Building your Deep Neural Network: Step by Step

Welcome to your week 4 assignment (part 1 of 2)! Previously you trained a 2-layer Neural Network with a single hidden layer. This week, you will build a deep neural network with as many layers as you want!

- In this notebook, you'll implement all the functions required to build a deep neural network.
- For the next assignment, you'll use these functions to build a deep neural network for image classification.

By the end of this assignment, you'll be able to:

- Use non-linear units like ReLU to improve your model
- Build a deeper neural network (with more than 1 hidden layer)
- Implement an easy-to-use neural network class

Notation:

- Superscript $[l]$ denotes a quantity associated with the l^{th} layer.
  - Example: $a^{[L]}$ is the L^{th} layer activation. $W^{[L]}$ and $b^{[L]}$ are the L^{th} layer parameters.
- Superscript (i) denotes a quantity associated with the i^{th} example.
  - Example: $x^{(i)}$ is the i^{th} training example.
- Lowerscript i denotes the i^{th} entry of a vector.
  - Example: $a^{[l]}_i$ denotes the i^{th} entry of the l^{th} layer's activations).

Let's get started!

Important Note on Submission to the AutoGrader

Before submitting your assignment to the AutoGrader, please make sure you are not doing the following:

1. You have not added any *extra* `print` statement(s) in the assignment.
2. You have not added any *extra* code cell(s) in the assignment.
3. You have not changed any of the function parameters.
4. You are not using any global variables inside your graded exercises. Unless specifically instructed to do so, please refrain from it and use the local variables instead.
5. You are not changing the assignment code where it is not required, like creating *extra* variables.

If you do any of the following, you will get something like, `Grader Error: Grader feedback not found` (or similarly unexpected) error upon submitting your assignment. Before asking for help/debugging the errors in your assignment, check for these first. If this is the case, and you don't remember the changes you have made, you can get a fresh copy of the assignment by following these [instructions (https://www.coursera.org/learn/neural-networks-deep-learning/supplement/iLwon/h-ow-to-refresh-your-workspace)](https://www.coursera.org/learn/neural-networks-deep-learning/supplement/iLwon/h-ow-to-refresh-your-workspace).

Table of Contents

1 - Packages

First, import all the packages you'll need during this assignment.

- numpy (www.numpy.org) is the main package for scientific computing with Python.
- matplotlib (http://matplotlib.org) is a library to plot graphs in Python.
- dnn_utils provides some necessary functions for this notebook.
- testCases provides some test cases to assess the correctness of your functions
- np.random.seed(1) is used to keep all the random function calls consistent. It helps grade your work. Please don't change the seed!

```
In [ ]: ### v1.2
```

In [ ]:
```
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 *

import copy
%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)
```

2 - Outline

To build your neural network, you'll 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 will have detailed instructions to walk you through the necessary steps. Here's an outline of the steps in this assignment:

- Initialize the parameters for a two-layer network and for an L-layer neural network
- Implement the forward propagation module (shown in purple in the figure below)
    - Complete the LINEAR part of a layer's forward propagation step (resulting in $Z^{[l]}$).
    - The ACTIVATION function is provided for you (relu/sigmoid)
    - Combine the previous two steps into a new [LINEAR->ACTIVATION] forward function.
    - Stack the [LINEAR->RELU] forward function L-1 time (for layers 1 through L-1) and add a [LINEAR->SIGMOID] at the end (for the final layer L). This gives you a new L_model_forward function.
- Compute the loss
- Implement the backward propagation module (denoted in red in the figure below)
    - Complete the LINEAR part of a layer's backward propagation step
    - The gradient of the ACTIVATION function is provided for you(relu_backward/sigmoid_backward)
    - Combine the previous two steps into a new [LINEAR->ACTIVATION] backward function
    - Stack [LINEAR->RELU] backward L-1 times and add [LINEAR->SIGMOID] backward in a new L_model_backward function
- Finally, update the parameters

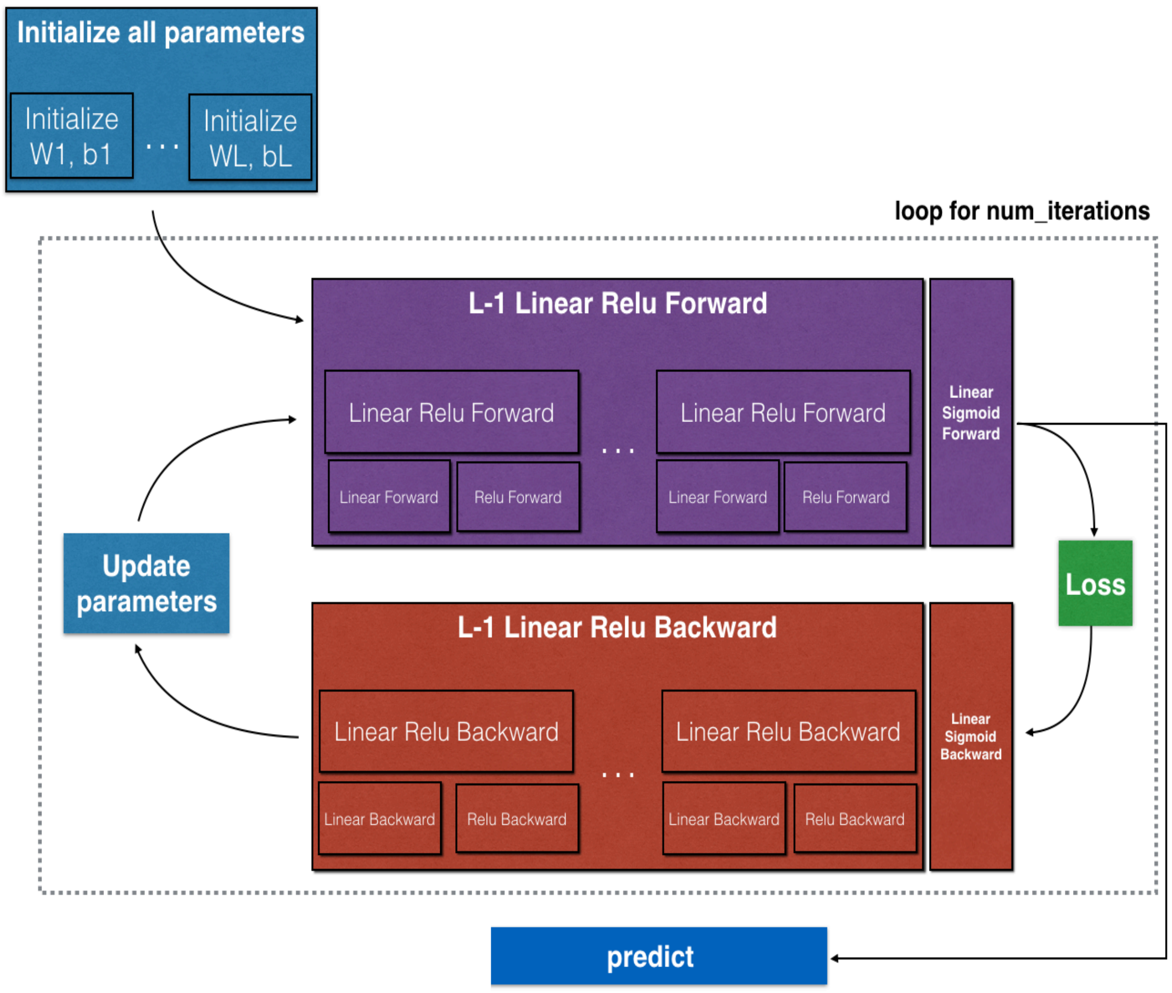

Figure 1

3 - Initialization

You will write two helper functions to initialize the parameters for your model. The first function will be used to initialize parameters for a two layer model. The second one generalizes this initialization process to L layers.

