Lab 3 - BCC406

REDES NEURAIS E APRENDIZAGEM EM PROFUNDIDADE

Construindo uma rede neural

Prof. Eduardo e Prof. Pedro Silva

Data da entrega : 15/04

- Complete o código (marcado com ToDo) e quando requisitado, escreva textos diretamente nos notebooks. Onde tiver *None*, substitua pelo seu código.
- Execute todo notebook e salve tudo em um PDF **nomeado** como "NomeSobrenome-Lab3.pdf"
- Envie o PDF para pelo FORM

Parte 1 - Rede neural do zero: passo a passo (10pt)

+ Código + Texto

Notação:

- Sobrescrito índice $[l]$ indica os valores associados a l-ésima camada.
  - **Exemplo:** $a^{[l]}$ é a ativação da l-ésima camada.
- Sobrescrito índice (i) indica os valores associados ao i-ésima exemplo.
  - **Exemplo:** $x^{(i)}$ é o i-ésima exemplo de treinamento.
- Subescrito índice j indica a j-ésima entrada de um vetor.
  - **Exemplo:** $a_j^{[l]}$ indica a j-ésima entrada da ativação da l-ésima camada.

1 - Importação dos pacotes

Primeiro, vamos executar a célula abaixo para importar todos os pacotes que precisaremos.

- numpy é o pacote fundamental para a computação científica com Python.
- h5py é um pacote comum para interagir com um conjunto de dados armazenado em um arquivo H5.
- matplotlib é uma biblioteca famosa para plotar gráficos em Python.
- PIL e scipy são usados aqui para testar seu modelo.
- dnn_utils fornece algumas funções necessárias para este notebook.

- testCases fornece alguns casos de teste para avaliar as funções.
- np.random.seed (1) é usado para manter todas as chamadas de funções aleatórias.

```
# Para Google Colab: Você vai precisar fazer o upload dos arquivos no seu drive e montá-lo
# não se esqueça de ajustar o path para o seu drive
from google.colab import drive
drive.mount('/content/drive')
```

```
    Mounted at /content/drive
```

```
# Você vai precisar inserir seu diretório para importar as "bibliotecas próprias" auxiliar
# não se esqueça de ajustar o path para o seu diretório

import sys
sys.path.append('/content/drive/My Drive/')
```

```
import numpy as np
import h5py
import matplotlib.pyplot as plt
# bibliotecas auxiliares (ver testCases_v4a.py e dnn_utils_v2.py)
from testCases_v4a 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 - Esboço das Funções auxiliares

- Inicialização dos parâmetros da rede.
- Implementação da fase forward propagation (roxo na figura abaixo).
  - Complete a parte LINEAR da etapa de forward propagation de uma camada (resultando em $Z^{[l]}$).
  - Fornecemos a função ATIVAÇÃO (relu / sigmóide).
  - Combine os dois passos anteriores em uma nova função de avanço [LINEAR-> ATIVAÇÃO].
  - Empilhe a função de avanço [LINEAR-> RELU] L-1 (para as camadas 1 a L-1) e adicione um [LINEAR-> SIGMOID] no final (para a camada final L). Isso fornece uma nova função L_model_forward.

- Cálculo a função loss.
- Implementação da fase backward propagation (vermelho na figura abaixo).
  - Complete a parte LINEAR da etapa de backward propagation de uma camada.
  - Fornecemos o gradiente da função (relu_backward / sigmoid_backward)
  - Combine as duas etapas anteriores em uma nova função [LINEAR-> ATIVAÇÃO] para trás.
  - Empilhe [LINEAR-> RELU] para trás L-1 vezes e adicione [LINEAR-> SIGMOID] para trás em uma nova função L_model_backward
- Atualização dos parâmetros.

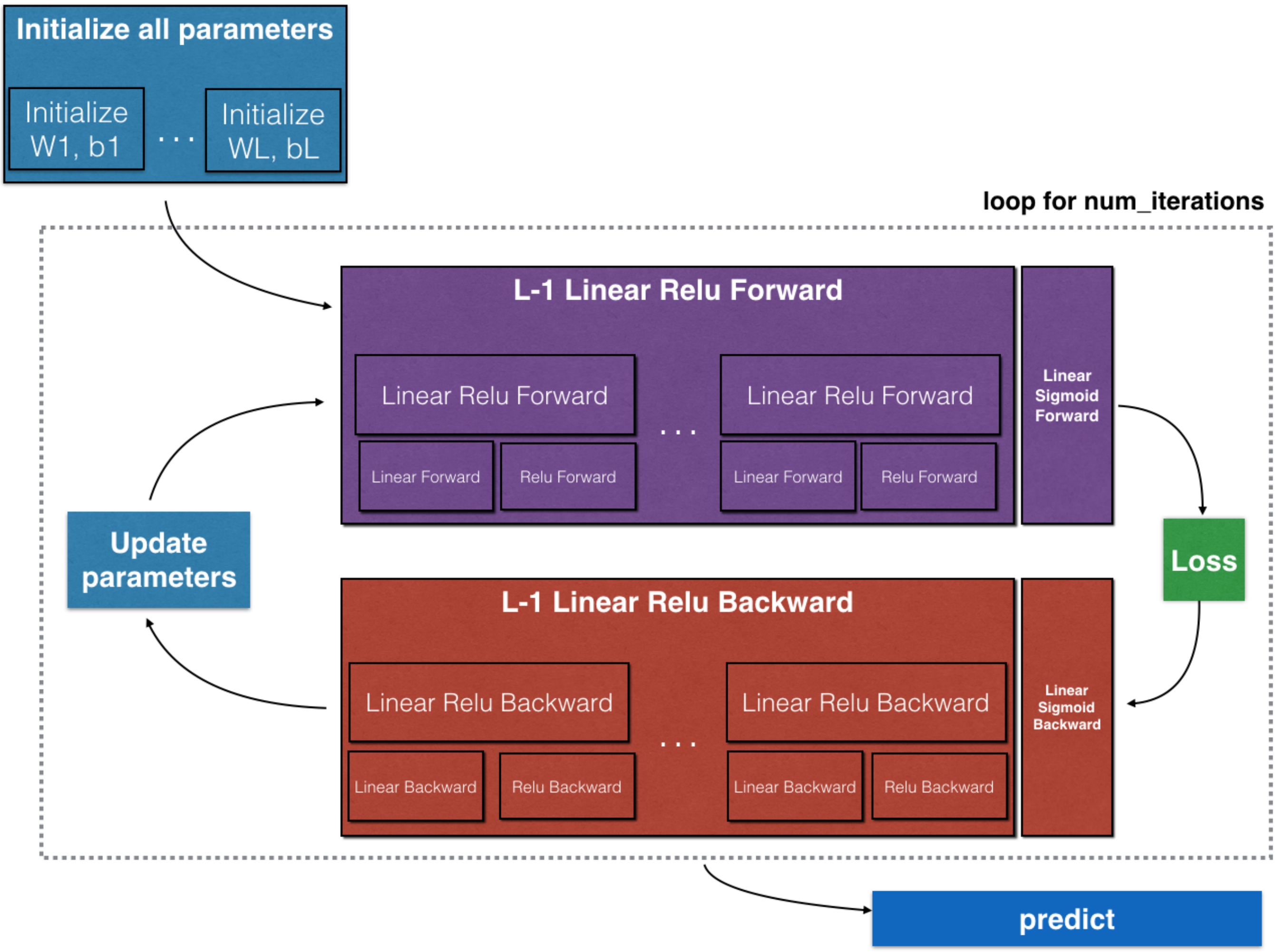

Figura 1

Observe que para todas as etapas forward, existe uma etapa backward correspondente. É por isso que em cada etapa forward você estará armazenando alguns valores em cache. Os valores em cache são úteis para calcular gradientes. Na etapa backward, você usará o cache para calcular os gradientes.

3 - Inicialização (1pt)

A função será usada para inicializar parâmetros para uma rede com L-camadas.

3.1 - Rede Neural com L-camadas

Instruções:

- A estrutura do modelo é * [LINEAR -> RELU] $\times$ (L-1) -> LINEAR -> SIGMOID *. Ou seja, possui (L-1) camadas usando uma função de ativação ReLU seguida por uma camada de saída com uma função de ativação sigmóide.
- Use inicialização aleatória para as matrizes de peso. Use `np.random.randn(shape) * 0,01`.
- Use a inicialização de zeros para os vieses. Use `np.zeros(shape)`.
- Armazenaremos $n^{[l]}$, o número de elementos/neurônios na camada l, em uma variável `camadas_dims`. Por exemplo, `camadas_dims = [2,4,1]` é uma rede com duas entradas, uma camada oculta com 4 unidades/neurônios e uma camada de saída com 1 unidade/neurônio de saída .

