#import libraries and functions to load the data

from digits import get\_mnist

from matplotlib import pyplot as plt

import numpy as np

import ast

import sys

import numpy.testing as npt

import pytest

import random

random.seed(1)

np.random.seed(1)

trX, trY, tsX, tsY = get\_mnist()

print('trX.shape: ', trX.shape)

print('trY.shape: ', trY.shape)

print('tsX.shape: ', tsX.shape)

print('tsY.shape: ', tsY.shape)

# The data is of the format (no\_samples, channels, img\_height, img\_width)

# In the training data trX, there are 60000 images. Each image has one channel (gray scale).

# Each image is of height=28 and width=28 pixels

# Lets sample a smaller subest to work with.

# We will use 2000 training examples and 1000 test samples.

# We define a function which we can use later as well

def sample\_mnist(n\_train=2000, n\_test=1000):

trX, trY, tsX, tsY = get\_mnist()

random.seed(1)

np.random.seed(1)

tr\_idx = np.random.choice(trX.shape[0], n\_train)

trX = trX[tr\_idx]

trY = trY[tr\_idx]

ts\_idx = np.random.choice(tsX.shape[0], n\_train)

tsX = tsX[ts\_idx]

tsY = tsY[ts\_idx]

trX = trX.reshape(-1, 28\*28).T

trY = trY.reshape(1, -1)

tsX = tsX.reshape(-1, 28\*28).T

tsY = tsY.reshape(1, -1)

return trX, trY, tsX, tsY

# Lets verify the function

trX, trY, tsX, tsY = sample\_mnist(n\_train=2000, n\_test=1000)

# Lets examine the data and see if it is normalized

print('trX.shape: ', trX.shape)

print('trY.shape: ', trY.shape)

print('tsX.shape: ', tsX.shape)

print('tsY.shape: ', tsY.shape)

print('Train max: value = {}, Train min: value = {}'.format(np.max(trX), np.min(trX)))

print('Test max: value = {}, Test min: value = {}'.format(np.max(tsX), np.min(tsX)))

print('Unique labels in train: ', np.unique(trY))

print('Unique labels in test: ', np.unique(tsY))

# Let's visualize a few samples and their labels from the train and test datasets.

print('\nDisplaying a few samples')

visx = np.concatenate((trX[:,:50],tsX[:,:50]), axis=1).reshape(28,28,10,10).transpose(2,0,3,1).reshape(28\*10,-1)

visy = np.concatenate((trY[:,:50],tsY[:,:50]), axis=1).reshape(10,-1)

print('labels')

print(visy)

plt.figure(figsize = (8,8))

plt.axis('off')

plt.imshow(visx, cmap='gray');

def relu(Z):

'''

Computes relu activation of input Z

Inputs:

Z: numpy.ndarray (n, m) which represent 'm' samples each of 'n' dimension

Outputs:

A: where A = ReLU(Z) is a numpy.ndarray (n, m) representing 'm' samples each of 'n' dimension

cache: a dictionary with {"Z", Z}

'''

cache = {}

# your code here

**if type(Z)==list:**

**Z = np.array(Z)**

**A =np.maximum(0,Z)**

**cache = {"Z": Z}**

return A, cache

#Test

z\_tst = [-2,-1,0,1,2]

a\_tst, c\_tst = relu(z\_tst)

npt.assert\_array\_equal(a\_tst,[0,0,0,1,2])

npt.assert\_array\_equal(c\_tst["Z"], [-2,-1,0,1,2])

def relu\_der(dA, cache):

'''

Computes derivative of relu activation

Inputs:

dA: derivative from the subsequent layer of dimension (n, m).

dA is multiplied elementwise with the gradient of ReLU

cache: dictionary with {"Z", Z}, where Z was the input

to the activation layer during forward propagation

Outputs:

dZ: the derivative of dimension (n,m). It is the elementwise

product of the derivative of ReLU and dA

'''

dZ = np.array(dA, copy=True)

Z = cache["Z"]

# your code here

**relu\_val = np.zeros(Z.shape)**

**indixes = np.where(Z > 0)**

**relu\_val[indixes] = 1**

**dZ = np.multiply(relu\_val,dZ)**

return dZ

#Test

dA\_tst = np.array([[0,2],[1,1]])

cache\_tst = {}

cache\_tst['Z'] = np.array([[-1,2],[1,-2]])

npt.assert\_array\_equal(relu\_der(dA\_tst,cache\_tst),np.array([[0,2],[1,0]]))

def linear(Z):