3.1 - 2-layer Neural Network

Exercise 1 - initialize_parameters

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

Instructions:

- The model's structure is: *LINEAR -> RELU -> LINEAR -> SIGMOID*.
- Use this random initialization for the weight matrices: `np.random.randn(d0, d1, ..., dn) * 0.01` with the correct shape. The documentation for np.random.randn (https://numpy.org/doc/stable/reference/random/generated/numpy.random.randn.html)
- Use zero initialization for the biases: `np.zeros(shape)` . The documentation for np.zeros (https://numpy.org/doc/stable/reference/generated/numpy.zeros.html)

In [2]:
```
import numpy as np

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)  # Ensures reproducibility

    # Initialize W1, W2 with random values scaled by 0.01
    W1 = np.random.randn(n_h, n_x) * 0.01
    W2 = np.random.randn(n_y, n_h) * 0.01

    # Initialize b1, b2 with zeros
    b1 = np.zeros((n_h, 1))
    b2 = np.zeros((n_y, 1))

    # Bundle all parameters into a dictionary
    parameters = {
        "W1": W1,
        "b1": b1,
        "W2": W2,
        "b2": b2
    }

    return parameters

# Test Case 1
print("Test Case 1:\n")
parameters = initialize_parameters(3, 2, 1)

print("W1 = " + str(parameters["W1"]))
print("b1 = " + str(parameters["b1"]))
print("W2 = " + str(parameters["W2"]))
print("b2 = " + str(parameters["b2"]))

# Check the shapes of the parameters
print("\nShapes of Parameters:")
print("Shape of W1:", parameters["W1"].shape)
print("Shape of b1:", parameters["b1"].shape)
print("Shape of W2:", parameters["W2"].shape)
print("Shape of b2:", parameters["b2"].shape)

# Test Case 2
print("\nTest Case 2:\n")
parameters = initialize_parameters(4, 3, 2)

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

```
# Check the shapes of the parameters
print("\nShapes of Parameters:")
print("Shape of W1:", parameters["W1"].shape)
print("Shape of b1:", parameters["b1"].shape)
print("Shape of W2:", parameters["W2"].shape)
print("Shape of b2:", parameters["b2"].shape)
```

```
Test Case 1:

W1 = [[ 0.01624345 -0.00611756 -0.00528172]
 [-0.01072969  0.00865408 -0.02301539]]
b1 = [[0.]
 [0.]]
W2 = [[ 0.01744812 -0.00761207]]
b2 = [[0.]]

Shapes of Parameters:
Shape of W1: (2, 3)
Shape of b1: (2, 1)
Shape of W2: (1, 2)
Shape of b2: (1, 1)

Test Case 2:

W1 = [[ 0.01624345 -0.00611756 -0.00528172 -0.01072969]
 [ 0.00865408 -0.02301539  0.01744812 -0.00761207]
 [ 0.00319039 -0.0024937   0.01462108 -0.02060141]]
b1 = [[0.]
 [0.]
 [0.]]
W2 = [[-0.00322417 -0.00384054  0.01133769]
 [-0.01099891 -0.00172428 -0.00877858]]
b2 = [[0.]
 [0.]]

Shapes of Parameters:
Shape of W1: (3, 4)
Shape of b1: (3, 1)
Shape of W2: (2, 3)
Shape of b2: (2, 1)
```

In [ ]:
```
print("Test Case 1:\n")
parameters = initialize_parameters(3,2,1)

print("W1 = " + str(parameters["W1"]))
print("b1 = " + str(parameters["b1"]))
print("W2 = " + str(parameters["W2"]))
print("b2 = " + str(parameters["b2"]))

initialize_parameters_test_1(initialize_parameters)

print("\033[90m\nTest Case 2:\n")
parameters = initialize_parameters(4,3,2)

print("W1 = " + str(parameters["W1"]))
print("b1 = " + str(parameters["b1"]))
print("W2 = " + str(parameters["W2"]))
print("b2 = " + str(parameters["b2"]))

initialize_parameters_test_2(initialize_parameters)
```

Expected output

```
Test Case 1:

W1 = [[ 0.01624345 -0.00611756 -0.00528172]
 [-0.01072969  0.00865408 -0.02301539]]
b1 = [[0.]
 [0.]]
W2 = [[ 0.01744812 -0.00761207]]
b2 = [[0.]]
 All tests passed.

Test Case 2:

W1 = [[ 0.01624345 -0.00611756 -0.00528172 -0.01072969]
 [ 0.00865408 -0.02301539  0.01744812 -0.00761207]
 [ 0.00319039 -0.0024937   0.01462108 -0.02060141]]
b1 = [[0.]
 [0.]
 [0.]]
W2 = [[-0.00322417 -0.00384054  0.01133769]
 [-0.01099891 -0.00172428 -0.00877858]]
b2 = [[0.]
 [0.]]
 All tests passed.
```

3.2 - L-layer Neural Network

The initialization for a deeper L-layer neural network is more complicated because there are many more weight matrices and bias vectors. When completing the `initialize_parameters_deep` function, you should make sure that your dimensions match between each layer. Recall that $n^{[l]}$ is the number of units in layer l. For example, if the size of your input X is $(12288, 209)$ (with $m = 209$ examples) then:

	Shape of W	**Shape of b**	**Activation**	**Shape of Activation**
Layer 1	$(n^{[1]}, 12288)$	$(n^{[1]}, 1)$	$Z^{[1]} = W^{[1]}X + b^{[1]}$	$(n^{[1]}, 209)$
Layer 2	$(n^{[2]}, n^{[1]})$	$(n^{[2]}, 1)$	$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$	$(n^{[2]}, 209)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
Layer L-1	$(n^{[L-1]}, n^{[L-2]})$	$(n^{[L-1]}, 1)$	$Z^{[L-1]} = W^{[L-1]}A^{[L-2]} + b^{[L-1]}$	$(n^{[L-1]}, 209)$
Layer L	$(n^{[L]}, n^{[L-1]})$	$(n^{[L]}, 1)$	$Z^{[L]} = W^{[L]}A^{[L-1]} + b^{[L]}$	$(n^{[L]}, 209)$

Remember that when you compute $WX + b$ in python, it carries out broadcasting. For example, if:

$$W = \begin{bmatrix} w_{00} & w_{01} & w_{02} \\ w_{10} & w_{11} & w_{12} \\ w_{20} & w_{21} & w_{22} \end{bmatrix} \;\;\; X = \begin{bmatrix} x_{00} & x_{01} & x_{02} \\ x_{10} & x_{11} & x_{12} \\ x_{20} & x_{21} & x_{22} \end{bmatrix} \;\;\; b = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} \tag{2}$$

Then $WX + b$ will be:

$$WX + b = \begin{bmatrix} (w_{00}x_{00} + w_{01}x_{10} + w_{02}x_{20}) + b_0 & (w_{00}x_{01} + w_{01}x_{11} + w_{02}x_{21}) + \\ (w_{10}x_{00} + w_{11}x_{10} + w_{12}x_{20}) + b_1 & (w_{10}x_{01} + w_{11}x_{11} + w_{12}x_{21}) + \end{bmatrix}$$

Exercise 2 - initialize_parameters_deep

Implement initialization for an L-layer Neural Network.

Instructions:

- The model's structure is *[LINEAR -> RELU] $\times$ (L-1) -> LINEAR -> SIGMOID*. I.e., it has $L-1$ layers using a ReLU activation function followed by an output layer with a sigmoid activation function.
- Use random initialization for the weight matrices. Use `np.random.randn(d0, d1, ..., dn) * 0.01` .
- Use zeros initialization for the biases. Use `np.zeros(shape)` .
- You'll store $n^{[l]}$, the number of units in different layers, in a variable `layer_dims` . For example, the `layer_dims` for last week's Planar Data classification model would have been [2,4,1]: There were two inputs, one hidden layer with 4 hidden units, and an output layer with 1 output unit. This means `W1` 's shape was (4,2), `b1` was (4,1), `W2` was (1,4) and `b2` was (1,1). Now you will generalize this to L layers!
- Here is the implementation for $L=1$ (one layer neural network). It should inspire you to implement the general case (L-layer neural network).

```
    if L == 1:
        parameters["W" + str(L)] = np.random.randn(layer_dims[1], layer_dims[0]) * 0.01
        parameters["b" + str(L)] = np.zeros((layer_dims[1], 1))
```

In [3]:
```
import numpy as np

def initialize_parameters_deep(layer_dims):
    """
    Arguments:
    layer_dims -- python array (list) containing the dimensions of each layer

    Returns:
    parameters -- python dictionary containing your parameters "W1", "b1", ...
                    Wl -- weight matrix of shape (layer_dims[l], layer_dims[l-
                    bl -- bias vector of shape (layer_dims[l], 1)
    """

    np.random.seed(3)  # Ensures reproducibility
    parameters = {}
    L = len(layer_dims)  # number of layers in the network

    for l in range(1, L):
        # Initialize weights using random values scaled by 0.01
        parameters['W' + str(l)] = np.random.randn(layer_dims[l], layer_dims[l
        # Initialize biases as zeros
        parameters['b' + str(l)] = np.zeros((layer_dims[l], 1))

        # Check the shapes of the parameters
        assert(parameters['W' + str(l)].shape == (layer_dims[l], layer_dims[l
        assert(parameters['b' + str(l)].shape == (layer_dims[l], 1))

    return parameters

# Test Case 1
print("Test Case 1:\n")
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"]))

# Test Case 2
print("\nTest Case 2:\n")
parameters = initialize_parameters_deep([4, 3, 2])

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

```
Test Case 1:

W1 = [[ 0.01788628  0.0043651   0.00096497 -0.01863493 -0.00277388]
 [-0.00354759 -0.00082741 -0.00627001 -0.00043818 -0.00477218]
 [-0.01313865  0.00884622  0.00881318  0.01709573  0.00050034]
 [-0.00404677 -0.0054536  -0.01546477  0.00982367 -0.01101068]]
b1 = [[0.]
 [0.]
 [0.]
 [0.]]
W2 = [[-0.01185047 -0.0020565   0.01486148  0.00236716]
 [-0.01023785 -0.00712993  0.00625245 -0.00160513]
 [-0.00768836 -0.00230031  0.00745056  0.01976111]]
b2 = [[0.]
 [0.]
 [0.]]

Test Case 2:

W1 = [[ 0.01788628  0.0043651   0.00096497 -0.01863493]
 [-0.00277388 -0.00354759 -0.00082741 -0.00627001]
 [-0.00043818 -0.00477218 -0.01313865  0.00884622]]
b1 = [[0.]
 [0.]
 [0.]]
W2 = [[ 0.00881318  0.01709573  0.00050034]
 [-0.00404677 -0.0054536  -0.01546477]]
b2 = [[0.]
 [0.]]
```

In [ ]:

```python
print("Test Case 1:\n")
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"]))

initialize_parameters_deep_test_1(initialize_parameters_deep)

print("\033[90m\nTest Case 2:\n")
parameters = initialize_parameters_deep([4,3,2])

print("W1 = " + str(parameters["W1"]))
print("b1 = " + str(parameters["b1"]))
print("W2 = " + str(parameters["W2"]))
print("b2 = " + str(parameters["b2"]))
initialize_parameters_deep_test_2(initialize_parameters_deep)
```

Expected output

```
Test Case 1:

W1 = [[ 0.01788628  0.0043651   0.00096497 -0.01863493 -0.00277388]
 [-0.00354759 -0.00082741 -0.00627001 -0.00043818 -0.00477218]
 [-0.01313865  0.00884622  0.00881318  0.01709573  0.00050034]
 [-0.00404677 -0.0054536  -0.01546477  0.00982367 -0.01101068]]
b1 = [[0.]
 [0.]
 [0.]
 [0.]]
W2 = [[-0.01185047 -0.0020565   0.01486148  0.00236716]
 [-0.01023785 -0.00712993  0.00625245 -0.00160513]
 [-0.00768836 -0.00230031  0.00745056  0.01976111]]
b2 = [[0.]
 [0.]
 [0.]]
 All tests passed.

Test Case 2:

W1 = [[ 0.01788628  0.0043651   0.00096497 -0.01863493]
 [-0.00277388 -0.00354759 -0.00082741 -0.00627001]
 [-0.00043818 -0.00477218 -0.01313865  0.00884622]]
b1 = [[0.]
```

4 - Forward Propagation Module

4.1 - Linear Forward

Now that you have initialized your parameters, you can do the forward propagation module. Start by implementing some basic functions that you can use again later when implementing the model. Now, you'll complete three functions in this order:

- LINEAR
- LINEAR -> ACTIVATION where ACTIVATION will be either ReLU or Sigmoid.
- [LINEAR -> RELU] $\times$ (L-1) -> 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 3 - linear_forward

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 also find `np.dot()` useful. If your dimensions don't match, printing `W.shape` may help.

