Construindo um Deep Neural Network: passo a passo

Bem-vindo a tarefa da semana 4 (partes 1 e 2)! Você já conseguiu treinar uma rede neural de 2 camadas (com uma única camada escondida). Nesta semana você irá construir uma "deep neural network" com o número de camadas escondidas que você desejar.

- Neste notebook, você irá implementar todas as funções necessárias para construir uma deep neural network.
- Na próxima tarefa você irá utilizar todas estas funções para construir uma deep neural network para classificação de imagens.

Após esta tarefa você será capaz de:

- Utilizar nós não-lineares como ReLu para melhorar o modelo.
- Construir uma rede neural mais "profunda" (com várias camadas escondidas)
- Implementar uma classe fácil de se utilizar de redes neurais.

Notação:

- Sobrescrito $[l]$ indica a quantidade associada com a l^{th} camada.
  - Exemplo: $a^{[L]}$ é a L^{th} camada de ativação. $W^{[L]}$ e $b^{[L]}$ são os parâmetros da L^{th} camada.
- Sobrescrito (i) indica a quantidade associada com o i^{th} exemplo.
  - Exemplo: $x^{(i)}$ é o i^{th} exemplo de treinamento.
- Subescrito i indica a i^{th} entrada de um vetor.
  - Exemplo: $a^{[l]}_i$ indica a i^{th} entrada da l^{th} camada de ativação.

Vamos começar!

1 - Pacotes

Primeiro iremos importar todos os pacotes que utilizaremos durante esta tarefa.

- numpy (www.numpy.org) é o pacote principal para computação científica em python.
- matplotlib (http://matplotlib.org) é a biblioteca do python para plotar gráficos.
- dnn_utils contém algumas funções necessárias neste notebook.
- testCases contém alguns exemplos que serão utilizados para testar o código deste notebook.
- np.random.seed(1) é utilizado para manter a chamada às funções deste notebook consistentes. Não modifique a semente..

In [1]:
```
import numpy as np
import h5py
import matplotlib.pyplot as plt
from testCases_v3 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 - Descrição da tarefa

Para construir a rede neural inicialmente iremos implementar algumas funções de apoio. Estas funções serão utilizadas na próxima tarefa para construir uma rede neural com duas camadas escondidas e também para construir uma rede neural com L camadas escondidas. Cada função terá instruções detalhadas para que você consiga implementá-la sem problemas. Abaixo damos uma descrição dos passos desta tarefa.

- Inicialize os parâmetros pra uma rede neural de 2 camadas e para uma rede neural de L camadas.
- Implemente a propagação para frente (mostrado em roxo na figura abaixo).
  - Complete a parte LINEAR para a etapa de propagação para frente (obtendo assim $Z^{[l]}$).
  - É fornecida a função de ativação (relu/sigmoid).
  - Combine as duas etapas anteriores em uma nova função para frente [LINEAR->ACTIVATION].
  - Empilhe a função para frente [LINEAR->RELU] L-1 vezes (para as camadas de 1 até L-1) e adicione a [LINEAR->SIGMOID] no final (para a camada final L). Isto irá criar um modelo novo: modelo_para_frente_L.
- Compute a perda.
- Implemente a propagação para trás (mostrado em vermelho na figura abaixo).
  - Complete a parte LINEAR da etapa de propagação para trás.
  - É fornecida a a função gradiente da função ACTIVATE function (relu_backward/sigmoid_backward)
  - Combine as duas etapas anteriores em uma nova função para trás [LINEAR->ACTIVATION].
  - Empilhe [LINEAR->RELU] para trás L-1 vezes e adicione [LINEAR->SIGMOID] em uma nova função modelo_para_tras_L.
- Finalmente atualize os parâmetros.

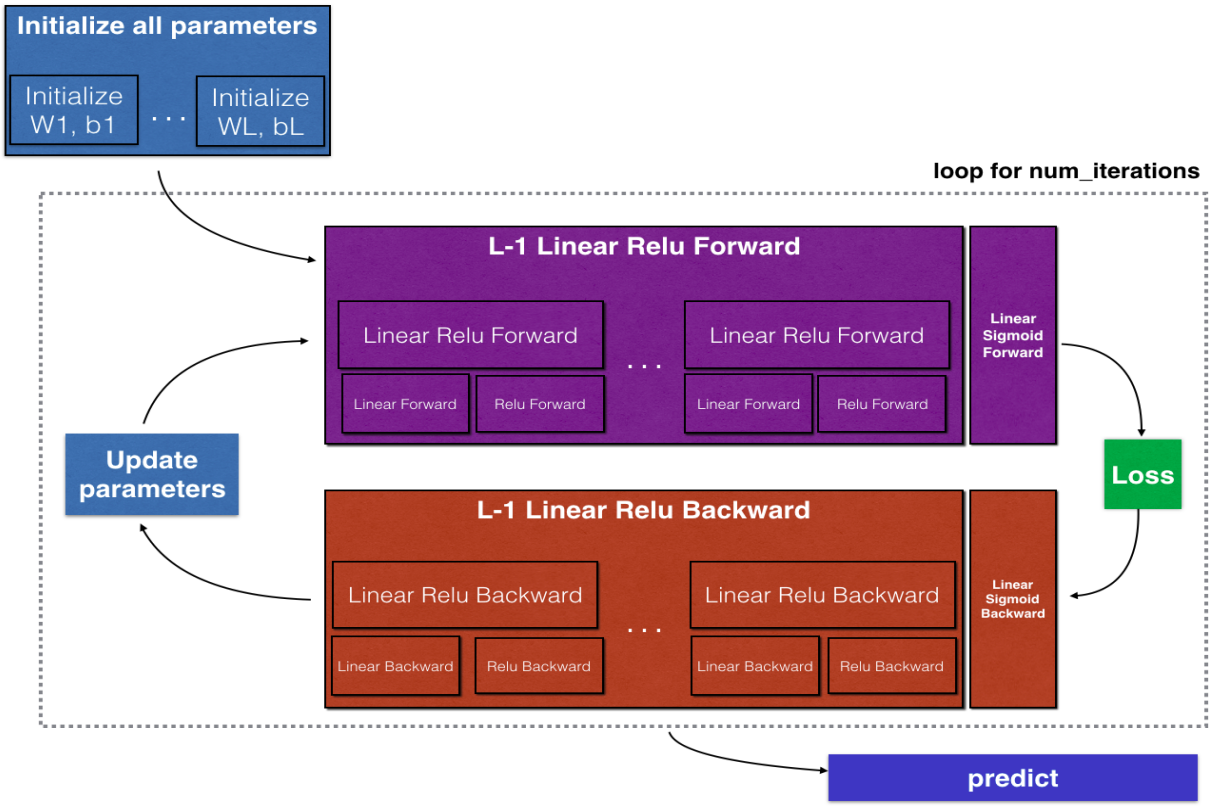

Figure 1

Nota para cada função para frente existe uma função correspondente para trás. Por isso que em cada etapa no módulo para frente você deverá armazenar alguns valores na cache. Os valores da cache serão utilizados para computar os gradientes. No módulo de propagação para trás você irá utilizar os valores da cache para determinar os gradientes. Esta tarefa irá te mostrar como executar cada uma destas etapas.

3 - Inicialização

Aqui você deverá implementar duas funções para inicialização do modelo. A primeira função será utilizada para incializar os parâmetros em uma rede neural com duas camadas escondidas. A segunda função irá generalizar o processo de inicialização para L camadas escondidas.

3.1 - Rede Neural com 2 camadas escondidas

Exercício: Crie e inicialize os parâmetros de uma rede neural com duas camadas escondidas.

Instruções:

- A estrutura do modelo é dada por: *LINEAR -> RELU -> LINEAR -> SIGMOID*.
- Use uma inicialização aleatória para as matrizes de pesos. Use `np.random.randn(formato)*0.01` com o formato correspondente.
- Use uma inicialização com zero para os bias. Use `np.zeros(formato)`.

In [41]:
```
# FUNÇÃO DE AVALIAÇÃO: inicializar_parametros

