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

In [40]:
```
# Usaremos a função zeros para criar um array de uma dimensão de tamanho 5
np.zeros(5)
```

Out[40]:
```
array([0., 0., 0., 0., 0.])
```

In [41]:
```
# Da mesma forma, para criar um array de duas dimensões:
np.zeros((3,4))
```

Out[41]:
```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

In [42]:
```
# ToDo: Crie um array de três dimensões com o shape (3, 3, 3)
```

In [43]:
```
np.zeros((3,3,3))
```

Out[43]:
```
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 [44]:
```
# Criando e mostrando o array
a = np.zeros((3,4))
a
```

Out[44]:
```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

In [45]:
```
# Verificando o shape do array
a.shape
```

Out[45]:
```
(3, 4)
```

In [46]:
```
# Verificando a quantidade de dimensões de um array
a.ndim
```

Out[46]:
```
2
```

In [47]:
```
# Verificando a quantidade de elemntos no array
a.size
```

Out[47]:
```
12
```

In [48]:
```
# ToDo : Criar um array de 3 dimensões, de shape (2,3,4) e mostrar o shape, quan
```

In [49]:
```
b = np.zeros((2, 3, 4))
print(f"Shape: {b.shape}\n")
print(f"Dimensão: {b.ndim}\n")
print(f"Tamanho: {b.size}\n")
b
```

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

Dimensão: 3

Tamanho: 24
```

Out[49]:
```
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.]]])
```

In [50]:
```
# ToDo : Criar um array de 3 dimensões mas trocando a função zeros por ones e mo
```

In [51]:
```
c = np.ones((2, 3, 4))
print(f"Shape: {c.shape}\n")
print(f"Dimensão: {c.ndim}\n")
print(f"Tamanho: {c.size}\n")
c
```

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

Dimensão: 3

Tamanho: 24
```

Out[51]:
```
array([[[1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]],

       [[1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]]])
```

In [52]:
```
# ToDo : Criar um array de 3 dimensões mas trocando a função zeros por full e mo
```

In [53]:
```
d = np.full((2, 3, 4), 5)
print(f"Shape: {d.shape}\n")
print(f"Dimensão: {d.ndim}\n")
print(f"Tamanho: {d.size}\n")
d
```

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

Dimensão: 3

Tamanho: 24
```

Out[53]:
```
array([[[5, 5, 5, 5],
        [5, 5, 5, 5],
        [5, 5, 5, 5]],

       [[5, 5, 5, 5],
        [5, 5, 5, 5],
        [5, 5, 5, 5]]])
```

In [54]:
```
# ToDo : Criar um array de 3 dimensões mas trocando a função zeros por empty e m
```

In [55]:
```
e = np.empty((2, 3, 4))
print(f"Shape: {e.shape}\n")
print(f"Dimensão: {e.ndim}\n")
print(f"Tamanho: {e.size}\n")
e
```

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

Dimensão: 3

Tamanho: 24
```

Out[55]:
```
array([[[1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]],

       [[1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]]])
```

> **ToDo**: O que você pode dizer sobre cada uma das quatro funções que você usou?
>
> Sua resposta aqui....

As funções que eu acabei de usar são interessantes pois são diferentes formas de inicializar o array conforme a minha demanda, poe exemplo se eu precisar do array todo nulo, eu usarei o np.zeros, já no caso de um array completo por 1's usaria o np.ones. O np.full é semelhante ao zeros e ones, porém vc passa o valor do número que deseja adicionar no array, por exemplo eu usei o 5. Como eu falei dependendo do meu uso cada uma tem seu valor.

O comando *np.arange* (5pt)

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

In [56]:
```
# Criando um array
np.arange(1, 5)
```

Out[56]:
```
array([1, 2, 3, 4])
```

In [57]:
```
# Para criar com ponto flutuante
np.arange(1.0, 5.0)
```

Out[57]:
```
array([1., 2., 3., 4.])
```

In [58]:
```
# ToDo : crie um array com arange, variando de 1 a 5, com um passo de 0.5
```

In [59]:
```
np.arange(1, 5.5, 0.5)
```

Out[59]:
```
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 [60]:
```
np.random.rand(3,4)
```

Out[60]:
```
array([[0.95441049, 0.72170377, 0.2864863 , 0.14076716],
       [0.23222781, 0.55673637, 0.24042562, 0.94438157],
       [0.31416027, 0.16304048, 0.41376027, 0.75111031]])
```

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

In [61]:
```
np.random.randn(3,4)
```

Out[61]:
```
array([[-1.61941992,  0.20644194, -0.96799141, -0.06393051],
       [ 1.1703297 ,  0.70827313, -0.14562783,  0.50308084],
       [-0.88237077, -0.57467522, -0.1457452 , -0.51215329]])
```