```
# Inicialize_parametros

def inicialize_parametros(camadas_dims):
    """
    Entrada:
    camadas_dims -- python array (lista) contendo a dimensão de cada camada da rede


    Saída:
    parametros   -- python dicionario contendo os parametros "W1", "b1", ..., "WL", "bL":
                    Wl -- vetor de pesos com formato (camadas_dims[l], camadas_dims[l-1])
                    bl -- vetor de vies com formato (camadas_dims[l], 1)
    """

    np.random.seed(3)
    parametros = {}
    L = np.size(camadas_dims)            # ToDo: número de camadas da rede

    ### Início do código ###
    for l in range(1, L):
      # dica: itere pelo número de camadas, inicializando pesos e viés de cada camada,
      # e armazenem em parameters (≈ 2 linhas de código)
      parametros['W' + str(l)] = np.random.randn(camadas_dims[l], camadas_dims[l-1]) * 0.0
      parametros['b' + str(l)] = np.zeros((camadas_dims[l], 1)) # ToDo
    ### Fim do código ###

    return parametros
```

Comentários Ao concluir o `inicialize_parametros`, certifique-se de que as dimensões entre cada camada estejam corretas. Lembre-se de que $n^{[l]}$ é o número de unidades na camada l.

Assim, por exemplo, se o tamanho da nossa entrada X for $(12288, 209)$ (com número de exemplos $m = 209$), 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	$(n^{[L]}, n^{[L-1]})$	$(n^{[L]}, 1)$	$Z^{[L]} = W^{[L]}A^{[L-1]} + b^{[L]}$	$(n^{[L]}, 209)$

```
# Teste
parametros = inicialize_parametros([5,4,3])
print("W1 = " + str(parametros["W1"]))
print("b1 = " + str(parametros["b1"]))
print("W2 = " + str(parametros["W2"]))
print("b2 = " + str(parametros["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.]]
```

Valores esperados:

W1	[[0.01788628 0.0043651 0.00096497 -0.01863493 -0.00277388] [-0.00354759 -0.00082741 -0.00627001 -0.00043818 -
b1	[[0.] [0.] [0.] [0.]]
W2	[[-0.01185047 -0.0020565 0.01486148 0.00236716] [-0.01023785 -0.00712993 0.00625245 -0.00160513] [-0.00768836
b2	[[0.] [0.] [0.]]

4 - Fase: Forward propagation (2pt)

Usaremos duas funções:

- LINEAR
- [LINEAR -> RELU] $\times$ (L-1) -> LINEAR -> SIGMOID

4.1 - Linear Forward

A função linear_forward (sobre todos os examples) é definida pela equação:

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

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

Lembrete Lembre-se de que quando calculamos $WX + b$ em python, ele realiza `broadcasting`. Por exemplo, se:

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

Então $WX + b$ será:

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

```
# Função linear_forward

def linear_forward(A, W, b):
    """
    Implementa a parte linear da fase de propogação nas camadas

    Entradas:
    A - dados de entrada da camada atual (ativações da camada anterior): formato (tamanho
    W - matriz de pesos: matriz numpy com formato (tamanho da camada atual, tamanho da cam
    b - vetor de viés, matriz numpy com formato (tamanho da camada atual, 1)

    Saídas:
    Z -- a entrada da função de ativação, também chamada de parâmetro de pré-ativação
    cache - uma tupla python contendo "A", "W" e "b"; (armazenado para usar na fase backwa
    """

    ### Início do código ### (≈ 2 linhas de código)
    Z = np.dot(W,A)+b # dica: use a funçao .dot()
    cache = (A, W, b)
    ### Fim do código ###



    return Z, cache
```

```
# Teste
```

```
A, W, b = linear_forward_test_case()

Z, linear_cache = linear_forward(A, W, b)

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

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

Valores Esperados:

Z [[3.26295337 -1.23429987]]

4.2 - Linear-Ativação Forward

Usaremos duas funções de ativação:

- **Sigmoid**: $\sigma(Z) = \sigma(WA + b) = \frac{1}{1+e^{-(WA+b)}}$. A função `sigmoid`, **retorna dois** itens: o valor de ativação " `a` "e um" `cache` " que contém " `Z` "(necessário para a fase backward correspondente). Para usá-lo, basta chamar:

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

- **ReLU**: A formula é $A = RELU(Z) = max(0, Z)$. A função `relu`, **retorna dois** itens: o valor de ativação " `a` "e um" `cache` " que contém " `Z` "(necessário para a fase backward correspondente). Para usá-lo, basta chamar:

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

Exercício: Implemente a *LINEAR-> ATIVAÇÃO* da camada da fase forward propagation. 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 função linear_forward ().

```
# Função linear_ativacao_forward

def linear_ativacao_forward(A_prev, W, b, ativacao):
    """
    Implementa a *LINEAR-> ATIVAÇÃO* da camada da fase forward propagation

    Entradas:
    A_prev -- dados de entrada da camada atual (ativações da camada anterior): formato (ta
    W - matriz de pesos: matriz numpy com formato (tamanho da camada atual, tamanho da cam
    b - vetor de viés, matriz numpy com formato (tamanho da camada atual, 1)
    ativacao -- "sigmoid" ou "relu"

    Saídas:
    A -- a saída da função de ativação, também chamada de valor da pós-ativação
```

```
    cache -- uma tupla python contendo "linear_cache" e "ativacao_cache";
    (armazenado para usar na fase backward propagation)
    """

    if ativacao == "sigmoid":
        # Entradas: "A_prev, W, b". Saídas: "A, ativacao_cache".
        ### Início do código ###
        # dicas: use sua funcao de propagação e as funções de ativação fornecidas em dnn_u
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = sigmoid(Z)
        ### Fim do código ###

    elif ativacao == "relu":
        # Entradas: "A_prev, W, b". Saídas: "A, ativacao_cache".
        ### Início do código ###
        # dicas: use sua funcao de propagação e as funções de ativação fornecidas em dnn_u
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = relu(Z)
        ### Fim do código ###

    cache= (linear_cache,activation_cache)

    return A, cache


# Teste

A_prev, W, b = linear_activation_forward_test_case()

A, linear_ativacao_cache = linear_ativacao_forward(A_prev, W, b, ativacao = "sigmoid")
print("com sigmoid: A = " + str(A))

A, linear_ativacao_cache = linear_ativacao_forward(A_prev, W, b, ativacao = "relu")
print("com ReLU: A = " + str(A))
```

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

Valores esperados:

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

d) Modelo de L-camadas

Replica a função `linear_ativacao_forward` com RELU $(L-1)$ vezes, depois uma vez `linear_ativacao_forward` com SIGMOID.