def inicializar_parametros(n_x, n_h, n_y):
    """
    Argumentos:
    n_x -- tamanho da camada de entrada (número de nós)
    n_h -- tamanho da camada escondida (número de nós)
    n_y -- tamanho da camada de saída (número de nós)

    Retorno:
    parametros -- dicionário python contendo os parâmetros:
                    W1 -- matriz de pesos no formato (n_h, n_x)
                    b1 -- vetor bias no formato (n_h, 1)
                    W2 -- matriz de pesos no formato (n_y, n_h)
                    b2 -- vetor bias no formato (n_y, 1)
    """

    np.random.seed(1)

    ### INICIE O SEU CÓDIGO AQUI ### (≈ 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))
    ### TÉRMINO DO CÓDIGO ###

    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 [42]:
```
parametros = inicializar_parametros(3,2,1)
print("W1 = " + str(parametros["W1"]))
print("b1 = " + str(parametros["b1"]))
print("W2 = " + str(parametros["W2"]))
print("b2 = " + str(parametros["b2"]))
```

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

Saída Esperada:

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

3.2 - Rede Neural com L camadas escondidas

A inicialização para uma rede profunda (com L camadas escondidas) é mais complicada porque existem mais matrizes de pesos e vetores de bias. Quando completar a função `inicializar_parametros_deep`, você deve ter certeza que as dimensões estão corretas entre cada camada. Lembrando que $n^{[l]}$ é o número de unidades na camada l. Por exemplo, se o tamanho da camada de entrada X é $(12288, 209)$ (com $m = 209$ exemplos) então:

	Formato de W	**Formato de b**	**Ativação**	**Formato da Ativação**
Camada 1	$(n^{[1]}, 12288)$	$(n^{[1]}, 1)$	$Z^{[1]} = W^{[1]}X + b^{[1]}$	$(n^{[1]}, 209)$
Camada 2	$(n^{[2]}, n^{[1]})$	$(n^{[2]}, 1)$	$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$	$(n^{[2]}, 209)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
Camada L-1	$(n^{[L-1]}, n^{[L-2]})$	$(n^{[L-1]}, 1)$	$Z^{[L-1]} = W^{[L-1]}A^{[L-2]} + b^{[L-1]}$	$(n^{[L-1]}, 209)$
Camada L	$(n^{[L]}, n^{[L-1]})$	$(n^{[L]}, 1)$	$Z^{[L]} = W^{[L]}A^{[L-1]} + b^{[L]}$	$(n^{[L]}, 209)$

Lembre-se que quando computamos $WX + b$ em python, é utilizado broadcasting. Por exemplo, se:

$$W = \begin{bmatrix} j & k & l \\ m & n & o \\ p & q & r \end{bmatrix} \quad X = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \quad b = \begin{bmatrix} s \\ t \\ u \end{bmatrix} \tag{1}$$

Então $WX + b$ será:

$$WX + b = \begin{bmatrix} (ja + kd + lg) + s & (jb + ke + lh) + s & (jc + kf + li) + s \\ (ma + nd + og) + t & (mb + ne + oh) + t & (mc + nf + oi) + t \\ (pa + qd + rg) + u & (pb + qe + rh) + u & (pc + qf + ri) + u \end{bmatrix} \tag{2}$$

Exercício: Implemente a função de inicialização para Rede Neural com L camadas.