'''

Computes linear activation of Z

This function is implemented for completeness

Inputs:

Z: numpy.ndarray (n, m) which represent 'm' samples each of 'n' dimension

Outputs:

A: where A = Linear(Z) is a numpy.ndarray (n, m) representing 'm' samples each of 'n' dimension

cache: a dictionary with {"Z", Z}

'''

A = Z

cache = {}

cache["Z"] = Z

return A, cache

def linear\_der(dA, cache):

'''

Computes derivative of linear activation

This function is implemented for completeness

Inputs:

dA: derivative from the subsequent layer of dimension (n, m).

dA is multiplied elementwise with the gradient of Linear(.)

cache: dictionary with {"Z", Z}, where Z was the input

to the activation layer during forward propagation

Outputs:

dZ: the derivative of dimension (n,m). It is the elementwise

product of the derivative of Linear(.) and dA

'''

dZ = np.array(dA, copy=True)

return dZ

def softmax\_cross\_entropy\_loss(Z, Y=np.array([])):

'''

Computes the softmax activation of the inputs Z

Estimates the cross entropy loss

Inputs:

Z: numpy.ndarray (n, m)

Y: numpy.ndarray (1, m) of labels

when y=[] loss is set to []

Outputs:

A: numpy.ndarray (n, m) of softmax activations

cache: a dictionary to store the activations which will be used later to estimate derivatives

loss: cost of prediction

'''

# your code here

Zmax = np.max(Z, *axis*=0)

    e = np.exp(Z-Zmax)

    t = e.sum(*axis* = 0)

    A = e/t

    if Y.size > 0:

        n,m = A.shape

        one\_hot = np.eye(n)[Y.reshape(-1)].T

        loss = np.sum(-np.multiply(one\_hot,np.log(A)))/m

    else:

        loss=[]

    cache = {}

    cache["A"] = A

    return A, cache, loss

#test cases for softmax\_cross\_entropy\_loss

np.random.seed(1)

Z\_t = np.random.randn(3,4)

Y\_t = np.array([[1,0,1,2]])

A\_t = np.array([[0.57495949, 0.38148818, 0.05547572, 0.36516899],

[0.26917503, 0.07040735, 0.53857622, 0.49875847],

[0.15586548, 0.54810447, 0.40594805, 0.13607254]])

A\_est, cache\_est, loss\_est = softmax\_cross\_entropy\_loss(Z\_t, Y\_t)

npt.assert\_almost\_equal(loss\_est,1.2223655548779273,decimal=5)

npt.assert\_array\_almost\_equal(A\_est,A\_t,decimal=5)

npt.assert\_array\_almost\_equal(cache\_est['A'],A\_t,decimal=5)

# hidden test cases follow

def softmax\_cross\_entropy\_loss\_der(Y, cache):

'''

Computes the derivative of the softmax activation and cross entropy loss

Inputs:

Y: numpy.ndarray (1, m) of labels

cache: a dictionary with cached activations A of size (n,m)

Outputs:

dZ: derivative dL/dZ - a numpy.ndarray of dimensions (n, m)

'''

A = cache["A"]

# your code here

**n,m = A.shape**

**labels=Y.flatten()**

**for i in range(m):**

**A[labels[i]][i] -= 1**

**dZ = A/m**

return dZ

#test cases for softmax\_cross\_entropy\_loss\_der

np.random.seed(1)

Z\_t = np.random.randn(3,4)

Y\_t = np.array([[1,0,1,2]])

A\_t = np.array([[0.57495949, 0.38148818, 0.05547572, 0.36516899],

[0.26917503, 0.07040735, 0.53857622, 0.49875847],

[0.15586548, 0.54810447, 0.40594805, 0.13607254]])

cache\_t={}

cache\_t['A'] = A\_t

dZ\_t = np.array([[ 0.14373987, -0.15462795, 0.01386893, 0.09129225],

[-0.18270624, 0.01760184, -0.11535594, 0.12468962],

[ 0.03896637, 0.13702612, 0.10148701, -0.21598186]])

dZ\_est = softmax\_cross\_entropy\_loss\_der(Y\_t, cache\_t)

npt.assert\_almost\_equal(dZ\_est,dZ\_t,decimal=5)

# hidden test cases follow

def initialize\_network(net\_dims):

'''

Initializes the parameters of a multi-layer neural network

Inputs:

net\_dims: List containing the dimensions of the network. The values of the array represent the number of nodes in

each layer. For Example, if a Neural network contains 784 nodes in the input layer, 800 in the first hidden layer,

500 in the secound hidden layer and 10 in the output layer, then net\_dims = [784,800,500,10].