In [4]:
```
import numpy as np

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 la
    W -- weights matrix: numpy array of shape (size of current layer, size of
    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 para
    cache -- a python tuple containing "A", "W" and "b" ; stored for computing
    """

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

    # Store values in cache for backward propagation
    cache = (A, W, b)

    return Z, cache

# Test the function with a simple test case
def linear_forward_test_case():
    A = np.array([[1, 2], [3, 4]])  # Example activations (2x2 matrix)
    W = np.array([[0.1, 0.2], [0.3, 0.4]])  # Example weights (2x2 matrix)
    b = np.array([[0.5], [0.6]])  # Example bias (2x1 matrix)
    return A, W, b

t_A, t_W, t_b = linear_forward_test_case()
t_Z, t_linear_cache = linear_forward(t_A, t_W, t_b)

print("Z = " + str(t_Z))
```

```
Z = [[1.2 1.5]
 [2.1 2.8]]
```

In [ ]:
```
t_A, t_W, t_b = linear_forward_test_case()
t_Z, t_linear_cache = linear_forward(t_A, t_W, t_b)
print("Z = " + str(t_Z))

linear_forward_test(linear_forward)
```

Expected output

```
Z = [[ 3.26295337 -1.23429987]]
```

4.2 - Linear-Activation Forward

In this notebook, you will use two activation functions:

- **Sigmoid**: $\sigma(Z) = \sigma(WA + b) = \frac{1}{1+e^{-(WA+b)}}$. You've been provided with the `sigmoid` function which returns **two** items: the activation value " `a` " and a " `cache` " that contains " `Z` " (it's what we will feed in to the corresponding backward function). To use it you could just call:

```
A, activation_cache = sigmoid(Z)
```

- **ReLU**: The mathematical formula for ReLu is $A = RELU(Z) = max(0, Z)$. You've been provided with the `relu` function. This function returns **two** items: the activation value " `A` " and a " `cache` " that contains " `Z` " (it's what you'll feed in to the corresponding backward function). To use it you could just call:

For added convenience, you're going to group two functions (Linear and Activation) into one function (LINEAR->ACTIVATION). Hence, you'll implement a function that does the LINEAR forward step, followed by an ACTIVATION forward step.

Exercise 4 - linear_activation_forward

Implement the forward propagation of the *LINEAR->ACTIVATION* layer. Mathematical relation is: $A^{[l]} = g(Z^{[l]}) = g(W^{[l]}A^{[l-1]} + b^{[l]})$ where the activation "g" can be sigmoid() or relu(). Use `linear_forward()` and the correct activation function.

In [5]:
```python
import numpy as np

def sigmoid(Z):
    A = 1 / (1 + np.exp(-Z))
    cache = Z
    return A, cache

def relu(Z):
    A = np.maximum(0, Z)
    cache = Z
    return A, 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 previc
    W -- weights matrix: numpy array of shape (size of current layer, size of
    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 st

    Returns:
    A -- the output of the activation function, also called the post-activatic
    cache -- a python tuple containing "linear_cache" and "activation_cache";
    """

    # Perform linear step
    Z, linear_cache = linear_forward(A_prev, W, b)

    # Perform activation step
    if activation == "sigmoid":
        A, activation_cache = sigmoid(Z)
    elif activation == "relu":
        A, activation_cache = relu(Z)

    # Combine both caches
    cache = (linear_cache, activation_cache)

    return A, cache

# Testing the function with a simple test case
def linear_activation_forward_test_case():
    A_prev = np.array([[1, 2], [3, 4]])  # Example activations (2x2 matrix)
    W = np.array([[0.1, 0.2], [0.3, 0.4]])  # Example weights (2x2 matrix)
    b = np.array([[0.5], [0.6]])  # Example bias (2x1 matrix)
    return A_prev, W, b

# Test with Sigmoid
t_A_prev, t_W, t_b = linear_activation_forward_test_case()
t_A, t_linear_activation_cache = linear_activation_forward(t_A_prev, t_W, t_b,
print("With sigmoid: A = " + str(t_A))

# Test with ReLU
t_A, t_linear_activation_cache = linear_activation_forward(t_A_prev, t_W, t_b,
print("With ReLU: A = " + str(t_A))
```

```
With sigmoid: A = [[0.76852478 0.81757448]
 [0.89090318 0.94267582]]
With ReLU: A = [[1.2 1.5]
 [2.1 2.8]]
```

In [ ]:
```
t_A_prev, t_W, t_b = linear_activation_forward_test_case()

t_A, t_linear_activation_cache = linear_activation_forward(t_A_prev, t_W, t_b,
print("With sigmoid: A = " + str(t_A))

t_A, t_linear_activation_cache = linear_activation_forward(t_A_prev, t_W, t_b,
print("With ReLU: A = " + str(t_A))

linear_activation_forward_test(linear_activation_forward)
```

```
***Expected output***
```
With sigmoid: A = [[0.96890023 0.11013289]]
With ReLU: A = [[3.43896131 0.   ]]
```
```

Note: In deep learning, the "[LINEAR->ACTIVATION]" computation is counted as a single layer in the neural network, not two layers.

4.3 - L-Layer Model

For even *more* convenience when implementing the L-layer Neural Net, you will need a function that replicates the previous one (`linear_activation_forward` with RELU) $L-1$ times, then follows that with one `linear_activation_forward` with SIGMOID.

