机器学习系列（1）

深度前馈神经网络--原理解释、公式推导及Python实现

深度神经网络的原理解释：

- 定义
- 变量约束
- 前向传播
- 反向传播
- 数据集

公式推导：

- 前向过程
- BP算法
- 梯度下降

Python实现：

- 见文章内容

申明

本文原理解释及公式推导部分均由LSayhi完成，供学习参考，可传播；代码实现部分的框架由Coursera提供，由LSayhi完成，详细数据及代码可在github查阅。 https://github.com/LSayhi/DeepLearning/tree/master/Coursera-deeplearning%E6%B7%B1%E5%BA%A6%E5%AD%A6%E4%B9%A0 (https://github.com/LSayhi/DeepLearning/tree/master/Coursera-deeplearning%E6%B7%B1%E5%BA%A6%E5%AD%A6%E4%B9%A0)

微信公众号：AI有点可ai

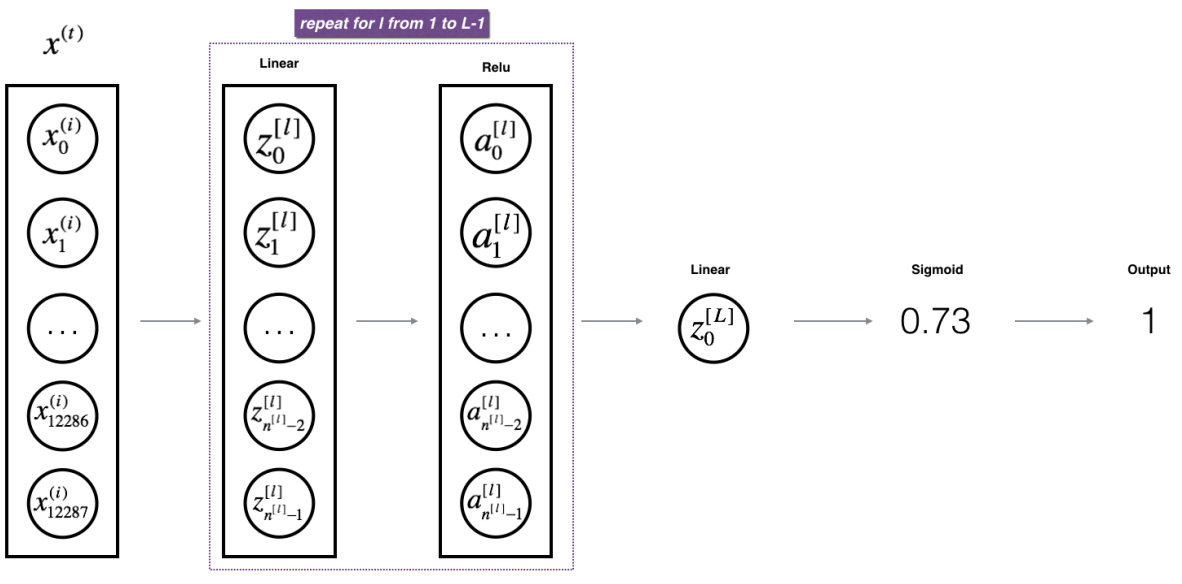
一、原理解释

1.定义： 前馈神经网络，亦称多层感知机，网络的目标是近似一个目标映射f,记为$y=f(x;\theta)$。对于预测型神经网络来说，通过学习参数θ的值，使得函数f拟合因变量自变量之间的映射关系；对于分类神经网络，学习参数θ的值，使映射f拟合各类别之间的边界。

- 神经网络模型由输入层、输出层、隐藏层及连接各层之间的权重（参数）组成。
- “深度”是指 除去模型中输入层后网络的层数；
- “前馈”是指 网络没有反馈的连接方式；
- “网络”是指 它是由不同函数g所复合的。

2.变量约束： 以一个输入层特征数为$n_x(=12888$，n_x定义为输入层的单元数），L（$=L-1+1, L$定义为除去输入层的网络层数）层，即隐藏层数为$L-1$的前馈神经网络为例：

- $z_i^{[l]}$表示第l层第i个线性单元，$z^{[l]}$表示第l层所有线性单元
- $a_i^{[l]}$表示第l层第i个激活单元，$a^{[l]}$表示第l层所有激活单元
- $n^{[l]}$表示第l层单元的数目
- $m=1,2,\ldots\ldots,M$，$z^{[l](m)}$、$a^{[l](m)}$中的l表示层数，m表示样本（数据）序号，$x_n^{[0](m)}$, m表示样本序号（通常省略[0]），n表示第n个特征序号



3.前向传播： 前向传播指的是输入数据（集）从输入层到输入层的计算过程。在输入数据之前，需要先进行参数初始化，即随机生成W矩阵，b矩阵。然后每个层的单元根据W矩阵和前一层的数据进行计算线性输出，再由激活函数非线性化，层叠往复，最后得到输出层的输出数据，称此过程为前向传播。

4.反向传播： 反向传播指的是当输出层计算出输出数据后，与数据集中对应的标签进行比对，求出损失(Loss，单个样本)和代价（损失的和，整个样本集），再求解参数$W^{[l]}$和$b^{[l]}$的偏导数,进而由梯度下降等方法更新参数W和b，当损失为0或者达到目标值时停止，由于求解参数W和b的偏导数是一个反向递推的过程，所以此过程也因此成为反向传播。