Instruções:

- A estrutura do modelo é *[LINEAR -> RELU]* × *(L-1) -> LINEAR -> SIGMOID*. Portanto, ela possui $L-1$ camadas usando a função de ativação ReLU seguido por uma camada de saída com a função de ativação sigmoid.
- Use a inicialização aleatória para a matriz de pesos. Utilize `np.random.rand(shape) * 0.01`.
- Use a inicialização com zeros para os vetores de bias. Utilize `np.zeros(shape)`.
- Vamos armazenar $n^{[l]}$, o número de nós em camadas diferentes, na variável `layer_dims`. Por exemplo, a `layer_dims` para o modelo "Classificação de Dados Planares" da última tarefa possuia [2,4,1]: Existiam 2 entradas, uma camada escondida com 4 nós, e um único nó na camada de saída. Isto significa que `W1` tem o formato (4,2), `b1` tem o formato (4,1), `W2` tem o formato (1,4) e `b2` tem o formato (1,1). Agora temos que generalizar isto para L camadas!
- Como exemplo, considere que $L = 1$ (uma rede neural com uma única camada escondida). isto deve inspirá-lo a implementar um caso geral (Rede Neural com L-camadas escondidas).

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

In [54]:
```
# FUNÇÃO DE AVALIAÇÃO: inicializar_parametros_deep

def inicializar_parametros_deep(layer_dims):
    """
    Argumentos:
    layer_dims -- array python (lista) contendo as dimensões de cada cam
ada da rede neural

    Retorna:
    parametros -- dicionário python contendo os parâmetros "W1", "b1", .
.., "WL", "bL":
                    Wl -- matriz de pesos no formato (layer_dims[l], lay
er_dims[l-1])
                    bl -- vetor de bias no formato (layer_dims[l], 1)
    """

    np.random.seed(3)
    parametros = {}
    L = len(layer_dims)            # número de camadas na rede neural

    for l in range(1, L):
        ### INICIE O SEU CÓDIGO AQUI ### (≈ 2 linhas de código)
        parametros['W' + str(l)] = np.random.randn(layer_dims[l], layer_
dims[l -1]) * 0.01
        parametros['b' + str(l)] = np.zeros((layer_dims[l], 1))

        ### TÉRMINO DO CÓDIGO ###

        assert(parametros['W' + str(l)].shape == (layer_dims[l], layer_d
ims[l-1]))
        assert(parametros['b' + str(l)].shape == (layer_dims[l], 1))


    return parametros
```

In [55]:
```
parameters = inicializar_parametros_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.]]
```

Saída Esperada:

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 - Módulo de propagação para frente

4.1 - Para frente - Linear

Agora que os parâmetros foram inicializados podemos implementa o módulo de propagação para frente. Implemente algumas funções básicas que serão utilizadas quando o modelo for implementado. Serão implementadas 3 funções, nesta ordem:

- LINEAR
- LINEAR -> ACTIVATION onde ACTIVATION será ou a ReLU ou a Sigmoid.
- [LINEAR -> RELU] ✕ (L-1) -> LINEAR -> SIGMOID (em todo o modelo)

O modulo de propagação linear para frente (vetorização sobre todos os exemplos) executa as seguintes equações:

$$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]} \tag{3}$$

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

Exercício: Construa a parte linear da propagação para frente.

Lembre-se: A representação matemática desta unidade é $Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$. A função `np.dot()` pode ser útil. Se as dimensões não baterem, imprima `W.shape` para verificar se as dimensões estão corretas.

In [64]:
```
# FUNÇÃO DE AVALIAÇÃO: para_frente_linear

def para_frente_linear(A, W, b):
    """
    Implemente a parte linear da propagação para frente.

    Argumentos:
    A -- ativações da camada anterior (ou dados de entrada): (tamanho da camada anterior, número de exemplos)
    W -- matriz de pesos: um array numpy array no formato (tamanho da camada atual, tamanho da camada anterior)
    b -- vetor bias, array numpy no formato (tamanho da camada corrente, 1)

    Retorna:
    Z -- a entrada da função de ativação, também chamada de parâmetro de pré ativação.
    cache -- um dicionário python contendo "A", "W" e "b" ; armazenados para computar a propagação para trás eficientemente.
    """

    ### INICIE O SEU CÓDIGO AQUI ### (≈ 1 linha de código)
    Z = np.dot(W, A) + b
    ### TÉRMINO DO CÓDIGO ###

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

    return Z, cache
```

