Trabalho #1 - Python com NumPy

Este trabalho fornece uma breve revisão de Python. Mesmo que você já tenha usado Python, esse trabalho irá ajudar você a se familiarizar com as funções que precisaremos nessa disciplina.

Instruções:

- Você estará usando o Python 3.
- Evite usar for-loops e while-loops, a menos que seja explicitamente instruído a fazê-lo.
- As suas tarefas nesse trabalho estão sinalizadas nas células pelo comentário " `# PARA VOCÊ FAZER:` ".
- Depois de incluir o seu código, execute a célula modificada para verificar se o resultado está correto.

Após esse trabalho você irá:

- Ser capaz de usar os iPython Notebooks
- Ser capaz de usar funções numpy e operações de tensores numpy
- Compreender o conceito de "broadcasting"
- Ser capaz de vetorizar um código

(Esse trabalho é um adaptação de Adrew Ng, deeplearnig.ai)

Coloque os nomes e RAs dos alunos que fizeram esse trabalho

Nome e número dos alunos da equipe:

Aluno 1: Rodrigo Franciozi Rodrigues da Silva RA:20.83984-7

Aluno 2:

Sobre os notebooks iPython

Os notebooks iPython são ambientes de codificação interativos incorporados em uma página da web. Você estará usando notebooks iPython nesta disciplina. Você só precisa escrever código entre os comentários `### COMEÇE AQUI ###` e `### TERMINE AQUI ###`. Depois de escrever seu código, você pode executar a célula pressionando "SHIFT" + "ENTER" ou clicando em "Run Cell" localizado na barra superior do bloco de notas.

Frequentemente, especificaremos " `(≈ X linhas de código)` " nos comentários para informar sobre quanto código você precisa escrever. Isso é apenas uma estimativa aproximada, por isso não se sinta mal se o seu código for mais longo ou mais curto.

Exercício #1:

Modifique a variável teste para `" Alô Mundo "` na célula abaixo para imprimir `Alô Mundo` e execute as duas células abaixo.

In [1]:

```
# PARA VOCÊ FAZER:

### COMEÇE AQUI ### (≈ 1 linha de código)
teste = "Alô Mundo"
### TERMINE AQUI ###
```

In [2]:

```
print ("teste: " + teste)
```

```
teste: Alô Mundo
```

Saída esperada:

```
teste: Alô Mundo
```

O que você precisa lembrar:

- Executar suas células usando SHIFT + ENTER (ou clicar em "RUN" na barra superior)
- Escrever código nas áreas designadas usando apenas Python 3
- Não modifique o código fora das áreas designadas

1 - Construindo funções básicas com numpy

Numpy é a principal biblioteca de funções para computação científica em Python. É mantido por uma grande comunidade (www.numpy.org). Neste exercício, você aprenderá várias funções de numpy, tais como, np.exp, np.log e np.reshape. Você precisará saber como usar essas funções para futuros trabalhos.

1.1 - Função sigmoide

A função sigmoide é baseada na função exponencial. Mas antes de usar a função exponencial da biblioteca numpy, `np.exp()`, você usará a função exponencial da bilbioteca, math `math.exp()`, para implementar a função sigmoide. Você verá então porque `np.exp()` é preferível a `math.exp()`.

Exercício #2:

Construa uma função que retorne a sigmoide de um número real `x`. Use `math.exp(x)` para a calcular a função exponencial.

Observação: a função $ sigmoid(x) = \frac{1} {1 + e ^ {- x}} $ é também conhecida como função logística, sendo uma função não linear usada não só em redes neurais mas também em Machine Learning (Regressão Logística).

Para se referir a uma função pertencente a um pacote específico, você pode chamá-la usando `package_name.function()`.

Modifique e execute o código abaixo para criar e ver a função `basic_sigmoid()` com `math.exp()`.

In [3]:

```
# PARA VOCÊ FAZER: basic_sigmoid

import math

def basic_sigmoid(x):
    """
    Calcula sigmoid de x.

    Argumentos:
    x -- A escalar

    Retorna:
    s -- sigmoid(x)
    """

    ### COMEÇE AQUI ### (≈ 1 linha de código)
    s = 1/(1+math.exp(-x))
    ### TERMINE AQUI ###

    return s
```

In [4]:

```
basic_sigmoid(3)
```

Out[4]:

```
0.9525741268224334
```

Resultado esperado:

```
0.9525741268224334
```

Na verdade, raramente usamos a biblioteca "matemática" em redes neurais porque as entradas das funções são números reais. Em redes neurais usamos principalmente matrizes e vetores (tensores). É por isso que numpy é mais útil. Execute a célula abaixo e veja

redes neurais usamos principalmente matrizes e vetores (tensores). É por isso que numpy é mais útil. Execute a célula abaixo e veja o que acontece.