5.数据集： 数据集是用来训练和测试网络的数据的集合，包括训练集和测试集。一般来说，训练集用于训练网络达到最佳，测试集用来测试网络泛化性能。

二、公式推导

前向传播:

对于一个 L 层的前馈网络，第 l 层的线性函数实现

$$z^{[l]} = w^{[l]}a^{[l-1]} + b^{[l]}$$

对M个样本向量化后，表示为

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

那么

$$A^{[l]} = g(Z^{[l]}) = g(W^{[l]}A^{[l-1]} + b^{[l]})$$

,其中g为激活函数，常见的激活函数有sigmoid、tanh、Relu等，$l = 1, 2, \dots\dots, L$。

特殊地，输入层$a^{[0]}$另记为x,输出层记为$a^{[L]}$,对M个样本向量化后为X和$A^{[L]}$。由此，只要给出X,由公式

$$A^{[l]} = g(Z^{[l]}) = g(W^{[l]}A^{[l-1]} + b^{[l]})$$

和

$$a^{[0]} = X$$

即可由前向后逐步求出$A^{[L]}$，这就是前向传播的过程。

损失和代价:

- 损失函数：对应于单个样本。

  $$L(\hat{y}, y) = y^{(m)}\log\left(a^{[L](m)}\right) + (1 - y^{(m)})\log\left(1 - a^{[L](m)}\right)$$

  损失函数定义了度量神经网络输出层输出的数据与标签之间的差异值（即损失），其功能类似于均方误差函数

  $$MSE = \sqrt{(\hat{y}^{[m]} - y^{(m)})^2}$$

  事实上，

  $$L(\hat{y}, y) = y^{(m)}\log\left(a^{[L](m)}\right) + (1 - y^{(m)})\log\left(1 - a^{[L](m)}\right)$$

  是交叉信息熵，最小化交叉信息熵也等效于最大似然准则（MLP），最小化交叉信息熵直观上可以认为和最小化误差相似，能够用来度量被比较对象的差异大小，但因均方误差函数的特点，在逻辑回归中很可能会导致收敛速度缓慢，并且代价函数不一定是凸函数，不一定能收敛于最小值，可能会收敛到某个非最小的极小值点，所以我们常用交叉信息熵代替均方误差。这里可以参考个人笔记**机器学习中的数学（1）--交叉信息熵**
- 代价函数：对于M个样本集。

  $$J(X, Y; W, b) = -\frac{1}{M}\sum_{m=1}^{M}(y^{(m)}\log\left(a^{[L](m)}\right) + (1 - y^{(m)})\log\left(1 - a^{[L](m)}\right)) \tag{7}$$

  代价函数关心的是整个样本集的损失

反向传播

- 反向传播过程可由下图表示:

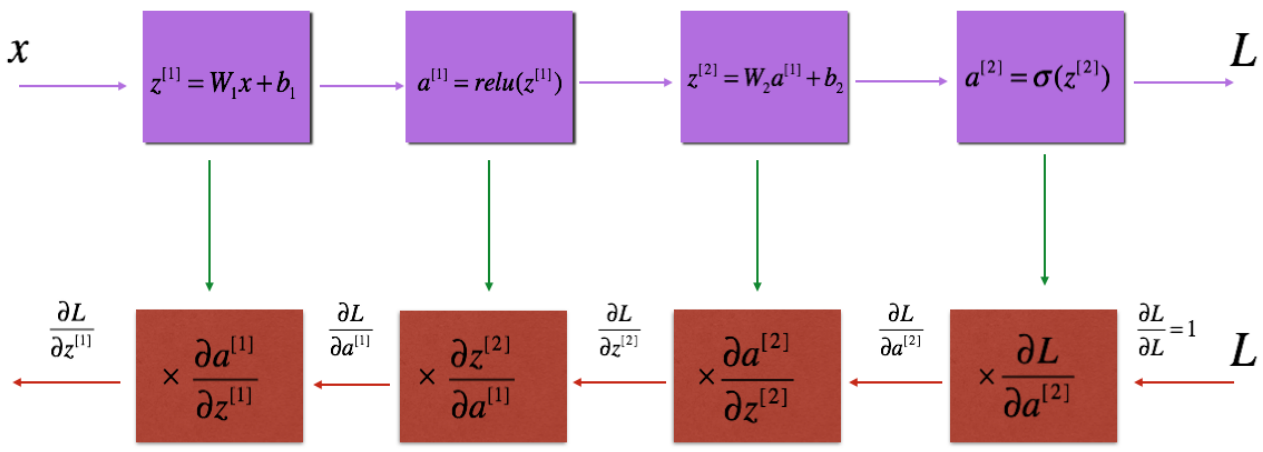


Figure : Forward and Backward propagation for *LINEAR->RELU->LINEAR->SIGMOID*
The purple blocks represent the forward propagation, and the red blocks represent the backward propagation.

