Lab 1 - BCC406/PCC177

REDES NEURAIS E APRENDIZAGEM EM PROFUNDIDADE

Pacote *NumPy*

Prof. Eduardo e Prof. Pedro

Objetivos:

- Uso de *NumPy*.

Data da entrega : 10/12/2024

- 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-LabX.pdf"
- Envie o PDF e o o *.ipynb* via Formulário Google.

Sugestão de leitura:

- Ler Capítulo 2 do livro texto. Dê ênfase para as seções 2.3 e 2.4. **Sugerimos fortemente** abrir com o Colab e executar estas duas seções passo a passo.

NumPy

NumPy é uma das bibliotecas mais populares para computação científica. Ela foi desenvolvida para dar suporte a operações com *arrays* de *N* dimensões e implementa métodos úteis para operações de álgebra linear, geração de números aleatórios, etc.

Criando arrays (5pt)

In [2]:
```
# Primeiramente, vamos importar a biblioteca
import numpy as np
```

In [ ]:
```
# Usaremos a função zeros para criar um array de uma dimensão de tamanho 5
np.zeros(5)
```

Out[ ]:
```
array([0., 0., 0., 0., 0.])
```

In [ ]:
```
# Da mesma forma, para criar um array de duas dimensões:
np.zeros((3,4))
```

Out[ ]:
```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

In [ ]:
```
# ToDo: Crie um array de três dimensões com o shape (3, 3, 3)
x = np.zeros((3,3,3))
x
```

Out[ ]:
```
array([[[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.]]])
```

Vocabulário comum (25pt)

- Em *NumPy*, cada dimensão é chamada eixo (***axis***).
- Um array é uma lista de axis e uma lista de tamanho dos axis é o que chamamos de **shape** do array.
  - Por exemplo, o shape da matrix acima é `(3, 4)`.
- O tamanho (***size***) de uma array é o número total de elementos, por exemplo, no array 2D acima = `3 * 4 = 12`.

In [ ]:
```
# Criando e mostrando o array
a = np.zeros((3,4))
a
```

Out[ ]:
```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

In [ ]:
```
# Verificando o shape do array
a.shape
```

Out[ ]:
```
(3, 4)
```

In [ ]:
```
# Verificando a quantidade de dimensões de um array
a.ndim
```

Out[ ]:
```
2
```

In [ ]:
```
# Verificando a quantidade de elemntos no array
a.size
```

Out[ ]:
```
12
```

In [ ]:
```
# ToDo : Criar um array de 3 dimensões, de shape (2,3,4) e mostrar o shape, quan
x = np.zeros((2,3,4))
print(f"Shape = {x.shape}\n")
print(f"Qtd de dimensões = {x.ndim}\n")
print(f"Número de elementos = {x.size}\n")
```

```
Shape = (2, 3, 4)

Qtd de dimensões = 3

Número de elementos = 24
```

In [ ]:
```
# ToDo : Criar um array de 3 dimensões mas trocando a função zeros por ones e mo
x = np.ones((3,3,3))
print(f"Shape = {x.shape}\n")
print(f"Qtd de dimensões = {x.ndim}\n")
print(f"Número de elementos = {x.size}\n")
```

```
Shape = (3, 3, 3)

Qtd de dimensões = 3

Número de elementos = 27
```

In [ ]:
```
# ToDo : Criar um array de 3 dimensões mas trocando a função zeros por full e mo
x = np.full((4,4,4), 4)
print(f"Shape = {x.shape}\n")
print(f"Qtd de dimensões = {x.ndim}\n")
print(f"Número de elementos = {x.size}\n")
```

```
Shape = (4, 4, 4)

Qtd de dimensões = 3

Número de elementos = 64
```