In [5]:

```
### Uma razão para usar "numpy" no lugar de "math" em redes neurais ###
x = [1, 2, 3]
basic_sigmoid(x) # você verá que essa função gera um erro, isso é causado porque x é um vetor.
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-f8d61ca34618> in <module>
      1 ### Uma razão para usar "numpy" no lugar de "math" em redes neurais ###
      2 x = [1, 2, 3]
----> 3 basic_sigmoid(x) # você verá que essa função gera um erro, isso é causado porque x é um vet
or.

<ipython-input-3-03009a5d1fea> in basic_sigmoid(x)
     15
     16     ### COMEÇE AQUI ### (≈ 1 linha de código)
---> 17     s = 1/(1+math.exp(-x))
     18     ### TERMINE AQUI ###
     19

TypeError: bad operand type for unary -: 'list'
```

De fato, se $ x = (x_1, x_2, ..., x_n)$ é um vetor linha, então $np.exp(x)$ aplica a função exponencial idependentemente a cada elemento do vetor x. A saída será então: $np.exp(x) = (e^{x_1}, e^{x_2}, ..., e^{x_n})$

In [6]:

```
import numpy as np # com isso você pode acessar as funções do numpy escrevendo somente
np.function() no lugar de numpy.function()

# Exemplo de np.exp
x = np.array([1, 2, 3])
print(np.exp(x)) # resultado é (exp(1), exp(2), exp(3))
```

```
[ 2.71828183  7.3890561  20.08553692]
```

Exercício #3:

Crie alguns tensores numpy e execute algumas operações com esses tensores usando as funções da biblioteca numpay.

Se x é um vetor **numpy**, então uma operação em Python, tal como, $s = x + 3$ or $s = \frac{1}{x}$ irá produzir um vetor de saída s do mesmo tamanho do vetor de entrada x.

In [7]:

```
# PARA VOCÊ FAZER: operações com vetores
"""
    Dados dois vetores u e v  calcule:
    s1 = v + 3
    s2 = 1/v
    umax = valor maximo de u
    vmin = valor mínimo de v
    uv = produto escalar de u por v
    uvi = produto elemento por elemento de v por u
"""

v = np.array([1, 2, 3])
u = np.array([-1, -2, -3])

### COMEÇE AQUI ### (≈ 6 linhas)
s1 = v + 3
s2 = 1/v
umax = np.max(u)
vmin = np.min(v)
uv = np.dot(u,v)
uvi = u*v
### TERMINE AQUI ###
```

```
print("v + 3 = ", s1)
print("1/v = ", s2)
print("umax = ", umax)
print("vmin = ", vmin)
print("uv = ", uv)
print("uvi = ", uvi)
```

```
v + 3 =  [4 5 6]
1/v =  [1.         0.5        0.33333333]
umax =  -1
vmin =  1
uv =  -14
uvi =  [-1 -4 -9]
```

Resultado esperado:

```
v + 3 = [4 5 6]
1/v = [1. 0.5 0.33333333]
umax = -1
vmin = 1
uv = -14
uvi = [-1 -4 -9]
```

A qualquer momento que você precisar de mais informação sobre uma função do numpy, olhe na documentação oficial.

Você pode também criar uma nova célula no notebook jupiter e escrever `np.exp?` (por exemplo) para conseguir acesso rápido à documentação.

Exercício #4:

Implemente a função `sigmoid()` usando numpy.

Instruções: x pode ser um número real, um vetor, ou uma matriz. A estrutura de dados que se usa em numpy para representar esses tipos de dados (vetores, matrizes, tensores...) são chamados de "numpy arrays".
$$ \text{Para } x \in \mathbb{R}^n \text{, } sigmoid(x) = sigmoid\begin{pmatrix} x_1 \\ x_2 \\ ... \\ x_n \\ \end{pmatrix} = \begin{pmatrix} \frac{1}{1+e^{-x_1}} \\ \frac{1}{1+e^{-x_2}} \\ ... \\ \frac{1}{1+e^{-x_n}} \\ \end{pmatrix} $$

In [8]:

```
# PARA VOCÊ FAZER: sigmoid

import numpy as np

def sigmoid(x):
    """
    Calcule a fnção sigmoid de x

    Argumentos:
    x -- Um escalar ou um numpy array de qualquer dimensão

    Retorna:
    s -- sigmoid(x)
    """

    ### COMECE AQUI ### (≈ 1 linha)
    s = (1/(1+np.exp(-x)))
    ### TERMINE AQUI ###

    return s
```

In [9]:

```
x = np.array([1, 2, 3])
sigmoid(x)
```

Out[9]:

```
array([0.73105858, 0.88079708, 0.95257413])
```

Saída esperada:

```
array([ 0.73105858,  0.88079708,  0.95257413])
```

1.2 - Derivada da função sigmoide

Como visto em aula, você precisa calcular derivadas para otimizar a função de custo usando a retro-propagação. Vamos codificar a derivada da função sigmoide.