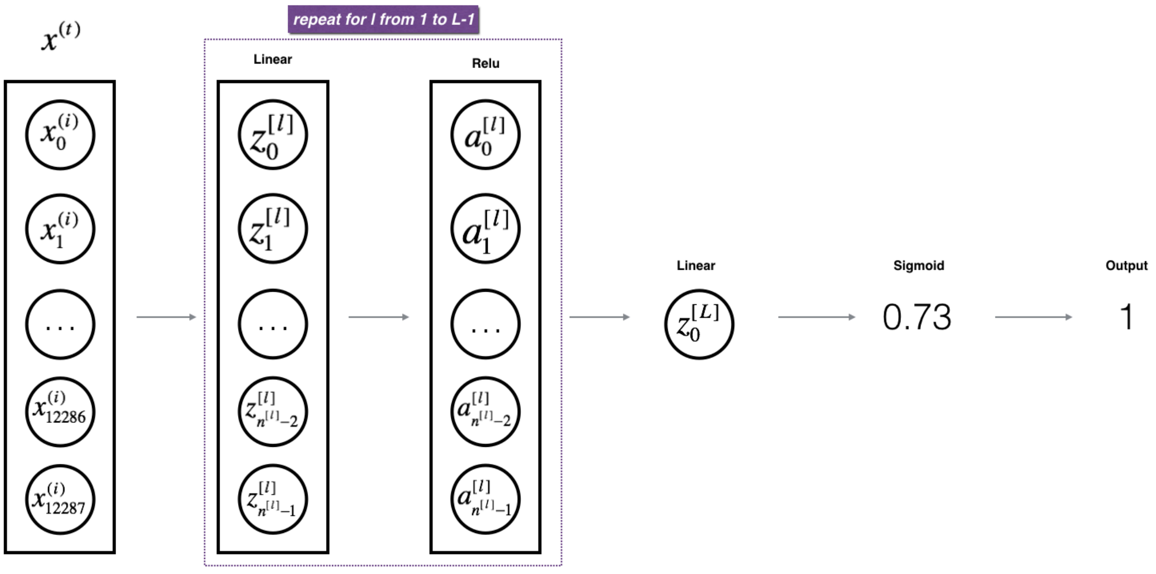

Figure 2 : *[LINEAR -> RELU] $\times$ (L-1) -> LINEAR -> SIGMOID* model

Exercise 5 - L_model_forward

Implement the forward propagation of the above model.

Instructions: In the code below, the variable `AL` will denote $A^{[L]} = \sigma(Z^{[L]}) = \sigma(W^{[L]} A^{[L-1]} + b^{[L]})$. (This is sometimes also called `Yhat`, i.e., this is $\hat{Y}$.)

Hints:

- Use the functions you've previously written
- Use a for loop to replicate [LINEAR->RELU] (L-1) times
- Don't forget to keep track of the caches in the "caches" list. To add a new value `c` to a `list`, you can use `list.append(c)`.

In [6]:
```
def L_model_forward(X, parameters):
    """
    Implement forward propagation for the [LINEAR->RELU]*(L-1)->LINEAR->SIGMOI

    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 the
    """

    caches = []  # List to store all the caches
    A = X        # Start with the input data
    L = len(parameters) // 2  # Number of layers in the neural network (as par

    # Implement [LINEAR -> RELU]*(L-1). Add "cache" to the "caches" list.
    for l in range(1, L):  # We use layer 1 to L-1 for LINEAR->RELU
        A_prev = A
        A, cache = linear_activation_forward(A_prev, parameters["W" + str(l)],
        caches.append(cache)  # Store the cache of this layer

    # Implement LINEAR -> SIGMOID for the final layer. Add "cache" to the "cac
    AL, cache = linear_activation_forward(A, parameters["W" + str(L)], paramet
    caches.append(cache)  # Store the cache of the last layer

    return AL, caches

# Example testing function
def L_model_forward_test_case_2hidden():
    """
    Returns a test case with input size (5, 4), number of examples 4, and para
    for a 3-layer neural network.
    """
    np.random.seed(1)
    X = np.random.randn(5, 4)  # Input data (5 features, 4 examples)
    parameters = {
        "W1": np.random.randn(4, 5),  # Weights for first layer (4 neurons, 5
        "b1": np.random.randn(4, 1),  # Bias for first layer (4 neurons)
        "W2": np.random.randn(3, 4),  # Weights for second layer (3 neurons, 4
        "b2": np.random.randn(3, 1),  # Bias for second layer (3 neurons)
        "W3": np.random.randn(1, 3),  # Weights for third layer (1 neuron, 3 f
        "b3": np.random.randn(1, 1)   # Bias for third layer (1 neuron)
    }
    return X, parameters

# Test the function
t_X, t_parameters = L_model_forward_test_case_2hidden()
t_AL, t_caches = L_model_forward(t_X, t_parameters)

print("AL = " + str(t_AL))  # AL should be the output layer activation after f
```

```
AL = [[0.77634609 0.9998399  0.99021857 0.33755508]]
```

In [ ]:
```
t_X, t_parameters = L_model_forward_test_case_2hidden()
t_AL, t_caches = L_model_forward(t_X, t_parameters)

print("AL = " + str(t_AL))

L_model_forward_test(L_model_forward)
```

Expected output

```
AL = [[0.03921668 0.70498921 0.19734387 0.04728177]]
```

Awesome! You've implemented a full forward propagation that takes the input X and outputs a row vector $A^{[L]}$ containing your predictions. It also records all intermediate values in "caches". Using $A^{[L]}$, you can compute the cost of your predictions.

5 - Cost Function

Now you can implement forward and backward propagation! You need to compute the cost, in order to check whether your model is actually learning.

Exercise 6 - compute_cost

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

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

In [7]:
```
import numpy as np

def compute_cost(AL, Y):
    """
    Implement the cost function defined by equation (7).

    Arguments:
    AL -- probability vector corresponding to your label predictions, shape (1
    Y -- true "label" vector (for example: containing 0 if non-cat, 1 if cat),

    Returns:
    cost -- cross-entropy cost
    """

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

    # Compute the cross-entropy cost
    cost = -(1/m) * np.sum(Y * np.log(AL) + (1 - Y) * np.log(1 - AL))

    cost = np.squeeze(cost)  # To ensure cost is a scalar (e.g., [[17]] become

    return cost

# Example test case for compute_cost
def compute_cost_test_case():
    """
    This test case will return a sample Y and AL for testing the cost functior
    """
    np.random.seed(1)
    Y = np.array([[1, 0, 1]])  # True labels (1 row, 3 examples)
    AL = np.array([[0.9, 0.1, 0.8]])  # Predicted probabilities (1 row, 3 exam
    return Y, AL

# Run the test
t_Y, t_AL = compute_cost_test_case()
t_cost = compute_cost(t_AL, t_Y)

print("Cost: " + str(t_cost))  # Expected output: a scalar cost value
```

```
Cost: 0.14462152754328741
```

In [ ]:
```
t_Y, t_AL = compute_cost_test_case()
t_cost = compute_cost(t_AL, t_Y)

print("Cost: " + str(t_cost))

compute_cost_test(compute_cost)
```

Expected Output:

cost	0.2797765635793422

6 - Backward Propagation Module

Just as you did for the forward propagation, you'll implement helper functions for backpropagation. Remember that backpropagation is used to calculate the gradient of the loss function with respect to the parameters.

