import numpy as np

import h5py

import matplotlib.pyplot as plt

from testCases import \*

from dnn\_utils import sigmoid, sigmoid\_backward, relu, relu\_backward

from public\_tests import \*

%matplotlib inline

plt.rcParams['figure.figsize'] = (5.0, 4.0) # set default size of plots

plt.rcParams['image.interpolation'] = 'nearest'

plt.rcParams['image.cmap'] = 'gray'

%load\_ext autoreload

%autoreload 2

np.random.seed(1)

def initialize\_parameters(n\_x, n\_h, n\_y):

"""

Argument:

n\_x -- size of the input layer

n\_h -- size of the hidden layer

n\_y -- size of the output layer

Returns:

parameters -- python dictionary containing your parameters:

W1 -- weight matrix of shape (n\_h, n\_x)

b1 -- bias vector of shape (n\_h, 1)

W2 -- weight matrix of shape (n\_y, n\_h)

b2 -- bias vector of shape (n\_y, 1)

"""

np.random.seed(1)

#(≈ 4 lines of code)

# W1 = ...

# b1 = ...

# W2 = ...

# b2 = ...

# YOUR CODE STARTS HERE

W1=np.random.randn(n\_h,n\_x)\*0.01

b1=np.zeros((n\_h, 1))

W2=np.random.randn(n\_y,n\_h)\*0.01

b2=np.zeros((n\_y,1))

# YOUR CODE ENDS HERE

parameters = {"W1": W1,

"b1": b1,

"W2": W2,

"b2": b2}

return parameters

def initialize\_parameters\_deep(layer\_dims):

"""

Arguments:

layer\_dims -- python array (list) containing the dimensions of each layer in our network

Returns:

parameters -- python dictionary containing your parameters "W1", "b1", ..., "WL", "bL":

Wl -- weight matrix of shape (layer\_dims[l], layer\_dims[l-1])

bl -- bias vector of shape (layer\_dims[l], 1)

"""

np.random.seed(3)

parameters = {}

L = len(layer\_dims) # number of layers in the network

for l in range(1, L):

#(≈ 2 lines of code)

# parameters['W' + str(l)] = ...

# parameters['b' + str(l)] = ...

# YOUR CODE STARTS HERE

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

parameters['b'+str(l)]=np.zeros((layer\_dims[l],1))

# YOUR CODE ENDS HERE

assert(parameters['W' + str(l)].shape == (layer\_dims[l], layer\_dims[l - 1]))

assert(parameters['b' + str(l)].shape == (layer\_dims[l], 1))

return parameters

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

"""

Implement the linear part of a layer's forward propagation.

Arguments:

A -- activations from previous layer (or input data): (size of previous layer, number of examples)

W -- weights matrix: numpy array of shape (size of current layer, size of previous layer)

b -- bias vector, numpy array of shape (size of the current layer, 1)

Returns:

Z -- the input of the activation function, also called pre-activation parameter

cache -- a python tuple containing "A", "W" and "b" ; stored for computing the backward pass efficiently

"""

#(≈ 1 line of code)

# Z = ...

# YOUR CODE STARTS HERE

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

# YOUR CODE ENDS HERE

cache = (A, W, b)

return Z, cache

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

"""

Implement the forward propagation for the LINEAR->ACTIVATION layer

Arguments:

A\_prev -- activations from previous layer (or input data): (size of previous layer, number of examples)

W -- weights matrix: numpy array of shape (size of current layer, size of previous layer)

b -- bias vector, numpy array of shape (size of the current layer, 1)

activation -- the activation to be used in this layer, stored as a text string: "sigmoid" or "relu"

Returns:

A -- the output of the activation function, also called the post-activation value

cache -- a python tuple containing "linear\_cache" and "activation\_cache";

stored for computing the backward pass efficiently

"""

if activation == "sigmoid":

#(≈ 2 lines of code)

# Z, linear\_cache = ...

# A, activation\_cache = ...

# YOUR CODE STARTS HERE

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

A,activation\_cache=sigmoid(Z)

