###TEAM\_16 : ASSIGNMENT 1 - CS6910 - FOUNDATIONS OF DL###

$$$PART (1) - //Save input file in local directory with the code and run-prereqs.

# Loading necessary libraries

import numpy as np

import csv

import matplotlib.pyplot as plt

%matplotlib inline

from mpl\_toolkits.mplot3d import Axes3D

import matplotlib.tri as mtri

# Parameter declarations

TRAIN\_SIZE = 100

VALID\_SIZE = 300

IN\_SIZE = 2

OUT\_SIZE = 1

lr = 0.05

momentum = 0.8

num\_hidden = 2

sizes = [10,8]

activation = 'sigmoid'

batch\_size = 1

# Loading training and validation data

train\_data = []

val\_data = []

with open('Team16/train100.txt') as f:

file\_reader = csv.reader(f, delimiter='\t')

for row in file\_reader:

train\_data.append(row[0].split())

with open('Team16/val.txt') as f:

file\_reader = csv.reader(f, delimiter='\t')

for row in file\_reader:

val\_data.append(row[0].split())

# Normalize the data

x1=[]

x2=[]

y=[]

for i in train\_data:

x1.append(float(i[0]))

x2.append(float(i[1]))

y.append(float(i[2]))

x1\_val=[]

x2\_val=[]

y\_val=[]

for i in val\_data:

x1\_val.append(float(i[0]))

x2\_val.append(float(i[1]))

y\_val.append(float(i[2]))

x1\_mean = np.mean(x1)

x1\_var = np.var(x1)

x2\_mean = np.mean(x2)

x2\_var = np.var(x2)

y\_mean = np.mean(y)

y\_var = np.var(y)

x1\_range = np.max(x1) - np.min(x1)

x2\_range = np.max(x2) - np.min(x2)

y\_range = np.max(y) - np.min(y)

x1\_st = np.std(x1)

x2\_st = np.std(x2)

y\_st = np.std(y)

X\_train = []

Y\_train = []

actual\_Y\_train = []

for i in train\_data:

X\_train.append(((float(i[0])-x1\_mean)/x1\_var,(float(i[1])-x2\_mean)/x2\_var))

Y\_train.append((float(i[2])-y\_mean)/y\_var)

actual\_Y\_train.append(float(i[2]))

X\_val = []

Y\_val = []

for i in val\_data:

X\_val.append(((float(i[0])-x1\_mean)/x1\_var,(float(i[1])-x2\_mean)/x2\_var))

Y\_val.append(float(i[2]))

size\_list = [IN\_SIZE] + sizes + [OUT\_SIZE]

np.random.seed(1234)

# Useful function definitions

def sigmoid(m):

return (1/(1+np.exp(-m))) # This is a vectorized function

def tanh(m):

return (np.exp(m)-np.exp(-m))/(np.exp(m)+np.exp(-m))

def softmax(m):

return (np.exp(m-np.max(m))/np.sum(np.exp(m-np.max(m))))

def sigmoid\_derivative(m):

return sigmoid(m)\*(1-sigmoid(m))

def tanh\_derivative(m):

return 1-tanh(m)\*tanh(m)

def forward\_prop(weights\_list, bias\_list, inputs, exp\_outputs):

""" Forward propagation function """

pre\_act\_list =[0]\*(num\_hidden+2)

post\_act\_list = [0]\*(num\_hidden+2)

post\_act\_list[0] = inputs

if activation=='sigmoid':

act = sigmoid

if activation=='tanh':

act=tanh

for k in range(1, num\_hidden+1):

pre\_act\_list[k] = np.dot(weights\_list[k],post\_act\_list[k-1])+bias\_list[k]

post\_act\_list[k] = act(pre\_act\_list[k])

pre\_act\_list[num\_hidden+1] = np.dot(weights\_list[num\_hidden+1],post\_act\_list[num\_hidden])+bias\_list[num\_hidden+1]

outputs = pre\_act\_list[num\_hidden+1]

loss = 0.5\*(exp\_outputs-outputs[0])\*\*2

return pre\_act\_list, post\_act\_list, outputs, loss

def forward\_prop\_for\_predict(weights\_list, bias\_list, inputs, exp\_outputs):

"""Forward prop with denormalized output """

pre\_act\_list =[0]\*(num\_hidden+2)

post\_act\_list = [0]\*(num\_hidden+2)

post\_act\_list[0] = inputs

if activation=='sigmoid':

act = sigmoid

if activation=='tanh':

act=tanh

for k in range(1, num\_hidden+1):

pre\_act\_list[k] = np.dot(weights\_list[k],post\_act\_list[k-1])+bias\_list[k]

post\_act\_list[k] = act(pre\_act\_list[k])

pre\_act\_list[num\_hidden+1] = np.dot(weights\_list[num\_hidden+1],post\_act\_list[num\_hidden])+bias\_list[num\_hidden+1]

outputs = pre\_act\_list[num\_hidden+1]

outputs = outputs[0]\*y\_var+y\_mean

loss = 0.5\*(exp\_outputs-outputs)\*\*2

return pre\_act\_list, post\_act\_list, outputs, loss

def backward\_prop(weights\_list, pre\_act\_list, post\_act\_list, outputs, exp\_output\_val):