Exercício #5:

Implemente a função `sigmoid_grad()` para calcular a derivada da função sigmóide em relação à sua entrada x.

A fórmula é: $$sigmoid\_derivative(x) = \sigma'(x) = \sigma(x) (1 - \sigma(x))$$

Você pode codificar essa função em duas etapas:

1. Defina s como sendo a sigmoide de x. Use a sua função sigmoide.
2. Calcule $\sigma'(x) = s(1-s)$

In [10]:

```
# PARA VOCÊ FAZER: sigmoid_derivative

def sigmoid_derivative(x):
    """
    Calcule a derivada da função sigmoide em relação à sua entrada.

    Arguments:
    x -- um escalar ou um tensor numpy

    Return:
    ds -- derivada da sigmoide
    """

    ### COMECE AQUI ### (≈ 2 linhas)
    s = sigmoid(x)
    ds = s*(1-s)
    ### TERMINE AQUI ###

    return ds
```

In [11]:

```
x = np.array([1, 2, 3])
print ("sigmoid_derivative(x) = " + str(sigmoid_derivative(x)))
```

```
sigmoid_derivative(x) = [0.19661193 0.10499359 0.04517666]
```

Saída esperada:

```
sigmoid_derivative(x) = [ 0.19661193  0.10499359  0.04517666]
```

1.3 - Alterando dimensões de tensores

Duas funções comuns usadas em deep-learning são np.shape() e np.reshape().

- `X.shape(X)` é usada para obter a dimensão de um tensor `X`.
- `X.reshape(X)` é usada para alterar a dimensão de `X`.

Por exemplo, uma imagem digital é representada por um tensor 3D de dimensões $(length, height, depth = 3)$. Contudo, quando se usa uma imagem como entrada de uma RNA ela em geral deve ser convertida para uma vetor de dimensão $(length*height*3, 1)$. Em outras palavras, deve-se desenrolar ou alterar a dimensão do tensor 3D para um vetor 1D.

(Andrew Ng, deeplearning.ai)

Exercício #6:

Implemente uma função `image2vector()` que recebe como entrada uma imagem (tensor) de dimensão (length, height, 3) e retorna um vetor de dimensão (length*height*3, 1). Por exemplo, se quiser redimensionar um tensor v de dimensão (a, b, c) para um vetor de dimensão (a*b,c) deve-se fazer:

```
v = v.reshape((v.shape[0]*v.shape[1], v.shape[2])) # v.shape[0] = a ; v.shape[1] = b ; v.shape[2] = c
```

Observação: nunca codifique as dimensões de uma imagem como uma constante. No lugar obtenha as dimensões da imagem com a função `image.shape[0]`, ou `image.shape[1]`, etc.

In [12]:

```
# PARA VOCÊ FAZER: image2vector
def image2vector(image):
    """
    Argumento:
    image -- um tensor numpy de dimensão (length, height, depth)

    Retorna:
    v -- um vetor de dimensão (length*height*depth, 1)
    """

    ### COMECE AQUI ### (≈ 1 linha)
    v = image.reshape((image.shape[0]*image.shape[1]*image.shape[2], 1))
    ### TERMINE AQUI ###

    return v
```

In [13]:

```
# Exemplo de tensor 3 por 3 by por 2, typicamente uma imagem tem dimensão (num_px_x, num_px_y,3) o
nde o terceiro eixo representa
# os planos de cor RGB
image = np.array([[[ 0.67826139,  0.29380381],
        [ 0.90714982,  0.52835647],
        [ 0.4215251 ,  0.45017551]],

       [[ 0.92814219,  0.96677647],
        [ 0.85304703,  0.52351845],
        [ 0.19981397,  0.27417313]],

       [[ 0.60659855,  0.00533165],
        [ 0.10820313,  0.49978937],
        [ 0.34144279,  0.94630077]]])

print ("image2vector(image) = " + str(image2vector(image)))
```

```
image2vector(image) = [[0.67826139]
 [0.29380381]
 [0.90714982]
 [0.52835647]
 [0.4215251 ]
 [0.45017551]
 [0.92814219]
 [0.96677647]
 [0.85304703]
 [0.52351845]
 [0.19981397]
 [0.27417313]
 [0.60659855]
 [0.00533165]
 [0.10820313]
 [0.49978937]
 [0.34144279]
 [0.94630077]]
```