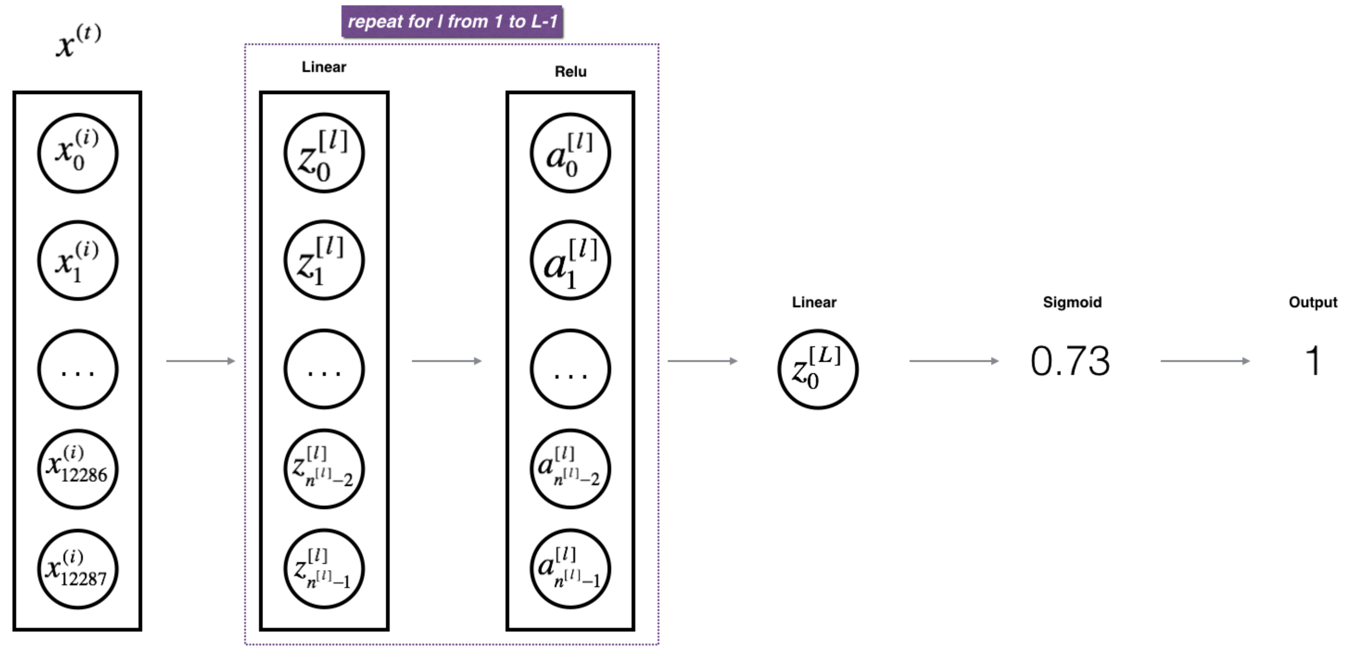

Figura 2 : Esquema do modelo *[LINEAR -> RELU] $\times$ (L-1) -> LINEAR -> SIGMOID*

Instrução: A variável `AL` é $A^{[L]} = \sigma(Z^{[L]}) = \sigma(W^{[L]} A^{[L-1]} + b^{[L]})$. (ativaçnao da última camada, i.e., $\hat{Y}$.)

```
# L_modelo_forward

def L_modelo_forward(X, parametros):
    """
    Implementa a fase forward propagation

    Entradas:
    X -- dados, numpy array de tamanho (input size, number of examples)
    parametros -- parametros iniciais

    Saídas:
    AL -- valor da pós-ativação da última camada
    caches -- lista dos caches contendo:
                todos caches da linear_ativacao_forward() (existem L-1 deles, indexados de
    """

    caches = []
    A = X                  # dados da camada inicial
    L = int((len(parametros))/2)         # números de camadas da rede

    # Implemente [LINEAR -> RELU]*(L-1). Adicione o "linear_cache" para a lista "caches".
    ### Início do código ###
    for l in range(1,L):
        A_prev = A
        A, cache = linear_ativacao_forward(A_prev, parametros["W"+str(l)], parametros["b"+
        caches.append(cache)
    ### Fim do código ###
    # Implement LINEAR -> SIGMOID. Add "cache" to the "caches" list.
```

```
    ### Início do código ###
    AL, cache = linear_ativacao_forward(A, parametros["W"+str(L)], parametros["b"+str(L)],
    #AL=sum(AL)
    caches.append(cache)
    ### Fim do código ###




    return AL, caches
```

```
X, parametros = L_model_forward_test_case_2hidden()
AL, caches = L_modelo_forward(X, parametros)

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
```

Valores Esperados:

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

Usando $A^{[L]}$, você deve calcular o custo da rede.

5 - Função Custo (cross-entropy) (2pt)

Para a fase backward propagation é necessário o cálculo da funcão custo.

Exercício: Use a seguinte função custo:

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

obs.: veja que é a mesma implementada para o Lab1b.

```
# Função custo

def custo(AL, Y):
    """
    Implementa a função custo da rede.

    Entradas:
    AL -- Probabiliade de predição da rede, (1, numero de exemplos)
    Y -- Vetor de rótulos dos exemplos de treinamento  ( 0 se não tem gato, 1 tem gato ),

    Saída:
```

```
    custo -- custo da rede
    """

    m = Y.size # número de exemplos

    # Compute loss from AL and y.
    ###Início do código ### (≈ 1 linha de código)
    custo = -1/m * np.sum(((Y * (np.log(AL))) + ( (1 - Y) * (np.log(1-AL)))))
    ### Fim do código ###

    custo = np.squeeze(custo)      # assegurar o formato experado ( [[17]] para 17).


    return custo

Y, AL = compute_cost_test_case()

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