""" Function for backward propagation """

if(activation=='sigmoid'):

fn = sigmoid\_derivative

elif activation=='tanh':

fn = tanh\_derivative

else:

print("Activation function is invalid!")

grad\_ak\_list = [0]\*(num\_hidden+2) #1st element of these lists are dummy values

grad\_hk\_list = [0]\*(num\_hidden+2)

grad\_wk\_list = [0]\*(num\_hidden+2)

grad\_bk\_list = [0]\*(num\_hidden+2)

grad\_ak\_list[num\_hidden+1] = -(exp\_output\_val - outputs)

for k in range(num\_hidden+1, 0, -1):

# Compute gradients w.r.t parameters

grad\_wk\_list[k] = np.dot(np.expand\_dims(grad\_ak\_list[k],axis=1), np.transpose(np.expand\_dims(post\_act\_list[k-1],axis=1)))

grad\_bk\_list[k] = grad\_ak\_list[k].copy()

# Compute gradients w.r.t layer below

grad\_hk\_list[k-1] = np.dot(np.transpose(weights\_list[k]), grad\_ak\_list[k])

# Compute gradients w.r.t layer below (pre-activation)

grad\_ak\_list[k-1] = grad\_hk\_list[k-1]\*fn(pre\_act\_list[k-1])

return grad\_wk\_list, grad\_bk\_list

def predict(weights\_list, bias\_list, inputs, exp\_outputs, verbatim=False):

""" Function for finding loss during evaluation """

val\_loss = 0

output\_list = []

for i in range(len(inputs)):

pre\_act\_list, post\_act\_list, outputs, loss = forward\_prop\_for\_predict(weights\_list, bias\_list, inputs[i], exp\_outputs[i])

val\_loss+=loss

output\_list.append(outputs)

val\_loss = val\_loss/len(inputs)

return val\_loss, output\_list

def momentum\_gradient\_descent(batch\_size, lr, momentum, train\_inputs, train\_label, validation\_inputs, validation\_label,actual\_Y\_train):

"Function for performing momentum-based Gradient descent"

val\_loss = 100000000 #Dummy

train\_loss = 100000000 #Dummy

train\_loss\_list=[]

val\_loss\_list = []

weights\_list = []

bias\_list = []

prev\_grad\_weights\_list = []

prev\_grad\_bias\_list = []

# Initialize weights and biases

for i in range(0, num\_hidden+2):

weights\_list.append(np.random.randn(size\_list[i], size\_list[i-1])) #weights & biases at index 0 are just dummy values

bias\_list.append(np.random.randn(size\_list[i]))

prev\_grad\_weights\_list.append(np.zeros((size\_list[i], size\_list[i-1])))

prev\_grad\_bias\_list.append(np.zeros(size\_list[i]))

epoch = 0

while(True):

num\_points\_seen = 0

cum\_loss = 0

steps = 0

for sample in range(TRAIN\_SIZE):

# Forward prop

pre\_act\_list, post\_act\_list, outputs, loss = forward\_prop(weights\_list, bias\_list, train\_inputs[sample], train\_label[sample])

# Backward prop

grad\_wk\_list, grad\_bk\_list = backward\_prop(weights\_list, pre\_act\_list, post\_act\_list, outputs, train\_label[sample])

cum\_loss += loss

if(num\_points\_seen==0):

grad\_wk\_to\_update = grad\_wk\_list.copy()

grad\_bk\_to\_update = grad\_bk\_list.copy()

else:

grad\_wk\_to\_update = [sum(x) for x in zip(grad\_wk\_to\_update, grad\_wk\_list)]

grad\_bk\_to\_update = [sum(x) for x in zip(grad\_bk\_to\_update, grad\_bk\_list)]

num\_points\_seen+=1

if (num\_points\_seen % 1 == 0): # Pattern mode - batch size = 1

curr\_weights\_update\_list = (num\_hidden+2)\*[0]

curr\_bias\_update\_list = (num\_hidden+2)\*[0]

for i in range(len(weights\_list)):

curr\_weights\_update\_list[i] = momentum\*(prev\_grad\_weights\_list[i])+lr\*(grad\_wk\_to\_update[i]/batch\_size)

curr\_bias\_update\_list[i] = momentum\*(prev\_grad\_bias\_list[i])+lr\*(grad\_bk\_to\_update[i]/batch\_size)

weights\_list[i] = weights\_list[i] - curr\_weights\_update\_list[i]

bias\_list[i] = bias\_list[i] - curr\_bias\_update\_list[i]

prev\_grad\_weights\_list = curr\_weights\_update\_list.copy()

prev\_grad\_bias\_list = curr\_bias\_update\_list.copy()

num\_points\_seen = 0

steps += 1

val\_loss\_old = val\_loss

train\_loss\_old = train\_loss

# Evaluate performance after each epoch

train\_loss,\_ = predict(weights\_list, bias\_list, train\_inputs, actual\_Y\_train)

val\_loss,\_ = predict(weights\_list, bias\_list, validation\_inputs, validation\_label)

train\_loss\_list.append(train\_loss)

val\_loss\_list.append(val\_loss)

print("Epoch {}, Train loss {}, Validation loss {}".format(epoch+1, train\_loss, val\_loss))

epoch = epoch+1

# Stopping condition

if train\_loss\_old-train\_loss<0.0006:

break

return weights\_list, bias\_list, train\_loss\_list, val\_loss\_list

# Momentum based gradient descent

weights\_list\_updated, bias\_list\_updated, train\_loss\_list, val\_loss\_list = momentum\_gradient\_descent(batch\_size, lr, momentum,