In [62]:
```
# ToDo : crie um array aleatório com o shape (1, 2, 3, 4)
```

In [63]:
```
np.random.rand(1, 2, 3, 4)
```

Out[63]:
```
array([[[[0.08204467, 0.45847662, 0.48977588, 0.09914683],
         [0.77743048, 0.03470154, 0.45572873, 0.36941915],
         [0.57788029, 0.78005257, 0.54333358, 0.89878734]],

        [[0.01051648, 0.24219217, 0.78061378, 0.55407951],
         [0.15539152, 0.23569381, 0.87634245, 0.8568265 ],
         [0.50419027, 0.00754336, 0.98255848, 0.97704045]]]])
```

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

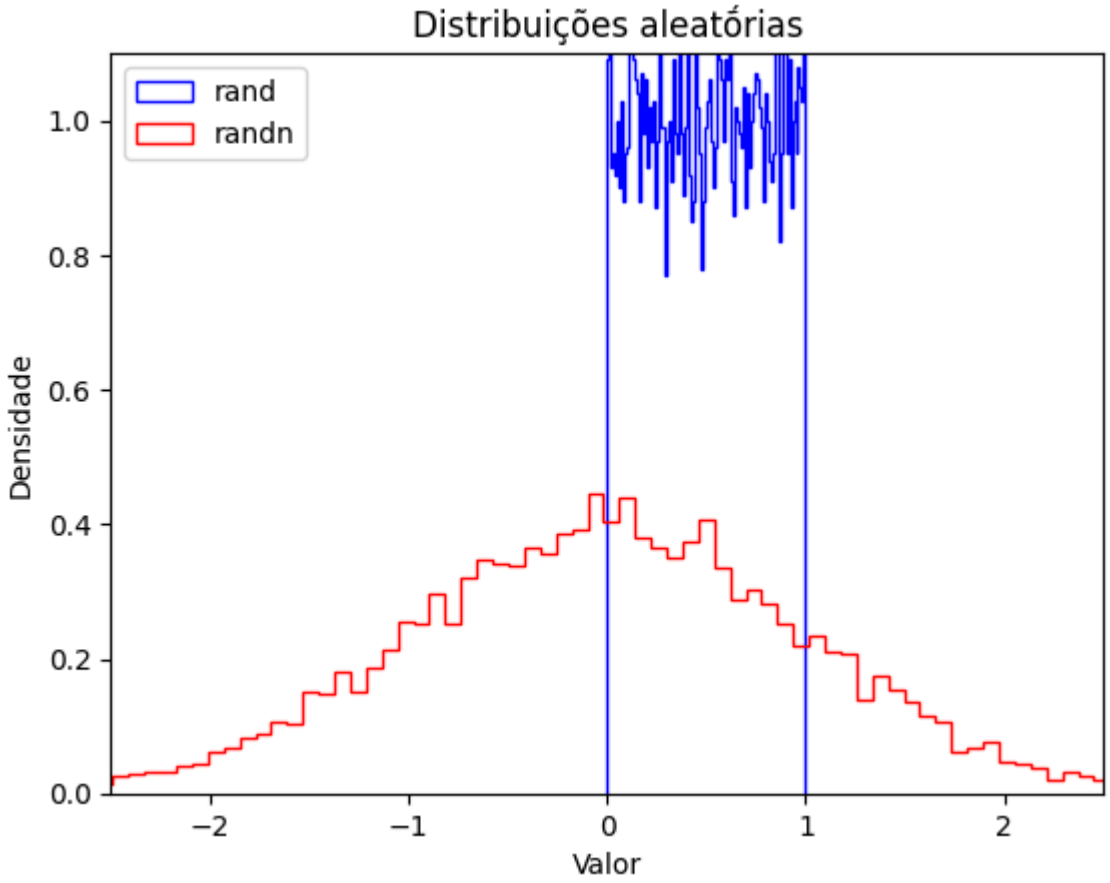
In [64]:
```
%matplotlib inline
import matplotlib.pyplot as plt
```

Primeiro os dados que serão plotados precisam ser criados

In [65]:
```
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 [66]:
```
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 [67]:
```
c = np.arange(1, 5)
print(c.dtype, c)
```

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

In [68]:
```
# ToDo: Crie um um array aleatório de shape (2, 3, 4) e verifique o seu tipo
```