In [65]:
```
A, W, b = linear_forward_test_case()

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

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

Saída Esperada:

Z	[[3.26295337 -1.23429987]]

4.2 - Para frente Linear-Ativação

Neste serão utilizadas duas funções de ativação:

- **Sigmoid**: $\sigma(Z) = \sigma(WA + b) = \frac{1}{1+e^{-(WA+b)}}$. A função `sigmoid` foi fornecida. Esta função retorna **dois** itens: o valor de ativação "a" e a "`cache`" que contém "Z" (o valor utilizado na propagação para trás correspondente). Para utilizar a função sigmoid faça a seguinte chamada:

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

- **ReLU**: A expressão matemática da ReLu é $A = RELU(Z) = max(0, Z)$. A função `relu` também é fornecida. Esta função também retorna **dois** itens: o valor de ativação "A" e a "`cache`" que contém "Z" (o valor utilizado na propagação para trás correspondente). Para utilizar a função ReLu faça a seguinte chamada:

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

Para uma maior conveniência iremos agrupar duas funções (Linear e Ativação) em apenas uma função (LINEAR->ATIVAÇÃO). Portanto, você implementará a função que executa a etapa LINEAR para frente seguida da etapa ATIVAÇÃO para frente.

Exercício: Implemente a propagação para frente da camada *LINEAR->ATIVAÇÃO*. A relação matemática é: $A^{[l]} = g(Z^{[l]}) = g(W^{[l]}A^{[l-1]} + b^{[l]})$ onde a ativação "g" pode ser sigmoid() ou relu(). Use a para_frente_linear() e a função de ativação correta.

In [86]:
```
# FUNÇÃO DE AVALIAÇÃO: para_frente_linear_ativacao

def para_frente_linear_ativacao(A_prev, W, b, ativacao):
    """
    Implemente a propagação para frente para a camada LINEAR->ATIVAÇÃO

    Argumentos:
    A_prev -- ativações da camada anterior (ou dados de entrada): (taman
ho da camada anterior, numero de exemplos)
    W -- matriz de pesos: array numpy no formato (tamanho da camada atua
l, tamanho da camada anterior)
    b -- vetor de bias, array numpy no formato (tamanho da camada atual,
1)
    ativacao -- the ativacao para ser utilizada nesta camada, armazenada
como testo: "sigmoid" ou "relu"

    Retorna:
    A -- a saída da função de ativação, também chamado de valor de pós a
tivação.
    cache -- um dicionário python contendo "cache_linear" e "cache_ativa
cao";
             armazenado para computar a propagação para trás de forma ef
iciente.
    """
    Z = np.dot(W, A_prev) + b
    activation_cache = None
    linear_cache= None

    if ativacao == "sigmoid":
        # Entrada: "A_prev, W, b". Saida: "A, activation_cache".
        ### INICIE O SEU CÓDIGO AQUI ### (≈ 2 lines of code)
        A, activation_cache = sigmoid(Z)

        ### TÉRMINO DO CÓDIGO ###

    elif ativacao == "relu":
        # Entrada: "A_prev, W, b". Saída: "A, activation_cache".
        ### INICIE O SEU CÓDIGO AQUI ### (≈ 2 lines of code)
        A, linear_cache = relu(Z)

        ### TÉRMINO DO CÓDIGO ###

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

    return A, cache
```

In [87]:
```
A_prev, W, b = linear_activation_forward_test_case()

A, linear_activation_cache = para_frente_linear_ativacao(A_prev, W, b, a
tivacao = "sigmoid")
print("Com sigmoid: A = " + str(A))

A, linear_activation_cache = para_frente_linear_ativacao(A_prev, W, b, a
tivacao = "relu")
print("Com ReLU: A = " + str(A))
```

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

Saída Esperada:

**Com sigmoid: A **	[[0.96890023 0.11013289]]
**Com ReLU: A **	[[3.43896131 0.]]

Nota: Em deep learning, a computação da camada "[LINEAR->ATIVAÇÃO]" é contada como uma camada única e não duas camadas da rede neural.

d) Modelo L-camadas

Para uma maior conveniência, quando implementarmos a Rede Neural com L camadas, será necessário uma função que replique a função anterior (`para_frente_linear_ativacao` com RELU) $L-1$ vezes, seguido de uma chamada para `para_frente_linear_ativacao` com SIGMOID.

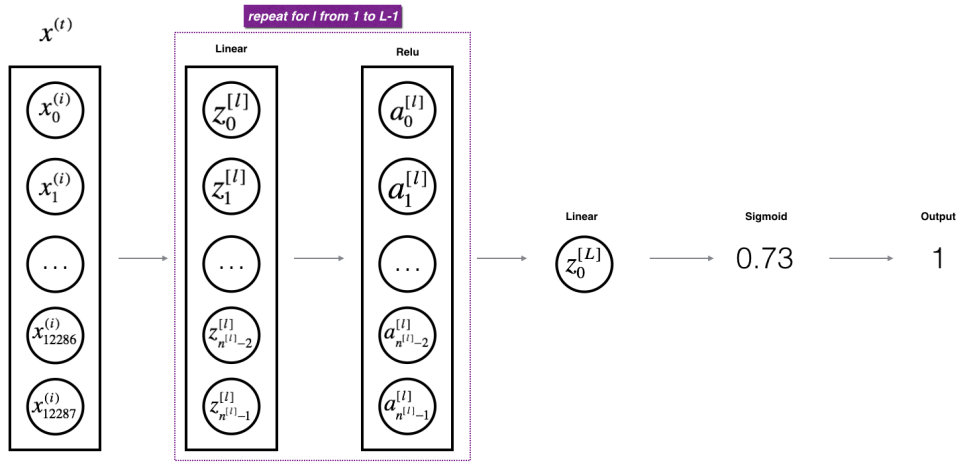

Figura 2 : modelo *[LINEAR -> RELU] ✕ (L-1) -> LINEAR -> SIGMOID*

Exercício: Implemente a propagação para frente do modelo acima.

Instrução: No código abaixo, a variável AL representará $A^{[L]} = \sigma(Z^{[L]}) = \sigma(W^{[L]}A^{[L-1]} + b^{[L]})$. (Isto, as vezes, é chamado também de `Yhat`, isto é $\hat{Y}$.)

Dicas:

- Use as funções que você já implementou
- Use um for loop para replicar [LINEAR->RELU] (L-1) vezes
- Não se esqueça de armazenar os resultados na lista "caches". Para adicionar um novo valor `c` na `lista`, utilize `list.append(c)`.

In [135]:
```
# FUNÇÃO DE AVALIAÇÃO: modelo_para_frente_L

def modelo_para_frente_L(X, parameters):
    """
    Implemente a propagação para frente da computação [LINEAR->RELU]*(L-
1)->LINEAR->SIGMOID