X\_train, Y\_train, X\_val, Y\_val, actual\_Y\_train)

# Save weights

np.save('wts\_sigmoid', weights\_list\_updated)

np.save('bias\_sigmoid', bias\_list\_updated)

# Average error vs epoch

plt.figure(figsize = (5,5))

plt.plot(np.array(train\_loss\_list[40:]))

plt.xlabel("Epoch Number")

plt.ylabel("Training Loss")

plt.grid(True)

plt.title("Avg Error on Training data vs Epoch")

plt.show()

# Loss vs epoch plot

plt.figure(figsize = (5,5))

plt.plot(np.array(train\_loss\_list[40:]),label='Training error')

plt.plot(np.array(val\_loss\_list[40:]), label = 'Validation error')

plt.xlabel("Epoch Number")

plt.ylabel("Training Loss")

plt.grid(True)

plt.title("Training and validation error vs Epoch")

plt.legend()

plt.show()

train\_loss, output\_list\_train = predict(weights\_list\_updated, bias\_list\_updated, X\_train, actual\_Y\_train, False)

val\_loss, output\_list\_val = predict(weights\_list\_updated, bias\_list\_updated, X\_val, Y\_val, False)

# Scatter plot of desired vs model output

plt.scatter(output\_list\_train, actual\_Y\_train,color='red')

plt.grid(True)

x=np.linspace(-20,140,100)

y=x

plt.plot(x,y,color='black')

plt.title('Model output vs Desired output')

plt.xlabel('Model output')

plt.ylabel('Desired output')

x = np.linspace(np.min(x1),np.max(x1), 100)

y = np.linspace(np.min(x1),np.max(x2), 100)

X, Y = np.meshgrid(x, y)

X\_norm = (X-x1\_mean)/x1\_var

Y\_norm = (Y-x2\_mean)/x2\_var

# Surface plot of approximated function

XY\_list = []

for i in range(100):

for j in range(100):

XY\_list.append((X\_norm[i][j],Y\_norm[i][j]))

output\_list\_for\_surface = []

for i in range(len(XY\_list)):

pre\_act\_list, post\_act\_list, outputs, loss = forward\_prop\_for\_predict(weights\_list\_updated, bias\_list\_updated, XY\_list[i], 0)

output\_list\_for\_surface.append(outputs)

Z = np.reshape(np.array(output\_list\_for\_surface), (100,100))

fig = plt.figure(figsize = (10,10))

ax = plt.axes(projection="3d")

#ax.plot\_wireframe(X, Y, Z, color='green')

ax.plot\_surface(X, Y, Z, rstride=1, cstride=1,

cmap='jet', edgecolor='none')

ax.set\_title('Approximated function')

ax.set\_xlabel('x1 co-ordinate')

ax.set\_ylabel('x2 co-ordinate')

ax.set\_zlabel('Predicted value')

plt.show()

# Surface plot of desired function

x = x1+x1\_val #Not normalized

y = x2+x2\_val #Not normalized

z=actual\_Y\_train+y\_val

triang = mtri.Triangulation(x, y)

plt.figure(figsize=(10,10))

#plt.triplot(triang, color='black',alpha=0.3)

ax=plt.axes(projection="3d")

ax.plot\_trisurf(triang,z,cmap='jet')

ax.set\_xlabel('x1 co-ordinate')

ax.set\_ylabel('x2 co-ordinate')

ax.set\_zlabel('Desired value')

ax.set\_title('Desired function (Using the given data points only)')

# # ############## For testing ##############

# # Load weights and biases

# wt = np.load('wts\_sigmoid.npy',allow\_pickle=True)

# bi = np.load('bias\_sigmoid.npy',allow\_pickle=True)

# test\_data = []

# with open('Team16/test.txt') as f:

# file\_reader = csv.reader(f, delimiter='\t')

# for row in file\_reader:

# test\_data.append(row[0].split())

# x1\_test=[]

# x2\_test=[]

# y\_test=[]

# for i in test\_data:

# x1\_test.append(float(i[0]))

# x2\_test.append(float(i[1]))

# y\_test.append(float(i[2]))

# X\_test = []

# Y\_test = []

# for i in test\_data:

# X\_test.append(((float(i[0])-x1\_mean)/x1\_var,(float(i[1])-x2\_mean)/x2\_var))

# Y\_test.append(float(i[2]))

# test\_loss, output\_list\_test = predict(weights\_list\_updated, bias\_list\_updated, X\_test, Y\_test, False)

# print(test\_loss)

# plt.scatter(output\_list\_test, Y\_test,color='red')

# plt.grid(True)

# x=np.linspace(-20,140,100)

# y=x

# plt.plot(x,y,color='black')

# plt.title('Model output vs Desired output')

# plt.xlabel('Model output')

# plt.ylabel('Desired output')

# x\_test = np.linspace(np.min(x1\_test),np.max(x1\_test), 100)

# y\_test = np.linspace(np.min(x1\_test),np.max(x2\_test), 100)

# X\_test\_mesh, Y\_test\_mesh = np.meshgrid(x\_test, y\_test)

# X\_norm\_test = (X\_test\_mesh-x1\_mean)/x1\_var

# Y\_norm\_test = (Y\_test\_mesh-x2\_mean)/x2\_var

# XY\_list\_test = []

# for i in range(100):

# for j in range(100):

# XY\_list\_test.append((X\_norm\_test[i][j],Y\_norm\_test[i][j]))

# output\_list\_for\_surface = []

# for i in range(len(XY\_list\_test)):

# pre\_act\_list, post\_act\_list, outputs, loss = forward\_prop\_for\_predict(weights\_list\_updated, bias\_list\_updated, XY\_list\_test[i], 0)