Saída esperada:

```
image2vector(image) = [[ 0.67826139]
 [ 0.29380381]
 [ 0.90714982]
 [ 0.52835647]
 [ 0.4215251 ]
 [ 0.45017551]
 [ 0.92814219]
 [ 0.96677647]
 [ 0.85304703]
 [ 0.52351845]
 [ 0.19981397]
 [ 0.27417313]
 [ 0.60659855]
 [ 0.00533165]
 [ 0.10820313]
 [ 0.49978937]
 [ 0.34144279]
 [ 0.94630077]]
```

1.4 - Normalização de vetores

Uma operação comum em Aprendizado de Máquina é normalizar os dados. Em geral a normalização dos dados gera um desemepnho melhor da RNA em razão do gradiente descendente convergir mais rápido com dados normalizados. Nessa tarefa, vamos normalizar um vetor linha dividindo-o pela sua norma 2, ou seja:
$$ \frac{x}{\| x\|} $$

Por exemplo, se:
$$x = \begin{bmatrix} 0 & 3 & 4 \\ 2 & 6 & 4 \\ \end{bmatrix}$$

então essa operação pode ser realizada pela seguinte função Numpy:
$$\| x\| = np.linalg.norm(x, axis = 1, keepdims = True) = \begin{bmatrix} 5 \\ \sqrt{56} \\ \end{bmatrix}$$

e por:
$$ x\_normalized = \frac{x}{\| x\|} = \begin{bmatrix} 0 & \frac{3}{5} & \frac{4}{5} \\ \frac{2}{\sqrt{56}} & \frac{6}{\sqrt{56}} & \frac{4}{\sqrt{56}} \\ \end{bmatrix}$$

Note que é possível dividir matrizes de dimensões diferentes sem problemas, pois o "broadcasting" faz isso automaticamente.

Observação: Verifique na documentação da biblioteca numpy o que significa os argumentos `axis` e `keepdims` na função `linalg.norm`.

Exercício #7:

Implemente a função `normalizeRows()` para normalizar as linhas de uma matriz. Após aplicar essa função em uma matriz cada linha deve ser um vetor com norma 2 igual a 1.

In [14]:

```
# PARA VOCÊ FAZER: normalizeRows

def normalizeRows(x):
    """
    Implemente uma função que normaliza cada linha de uma matriz de forma que tenham norma unitári
a.

    Argumento:
    x -- Uma matriz numpy de dimensões (n, m)

    Retorna:
    x -- A matriz normalizada (linha por linha).
    """

    ### COMECE AQUI ### (≈ 2 linHAS)
    # Calcule x_norm como sendo a norma 2 de x. Use np.linalg.norm(..., ord = 2, axis = ..., keepd
ims = True)
    x_norm = np.linalg.norm(x, axis = 1, keepdims = True)
```

```
    x = x/x_norm
    ### END CODE HERE ###

    return x
```

In [15]:

```
x = np.array([
    [0, 3, 4],
    [1, 6, 4]])
print("Matriz normalizada = " + str(normalizeRows(x)))
```

```
Matriz normalizada = [[0.         0.6        0.8       ]
 [0.13736056 0.82416338 0.54944226]]
```

Saída esperada:

```
Matriz normalizada = [[ 0.          0.6         0.8       ]
 [ 0.13736056  0.82416338  0.54944226]]
```

Observação:

Tente na função `normalizeRows()` imprimir as dimensões de `x_norm` e `x` e executar novamente. Você verá que eles tem dimensões diferentes. Isso é óbvio dado que `x_norm` é um único valor para cada linha de `x`. Portanto a dimensão de `x_norm` é igual ao número de linhas de `x`, mas é somente um vetor coluna. A divisão de `x` por `x_norm` só é possível pelo "broadcast" (ajuste) feito automaticamente pelo Numpy.

1.5 - Broadcasting e a função softmax

Como vimos na teoria um conceito muito importante em Numpy é o ajuste automático de dimensões ("broadcasting"). O "broadcasting" é muito útil para realizar operações matemáticas entre tensores de dimensões diferentes. Para ver todos os detalhes do conceito de "broadcasting", você pode ver a documentação oficial do broadcasting broadcasting documentation.