Outputs:

parameters: Python Dictionary for storing the Weights and bias of each layer of the network

'''

numLayers = len(net\_dims)

parameters = {}

for l in range(numLayers-1):

# Hint:

# parameters["W"+str(l+1)] =

# parameters["b"+str(l+1)] =

# your code here

**parameters["W"+str(l+1)] = np.random.randn(net\_dims[l+1],net\_dims[l])\*0.01**

**parameters["b"+str(l+1)] = np.zeros((net\_dims[l+1],1))**

return parameters

#Test

net\_dims\_tst = [5,4,1]

parameters\_tst = initialize\_network(net\_dims\_tst)

assert parameters\_tst['W1'].shape == (4,5)

assert parameters\_tst['W2'].shape == (1,4)

assert parameters\_tst['b1'].shape == (4,1)

assert parameters\_tst['b2'].shape == (1,1)

assert parameters\_tst['b1'].all() == 0

assert parameters\_tst['b2'].all() == 0

# There are hidden tests

def linear\_forward(A\_prev, W, b):

'''

Input A\_prev propagates through the layer

Z = WA + b is the output of this layer.

Inputs:

A\_prev: numpy.ndarray (n,m) the input to the layer

W: numpy.ndarray (n\_out, n) the weights of the layer

b: numpy.ndarray (n\_out, 1) the bias of the layer

Outputs:

Z: where Z = W.A\_prev + b, where Z is the numpy.ndarray (n\_out, m) dimensions

cache: a dictionary containing the inputs A

'''

# your code here

**Z = np.dot(W,A\_prev) + b**

cache = {}

cache["A"] = A\_prev

return Z, cache

#Hidden test cases follow

np.random.seed(1)

n1 = 3

m1 = 4

A\_prev\_t = np.random.randn(n1,m1)

W\_t = np.random.randn(n1, n1)

b\_t = np.random.randn(n1, 1)

Z\_est, cache\_est = linear\_forward(A\_prev\_t, W\_t, b\_t)

def layer\_forward(A\_prev, W, b, activation):

'''

Input A\_prev propagates through the layer and the activation

Inputs:

A\_prev: numpy.ndarray (n,m) the input to the layer

W: numpy.ndarray (n\_out, n) the weights of the layer

b: numpy.ndarray (n\_out, 1) the bias of the layer

activation: is the string that specifies the activation function

Outputs:

A: = g(Z), where Z = WA + b, where Z is the numpy.ndarray (n\_out, m) dimensions

g is the activation function

cache: a dictionary containing the cache from the linear and the nonlinear propagation

to be used for derivative

'''

Z, lin\_cache = linear\_forward(A\_prev, W, b)

if activation == "relu":

A, act\_cache = relu(Z)

elif activation == "linear":

A, act\_cache = linear(Z)

cache = {}

cache["lin\_cache"] = lin\_cache

cache["act\_cache"] = act\_cache

return A, cache

def multi\_layer\_forward(A0, parameters):

'''

Forward propgation through the layers of the network

Inputs:

A0: numpy.ndarray (n,m) with n features and m samples

parameters: dictionary of network parameters {"W1":[..],"b1":[..],"W2":[..],"b2":[..]...}

Outputs:

AL: numpy.ndarray (c,m) - outputs of the last fully connected layer before softmax

where c is number of categories and m is number of samples

caches: a dictionary of associated caches of parameters and network inputs

'''

L = len(parameters)//2

A = A0

caches = []

for l in range(1,L):

A, cache = layer\_forward(A, parameters["W"+str(l)], parameters["b"+str(l)], "relu")

caches.append(cache)

AL, cache = layer\_forward(A, parameters["W"+str(L)], parameters["b"+str(L)], "linear")

caches.append(cache)

return AL, caches

def linear\_backward(dZ, cache, W, b):

'''

Backward prpagation through the linear layer

Inputs:

dZ: numpy.ndarray (n,m) derivative dL/dz

cache: a dictionary containing the inputs A, for the linear layer

where Z = WA + b,

Z is (n,m); W is (n,p); A is (p,m); b is (n,1)