# output\_list\_for\_surface.append(outputs)

# Z\_test = np.reshape(np.array(output\_list\_for\_surface), (100,100))

# fig = plt.figure(figsize = (10,10))

# ax = plt.axes(projection="3d")

# ax.plot\_surface(X\_test\_mesh, Y\_test\_mesh, Z\_test, rstride=1, cstride=1,

# cmap='jet', edgecolor='none')

# ax.set\_title('Approximated function')

# ax.set\_xlabel('x1 co-ordinate')

# ax.set\_ylabel('x2 co-ordinate')

# ax.set\_zlabel('Predicted value')

# plt.show()

# x\_test = x1\_test #Not normalized

# y\_test = x2\_test #Not normalized

# z\_test = Y\_test

# triang = mtri.Triangulation(x\_test, y\_test)

# plt.figure(figsize=(10,10))

# ax=plt.axes(projection="3d")

# ax.plot\_trisurf(triang,z\_test,cmap='jet')

# ax.set\_xlabel('x1 co-ordinate')

# ax.set\_ylabel('x2 co-ordinate')

# ax.set\_zlabel('Desired value')

# ax.set\_title('Desired function (Using the given data points only)')

$$$PART (2): //Run with input file in local directory along with the code-prereqs.

import numpy as np

import pandas as pd

import itertools

import time

import matplotlib.pyplot as plt

import pickle

### Definition of all the activation functions ###

### implementation of tan hyperbolic activation function ###

def tan\_hyp(y) :

return np.tanh(y)

### implementation of the softmax activation function ###

def softmax(y) :

return np.exp(y-max(y))/np.sum(np.exp(y-max(y)))

### Derivatives of the activation functions ###

### implmentation of the softmax function derivative ###

def der\_softmax(y) :

### derivative is a matrix which can be written as derivative = diag(y) - y.yT ###

a = np.zeros((len(y),len(y)))

np.fill\_diagonal(a,y)

y = np.reshape(y,(len(y),-1))

b = np.dot(y,np.transpose(y))

return a - b

### implementation of tan hyperbolic derivative ###

def der\_tan\_hyp(y) :

return (1-y)\*(1+y)

### Definition of loss functions for a single example ###

### cross entropy loss function L = -t\_iln(yi) ###

def cross\_entropy(y,t):

return -np.sum(t\*np.log(y))

### gradient of the cross entropy function wrt the output layer nodes ###

def grad\_cross\_entropy(y,t) :

return -np.divide(t,y)

### to check the accuracy with the test data

def test\_custom\_weights(activation\_functions,test\_sample,test\_output,w) :

a\_list = [training\_sample]

h\_list = [training\_sample]

output = training\_sample

for (activation\_function,weight) in zip(activation\_functions,w) :

output = np.append(output,np.ones((1,1)),axis = 0)

output = np.dot(weight,output)

a\_list.append(output)

output = eval(activation\_function)(output)

h\_list.append(output)

if np.argmax(test\_sample) == np.argmax(h\_list[-1]):

accuracy = 1

return accuracy

###class definition

class neural\_network :

def \_\_init\_\_(self, n\_hidden,list\_nodes,loss\_fn,list\_activation\_functions,learning\_rate,momentum) :

self.n\_hidden = n\_hidden

self.list\_nodes = list\_nodes

self.loss\_fn = loss\_fn

self.list\_activation\_functions = list\_activation\_functions

self.learning\_rate = learning\_rate

self.momentum = momentum

self.weights = []

self.grad\_past = []

for i in range(n\_hidden+1) :

np.random.seed(42) ### setting the seed so that the weights generated remain constant ###

weight\_layer = np.random.randn(list\_nodes[i+1],list\_nodes[i]+1) ### initialising the weights. The bias parameters are incorporated into this ###

grad\_past\_layer = np.zeros((list\_nodes[i+1],list\_nodes[i]+1)) ### setting the past gradinent to zero ###

self.weights.append(weight\_layer)

self.grad\_past.append(grad\_past\_layer)

def forward\_pass(self,training\_sample): ### function to allow the calculation of the forward weights ###

a\_list = [training\_sample]

h\_list = [training\_sample]

output = training\_sample

for (activation\_function,weight) in zip(self.list\_activation\_functions,self.weights) : ### to loop over all the hidden layers to output ###

output = np.append(output,np.ones((1,1)),axis = 0) ### np.ones() to account for the bias parameter ###

output = np.dot(weight,output)

a\_list.append(output)