网络的目标是使得$J(X, Y; W, b)$最小，在数学上就是多变量函数求最值问题，首先想到的可能是求偏导数，然后另偏导数等于零，找到最小值点，得到对应参数W和b的值，此法称正规方程法，但是由于矩阵XX^T不一定可逆，所以我们使用梯度下降法来寻找最优的参数W和b。 梯度下降法表示为:

$$W = W - \alpha\frac{\partial J}{\partial W}$$

$$b = b - \alpha\frac{\partial J}{\partial b}$$

梯度下降法也需要求出代价函数关于参数W和b的偏导数，由求导数的链式法则可知，

$$\frac{\partial J}{\partial W_{ij}^{[l]}} = \frac{\partial J}{\partial a_i^{[l]}}\frac{\partial a_i^{[l]}}{\partial z_i^{[l]}}\frac{\partial z_i^{[l]}}{\partial W_{ij}^{[l]}} = \frac{\partial J}{\partial a_i^{[l]}}\frac{\partial a_i^{[l]}}{\partial z_i^{[l]}}a_j^{[l-1]} = \frac{\partial J}{\partial a_i^{[l]}}g'(z_i^{[l]})a_j^{[l-1]} \tag{1}$$

$$\frac{\partial J}{\partial b_i^{[l]}} = \frac{\partial J}{\partial a_i^{[l]}}\frac{\partial a_i^{[l]}}{\partial z_i^{[l]}}\frac{\partial z_i^{[l]}}{\partial b_i^{[l]}} = \frac{\partial J}{\partial a_i^{[l]}}g'(z_i^{[l]}) \tag{2}$$

由于，

$$\frac{\partial J}{\partial z_i^{[l]}} = \frac{\partial J}{\partial a_i^{[l]}}\frac{\partial a_i^{[l]}}{\partial z_i^{[l]}} = \frac{\partial J}{\partial a_i^{[l]}}g'(z_i^{[l]}) \tag{3}$$

将③代入①和②，可知:

$$\frac{\partial J}{\partial W_{ij}^{[l]}} = \frac{\partial J}{\partial z_i^{[l]}}a_j^{[l-1]} \tag{4}$$

$$\frac{\partial J}{\partial b_i^{[l]}} = \frac{\partial J}{\partial z_i^{[l]}} \tag{5}$$

将式③、④、⑤矢量化，并记为

$$dz^{[l]} = da^{[l]} * g'(z^{[l]}) \tag{6}$$
$$dW^{[l]} = dz^{[l]} a^{[l-1]} \tag{7}$$
$$db^{[l]} = dz^{[l]} \tag{8}$$

“*”表示矩阵对应元素相乘，⑥式中的$da^{[l]} = (W^{[l+1]})^T dz^{[l+1]}$,推导过程如下,对于第$l$层,应用链式法则

$$da_i^{[l]} = \frac{\partial J}{\partial a_i^{[l]}} = \sum_{j=1}^{n_{l+1}} \frac{\partial J}{\partial a_j^{[l+1]}} \frac{\partial a_j^{[l+1]}}{\partial z_j^{[l+1]}} \frac{\partial z_j^{[l+1]}}{\partial a_i^{[l]}}$$

$$即 da_i^{[l]} = \sum_{j=1}^{n_{l+1}} \frac{\partial J}{\partial a_j^{[l+1]}} g'(z_j^{[l+1]}) W_{ji}^{[l+1]} = \sum_{j=1}^{n_{l+1}} \frac{\partial J}{\partial z_j^{[l+1]}} W_{ji}^{[l+1]} = (W_{(,i)}^{[l+1]})^T \frac{\partial J}{\partial z_j^{[l+1]}}$$

向量化后为：

$$da^{[l]} = (W^{[l+1]})^T dz^{[l+1]}$$

代入⑥，故⑥式可改写为：

$$dz^{[l]} = (W^{[l+1]})^T dz^{[l+1]} * g'(z^{[l]}) \tag{9}$$

再对M个数据向量化：

$$dZ^{[l]} = dA^{[l]} * g'(Z^{[l]}) = (W^{[l+1]})^T dZ^{[l+1]} * g'(Z^{[l]}) \tag{10}$$
$$dW^{[l]} = \frac{\partial J}{\partial W^{[l]}} = \frac{1}{m} dZ^{[l]} A^{[l-1]T} \tag{11}$$
$$db^{[l]} = \frac{\partial J}{\partial b^{[l]}} = \frac{1}{m} \sum_{i=1}^{m} dZ^{[l](i)} \tag{12}$$
$$dA^{[l-1]} = \frac{\partial J}{\partial A^{[l-1]}} = W^{[l]T} dZ^{[l]} \tag{13}$$

梯度下降： 由（10）、（11）、（12）、（13）便可以从输出层反向递推到第1层，得到关于参数W和b的所有偏导数，再应用梯度下降法，不断更新参数：

$$W = W - \alpha \frac{\partial J}{\partial W}$$
$$b = b - \alpha \frac{\partial J}{\partial b}$$

就可以得到最佳的W和b参数，使得训练集的代价函数最小。注：梯度的方向是函数值上升最快的方向，因此梯度的负方向即是函数值下降最快的方向，应用梯度下降即是使函数在某个位置（W,b）沿最快的方向减小。

网络优化：

至此，网络的训练已经完成，可以在测试集上进行泛化测试。如果泛化的准确率不高，分析原因：

- 如果过拟合了，可以考虑增加数据集、正则化、dropout、适当减小网络层数L等
- 如果欠拟合了，可以考虑增加特征数量、增加网络层数、隐藏层神经元数目等

三、Python实现

- 本文的代码框架由Andrew Ng在Coursera.deeplearning.ai的作业中给出，由LSayhi（https://github.com/LSayhi (https://github.com/LSayhi)) 补全，仅供学习参考，勿用于Coursera刷分.