Exercício #8:

Implemente uma função chamada `softmax` usando numpy. Você pode considerar que a função `softmax` é uma função que normaliza as linhas da exponencial de cada elemento de uma matriz.

Instrucões:

- $ \text{para um vetor } x \in \mathbb{R}^{1\times n} \text{, } softmax(x) = softmax(\begin{bmatrix} x_1 && x_2 && ... && x_n \end{bmatrix}) = \begin{bmatrix} \frac{e^{x_1}}{\sum_{j}e^{x_j}} && \frac{e^{x_2}}{\sum_{j}e^{x_j}} && ... && \frac{e^{x_n}}{\sum_{j}e^{x_j}} \end{bmatrix}$
- $\text{para uma matrix } x \in \mathbb{R}^{m \times n} \text{: }$ $$softmax(x) = softmax\begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1n} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & x_{m3} & \dots & x_{mn} \end{bmatrix} = \begin{bmatrix} \frac{e^{x_{11}}}{\sum_{j}e^{x_{1j}}} & \frac{e^{x_{12}}}{\sum_{j}e^{x_{1j}}} & \frac{e^{x_{13}}}{\sum_{j}e^{x_{1j}}} & \dots & \frac{e^{x_{1n}}}{\sum_{j}e^{x_{1j}}} \\ \frac{e^{x_{21}}}{\sum_{j}e^{x_{2j}}} & \frac{e^{x_{22}}}{\sum_{j}e^{x_{2j}}} & \frac{e^{x_{23}}}{\sum_{j}e^{x_{2j}}} & \dots & \frac{e^{x_{2n}}}{\sum_{j}e^{x_{2j}}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{e^{x_{m1}}}{\sum_{j}e^{x_{mj}}} & \frac{e^{x_{m2}}}{\sum_{j}e^{x_{mj}}} & \frac{e^{x_{m3}}}{\sum_{j}e^{x_{mj}}} & \dots & \frac{e^{x_{mn}}}{\sum_{j}e^{x_{mj}}} \end{bmatrix} = \begin{pmatrix} softmax\text{(primeira linha de x)} \\ softmax\text{(segunda linha de x)} \\ ... \\ softmax\text{(última linha de x)} \\ \end{pmatrix}$$
- $\text{para calcular a somatoria dos elementos de um vetor pode ser usada a função np.sum() da seguinte forma:}$ $\| x\| = np.sum(x, axis = 1, keepdims = True)$

In [16]:

```
# PARA VOCÊ FAZER: softmax

def softmax(x):
    """Calcule a função softmax para cada linha da matriz de entrada x.

    Seu programa deve funcionar para um vetor linha e também para matrizes de dimensões genéricas
(n, m).
```

```
    Argumento:
    x -- Uma matriz numpy de dimensões (n,m)

    Retorna:
    s -- Uma matriz numpy de dimensões A numpy matrix equal to the softmax of x, of shape (n,m)
    """

    ### COMECE AQUI ### (≈ 3 lINHAS)
    x_exp = np.exp(x)
    x_sum = np.sum(x_exp, axis =1, keepdims = True)
    s = x_exp/x_sum
    ### TERMINE AQUI ###

    return s
```

In [17]:

```
x = np.array([
    [9, 2, 5, 0, 0],
    [7, 5, 0, 0 ,0]])
print("softmax(x) = " + str(softmax(x)))
```

```
softmax(x) = [[9.80897665e-01 8.94462891e-04 1.79657674e-02 1.21052389e-04
  1.21052389e-04]
 [8.78679856e-01 1.18916387e-01 8.01252314e-04 8.01252314e-04
  8.01252314e-04]]
```

Saída esperada:

```
softmax(x) = [[  9.80897665e-01   8.94462891e-04   1.79657674e-02   1.21052389e-04
  1.21052389e-04]
 [8.78679856e-01   1.18916387e-01   8.01252314e-04   8.01252314e-04
  8.01252314e-04]]
```

Nota: Se você imprimir as dimensões de `x_exp`, `x_sum` e `s` acima, você verá que `x_sum` tem dimensão (2,1) enquanto que `x_exp` e `s` tem dimensões (2,5). O cálculo **x_exp/x_sum** funciona em razão do "broadcasting" em Python.

O que é preciso lembrar:

- `np.exp(x)` funciona para qualquer tensor numpy e aplica a função exponencial em todos os elementos do tensor;
- a função `sigmoide`, sua derivada e a função `image2vector` são muito usadas em deep-learning;
- `np.reshape` é amplamente utilizada;
- manter as dimensões dos tensores corretas é muito importante;
- numpy possui muitas funcções úteis;
- broadcasting é extremamente útil.