# YOUR CODE ENDS HERE

elif activation == "relu":

#(≈ 2 lines of code)

# Z, linear\_cache = ...

# A, activation\_cache = ...

# YOUR CODE STARTS HERE

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

A,activation\_cache=relu(Z)

# YOUR CODE ENDS HERE

cache = (linear\_cache, activation\_cache)

return A, cache

def L\_model\_forward(X, parameters):

"""

Implement forward propagation for the [LINEAR->RELU]\*(L-1)->LINEAR->SIGMOID computation

Arguments:

X -- data, numpy array of shape (input size, number of examples)

parameters -- output of initialize\_parameters\_deep()

Returns:

AL -- activation value from the output (last) layer

caches -- list of caches containing:

every cache of linear\_activation\_forward() (there are L of them, indexed from 0 to L-1)

"""

caches = []

A = X

L = len(parameters) // 2 # number of layers in the neural network

# Implement [LINEAR -> RELU]\*(L-1). Add "cache" to the "caches" list.

# The for loop starts at 1 because layer 0 is the input

for l in range(1, L):

A\_prev = A

#(≈ 2 lines of code)

# A, cache = ...

# caches ...

# YOUR CODE STARTS HERE

A,cache=linear\_activation\_forward(A\_prev,parameters['W'+str(l)],parameters['b'+str(l)],"relu")

caches=caches+[cache]

# YOUR CODE ENDS HERE

# Implement LINEAR -> SIGMOID. Add "cache" to the "caches" list.

#(≈ 2 lines of code)

# AL, cache = ...

# caches ...

# YOUR CODE STARTS HERE

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

caches=caches+[cache]

# YOUR CODE ENDS HERE

return AL, caches

def compute\_cost(AL, Y):

"""

Implement the cost function defined by equation (7).

Arguments:

AL -- probability vector corresponding to your label predictions, shape (1, number of examples)

Y -- true "label" vector (for example: containing 0 if non-cat, 1 if cat), shape (1, number of examples)

Returns:

cost -- cross-entropy cost

"""

m = Y.shape[1]

# Compute loss from aL and y.

# (≈ 1 lines of code)

# cost = ...

# YOUR CODE STARTS HERE

cost=-(np.sum(np.dot(Y,np.log(AL).T))+np.sum(np.dot(1-Y,np.log(1-AL).T)))/m

# YOUR CODE ENDS HERE

cost = np.squeeze(cost) # To make sure your cost's shape is what we expect (e.g. this turns [[17]] into 17).

return cost

def linear\_backward(dZ, cache):

"""

Implement the linear portion of backward propagation for a single layer (layer l)

Arguments:

dZ -- Gradient of the cost with respect to the linear output (of current layer l)

cache -- tuple of values (A\_prev, W, b) coming from the forward propagation in the current layer

Returns:

dA\_prev -- Gradient of the cost with respect to the activation (of the previous layer l-1), same shape as A\_prev

dW -- Gradient of the cost with respect to W (current layer l), same shape as W

db -- Gradient of the cost with respect to b (current layer l), same shape as b

"""

A\_prev, W, b = cache

m = A\_prev.shape[1]

### START CODE HERE ### (≈ 3 lines of code)

# dW = ...

# db = ... sum by the rows of dZ with keepdims=True

# dA\_prev = ...

# YOUR CODE STARTS HERE

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

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

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

# YOUR CODE ENDS HERE

return dA\_prev, dW, db

def linear\_activation\_backward(dA, cache, activation):

"""

Implement the backward propagation for the LINEAR->ACTIVATION layer.