Building your Deep Neural Network: Step by Step

Welcome to your week 4 assignment (part 1 of 2)! You have previously 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 will implement all the functions required to build a deep neural network.
- In the next assignment, you will use these functions to build a deep neural network for image classification.

After this assignment you will 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: $a_i^{[l]}$ denotes the i^{th} entry of the l^{th} layer's activations).

Let's get started!

1 - Packages

Let's first import all the packages that you will 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 will help us grade your work. Please don't change the seed.

In [1]:
```
import numpy as np
import h5py
import matplotlib.pyplot as plt
from testCases_v2 import *
from dnn_utils_v2 import sigmoid, sigmoid_backward, relu, relu_backward

%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 of the Assignment

To build your 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 walk you through the necessary steps. Here is an outline of this assignment, you will:

- 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 $Z^{[l]}$).
  - We give you the ACTIVATION function (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.
  - We give you the gradient of the ACTIVATE function (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.

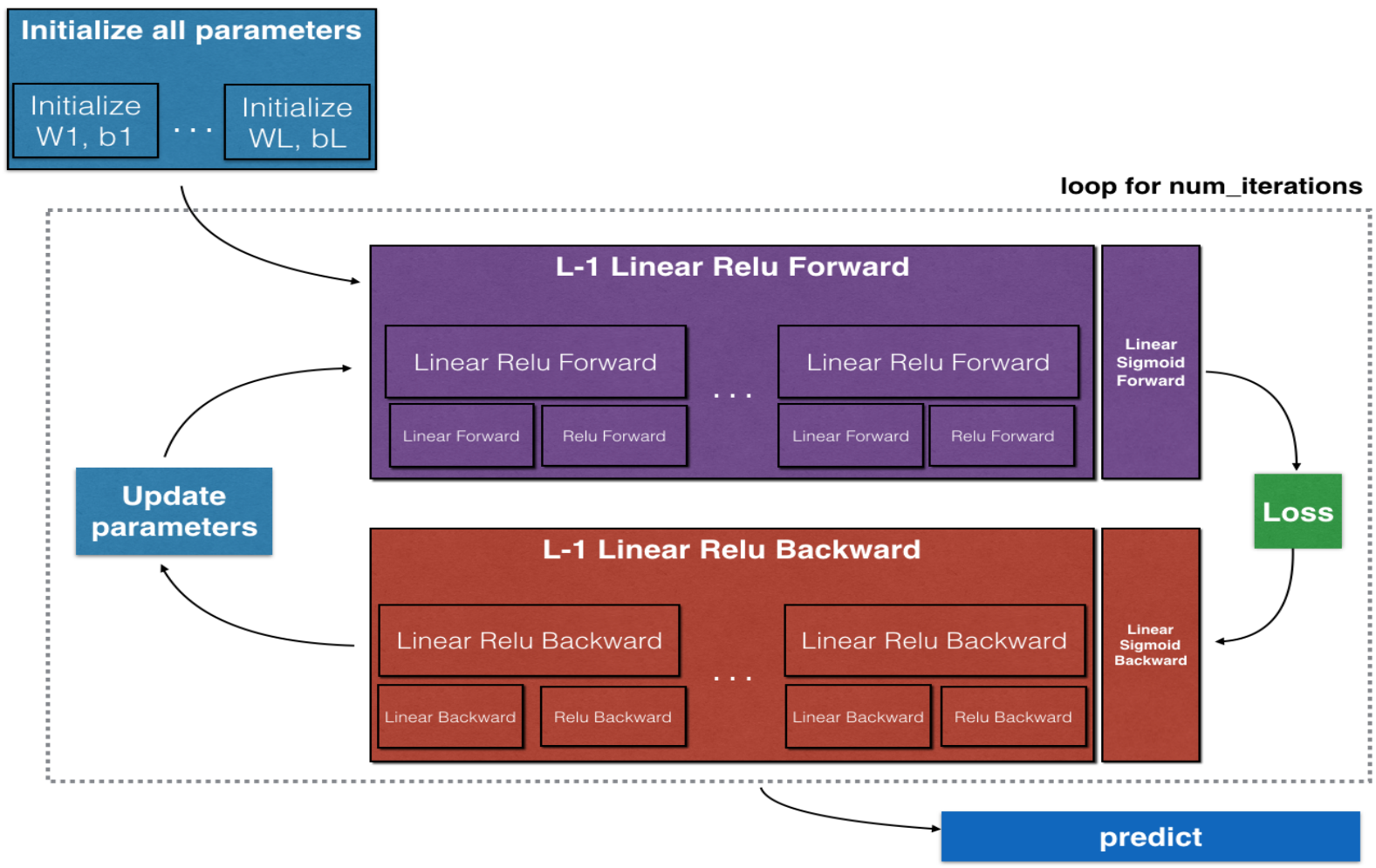


Figure 1

Note that for every forward function, there is a corresponding backward function. That is why at every step of your forward module you will be storing some values in a cache. The cached values are useful for computing gradients. In the backpropagation module you will then use the cache to calculate the gradients. This assignment will show you exactly how to carry out each of these steps.

3 - Initialization

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

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*.
- Use random initialization for the weight matrices. Use `np.random.randn(shape)*0.01` with the correct shape.
- Use zero initialization for the biases. Use `np.zeros(shape)`.

In [2]:
```
# GRADED FUNCTION: initialize_parameters

