import os

import numpy as np

import tensorflow as tf

import h5py

import math

def load\_dataset():

train\_dataset = h5py.File('train\_signs.h5', "r")

train\_set\_x\_orig = np.array(train\_dataset["train\_set\_x"][:]) # your train set features

train\_set\_y\_orig = np.array(train\_dataset["train\_set\_y"][:]) # your train set labels

test\_dataset = h5py.File('test\_signs.h5', "r")

test\_set\_x\_orig = np.array(test\_dataset["test\_set\_x"][:]) # your test set features

test\_set\_y\_orig = np.array(test\_dataset["test\_set\_y"][:]) # your test set labels

classes = np.array(test\_dataset["list\_classes"][:]) # the list of classes

train\_set\_y\_orig = train\_set\_y\_orig.reshape((1, train\_set\_y\_orig.shape[0]))

test\_set\_y\_orig = test\_set\_y\_orig.reshape((1, test\_set\_y\_orig.shape[0]))

return train\_set\_x\_orig, train\_set\_y\_orig, test\_set\_x\_orig, test\_set\_y\_orig, classes

def random\_mini\_batches(X, Y, mini\_batch\_size = 64, seed = 0):

"""

Creates a list of random minibatches from (X, Y)

Arguments:

X -- input data, of shape (input size, number of examples) (m, Hi, Wi, Ci)

Y -- true "label" vector (containing 0 if cat, 1 if non-cat), of shape (1, number of examples) (m, n\_y)

mini\_batch\_size - size of the mini-batches, integer

seed -- this is only for the purpose of grading, so that you're "random minibatches are the same as ours.

Returns:

mini\_batches -- list of synchronous (mini\_batch\_X, mini\_batch\_Y)

"""

m = X.shape[0] # number of training examples

mini\_batches = []

np.random.seed(seed)

# Step 1: Shuffle (X, Y)

permutation = list(np.random.permutation(m))

shuffled\_X = X[permutation,:,:,:]

shuffled\_Y = Y[permutation,:]

# Step 2: Partition (shuffled\_X, shuffled\_Y). Minus the end case.

num\_complete\_minibatches = math.floor(m/mini\_batch\_size) # number of mini batches of size mini\_batch\_size in your partitionning

for k in range(0, num\_complete\_minibatches):

mini\_batch\_X = shuffled\_X[k \* mini\_batch\_size : k \* mini\_batch\_size + mini\_batch\_size,:,:,:]

mini\_batch\_Y = shuffled\_Y[k \* mini\_batch\_size : k \* mini\_batch\_size + mini\_batch\_size,:]

mini\_batch = (mini\_batch\_X, mini\_batch\_Y)

mini\_batches.append(mini\_batch)

# Handling the end case (last mini-batch < mini\_batch\_size)

if m % mini\_batch\_size != 0:

mini\_batch\_X = shuffled\_X[num\_complete\_minibatches \* mini\_batch\_size : m,:,:,:]

mini\_batch\_Y = shuffled\_Y[num\_complete\_minibatches \* mini\_batch\_size : m,:]

mini\_batch = (mini\_batch\_X, mini\_batch\_Y)

mini\_batches.append(mini\_batch)

return mini\_batches

def convert\_to\_one\_hot(Y, C):

Y = np.eye(C)[Y.reshape(-1)].T

return Y

def forward\_propagation\_for\_predict(X, parameters):

"""

Implements the forward propagation for the model: LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> SOFTMAX

Arguments:

X -- input dataset placeholder, of shape (input size, number of examples)

parameters -- python dictionary containing your parameters "W1", "b1", "W2", "b2", "W3", "b3"

the shapes are given in initialize\_parameters

Returns:

Z3 -- the output of the last LINEAR unit

"""

# Retrieve the parameters from the dictionary "parameters"

W1 = parameters['W1']

b1 = parameters['b1']

W2 = parameters['W2']

b2 = parameters['b2']

W3 = parameters['W3']

b3 = parameters['b3']

# Numpy Equivalents:

Z1 = tf.add(tf.matmul(W1, X), b1) # Z1 = np.dot(W1, X) + b1

A1 = tf.nn.relu(Z1) # A1 = relu(Z1)

Z2 = tf.add(tf.matmul(W2, A1), b2) # Z2 = np.dot(W2, a1) + b2

A2 = tf.nn.relu(Z2) # A2 = relu(Z2)

Z3 = tf.add(tf.matmul(W3, A2), b3) # Z3 = np.dot(W3,Z2) + b3

return Z3

def predict(X, parameters):

W1 = tf.convert\_to\_tensor(parameters["W1"])

b1 = tf.convert\_to\_tensor(parameters["b1"])

W2 = tf.convert\_to\_tensor(parameters["W2"])

b2 = tf.convert\_to\_tensor(parameters["b2"])

W3 = tf.convert\_to\_tensor(parameters["W3"])

b3 = tf.convert\_to\_tensor(parameters["b3"])

params = {"W1": W1,

"b1": b1,

"W2": W2,

"b2": b2,

"W3": W3,

"b3": b3}

x = tf.placeholder("float", [12288, 1])

z3 = forward\_propagation\_for\_predict(x, params)

p = tf.argmax(z3)

sess = tf.Session()

prediction = sess.run(p, feed\_dict = {x: X})

return prediction