^th feature

This z(x) is the weighted sum which is then passed to an acitvation function like sigmoid which returns a probability value in the range between 0 and 1
The weights determined while training the data set are then used while testing the data to determine probability

Algorithm of Logistic regression:

```
1. Set the learning rate and number of epochs
2. Initialise weight and a bias
3. Repeat for number of epochs:
    * Take any random sample from the data (stochastic gradient descent)
    * calculate the weighted sum by the formula for z(x) above
    * claculate the sigmoid value from weighted sum.
    * calculate error as the difference between the calculated value (output fro
m sigmoid) and actual value.
    * calculate delta bias as the product of initial matrix and error
    * delta bias is the error
    Sochastic Gradient descent Step:
    * Adjust the weights and bias according to the delta values calculated
 4. Calculate the weighted sum from the test data with the predicted values of w
eights and bias
 5. Sigmoid on the weighted sum to return probability
 6. If probability > 0.5 return 1 else return 0
 7. Calculate accuracy of the model
```

In [6]:

```
class Logistic_Regression:
    def __init__(self):
        self.w1 = 1
#         self.w2 = 1
        self.bias = 1

    def fit(self, X, y,learn_rate, epochs):
        self.w1, self.bias, cost = self.stch_gradient_descent(X,y, learn_rate,epochs)
#         print(f"Final weights and bias: w1: {self.w1}, bias: {self.bias}, cost: {cos
t}")

    def predict(self, X_test):
        nsamples, nattribs = np.shape(X_test)
        weighted_sum=[]
        for i in range(0,nsamples-1):
            x_temp=[]
            x_temp=X_test[i].T
            weighted_sum1 = np.dot(self.w1,x_temp) + self.bias
            weighted_sum1 = sigmoid_numpy(weighted_sum1)
            #print(f"iter: {i} -- w1 : {self.w1} -- x[i]: {X_test[i]} -- weighted sum:
 {self.sigmoid_numpy(weighted_sum1)}")
            if (weighted_sum1> 0.5):
                weighted_sum.append(1)
            else :
                weighted_sum.append(0)

        return weighted_sum

    def stch_gradient_descent(self, X_tem, y_true,rate, epochs):
        (nsamples, nattribs) = np.shape(X_tem)
        w = np.ones(shape=nattribs)
        bias = 1
        rate = 0.01

        for i in range(epochs):

            random_index= random.randint(0,nsamples-1) # gather a random sample from tr
aining data Stochastic Gradient
            x_sample=X_tem[random_index]
            y_sample=y_true[random_index]
            y_predicted= np.dot(w,x_sample)+bias

            y_predicted = sigmoid_numpy(y_predicted) #fun Activation function
            loss = cost_function(y_true, y_predicted)

            w_d = x_sample.T.dot(y_predicted-y_sample)
            bias_d = y_predicted-y_sample
            #Update weights and bias
            w = w - rate * w_d
            bias = bias - rate * bias_d

            cost = np.square(y_sample - y_predicted)



        return w, bias, cost
```

Neural Network Implementation

1. Step 1 : Initialisation
   - Set Learning Rate
   - Set epoch
   - Initialise Weights and bias to random values If **X** is number of features present in our X_train data set & **N** is number of hidden nodes, from input to hidden layer : Then Weight will be an (X x N) matrix Bias_1 will be an (N x 1) matrix From Hidden to output layer: Weights will be an (N x 1) matrix bias_2 will be a (1 x1) matrix (This is because we have only one output node)
2. Step 2: Repeat for maximum iterations:
   - Since we implement Stochastic gradient descent algorithm we take one random training sample in every iteration
   - Forward Propagation Step:
     - Calculate the z_hidden value using the Weight1, input values and bias_1
     - Calculate the sigmoid of z_hidden
     - Calculate the z_output using sigmoid(z_hidden), weight2 and bias_2
   - Back propagation Step:
     - Send the errors back in the reverse order (output node -> hidden -> input) and update the weights and bias to calulate the delta values. Delta is the change that is present between the observed and expected behavior of the network
   - Stochastic Gradient Descent Update Step
     - Using the delta values calculated above, update the weights and bias across the network and repeat the process again.