    Argumentos:
    X -- dados, um array numpy array no formato (tamanho da entrada, num
ero de exemplos)
    parameters -- saída da função inicializar_parametros_deep()

    Retorna:
    AL -- último valor de pós ativação
    caches -- lista das caches contendo:
                cada uma das cache da função para_frente_linear_relu() (
existem L-1 delas, indexadas de 0 até L-2)
                a cache da para_frente_linear_sigmoid() (existe apenas u
ma, indexada L-1)
    """
    caches = []
    A = X
    L = len(parameters) // 2                  # numero de camadas na red
e neural
    # Implemente [LINEAR -> RELU]*(L-1). Adicione a "cache" para a lista
"caches".
    for l in range(1, L + 1):
        A_prev = A
        ### INICIE O SEU CÓDIGO AQUI ### (≈ 2 linhas de código)
        W = parameters['W' + str(l)]
        b = parameters['b' + str(l)]
        Z = np.dot(W, A_prev) + b
        A, linear_cache = relu(Z)
        caches.append(linear_cache)
        ### TÉRMINO DO CÓDIGO ###

    # Implemente LINEAR -> SIGMOID. Adicione a "cache" para a lista "cac
hes".
    ### INICIE O SEU CÓDIGO AQUI ### (≈ 2 lines of code)

    Z = np.dot(W, A_prev) + b
    AL, sig_cache = sigmoid(Z)
    ### TÉRMINO DO CÓDIGO ###

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

    return AL, caches
```

In [136]:
```
X, parameters = L_model_forward_test_case_2hidden()
AL, caches = modelo_para_frente_L(X, parameters)
print("AL = " + str(AL))
print("Tamanho da lista caches = " + str(len(caches)))
```

```
AL = [[0.03921668 0.70498921 0.19734387 0.04728177]]
Tamanho da lista caches = 3
```

Saída Esperada:

AL	[[0.03921668 0.70498921 0.19734387 0.04728177]]
**Tamanho da lista caches **	3

Muito bem! Agora você tem uma propagação para frente completa que recebe como entrada X e retorna como saída um vetor linha $A^{[L]}$ contendo as previsões. Ela também armazena os valores intermediários em "caches". Usando $A^{[L]}$, você pode computar o custo das previsões.

5 - Função Custo

Vamos implementar a propagação para frente e para trás. É necessário determinar o custo, porque é preciso verificar se o modelo está aprendendo.

Exercício: Compute o custo de entropia cruzada J, utilizando a seguinte fórmula:

$$-\frac{1}{m}\sum_{i=1}^{m}(y^{(i)}\log\left(a^{[L](i)}\right) + (1-y^{(i)})\log\left(1-a^{[L](i)}\right)) \tag{4}$$

In [149]:
```
# FUNÇÃO DE AVALIAÇÃO: compute_custo

def compute_custo(AL, Y):
    """
    Implemente a função de custo definida pela equação (4).

    Argumentos:
    AL -- vetor de probabilidade correspondente as previsões, formato (1
, número de exemplos)
    Y -- vetor com a classificação correta (por exemplo: 0 se for não-ga
to, 1 se for gato), formato (1, numero de exemplos)

    Retorna:
    custo -- custo por entropia cruzada
    """

    m = Y.shape[1]

    # Compute a perda entre aL e y.
    ### INICIE O SEU CÓDIGO AQUI ### (≈ 1 linha de código)
    cost = (-1 / m) * np.sum((Y * np.log(AL)) + ((1 - Y) * (np.log(1 - A
L))))
    ### TÉRMINO DO CÓDIGO ###
    cost = np.squeeze(cost)      # certifique-se que o formato do custo
é aquele esperado (e.g. torna [[17]] em 17).
    assert(cost.shape == ())

    return cost
```

In [150]:
```
Y, AL = compute_cost_test_case()