Out[ ]:
```
array([[[4, 4, 4, 4],
        [4, 4, 4, 4],
        [4, 4, 4, 4],
        [4, 4, 4, 4]],

       [[4, 4, 4, 4],
        [4, 4, 4, 4],
        [4, 4, 4, 4],
        [4, 4, 4, 4]],

       [[4, 4, 4, 4],
        [4, 4, 4, 4],
        [4, 4, 4, 4],
        [4, 4, 4, 4]],

       [[4, 4, 4, 4],
        [4, 4, 4, 4],
        [4, 4, 4, 4],
        [4, 4, 4, 4]]])
```

In [ ]:
```
# ToDo : Criar um array de 3 dimensões mas trocando a função zeros por empty e m
x = np.empty((4,4,4))
```

```
print(f"Shape = {x.shape}\n")
print(f"Qtd de dimensões = {x.ndim}\n")
print(f"Número de elementos = {x.size}\n")
x
```

```
Shape = (4, 4, 4)

Qtd de dimensões = 3

Número de elementos = 64
```

Out[ ]:
```
array([[[2.e-323, 2.e-323, 2.e-323, 2.e-323],
        [2.e-323, 2.e-323, 2.e-323, 2.e-323],
        [2.e-323, 2.e-323, 2.e-323, 2.e-323],
        [2.e-323, 2.e-323, 2.e-323, 2.e-323]],

       [[2.e-323, 2.e-323, 2.e-323, 2.e-323],
        [2.e-323, 2.e-323, 2.e-323, 2.e-323],
        [2.e-323, 2.e-323, 2.e-323, 2.e-323],
        [2.e-323, 2.e-323, 2.e-323, 2.e-323]],

       [[2.e-323, 2.e-323, 2.e-323, 2.e-323],
        [2.e-323, 2.e-323, 2.e-323, 2.e-323],
        [2.e-323, 2.e-323, 2.e-323, 2.e-323],
        [2.e-323, 2.e-323, 2.e-323, 2.e-323]],

       [[2.e-323, 2.e-323, 2.e-323, 2.e-323],
        [2.e-323, 2.e-323, 2.e-323, 2.e-323],
        [2.e-323, 2.e-323, 2.e-323, 2.e-323],
        [2.e-323, 2.e-323, 2.e-323, 2.e-323]]])
```

> **ToDo**: O que você pode dizer sobre cada uma das quatro funções que você usou?
>
> ```
>   A função np.zeros() recebe como parâmetro o shape desejado
> e retorna um array, com o shape informado, com todas as
> posições com o valor 0.
>   A função np.ones() recebe como parâmetro o shape desejado
> e retorna um array, com o shape informado, com todas as
> posições com o valor 1.
>   A função np.full() recebe como parâmetros o shape desejado
> e o valor em que se deseja iniciar todas as posições, e
> retorna um array, com o shape desejado e todas as posições
> com o valor informado.
>   A função np.empty() recebe como parâmetro o shape desejado
> e retorna um array, com o shape informado, e com todas as
> posições não-inicializadas, ou seja, com lixo de memória.
> ```

O comando ***np.arange*** (5pt)

Você pode criar um array usando a função arange, similar a função range do Python.

In [ ]:
```
# Criando um array
np.arange(1, 5)
```

Out[ ]:
```
array([1, 2, 3, 4])
```

In [ ]:
```
# Para criar com ponto flutuante
np.arange(1.0, 5.0)
```

Out[ ]:
```
array([1., 2., 3., 4.])
```

In [ ]:
```
# ToDo : crie um array com arange, variando de 1 a 5, com um passo de 0.5
x = np.arange(1, 5.5, 0.5)
x
```

Out[ ]:
```
array([1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5, 5. ])
```

Os comandos *np.rand* e *np.randn* (5pt)

O *NumPy* tem várias funções para criação de números aleatórios. Estas funções são muito úteis para inicialização dos pesos das redes neurais. Por exemplo, abaixo criamos uma matrix `(3, 4)` inicializada com números em ponto flutuante (*floats*) e distribuição uniforme:

In [ ]:
```
np.random.rand(3,4)
```

Out[ ]:
```
array([[2.46843515e-01, 7.94475662e-01, 6.38473436e-01, 4.78734788e-01],
       [9.48465164e-01, 8.42491926e-01, 2.89465588e-01, 7.62775910e-01],
       [3.36609671e-01, 9.04284383e-01, 3.69774104e-01, 4.15132077e-04]])
```

Abaixo um matriz inicializada com distribuição gaussiana (normal distribution) com média 0 e variância 1

In [ ]:
```
np.random.randn(3,4)
```

Out[ ]:
```
array([[ 2.38260871, -0.01632657,  1.29354616,  0.68924774],
       [ 0.13309159, -0.09422064, -0.27884781,  1.07440676],
       [-1.33768807, -1.15160529,  2.04455882,  0.64608241]])
```

In [ ]:
```
# ToDo : crie um array aleatório com o shape (1, 2, 3, 4)
x = np.random.rand(1,2,3,4)
x
```

Out[ ]:
```
array([[[[0.99647359, 0.9638146 , 0.18090359, 0.29825922],
         [0.87088971, 0.57648185, 0.9896967 , 0.8498117 ],
         [0.31449193, 0.94317867, 0.49077926, 0.42229894]],

        [[0.38185164, 0.32098448, 0.44432428, 0.79848438],
         [0.32210363, 0.37269148, 0.89829264, 0.62315878],
         [0.99902568, 0.57644136, 0.19501458, 0.82061651]]]])
```

A biblioteca *MatplotLib* (5pt)

Vamos usar a biblioteca matplotlib (para mais detalhes veja o tutorial de *matplotlib*) para plotar dois arrays de tamanho 10.000, um inicializado com distribuição normal e o outro com uniforme

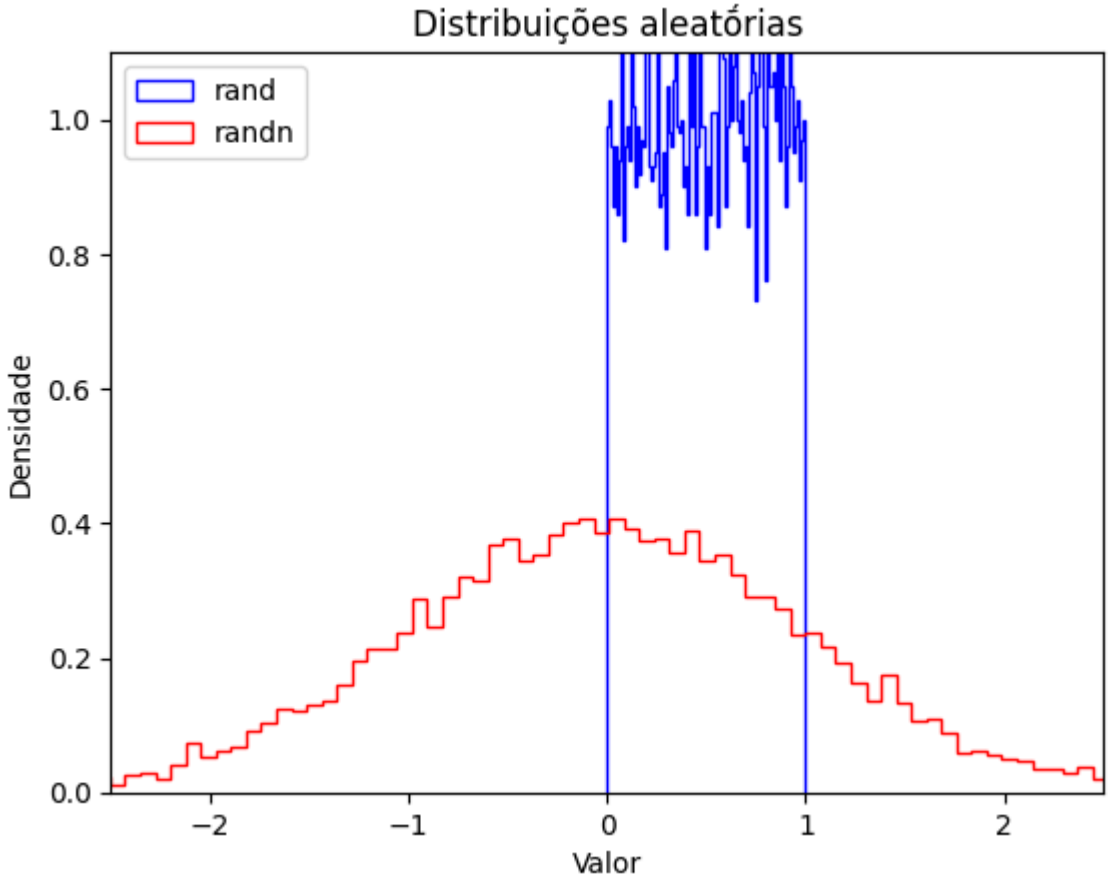
In [ ]:
```
%matplotlib inline
import matplotlib.pyplot as plt
```