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)

    ### START CODE HERE ### (≈ 4 lines of code)
    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))
    ### END CODE HERE ###

    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
```

In [3]:
```
parameters = initialize_parameters(2,2,1)
print("W1 = " + str(parameters["W1"]))
print("b1 = " + str(parameters["b1"]))
print("W2 = " + str(parameters["W2"]))
print("b2 = " + str(parameters["b2"]))
```

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

Expected output:

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

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`, you should make sure that 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:

	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	$(n^{[2]}, n^{[1]})$	$(n^{[2]}, 1)$	$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$	$(n^{[2]}, 209)$

⋮	⋮	⋮	⋮	⋮
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 we compute $WX + b$ in python, it carries out broadcasting. For example, if:

$$W = \begin{bmatrix} j & k & l \\ m & n & o \\ p & q & r \end{bmatrix} \quad X = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \quad b = \begin{bmatrix} s \\ t \\ u \end{bmatrix} \tag{2}$$

Then $WX + b$ will be:

$$WX + b = \begin{bmatrix} (ja + kd + lg) + s & (jb + ke + lh) + s & (jc + kf + li) + s \\ (ma + nd + og) + t & (mb + ne + oh) + t & (mc + nf + oi) + t \\ (pa + qd + rg) + u & (pb + qe + rh) + u & (pc + qf + ri) + u \end{bmatrix} \tag{3}$$

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

Instructions:

- The model's structure is *[LINEAR -> RELU] × (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.rand(shape) * 0.01`.
- Use zeros initialization for the biases. Use `np.zeros(shape)`.
- We will store $n^{[l]}$, the number of units in different layers, in a variable `layer_dims`. For example, the `layer_dims` for the "Planar Data classification model" from last week 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. Thus 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 [12]:
```
# 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):
        ### START CODE HERE ### (≈ 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((layer_dims[l], 1))
        ### END CODE 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
```

In [13]:
```
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"]))
```

```
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.]]
```

Expected output:

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.]]

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 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: 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 [14]:
```
# 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 python dictionary containing "A", "W" and "b" ; stored for computing the backward pass efficiently
    """

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

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

    return Z, cache
```

In [15]:
```
A, W, b = linear_forward_test_case()

Z, linear_cache = linear_forward(A, W, b)
print("Z = " + str(Z))
```

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

Expected output:

Z [[3.26295337 -1.23429987]]

4.2 - Linear-Activation Forward

In this notebook, you will use two activation functions:

- **Sigmoid**: $\sigma(Z) = \sigma(WA + b) = \frac{1}{1+e^{-(WA+b)}}$We have provided you with the `sigmoid` function. This function 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 $A = RELU(Z) = max(0, Z)$. We have provided you with the `relu` function. This function 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 = relu(Z)
```

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

Exercise: 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 [16]:
```
# GRADED FUNCTION: linear_activation_forward

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 dictionary containing "linear_cache" and "activation_cache";
             stored for computing the backward pass efficiently
    """

    if activation == "sigmoid":
        # Inputs: "A_prev, W, b". Outputs: "A, activation_cache".
        ### START CODE HERE ### (≈ 2 lines of code)
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = sigmoid(Z)
        ### END CODE HERE ###

    elif activation == "relu":
        # Inputs: "A_prev, W, b". Outputs: "A, activation_cache".
        ### START CODE HERE ### (≈ 2 lines of code)
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = relu(Z)
        ### END CODE HERE ###

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

    return A, cache
```

In [17]:
```
A_prev, W, b = linear_activation_forward_test_case()

A, linear_activation_cache = linear_activation_forward(A_prev, W, b, activation = "sigmoid")
print("With sigmoid: A = " + str(A))

A, linear_activation_cache = linear_activation_forward(A_prev, W, b, activation = "relu")
print("With ReLU: A = " + str(A))
```

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

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.

d) 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.