print("custo = " + str(compute_custo(AL, Y)))
```

```
custo = 0.41493159961539694
```

Saída Esperada:

custo	0.41493159961539694

6 - Modulo de propagação para trás

Como na propagação para frente, iremos implementar funções auxiliares para a propagação para trás. Lembre-se que a propagação para trás é utilizada para calcular o gradiente da função de perda com relação aos parâmetros.

Lembre-se:

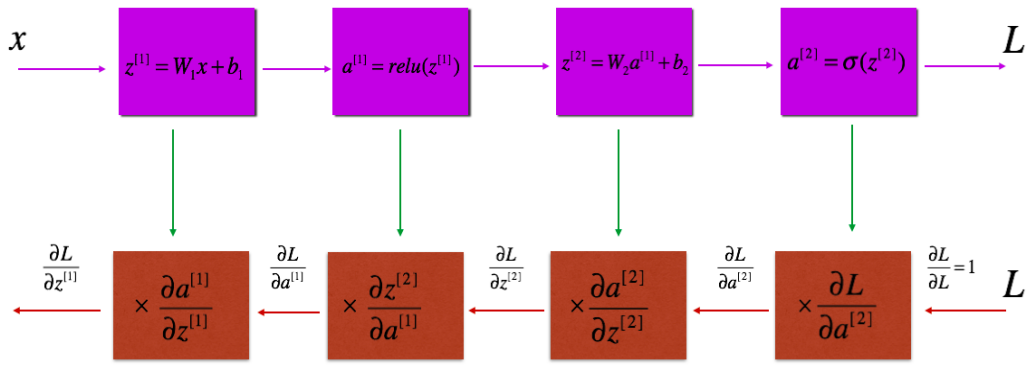

Figura 3 : Propagação para frente e para trás para *LINEAR->RELU->LINEAR->SIGMOID*
Os blocos roxos representam a propagação para frente, e os blocos vermelhos representam a propagação para trás.

Agora, de forma similar a propagação para frente, você irá construir a propagação para trás em três etapas:

- LINEAR para trás
- LINEAR -> ATIVAÇÃO para trás onde ATIVAÇÃO determina a derivativa da ativação da ReLu ou da sigmoid
- [LINEAR -> RELU] ✕ (L-1) -> LINEAR -> SIGMOID para trás (modelo completo)

6.1 - Para trás Linear

Para a camada l, a parte linear é: $Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$ (seguido por uma ativação).

Suponha que já tenha sido calculado a derivativa $dZ^{[l]} = \frac{\partial \mathcal{L}}{\partial Z^{[l]}}$. Você deseja obter $(dW^{[l]}, db^{[l]} dA^{[l-1]})$.

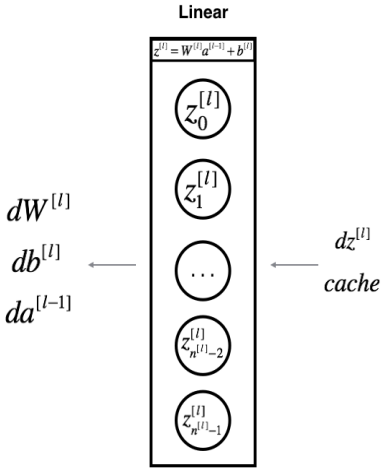

Figura 4

As três saídas $(dW^{[l]}, db^{[l]}, dA^{[l]})$ são computadas usando a entrada $dZ^{[l]}$. Abaixo estão as fórmulas que você precisa:

$$dW^{[l]} = \frac{\partial \mathcal{L}}{\partial W^{[l]}} = \frac{1}{m} dZ^{[l]} A^{[l-1]T} \tag{5}$$

$$db^{[l]} = \frac{\partial \mathcal{L}}{\partial b^{[l]}} = \frac{1}{m} \sum_{i=1}^{m} dZ^{[l](i)} \tag{6}$$

$$dA^{[l-1]} = \frac{\partial \mathcal{L}}{\partial A^{[l-1]}} = W^{[l]T} dZ^{[l]} \tag{7}$$

Exercício: Use as 3 fórmulas acima para implementar para_tras_linear().

In [354]:
```
# FUNÇÃO DE AVALIAÇÃO: para_tras_linear

def para_tras_linear(dZ, cache):
    """
    Implemente a porção linear da propagação para trás em uma única cama
da (camada l)

    Argumentos:
    dZ -- Gradiente do custo com relação à saída linear (d camada atual
l)
    cache -- tupla de valores (A_prev, W, b) que vem da propagação para
frente na camada atual.

    Retorna:
    dA_prev -- Gradiente do custo com relação a ativação (da camada ante
rior l-1), no mesmo formato que A_prev
    dW -- Gradiente do custo com relação a W (camada corrente l), mesmo
formato que W.
    db -- Gradiente do custo com relação a b (camada corrente l), mesmo
formato de b.
    """
    A_prev, W, b = cache
    m = A_prev.shape[1]

    ### INICIE O SEU CÓDIGO AQUI ### (≈ 3 linhas de código)
    dW = (1 / m) * np.dot(dZ, A_prev.T)
    db = 1./m * np.sum(dZ, axis = 1, keepdims = True)
    dA = np.dot(W.T,dZ)

    ### TÉRMINO DO CÓDIGO ###
    assert (dA.shape == A_prev.shape)
    assert (dW.shape == W.shape)
    assert (db.shape == b.shape)

    return dA, dW, db
```

In [355]:
```
# Ajuste de valores de teste
dZ, linear_cache = linear_backward_test_case()

dA_prev, dW, db = para_tras_linear(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]]
```

Saída Esperada:

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 - Para trás Linear-Ativação

Agora, iremos criar uma função que junta as duas funções auxiliares: **para_tras_linear** e a etapa para trás da ativação **para_tras_linear_ativacao**.

Para ajudar a implementar `para_tras_linear_ativacao`, são fornecidas duas funções para trás:

- **para_tras_sigmoid**: Implementa a propagação para trás para nós com função de ativação SIGMOID. Sua chamada é feita da seguinte forma:

  `dZ = sigmoid_backward(dA, activation_cache)`

- **para_tras_relu**: Implementa a propagação para trás para os nós que utilizam RELU. Sua chamada é feita da seguinte forma:

  `dZ = relu_backward(dA, activation_cache)`

Se $g(.)$ é a função de ativação, `sigmoid_backward` e `relu_backward` computam

$$dZ^{[l]} = dA^{[l]} * g'(Z^{[l]}) \tag{8}$$

.

Exercício: Implemente a propagação para trás para a camada *LINEAR->ATIVAÇÃO*.

In [321]:
```
# FUNÇÃO DE AVALIAÇÃO: para_tras_linear_ativacao

def para_tras_linear_ativacao(dA, cache, activation):
    """
    Implemente a propagação para trás para a camada LINEAR->ATIVAÇÃO.

