Building your Deep Neural Network: Step by Step

Welcome to your week 4 programming report. In this, this exercise, I will walk you through the steps in a deep neural network. This week, you will build a deep neural network, with as many layers you want!

In this exercise, you will implement all the functions required to build a deep neural network.
In this week assignment, you will use these functions to build a deep neural network for image classification.

After this assignment you will be able to:

- Make matrix and vector dimensions fit properly in a deep neural network.
- Perform vectorization of deep learning models.
- Understand how forward and backward propagation work in deep learning models.
- Implement a deep neural network for image classification.

Let's get started!

1 - Packages

Let's first import all the packages you'll need during this assignment.

- numpy is the main package for scientific computing with Python.
- matplotlib is a library to plot graphs in Python.
- h5py is a common package to interact with a dataset that is stored on an H5 file.
- lr_utils provides the functions we used in week 2 programming assignment to load the dataset into your environment.
- dnn_utils provides some necessary functions for this notebook.
- testCases provides some test examples to assess the correctness of your functions

```
# Package imports
import numpy as np
import matplotlib.pyplot as plt
import h5py
import scipy
from PIL import Image
from scipy import ndimage
from lr_utils import load_dataset
from dnn_utils import sigmoid, sigmoid_backward, relu, relu_backward

%matplotlib inline
plt.rcParams['figure.figure.size'] = (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)
```

Out[1]:

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

2 - Outline of the Assignment

To build your deep neural network, you will be implementing several "helper functions". These helper functions will be used in the next assignment to build a two-layer neural network and an L-layer neural network. Each small helper function you will implement will have detailed instructions that will tell you what exactly it needs to do, and where to use it in the larger model of your deep neural network. Each of these helper functions will also help you organize the flow of your deep neural network.

It occurs to us that the best way to build a deep neural network is to think about it as building blocks:

- Initialize the model's parameters
- Loop for num_iterations:
  - Forward propagation module (compute all intermediate activations, in the order from left to right)
  - Compute cost function
  - Backward propagation module (compute all gradients, in the reverse order from right to left)
  - Update parameters (using parameters, and grads from backprop)
- Use trained parameters to predict labels

Given the model architecture, you will be asked to implement the functions in the order they appear in this assignment. Each section of this assignment will ask you to fill in the relevant function, and then at the end you will be able to put all the functions together to build a two-layer neural network and an L-layer neural network. Finally, you will use these functions to build a deep neural network for image classification.

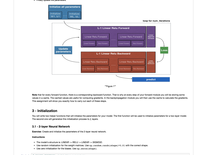

Figure 1

Note For the Deep Neural Network, there are corresponding backward functions. We will also store A, W and b in a variable we call the "cache" (A is the activation output of the previous layer, and W and b are the parameters of the current layer). The cache will be used for some computations during the corresponding backward function.

This assignment will ask you to specially fixed for a vector) and use of Python broadcasting.

3 - Initialization

You will start your deep neural network then the two-layer neural network, then the L-layer neural network. Each small helper function you will implement for initializing parameters for a two-layer model and for the L-layer model.

3.1 - 2-layer Neural Network

Exercise: Create and initialize the parameters of the 2-layer neural network.

Instructions:

- The model's structure is: LINEAR -> RELU -> LINEAR -> SIGMOID.
- You will use random initialization for the matrices. You will use np.random.randn(shape)*0.01 to randomize your matrix.
- You will use zero initialization for the biases. Use np.zeros(shape).

```
# GRADED FUNCTION: initialize_parameters_deep

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 = 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))
    ...

    assert(W1.shape == (n_h, n_x))
    assert(b1.shape == (n_h, 1))
    assert(W2.shape == (n_y, n_h))
    assert(b2.shape == (n_y, 1))

    parameters = {"W1": W1,
                  "b1": b1,
                  "W2": W2,
                  "b2": b2}

    return parameters
```

```
parameters = initialize_parameters(2,2,2)
print("W1 = " + str(parameters["W1"]))
print("b1 = " + str(parameters["b1"]))
print("W2 = " + str(parameters["W2"]))
print("b2 = " + str(parameters["b2"]))
```

Expected output:

W1	array([[0.01624345, -0.00611756, -0.00528172], [-0.01072969, 0.00865408, -0.02301539]])
b1	array([[0.], [0.]])
W2	array([[0.01744812, -0.00761207]])
b2	array([[0.]]