2 - Vectorização

Como visto na teoria, em deep-learning lidamos com conjunto de dados extremamente grandes. Portanto, as funções devem ser otimizadas para não criar um gargalo computacional e demorar muito para realizar os cálculos. Para tornar um programa eficiente devemos usar vetorização dos cálculos. Por exemplo, vamos ver a diferencça entre as seguintes implementações das funções produto escalar (produto interno), produto externo e produto de elemento por elemento de dois vetores implementadas classicamente com loops e de forma vetorizada.

In [18]:

```
import time

x1 = [9, 2, 5, 0, 0, 7, 5, 0, 0, 0, 9, 2, 5, 0, 0]
x2 = [9, 2, 2, 9, 0, 9, 2, 5, 0, 0, 9, 2, 5, 0, 0]

### IMPLEMENTAÇÃO CLÁSSICA DO PRODUTO ESCALAR ENTRE DOIS VETORES ###
tic = time.process_time()
dot = 0
for i in range(len(x1)):
    dot+= x1[i]*x2[i]
```

```
print ("produto escalar = " + str(dot))

### IMPLEMENTAÇÃO CLÁSSICA DO PRODUTO EXTERNO ENTRE DOIS VETORES ###
outer = np.zeros((len(x1),len(x2))) # primeiramente criamos uma matriz de zeros com dimensão
len(x1) x len(x2)
for i in range(len(x1)):
    for j in range(len(x2)):
        outer[i,j] = x1[i]*x2[j]
#toc = time.process_time()
print ("produto de dois vetores = " + str(outer))

### IMPLEMENTAÇÃO CLÁSSICA DO PRODUTO DE DOIS VETORES ELEMENTO POR ELEMENTO ###
mul = np.zeros(len(x1))
for i in range(len(x1)):
    mul[i] = x1[i]*x2[i]
#toc = time.process_time()
print ("multiplicação elemento-por-elemento = " + str(mul))

### IMPLEMENTAÇÃO CLÁSSICA DO PRODUTO DE UMA MATRIZ POR UM VETOR ###
np.random.seed(3)
W = np.random.rand(3,len(x1)) # Vetor numpy com números aletórios de dimensão 3 x len(x1)
gdot = np.zeros(W.shape[0])
for i in range(W.shape[0]):
    for j in range(len(x1)):
        gdot[i] += W[i,j]*x1[j]
toc = time.process_time()
print ("produto matriz-vetor = " + str(gdot) + "\n ----- Tempo de computação = " + str(1000*(toc -
tic)) + "ms")
```

```
produto escalar = 278
produto de dois vetores = [[81. 18. 18. 81.  0. 81. 18. 45.  0.  0. 81. 18. 45.  0.  0.]
 [18.  4.  4. 18.  0. 18.  4. 10.  0.  0. 18.  4. 10.  0.  0.]
 [45. 10. 10. 45.  0. 45. 10. 25.  0.  0. 45. 10. 25.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [63. 14. 14. 63.  0. 63. 14. 35.  0.  0. 63. 14. 35.  0.  0.]
 [45. 10. 10. 45.  0. 45. 10. 25.  0.  0. 45. 10. 25.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [81. 18. 18. 81.  0. 81. 18. 45.  0.  0. 81. 18. 45.  0.  0.]
 [18.  4.  4. 18.  0. 18.  4. 10.  0.  0. 18.  4. 10.  0.  0.]
 [45. 10. 10. 45.  0. 45. 10. 25.  0.  0. 45. 10. 25.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]]
multiplicação elemento-por-elemento = [81.  4. 10.  0.  0. 63. 10.  0.  0.  0. 81.  4. 25.  0.
0.]
produto matriz-vetor = [19.15825122 22.35571409 24.49820791]
 ----- Tempo de computação = 0.0ms
```

Exercício #9:

Implemente os cálculos da célula anterior mas de forma vetorizada. Para fazer isso você pode usar as seguintes funções da biblioteca numpy: `dot`, `outer`, `multiply` e o operador `*`.