W: numpy.ndarray (n,p)

b: numpy.ndarray (n, 1)

Outputs:

dA\_prev: numpy.ndarray (p,m) the derivative to the previous layer

dW: numpy.ndarray (n,p) the gradient of W

db: numpy.ndarray (n, 1) the gradient of b

'''

A = cache["A"]

# your code here

**n,m = A.shape**

**dA\_prev = np.dot(W.T, dZ)**

**dW = np.dot(dZ,A.T)/m**

**db = np.sum(dZ, axis=1, keepdims=True)**

return dA\_prev, dW, db

**NOW FIXED removed /m at dW = ….**

#Hidden test cases follow

np.random.seed(1)

n1 = 3

m1 = 4

p1 = 5

dZ\_t = np.random.randn(n1,m1)

A\_t = np.random.randn(p1,m1)

cache\_t = {}

cache\_t['A'] = A\_t

W\_t = np.random.randn(n1,p1)

b\_t = np.random.randn(n1,1)

dA\_prev\_est, dW\_est, db\_est = linear\_backward(dZ\_t, cache\_t, W\_t, b\_t)

def layer\_backward(dA, cache, W, b, activation):

'''

Backward propagation through the activation and linear layer

Inputs:

dA: numpy.ndarray (n,m) the derivative to the previous layer

cache: dictionary containing the linear\_cache and the activation\_cache

activation - activation of the layer

W: numpy.ndarray (n,p)

b: numpy.ndarray (n, 1)

Outputs:

dA\_prev: numpy.ndarray (p,m) the derivative to the previous layer

dW: numpy.ndarray (n,p) the gradient of W

db: numpy.ndarray (n, 1) the gradient of b

'''

lin\_cache = cache["lin\_cache"]

act\_cache = cache["act\_cache"]

if activation == "relu":

dZ = relu\_der(dA, act\_cache)

elif activation == "linear":

dZ = linear\_der(dA, act\_cache)

dA\_prev, dW, db = linear\_backward(dZ, lin\_cache, W, b)

return dA\_prev, dW, db

def multi\_layer\_backward(dAL, caches, parameters):

'''

Back propgation through the layers of the network (except softmax cross entropy)

softmax\_cross\_entropy can be handled separately

Inputs:

dAL: numpy.ndarray (n,m) derivatives from the softmax\_cross\_entropy layer

caches: a dictionary of associated caches of parameters and network inputs

parameters - dictionary of network parameters {"W1":[..],"b1":[..],"W2":[..],"b2":[..]...}

Outputs:

gradients: dictionary of gradient of network parameters

{"dW1":[..],"db1":[..],"dW2":[..],"db2":[..],...}

'''

L = len(caches)

gradients = {}

dA = dAL

activation = "linear"

for l in reversed(range(1,L+1)):

dA, gradients["dW"+str(l)], gradients["db"+str(l)] = \

layer\_backward(dA, caches[l-1], \

parameters["W"+str(l)],parameters["b"+str(l)],\

activation)

activation = "relu"

return gradients

def classify(X, parameters):

'''

Network prediction for inputs X

Inputs:

X: numpy.ndarray (n,m) with n features and m samples

parameters: dictionary of network parameters

{"W1":[..],"b1":[..],"W2":[..],"b2":[..],...}

Outputs:

YPred: numpy.ndarray (1,m) of predictions

'''

# Forward propagate input 'X' using multi\_layer\_forward(.) and obtain the final activation 'A'

# Using 'softmax\_cross\_entropy loss(.)', obtain softmax activation 'AL' with input 'A' from step 1

# Predict class label 'YPred' as the 'argmax' of softmax activation from step-2.

# Note: the shape of 'YPred' is (1,m), where m is the number of samples

# your code here

**AL, caches = multi\_layer\_forward(X, parameters)**

**A, cache, loss = softmax\_cross\_entropy\_loss(AL)**

**YPred = np.argmax(A, axis=0)**

return YPred

#Hidden test cases follow

np.random.seed(1)

n1 = 3

m1 = 4

p1 = 2

X\_t = np.random.randn(n1,m1)

W1\_t = np.random.randn(p1,n1)

b1\_t = np.random.randn(p1,1)

W2\_t = np.random.randn(p1,p1)

b2\_t = np.random.randn(p1,1)

parameters\_t = {'W1':W1\_t, 'b1':b1\_t, 'W2':W2\_t, 'b2':b2\_t}

YPred\_est = classify(X\_t, parameters\_t)

def update\_parameters(parameters, gradients, epoch,alpha):