3.2 - L-layer Neural Network

The initialization for a deep learning model is more complicated because there are more weight matrices and bias vectors. When initializing the parameters, make sure your dimensions match between each layer. Recall that $n^{[l]}$ is the number of units in layer l. Thus for example if the size of our input X is $(12288, 209)$ (with $m=209$ examples) then:

$$\begin{array}{l} \text{Shape of } W^{[l]} = (n^{[l]}, n^{[l-1]}) \\ \text{Shape of } b^{[l]} = (n^{[l]}, 1) \\ \text{Shape of } W^{[l]}X + b^{[l]} = (n^{[l]}, 209) \end{array} \qquad \begin{array}{l} \text{Shape of } W^{[l]} = (n^{[l]}, n^{[l-1]}) \\ \text{Shape of } b^{[l]} = (n^{[l]}, 1) \\ \text{Shape of } Z^{[l]} = (n^{[l]}, 209) \end{array}$$

Remember that when we compute $W^{[l]}X + b^{[l]}$ there is a broadcasting, for example:

$$W^{[l]}X + b^{[l]} = \begin{bmatrix} w^{[l]}_{11} X_{11} + w^{[l]}_{12} X_{21} + \cdots + w^{[l]}_{1n^{[l-1]}} X_{n^{[l-1]}1} + b^{[l]}_1 & \cdots & w^{[l]}_{11} X_{11} + w^{[l]}_{12} X_{21} + \cdots + w^{[l]}_{1n^{[l-1]}} X_{n^{[l-1]}m} + b^{[l]}_1 \\ \vdots & \ddots & \vdots \\ w^{[l]}_{n^{[l]}1} X_{11} + w^{[l]}_{n^{[l]}2} X_{21} + \cdots + w^{[l]}_{n^{[l]}n^{[l-1]}} X_{n^{[l-1]}1} + b^{[l]}_{n^{[l]}} & \cdots & w^{[l]}_{n^{[l]}1} X_{11} + w^{[l]}_{n^{[l]}2} X_{21} + \cdots + w^{[l]}_{n^{[l]}n^{[l-1]}} X_{n^{[l-1]}m} + b^{[l]}_{n^{[l]}} \end{bmatrix} \tag{3}$$

Then the Z is of shape

$$Z^{[l]} = \begin{bmatrix} z^{[l]}_{11} & \cdots & z^{[l]}_{1m} \\ \vdots & \ddots & \vdots \\ z^{[l]}_{n^{[l]}1} & \cdots & z^{[l]}_{n^{[l]}m} \end{bmatrix} = W^{[l]}A^{[l-1]} + b^{[l]} \tag{4}$$

Exercise: Implement initialization for a L-layer Neural Network.

Instructions:

- The model's structure is [LINEAR -> RELU] * (L-1) -> LINEAR -> SIGMOID. I.e. it is a cache of ((number of layers - 1) times [LINEAR -> RELU]) plus [LINEAR -> SIGMOID].
- Use random initialization for the matrices. Use np.random.randn(shape) * 0.01.
- Use zero initialization for the biases. Use np.zeros(shape).
- Similar to the previous function you will need to initialize the parameters's values in a python dictionary.
- Now you will need to initialize parameters for the deep layers. Therefore, for layers 1 through L, you will need to initialize W1, b1, W2, b2, ..., WL, bL. For example: parameter["W1"] = np.random.randn(layer_dims[1], layer_dims[0]) * 0.01.

```
# GRADED FUNCTION: initialize_parameters_deep

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)] = np.random.randn(layer_dims[l], layer_dims[l-1])*0.01
        parameters['b' + str(l)] = np.zeros(shape=(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("W1 = " + str(parameters["W1"]))
print("b1 = " + str(parameters["b1"]))
print("W2 = " + str(parameters["W2"]))
print("b2 = " + str(parameters["b2"]))
```

Expected output:

W1	array([[0.01788628, 0.00633482, -0.00908678, -0.00162425, -0.01618726], [-0.00268392, -0.00374881, 0.01133383, -0.00744686, -0.00635909], [-0.00797498, 0.00965884, 0.00187811, -0.00069728, 0.00443499], [0.01354616, 0.01773744, 0.00828828, 0.00203523, -0.01392575], [-0.00354759, -0.00768836, -0.00230616, 0.00099649, -0.00240026]])
b1	array([[0.], [0.], [0.], [0.], [0.]])
W2	array([[0.00576875, 0.00488301, -0.00445462, -0.00291321, -0.00929942], [0.00974772, 0.00295267, 0.00395509, 0.00962211, 0.00964822], [-0.00781201, 0.01067338, -0.00368393, -0.00132537, 0.00788732], [-0.00748758, 0.00543359, 0.00975366, -0.00890648, 0.00979083], [-0.00546037, -0.00551364, -0.00167772, 0.00333779, 0.00771286]])
b2	array([[0.], [0.], [0.], [0.], [0.]])

