###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,p
2) :
        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[l
en(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-sel
f.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,mo
mentum_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'));
```