'''

Updates the network parameters with gradient descent

Inputs:

parameters: dictionary of network parameters

{"W1":[..],"b1":[..],"W2":[..],"b2":[..],...}

gradients: dictionary of gradient of network parameters

{"dW1":[..],"db1":[..],"dW2":[..],"db2":[..],...}

epoch: epoch number

alpha: step size or learning rate

Outputs:

parameters: updated dictionary of network parameters

{"W1":[..],"b1":[..],"W2":[..],"b2":[..],...}

'''

L = len(parameters)//2

for i in range(L):

#parameters["W"+str(i+1)] =

#parameters["b"+str(i+1)] =

# your code here

**parameters["W"+str(i+1)] = parameters["W"+str(i+1)]- alpha \* gradients["dW"+str(i+1)]**

**parameters["b"+str(i+1)] = parameters["b"+str(i+1)]- alpha \* gradients["db"+str(i+1)]**

return parameters

def multi\_layer\_network(X, Y, net\_dims,num\_iterations,learning\_rate, log=False):

''' learning\_rate: step size for gradient descent

log: boolean to print training progression

Outputs:

costs: list of costs (or loss) over training

parameters: dictionary of trained network parameters

'''

parameters = initialize\_network(net\_dims)

A0 = X.T

costs = []

num\_classes = 10

alpha = learning\_rate

for ii in range(num\_iterations):

## Forward Propagation

# Step 1: Input 'A0' and 'parameters' into the network using multi\_layer\_forward()

# and calculate output of last layer 'A' (before softmax) and obtain cached activations as 'caches'

# Step 2: Input 'A' and groundtruth labels 'Y' to softmax\_cros\_entropy\_loss(.) and estimate

# activations 'AL', 'softmax\_cache' and 'loss'

## Back Propagation

# Step 3: Estimate gradient 'dAL' with softmax\_cros\_entropy\_loss\_der(.) using groundtruth

# labels 'Y' and 'softmax\_cache'

# Step 4: Estimate 'gradients' with multi\_layer\_backward(.) using 'dAL' and 'parameters'

# Step 5: Estimate updated 'parameters' and updated learning rate 'alpha' with update\_parameters(.)

# using 'parameters', 'gradients', loop variable 'ii' (epoch number) and 'learning\_rate'

# Note: Use the same variable 'parameters' as input and output to the update\_parameters(.) function

# your code here

**AL, caches = multi\_layer\_forward(A0, parameters)**

**A, cache, loss = softmax\_cross\_entropy\_loss(AL, Y)**

**dAL = softmax\_cross\_entropy\_loss\_der(Y, cache)**

**gradients = multi\_layer\_backward(dAL, caches, parameters)**

**parameters = update\_parameters(parameters,gradients, alpha)**

**if ii % 20 == 0:**

**costs.append(loss)**

**if log:**

**print("Cost at iteration %i is: %.05f, learning rate: %.05f" %(ii+1, loss, learning\_rate))**

return costs, parameters

# You should be able to get a train accuracy of >90% and a test accuracy >85%

# The settings below gave >95% train accuracy and >90% test accuracy

# Feel free to adjust the values and explore how the network behaves

net\_dims = [784,200,10]

#784 is for image dimensions

#10 is for number of categories

#200 is arbitrary

# initialize learning rate and num\_iterations

learning\_rate = 0.1

num\_iterations = 500

np.random.seed(1)

print("Network dimensions are:" + str(net\_dims))

# getting the subset dataset from MNIST

trX, trY, tsX, tsY = sample\_mnist(n\_train=2000, n\_test=1000)

costs, parameters = multi\_layer\_network(trX, trY, net\_dims, \

num\_iterations=num\_iterations, learning\_rate=learning\_rate)

# compute the accuracy for training set and testing set

train\_Pred = classify(trX, parameters)

test\_Pred = classify(tsX, parameters)

#Estimate the training accuracy 'trAcc' and the testing accuracy 'teAcc'

# your code here

**Syntax errors solved**

trAcc = np.mean(np.double(train\_Pred == trY)) \* 100

teAcc = np.mean(np.double(test\_Pred == tsY)) \* 100

# print("Accuracy for training set is {0:0.3f} %".format(trAcc))

# print("Accuracy for testing set is {0:0.3f} %".format(teAcc))

plt.plot(costs)

plt.xlabel("Iterations")

plt.ylabel("Loss")

plt.show()