4 - Forward propagation module

4.1 - Linear Forward

Now that you have initialized your parameters, you will do the forward propagation module. You will start by implementing some basic functions that you will use later when implementing the model. You will now implement the following functions:

- LINEAR
- LINEAR -> ACTIVATION where ACTIVATION will be either ReLU or Sigmoid.
- LINEAR -> RELU -> LINEAR -> SIGMOID (whole model)

The linear forward module (vectorized over all the examples) computes the following equations:

$$Z^{[l]} = W^{[l]}A^{[l-1]} +b^{[l]} \tag{4}$$

where $A^{[0]} = X$.

Exercise: Build the linear part of forward propagation.

Reminder: The mathematical representation of this unit is $Z^{[l]} = W^{[l]}A^{[l-1]} +b^{[l]}$. You may find np.dot(A, W) and np.sum(axis=1, keepdims=True) useful.

```
# GRADED FUNCTION: linear_forward

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 tuple containing "A", "W" and "b" ; stored for computing the backward pass efficiently
    ...

    #(≈ 1 line of code)
    Z = np.dot(W,A)+b
    ...

    assert(Z.shape == (W.shape[0], A.shape[1]))
    cache = (A, W, b)

    return Z, cache
```

```
A, W, b = testCases_linear_forward()
print("With shape explanation: ")
print("Z = " + str(linear_forward(A, W, b)[0]))
```

Expected output:

With shape explanation:	
Z =	array([[3.26295337, -1.23429987], [2.26988489, 0.67511163], [1.098753 , -0.12289023]])

Note: A has dot(W, A) + b, W has dot(W, A) + b, and A has dot(W, A) + b. The "broadcasting" is a little different here.

4.2 - Linear Model

Next you will complete two functions that will be used in the future:

- linear_activation_forward(A_prev, W, b, activation)
- linear_activation_forward(A_prev, W, b, activation)

Figure 2 : LINEAR -> RELU -> LINEAR -> SIGMOID (whole model)

Exercise: Implement the forward propagation of the unit layer.

Instructions:: In this exercise, you'll need to implement the activations $A^{[l]} = \sigma(Z^{[l]}) = \sigma(W^{[l]}A^{[l-1]} + b^{[l]})$. Other functions you will use in this function are:

- sigmoid(Z) -> sigmoid(Z), A_prev, cache = linear_forward(A_prev, W, b)
- relu(Z) -> relu(Z), A_prev, cache = linear_forward(A_prev, W, b)

```
# GRADED FUNCTION: linear_activation_forward

def linear_activation_forward(A_prev, W, b, activation):
    ...
    Implement 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 tuple containing "linear_cache" and "activation_cache";
             stored for computing the backward pass efficiently
    ...

    if activation == "sigmoid":
        #(≈ 2 lines of code)
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = sigmoid(Z)
    elif activation == "relu":
        #(≈ 2 lines of code)
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = relu(Z)
    ...

    assert (A shape == (W.shape[0], A_prev.shape[1]))
    cache = (linear_cache, activation_cache)

    return A, cache
```

```
A_prev, W, b = testCases_linear_activation_forward()
print("With sigmoid: A = " + str(linear_activation_forward(A_prev, W, b, activation = "sigmoid")[0]))
print("With ReLU: A = " + str(linear_activation_forward(A_prev, W, b, activation = "relu")[0]))
```

Expected output:

With sigmoid: A =	array([[0.96890023, 0.11013289], [0.99351508, 0.37328607], [0.96797577, 0.12482558]])
With ReLU: A =	array([[3.43896131, 0.35115108], [2.67065448, 0.07799084], [1.10843898, 0.12384321]])

5 - Cost function

Now you can implement forward and backward propagation. You need to compute the cost, because you want to check if your model is actually learning.