In [26]:

```
# PARA VOCÊ FAZER: vetorização
import time

x1 = np.array([9, 2, 5, 0, 0, 7, 5, 0, 0, 0, 9, 2, 5, 0, 0])
x2 = np.array([9, 2, 2, 9, 0, 9, 2, 5, 0, 0, 9, 2, 5, 0, 0])

tic = time.process_time()

### PRODUTO ESCALAR VETORIZADO (x1 por x2) ###
### COMECE AQUI ### (≈ 1 lINHA)
dotv1 = np.dot(x1,x2)
### TERMINE AQUI ###
print ("produto escalar = " + str(dotv1))

### PRODUTO MATRICIAL DE DOIS VETORES VETORIZADO (x1 por x2) ###
### COMECE AQUI ### (≈ 1 lINHA)
```

```
outerv = np.outer(x1,x2)
### TERMINE AQUI ###
print ("produto de dois vetores = " + str(outerv))

### PRODUTO DE DOIS VETORES ELEMENTO-POR-ELEMENTO VETORIZADO (x1 por x2) ###
### COMECE AQUI ### (≈ 1 lINHA)
mulv = x1*x2
### TERMINE AQUI ###
print ("produto elemento-por-elemento = " + str(mulv))

### PRODUTO MATRIX-VETOR VETORIZADO (W por x1) ###
### COMECE AQUI ### (≈ 1 lINHA)
dotv2 = np.dot(W,x1)
### TERMINE AQUI ###

toc = time.process_time()
print (" produto matriz-vetor = " + str(dotv2) + "\n ----- Computation time = " + str(1000*(toc - t
ic)) + "ms")
```

```
produto escalar = 278
produto de dois vetores = [[81 18 18 81  0 81 18 45  0  0 81 18 45  0  0]
 [18  4  4 18  0 18  4 10  0  0 18  4 10  0  0]
 [45 10 10 45  0 45 10 25  0  0 45 10 25  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [63 14 14 63  0 63 14 35  0  0 63 14 35  0  0]
 [45 10 10 45  0 45 10 25  0  0 45 10 25  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [81 18 18 81  0 81 18 45  0  0 81 18 45  0  0]
 [18  4  4 18  0 18  4 10  0  0 18  4 10  0  0]
 [45 10 10 45  0 45 10 25  0  0 45 10 25  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]]
produto elemento-por-elemento = [81  4 10  0  0 63 10  0  0  0 81  4 25  0  0]
 produto matriz-vetor = [19.15825122 22.35571409 24.49820791]
 ----- Computation time = 0.0ms
```

Saída esperada:

```
produto escalar = 278
produto de dois vetores = [[81 18 18 81  0 81 18 45  0  0 81 18 45  0  0]
 [18  4  4 18  0 18  4 10  0  0 18  4 10  0  0]
 [45 10 10 45  0 45 10 25  0  0 45 10 25  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [63 14 14 63  0 63 14 35  0  0 63 14 35  0  0]
 [45 10 10 45  0 45 10 25  0  0 45 10 25  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [81 18 18 81  0 81 18 45  0  0 81 18 45  0  0]
 [18  4  4 18  0 18  4 10  0  0 18  4 10  0  0]
 [45 10 10 45  0 45 10 25  0  0 45 10 25  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]]
produto elemento-por-elemento = [81  4 10  0  0 63 10  0  0  0 81  4 25  0  0]
 produto matriz-vetor = [19.15825122 22.35571409 24.49820791]
 ----- Computation time = 0.0ms
```

Como se pode ver, a implementação vetorizada é muito mais simples e mais eficiente. Para tensores vetores/matrizes maiores a diferença de tempo computacional é ainda maior.

Note que a função `np.dot()` realiza uma multiplicação entre matriz-matriz ou matriz-vetor, ou ainda entre quaisquer tensores desde que as dimensões sejam compatíveis. A função `np.multiply()` e o operador `*` realizam multiplicação elemento-por-elemento.

2.1 - Implementação de funções de erro, ou de perda

Como visto em aula, funções de erro (ou de perda) são utilizadas em deep-learning para calcular o erro entre a saída esperada e a saída calculada pela rede.

Essas funções são usadas para avaliar a RNA. Quanto maior o erro, pior são as previsões ($ \hat{y} $) em relação aos valores reais (y).

Exercício #10:

Implemente uma versão vetorizada da função de erro absoluto:
$$\begin{align*} & L_1(\hat{y}, y) = \sum_{i=1}^m|y^{(i)} - \hat{y}^{(i)}| \end{align*}$$

Utilize as funções numpy `np.sum(x)` (soma de todos os elementos do vetor x) e `np.abs(x)` (valor absoluto dos elementos do vetor x).