```
custo = 0.2797765635793422
```

Valores Esperados:

custo	0.2797765635793422

6 - Fase: Backward propagation (2pt)

Com funções auxiliares, a fase back propagation é usada para calcular o gradiente da função loss em relação aos parâmetros.

Lembrete:

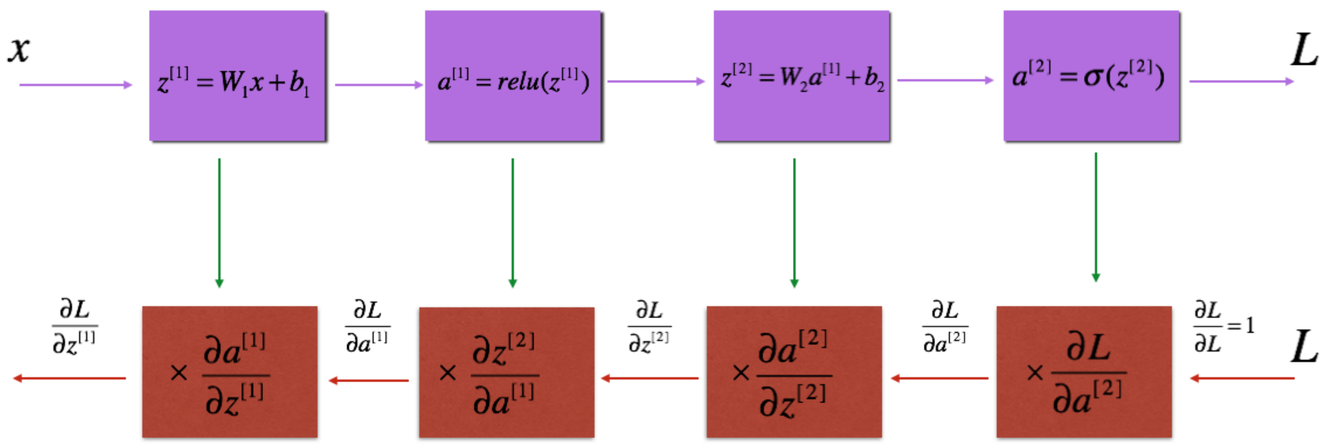

Figura 3 :
Os blocos roxos representam a fase forward propagation, e os vermelhos representam a fase backward propagation.

Usaremos duas funções, igualmente feito na fase forward:

- LINEAR
- [LINEAR -> RELU] $\times$ (L-1) -> LINEAR -> SIGMOID

6.1 - Linear backward

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

Suponha que $dZ^{[l]} = \frac{\partial \mathcal{L}}{\partial Z^{[l]}}$ já foi calculado.

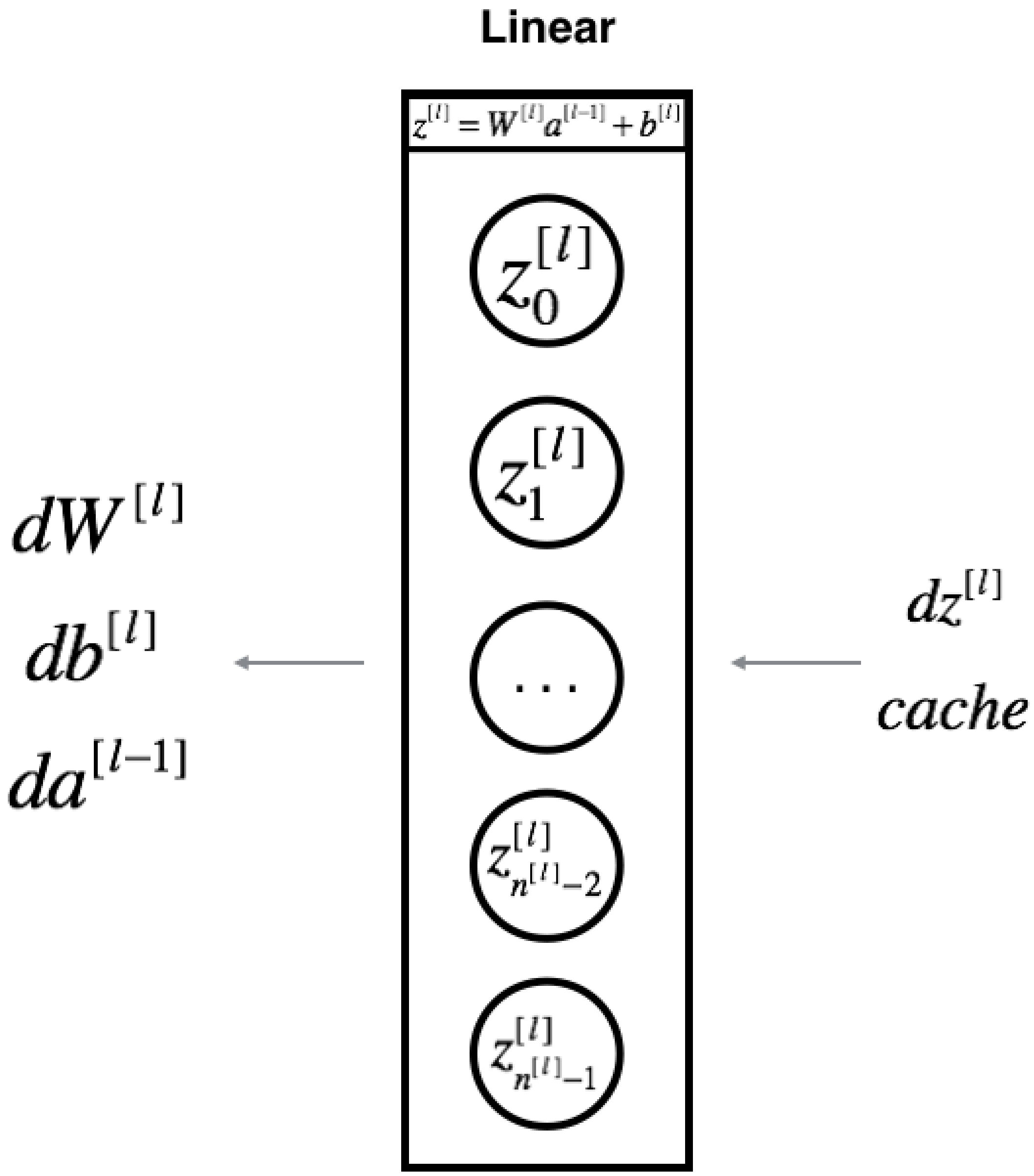

Figura 4

As saídas $(dW^{[l]}, db^{[l]}, dA^{[l-1]})$ são calculadas usando $dZ^{[l]}$:

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

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

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

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

```
# linear_backward

def linear_backward(dZ, cache):
    """
    Implementa a parte linear da fase backward propagation em uma camada l

    Entradas:
    dZ -- gradiente do custo em relação a saída linear da camada l
    cache -- tupla (A_prev, W, b) vindo da forward propagation da camada l

    Saídas:
    dA_prev -- gradiente do custo em relação a ativação da camada l-1,
    dW -- gradiente do custo em relação a W  da camada l,
    db -- gradiente do custo em relação a b,
    """
    A_prev, W, b = cache
    m = A_prev.shape[1]

    #print(A_prev.shape[1])
    #print(dZ.shape[1])

    n=b.size

    db=b

    ### Início do código ###
    dW = 1/m * np.dot(dZ, np.transpose(A_prev))
    for i in range(0,n):
      db[[i]]= 1/m * sum(dZ[i])
    #db = 1/m * sum(dZ)
    dA_prev = np.dot(np.transpose(W), dZ)
    ### Fim do código ###

    print(str(dZ))

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

    return dA_prev, dW, db


# Teste
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))
```

```
[[ 1.62434536 -0.61175641 -0.52817175 -1.07296862]
 [ 0.86540763 -2.3015387   1.74481176 -0.7612069 ]
 [ 0.3190391  -0.24937038  1.46210794 -2.06014071]]
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]]
```

Valores Esperados:

```
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-Ativação backward

A etapa backward para a ativação **`linear_ativacao_backward`**.

Use as funções:

- **`sigmoid_backward`**: backward propagation para SIGMOID:

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

- **`relu_backward`**: backward propagation para RELU:

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

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

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

.

```
# linear_ativacao_backward

def linear_ativacao_backward(dA, cache, ativacao):
    """
    Implementa a backward propagation para ativação.

    Entradas:
    dA -- gradiente da pos-ativacao gradient para camada l
    cache -- tupla de valores (linear_cache, ativacao_cache)
    ativacao -- "sigmoid" or "relu"

    Saídas:
    dA_prev -- gradiente do custo em relação a ativação da camada l-1,
    dW -- gradiente do custo em relação a W  da camada l,
    db -- gradiente do custo em relação a b,
    """
    linear_cache, ativacao_cache = cache

    if ativacao == "relu":
        ### Início do código ###
        dZ =  relu_backward(dA, ativacao_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)
        ### Fim do código ###

    elif ativacao == "sigmoid":
        ### Início do código ###
        dZ = sigmoid_backward(dA, ativacao_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)
        ### Fim do código ###

    return dA_prev, dW, db


# Teste

dAL, linear_ativacao_cache = linear_activation_backward_test_case()

dA_prev, dW, db = linear_ativacao_backward(dAL, linear_ativacao_cache, ativacao = "sigmoid
print ("sigmoid:")
print ("dA_prev = "+ str(dA_prev))
print ("dW = " + str(dW))
print ("db = " + str(db) + "\n")

dA_prev, dW, db = linear_ativacao_backward(dAL, linear_ativacao_cache, ativacao = "relu")
print ("relu:")
print ("dA_prev = "+ str(dA_prev))
print ("dW = " + str(dW))
print ("db = " + str(db))
```

```
[[-0.10414453 -0.01044791]]
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]]

[[-0.41675785  0.        ]]
relu:
dA_prev = [[ 0.44090989  0.        ]
 [ 0.37883606  0.        ]
 [-0.2298228   0.        ]]
dW = [[ 0.44513824  0.37371418 -0.10478989]]
db = [[-0.20837892]]
```

Valores esperados com:

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

Valores esperados 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 - L-Modelo Backward

A Figura mostra a fase backward.

