Practical No.4

import numpy as np

def linear\_forward\_test\_case():

np.random.seed(1)

"""

X = np.array([[-1.02387576, 1.12397796],

[-1.62328545, 0.64667545],

[-1.74314104, -0.59664964]])

W = np.array([[ 0.74505627, 1.97611078, -1.24412333]])

b = np.array([[1]])

"""

A = np.random.randn(3,2)

W = np.random.randn(1,3)

b = np.random.randn(1,1)

return A, W, b

def linear\_activation\_forward\_test\_case():

"""

X = np.array([[-1.02387576, 1.12397796],

[-1.62328545, 0.64667545],

[-1.74314104, -0.59664964]])

W = np.array([[ 0.74505627, 1.97611078, -1.24412333]])

b = 5

"""

np.random.seed(2)

A\_prev = np.random.randn(3,2)

W = np.random.randn(1,3)

b = np.random.randn(1,1)

return A\_prev, W, b

def L\_model\_forward\_test\_case():

"""

X = np.array([[-1.02387576, 1.12397796],

[-1.62328545, 0.64667545],

[-1.74314104, -0.59664964]])

parameters = {'W1': np.array([[ 1.62434536, -0.61175641, -0.52817175],

[-1.07296862, 0.86540763, -2.3015387 ]]),

'W2': np.array([[ 1.74481176, -0.7612069 ]]),

'b1': np.array([[ 0.],

[ 0.]]),

'b2': np.array([[ 0.]])}

"""

np.random.seed(1)

X = np.random.randn(4,2)

W1 = np.random.randn(3,4)

b1 = np.random.randn(3,1)

W2 = np.random.randn(1,3)

b2 = np.random.randn(1,1)

parameters = {"W1": W1,

"b1": b1,

"W2": W2,

"b2": b2}

return X, parameters

def compute\_cost\_test\_case():

Y = np.asarray([[1, 1, 1]])

aL = np.array([[.8,.9,0.4]])

return Y, aL

def linear\_backward\_test\_case():

"""

z, linear\_cache = (np.array([[-0.8019545 , 3.85763489]]), (np.array([[-1.02387576, 1.12397796],

[-1.62328545, 0.64667545],

[-1.74314104, -0.59664964]]), np.array([[ 0.74505627, 1.97611078, -1.24412333]]), np.array([[1]]))

"""

np.random.seed(1)

dZ = np.random.randn(1,2)

A = np.random.randn(3,2)

W = np.random.randn(1,3)

b = np.random.randn(1,1)

linear\_cache = (A, W, b)

return dZ, linear\_cache

def linear\_activation\_backward\_test\_case():

"""

aL, linear\_activation\_cache = (np.array([[ 3.1980455 , 7.85763489]]), ((np.array([[-1.02387576, 1.12397796], [-1.62328545, 0.64667545], [-1.74314104, -0.59664964]]), np.array([[ 0.74505627, 1.97611078, -1.24412333]]), 5), np.array([[ 3.1980455 , 7.85763489]])))

"""

np.random.seed(2)

dA = np.random.randn(1,2)

A = np.random.randn(3,2)

W = np.random.randn(1,3)

b = np.random.randn(1,1)

Z = np.random.randn(1,2)

linear\_cache = (A, W, b)

activation\_cache = Z

linear\_activation\_cache = (linear\_cache, activation\_cache)

return dA, linear\_activation\_cache

def L\_model\_backward\_test\_case():

"""

X = np.random.rand(3,2)

Y = np.array([[1, 1]])

parameters = {'W1': np.array([[ 1.78862847, 0.43650985, 0.09649747]]), 'b1': np.array([[ 0.]])}

aL, caches = (np.array([[ 0.60298372, 0.87182628]]), [((np.array([[ 0.20445225, 0.87811744],

[ 0.02738759, 0.67046751],

[ 0.4173048 , 0.55868983]]),

np.array([[ 1.78862847, 0.43650985, 0.09649747]]),

np.array([[ 0.]])),

np.array([[ 0.41791293, 1.91720367]]))])

"""

np.random.seed(3)

AL = np.random.randn(1, 2)

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

A1 = np.random.randn(4,2)

W1 = np.random.randn(3,4)

b1 = np.random.randn(3,1)

Z1 = np.random.randn(3,2)

linear\_cache\_activation\_1 = ((A1, W1, b1), Z1)

A2 = np.random.randn(3,2)

W2 = np.random.randn(1,3)

b2 = np.random.randn(1,1)

Z2 = np.random.randn(1,2)

linear\_cache\_activation\_2 = ( (A2, W2, b2), Z2)

caches = (linear\_cache\_activation\_1, linear\_cache\_activation\_2)

return AL, Y, caches

def update\_parameters\_test\_case():