In [69]:
```
x = np.random.rand(2, 3, 4)
print(x.dtype, x)
```

```
float64 [[[0.91119438 0.7004791  0.33683053 0.15013194]
  [0.30309552 0.23478443 0.94922687 0.86527201]
  [0.45722209 0.54408782 0.32619139 0.0761196 ]]

 [[0.64693001 0.69106466 0.90290852 0.93348144]
  [0.54952    0.60018052 0.97101252 0.40684378]
  [0.52026368 0.11205183 0.72690992 0.59551458]]]
```

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 [70]:
```
e = np.arange(1, 5, dtype=np.complex64)
e.itemsize
```

Out[70]:
```
8
```

In [71]:
```
# Na memória, um array é armazenado de forma contígua
f = np.array([[1,2],[1000, 2000]], dtype=np.int32)
f.data
```

Out[71]:
```
<memory at 0x7d09c919d490>
```

In [72]:
```
# ToDo: Crie arrays de shape (2, 2) dos tipos int8, int64, float16, float64, com
```

In [73]:
```
a = np.array([[1, 2], [3, 4]], dtype=np.int8)
print(a.dtype)

a = np.array([[1, 2], [3, 4]], dtype=np.int64)
print(a.dtype)

a = np.array([[1, 2], [3, 4]], dtype=np.float64)
print(a.dtype)

a = np.array([[1, 2], [3, 4]], dtype=np.complex64)
print(a.dtype)

a = np.array([[1, 2], [3, 4]], dtype=np.complex128)
print(a.dtype)
```

```
int8
int64
float64
complex64
complex128
```

> **ToDo**: O que você pode dizer sobre esses arrays criados?
>
> Sua resposta aqui...

De novo como eu havia dito acima, são armas poderosas pois dependendo do que preciso fazer é de extrema importância poder estipular os tipos do array, como python é

uma linguagem que não é fortemente tipada, as vezes faz falta.

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*.
>
> Sua resposta aqui...

O mecanismo de broadcasting funciona ajustando inicialmente as formas dos arrays para que fiquem compatíveis. Isso é feito repetindo os valores ao longo dos eixos cujo comprimento seja igual a 1, garantindo que os dois arrays tenham a mesma forma após essa etapa. Em seguida, realiza-se a operação desejada diretamente entre os elementos correspondentes dos arrays ajustados, como se ambos tivessem originalmente o mesmo formato.

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

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

In [74]:
```
c = np.arange(24).reshape(2, 3, 4)  # Um array 3D (coposto de duas matrizes de 3
c
```

Out[74]:
```
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 [75]:
```
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 [76]:
```
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 [77]:
```
# 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 [78]:
```
# 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[78]:
```
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 [79]:
```
# ToDo: imprima o shape resultante da concatenação dos arrays de shape a = (2, 3
```

In [ ]:
```
a = np.full((2, 3, 4), 1.0)

b = np.full((2, 3, 4), 2.0)

c = np.concatenate((a, b), axis=0)
print(c)
```

```
[[[1. 1. 1. 1.]
  [1. 1. 1. 1.]
  [1. 1. 1. 1.]]

 [[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.]
  [2. 2. 2. 2.]
  [2. 2. 2. 2.]]]
```

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
```

In [ ]:
```
m2 = m1.T
m2
```

Out[ ]:
```
array([[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 [80]:
```
# ToDo: Crie um array de 1 a 25 com shape (5, 5) e faça a multiplicação por uma
```

In [81]:
```
n1 = np.arange(1, 26).reshape(5,5)
n2 = np.zeros((5, 1))

n1.dot(n2)
```

Out[81]:
```
array([[0.],
       [0.],
       [0.],
       [0.],
       [0.]])
```

Matriz Inversa (5pt)

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

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

m3
```

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

In [25]:
```
linalg.inv(m3)
```

Out[25]:
```
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?
>
> Sua resposta aqui...

A função linalg.inv, calcula a inversa de uma matriz quadrada.

Matriz identidade (5pt)