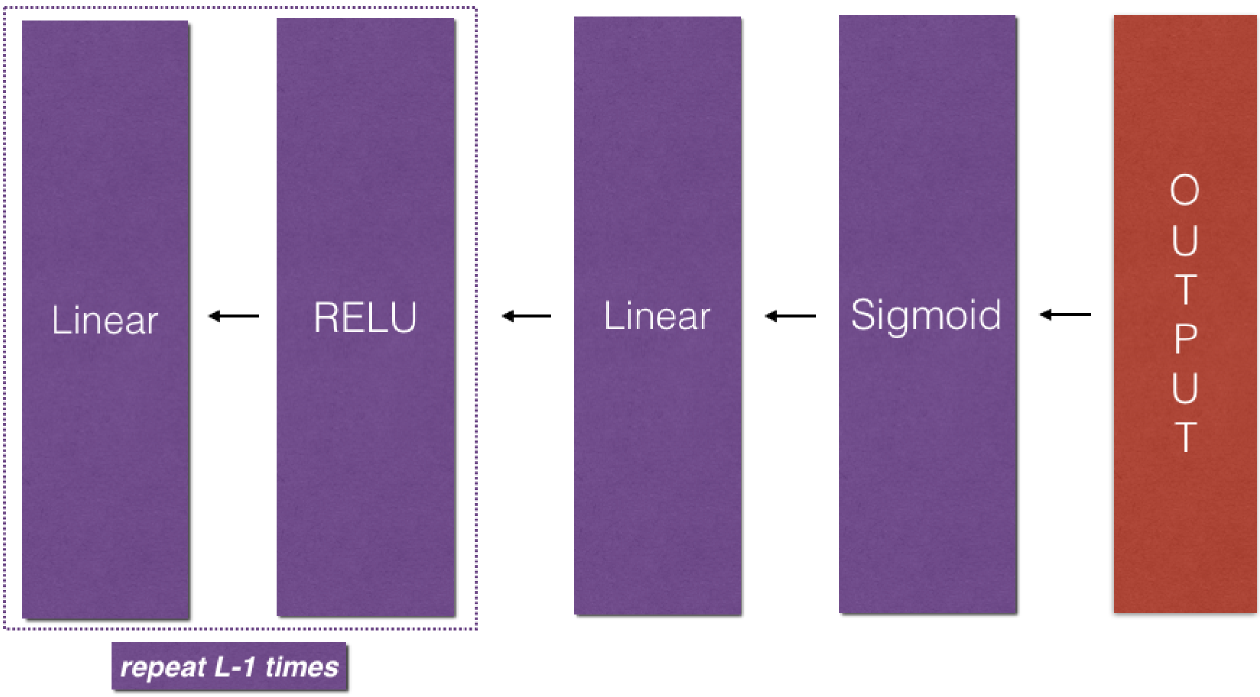

Figura 5 : Fase Backward

Inicializando a fase backpropagation: A saída da rede é, $A^{[L]} = \sigma(Z^{[L]})$. Então temos que calcualar `dAL` $= \frac{\partial \mathcal{L}}{\partial A^{[L]}}$:

`dAL` $= \frac{Y}{AL} - \frac{1-Y}{1-Al}$

O gradiente `dAL` para continuar propagando. Como visto na Figura 5, `dAL` vai alimentar a linear_ativacao_backward com ativação SIGMOID (que utilizará os valores armazenados em cache armazenados pela função L_modelo_forward). Depois disso, você terá que usar um loop `for` para percorrer todas as outras camadas usando linear_ativacao_backward com ativação RELU. Você deve armazenar cada dA, dW e db no dicionário grads.

```
# L_modelo_backward

def L_modelo_backward(AL, Y, caches):
    """
    Implementa a backward propagation para [LINEAR->RELU] * (L-1) -> LINEAR -> SIGMOID


    Entradas:
    AL -- Probabiliade de predição da rede, saída da fase forward propagation (L_modelo_fo
    Y -- Vetor de rótulos dos exemplos de treinamento  ( 0 se não tem gato, 1 tem gato )
    caches -- lista de caches contendo:
                todos cache da linear_ativacao_forward() com "relu" ( caches[l], l = 0...L
                o cache da linear_ativacao_forward() com "sigmoid" (caches[L-1])

    Saídas:
    grads -- Um dicionário com os gradientes

    """
    grads = {}
    L = len(caches) # número de camadas
    m = AL.shape[1] # número de exemplos

    Y = Y.reshape(AL.shape) # Y deve ter o mesmo formato que AL

    # Inicilizando a fase backpropagation
    ### Início do código ###
    dAL = None # gradiente do custo em relação a AL
    ### Fim do código ###

    # gradiente da l-ésima camada (SIGMOID -> LINEAR).
    # Entrada: "dAL, corrente_cache". Saida: "d(AL-1), dWL, dbL"
    ### Início do código ###
    current_cache = caches[L-1]
    grads["dA" + str(L-1)], grads["dW" + str(L)], grads["db" + str(L)] = None
    ### Fim do código ###

    # Gradientes das camadas anterios: (RELU -> LINEAR)
    # Entradas: "dA(l+1), corrente_cache".
    # Saídas: "dA(l), dW(l+1), db(l+1)"
    ### Início do código ###
```

```
    # Loop de l=L-2 até l=0
    for l in reversed(range(L-1)):
      current_cache = caches[l]
      dA_prev_temp, dW_temp, db_temp = None # dice: ver linear_activation_backward()
      grads["dA" + str(l)] = None
      grads["dW" + str(l + 1)] = None
      grads["db" + str(l + 1)] = None
    ### Fim do código ###

    return grads
```

```
AL, Y_teste, caches = L_modelo_backward_teste()

grads = L_modelo_backward(AL, Y_teste, caches)
print_grads(grads)
```

Valores esperados

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 dos parâmetros

Usando gradiente descendente:

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

onde α é a taxa de aprendizagem.

Instruções: Atualização dos parâmetros usando gradiente descendente: $W^{[l]}$ and $b^{[l]}$ para $l = 1, 2, \ldots, L$.

```
# atualize_parametros

def atualize_parametros(parametros, grads, learning_rate):
    """
    Atualização dos parâmetros usando gradiente descendente:

    Entradas:
    parametros -- python dicionario contendo os parametros
    grads -- python dicionario contendo os gradientes, saída L_modelo_backward

    Saídas:
    parametros -- python dicionario contendo os parametros

    """

    L = None  # número de camadas da rede
```

```
    # Atualiza os parametros.
    ### Início do código ###
    for l in range(L):
        parameters["W" + str(l+1)] = None
        parameters["b" + str(l+1)] = None
    ### Fim do código ###
    return parametros
```

```
parametros, grads = update_parameters_test_case()
parametros = atualize_parametros(parametros, grads, 0.1)

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

Valores esperados:

W1	[[-0.59562069 -0.09991781 -2.14584584 1.82662008] [-1.76569676 -0.80627147 0.51115557 -1.18258802] [-1.0535704 -0.8
b1	[[-0.04659241] [-1.28888275] [0.53405496]]
W2	[[-0.55569196 0.0354055 1.32964895]]
b2	[[-0.84610769]]

7 - Construa o modelo (2pt)

Implemente o modelo usando as funções anteriores para treinar os parâmetros da rede no conjunto de dados.

```
# L_layer_modelo

def L_layer_modelo(X, Y, camada_dims, learning_rate = 0.0075, num_iter = 3000, print_custo
    """
    Implementa a uma rede neural com L-camadas: [LINEAR->RELU]*(L-1)->LINEAR->SIGMOID.