Exercise: Compute the cross-entropy cost J, using the following formula:

$$-\frac{1}{m} \sum\limits_{i = 1}^{m} (y^{(i)}\log\left(a^{[L](i)}\right) + (1-y^{(i)})\log\left(1- a^{[L](i)}\right)) \tag{7}$$

```
# GRADED FUNCTION: compute_cost

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 cat, 1 if non-cat), shape (1, number of examples)

    Returns:
    cost -- cross-entropy cost
    ...

    m = Y.shape[1]

    #(≈ 1 line of code)
    cost = -np.sum(np.multiply(np.log(AL),Y) + np.multiply(np.log(1-AL),(1-Y)))/m
    ...

    cost = np.squeeze(cost)
    assert(cost.shape == ())
    ...

    return cost
```

```
Y, AL = testCases_compute_cost();
print("cost = " + str(compute_cost(AL, Y)))
```

Expected Output:

cost =	0.414931599615397

6 - Backward propagation module

Just like with forward propagation, you will implement helper functions for backpropagation. Remember that each backpropagation is used to calculate the gradient of the loss function.

Figure 3 : LINEAR -> RELU -> LINEAR -> SIGMOID (whole model)

The cache stores the intermediate values of the forward propagation module. It will be used to compute the gradients. For example, if you store {Z[1], W[2], b[2]} in the cache, it will be used to compute the gradients for the corresponding backward function.

Here is the linear forward module (vectorized over all the examples) and its equations:

$$Z^{[l]} = W^{[l]}A^{[l-1]} +b^{[l]} \tag{4}$$

where $A^{[0]} = X$.

The linear backward for l (layer l) is calculated using the previously computed derivatives $dZ^{[l]} = \frac{\partial \mathcal{L}}{\partial Z^{[l]}}$ and the cache containing $A^{[l-1]}$, $W^{[l]}$ and $b^{[l]}$.

$$dW^{[l]} = \frac{\partial \mathcal{L}}{\partial W^{[l]}} = \frac{1}{m} dZ^{[l]} A^{[l-1]T} \tag{8}$$

$$db^{[l]} = \frac{\partial \mathcal{L}}{\partial b^{[l]}} = \frac{1}{m} \sum_{i=1}^{m} dZ^{[l](i)} \tag{9}$$

$$dA^{[l-1]} = \frac{\partial \mathcal{L}}{\partial A^{[l-1]}} = W^{[l]T} dZ^{[l]} \tag{10}$$

Exercise: Use the 3 formulas above to implement linear_backward().

```
# GRADED FUNCTION: linear_backward

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]

    #(≈ 1 line of code)
    dW = np.dot(dZ,A_prev.T)/m
    ...

    #(≈ 1 line of code)
    db = np.sum(dZ,axis=1,keepdims=True)/m
    ...

    #(≈ 1 line of code)
    dA_prev = np.dot(W.T,dZ)
    ...

    assert (dA_prev.shape == A_prev.shape)
    assert (dW.shape == W.shape)
    assert (db.shape == b.shape)

    return dA_prev, dW, db
```

```
dZ, linear_cache = testCases_linear_backward();
print("dA_prev = " + str(linear_backward(dZ, linear_cache)[0]))
print("dW = " + str(linear_backward(dZ, linear_cache)[1]))
print("db = " + str(linear_backward(dZ, linear_cache)[2]))
```

Expected Output:

dA_prev =	array([[0.51822968, -0.19517421], [-0.62835779, 0.22265301], [0.07148926, -0.02443055]])
dW =	array([[0.45188968, 0.44513824, 0.37980664]])
db =	array([[-0.54314555]])

6.1 - Linear-Activation Backward

Next, you will create a function that merges the two helper functions: **linear_backward** and the backward step for the activation **linear_activation_backward**.

If $g(.)$ is the activation function, sigmoid_backward and relu_backward compute

$$dZ^{[l]} = dA^{[l]} * g'(Z^{[l]}) \tag{11}$$

where $A^{[l]} = g(Z^{[l]})$, and $Z^{[l]} = W^{[l]}A^{[l-1]} +b^{[l]}$ and you need to compute $dZ^{[l]}$ and $dA^{[l-1]}$. This gives you:

$$dZ^{[l]} = dA^{[l]} * g'(\underbrace{W^{[l]}A^{[l-1]} +b^{[l]}}_{Z^{[l]}}) \tag{12}$$