### eval() enables us to pass the activation function names as strings. Thereby can change the model easily ###

output = eval(activation\_function)(output)

h\_list.append(output)

return a\_list , h\_list

def calc\_loss(self,calc\_output,actual\_output) : ### function to call the loss function ###

return eval(self.loss\_fn)(calc\_output,actual\_output)

def backward\_pass(self,h\_list,intermediate\_loss,actual\_output) : ### function to allow the calculation of the gradients for back propagation ###

'''implementation of backpropagation'''

### calculating the derivative of the output pre activation wrt the output activation nodes ###

der\_output\_act = eval('der\_'+self.list\_activation\_functions[-1])(h\_list[-1])

### gradient wrt to the loss function ###

grad\_lf = eval('grad\_'+self.loss\_fn)(h\_list[-1],actual\_output)

grad\_layer = np.dot(der\_output\_act,grad\_lf)

for i in range(len(self.weights)) : ### looping over all the layers backwards ###

### calculating the gradients wrt to the weights of that layer i.e. gradients = input node \* gradient wrt pre activation connected to weight ###

grad\_weight\_layer = np.dot(grad\_layer,np.transpose(np.append(h\_list[len(h\_list)-i-2],np.ones((1,1)),axis = 0)))

### gradient for the next layer activation ###

grad\_prev\_output = np.dot(np.transpose(self.weights[len(self.weights)-i-1])[:-1,:],grad\_layer)

### gradients of the next layer pre activation ###

grad\_layer = grad\_prev\_output \* eval('der\_'+self.list\_activation\_functions[len(self.list\_activation\_functions)-i-2])(h\_list[len(h\_list)-i-2])

### weight update rule ###

self.weights[len(self.weights)-i-1] = self.gen\_delta\_rule(self.weights[len(self.weights)-i-1],grad\_weight\_layer,self.grad\_past[len(self.grad\_past)-i-1])

### storing the values of the weight changes which are used in weight update ###

self.grad\_past[len(self.grad\_past)-i-1] = self.learning\_rate\*grad\_weight\_layer

def gen\_delta\_rule(self,weights\_present,grad\_present,weight\_change\_previous) :

### generalised delta rule equation ###

return weights\_present - (grad\_present\*self.learning\_rate) - (weight\_change\_previous\*self.momentum)

def train(self,input\_array,output\_array,valid\_input\_array,valid\_output\_array) :

loss\_list = []

valid\_loss\_list = []

epoch\_num = 0

prev\_loss\_avg = 0

loss\_avg = 0

while (epoch\_num<=0 or abs(prev\_loss\_avg - loss\_avg)>=1e-5) : ### stop parameter set as the difference between the previous and present error ###

prev\_loss\_avg = loss\_avg

loss\_avg = 0

print('Epoch Number : ',epoch\_num)

print('Started')

print('----------------------------------------')

### looping over all the training examples in a given epoch ###

for (training\_sample,actual\_output) in zip(input\_array,output\_array) :

### Running the forward pass ###

a\_list,h\_list = self.forward\_pass(training\_sample)

### calculating the loss ###

intermediate\_loss = self.calc\_loss(h\_list[-1],actual\_output)

### back propagating the loss and weight update ###

self.backward\_pass(h\_list,intermediate\_loss,actual\_output)

### estimating the average loss ###

loss\_avg = loss\_avg + intermediate\_loss

loss\_avg = loss\_avg/len(input\_array)

loss\_list.append(loss\_avg)

valid\_loss\_avg = 0

### calulating the validation error for all the epochs ###

for (training\_sample,actual\_output) in zip(valid\_input\_array,valid\_output\_array) :

a\_list,h\_list = self.forward\_pass(training\_sample)

intermediate\_loss = self.calc\_loss(h\_list[-1],actual\_output)

valid\_loss\_avg = valid\_loss\_avg + intermediate\_loss

valid\_loss\_avg = valid\_loss\_avg/len(valid\_input\_array)

valid\_loss\_list.append(valid\_loss\_avg)

print('Average Loss : ',loss\_avg)

print('----------------------------------------')

epoch\_num = epoch\_num + 1

return loss\_list,valid\_loss\_list

def test(self,input\_array,output\_array) : ### function to return the accuracy of the validation set after training is over ###

accuracy = 0

for (sample,output) in zip(input\_array,output\_array) :

a\_list,h\_list = self.forward\_pass(sample)

calc\_output = h\_list[-1]

### checking if the calculated and the actual match ###

if np.where(calc\_output == np.amax(calc\_output)) == np.where(output == np.amax(output)) :

accuracy = accuracy + 1

accuracy = accuracy / len(input\_array)

return accuracy

##### reading the data for the code #####

data = None

input\_array = []

output\_array = []

test\_input\_array = []

train\_input\_array= []

### reading the input data ###

data = pd.read\_csv('traingroup16.csv')

### loop to convert it into required input and output form i.e. perform one hot encoding ###

for i in range(len(data)) :

input\_array.append(np.array([data['x1'][i],data['x2'][i]]).reshape(2,1))

if data['label'][i] == 0 :

output\_array.append(np.array([1.0,0.0,0.0]).reshape(3,1))

if data['label'][i] == 1 :

output\_array.append(np.array([0.0,1.0,0.0]).reshape(3,1))

if data['label'][i] == 2 :

output\_array.append(np.array([0.0,0.0,1.0]).reshape(3,1))