"""

parameters = {'W1': np.array([[ 1.78862847, 0.43650985, 0.09649747],

[-1.8634927 , -0.2773882 , -0.35475898],

[-0.08274148, -0.62700068, -0.04381817],

[-0.47721803, -1.31386475, 0.88462238]]),

'W2': np.array([[ 0.88131804, 1.70957306, 0.05003364, -0.40467741],

[-0.54535995, -1.54647732, 0.98236743, -1.10106763],

[-1.18504653, -0.2056499 , 1.48614836, 0.23671627]]),

'W3': np.array([[-1.02378514, -0.7129932 , 0.62524497],

[-0.16051336, -0.76883635, -0.23003072]]),

'b1': np.array([[ 0.],

[ 0.],

[ 0.],

[ 0.]]),

'b2': np.array([[ 0.],

[ 0.],

[ 0.]]),

'b3': np.array([[ 0.],

[ 0.]])}

grads = {'dW1': np.array([[ 0.63070583, 0.66482653, 0.18308507],

[ 0. , 0. , 0. ],

[ 0. , 0. , 0. ],

[ 0. , 0. , 0. ]]),

'dW2': np.array([[ 1.62934255, 0. , 0. , 0. ],

[ 0. , 0. , 0. , 0. ],

[ 0. , 0. , 0. , 0. ]]),

'dW3': np.array([[-1.40260776, 0. , 0. ]]),

'da1': np.array([[ 0.70760786, 0.65063504],

[ 0.17268975, 0.15878569],

[ 0.03817582, 0.03510211]]),

'da2': np.array([[ 0.39561478, 0.36376198],

[ 0.7674101 , 0.70562233],

[ 0.0224596 , 0.02065127],

[-0.18165561, -0.16702967]]),

'da3': np.array([[ 0.44888991, 0.41274769],

[ 0.31261975, 0.28744927],

[-0.27414557, -0.25207283]]),

'db1': 0.75937676204411464,

'db2': 0.86163759922811056,

'db3': -0.84161956022334572}

"""

np.random.seed(2)

W1 = np.random.randn(3,4)

b1 = np.random.randn(3,1)

W2 = np.random.randn(1,3)

b2 = np.random.randn(1,1)

parameters = {"W1": W1,

"b1": b1,

"W2": W2,

"b2": b2}

np.random.seed(3)

dW1 = np.random.randn(3,4)

db1 = np.random.randn(3,1)

dW2 = np.random.randn(1,3)

db2 = np.random.randn(1,1)

grads = {"dW1": dW1,

"db1": db1,

"dW2": dW2,

"db2": db2}

return parameters, grads

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)

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

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

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

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

parameters = {"W1": W1,

"b1": b1,

"W2": W2,

"b2": b2}

return parameters

parameters = initialize\_parameters(3,2,1)

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

print("W1 = " + str(parameters["W1"]))

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

print("b1 = " + str(parameters["b1"]))

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

print("W2 = " + str(parameters["W2"]))

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

print("b2 = " + str(parameters["b2"]))

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):

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))

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

parameters = initialize\_parameters\_deep([5,4,3])

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

print("W1 = " + str(parameters["W1"]))

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

print("b1 = " + str(parameters["b1"]))

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

print("W2 = " + str(parameters["W2"]))

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

print("b2 = " + str(parameters["b2"]))

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

"""

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

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":

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

A, activation\_cache =sigmoid(Z)

elif activation == "relu":

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

A, activation\_cache =relu(Z)

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

for l in range(1, L):

A\_prev = A

A, cache =linear\_activation\_forward(A\_prev,

parameters['W'+str(l)],

parameters['b'+str(l)],

activation='relu')

caches.append(cache)

AL, cache =linear\_activation\_forward(A,

parameters['W'+str(L)],

parameters['b'+str(L)],

activation='sigmoid')

caches.append(cache)

#assert(AL.shape == (1, X.shape[1]))

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]

cost = (-1 / m) \* np.sum(np.multiply(Y, np.log(AL)) + np.multiply(1 - Y, np.log(1 - AL)))

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

return cost

A = np.array([[1, 2], [3, 4]])

print('axis=1 and keepdims=True')

print(np.sum(A, axis=1, keepdims=True))

print('axis=1 and keepdims=False')

print(np.sum(A, axis=1, keepdims=False))

print('axis=0 and keepdims=True')

print(np.sum(A, axis=0, keepdims=True))

print('axis=0 and keepdims=False')

print(np.sum(A, axis=0, keepdims=False))

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]

dW = np.dot(dZ, cache[0].T) / m

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

dA\_prev = np.dot(cache[1].T, dZ)

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":

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

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

elif activation == "sigmoid":

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

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

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

print("L = "+str(L))

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

current\_cache = caches[L-1]

grads["dA" + str(L-1)], grads["dW" + str(L)], grads["db" + str(L)] = linear\_activation\_backward(dAL, current\_cache, activation = "sigmoid")

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

print("dW"+ str(L)+" = "+str(grads["dW" + str(L)]))

print("db"+ str(L)+" = "+str(grads["db" + str(L)]))

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

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

current\_cache = caches[l]

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

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

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

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

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.

for l in range(L):

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)]

return parameters

parameters, grads = update\_parameters\_test\_case()

parameters = update\_parameters(parameters, grads, 0.1)

print ("W1 = "+ str(parameters["W1"]))

print ("b1 = "+ str(parameters["b1"]))

print ("W2 = "+ str(parameters["W2"]))

print ("b2 = "+ str(parameters["b2"]))

Output:-

W1 = [[-0.59562069 -0.09991781 -2.14584584 1.82662008]

[-1.76569676 -0.80627147 0.51115557 -1.18258802]

[-1.0535704 -0.86128581 0.68284052 2.20374577]]

b1 = [[-0.04659241]

[-1.28888275]

[ 0.53405496]]

W2 = [[-0.55569196 0.0354055 1.32964895]]

b2 = [[-0.84610769]]