$$dA^{[l-1]} = W^{[l]T} dZ^{[l]} \tag{13}$$

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

```
# GRADED FUNCTION: linear_activation_backward

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 = relu_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)
    elif activation == "sigmoid":
        #(≈ 2 lines of code)
        dZ = sigmoid_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)
    ...

    return dA_prev, dW, db
```

```
AL, linear_activation_cache = testCases_linear_activation_backward();
print("sigmoid:")
print("dA_prev = " + str(linear_activation_backward(AL, linear_activation_cache, activation = "sigmoid")[0]))
print("dW = " + str(linear_activation_backward(AL, linear_activation_cache, activation = "sigmoid")[1]))
print("db = " + str(linear_activation_backward(AL, linear_activation_cache, activation = "sigmoid")[2]))

print("relu:")
print("dA_prev = " + str(linear_activation_backward(AL, linear_activation_cache, activation = "relu")[0]))
print("dW = " + str(linear_activation_backward(AL, linear_activation_cache, activation = "relu")[1]))
print("db = " + str(linear_activation_backward(AL, linear_activation_cache, activation = "relu")[2]))
```

Expected output:

sigmoid:	
dA_prev =	array([[0.11017994, 0.01105339], [0.09466817, 0.00949723], [-0.05743092, -0.00576154]])
dW =	array([[0.10266786, 0.09778551, -0.01968084]])
db =	array([[-0.05729622]])
relu:	
dA_prev =	array([[0.44090989, 0.43571474], [0.24168393, 0.23878484], [-0.62807042, -0.62076127]])
dW =	array([[0.44513824, 0.37980664, -0.54314555]])
db =	array([[-0.20837892]])

6.2 - L-Model Backward

Now you will implement the backward function for the whole network. Recall that when you implemented the L_model_forward function, at each iteration, you stored a cache which contains (X,W,b, and z). In the back propagation module, you will use those variables to compute the gradients. Therefore, in the L_model_backward function, you will iterate through all the hidden layers backward, starting from layer L. On each step, you will use the cached values for layer l, to backpropagate through layer l. Figure 3 shows the backward propagation.

Figure 3 : Backward cycle

Initialization backpropagation: To backpropagate through this network, we know that the output is, $A^{[L]} = \sigma(Z^{[L]})$. You could find it useful to compute dAL using the formula:

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

You can then use this post-activation gradient dAL to keep going backward. As an output, you can use dA for layer L. Then your output should be the vector of dA's for each layer coming from the cache. To feed in the right activation cache into the right layer, you'll also need to extract the cache for layer L from the cache for all layers. Use this output to update the gradients for the last layer.

For instance, for L = 3 this would look like this: grads["dA3"] = dA3

Exercise: Implement backpropagation for the [LINEAR->RELU] * (L-1) -> LINEAR -> SIGMOID model.

```
# GRADED FUNCTION: L_model_backward

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 (for example: containing 0 if cat, 1 if non-cat), shape (1, number of examples)
    caches -- list of caches containing:
                every cache of linear_activation_forward() with "relu" (it's caches[l], for l in range(0, L-1))
                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

    # Initialization of the backpropagation
    dAL = -(np.divide(Y, AL) - np.divide(1 - Y, 1 - AL))

    # Lastly-layer-backward: SIGMOID -> LINEAR gradients. Inputs: "dAL, current_cache". Outputs: "grads["dAL-1"], grads["dW"], grads["db"]
    #(≈ 2 lines of code)
    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")
    ...

    # 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)]
        #(≈ 5 lines of code)
        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
```

```
AL, Y_assessment, caches = testCases_L_model_backward();
print("dW1 = " + str(L_model_backward(AL, Y_assessment, caches)['dW1']))
print("db1 = " + str(L_model_backward(AL, Y_assessment, caches)['db1']))
print("dA1 = " + str(L_model_backward(AL, Y_assessment, caches)['dA1']))
```

Expected Output:

dW1 =	array([[0.41010002, 0.07807203, 0.13798444, 0.10502167], [0.00000000, 0.00000000, 0.00000000, 0.00000000], [0.05283652, 0.01005865, 0.01777766, 0.01353616]])
db1 =	array([[-0.22007063], [0.], [0.07313866]])
dA1 =	array([[0.12913162, -0.44014127], [-0.14175655, 0.48317296], [0.01663708, -0.05670698]])

7 - Conclusion

Congrats! so implementing all the functions required for building a deep neural network!

It was a lot of work but just your work for the rest of the assignment. The next part of the assignment is easier.

At the end of this assignment you will be able to apply deep learning to a new application.

Let's go!

You will 1 that use these models to classify cat vs non-cat images!