Primeiro os dados que serão plotados precisam ser criados

In [ ]:
```
array_a = np.random.rand(10000,)# ToDo : criar um array de shape (10.000,)
array_b = np.random.randn(10000,)# ToDo : criar um array de shape (10.000,)
```

Depois eles podem ser plotados

In [ ]:
```
plt.hist(array_a, density=True, bins=100, histtype="step", color="blue", label="
plt.hist(array_b, density=True, bins=100, histtype="step", color="red", label="r
plt.axis([-2.5, 2.5, 0, 1.1])
plt.legend(loc = "upper left")
plt.title("Distribuições aleatórias")
plt.xlabel("Valor")
plt.ylabel("Densidade")
plt.show()
```



Tipo de dados (*dtype*) (5pt)

Você pode ver qual o tipo de dado pelo atributo `dtype`. Verifique abaixo:

In [ ]:
```
c = np.arange(1, 5)
print(c.dtype, c)
```

```
int64 [1 2 3 4]
```

In [ ]:
```
# ToDo: Crie um um array aleatório de shape (2, 3, 4) e verifique o seu tipo
x = np.random.rand(2,3,4)
print(x.dtype, x)
```

```
float64 [[[0.44689675 0.26727742 0.84231738 0.77175053]
  [0.77454903 0.58649886 0.74877748 0.99664755]
  [0.00602587 0.85935444 0.64803404 0.54161591]]

 [[0.84985727 0.9160355  0.06144082 0.64668953]
  [0.50428518 0.04566941 0.08643905 0.24791305]
  [0.91071396 0.8374396  0.62906173 0.87078221]]]
```

Tipos disponíveis: `int8`, `int16`, `int32`, `int64`, `uint8 | 16 | 32 | 64`, `float16 | 32 | 64` e `complex64 | 128`. Veja a documentação para a lista completa.

Atributo *itemsize* (5pt)

O atributo `itemsize` retorna o tamanho em bytes

In [ ]:
```
e = np.arange(1, 5, dtype=np.complex64)
e.itemsize
```

Out[ ]:
```
8
```

In [ ]:
```
# Na memória, um array é armazenado de forma contígua
f = np.array([[1,2],[1000, 2000]], dtype=np.int32)
f.data
```

Out[ ]:
```
<memory at 0x7e0b26365490>
```

In [ ]:
```
# ToDo: Crie arrays de shape (2, 2) dos tipos int8, int64, float16, float64, com
x = np.array([[1,2], [3,4]], dtype=np.int8)
print(x.dtype, x.itemsize)
x = np.array([[1,2], [3,4]], dtype=np.int64)
print(x.dtype, x.itemsize)
x = np.array([[1,2], [3,4]], dtype=np.float16)
print(x.dtype, x.itemsize)
x = np.array([[1,2], [3,4]], dtype=np.float64)
print(x.dtype, x.itemsize)
x = np.array([[1,2], [3,4]], dtype=np.complex64)
print(x.dtype, x.itemsize)
x = np.array([[1,2], [3,4]], dtype=np.complex128)
print(x.dtype, x.itemsize)
```

```
int8 1
int64 8
float16 2
float64 8
complex64 8
complex128 16
```