    Entradas:
    X -- conjunto de treinamento representado por uma matriz numpy da forma (num_px * num_
    Y -- rótulos de treinamento representados por uma matriz numpy (vetor) da forma (1, nu
    camadas_dims -- lista contendo a dimensão dos dados de entrada e tamanho de cada camad
    learning_rate -- lhiperparâmetro que representa a taxa de aprendizado usada na regra d
    num_iter -- hiperparâmetro que representa o número de iterações para otimizar os parâm
    print_custo -- imprime o custo a cada 100 iterações

    Saida:
    parametros -- parametros aprendidos do modelo.
    """

    np.random.seed(1)
    custos = []                         # guarda o custo
```

```
    # Inicialização dos parametros
    ### Início do código ###
    parameters = None # dica : ver sua função de inicializacao
    ### Fim do código ###

    # Gradiente descendente. Dica : use as funções que você escreveu acima
    for i in range(0, num_iter):

        # Fase Forward propagation: [LINEAR -> RELU]*(L-1) -> LINEAR -> SIGMOID.
        ### Início do código ###
        AL, caches = None
        ### Fim do código ###

        # Calculo do Custo.
        ### Início do código ###
        cost = None
        ### Fim do código ###

        # Fase Backward propagation.
        ### Início do código ###
        grads = None
        ### Fim do código ###

        # Atualização dos parametros.
        ### Início do código ###
        parameters = None
        ### Fim do código ###

        # Imprime o custo cada 100 iterações
        if print_custo and i % 100 == 0:
            print ("Custo depois da iteração %i: %f" %(i, cost))
        if print_custo and i % 100 == 0:
            custos.append(cost)

    # plot the cost
    plt.plot(np.squeeze(custos))
    plt.ylabel('custo')
    plt.xlabel('iterações (por centenas)')
    plt.title("Taxa de aprendizagem =" + str(learning_rate))
    plt.show()

    return parametros
```

8- Pronto! (1pt)

Pre-processamento dos dados

Vamos construir o modelo para treinar um classificador de imagens (o mesmo da regressão logística)

```
# Lendo os dados (gato/não-gato)
def load_dataset():

  train_dataset = h5py.File('/<caminho para os dados>/train_catvnoncat.h5', "r")
  train_set_x_orig = np.array(train_dataset["train_set_x"][:]) # your train set features
  train_set_y_orig = np.array(train_dataset["train_set_y"][:]) # your train set labels

  test_dataset = h5py.File('/<caminho para os dados>/test_catvnoncat.h5', "r")
  test_set_x_orig = np.array(test_dataset["test_set_x"][:]) # your test set features
  test_set_y_orig = np.array(test_dataset["test_set_y"][:]) # your test set labels

  classes = np.array(test_dataset["list_classes"][:]) # the list of classes
  train_set_y_orig = train_set_y_orig.reshape((1, train_set_y_orig.shape[0]))
  test_set_y_orig = test_set_y_orig.reshape((1, test_set_y_orig.shape[0]))

  return train_set_x_orig, train_set_y_orig, test_set_x_orig, test_set_y_orig, classes
```

```
# Lendo os dados (gato/não-gato)
treino_x_orig, treino_y, teste_x_orig, teste_y, classes = load_dataset()
```

Pre-processamento necessário.

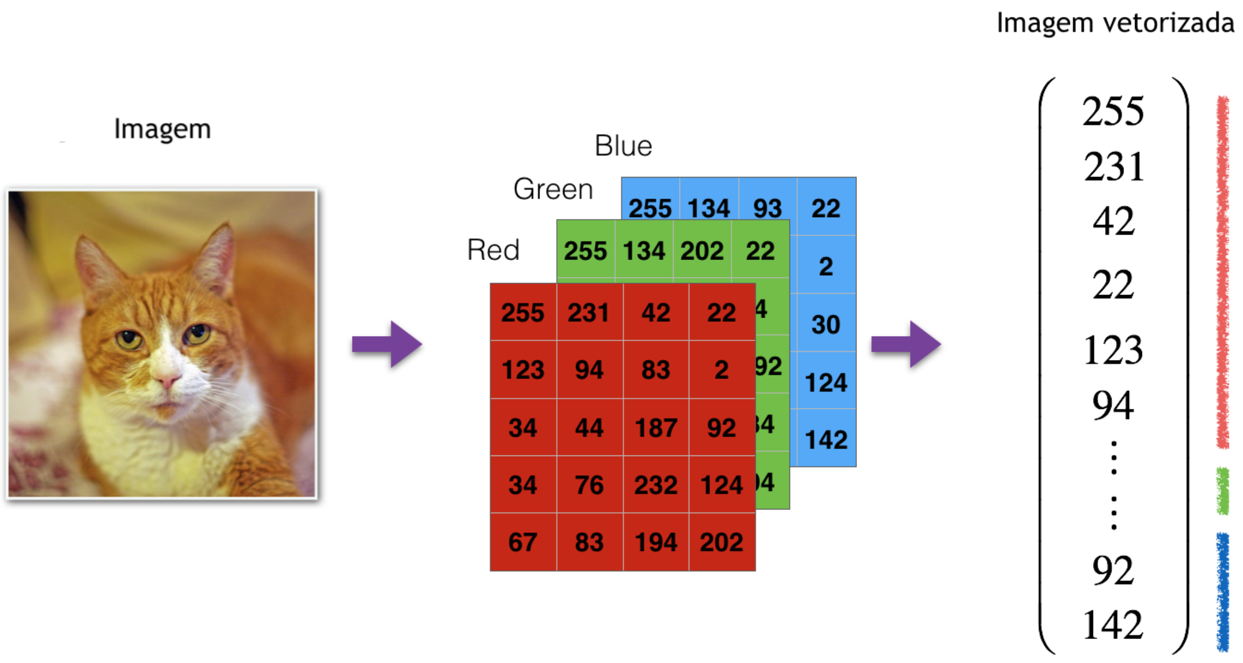

Figura 6: Vetorização de uma imagem.

```
m_treino = len(treino_x_orig)
m_teste = len(teste_x_orig)
num_px = teste_x_orig[1].shape[1]

#  Vetorizando as imagens de treinamento e teste
```

```
### Início do código ###
None # dica : utilize reshape para mudar o formato dos dados
### Fim do código ###