### splitting the data into test and validation ###

test\_input\_array = input\_array[int(np.floor(len(data)\*3/4)):]

test\_output\_array = output\_array[int(np.floor(len(data)\*3/4)):]

train\_input\_array = input\_array[:int(np.floor(len(data)\*3/4))]

train\_output\_array = output\_array[:int(np.floor(len(data)\*3/4))]

### instantiating the neural network class ###

nn = neural\_network(2,[2,5,5,3],'cross\_entropy',['tan\_hyp','tan\_hyp','softmax'],0.001,0.8)

### training the network ###

list\_avg\_error,list\_valid\_avg\_error = nn.train(train\_input\_array,train\_output\_array,test\_input\_array,test\_output\_array)

### accuracy on the validation set ###

accuracy = nn.test(test\_input\_array,test\_output\_array)

### storing the weights in a file ###

pickle.dump(nn.weights, open('non\_linear\_weights.sav','wb'));

print("Training Finished")

print("---------------------------------------------")

print("Accuracy : ",accuracy)

$$$ PART (3) //Run with ‘generate\_features\_from\_image.py’, ‘load\_extracted\_features.py’, ‘Team\_16’ (input data file) in local directory with the code-prereqs

import numpy as np

import pandas as pd

import itertools

import time

import matplotlib.pyplot as plt

import sys

import pickle

import load\_extracted\_features as lf #Shuffling is removed in 'feature\_extraction' code!

from sklearn.decomposition import PCA #For 'Principle Component Analysis'.

n\_epochs=500 #Used only for debugging and not as condition for final update completion.

#expected\_obtained -- for confusion matrix.

zero\_zero,zero\_one,zero\_two,zero\_three,zero\_four=0,0,0,0,0

one\_zero,one\_one,one\_two,one\_three,one\_four=0,0,0,0,0

two\_zero,two\_one,two\_two,two\_three,two\_four=0,0,0,0,0

three\_zero,three\_one,three\_two,three\_three,three\_four=0,0,0,0,0

four\_zero,four\_one,four\_two,four\_three,four\_four=0,0,0,0,0

### Definition of all the activation functions ###

def tan\_hyp(y) :

return np.tanh(y)

def softmax(y) :

return np.exp(y-max(y))/np.sum(np.exp(y-max(y)))

def linear(y) :

return y

### Derivatives of the activation functions ###

def der\_softmax(y) :

a = np.zeros((len(y),len(y)))

np.fill\_diagonal(a,y)

y = np.reshape(y,(len(y),-1))

b = np.dot(y,np.transpose(y))

return a - b

def der\_tan\_hyp(y) :

return (1-y)\*(1+y)

def der\_linear(y) :

a = np.zeros((len(y),len(y)))

np.fill\_diagonal(a,y)

return a

def t\_exp(t):

if t == 0: #data['label'][i] == 0 :

return np.array([1.0,0.0,0.0,0.0,0.0]).reshape(n\_o,1)

elif t == 1:

return np.array([0.0,1.0,0.0,0.0,0.0]).reshape(n\_o,1)

elif t == 2:

return np.array([0.0,0.0,1.0,0.0,0.0]).reshape(n\_o,1)

elif t == 3:

return np.array([0.0,0.0,0.0,1.0,0.0]).reshape(n\_o,1)

elif t == 4:

return np.array([0.0,0.0,0.0,0.0,1.0]).reshape(n\_o,1)

else:

print('Unrecognized class label!!!')

pass

### Definition of loss functions for a single example ###

def mse(y,t) :

return 0.5\*np.sum((y-t)\*\*2)

def cross\_entropy(y,t):

return -np.sum(t\*np.log(y))

def grad\_cross\_entropy(y,t) :

return -np.divide(t,y)

def grad\_mse(y,t) :

return y-t

def forward\_pass\_customweights(activation\_functions,training\_sample,w) :

a\_list = [training\_sample]

h\_list = [training\_sample]

output = training\_sample

for (activation\_function,weight) in zip(activation\_functions,w) :

output = np.append(output,np.ones((1,1)),axis = 0)

output = np.dot(weight,output)

a\_list.append(output)

output = eval(activation\_function)(output)

h\_list.append(output)

return a\_list , h\_list

###class definition

class neural\_network :

def \_\_init\_\_(self, n\_hidden,list\_nodes,loss\_fn,list\_activation\_functions,learning\_rate,momentum,bp\_method,p1,p2) :

self.n\_hidden = n\_hidden

self.list\_nodes = list\_nodes

self.loss\_fn = loss\_fn

self.list\_activation\_functions = list\_activation\_functions

self.learning\_rate = learning\_rate

self.momentum = momentum

self.weights = []

self.delta\_weight\_past = []

self.bp\_method=bp\_method

self.q\_previous=[]

self.r\_previous=[]

self.m=1 #Used for updating adam optimizer's 'm' parameter.