> **ToDo**: O que você pode dizer sobre esses arrays criados?
>
> Os itens de cada arrays são semelhantes, tendo em vista que o mesmo vetor original foi informado, entretanto, de acordo com o tipo de dados informado no dtype, os arrays ocupam

> espaços em memórias diferentes. Por exemplo, o array com o dtype int8 ocupa mens espaço que o float64.

Reshaping (5pt)

Alterar o shape de uma array é muto fácil com NumPy e muito útil para adequação das matrizes para métodos de machine learning. Contudo, o tamanho (size) não pode ser alterado.

In [ ]:
```
# O núemro de dimensões também é chamado de rank
g = np.arange(24)
print(g)
print("Rank:", g.ndim)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
Rank: 1
```

In [ ]:
```
g.shape = (6, 4)
print(g)
print("Rank:", g.ndim)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
Rank: 2
```

In [ ]:
```
g.shape = (2, 3, 4)
print(g)
print("Rank:", g.ndim)
```

```
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
Rank: 3
```

Mudando o formato do dado (***reshape***)

In [ ]:
```
g2 = g.reshape(4,6)
print(g2)
print("Rank:", g2.ndim)
```

```
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]]
Rank: 2
```

In [ ]:
```
# Pode-se alterar diretamente um item da matriz, pelo índice
g2[1, 2] = 999
g2
```

Out[ ]:
```
array([[  0,   1,   2,   3,   4,   5],
       [  6,   7, 999,   9,  10,  11],
       [ 12,  13,  14,  15,  16,  17],
       [ 18,  19,  20,  21,  22,  23]])
```

Repare que o objeto `'g'` foi modificado também!

In [ ]:
```
g
```

Out[ ]:
```
array([[[  0,   1,   2,   3],
        [  4,   5,   6,   7],
        [999,   9,  10,  11]],

       [[ 12,  13,  14,  15],
        [ 16,  17,  18,  19],
        [ 20,  21,  22,  23]]])
```

Todas a operçãoes aritméticas comuns podem ser feitas com o *ndarray*

In [ ]:
```
a = np.array([14, 23, 32, 41])
b = np.array([5,  4,  3,  2])
print("a + b  =", a + b)
print("a - b  =", a - b)
print("a * b  =", a * b)
print("a / b  =", a / b)
print("a // b  =", a // b)
print("a % b  =", a % b)
print("a ** b =", a ** b)
```

```
a + b  = [19 27 35 43]
a - b  = [ 9 19 29 39]
a * b  = [70 92 96 82]
a / b  = [ 2.8         5.75       10.66666667 20.5       ]
a // b  = [ 2  5 10 20]
a % b  = [4 3 2 1]
a ** b = [537824 279841  32768   1681]
```

Repare que a multiplicação acima **NÃO** é um multiplicação de martizes

Arrays devem ter o mesmo shape, caso contrário, NumPy vai aplicar a regra de *broadcasting* (Ver seção 2.1.3 do livro texto). Pesquise sobre a operação ed bradcasting do NumPy e explique com suas palavras, abaixo:

> **ToDo**: Explique o conceito de *broadcasting*.
>
> O broadcasting diz respeito à uma técnica para a realização de operações binárias entre arrays com shapes distintos. Neste caso, o NumPy expande os arrays para que tenham shapes iguais, copiando elementos para as colunas ou linhas. Por exemplo, suponha um array X com shape (1,3) e um array Y com shape (4,1). O Numpy, neste exemplo, fará com que todos os arrays tenham o shape (4,3), que seria a maior quantidade de linhas entre os shapes (4) e a maior quantidade de colunas (3). No array X, os elementos serão copiados para novas linhas e no array Y serão copiados para novas colunas. Desse modo, é possível efetuar as operações binárias, após utilizar o broadcasting.