In [26]:
```
m3.dot(linalg.inv(m3))
```

Out[26]:
```
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 [82]:
```
# ToDo: Crie uma matriz identidade de tamanho (5, 5)
```

In [27]:
```
identidade = np.eye(5)
identidade
```

Out[27]:
```
array([[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 [83]:
```
# Seu código aqui

def determinante_laplace(matriz):
    n, m = matriz.shape
    if n != m:
        raise ValueError("A matriz deve ser quadrada.")
    if n == 1:
        return matriz[0, 0]
    if n == 2:
        return matriz[0, 0] * matriz[1, 1] - matriz[0, 1] * matriz[1, 0]
    det = 0
    for i in range(n):
        submatriz = np.delete(np.delete(matriz, 0, axis=0), i, axis=1)
        cofator = ((-1) ** i) * matriz[0, i] * determinante_laplace(submatriz)
        det += cofator
    return det

matriz1 = np.array([[-2, 3, 1, 7], [0, -1, 2, 1], [3, -4, 5, 1], [1, 0, -2, -1]])
print(matriz1)
resultado = determinante_laplace(matriz1)
print(f"Determinante: {resultado}")

matriz2 = np.array(([1, 0, 5, 0], [2, -1, 0, 3], [3, 0 , 2, 0], [7, 0, 6, 5]))
print(matriz2)
result = determinante_laplace(matriz2)
print(f"Determinante: {result}")
```

```
[[-2  3  1  7]
 [ 0 -1  2  1]
 [ 3 -4  5  1]
 [ 1  0 -2 -1]]
Determinante: 24
[[ 1  0  5  0]
 [ 2 -1  0  3]
 [ 3  0  2  0]
 [ 7  0  6  5]]
Determinante: 65
```

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 [ ]:
```
# Seu código aqui

def regra_de_cramer():
    print("Regra de Cramer: Solução de um sistema de equações lineares.")
    n = int(input("Digite o número de equações (e variáveis): "))
```

```
    print("Digite os coeficientes da matriz (linha por linha):")
    matriz = np.array([[float(input(f"a[{i+1}][{j+1}]: ")) for j in range(n)] fo

    print("Digite os termos independentes:")
    termos_independentes = np.array([float(input(f"b[{i+1}]: ")) for i in range(
    det_principal = determinante_laplace(matriz)

    if det_principal == 0:
        print("O sistema não possui solução única, pois o determinante é zero.")
        return

    solucoes = []
    for i in range(n):
        matriz_alterada = matriz.copy()
        matriz_alterada[:, i] = termos_independentes
        det_alterado = determinante_laplace(matriz_alterada)
        solucoes.append(det_alterado / det_principal)
    print("As soluções do sistema são:")
    for i, sol in enumerate(solucoes):
        print(f"x[{i+1}] = {sol}")

regra_de_cramer()
```

```
Regra de Cramer: Solução de um sistema de equações lineares.
```

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 [39]:
```
# Seu código aqui

import time

def eliminacao_de_gauss(matriz, termos_independentes):
    n = len(termos_independentes)
    A = np.copy(matriz)
    b = np.copy(termos_independentes)

    for i in range(n):
        if A[i, i] == 0:
            for k in range(i+1, n):
                if A[k, i] != 0:
                    A[[i, k]] = A[[k, i]]
                    b[[i, k]] = b[[k, i]]
                    break

        for j in range(i+1, n):
            fator = A[j, i] / A[i, i]
            A[j, i:] -= fator * A[i, i:]
            b[j] -= fator * b[i]

    x = np.zeros(n)
    for i in range(n-1, -1, -1):
        x[i] = (b[i] - np.dot(A[i, i+1:], x[i+1:])) / A[i, i]

    return x
```

```python
def regra_de_cramer(matriz, termos_independentes):
    n = len(termos_independentes)
    det_principal = np.linalg.det(matriz)
    if det_principal == 0:
        raise ValueError("O sistema não possui solução única.")
    solucoes = []
    for i in range(n):
        matriz_alterada = matriz.copy()
        matriz_alterada[:, i] = termos_independentes
        det_alterado = np.linalg.det(matriz_alterada)
        solucoes.append(det_alterado / det_principal)
    return np.array(solucoes)

matriz = np.array([[2, 1, -1], [-3, -1, 2], [-2, 1, 2]], dtype=float)
termos_independentes = np.array([8, -11, -3], dtype=float)

inicio_gauss = time.time()
sol_gauss = eliminacao_de_gauss(matriz, termos_independentes)
fim_gauss = time.time()

inicio_cramer = time.time()
sol_cramer = regra_de_cramer(matriz, termos_independentes)
fim_cramer = time.time()

print("Solução pelo método de eliminação de Gauss:", sol_gauss)
print("Tempo de execução (Gauss):", fim_gauss - inicio_gauss, "segundos")
print("Solução pela Regra de Cramer:", sol_cramer)
print("Tempo de execução (Cramer):", fim_cramer - inicio_cramer, "segundos")
```

```
Solução pelo método de eliminação de Gauss: [ 2.  3. -1.]
Tempo de execução (Gauss): 0.00030075599670410156 segundos
Solução pela Regra de Cramer: [ 2.  3. -1.]
Tempo de execução (Cramer): 0.0002598762512207031 segundos
```

Com as funções que eu implementei, e com o exemplo que eu usei, a solução pela regra de Cramer se mostrou mmais eficiente.