# Contains hidden tests testing for test data accuracy > 85%

[ValidateApp | INFO] Validating '/home/jovyan/work/submitted/courseraLearner/Assignment2/assignment\_2.ipynb' [ValidateApp | INFO] Executing notebook with kernel: python3 Tests failed on 4 cell(s)! These tests could be hidden. Please check your submission. ========================================================================================== The following cell failed: #test cases for softmax\_cross\_entropy\_loss np.random.seed(1) Z\_t = np.random.randn(3,4) Y\_t = np.array([[1,0,1,2]]) A\_t = np.array([[0.57495949, 0.38148818, 0.05547572, 0.36516899], [0.26917503, 0.07040735, 0.53857622, 0.49875847], [0.15586548, 0.54810447, 0.40594805, 0.13607254]]) A\_est, cache\_est, loss\_est = softmax\_cross\_entropy\_loss(Z\_t, Y\_t) npt.assert\_almost\_equal(loss\_est,1.2223655548779273,decimal=5) npt.assert\_array\_almost\_equal(A\_est,A\_t,decimal=5) npt.assert\_array\_almost\_equal(cache\_est['A'],A\_t,decimal=5) # hidden test cases follow The error was: --------------------------------------------------------------------------- NameError Traceback (most recent call last) <ipython-input-11-5daa2e808fb3> in <module> 7 [0.15586548, 0.54810447, 0.40594805, 0.13607254]]) 8 ----> 9 A\_est, cache\_est, loss\_est = softmax\_cross\_entropy\_loss(Z\_t, Y\_t) 10 npt.assert\_almost\_equal(loss\_est,1.2223655548779273,decimal=5) 11 npt.assert\_array\_almost\_equal(A\_est,A\_t,decimal=5) <ipython-input-10-231f971d4619> in softmax\_cross\_entropy\_loss(Z, Y) 25 indices = Y.flatten() 26 n,m = A.shape ---> 27 Y = np.array(tf.one\_hot(indices,n)).T 28 loss = np.sum(-np.multiply(Y, np.log(A)))/m 29 cache = {} NameError: name 'tf' is not defined ========================================================================================== The following cell failed: #Hidden test cases follow np.random.seed(1) n1 = 3 m1 = 4 p1 = 5 dZ\_t = np.random.randn(n1,m1) A\_t = np.random.randn(p1,m1) cache\_t = {} cache\_t['A'] = A\_t W\_t = np.random.randn(n1,p1) b\_t = np.random.randn(n1,1) dA\_prev\_est, dW\_est, db\_est = linear\_backward(dZ\_t, cache\_t, W\_t, b\_t) The error was: --------------------------------------------------------------------------- NameError Traceback (most recent call last) <ipython-input-21-c95711d99bac> in <module> 11 b\_t = np.random.randn(n1,1) 12 ---> 13 dA\_prev\_est, dW\_est, db\_est = linear\_backward(dZ\_t, cache\_t, W\_t, b\_t) <ipython-input-20-0f3360ffe2f0> in linear\_backward(dZ, cache, W, b) 19 # your code here 20 ---> 21 return dA\_prev, dW, db NameError: name 'dA\_prev' is not defined ========================================================================================== The following cell failed: #Hidden test cases follow np.random.seed(1) n1 = 3 m1 = 4 p1 = 2 X\_t = np.random.randn(n1,m1) W1\_t = np.random.randn(p1,n1) b1\_t = np.random.randn(p1,1) W2\_t = np.random.randn(p1,p1) b2\_t = np.random.randn(p1,1) parameters\_t = {'W1':W1\_t, 'b1':b1\_t, 'W2':W2\_t, 'b2':b2\_t} YPred\_est = classify(X\_t, parameters\_t) The error was: --------------------------------------------------------------------------- NameError Traceback (most recent call last) <ipython-input-25-1a8098478e06> in <module> 10 b2\_t = np.random.randn(p1,1) 11 parameters\_t = {'W1':W1\_t, 'b1':b1\_t, 'W2':W2\_t, 'b2':b2\_t} ---> 12 YPred\_est = classify(X\_t, parameters\_t) <ipython-input-24-55372e8ea7e7> in classify(X, parameters) 17 # your code here 18 ---> 19 return YPred NameError: name 'YPred' is not defined