self.p1=p1

self.p2=p2

self.epsilon=1e-8

for i in range(n\_hidden+1) :

np.random.seed(42)

weight\_layer = np.random.randn(list\_nodes[i+1],list\_nodes[i]+1)

self.q\_previous.append(np.zeros((list\_nodes[i+1],list\_nodes[i]+1)))

self.r\_previous.append(np.zeros((list\_nodes[i+1],list\_nodes[i]+1)))

delta\_weight\_past\_layer = np.zeros((list\_nodes[i+1],list\_nodes[i]+1))

self.weights.append(weight\_layer)

self.delta\_weight\_past.append(delta\_weight\_past\_layer)

def forward\_pass(self,training\_sample):

a\_list = [training\_sample]

h\_list = [training\_sample]

output = training\_sample

for (activation\_function,weight) in zip(self.list\_activation\_functions,self.weights) :

output = np.append(output,np.ones((1,1)),axis = 0)

output = np.dot(weight,output)

a\_list.append(output)

output = eval(activation\_function)(output)

h\_list.append(output)

return a\_list , h\_list

def calc\_loss(self,calc\_output,actual\_output) :

return eval(self.loss\_fn)(calc\_output,actual\_output)

def backward\_pass(self,h\_list,intermediate\_loss,actual\_output) :

'''implementation of backpropagation'''

der\_output\_act = eval('der\_'+self.list\_activation\_functions[-1])(h\_list[-1])

grad\_lf = eval('grad\_'+self.loss\_fn)(h\_list[-1],actual\_output)

grad\_layer = np.dot(der\_output\_act,grad\_lf)

for i in range(len(self.weights)) :

grad\_weight\_layer = np.dot(grad\_layer,np.transpose(np.append(h\_list[len(h\_list)-i-2],np.ones((1,1)),axis = 0)))

grad\_prev\_output = np.dot(np.transpose(self.weights[len(self.weights)-i-1])[:-1,:],grad\_layer)

grad\_layer = grad\_prev\_output \* eval('der\_'+self.list\_activation\_functions[len(self.list\_activation\_functions)-i-2])(h\_list[len(h\_list)-i-2])

if self.bp\_method==0:

self.weights[len(self.weights)-i-1] = self.gen\_delta\_rule(self.weights[len(self.weights)-i-1],grad\_weight\_layer,self.delta\_weight\_past[len(self.delta\_weight\_past)-i-1])

elif self.bp\_method==1:

self.weights[len(self.weights)-i-1] = self.adam\_rule(self.weights[len(self.weights)-i-1],grad\_weight\_layer,len(self.weights)-i-1)

self.delta\_weight\_past[len(self.delta\_weight\_past)-i-1] = self.learning\_rate\*grad\_weight\_layer

def gen\_delta\_rule(self,weights\_present,grad\_present,weight\_change\_previous) :

return weights\_present - (grad\_present\*self.learning\_rate) - (weight\_change\_previous\*self.momentum)

def adam\_rule(self,weights\_present,grad\_present,index):

q\_current = self.p1\*self.q\_previous[index] + (1-self.p1)\*grad\_present

r\_current = self.p2\*self.r\_previous[index] + (1-self.p2)\*grad\_present\*\*2

self.q\_previous[index]=q\_current;

self.r\_previous[index]=r\_current;

new\_weights = weights\_present - (self.learning\_rate\*np.divide((q\_current/(1-self.p1\*\*self.m)),(self.epsilon+np.sqrt((r\_current/(1-self.p2\*\*self.m))))))

self.m+=1

return new\_weights

def train(self,input\_array,output\_array,valid\_input\_array,valid\_output\_array) :

loss\_list = []

valid\_loss\_list = []

epoch\_num = 0

prev\_loss\_avg = 0

loss\_avg = 0

while (epoch\_num<=0 or abs(prev\_loss\_avg - loss\_avg)>=1e-4):

prev\_loss\_avg = loss\_avg

loss\_avg = 0

print('Epoch Number : ',epoch\_num)

print('Started')

print('----------------------------------------')

for (training\_sample,actual\_output) in zip(input\_array,output\_array) :

a\_list,h\_list = self.forward\_pass(training\_sample)

intermediate\_loss = self.calc\_loss(h\_list[-1],actual\_output)

self.backward\_pass(h\_list,intermediate\_loss,actual\_output)

loss\_avg = loss\_avg + intermediate\_loss

loss\_avg = loss\_avg/len(input\_array)

loss\_list.append(loss\_avg)

valid\_loss\_avg = 0

for (training\_sample,actual\_output) in zip(valid\_input\_array,valid\_output\_array) :

a\_list,h\_list = self.forward\_pass(training\_sample)

intermediate\_loss = self.calc\_loss(h\_list[-1],actual\_output)

valid\_loss\_avg = valid\_loss\_avg + intermediate\_loss

valid\_loss\_avg = valid\_loss\_avg/len(valid\_input\_array)

valid\_loss\_list.append(valid\_loss\_avg)

print('Average Loss : ',loss\_avg)

print('----------------------------------------')

epoch\_num = epoch\_num + 1

return loss\_list,valid\_loss\_list

def test(self,input\_array,output\_array) :

global zero\_zero,zero\_one,zero\_two,zero\_three,zero\_four;

global one\_zero,one\_one,one\_two,one\_three,one\_four;

global two\_zero,two\_one,two\_two,two\_three,two\_four;

global three\_zero,three\_one,three\_two,three\_three,three\_four;

global four\_zero,four\_one,four\_two,four\_three,four\_four;

accuracy = 0

for (sample,output) in zip(input\_array,output\_array) :

a\_list,h\_list = self.forward\_pass(sample)

calc\_output = h\_list[-1]