Reminder:

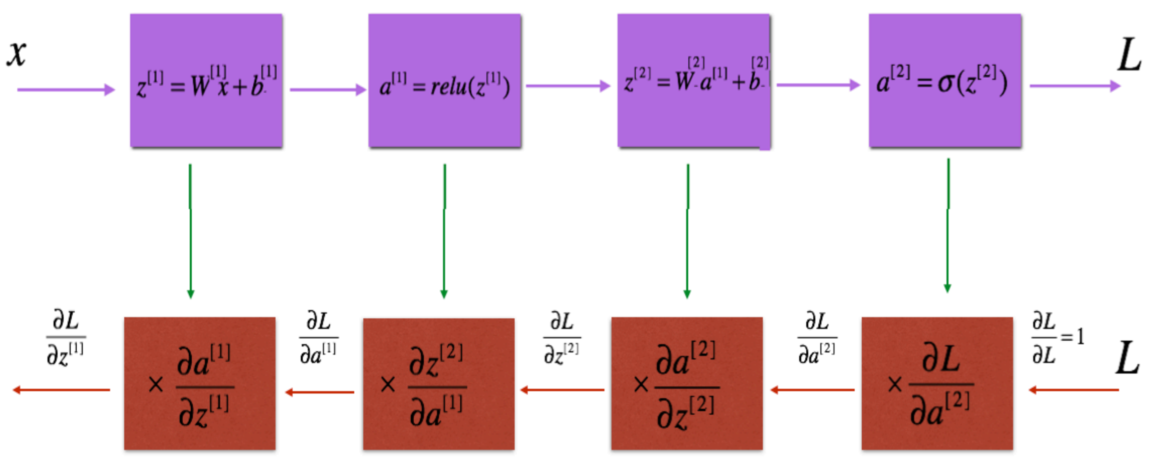
x
$z^{[1]} = W^{[1]}x + b^{[1]}$
$a^{[1]} = relu(z^{[1]})$
$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$
$a^{[2]} = \sigma(z^{[2]})$
L
$\frac{\partial L}{\partial z^{[1]}}$
$\times \frac{\partial a^{[1]}}{\partial z^{[1]}}$
$\frac{\partial L}{\partial a^{[1]}}$
$\times \frac{\partial z^{[2]}}{\partial a^{[1]}}$
$\frac{\partial L}{\partial z^{[2]}}$
$\times \frac{\partial a^{[2]}}{\partial z^{[2]}}$
$\frac{\partial L}{\partial a^{[2]}}$
$\times \frac{\partial L}{\partial a^{[2]}}$
$\frac{\partial L}{\partial L} = 1$
L

Figure 3: Forward and Backward propagation for LINEAR->RELU->LINEAR->SIGMOID
The purple blocks represent the forward propagation, and the red blocks represent the backward propagation.

Now, similarly to forward propagation, you're going to build the backward propagation in three steps:

1. LINEAR backward
2. LINEAR -> ACTIVATION backward where ACTIVATION computes the derivative of either

For the next exercise, you will need to remember that:

- `b` is a matrix(np.ndarray) with 1 column and n rows, i.e: b = [[1.0], [2.0]] (remember that `b` is a constant)
- np.sum performs a sum over the elements of a ndarray
- axis=1 or axis=0 specify if the sum is carried out by rows or by columns respectively
- keepdims specifies if the original dimensions of the matrix must be kept.
- Look at the following example to clarify:

In [9]:
```
def linear_backward(dZ, cache):
    """
    Implements the backward propagation for a single linear layer.

    Arguments:
    dZ -- gradient of the cost with respect to Z (output of the activation fur
    cache -- tuple containing A_prev, W, b (from the forward pass)

    Returns:
    dA_prev -- gradient of the cost with respect to the activations of the pre
    dW -- gradient of the cost with respect to the weights
    db -- gradient of the cost with respect to the bias
    """
    A_prev, W, b = cache
    m = A_prev.shape[1]

    # Compute gradients
    dW = (1/m) * np.dot(dZ, A_prev.T)
    db = (1/m) * np.sum(dZ, axis=1, keepdims=True)
    dA_prev = np.dot(W.T, dZ)

    return dA_prev, dW, db
```

6.1 - Linear Backward

For layer l, the linear part is: $Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}$ (followed by an activation).

Suppose you have already calculated the derivative $dZ^{[l]} = \frac{\partial \mathcal{L}}{\partial Z^{[l]}}$. You want to get $(dW^{[l]}, db^{[l]}, dA^{[l-1]})$.

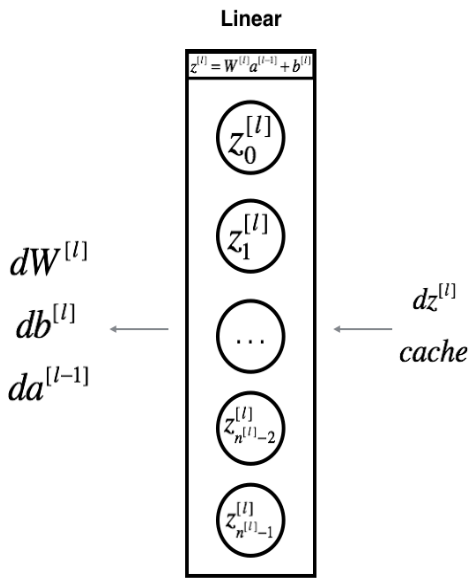

Figure 4

The three outputs $(dW^{[l]}, db^{[l]}, dA^{[l-1]})$ are computed using the input $dZ^{[l]}$.

Here are the formulas you need:

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

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

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

Exercise 7 - linear_backward

Use the 3 formulas above to implement `linear_backward()` .

Hint:

- In numpy you can get the transpose of an ndarray `A` using `A.T` or `A.transpose()`

In [11]:
```
def linear_backward(dZ, cache):
    """
    Implement the linear portion of backward propagation for a single layer (l

    Arguments:
    dZ -- Gradient of the cost with respect to the linear output (of current l
    cache -- tuple of values (A_prev, W, b) coming from the forward propagatio

    Returns:
    dA_prev -- Gradient of the cost with respect to the activation (of the pre
    dW -- Gradient of the cost with respect to W (current layer l), same shape
    db -- Gradient of the cost with respect to b (current layer l), same shape
    """
    A_prev, W, b = cache
    m = A_prev.shape[1]  # number of training examples

    # Compute gradients
    dW = (1 / m) * np.dot(dZ, A_prev.T)  # (dZ[l] @ A_prev.T)
    db = (1 / m) * np.sum(dZ, axis=1, keepdims=True)  # sum across columns
    dA_prev = np.dot(W.T, dZ)  # (W[l]^T @ dZ[l])

    return dA_prev, dW, db
```

In [12]:
```
t_dZ, t_linear_cache = linear_backward_test_case()
t_dA_prev, t_dW, t_db = linear_backward(t_dZ, t_linear_cache)

print("dA_prev: " + str(t_dA_prev))
print("dW: " + str(t_dW))
print("db: " + str(t_db))

linear_backward_test(linear_backward)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-12-3427f65cb756> in <module>
----> 1 t_dZ, t_linear_cache = linear_backward_test_case()
      2 t_dA_prev, t_dW, t_db = linear_backward(t_dZ, t_linear_cache)
      3 
      4 print("dA_prev: " + str(t_dA_prev))
      5 print("dW: " + str(t_dW))

NameError: name 'linear_backward_test_case' is not defined
```

Expected Output:

```
dA_prev: [[-1.15171336  0.06718465 -0.3204696   2.09812712]
 [ 0.60345879 -3.72508701  5.81700741 -3.84326836]
 [-0.4319552  -1.30987417  1.72354705  0.05070578]
 [-0.38981415  0.60811244 -1.25938424  1.47191593]
 [-2.52214926  2.67882552 -0.67947465  1.48119548]]
dW: [[ 0.07313866 -0.0976715  -0.87585828  0.73763362  0.00785716]
 [ 0.85508818  0.37530413 -0.59912655  0.71278189 -0.58931808]
 [ 0.97913304 -0.24376494 -0.08839671  0.55151192 -0.10290907]]
db: [[-0.14713786]
 [-0.11313155]
 [-0.13209101]]
```

6.2 - 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`**.