    Argumentos:
    dA -- gradiente de pós-ativação para a camada atual l
    cache -- tupla de valores (linear_cache, activation_cache) usados pa
ra determinar a propagação para trás de forma eficiente.
    activation -- a ativação a ser utilizada nesta camada, armazenada co
mo texto: "sigmoid" ou "relu"

    Retorna:
    dA_prev -- Gradiente do custo com relação a ativação (da camada ante
rior l-1), mesmo formato que A_prev
    dW -- Gradiente do custo com relação a W (camada atual l), mesmo for
mato que W.
    db -- Gradiente de custo com relação a b (camada atual l), mesmo for
mato que b.
    """
    linear_cache, activation_cache = cache

    if activation == "relu":
        ### INICIE O SEU CÓDIGO AQUI ### (≈ 2 linhas de código)
        dZ = relu_backward(dA, activation_cache)
        dA_prev, dW, db = para_tras_linear(dZ, linear_cache)
        ### TÉRMINO DO CÓDIGO ###

    elif activation == "sigmoid":
        ### INICIE O SEU CÓDIGO AQUI ### (≈ 2 linhas de código)
        dZ = sigmoid_backward(dA, activation_cache)
        dA_prev, dW, db = para_tras_linear(dZ, linear_cache)
        ### TÉRMINO DO CÓDIGO ###

    return dA_prev, dW, db
```

In [322]:
```
AL, linear_activation_cache = linear_activation_backward_test_case()

dA_prev, dW, db = para_tras_linear_ativacao(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 = para_tras_linear_ativacao(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]]
```

Saída esperada para 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]]

Saída esperada com relu:

dA_prev	[[0.44090989 0.] [0.37883606 0.] [-0.2298228 0.]]
dW	[[0.44513824 0.37371418 -0.10478989]]
db	[[-0.20837892]]

6.3 - Modelo para trás L

Agora iremos implementar a função de propagação para trás para a rede neural toda. Lembre-se que quando implementamos o `modelo_para_frente_L`, em cada interação a cache era armazenada com os valores de (X,W,b, e z). Na propagação para trás estes valores serão utilizados para determinar os gradientes. Então, no `modelo_para_tras_L` iremos interagir por todas as camadas escondidas de trás para frente, começando pela camada L. Em cada etapa, voc6e utilizará os valores armazenados para a camada l para propagar para trás através da camada l. A Figura 5 mostra o passo para trás.

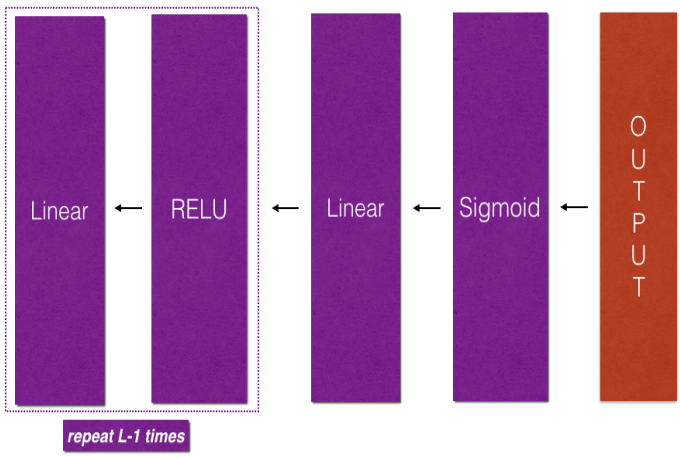

Figura 5 : Passo para trás

Inicialização da propagação para trás: Para propagar para trás através desta rede sabemos que a saída é $A^{[L]} = \sigma(Z^{[L]})$. Seu código deve computar `dAL` $= \frac{\partial \mathcal{L}}{\partial A^{[L]}}$. Para fazer isto utilize a fórmula (derivada utilizando cálculo, o qual você não precisa compreender totalmente):

```
dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL)) # derivativa do custo com relação a AL
```

Você pode então utilizar o gradiente da pós ativação `dAL` para continuar indo para trás. Como visto na Figura 5, você pode alimentar a função para trás implementada LINEAR->SIGMOID com `dAL` (que utilizará os valores armazenados na cache pelo modelo_para_frente_L). Após isto você deverá utilizar um `for` para interagir através das outras camadas utilizando a função para trás LINEAR->RELU. Você deve armazenar cada dA, dW, e db no dicionário grads. Para fazer isto, use a expressão:

$$grads[" \, dW \, " + str(l)] = dW^{[l]} \tag{9}$$

Por exemplo, para $l = 3$, $dW^{[l]}$ estaria armazendo em `grads["dW3"]`.

Exercício: Implemente a propagação para trás para o modelo *[LINEAR->RELU]* × *(L-1)* -> *LINEAR* -> *SIGMOID*.

In [356]:
```
# FUNÇÃO DE AVALIAÇÃO: modelo_para_tras_L

def modelo_para_tras_L(AL, Y, caches):
    """
    Implemente a propagação para trás para o grupo [LINEAR->RELU] * (L-1
) -> LINEAR -> SIGMOID.