Arguments:

dA -- post-activation gradient for current layer l

cache -- tuple of values (linear\_cache, activation\_cache) we store for computing backward propagation efficiently

activation -- the activation to be used in this layer, stored as a text string: "sigmoid" or "relu"

Returns:

dA\_prev -- Gradient of the cost with respect to the activation (of the previous layer l-1), same shape as A\_prev

dW -- Gradient of the cost with respect to W (current layer l), same shape as W

db -- Gradient of the cost with respect to b (current layer l), same shape as b

"""

linear\_cache, activation\_cache = cache

if activation == "relu":

#(≈ 2 lines of code)

# dZ = ...

# dA\_prev, dW, db = ...

# YOUR CODE STARTS HERE

dZ=relu\_backward(dA,activation\_cache)

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

# YOUR CODE ENDS HERE

elif activation == "sigmoid":

#(≈ 2 lines of code)

# dZ = ...

# dA\_prev, dW, db = ...

# YOUR CODE STARTS HERE

dZ=sigmoid\_backward(dA,activation\_cache)

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

# YOUR CODE ENDS HERE

return dA\_prev, dW, db

def L\_model\_backward(AL, Y, caches):

"""

Implement the backward propagation for the [LINEAR->RELU] \* (L-1) -> LINEAR -> SIGMOID group

Arguments:

AL -- probability vector, output of the forward propagation (L\_model\_forward())

Y -- true "label" vector (containing 0 if non-cat, 1 if cat)

caches -- list of caches containing:

every cache of linear\_activation\_forward() with "relu" (it's caches[l], for l in range(L-1) i.e l = 0...L-2)

the cache of linear\_activation\_forward() with "sigmoid" (it's caches[L-1])

Returns:

grads -- A dictionary with the gradients

grads["dA" + str(l)] = ...

grads["dW" + str(l)] = ...

grads["db" + str(l)] = ...

"""

grads = {}

L = len(caches) # the number of layers

m = AL.shape[1]

Y = Y.reshape(AL.shape) # after this line, Y is the same shape as AL

# Initializing the backpropagation

#(1 line of code)

# dAL = ...

# YOUR CODE STARTS HERE

dAL=-(np.divide(Y, AL) - np.divide(1 - Y, 1 - AL))

# YOUR CODE ENDS HERE

# Lth layer (SIGMOID -> LINEAR) gradients. Inputs: "dAL, current\_cache". Outputs: "grads["dAL-1"], grads["dWL"], grads["dbL"]

#(approx. 5 lines)

# current\_cache = ...

# dA\_prev\_temp, dW\_temp, db\_temp = ...

# grads["dA" + str(L-1)] = ...

# grads["dW" + str(L)] = ...

# grads["db" + str(L)] = ...

# YOUR CODE STARTS HERE

current\_cache=caches[L-1]

dA\_prev\_temp,dW\_temp,db\_temp=linear\_activation\_backward(dAL,current\_cache,"sigmoid")

grads["dA"+str(L-1)]=dA\_prev\_temp

grads["dW"+str(L)]=dW\_temp

grads["db"+str(L)]=db\_temp

# YOUR CODE ENDS HERE

# Loop from l=L-2 to l=0

for l in reversed(range(L-1)):

# lth layer: (RELU -> LINEAR) gradients.

# Inputs: "grads["dA" + str(l + 1)], current\_cache". Outputs: "grads["dA" + str(l)] , grads["dW" + str(l + 1)] , grads["db" + str(l + 1)]

#(approx. 5 lines)

# current\_cache = ...

# dA\_prev\_temp, dW\_temp, db\_temp = ...

# grads["dA" + str(l)] = ...

# grads["dW" + str(l + 1)] = ...

# grads["db" + str(l + 1)] = ...

# YOUR CODE STARTS HERE

current\_cache=caches[l]

dA\_prev\_temp,dW\_temp,db\_temp=linear\_activation\_backward(grads["dA"+str(l+1)],current\_cache,"relu")

grads["dA"+str(l)]=dA\_prev\_temp

grads["dW"+str(l+1)]=dW\_temp

grads["db"+str(l+1)]=db\_temp

# YOUR CODE ENDS HERE

return grads

def update\_parameters(params, grads, learning\_rate):

"""

Update parameters using gradient descent

Arguments:

params -- python dictionary containing your parameters

grads -- python dictionary containing your gradients, output of L\_model\_backward

Returns:

parameters -- python dictionary containing your updated parameters

parameters["W" + str(l)] = ...

parameters["b" + str(l)] = ...