In [7]:

```
class myNN_WithHidden:
    def __init__(self):
        self.w1 = [] # Weights assigned from input layer to Hidden Layer
        self.w2 = [] #Weights assigned from hidden layer to output layer
        self.bias = []
        self.bias1 = 0
        self.hidden = 4


    def fit(self, X, y,learn_rate, epochs):

        cost = self.stch_gradient_descent(X,y,learn_rate, epochs)

    def predict(self, X_test):
        nsamples, nattribs = np.shape(X_test)
        weighted_sum=[]
        final_output=[]
        for i in range(0,nsamples-1):
            layer1_output=[]
            x_temp=np.dot(X_test[i],self.w1)+self.bias.T

            for j in range(self.hidden):
                layer1_out=sigmoid_numpy(x_temp[:,j]) #Running activation function for
 hidden node
                layer1_output.append(layer1_out)
            layer1_output=np.array(layer1_output)
            output=np.dot(layer1_output.T,self.w2)+self.bias1
            final_output=sigmoid_numpy(output) #activation function for output node
            if (final_output> 0.5): # Hard Threshold
                weighted_sum.append(1)
            else :
                weighted_sum.append(0)

        return weighted_sum

    def delta(self,y):
        return sigmoid_numpy(y)*(1-sigmoid_numpy(y))



    def stch_gradient_descent(self, X_tem, y_true,rate, epochs):
        (nsamples, nattribs) = np.shape(X_tem)
        #Initialising weights and bias
        self.w1 = np.random.randn(nattribs,self.hidden)
        self.w2 = np.random.randn(self.hidden,1)
        self.bias = np.random.randn(self.hidden,1)
        self.bias1=np.random.randn()
#         rate = 0.01
        for i in range(epochs):
            z_hidden = []
            a_hidden = []
            z_output = []
            #a_output = []
            delta_z1= []
            random_index= random.randint(0,nsamples-1)
            x_sample=X_tem[random_index]
            y_sample=y_true[random_index]
            #Forward Propagation step
            for j in range(self.hidden): #this will generate all the
```

```
                #output from the first hidden node.
                #applying logistic regression on the hidden nodes
                self.z1 = np.dot(x_sample,self.w1[:,j])+self.bias[j]
                z_hidden.append(self.z1)
                self.a = sigmoid_numpy(self.z1)
                a_hidden.append(self.a)
            a_hidden=np.array(a_hidden)
            z_output= np.dot(a_hidden.T,self.w2)+self.bias1
            y_predicted = sigmoid_numpy(z_output)
            delta_z_output= y_predicted - y_sample
            delta_bias1 = delta_z_output
            cost = cost_function(y_sample,y_predicted)

            #Backpropagation Calculations
            delta_w2 = np.dot(a_hidden,delta_z_output)
            #updating bias for the hidden node:
#             delta_bias= np.array((3))
            for j in range(self.hidden):
                temp = np.dot(delta_z_output,self.w2[j])
                delta_z1_temp= self.delta(z_hidden[j])*temp
                delta_z1.append(delta_z1_temp)
            delta_z1 = np.array(delta_z1)
            delta_bias = np.array(delta_z1)
#             self.bias = self.delta1z
            x_sample1= x_sample.reshape(nattribs,1)
            delta_w1 = np.dot(x_sample1,delta_z1.T)

            #Stochastic Gradient Descent update step
            self.w2 -= rate*delta_w2
            self.bias1 -= rate*delta_bias1
            self.w1 -= rate*delta_w1
            self.bias -= rate*delta_bias

        return cost
```

Enhancing the above implementation by adding RMS(Root Mean Squared) Propagation

We have used the same algorithm that we created above. we keep a squared gradient of each parameter as its moving average, in the stochastic gradient descent update step, we divide the current gradient by the square root of the average squared gradient. (Epsilon is added to avoid divide by zero erros)

Working of RMSProp:

- Set the value of beta (moving average parameter).
- Calculate the squared gradient using formula:

$$\text{Sq\_grad\_Weight} = (1- \beta)(\text{delta\_weight})^2$$

$$\text{sq\_grad\_bias} = (1- \beta)(\text{delta\_bias})^2$$

- Perform the stochastic gradient descent update step for weights and bias of hidden and output layers:

$$\text{Weights} = \text{weights} - \text{delta\_weights}/(\text{square\_root}(\text{Sq\_grad\_Weight}) + \text{Epsilon})$$

$$\text{Bias} = \text{Bias} - \text{delta\_Bias}/(\text{square\_root}(\text{Sq\_grad\_Bias}) + \text{Epsilon})$$

In [8]:

```
class MyNN_RMSProp:
    def __init__(self):
        self.w1 = [] # Weights assigned from input layer to Hidden Layer
        self.w2 = [] #Weights assigned from hidden layer to output layer
        self.bias = [] # Bias for the hidden layer
        self.bias1 = 0 # bias for the final output
        self.hidden = 3 # Number of hidden layers
        self.beta = 0.9
        self.s_del_w1=0
        self.s_del_w2=0
        self.s_del_b=0
        self.s_del_b1=0
        self.epsilon = 1e-15

    def fit(self, X, y,learn_rate, epochs):
        cost = self.stch_gradient_descent(X,y,learn_rate, epochs)

#         self.w1, self.w2, self.bias,self.bias1, cost = self.stch_gradient_descent(X,
y,learn_rate, epochs)

    def predict(self, X_test):
        nsamples, nattribs = np.shape(X_test)
        weighted_sum=[]
        final_output=[]
        for i in range(0,nsamples-1):
            layer1_output=[]
            x_temp=np.dot(X_test[i],self.w1)+self.bias.T

            for j in range(self.hidden):
                layer1_out=sigmoid_numpy(x_temp[:,j]) #Running activation function for
 hidden node
                layer1_output.append(layer1_out)
            layer1_output=np.array(layer1_output)
            output=np.dot(layer1_output.T,self.w2)+self.bias1
            final_output=sigmoid_numpy(output) #activation function for output node
            if (final_output> 0.5): # Hard Threshold
                weighted_sum.append(1)
            else :
                weighted_sum.append(0)

        return weighted_sum

    def delta(self,y):
        return sigmoid_numpy(y)*(1-sigmoid_numpy(y))

    def rms_prop(self,delta_w1,delta_w2,delta_bias , delta_bias1,rate):
        self.s_del_w1= (1-self.beta)*np.square(delta_w1)+ self.beta*self.s_del_w1
        self.s_del_w2= (1-self.beta)*np.square(delta_w2)+ self.beta*self.s_del_w2
        self.s_del_b= (1-self.beta)*np.square(delta_bias)+ self.beta*self.s_del_b
        self.s_del_b1= (1-self.beta)*np.square(delta_bias1)+ self.beta*self.s_del_b1


    def stch_gradient_descent(self, X_tem, y_true,rate, epochs):
        (nsamples, nattribs) = np.shape(X_tem)
        #Initialising weights and bias
        self.w1 = np.random.randn(nattribs,self.hidden)
        self.w2 = np.random.randn(self.hidden,1)
        self.bias = np.random.randn(self.hidden,1)
```

```
        self.bias1=np.random.randn()

        for i in range(epochs):
            z_hidden = []
            a_hidden = []
            z_output = []
            delta_z1= []
            random_index= random.randint(0,nsamples-1)
            x_sample=X_tem[random_index]
            y_sample=y_true[random_index]
            #Forward Propagation step
            for j in range(self.hidden): #this will generate all the
                #output from the first hidden node.
                #applying logistic regression on the hidden nodes
                self.z1 = np.dot(x_sample,self.w1[:,j])+self.bias[j]
                z_hidden.append(self.z1)
                self.a = sigmoid_numpy(self.z1)
                a_hidden.append(self.a)
            a_hidden=np.array(a_hidden)
            z_output= np.dot(a_hidden.T,self.w2)+self.bias1
            y_predicted = sigmoid_numpy(z_output)
            delta_z_output= y_predicted - y_sample
            delta_bias1 = delta_z_output
            cost = cost_function(y_sample,y_predicted)

            #Backpropagation Calculations
            delta_w2 = np.dot(a_hidden,delta_z_output)
            #updating bias for the hidden node:
#             delta_bias= np.array((3))
            for j in range(self.hidden):
                temp = np.dot(delta_z_output,self.w2[j])
                delta_z1_temp= self.delta(z_hidden[j])*temp
                delta_z1.append(delta_z1_temp)
            delta_z1 = np.array(delta_z1)
            delta_bias = np.array(delta_z1)
#             self.bias = self.delta1z
            x_sample1= x_sample.reshape(nattribs,1)
            delta_w1 = np.dot(x_sample1,delta_z1.T)

            #getting values for squared gradient for root mean square propagation
            #these values are initialised to 0 in the constructor of the class and then
adjusted according to the delta weights
            self.rms_prop(delta_w1,delta_w2,delta_bias , delta_bias1,rate)


            #Stochastic Gradient Descent update step
            self.w2 -= rate*(delta_w2/(np.sqrt(self.s_del_w2)+self.epsilon))
            self.bias1 -= rate*(delta_bias1/(np.sqrt(self.s_del_b1)+self.epsilon))
            self.w1 -= rate*(delta_w1/(np.sqrt(self.s_del_w1)+self.epsilon))
            self.bias -= rate*(delta_bias/(np.sqrt(self.s_del_b)+self.epsilon))
        return cost
```