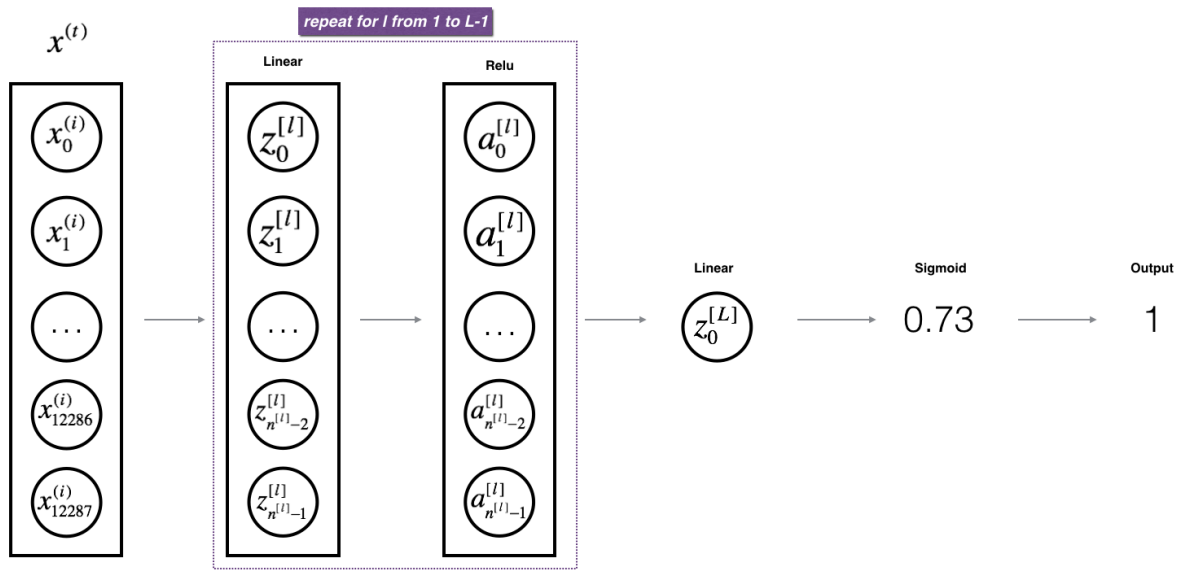


Figure 2 : *[LINEAR -> RELU] × (L-1) -> LINEAR -> SIGMOID* model

Exercise: Implement the forward propagation of the above model.

Instruction: 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 $\hat{Y}$.)

Tips:

- Use the functions you had 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 [26]:
```
# GRADED FUNCTION: L_model_forward

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 -- last post-activation value
    caches -- list of caches containing:
                every cache of linear_relu_forward() (there are L-1 of them, indexed from 0 to L-2)
                the cache of linear_sigmoid_forward() (there is one, indexed 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.
    for l in range(1, L): # l=1 to L-1
        A_prev = A
        ### START CODE HERE ### (≈ 2 lines of code)
        A, cache =linear_activation_forward(A_prev, parameters['W'+str(l)],parameters['b'+str(l)], 'relu')
        caches.append(cache)#cache=A1,W1,b1,Z1,A2,W2,b2,Z2...A(L--1),W(L-1),b(L-1),Z(L-1)

        ### END CODE HERE ###

    # Implement LINEAR -> SIGMOID. Add "cache" to the "caches" list.
    ### START CODE HERE ### (≈ 2 lines of code)
    AL, cache = linear_activation_forward(A, parameters["W"+str(L)],parameters["b"+str(L)], 'sigmoid')
    caches.append(cache)#cache=A1,W1,b1,Z1,A2,W2,b2,Z2...A(L--1),W(L-1),b(L-1),Z(L-1),AL,WL,bL,ZL

    ### END CODE HERE ###

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

    return AL, caches
```

In [29]:
```
X, parameters = L_model_forward_test_case()
AL, caches = L_model_forward(X, parameters)
print("AL = " + str(AL))
print("Length of caches list = " + str(len(caches)))
```

```
AL = [[ 0.17007265  0.2524272 ]]
Length of caches list = 2
```

AL	[[0.17007265 0.2524272]]
Length of caches list	2

Great! Now you have 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 will 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:

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

In [32]:
```
# 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 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.
    ### START CODE HERE ### (≈ 1 lines of code)
    cost = -np.sum(Y*np.log(AL)+(1-Y)*np.log(1-AL))/m
    ### END CODE HERE ###

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

    return cost
```

In [33]:
```
Y, AL = compute_cost_test_case()

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