"""

parameters = params.copy()

L = len(parameters) // 2 # number of layers in the neural network

# Update rule for each parameter. Use a for loop.

#(≈ 2 lines of code)

for l in range(L):

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

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

# YOUR CODE STARTS HERE

parameters["W"+str(l+1)]=parameters["W"+str(l+1)]-learning\_rate\*grads["dW"+str(l+1)]

parameters["b"+str(l+1)]=parameters["b"+str(l+1)]-learning\_rate\*grads["db"+str(l+1)]

# YOUR CODE ENDS HERE

return parameters

**def** initialize\_parameters(n\_x, n\_h, n\_y):

**...**

**return** parameters

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

**...**

**return** A, cache

**def** compute\_cost(AL, Y):

**...**

**return** cost

**def** linear\_activation\_backward(dA, cache, activation):

**...**

**return** dA\_prev, dW, db

**def** update\_parameters(parameters, grads, learning\_rate):

**...**

**return** parameters

### CONSTANTS DEFINING THE MODEL ####

n\_x = 12288 # num\_px \* num\_px \* 3

n\_h = 7

n\_y = 1

layers\_dims = (n\_x, n\_h, n\_y)

learning\_rate = 0.0075

def two\_layer\_model(X, Y, layers\_dims, learning\_rate = 0.0075, num\_iterations = 3000, print\_cost=False):

"""

Implements a two-layer neural network: LINEAR->RELU->LINEAR->SIGMOID.

Arguments:

X -- input data, of shape (n\_x, number of examples)

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

layers\_dims -- dimensions of the layers (n\_x, n\_h, n\_y)

num\_iterations -- number of iterations of the optimization loop

learning\_rate -- learning rate of the gradient descent update rule

print\_cost -- If set to True, this will print the cost every 100 iterations

Returns:

parameters -- a dictionary containing W1, W2, b1, and b2

"""

np.random.seed(1)

grads = {}

costs = [] # to keep track of the cost

m = X.shape[1] # number of examples

(n\_x, n\_h, n\_y) = layers\_dims

# Initialize parameters dictionary, by calling one of the functions you'd previously implemented

#(≈ 1 line of code)

# parameters = ...

# YOUR CODE STARTS HERE

parameters=initialize\_parameters(n\_x,n\_h,n\_y)

# YOUR CODE ENDS HERE

# Get W1, b1, W2 and b2 from the dictionary parameters.

W1 = parameters["W1"]

b1 = parameters["b1"]

W2 = parameters["W2"]

b2 = parameters["b2"]

# Loop (gradient descent)

for i in range(0, num\_iterations):

# Forward propagation: LINEAR -> RELU -> LINEAR -> SIGMOID. Inputs: "X, W1, b1, W2, b2". Output: "A1, cache1, A2, cache2".

#(≈ 2 lines of code)

# A1, cache1 = ...

# A2, cache2 = ...

# YOUR CODE STARTS HERE

A1,cache1=linear\_activation\_forward(X, W1, b1, "relu")

A2,cache2=linear\_activation\_forward(A1, W2, b2, "sigmoid")

# YOUR CODE ENDS HERE

# Compute cost

#(≈ 1 line of code)

# cost = ...

# YOUR CODE STARTS HERE

cost=compute\_cost(A2,Y)

# YOUR CODE ENDS HERE

# Initializing backward propagation

dA2 = - (np.divide(Y, A2) - np.divide(1 - Y, 1 - A2))

# Backward propagation. Inputs: "dA2, cache2, cache1". Outputs: "dA1, dW2, db2; also dA0 (not used), dW1, db1".

#(≈ 2 lines of code)

# dA1, dW2, db2 = ...

# dA0, dW1, db1 = ...

# YOUR CODE STARTS HERE

dA1,dW2,db2=linear\_activation\_backward(dA2, cache2, "sigmoid")

dA0,dW1,db1=linear\_activation\_backward(dA1, cache1, "relu")

# YOUR CODE ENDS HERE

# Set grads['dWl'] to dW1, grads['db1'] to db1, grads['dW2'] to dW2, grads['db2'] to db2

grads['dW1'] = dW1

grads['db1'] = db1

grads['dW2'] = dW2

grads['db2'] = db2

# Update parameters.

#(approx. 1 line of code)