To help you implement `linear_activation_backward`, two backward functions have been provided:

- **`sigmoid_backward`**: Implements the backward propagation for SIGMOID unit. You can call it as follows:

  ```
  dZ = sigmoid_backward(dA, activation_cache)
  ```

- **`relu_backward`**: Implements the backward propagation for RELU unit. You can call it as follows:

  ```
  dZ = relu_backward(dA, activation_cache)
  ```

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

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

Exercise 8 - linear_activation_backward

Implement the backpropagation for the *LINEAR->ACTIVATION* layer.

In [16]:
```
import numpy as np

# Define the test case for linear_activation_backward
def linear_activation_backward_test_case():
    # Create some mock data for testing
    A_prev = np.array([[0.2, 0.4], [0.1, 0.7], [0.5, 0.9], [0.8, 0.6]])
    W = np.array([[0.1, 0.2], [0.3, 0.4], [0.5, 0.6]])
    b = np.array([[0.1], [0.2], [0.3]])

    # Simulate the output of the activation function (for simplicity)
    activation_cache = np.array([[0.5, 0.6], [0.4, 0.8], [0.3, 0.2], [0.7, 0.5

    # Simulate the gradient from the next layer (post-activation)
    dA = np.array([[0.1, 0.2], [0.3, 0.4], [0.5, 0.6], [0.7, 0.8]])

    # Return a tuple containing the necessary cache and gradient
    return dA, (A_prev, W, b), activation_cache

# Now use the test case to test your function
t_dAL, t_linear_activation_cache = linear_activation_backward_test_case()

# Test with Sigmoid activation
t_dA_prev, t_dW, t_db = linear_activation_backward(t_dAL, t_linear_activation_
print("With sigmoid: dA_prev = " + str(t_dA_prev))
print("With sigmoid: dW = " + str(t_dW))
print("With sigmoid: db = " + str(t_db))

# Test with ReLU activation
t_dA_prev, t_dW, t_db = linear_activation_backward(t_dAL, t_linear_activation_
print("With relu: dA_prev = " + str(t_dA_prev))
print("With relu: dW = " + str(t_dW))
print("With relu: db = " + str(t_db))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-16-b98eeb9c9fe4> in <module>
     18 
     19 # Now use the test case to test your function
---> 20 t_dAL, t_linear_activation_cache = linear_activation_backward_test_c
ase()
     21 
     22 # Test with Sigmoid activation

ValueError: too many values to unpack (expected 2)
```

In [14]:
```
t_dAL, t_linear_activation_cache = linear_activation_backward_test_case()

t_dA_prev, t_dW, t_db = linear_activation_backward(t_dAL, t_linear_activation_
print("With sigmoid: dA_prev = " + str(t_dA_prev))
print("With sigmoid: dW = " + str(t_dW))
print("With sigmoid: db = " + str(t_db))

t_dA_prev, t_dW, t_db = linear_activation_backward(t_dAL, t_linear_activation_
print("With relu: dA_prev = " + str(t_dA_prev))
print("With relu: dW = " + str(t_dW))
print("With relu: db = " + str(t_db))

linear_activation_backward_test(linear_activation_backward)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-14-1dd7958789b5> in <module>
----> 1 t_dAL, t_linear_activation_cache = linear_activation_backward_test_c
ase()
      2 
      3 t_dA_prev, t_dW, t_db = linear_activation_backward(t_dAL, t_linear_a
ctivation_cache, activation = "sigmoid")
      4 print("With sigmoid: dA_prev = " + str(t_dA_prev))
      5 print("With sigmoid: dW = " + str(t_dW))

NameError: name 'linear_activation_backward_test_case' is not defined
```

Expected output:

```
With sigmoid: dA_prev = [[ 0.11017994  0.01105339]
 [ 0.09466817  0.00949723]
 [-0.05743092 -0.00576154]]
With sigmoid: dW = [[ 0.10266786  0.09778551 -0.01968084]]
With sigmoid: db = [[-0.05729622]]
With relu: dA_prev = [[ 0.44090989  0.        ]
 [ 0.37883606  0.        ]
 [-0.2298228   0.        ]]
With relu: dW = [[ 0.44513824  0.37371418 -0.10478989]]
With relu: db = [[-0.20837892]]
```

6.3 - 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'll use those variables to compute the gradients. Therefore, in the `L_model_backward` function, you'll 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 5 below shows the backward pass.

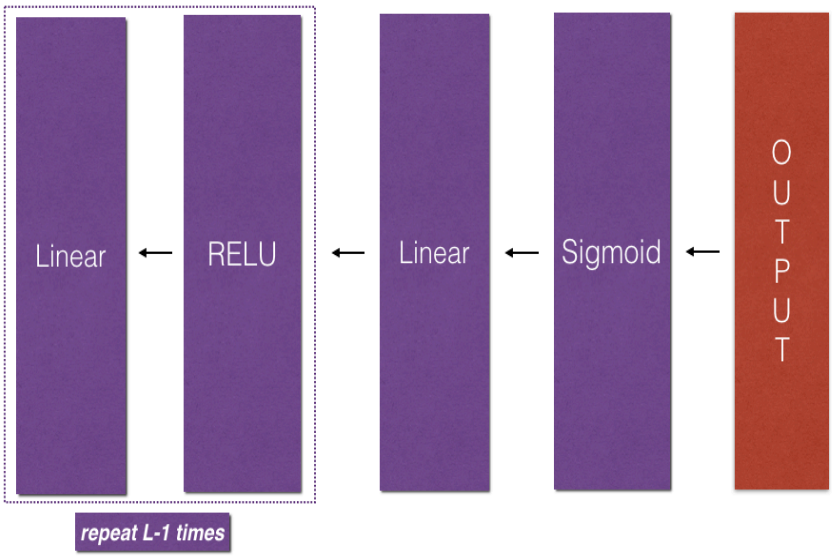

Figure 5: Backward pass

Initializing backpropagation:

To backpropagate through this network, you know that the output is: $A^{[L]} = \sigma(Z^{[L]})$. Your code thus needs to compute `dAL` $= \frac{\partial \mathcal{L}}{\partial A^{[L]}}$. To do so, use this formula (derived using calculus which, again, you don't need in-depth knowledge of!):

```
dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL)) # derivative of cost with respect to AL
```

You can then use this post-activation gradient `dAL` to keep going backward. As seen in Figure 5, you can now feed in `dAL` into the LINEAR->SIGMOID backward function you implemented (which will use the cached values stored by the L_model_forward function).