Iteração e Concatenação de arrays de *NumPy* (5pt)

Repare que você pode iterar pelos `ndarrays`, e que ela é feita pelos *axis*.

In [ ]:
```
c = np.arange(24).reshape(2, 3, 4)  # Um array 3D (coposto de duas matrizes de 3
c
```

Out[ ]:
```
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])
```

In [ ]:
```
for m in c:
    print("Item:")
    print(m)
```

```
Item:
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
Item:
[[12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
```

In [ ]:
```
for i in range(len(c)):  # Observe que len(c) == c.shape[0]
    print("Item:")
    print(c[i])
```

```
Item:
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
Item:
[[12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
```

In [ ]:
```
# Para iterar por todos os elementos
for i in c.flat:
    print("Item:", i)
```

```
Item: 0
Item: 1
Item: 2
Item: 3
Item: 4
Item: 5
Item: 6
Item: 7
Item: 8
Item: 9
Item: 10
Item: 11
Item: 12
Item: 13
Item: 14
Item: 15
Item: 16
Item: 17
Item: 18
Item: 19
Item: 20
Item: 21
Item: 22
Item: 23
```

Também é possível concatenar `ndarrays`, e isso pode ser feito em um eixo específico.

In [ ]:
```
# Pode-se concatenar arrays pelos axis
q1 = np.full((3,4), 1.0)

q2 = np.full((4,4), 2.0)

q3 = np.full((3,4), 3.0)

q = np.concatenate((q1, q2, q3), axis=0)
q
```

Out[ ]:
```
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [2., 2., 2., 2.],
       [2., 2., 2., 2.],
       [2., 2., 2., 2.],
       [2., 2., 2., 2.],
       [3., 3., 3., 3.],
       [3., 3., 3., 3.],
       [3., 3., 3., 3.]])
```

In [ ]:
```
# ToDo: imprima o shape resultante da concatenação dos arrays de shape a = (2, 3
a = np.full((2,3,4), 2.0)
b= np.full((2,3,4), 3.0)

ab0 = np.concatenate((a,b), axis=0)
ab1 = np.concatenate((a,b), axis=1)
ab2 = np.concatenate((a,b), axis=2)

print(f"Shape da concatenação a e b no eixo 0 = {ab0.shape}\n")
print(f"Shape da concatenação a e b no eixo 1 = {ab1.shape}\n")
print(f"Shape da concatenação a e b no eixo 2 = {ab2.shape}\n")
```

```
Shape da concatenação a e b no eixo 0 = (4, 3, 4)

Shape da concatenação a e b no eixo 1 = (2, 6, 4)

Shape da concatenação a e b no eixo 2 = (2, 3, 8)
```

Transposta (5pt)

In [ ]:
```
m1 = np.arange(10).reshape(2,5)
m1
```

Out[ ]:
```
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

In [ ]:
```
# ToDo : imprima a matriz transposta de m1
m1t = m1.T
for m in m1t:
  print(m)
```

```
[0 5]
[1 6]
[2 7]
[3 8]
[4 9]
```

Produto de matrizes (5pt)

In [ ]:
```
n1 = np.arange(10).reshape(2, 5)
n1
```

Out[ ]:
```
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