# parameters = ...

# YOUR CODE STARTS HERE

parameters=update\_parameters(parameters, grads, learning\_rate)

# YOUR CODE ENDS HERE

# Retrieve W1, b1, W2, b2 from parameters

W1 = parameters["W1"]

b1 = parameters["b1"]

W2 = parameters["W2"]

b2 = parameters["b2"]

# Print the cost every 100 iterations

if print\_cost and i % 100 == 0 or i == num\_iterations - 1:

print("Cost after iteration {}: {}".format(i, np.squeeze(cost)))

if i % 100 == 0 or i == num\_iterations:

costs.append(cost)

return parameters, costs

def plot\_costs(costs, learning\_rate=0.0075):

plt.plot(np.squeeze(costs))

plt.ylabel('cost')

plt.xlabel('iterations (per hundreds)')

plt.title("Learning rate =" + str(learning\_rate))

plt.show()

**def** initialize\_parameters\_deep(layers\_dims):

**...**

**return** parameters

**def** L\_model\_forward(X, parameters):

**...**

**return** AL, caches

**def** compute\_cost(AL, Y):

**...**

**return** cost

**def** L\_model\_backward(AL, Y, caches):

**...**

**return** grads

**def** update\_parameters(parameters, grads, learning\_rate):

**...**

**return** parameters

layers\_dims = [12288, 20, 7, 5, 1] # 4-layer model

def L\_layer\_model(X, Y, layers\_dims, learning\_rate = 0.0075, num\_iterations = 3000, print\_cost=False):

"""

Implements a L-layer neural network: [LINEAR->RELU]\*(L-1)->LINEAR->SIGMOID.

Arguments:

X -- data, numpy array of shape (num\_px \* num\_px \* 3, number of examples)

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

layers\_dims -- list containing the input size and each layer size, of length (number of layers + 1).

learning\_rate -- learning rate of the gradient descent update rule

num\_iterations -- number of iterations of the optimization loop

print\_cost -- if True, it prints the cost every 100 steps

Returns:

parameters -- parameters learnt by the model. They can then be used to predict.

"""

np.random.seed(1)

costs = [] # keep track of cost

# Parameters initialization.

#(≈ 1 line of code)

# parameters = ...

# YOUR CODE STARTS HERE

parameters=initialize\_parameters\_deep(layers\_dims)

# YOUR CODE ENDS HERE

# Loop (gradient descent)

for i in range(0, num\_iterations):

# Forward propagation: [LINEAR -> RELU]\*(L-1) -> LINEAR -> SIGMOID.

#(≈ 1 line of code)

# AL, caches = ...

# YOUR CODE STARTS HERE

AL,caches=L\_model\_forward(X, parameters)

# YOUR CODE ENDS HERE

# Compute cost.

#(≈ 1 line of code)

# cost = ...

# YOUR CODE STARTS HERE

cost=compute\_cost(AL, Y)

# YOUR CODE ENDS HERE

# Backward propagation.

#(≈ 1 line of code)

# grads = ...

# YOUR CODE STARTS HERE

grads=L\_model\_backward(AL, Y, caches)

# YOUR CODE ENDS HERE

# Update parameters.

#(≈ 1 line of code)

# parameters = ...

# YOUR CODE STARTS HERE

parameters=update\_parameters(parameters, grads, learning\_rate)

# YOUR CODE ENDS HERE

# Print the cost every 100 iterations

if print\_cost and i % 100 == 0 or i == num\_iterations - 1:

print("Cost after iteration {}: {}".format(i, np.squeeze(cost)))

if i % 100 == 0 or i == num\_iterations:

costs.append(cost)

return parameters, costs

parameters, costs = L\_layer\_model(train\_x, train\_y, layers\_dims, num\_iterations = 2500, print\_cost = True)

pred\_train = predict(train\_x, train\_y, parameters)

pred\_test = predict(test\_x, test\_y, parameters)