In [27]:

```
# PARA VOCÊ FAZER: L1

def L1(yhat, y):
    """
    ArgumentOs:
    yhat -- vetor de dimensão m (valores previstos)
    y -- vetor de dimensão m (valores reais)

    Retorna:
    loss -- valor da função de erro L1
    """

    ### COMEÇE AQUI ### (≈ 1 linha)
    loss = np.sum(np.abs(y - yhat))
    ### TERMINE AQUI ###

    return loss
```

In [28]:

```
yhat = np.array([.9, 0.2, 0.1, .4, .9])
y = np.array([1, 0, 0, 1, 1])
print("L1 = " + str(L1(yhat,y)))
```

```
L1 = 1.1
```

Saída esperada:

```
L1 = 1.1
```

Exercício #11:

Implemente uma versão vetorizada da função de erro quadrático:
$$\begin{align*} & L_2(\hat{y},y) = \sum_{i=0}^m(y^{(i)} - \hat{y}^{(i)})^2 \end{align*}$$

Observação: Existem diversas formas de implementar essa função, mas talvez a forma mais fácil é usando a função numpy `np.dot`. Relembrando que, se $x = [x_1, x_2, ..., x_n]$, então `np.dot(x,x)` = $\sum_{j=1}^n x_j^{2}$.

In [50]:

```
# PARA VOCÊ FAZER: FUNÇÃO L2

def L2(yhat, y):
    """
    Argumentos:
    yhat -- vetor de dimensão mx1 (valores previstos)
    y -- vetor de dimensão mx1 (valores reais)

    Retorna:
    loss -- o valor da função de erro L2
    """
```

```
    ### COMEÇE AQUI ### (≈ 1 linha)
    loss = np.sum(np.dot(y-yhat,y-yhat))
    ### TERMINE AQUI ###

    return loss
```

In [51]:

```
yhat = np.array([.9, 0.2, 0.1, .4, .9])
y = np.array([1, 0, 0, 1, 1])
print("L2 = " + str(L2(yhat,y)))
```

```
L2 = 0.43
```

Saída esperada:

```
    L2 = 0.43
```

3 - Neurônio

Na Figura abaixo é apresentado um neurônio com 3 entradas.

As equações que descrevem o funcionamento de um neurônio foram vistas em aula e são as seguintes:

$$z = \mathbf{Wx} + b$$

$$a = g(z)$$

onde:

- z = estado do neurônio
- $\mathbf{W}$ = matriz de pesos das ligações dos neurônios de dimensão (1, nx)
- $\mathbf{x}$ = vetor de entradas do neurônio de dimensão (nx, 1)
- b = viés do neurônio
- a = ativação do neurônio
- $g()$ = função de ativação do neurônio.

Exercício #12:

Na célula abaixo defina uma função que implementa uma versão vetorizada para um único exemplo do funcionamento de um neurônio. Utilize a função `sigmoid(x)` feita no exercício #5. Observe que nessa função os parâmetros do neurônio ($\mathbf{W}$ e b) são passados em um dicionário. Passar parâmetros para funções em dicionários simplifica o código quando temos muitos parâmetros, tal como é o caso de uma RNA de várias camadas.

Para criar um dicionário de parametros usa-se, por exemplo, `parameters = {'W': w, 'b': b}`. Para recuperar os parâmetros do dicionário parameters você deve usar `parameters[".."]`.

In [70]:

```
# PARA VOCÊ FAZER: FUNÇÃO neuronio

def neuronio(x, parameters):
    """
    Argumentos:
    x = vetor de entradas de dimensão (1, nx)
    parameters = dicionário com parâmetros do neurônio contendo o vetor de pesos W de dimensão (1,
nx) e o viés b

    Retorna:
    a = ativação do neurônio
    """

    ### COMEÇE AQUI ### (≈ 2 linhas)
    # Recupere o vetor de pesos W e o viés b do dicionário par
    w = parameters["W"]
```

```
    b = parameters["b"]
    ### TERMINE AQUI ###

    ### COMEÇE AQUI ### (≈ 2 linhas)
    # Calcula a saída do neurônio tendo a sua entrada (x) e os seus parâmetros (W e b)
    z = np.dot(w,x) + b
    a = sigmoid(z)
    ### TERMINE AQUI ###

    return a
```

In [71]:

```
np.random.seed(13)
x = np.random.random((5,1))
w = np.random.random((1,5))
b = np.random.random((1))
parameters = {'W': w, 'b': b}

a = neuronio(x,parameters)
print('Ativação do neurônio=', a)
```

```
Ativação do neurônio= [[0.92368354]]
```

Saída esperada:

```
Ativação do neurônio= [[0.92368354]]
```

O que deve ser lembrado:

- Vectorização é muito importante em deep-learning. Ela fornece eficiência computacional, simplicidade e clareza;
- Algumas funções de erro bastante utilizadas;
- Familiarização com muitas funções numpy, tais como, np.dot, np.sum, np.multiply, np.maximum etc;
- Uso de dicionários facilita passar arqumentos para funções.