In [ ]:
```
n2 = np.arange(15).reshape(5, 3)
n2
```

Out[ ]:
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])
```

In [ ]:
```
n1.dot(n2)
```

Out[ ]:
```
array([[ 90, 100, 110],
       [240, 275, 310]])
```

In [ ]:
```
# ToDo: Crie um array de 1 a 25 com shape (5, 5) e faça a multiplicação por uma
a1 = np.arange(1, 26).reshape(5,5)
a2 = np.zeros((5,1))

a3 = a1.dot(a2)
a3
```

Out[ ]:
```
array([[0.],
       [0.],
       [0.],
       [0.],
       [0.]])
```

Matriz Inversa (5pt)

In [16]:
```
import numpy.linalg as linalg

m3 = np.array([[1,2,3],[5,7,11],[21,29,31]])

m3
```

Out[16]:
```
array([[ 1,  2,  3],
       [ 5,  7, 11],
       [21, 29, 31]])
```

In [ ]:
```
linalg.inv(m3)
```

Out[ ]:
```
array([[-2.31818182,  0.56818182,  0.02272727],
       [ 1.72727273, -0.72727273,  0.09090909],
       [-0.04545455,  0.29545455, -0.06818182]])
```

> **ToDo**: O que a função `linalg.inv` faz?
>
> A função linalg.inv recebe como parâmetro uma matriz quadrada e retorna a sua matriz inversa.

Matriz identidade (5pt)

In [ ]:
```
m3.dot(linalg.inv(m3))
```

Out[ ]:
```
array([[ 1.00000000e+00, -1.66533454e-16,  0.00000000e+00],
       [ 6.31439345e-16,  1.00000000e+00, -1.38777878e-16],
       [ 5.21110932e-15, -2.38697950e-15,  1.00000000e+00]])
```

In [17]:
```
## ToDo: Crie uma matriz identidade de tamanho (5, 5)
#Propriedade da matriz inversa
while True:
  a1 = np.random.randint(0, 20, (5,5))
  if(np.linalg.det(a1) != 0):
    break

id = a1.dot(linalg.inv(a1))
print(id)

#Concatenação de arrays
identidade = np.zeros((1,5))
identidade[0][0] = 1

for i in range(4):
   temp = np.zeros((1, 5))
   temp[0][i+1] = 1
   identidade = np.concatenate((identidade, temp), axis=0)
print(identidade)
```

```
[[ 1.00000000e+00 -4.44089210e-16 -1.66533454e-16  4.99600361e-16
  -2.42861287e-17]
 [ 1.11022302e-16  1.00000000e+00 -1.38777878e-16 -5.55111512e-17
   3.98986399e-16]
 [ 0.00000000e+00 -2.22044605e-16  1.00000000e+00  5.55111512e-16
   5.34294831e-16]
 [-1.11022302e-16  2.22044605e-16  0.00000000e+00  1.00000000e+00
   1.94289029e-16]
 [-1.11022302e-16 -2.22044605e-16 -5.55111512e-17  0.00000000e+00
   1.00000000e+00]]
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
```

Exercícios (15pt)

Para os próximos exercícios, use o numpy.

Questão 1: Escreva uma função recursiva que calcule o determinante de uma matriz $n \times n$ usando o teorema de Laplace.

In [3]:
```
def deleterowandcol(mat, row, col):
  matWithoutRow = np.delete(mat, row, axis = 0)
  matFinal = np.delete(matWithoutRow, col, axis=1)
  return matFinal

def laplace(mat):
  if mat.shape[0] == 1:
    return mat[0][0]

  else:
    det = 0
    for i in range(mat.shape[0]):
      det += mat[0][i] * ((-1) ** (1+(i+1))) * laplace(deleterowandcol(mat, 0, i
    return det

a = np.array([[4,5,-3,0], [2,-1,3,1], [1,-3,2,1], [0,2,-2,5]])
print(laplace(a))
```

```
210
```

Questão 2: Escreva um programa que calcule a solução de um sistema de equações lineares por meio da regra de Cramer. Seu programa deve iniciar lendo o numero de equações e variáveis, e, logo após, ler as matrizes de entrada do teclado coeficiente a coeficiente. Para o cálculo dos determinantes, você pode utilizar a função `det` do pacote `numPy`.

In [34]:
```
def cramer():

  neq = int(input("Digite o numero de equacoes e variaveis: "))
  mat = np.zeros((neq, neq+1)) #Matriz de entrada englobando os coef das incógni

  for i in range(neq):
    for j in range(neq+1):
      mat[i][j] = float(input(f"Informe o coeficiente da linha {i+1} e da coluna
```