```
cost = 0.414931599615
```

Expected Output:

cost	0.41493159961539694

6 - Backward propagation module

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

Reminder:

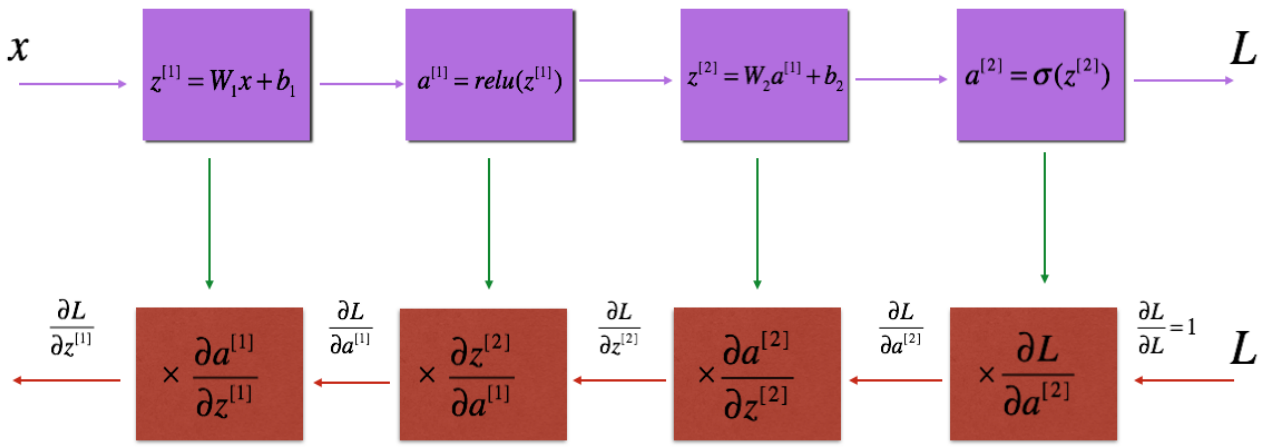


Figure 3 : Forward and Backward propagation for *LINEAR->RELU->LINEAR->SIGMOID*
The purple blocks represent the forward propagation, and the red blocks represent the backward propagation.

Now, similar to forward propagation, you are going to build the backward propagation in three steps:

- LINEAR backward
- LINEAR -> ACTIVATION backward where ACTIVATION computes the derivative of either the ReLU or sigmoid activation
- [LINEAR -> RELU] $\times$ (L-1) -> LINEAR -> SIGMOID backward (whole model)

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 $(dW^{[l]}, db^{[l]} dA^{[l-1]})$.

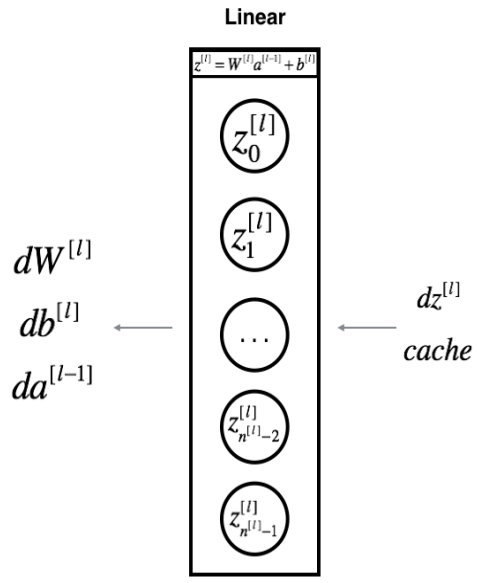


Figure 4

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

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

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

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

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

In [34]:
```
# 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]

    ### START CODE HERE ### (≈ 3 lines of code)
    dW = np.dot(dZ,A_prev.T)/m
    db = np.sum(dZ,axis=1,keepdims=True)/m
    dA_prev = np.dot(W.T,dZ)
    ### END CODE HERE ###

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

    return dA_prev, dW, db
```

In [35]:
```
# Set up some test inputs
dZ, linear_cache = linear_backward_test_case()

dA_prev, dW, db = linear_backward(dZ, linear_cache)
print ("dA_prev = "+ str(dA_prev))
print ("dW = " + str(dW))
print ("db = " + str(db))
```

```
dA_prev = [[ 0.51822968 -0.19517421]
 [-0.40506361  0.15255393]
 [ 2.37496825 -0.89445391]]
dW = [[-0.10076895  1.40685096  1.64992505]]
db = [[ 0.50629448]]
```

Expected Output:

dA_prev	[[0.51822968 -0.19517421] [-0.40506361 0.15255393] [2.37496825 -0.89445391]]
dW	[[-0.10076895 1.40685096 1.64992505]]
db	[[0.50629448]]

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`, we provided two backward functions:

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

In [39]:
```
# 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":
        ### START CODE HERE ### (≈ 2 lines of code)
        dZ = relu_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)
        ### END CODE HERE ###

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

    return dA_prev, dW, db
```

In [40]:
```
AL, linear_activation_cache = linear_activation_backward_test_case()
dA_prev, dW, db = linear_activation_backward(AL, linear_activation_cache, activation = "sigmoid")
print ("sigmoid:")
print ("dA_prev = "+ str(dA_prev))
print ("dW = " + str(dW))
print ("db = " + str(db) + "\n")

dA_prev, dW, db = linear_activation_backward(AL, linear_activation_cache, activation = "relu")
print ("relu:")
print ("dA_prev = "+ str(dA_prev))
print ("dW = " + str(dW))
print ("db = " + str(db))
```

```
sigmoid:
dA_prev = [[ 0.11017994  0.01105339]
 [ 0.09466817  0.00949723]
 [-0.05743092 -0.00576154]]
dW = [[ 0.10266786  0.09778551 -0.01968084]]
db = [[-0.05729622]]