Calling all functions and classes for Blobs 250 data

In [9]:

```
df = pd.read_csv("blobs250.csv")
y = df['Class'].values

learn_rate=0.01
del df['Class']   # drop the 'Class' column from the dataframe
X = df.values     # convert the remaining columns to a numpy array
X = Scalar_function(X)
X_train, X_test_1, y_train, y_test_1 = train_test_split(X, y, test_size=0.3)
X_val, X_test, y_val, y_test = train_test_split(X_test_1, y_test_1, test_size=0.5)
```

Running Simple Logistic Regression algorithm for Blobs250 data which is linearly seperable

In [10]:

```
obj1 = Logistic_Regression()
obj1.fit(X_train, y_train,learn_rate, epochs=10000)
print("Running Model on training Data set")
obj1.predict(X_train)
y_predicted = obj1.predict(X_train)
accuracy(y_predicted,np.array(y_train))
print("\nRunning Model on Test data set")
y_predicted = obj1.predict(X_test)
accuracy(y_test,np.array(y_predicted))
```

```
Running Model on training Data set
score is : 100.0%

Running Model on Test data set
score is : 94.74%
```

Running Neural Net implemented above for Blobs250 data

In [11]:

```
obj1 = myNN_WithHidden()
obj1.fit(X_train, y_train,learn_rate, epochs=10000)
print("Running Model on training Data set")
obj1.predict(X_train)
y_predicted = obj1.predict(X_train)
accuracy(y_predicted,np.array(y_train))
print("\nRunning Model on Test data set")
obj1.predict(X_test)
y_predicted = obj1.predict(X_test)
accuracy(y_test,np.array(y_predicted))
```

```
Running Model on training Data set
score is : 100.0%

Running Model on Test data set
score is : 94.74%
```

Running ReLU implemented NN algorithm for Blobs250

In [12]:

```
obj1 = MyNN_RMSProp()
obj1.fit(X_train, y_train,learn_rate, epochs=10000)
print("Running Model on training Data set")
obj1.predict(X_train)
y_predicted = obj1.predict(X_train)
accuracy(y_predicted,np.array(y_train))
print("\nRunning Model on Test data set")
obj1.predict(X_test)
y_predicted = obj1.predict(X_test)
accuracy(y_test,np.array(y_predicted))
```

```
Running Model on training Data set
score is : 100.0%

Running Model on Test data set
score is : 94.74%
```

Observations on the linearly Seperable data set

From the above scores we observe that the accuracy has remained constant for linearly sperable dataset. There is a slight overfitting of data thats why we see accuracy of 100% on the training set and 94.74 on the test data