    Argumentos:
    AL -- vetor de probabilidade, saída da propagação para frente (model
o_para_frente_L())
    Y -- vetor de classificação correta (contendo 0 se nao-gato, 1 se ga
to)
    caches -- lista de caches contendo:
                cada cache obtida na para_frente_linear_ativacao() com "
relu" (isto é, caches[l], para l no range(L-1) i.e l = 0...L-2)
                a cache da para_frente_linear_ativacao() com "sigmoid" (
isto é, caches[L-1])

    Retorna:
    grads -- Um dicionário com os gradientes
             grads["dA" + str(l)] = ...
             grads["dW" + str(l)] = ...
             grads["db" + str(l)] = ...
    """
    grads = {}
    L = len(caches) # o número de camadas
    m = AL.shape[1]
    Y = Y.reshape(AL.shape) # após esta linha, Y tem o mesmo formato de
AL

    # Inicializando a propagação para trás
    ### INICIE O SEU CÓDIGO AQUI ### (1 linha de código)
    dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL))
    ### TÉRMINO DO CÓDIGO ###

    # L-ésima camada (SIGMOID -> LINEAR), gradientes. Entrada: "AL, Y, c
aches". Saída: "grads["dAL"], grads["dWL"], grads["dbL"]
    ### INICIE O SEU CÓDIGO AQUI ### (approx. 2 linhas)

    grads["dA" + str(L)], grads["dW" + str(L)], grads["db" + str(L)] = p
ara_tras_linear_ativacao(dAL, caches[L -1], activation = "sigmoid")

    ### TÉRMINO DO CÓDIGO ###

    for l in reversed(range(L-1)):
        # l-ésima camada: (RELU -> LINEAR), gradientes.
        # Entrada: "grads["dA" + str(l + 2)], caches". Saída: "grads["dA
" + str(l + 1)] , grads["dW" + str(l + 1)] , grads["db" + str(l + 1)]
        ### INICIE O SEU CÓDIGO AQUI ### (approx. 5 linhas)
        dA_prev, dW, db = para_tras_linear_ativacao(grads["dA" + str(l +
2)], caches[l], activation = "relu")
        grads["dA" + str(l + 1)] = dA_prev
        grads["dW" + str(l + 1)] = dW
        grads["db" + str(l + 1)] = db

        ### TÉRMINO DO CÓDIGO ###

    return grads
```

In [357]:
```
AL, Y_assess, caches = L_model_backward_test_case()
grads = modelo_para_tras_L(AL, Y_assess, caches)
print_grads(grads)
```

```
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 = [[ 0.12913162 -0.44014127]
 [-0.14175655  0.48317296]
 [ 0.01663708 -0.05670698]]
```

Saída Esperada

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	[[0.12913162 -0.44014127] [-0.14175655 0.48317296] [0.01663708 -0.05670698]]

6.4 - Atualização de Parâmetros

Nesta parte da tarefa iremos atualizar os parâmetros do modelo utilizando gradiente descendente:

$$W^{[l]} = W^{[l]} - \alpha \, dW^{[l]} \tag{10}$$
$$b^{[l]} = b^{[l]} - \alpha \, db^{[l]} \tag{11}$$

onde α é a taxa de aprendizado. Após computar os parâmetros atualizados armazene estes parâmetros no dicionário de parâmetros.

Exercício: Implemente `atualize_parametros()` para atualizar os parâmetros utilizando gradiente descendente.

Instruções: Atualize os parâmetros utilizando gradiente descendente para cada $W^{[l]}$ e $b^{[l]}$ for $l = 1, 2, \ldots, L$.

In [378]:
```
# FUNÇÃO DE AVALIAÇÃO: atualizacao_parametros

def atualizacao_parametros(parameters, grads, learning_rate):
    """
    Atualize os parametros utilizando gradiente descendente.

    Argumentos:
    parameters -- dicionario python contendo os parâmetros.
    grads -- dicionário python contendo os gradientes, saída do modelo_p
ara_tras_L.
    learning_rate -- taxa de aprendizado

    Retorna:
    parameters -- dicionario python contendo os parâmetros atualizados.
                  parameters["W" + str(l)] = ...
                  parameters["b" + str(l)] = ...
    """

    L = len(parameters) // 2 # número de camadas na rede neural
    # Regra de atualizacao para cada parâmetro. Utilize um for loop.
    ### INICIE O SEU CÓDIGO AQUI ### (≈ 3 linhas de código)
    for l in range(L):
        l_plus = l + 1
        parameters["W" + str(l_plus)] = parameters["W" + str(l_plus)] -
learning_rate * grads["dW" + str(l_plus)]
        parameters["b" + str(l_plus)] = parameters["b" + str(l_plus)] -
learning_rate * grads["db" + str(l_plus)]

    ### TÉRMINO DO CÓDIGO ###
    return parameters
```

In [379]:
```
parameters, grads = update_parameters_test_case()
parameters = atualizacao_parametros(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]]
```

Saída Esperada:

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 - Conclusão

Parabéns ao completar esta tarefa. Você implementou todas as funções necessárias para construir uma rede neural profunda!

Sabemos que esta era uma tarefa longa porém, olhando para frente, isto apenas vai melhorar. A proxima tarefa deve ser mais fácil.

Na próxima tarefa você irá colocar tudo junto e construir dois modelos:

- Uma rede neural com duas camadas escondidas.
- Uma rede neural com L camadas.

Você irá utilizar estes modelos para classificar imagens entre gatos e não-gatos. Divirta-se!