### Início do código ###
# Normalize os dados para ter valores de recurso entre 0 e 1.
None
### Fim do código ###
```

Testando com rede neural com 2 camadas

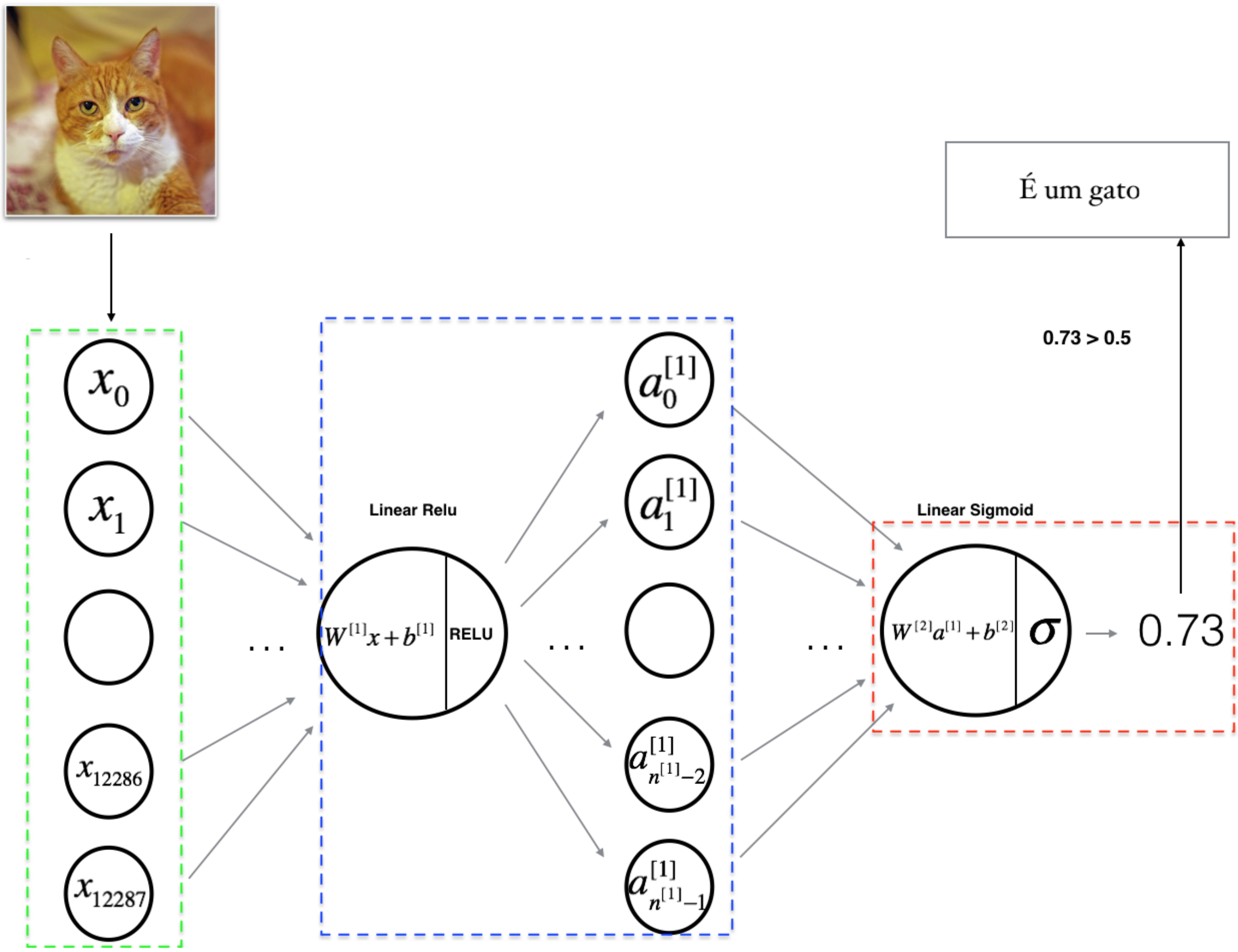

Figura 7: Rede neural com 2 camadas.

Resumo do modelo: ***ENTRADA -> LINEAR -> RELU -> LINEAR -> SIGMOID -> SAIDA***.

[ ] ↳ *5 células ocultas*

Testando com uma rede com 4 camadas

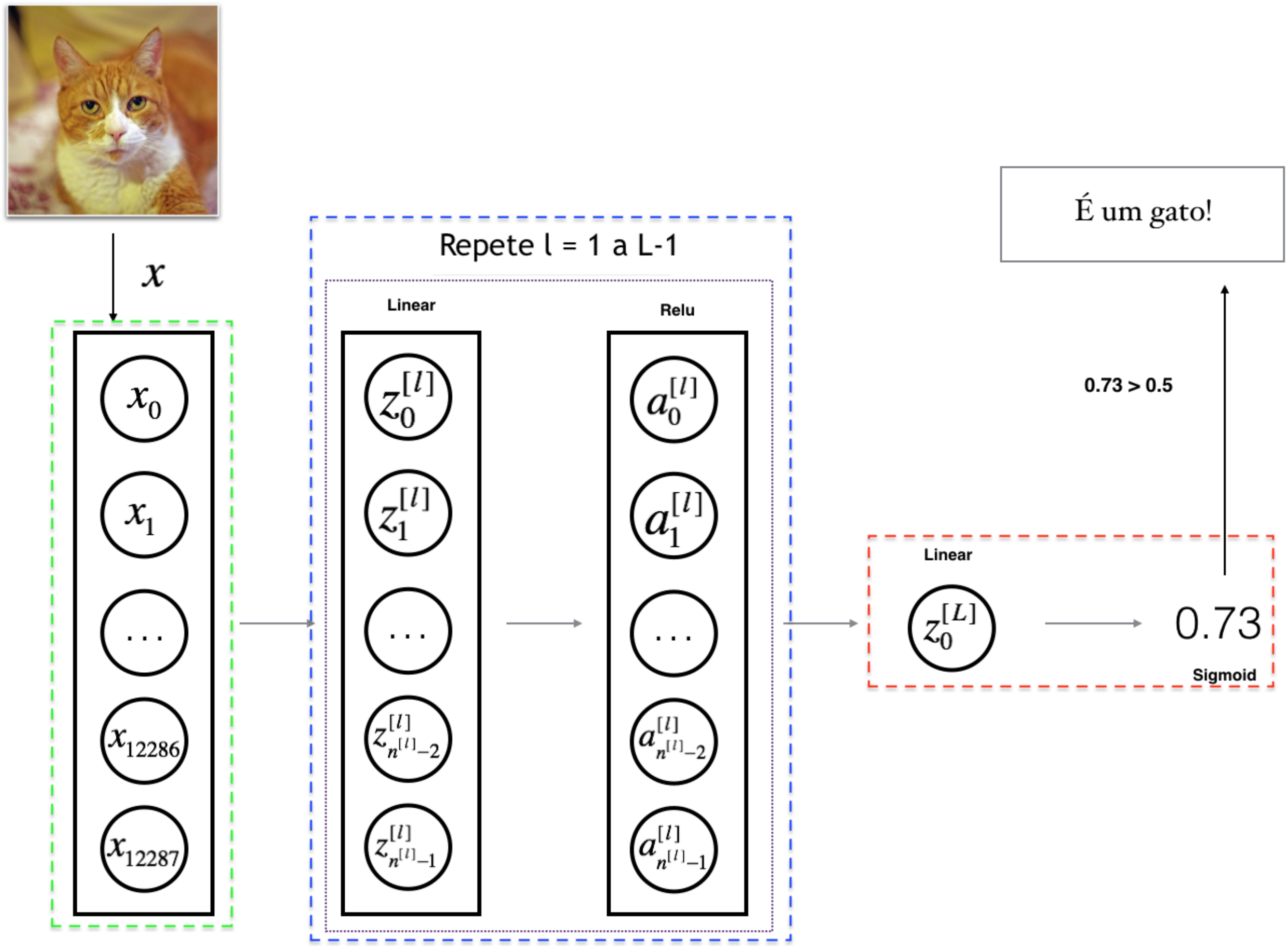

Figura 8: Rede neural com L camadas.
Resumo do modelo: ***ENTRADA -> LINEAR -> RELU -> LINEAR -> SIGMOID -> SAIDA***.

[ ] ↳ *5 células ocultas*

Parte 2 - Classificação de múltiplas classes e uso de frameworks

No exemplo anterior, usamos uma arquitetura para classificação binária. Para classificaçõ de múltiplas classes, tem-se um neurônio de saída para cada classe (como ilustrado no exemplo da Figura 9) e deve-se usar a operação Softmax antes de se calcular o custo (entropia cruzada ou cross-entropy como no exemplo anterior). Consute o capítulo 3.6 do livro para entender melhor. No caso de se usar softmax, deve-se usar a função *one_hot* para transformar a saída em logits. Veja a função *one_hot* fornecida. Ela transforma um escalar em um *hot encoder*, de acordo com o número de classes.

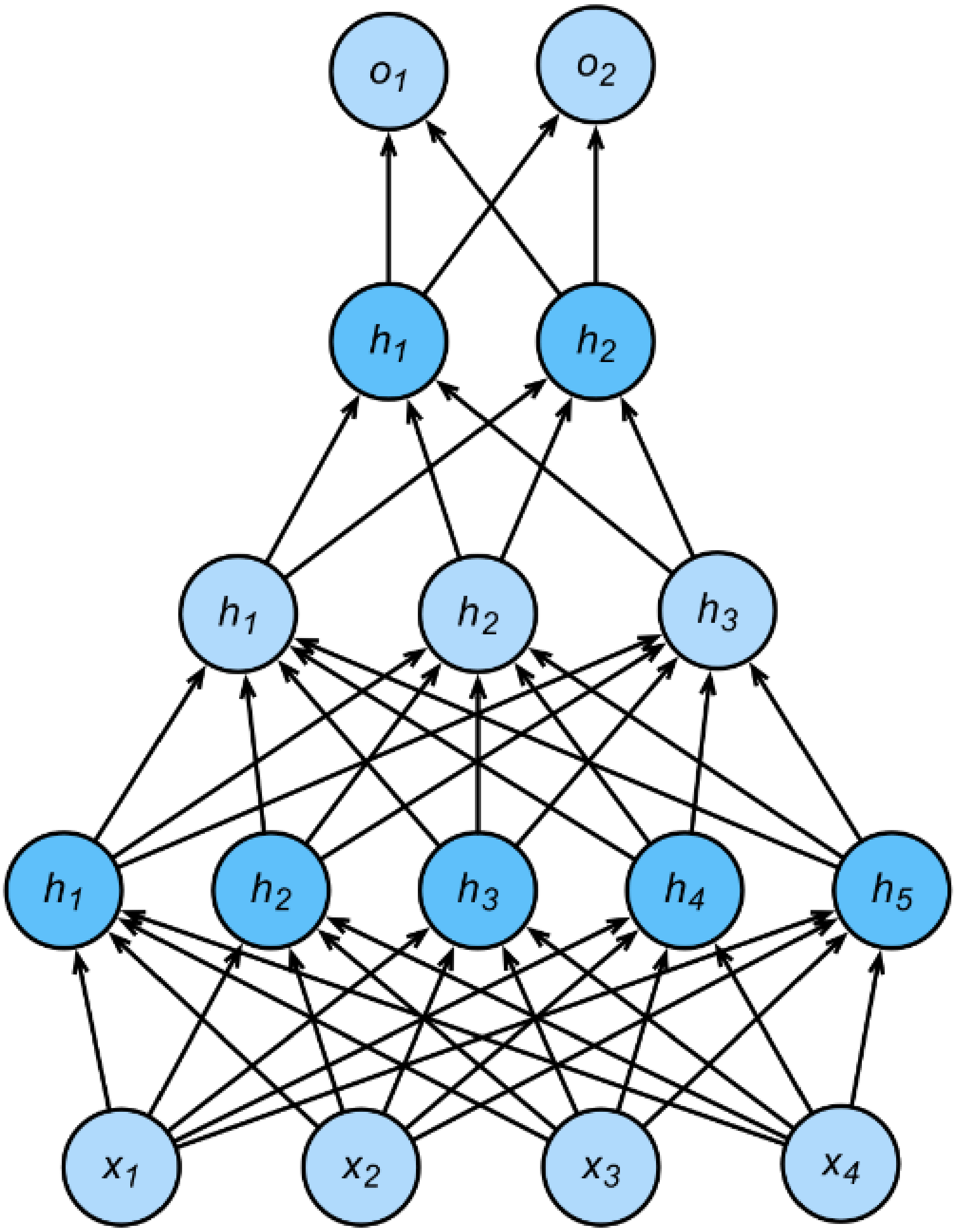

Figura 9: Rede neural dois neurônios de saída.