# print('##',np.argmax(calc\_output),np.wher

# print('y t: ',np.argmax(calc\_output),np.argmax(output))

if(np.argmax(output)==0):

if(np.argmax(calc\_output)==0):

zero\_zero+=1

elif(np.argmax(calc\_output)==1):

zero\_one+=1

elif(np.argmax(calc\_output)==2):

zero\_two+=1

elif(np.argmax(calc\_output)==3):

zero\_three+=1

elif(np.argmax(calc\_output)==4):

zero\_four+=1

if(np.argmax(output)==1):

if(np.argmax(calc\_output)==0):

one\_zero+=1

elif(np.argmax(calc\_output)==1):

one\_one+=1

elif(np.argmax(calc\_output)==2):

one\_two+=1

elif(np.argmax(calc\_output)==3):

one\_three+=1

elif(np.argmax(calc\_output)==4):

one\_four+=1

if(np.argmax(output)==2):

if(np.argmax(calc\_output)==0):

two\_zero+=1

elif(np.argmax(calc\_output)==1):

two\_one+=1

elif(np.argmax(calc\_output)==2):

two\_two+=1

elif(np.argmax(calc\_output)==3):

two\_three+=1

elif(np.argmax(calc\_output)==4):

two\_four+=1

if(np.argmax(output)==3):

if(np.argmax(calc\_output)==0):

three\_zero+=1

elif(np.argmax(calc\_output)==1):

three\_one+=1

elif(np.argmax(calc\_output)==2):

three\_two+=1

elif(np.argmax(calc\_output)==3):

three\_three+=1

elif(np.argmax(calc\_output)==4):

three\_four+=1

if(np.argmax(output)==4):

if(np.argmax(calc\_output)==0):

four\_zero+=1

elif(np.argmax(calc\_output)==1):

four\_one+=1

elif(np.argmax(calc\_output)==2):

four\_two+=1

elif(np.argmax(calc\_output)==3):

four\_three+=1

elif(np.argmax(calc\_output)==4):

four\_four+=1

if np.argmax(calc\_output) == np.argmax(output) :

accuracy = accuracy + 1

accuracy = accuracy / len(input\_array)

return accuracy

##### Reading the data for the code #####

data = None

input\_array = []

output\_array = []

test\_input\_array = []

train\_input\_array= []

n\_pca,n\_h1, n\_h2,n\_o,etta=20,10,5,5,0.001 #hp1 #int(sys.argv[1]), int(sys.argv[2]), float(sys.argv[3]) -- getting inputs from user

momentum\_factor=0.9 #hp2 - for generalized delta rule

p1,p2=0.9,0.999 #hp3 - for adam optimizer

pca = PCA(n\_components = n\_pca)

data=np.array(pca.fit\_transform(lf.data\_points))

print('Total Number of Training Examples: ',data,np.shape(data))

#lf.shuffle(data) //Shuffle data if required.

for i in range(int(len(data)/n\_o)): #Doing this to ensure equal output class contribution to both validation and training.

input\_array.append(np.array(data[i]).reshape(n\_pca,1));

output\_array.append(t\_exp(lf.data\_points\_class[i]));

input\_array.append(np.array(data[int(len(data)/n\_o)+i]).reshape(n\_pca,1));

output\_array.append(t\_exp(lf.data\_points\_class[int(len(data)/n\_o)+i]));

input\_array.append(np.array(data[2\*int(len(data)/n\_o)+i]).reshape(n\_pca,1));

output\_array.append(t\_exp(lf.data\_points\_class[2\*int(len(data)/n\_o)+i]));

input\_array.append(np.array(data[3\*int(len(data)/n\_o)+i]).reshape(n\_pca,1));

output\_array.append(t\_exp(lf.data\_points\_class[3\*int(len(data)/n\_o)+i]));

input\_array.append(np.array(data[4\*int(len(data)/n\_o)+i]).reshape(n\_pca,1));

output\_array.append(t\_exp(lf.data\_points\_class[4\*int(len(data)/n\_o)+i]));

test\_input\_array = input\_array[int(np.floor(len(data)\*3/4)):]

test\_output\_array = output\_array[int(np.floor(len(data)\*3/4)):]

train\_input\_array = input\_array[:int(np.floor(len(data)\*3/4))]

train\_output\_array = output\_array[:int(np.floor(len(data)\*3/4))]

nn = neural\_network(2,[n\_pca,n\_h1,n\_h2,n\_o],'cross\_entropy',['tan\_hyp','tan\_hyp','softmax'],etta,momentum\_factor,1,p1,p2) #$$$'1' for adam and '0' for momentum backprop updates.

list\_avg\_error,list\_valid\_avg\_error = nn.train(train\_input\_array,train\_output\_array,test\_input\_array,test\_output\_array)

accuracy = nn.test(test\_input\_array,test\_output\_array)

print("Training Finished")

print("---------------------------------------------")

print("Accuracy : ",accuracy)

print("0: ",zero\_zero,zero\_one,zero\_two,zero\_three,zero\_four)

print("1: ",one\_zero,one\_one,one\_two,one\_three,one\_four)

print("2 :",two\_zero,two\_one,two\_two,two\_three,two\_four)

print("3 :",three\_zero,three\_one,three\_two,three\_three,three\_four)

print("4: ",four\_zero,four\_one,four\_two,four\_three,four\_four)

pickle.dump(nn.weights, open('weights.sav','wb'));