After that, you will have to use a `for` loop to iterate through all the other layers using the LINEAR->RELU backward function. You should store each dA, dW, and db in the grads dictionary. To do so, use this formula :

$$grads["dW" + str(l)] = dW^{[l]} \tag{15}$$

For example, for $l = 3$ this would store $dW^{[l]}$ in `grads["dW3"]`.

Exercise 9 - L_model_backward

In [17]:
```
def L_model_backward(AL, Y, caches):
    """
    Implement the backward propagation for the [LINEAR->RELU] * (L-1) -> LINEA

    Arguments:
    AL -- probability vector, output of the forward propagation (L_model_forwa
    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 c
                the cache of linear_activation_forward() with "sigmoid" (it's

    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
    dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL))  # derivative of cos

    # Lth layer (SIGMOID -> LINEAR) gradients.
    current_cache = caches[L-1]
    dA_prev_temp, dW_temp, db_temp = linear_activation_backward(dAL, current_c
    grads["dA" + str(L-1)] = dA_prev_temp
    grads["dW" + str(L)] = dW_temp
    grads["db" + str(L)] = db_temp

    # Loop from l = L-2 to l = 0 (go backward through all hidden layers)
    for l in reversed(range(L-1)):
        current_cache = caches[l]
        dA_prev_temp, dW_temp, db_temp = linear_activation_backward(grads["dA'
        grads["dA" + str(l)] = dA_prev_temp
        grads["dW" + str(l + 1)] = dW_temp
        grads["db" + str(l + 1)] = db_temp

    return grads
```

In [18]:
```
t_AL, t_Y_assess, t_caches = L_model_backward_test_case()
grads = L_model_backward(t_AL, t_Y_assess, t_caches)

print("dA0 = " + str(grads['dA0']))
print("dA1 = " + str(grads['dA1']))
print("dW1 = " + str(grads['dW1']))
print("dW2 = " + str(grads['dW2']))
print("db1 = " + str(grads['db1']))
print("db2 = " + str(grads['db2']))

L_model_backward_test(L_model_backward)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-18-3ace16762626> in <module>
----> 1 t_AL, t_Y_assess, t_caches = L_model_backward_test_case()
      2 grads = L_model_backward(t_AL, t_Y_assess, t_caches)
      3 
      4 print("dA0 = " + str(grads['dA0']))
      5 print("dA1 = " + str(grads['dA1']))

NameError: name 'L_model_backward_test_case' is not defined
```

Expected output:

```
dA0 = [[ 0.         0.52257901]
 [ 0.         -0.3269206 ]
 [ 0.         -0.32070404]
 [ 0.         -0.74079187]]
dA1 = [[ 0.12913162 -0.44014127]
 [-0.14175655  0.48317296]
 [ 0.01663708 -0.05670698]]
dW1 = [[0.41010002 0.07807203 0.13798444 0.10502167]
 [0.         0.         0.         0.        ]
 [0.05283652 0.01005865 0.01777766 0.0135308 ]]
dW2 = [[-0.39202432 -0.13325855 -0.04601089]]
db1 = [[-0.22007063]
 [ 0.        ]
 [-0.02835349]]
db2 = [[0.15187861]]
```

6.4 - Update Parameters

In this section, you'll update the parameters of the model, using gradient descent:

$$W^{[l]} = W^{[l]} - \alpha \; dW^{[l]} \tag{16}$$
$$b^{[l]} = b^{[l]} - \alpha \; db^{[l]} \tag{17}$$

where α is the learning rate.

After computing the updated parameters, store them in the parameters dictionary.

Exercise 10 - update_parameters

Implement `update_parameters()` to update your parameters using gradient descent.

Instructions: Update parameters using gradient descent on every $W^{[l]}$ and $b^{[l]}$ for $l = 1, 2, \ldots, L$.

In [23]:
```python
import numpy as np

def update_parameters_test_case():
    # Define sample parameters (weights and biases) for testing
    t_parameters = {
        "W1": np.array([[-0.6, -0.1, -2.1, 1.8],
                        [-1.8, -0.8, 0.5, -1.2],
                        [-1.1, -0.9, 0.7, 2.2]]),
        "b1": np.array([[-0.05], [-1.3], [0.5]]),
        "W2": np.array([[-0.6, 0.04, 1.33]]),
        "b2": np.array([[-0.85]])
    }

    # Define sample gradients for testing (dW and db)
    grads = {
        "dW1": np.array([[0.4, 0.08, 0.14, 0.1],
                         [0.0, 0.0, 0.0, 0.0],
                         [0.05, 0.01, 0.02, 0.01]]),
        "db1": np.array([[-0.22], [0.0], [-0.03]]),
        "dW2": np.array([[-0.39, -0.13, -0.05]]),
        "db2": np.array([[0.15]])
    }

    return t_parameters, grads
```

In [24]:
```
t_parameters, grads = update_parameters_test_case()
t_parameters = update_parameters(t_parameters, grads, 0.1)

print ("W1 = "+ str(t_parameters["W1"]))
print ("b1 = "+ str(t_parameters["b1"]))
print ("W2 = "+ str(t_parameters["W2"]))
print ("b2 = "+ str(t_parameters["b2"]))

update_parameters_test(update_parameters)
```

```
W1 = [[-0.64  -0.108 -2.114  1.79 ]
 [-1.8   -0.8    0.5   -1.2  ]
 [-1.105 -0.901  0.698  2.199]]
b1 = [[-0.028]
 [-1.3  ]
 [ 0.503]]
W2 = [[-0.561  0.053  1.335]]
b2 = [[-0.865]]

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-24-e0da3f3aab9d> in <module>
      7 print ("b2 = "+ str(t_parameters["b2"]))
      8 
----> 9 update_parameters_test(update_parameters)

NameError: name 'update_parameters_test' is not defined
```

Expected 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]]
```

Congratulations!

You've just implemented all the functions required for building a deep neural network, including:

- Using non-linear units improve your model
- Building a deeper neural network (with more than 1 hidden layer)
- Implementing an easy-to-use neural network class

This was indeed a long assignment, but the next part of the assignment is easier. ;)

In the next assignment, you'll be putting all these together to build two models:

- A two-layer neural network
- An L-layer neural network

You will in fact use these models to classify cat vs non-cat images! (Meow!) Great work and

In [ ]:

```
```

In [ ]:

```
```

In [ ]:

```
```