Calling all functions and classes for Moons 400 data which is not linearly seperable

In [13]:

```
df = pd.read_csv("moons400.csv")
y = df['Class'].values

learn_rate=0.01
del df['Class']   # drop the 'Class' column from the dataframe
X = df.values     # convert the remaining columns to a numpy array

X = Scalar_function(X)
X_train, X_test_1, y_train, y_test_1 = train_test_split(X, y, test_size=0.3)
X_val, X_test, y_val, y_test = train_test_split(X_test_1, y_test_1, test_size=0.5)
```

Running Logistic Regression algorithm for moons400 data

In [14]:

```
obj1 = Logistic_Regression()
obj1.fit(X_train, y_train,learn_rate, epochs=10000)
print("Running Model on training Data set")
obj1.predict(X_train)
y_predicted = obj1.predict(X_train)
accuracy(y_predicted,np.array(y_train))
print("\nRunning Model on Test data set")
y_predicted = obj1.predict(X_test)
accuracy(y_test,np.array(y_predicted))
```

```
Running Model on training Data set
score is : 77.78%

Running Model on Test data set
score is : 80.0%
```

Running NN Implemented algorithm for moons400 data

In [15]:

```
obj1 = myNN_WithHidden()
obj1.fit(X_train, y_train,learn_rate, epochs=10000)
print("Running Model on training Data set")
obj1.predict(X_train)
y_predicted = obj1.predict(X_train)
accuracy(y_predicted,np.array(y_train))
print("\nRunning Model on Test data set")
obj1.predict(X_test)
y_predicted = obj1.predict(X_test)
accuracy(y_test,np.array(y_predicted))
```

```
Running Model on training Data set
score is : 76.7%

Running Model on Test data set
score is : 78.33%
```

Running ReLU implemented NN algorithm for moons400

In [16]:

```
obj1 = MyNN_RMSProp()
obj1.fit(X_train, y_train,learn_rate, epochs=50000)
print("Running Model on training Data set")
obj1.predict(X_train)
y_predicted = obj1.predict(X_train)
accuracy(y_predicted,np.array(y_train))
print("\nRunning Model on Test data set")
# obj1.predict(X_test)
y_predicted = obj1.predict(X_test)
accuracy(y_test,np.array(y_predicted))
```

```
Running Model on training Data set
score is : 86.74%

Running Model on Test data set
score is : 85.0%
```

Observations

We know that the moons400 dataset is not linearly seperable.
We also notice that the accuracy score for training data set reduced from 77.78 (logistic) to 76.7 (NN) and then increased to 86.74 (RMSProp).
The accuracy of the test data was infact reduced from 80.0 (logistic) to 78.33 (NN) and then increased to 85.0 (RMSPRop).

Effect of the implementation on linear vs non-linear data

We observe that the accuracy on the non-linear data improves on the implementation of RMS Propagation algorithm. It however does not degrade on the linearly seperable data.

Loading the CIFAR Data set

In [17]:

```
def unpickle(file):
    import pickle
    with open(file, 'rb') as fo:
        dict = pickle.load(fo, encoding='bytes')
    return dict

# Loaded in this way, each of the batch files contains a dictionary with the following
 elements:
#   data -- a 10000x3072 numpy array of uint8s. Each row of the array stores a 32x32 co
lour image.
#           The first 1024 entries contain the red channel values, the next 1024 the gr
een, and the final 1024 the blue.
#           The image is stored in row-major order, so that the first 32 entries of the
array are the red channel values
#           of the first row of the image.
#   labels -- a list of 10000 numbers in the range 0-9.
#             The number at index i indicates the label of the ith image in the array d
ata.

def loadbatch(batchname):
    folder = 'cifar-10-batches-py'
    batch = unpickle(folder+"/"+batchname)
    return batch
def loadlabelnames():
    folder = 'cifar-10-batches-py'
    meta = unpickle(folder+"/"+'batches.meta')
    return meta[b'label_names']
```