```
  detmat = laplace(mat)

  terind = mat[:,neq].copy()

  mat = np.delete(mat, neq, axis=1)

  sol = np.zeros((neq))

  for i in range(neq):
    oldcol = mat[:, i].copy() #Copiando coluna para registro
    mat[:, i] = terind
    sol[i] = laplace(mat) / detmat
    mat[:, i] = oldcol

  print(f"Vetor solucao: {sol}\n")

cramer()
```

```
Digite o numero de equacoes e variaveis: 2
Informe o coeficiente da linha 1 e da coluna 1: 1
Informe o coeficiente da linha 1 e da coluna 2: 2
Informe o coeficiente da linha 1 e da coluna 3: 3
Informe o coeficiente da linha 2 e da coluna 1: 1
Informe o coeficiente da linha 2 e da coluna 2: 4
Informe o coeficiente da linha 2 e da coluna 3: 5
Vetor solucao: [1. 1.]
```

Questão 3: Implemente uma função que resolva sistemas de equações lineares através do método de eliminação de Gauss. Rode a função para algum exemplo, e compare com a solução obtida com o código da questão 2. Meça o tempo de execução para verificar qual algoritmo executa mais rápido.

In [138…

```
import time

def gauss(mat):
  mat = np.array(mat, dtype=np.float64)
  #Redução por Gauss
  num_pivo = 0
  for i in range(mat.shape[0]-1): #Colunas a serem zeradas
    for j in range(mat.shape[0]): #Linhas
      if(j > num_pivo):
        fator = float(mat[j][i] / mat[i][i])
        for k in range(mat.shape[1]):
          mat[j][k] = float(mat[j][k] - (fator * mat[i][k]))
    num_pivo += 1

  #Substituição regressiva
  sol = np.zeros(mat.shape[0])
  for i in range(mat.shape[0] - 1, -1 , -1):
    soma = 0
    #Soluções conhecidas
    for j in range(i+1, mat.shape[0]):
      soma += mat[i][j] * sol[j]

    sol[i] = (mat[i][mat.shape[1] - 1] - soma) / mat[i][i]

  return sol
```

```
#Função anterior adaptada para receber array via parâmetro ao invés do teclado,
def cramer(mat):
  neq = mat.shape[0]
  detmat = laplace(mat)

  terind = mat[:,neq].copy()
  mat = np.delete(mat, neq, axis=1)

  sol = np.zeros((neq))
  for i in range(neq):
    oldcol = mat[:, i].copy() #Copiando coluna para registro
    mat[:, i] = terind
    sol[i] = laplace(mat) / detmat
    mat[:, i] = oldcol

  return sol

#Medição do tempo de execução
start_cramer = time.time()
print(cramer(np.array([[1,1,1,6], [4,2,-1,5], [1,3,2,13]])))
end_cramer = time.time()

start_gauss = time.time()
print(gauss(np.array([[1,1,1,6], [4,2,-1,5], [1,3,2,13]])))
end_gauss = time.time()

cramer_time = end_cramer -  start_cramer
gauss_time = end_gauss -  start_gauss

print(f"Tempo do Cramer = {cramer_time}s\nTempo do Gauss = {gauss_time}s\n")
```

```
[1. 2. 3.]
[1. 2. 3.]
Tempo do Cramer = 0.0009992122650146484s
Tempo do Gauss = 0.00041031837463378906s
```

Análise: Em todos os testes executados, o tempo do Gauss foi menor em relação ao tempo do Cramer, estando na mesma casa decimal. Portanto, infere-se que a função do Gauss executa mais rápido que a função do Cramer para uma mesma entrada.