relu:
dA_prev = [[ 0.44090989 -0.        ]
 [ 0.37883606 -0.        ]
 [-0.2298228   0.        ]]
dW = [[ 0.44513824  0.37371418 -0.10478989]]
db = [[-0.20837892]]
```

Expected output with sigmoid:

dA_prev	[[0.11017994 0.01105339] [0.09466817 0.00949723] [-0.05743092 -0.00576154]]
dW	[[0.10266786 0.09778551 -0.01968084]]
db	[[-0.05729622]]

Expected output with relu

dA_prev	[[0.44090989 0.] [0.37883606 0.] [-0.2298228 0.]]
dW	[[0.44513824 0.37371418 -0.10478989]]
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 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 5 below shows the backward pass.

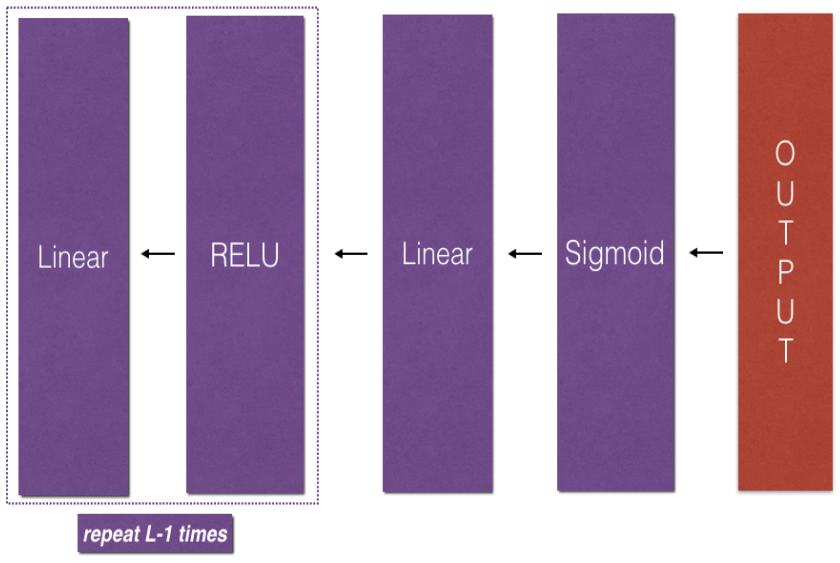


Figure 5 : Backward pass

Initializing backpropagation: To backpropagate through this network, we 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 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: Implement backpropagation for the *[LINEAR->RELU] × (L-1) -> LINEAR -> SIGMOID* model.

In [51]:
```
# 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 (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
    ### START CODE HERE ### (1 line of code)
    dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL))
    ### END CODE HERE ###

    # Lth layer (SIGMOID -> LINEAR) gradients. Inputs: "AL, Y, caches". Outputs: "grads["dAL"], grads["dWL"], grads["dbL"]
    ### START CODE HERE ### (approx. 2 lines)
    current_cache = caches[L-1]
    grads["dA" + str(L)], grads["dW" + str(L)], grads["db" + str(L)] = linear_activation_backward(dAL, current_cache, 'sigmoid')
    ### END CODE HERE ###

    for l in reversed(range(L - 1)):
        # lth layer: (RELU -> LINEAR) gradients.
        # Inputs: "grads["dA" + str(l + 2)], caches". Outputs: "grads["dA" + str(l + 1)] , grads["dW" + str(l + 1)] , grads["db" + str
        ### START CODE HERE ### (approx. 5 lines)
        current_cache = caches[l]
        dA_prev_temp, dW_temp, db_temp = linear_activation_backward(grads["dA" + str(l + 2)], current_cache, 'relu')
        grads["dA" + str(l + 1)] = dA_prev_temp
        grads["dW" + str(l + 1)] = dW_temp
        grads["db" + str(l + 1)] = db_temp
        ### END CODE HERE ###

    return grads
```

In [52]:
```
AL, Y_assess, caches = L_model_backward_test_case()
grads = L_model_backward(AL, Y_assess, caches)
print ("dW1 = "+ str(grads["dW1"]))
print ("db1 = "+ str(grads["db1"]))
print ("dA1 = "+ str(grads["dA1"]))
```

```
dW1 = [[ 0.41010002  0.07807203  0.13798444  0.10502167]
 [ 0.          0.          0.          0.        ]
 [ 0.05283652  0.01005865  0.01777766  0.0135308 ]]
db1 = [[-0.22007063]
 [ 0.        ]
 [-0.02835349]]
dA1 = [[ 0.          0.52257901]
 [ 0.         -0.3269206 ]
 [ 0.         -0.32070404]
 [ 0.         -0.74079187]]
```

Expected Output

dW1	[[0.41010002 0.07807203 0.13798444 0.10502167] [0. 0. 0. 0.] [0.05283652 0.01005865 0.01777766 0.0135308]]
db1	[[-0.22007063] [0.] [-0.02835349]]
dA1	[[0. 0.52257901] [0. -0.3269206] [0. -0.32070404] [0. -0.74079187]]

6.4 - Update Parameters

In this section you will 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: 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 [61]:
```
# GRADED FUNCTION: update_parameters

def update_parameters(parameters, grads, learning_rate):
    """
    Update parameters using gradient descent

    Arguments:
    parameters -- 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)] = ...
    """

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

    # Update rule for each parameter. Use a for loop.
    ### START CODE HERE ### (≈ 3 lines of code)
    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)]
    ### END CODE HERE ###
    return parameters
```

In [62]:
```
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"]))
```

```
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]]
```

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]]

7 - Conclusion

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

We know it was a long assignment but going forward it will only get better. The next part of the assignment is easier.

In the next assignment you will put 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!