```
# nclasses : numero de classes do prolema, y : um escalar ou vetor de escalares
def one_hot(n_classes, y):
    return np.eye(n_classes)[y]
```

```
# ToDo : execute o exemplo e veja o resultado para 4 escalares no vetor de variáveis depen
one_hot(n_classes=10, y=[0, 4, 9, 1])
```

Função softmax

A função softmax transforma a saída em uma distribuição de probabilidades. Assim, a soma de todas as saídas dos neurônio da última camada sempre vai ser igual a 1:

$$softmax(\mathbf{x}) = \frac{1}{\sum_{i=1}^{n} e^{x_i}} \cdot \begin{bmatrix} e^{x_1} \\ e^{x_2} \\ \vdots \\ e^{x_n} \end{bmatrix}$$

o gradiente para o custo usando-se a função softmax é trivial de se calcular:

$$dw = softmax(\mathbf{y_{pred}}) - y$$

```
def softmax(X):
    exp = np.exp(X)
    return exp / np.sum(exp, axis=-1, keepdims=True)
```

```
# ToDo : teste sua função softmax com a instância do exemplo abaixo
print(softmax([10, 2, -3]))

# As saídas individuais devem ser entre 0 e 1 de forma que a soma seja 1. lembre-se, com s
print(np.sum(softmax([10, 2, -3])))
```

```
[9.99662391e-01 3.35349373e-04 2.25956630e-06]
1.0
```

Perceba que nosso código também funciona se você passar um lote (batch) de amostras

```
# Veja a saída abaixo
X = np.array([[10, 2, -3],
              [-1, 5, -20]])
print(softmax(X))
```

```
[[9.99662391e-01 3.35349373e-04 2.25956630e-06]
 [2.47262316e-03 9.97527377e-01 1.38536042e-11]]
```

Em seguida, deve-se computar o erro entre um vetor predito Y_pred e o vetor de rótulos Y_true. para tal, deve-se usar cross entropy loss, ou verossimilhança negativa (negative log likelihood). A função cross_entrppy() implementa a verossimilhança negativa.

```
def cross_entropy(Y_true, Y_pred):
    EPSILON = 1e-8

    Y_true, Y_pred = np.atleast_2d(Y_true), np.atleast_2d(Y_pred)
    loglikelihoods = np.sum(np.log(EPSILON + Y_pred) * Y_true, axis=1)

    return -np.mean(loglikelihoods)
```

verifique o erro de uma predição bem ruim

```
print(cross_entropy([1, 0, 0], softmax([0.12, 4, 10])))
```

```
9.882330913250298
```

verifique o erro de uma boa predição

```
print(cross_entropy([1, 0, 0], [0.98, 0.01, .01]))
```

```
0.020202697113437834
```

A função cross_entropy() também deve funcionar para um lote de dados

```
# Verifique a cross-entropy das três amostas seguintes:

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

Y_pred = np.array([[0,   1,    0],
                   [.99, 0.01, 0],
                   [0,   0,    1]])

# repare que as amostras são praticamente predições perfeitas
print(cross_entropy(Y_true, Y_pred))
```

```
0.0033501019174971905
```

Pré-processamento dos dados

Vamos usar a biblioteca scikit learn para nos auxiliar na execução da prática. Veja a documentação em https://scikit-learn.org/stable/index.html

Considere a base de dados abaixo. Ela é referente a um atividade em um site de vendas qualquer. O objetivo com esta base é tentar predizer quais clientes futuros terão probabildiade de comprar algum produto, com base em algumas características, como cidade em que mora, idade e salário.

Carregando os dados

```
# Importe as bibliotecas NumPy, Pandas e Matplotlib
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

```
# carregue os dados do arquivo e armazenar em um Dataframe
dataset = pd.read_csv('<caminho>/datasets/Data.csv')

# imprima a estrutura dataset
print(dataset)
```

Crie dois objetos, uma chamado X, para receber as caracteísticas das instâncias e um chamado y para receber as classes. Observe que as instâncias devem ser organizadas em linha Assm, as características da linha 0 de X devem corresponder a classe da linha 0 de y Podemos chamar as variáveis de X (ou características -usadas para fazer a predição) de variáveis independentes e a variável de y (classe a ser predita) de variável dependente.

```
X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, 3].values

# imprima X e y
print(X)
print(y)

# imprima e analise o formato dos objetos
print(X.shape)
print(y.shape)
```

```
from sklearn.preprocessing import Imputer
imputer = Imputer(missing_values = 'NaN', strategy = 'mean', axis = 0)
imputer.fit(X[:, 1:3])
X[:, 1:3] = imputer.transform(X[:, 1:3])

# imprima a nova matriz X
print(X)
```

✓ 0s conclusão: 14:31 ×