In [18]:

```
def LoadCifar(filename):
    batch1 = loadbatch(filename)
#     print("Number of items in the batch is", len(batch1))

    # Display all keys, so we can see the ones we want
#     print('All keys in the batch:', batch1.keys())
    data = batch1[b'data']
    labels = batch1[b'labels']
#     print ("size of data in this batch:", len(data), ", size of labels:", len(label
s))
#     print (type(data))
#     print(data.shape)
    new_img=[]
    new_label_list=[]
    names = loadlabelnames()
    for i in range(len(labels)):
        if(b"frog"== names[labels[i]]):
            new_img.append(data[i][:1024])
            new_label_list.append(0)
        elif(b"deer" == names[labels[i]]):
            new_img.append(data[i][:1024])
            new_label_list.append(1)
    X_test1=np.array(new_img)
    y_test1=np.array(new_label_list)
#     print(np.array(new_img).shape)
#     print(np.array(new_label_list).reshape(len(new_label_list),1).shape)
    return X_test1 , y_test1
```

In [19]:

```
from sklearn.preprocessing import StandardScaler
X1,y1 = LoadCifar("data_batch_3")
scaler = StandardScaler()
# X1 = Scalar_function(X1)
X1 = scaler.fit_transform(X1)
X_train, X_test_1, y_train, y_test_1 = train_test_split(X1, y1, test_size=0.3)

# X_val, X_test, y_val, y_test = train_test_split(X_test_1, y_test_1, test_size=0.5)
learn_rate =0.01
```

In [20]:

```
obj = Logistic_Regression()
obj.fit(X_train, y_train,learn_rate, epochs=10000)
print("Running Model on training Data set")
obj.predict(X_train)
y_predicted = obj.predict(X_train)
accuracy(y_predicted,np.array(y_train))
print("\nRunning Model on Test data set")
# obj.predict(X_test)
y_predicted = obj.predict(X_test_1)
accuracy(y_test_1,np.array(y_predicted))
```

```
<ipython-input-3-261c23558eb7>:2: RuntimeWarning: overflow encountered in
exp
  return (1/(1+np.exp(-x)))

Running Model on training Data set
score is : 72.17%

Running Model on Test data set
score is : 56.01%
```

In [21]:

```
obj1 = myNN_WithHidden()
obj1.fit(X_train, y_train,learn_rate, epochs=10000)
print("Running Model on training Data set")
obj1.predict(X_train)
y_predicted = obj1.predict(X_train)
accuracy(y_predicted,np.array(y_train))
print("\nRunning Model on Test data set")
# obj1.predict(X_test)
y_predicted = obj1.predict(X_test_1)
accuracy(y_test_1,np.array(y_predicted))
```

```
Running Model on training Data set
score is : 62.14%

Running Model on Test data set
score is : 54.99%
```

In [22]:

```
obj2 = MyNN_RMSProp()
obj2.fit(X_train, y_train,learn_rate, epochs=10000)
print("Running Model on training Data set")
obj2.predict(X_train)
y_predicted = obj2.predict(X_train)
accuracy(y_predicted,np.array(y_train))
print("\nRunning Model on Test data set")
# obj2.predict(X_test)
y_predicted = obj2.predict(X_test_1)
accuracy(y_test_1,np.array(y_predicted))
```

```
Running Model on training Data set
score is : 67.37%

Running Model on Test data set
score is : 59.56%
```

In [23]:

```
X_test1,y_test1 = LoadCifar("test_batch")
X1 = scaler.fit_transform(X_test1)
learn_rate =0.01
```

In [24]:

```
obj = Logistic_Regression()
obj.fit(X_train, y_train,learn_rate, epochs=10000)
print("Running Model on training Data set")
obj.predict(X_train)
y_predicted = obj.predict(X_train)
accuracy(y_predicted,np.array(y_train))
print("\nRunning Model on Test data set")
# obj.predict(X_test)
y_predicted = obj.predict(X1)
accuracy(y_test1,np.array(y_predicted))
```

```
<ipython-input-3-261c23558eb7>:2: RuntimeWarning: overflow encountered in
exp
  return (1/(1+np.exp(-x)))

Running Model on training Data set
score is : 72.89%

Running Model on Test data set
score is : 58.15%
```

In [25]:

```
obj1 = myNN_WithHidden()
obj1.fit(X_train, y_train,learn_rate, epochs=10000)
print("Running Model on training Data set")
obj1.predict(X_train)
y_predicted = obj1.predict(X_train)
accuracy(y_predicted,np.array(y_train))
print("\nRunning Model on Test data set")
# obj1.predict(X_test)
y_predicted = obj1.predict(X1)
accuracy(y_test1,np.array(y_predicted))
```

```
Running Model on training Data set
score is : 57.05%

Running Model on Test data set
score is : 53.0%
```

In [26]:

```
obj2 = MyNN_RMSProp()
obj2.fit(X_train, y_train,learn_rate, epochs=10000)
print("Running Model on training Data set")
obj2.predict(X_train)
y_predicted = obj2.predict(X_train)
accuracy(y_predicted,np.array(y_train))
print("\nRunning Model on Test data set")
# obj2.predict(X_test)
y_predicted = obj2.predict(X1)
accuracy(y_test1,np.array(y_predicted))
```

```
Running Model on training Data set
score is : 65.12%

Running Model on Test data set
score is : 60.2%
```

Observations

We see that with CIFAR data set we are getting maximum score using Logistic regression for the training data set. However we see that overfitting has reduced that the difference between the accuracy score of the training dat (data_batch_3) and testng (test_batch) has reduced drastically. Best Performance is seen with the RMS Propagation implementation

References:

1.Idea and implementation of standard Scalar : StandardScalar (https://towardsdatascience.com/scale-standardize-or-normalize-with-scikit-learn-6ccc7d176a02#:~:text=turn%20to%20StandardScaler.-,StandardScaler,values%20by%20the%20standard%20
2. Week 2 Videos and notes : Week 2 Pdf (https://learn-eu-central-1-prod-fleet01-xythos.content.blackboardcdn.com/5cf4ce7a4424f/6653344?X-Blackboard-Expiration=1615852800000&X-Blackboard-Signature=LBv4yTw7XFYEKaYFovrhUNXvGIuNB26v8xWIL8aifck%3D&X-Blackboard-Client-Id=130002&response-cache-control=private%2C%20max-age%3D21600&response-content-disposition=inline%3B%20filename%2A%3DUTF-8%27%27DeepLearning-02-FundamentalsOfNNs-Pt1.pdf&response-content-type=application%2Fpdf&X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Date=20210315T180000Z&X-Amz-SignedHeaders=host&X-Amz-Expires=21600&X-Amz-Credential=AKIAZH6WM4PL5M5HI5WH%2F20210315%2Feu-central-1%2Fs3%2Faws4_request&X-Amz-Signature=9528656addd3c01f68cb371a319697b5617f942a2ff72c112522e431c3ba20be).
3. Week 3 videos and notes for Neural net & ReLU implementation : Week 3 Pdf (https://learn-eu-central-1-prod-fleet01-xythos.content.blackboardcdn.com/5cf4ce7a4424f/6739172?X-Blackboard-Expiration=1615852800000&X-Blackboard-Signature=l%2B22bmKmymDYdmpUXBm9Oazj%2Bym2MK3ZXr60kMaXO28%3D&X-Blackboard-Client-Id=130002&response-cache-control=private%2C%20max-age%3D21600&response-content-disposition=inline%3B%20filename%2A%3DUTF-8%27%27DeepLearning-03-FundamentalsOfNNs-Pt2.pdf&response-content-type=application%2Fpdf&X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Date=20210315T180000Z&X-Amz-SignedHeaders=host&X-Amz-Expires=21600&X-Amz-Credential=AKIAZH6WM4PL5M5HI5WH%2F20210315%2Feu-central-1%2Fs3%2Faws4_request&X-Amz-Signature=e8f3368bd6514b2545ef895135eb4c28ee38b843e2a60cfb86229dcfdf1210c1).
4. sigmoid and other activation function definitions: Activation Functions (https://www.geeksforgeeks.org/activation-functions-neural-networks/)

In [ ]: