Lab 1 - BCC406/PCC177

REDES NEURAIS E APRENDIZAGEM EM PROFUNDIDADE

Pacote *NumPy e Tensor do PyTorch*

Prof. Eduardo e Prof. Vander

Objetivos:

- Exercitar n-dimensional arrays.

Data da entrega : 06/10/2025

- 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.
- Ler Capítulo 3 do livro texto.

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)

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

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

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

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

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

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

```
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

- 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`.

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

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

```
# Verificando o shape do array
a.shape
```

```
(3, 4)
```

```
# Verificando a quantidade de dimensões de um array
a.ndim
```

```
2
```

```
# Verificando a quantidade de elemntos no array
a.size
```

```
12
```

```
# ToDo : Criar um array de 3 dimensões, de shape (2,3,4) e mostrar o shape, quantidade de dimensões
a = np.zeros((2, 3, 4))
print(a)
print(a.shape)
print(a.ndim)
print(a.size)
```

```
[[[0. 0. 0. 0.]
  [0. 0. 0. 0.]
  [0. 0. 0. 0.]]

 [[0. 0. 0. 0.]
  [0. 0. 0. 0.]
  [0. 0. 0. 0.]]]
(2, 3, 4)
3
24
```

```
# ToDo : Criar um array de 3 dimensões mas trocando a função zeros por ones e mostrar o shape, quantid
a = np.ones((2, 3, 4))
print(a)
print(a.shape)
print(a.ndim)
print(a.size)
```

```
[[[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, 3, 4)
3
24
```

```
# ToDo : Criar um array de 3 dimensões mas trocando a função zeros por full e mostrar o shape, quantid
a = np.full((2, 3, 4), 5)
print(a)
print(a.shape)
print(a.ndim)
print(a.size)
```

```
[[[5 5 5 5]
  [5 5 5 5]
  [5 5 5 5]]

 [[5 5 5 5]
  [5 5 5 5]
  [5 5 5 5]]]
(2, 3, 4)
3
24
```

```
# ToDo : Criar um array de 3 dimensões mas trocando a função zeros por empty e mostrar o shape, quant:
a = np.empty((2, 3, 4))
print(a)
print(a.shape)
print(a.ndim)
print(a.size)
```

```
[[[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, 3, 4)
3
24
```

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

`zeros:` cria array preenchido com 0.

`ones:` cria array preenchido com 1.

`full:` cria array preenchido com um valor definido.

`empty:` cria array sem inicializar valores (lixo de memória).

O comando *np.arange*

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

```
# Criando um array
np.arange(1, 5)
```

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

```
# Para criar com ponto flutuante
np.arange(1.0, 5.0)
```

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

```
# ToDo : crie um array com arange, variando de 1 a 5, com um passo de 0.5
np.arange(1.0, 5.0, 0.5)
```

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

Os comandos *np.rand* e *np.randn*

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:

```
np.random.rand(3,4)
```

```
array([[0.14142744, 0.77302511, 0.7793557 , 0.63504211],
       [0.59487907, 0.53620587, 0.90926345, 0.4201612 ],
       [0.04636207, 0.17737144, 0.22183328, 0.42542646]])
```

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

```
np.random.randn(3,4)
```

```
array([[-0.17767661,  0.31224186,  0.78679205, -0.15716736],
       [ 0.21357056,  0.35967241,  0.44376962,  0.57378597],
       [-0.64232905,  0.80098536,  0.80802256,  0.35976225]])
```

```
# ToDo : crie um array aleatório com o shape (1, 2, 3, 4)
np.random.randn(1, 2, 3, 4)
```

```
array([[[[ 0.63980285, -2.10538185,  1.27131449,  0.47978287],
         [-0.96666832, -0.91183433, -1.59667957, -0.18973863],
```

```
 [ 0.98716216, -2.24266924, -0.30400973, -0.70843055]],

[[-0.81976387,  0.00279809, -0.35599406, -0.40666468],
 [ 1.29867245,  0.51317321, -0.05362184,  0.47836815],
 [-1.27766356, -0.12697755,  1.43027034,  0.25040025]]]])
```

A biblioteca *MatplotLib*

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

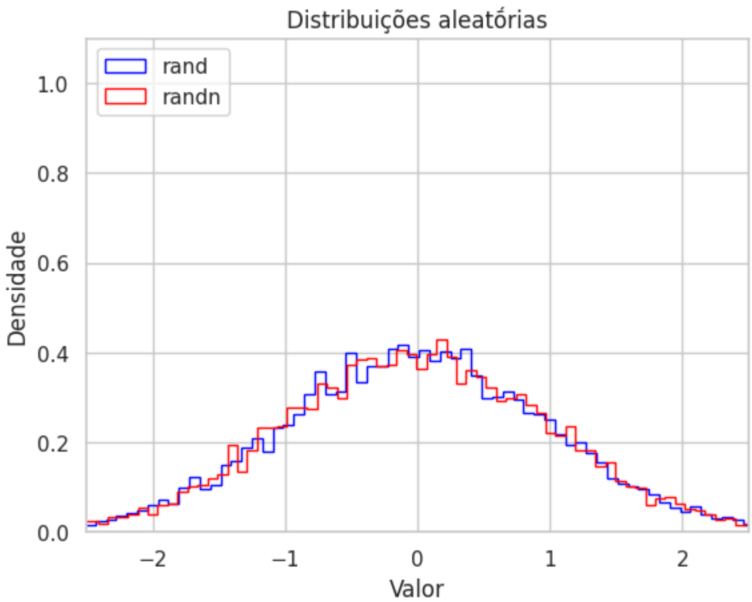
```
%matplotlib inline
import matplotlib.pyplot as plt
```

Primeiro os dados que serão plotados precisam ser criados

```
array_a = np.random.randn(10000)
array_b = np.random.randn(10000)
```

Depois eles podem ser plotados

```
plt.hist(array_a, density=True, bins=100, histtype="step", color="blue", label="rand")
plt.hist(array_b, density=True, bins=100, histtype="step", color="red", label="randn")
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***)

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

```
c = np.arange(1, 5)
print(c.dtype, c)
```

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

```
# ToDo: Crie um um array aleatório de shape (2, 3, 4) e verifique o seu tipo
d = np.random.randn(2, 3, 4)
print(d.dtype, d)
```

```
float64 [[[ 0.93438195  0.39970972  1.16630012  0.47328075]
  [ 1.67489884 -0.25314604 -0.14906415 -0.57788085]
  [-1.7523415   0.79962879  0.48479152 -0.2824134 ]]
```

```
[[-1.26362736 -0.12422629  0.47441827 -0.73472944]
 [-0.78479461  1.0860352   0.42086812  1.22441504]
 [-0.67578086  0.50677444 -0.66903631  0.09884928]]
```

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*

O atributo `itemsize` retorna o tamanho em bytes

```
e = np.arange(1, 5, dtype=np.complex64)
e.itemsize
```

```
8
```

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

```
<memory at 0x7d68c11928e0>
```

```
# ToDo: Crie arrays de shape (2, 2) dos tipos int8, int64, float16, float64, complex64 e complex128
arr_1 = np.ones((2, 2), dtype=np.int8)
arr_2 = np.ones((2, 2), dtype=np.int64)
arr_3 = np.ones((2, 2), dtype=np.float16)
arr_4 = np.ones((2, 2), dtype=np.float64)
arr_5 = np.ones((2, 2), dtype=np.complex64)
arr_6 = np.ones((2, 2), dtype=np.complex128)
```

```
print(arr_1)
print()
print(arr_2)
print()
print(arr_3)
print()
print(arr_4)
print()
print(arr_5)
print()
print(arr_6)
```

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

[[1 1]
 [1 1]]

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

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

[[1.+0.j 1.+0.j]
 [1.+0.j 1.+0.j]]

[[1.+0.j 1.+0.j]
 [1.+0.j 1.+0.j]]
```

> **ToDo**: O que você pode dizer sobre esses arrays criados?

Cada array tem o mesmo shape (2, 2) e valores iguais a 1, mas diferem pelo tipo de dado armazenado:

`int8:` inteiros de 8 bits, variam de –128 a 127.

`int64:` inteiros de 64 bits, usados para números inteiros muito grandes.

`float16:` números decimais em 16 bits, baixa precisão e baixo uso de memória.

`float64:` números decimais em 64 bits, alta precisão (padrão do NumPy).

`complex64:` números complexos em 64 bits (32 bits para parte real e 32 bits para parte imaginária).

`complex128:` números complexos em 128 bits (64 bits para parte real e 64 bits para parte imaginária).

Reshaping

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.

```
# 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
```

```
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
```

```
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***)

```
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
```

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

```
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!

```
g
```

```
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*

```
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* no NumPy permite realizar operações entre arrays de shapes diferentes, ajustando automaticamente suas dimensões para que sejam compatíveis. Isso é feito sem copiar dados na memória, tornando as operações mais eficientes. Fonte: https://numpy.org/doc/stable/user/basics.broadcasting.html.

Regras principais:

1. As dimensões são comparadas da direita para a esquerda.
2. São compatíveis se forem iguais ou uma delas for 1.
3. Arrays com menos dimensões têm dimensões faltantes assumidas como tamanho 1.

Iteração e Concatenação de arrays de *NumPy*

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

```
c = np.arange(24).reshape(2, 3, 4)  # Um array 3D (coposto de duas matrizes de 3x4)
c
```

```
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]]])
```

```
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]]
```

```
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]]
```

```
# 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.

```
# 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
```

```
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.]])
```

```
# ToDo: imprima o shape resultante da concatenação dos arrays de shape a = (2, 3, 4) e b = (2, 3, 4) 
a = np.full((2, 3, 4), 1.0)
b = np.full((2, 3, 4), 2.0)
print(np.concatenate((a, b), axis=0).shape)
print(np.concatenate((a, b), axis=1).shape)
print(np.concatenate((a, b), axis=2).shape)
```

```
(4, 3, 4)
(2, 6, 4)
(2, 3, 8)
```

Transposta

```
m1 = np.arange(10).reshape(2,5)
m1
```

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

```
# ToDo : imprima a matriz transposta de m1
m1.T
```

```
array([[0, 5],
       [1, 6],
```

```
       [2, 7],
       [3, 8],
       [4, 9]])
```

Produto de matrizes

```
n1 = np.arange(10).reshape(2, 5)
n1
```

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

```
n2 = np.arange(15).reshape(5, 3)
n2
```

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

```
n1.dot(n2)
```

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

```
# ToDo: Crie um array de 1 a 25 com shape (5, 5) e faça a multiplicação por uma matriz de zeros de (
a = np.arange(1, 26).reshape(5, 5)
b = np.zeros((5, 1))
c = a @ b # multiplicação matricial

print("a =\n", a)
print("b =\n", b)
print("c =\n", c)
```

```
a =
 [[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]
 [16 17 18 19 20]
 [21 22 23 24 25]]
b =
 [[0.]
 [0.]
 [0.]
 [0.]
 [0.]]
c =
 [[0.]
 [0.]
 [0.]
 [0.]
 [0.]]
```

Matriz Inversa

```
import numpy.linalg as linalg

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

m3
```

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

```
linalg.inv(m3)
```

```
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 `np.linalg.inv` calcula a matriz inversa de uma matriz quadrada A, ou seja, retorna uma matriz B tal que $A \cdot B = I$, onde I é a matriz identidade.

Matriz identidade

```
m3.dot(linalg.inv(m3))
```

```
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]])
```

```
# ToDo: Crie uma matriz identidade de tamanho (5, 5)
np.identity(5)
```

```
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.]])
```

Produto Interno (Dot Product)

O produto interno de dois vetores, é uma operação que resulta em um único número escalar. Geometricamente, ele está relacionado ao comprimento dos vetores e ao ângulo entre eles. Em machine learning, o produto interno é a base para o cálculo de similaridades (como a similaridade de cosseno) e é a operação central nas camadas densamente conectadas das redes neurais.

```
v1, v2 = np.array([1, 2, 3, 4]), np.array([5, 6, 7, 8])

# Exemplo de produto interno usando np.dot()
prod_interno_dot = np.dot(v1, v2)
print("Produto interno (np.dot):", prod_interno_dot)


# ToDo: Verifique se o produto interno entre os dois vetores é o mesmo em ambos os casos.
# O resultado deve ser (1*4) + (2*5) + (3*6) = 4 + 10 + 18 = 32
v1, v2 = np.array([1, 2, 3]), np.array([4, 5, 6])
print("Produto interno (np.dot):", np.dot(v1, v2))
```

```
Produto interno (np.dot): 70
Produto interno (np.dot): 32
```

O produto interno também está relacionado à norma (comprimento) de um vetor:

$$||\mathbf{a}|| = \sqrt{\mathbf{a} \cdot \mathbf{a}}$$

```
# ToDo: Calcule a norma do vetor v1.
# Dica: Use a função np.sqrt() para a raiz quadrada.
# O resultado deve ser sqrt(1^2 + 2^2 + 3^2) = sqrt(14) ~= 3.74

norma_v1 = np.sqrt(np.dot(v1, v1))
print("Norma do vetor v1:", norma_v1)
```

```
Norma do vetor v1: 3.7416573867739413
```

Usando a definição de produto interno, podemos encontrar o ângulo entre os vetores:

$$\cos(\theta) = \frac{\mathbf{a} \cdot \mathbf{b}}{||\mathbf{a}||\,||\mathbf{b}||}$$

```
# ToDo: Calcule o cosseno do ângulo entre os vetores v1 e v2.
# Use o produto interno e as normas calculadas.

prod_interno_v1_v2 = np.dot(v1, v2) # Use np.dot(v1, v2)
norma_v1 = np.sqrt(np.dot(v1, v1))
norma_v2 = np.sqrt(np.dot(v2, v2))

cos_theta = prod_interno_v1_v2 / (norma_v1 * norma_v2)
print("Cosseno do ângulo entre v1 e v2:", cos_theta)
```

```
Cosseno do ângulo entre v1 e v2: 0.9746318461970762
```

> **ToDo**: O que você pode dizer sobre o produto interno entre dois vetores? E qual a relação entre o resultado do produto interno e a similaridade de cosseno entre eles?

O produto interno (ou escalar) entre dois vetores $\mathbf{u}$ e $\mathbf{v}$ é definido como a soma dos produtos de suas componentes:

$$\mathbf{u} \cdot \mathbf{v} = \sum_i u_i v_i$$

Ele mede o quanto os vetores estão alinhados na mesma direção.

A similaridade de cosseno entre dois vetores é o produto interno normalizado pelos comprimentos dos vetores:

$$\cos(\theta) = \frac{\mathbf{u} \cdot \mathbf{v}}{||\mathbf{u}|| \, ||\mathbf{v}||}$$

Ou seja, quanto maior o produto interno relativo aos comprimentos dos vetores, maior a similaridade de direção entre eles.

Fonte: Mathematical Methods for Physicists - Arfken, Harris & Weber - 7th Edition

PyTorch

```
# ToDo: Repitir TODOS OS TESTES ACIMA com o objeto ndarray do Pytorch

import torch

# =========================
#   Criação de tensores
# =========================
zeros = torch.zeros((3, 3))
ones = torch.ones((2, 4))
full = torch.full((2, 2), 7)
empty = torch.empty((2, 3))

print("Zeros:\n", zeros)
print("Ones:\n", ones)
print("Full:\n", full)
print("Empty (conteúdo aleatório):\n", empty)


print()

# =========================
#      Arange / Range
# =========================
arange1 = torch.arange(0, 5, step=1)
arange2 = torch.arange(1, 5, step=0.5)

print("Arange1:", arange1)
print("Arange2:", arange2)

print()

# =========================
#    Números aleatórios
# =========================
rand_uniforme = torch.rand((2, 3))      # uniforme [0,1)
rand_normal = torch.randn((2, 3))       # normal padrão

print("Uniforme:\n", rand_uniforme)
print("Normal:\n", rand_normal)

print()

# =========================
#   Propriedades do tensor
# =========================
tensor = torch.rand((3, 4))
print("Shape:", tensor.shape)
print("Número de dimensões (ndim):", tensor.ndim)
print("Número de elementos (numel):", tensor.numel())
print("Tipo (dtype):", tensor.dtype)
print("Bytes por elemento (element_size):", tensor.element_size())

print()
```

```
# ========================
# Tipos de dados diferentes
# ========================
t_int = torch.tensor([1, 2, 3], dtype=torch.int64)
t_float = torch.tensor([1.0, 2.0, 3.0], dtype=torch.float32)
t_complex = torch.tensor([1+2j, 3+4j], dtype=torch.complex64)

print("Int64:", t_int)
print("Float32:", t_float)
print("Complex64:", t_complex)

print()

# ========================
#   Reshape e transposição
# ========================
a = torch.arange(6)
a_reshaped = a.reshape((2, 3))
a_transposed = a_reshaped.T

print("Original:", a)
print("Reshape 2x3:\n", a_reshaped)
print("Transposta:\n", a_transposed)

print()

# ========================
#   Operações aritméticas
# ========================
a = torch.tensor([1, 2, 3])
b = torch.tensor([4, 5, 6])

print("Soma:", a + b)
print("Subtração:", a - b)
print("Multiplicação:", a * b)
print("Divisão:", a / b)
print("Divisão inteira:", a // b)
print("Resto:", a % b)
print("Exponenciação:", a ** 2)

print()

# ========================
#       Broadcasting
# ========================
x = torch.ones((2, 3))
y = torch.tensor([1, 2, 3])
print("Broadcast (x + y):\n", x + y)

print()

# ========================
#  Iteração e concatenação
# ========================
for row in x:
    print("Iteração linha:", row)

concat = torch.cat((x, x), dim=0)
print("Concatenação na dimensão 0:\n", concat)

print()

# ===============================
# Produto de matrizes e identidade
# ===============================
A = torch.rand((2, 3))
B = torch.rand((3, 2))
C = torch.matmul(A, B)   # ou A @ B

I = torch.eye(3)         # matriz identidade 3x3

print("Produto de matrizes A @ B:\n", C)
print("Matriz identidade 3x3:\n", I)
```

```
print()

# ========================
#     Matriz inversa
# ========================
M = torch.rand((3, 3))
inv_M = torch.linalg.inv(M)
print("Matriz M:\n", M)
print("Inversa de M:\n", inv_M)

print()

# ==============================
#  Produto interno (dot product)
# ==============================
v1, v2 = torch.tensor([1, 2, 3]), torch.tensor([4, 5, 6])
prod_interno_dot = torch.dot(v1, v2)
print("Produto interno (torch.dot):", prod_interno_dot.item())

print()

# ========================
#      Norma do vetor
# ========================
norma_v1 = torch.sqrt(torch.dot(v1, v1).float())
print("Norma do vetor v1:", norma_v1.item())

print()

# ====================================
# Cosseno do ângulo entre dois vetores
# ====================================
prod_interno_v1_v2 = torch.dot(v1, v2).float()
norma_v2 = torch.sqrt(torch.dot(v2, v2).float())
cos_theta = prod_interno_v1_v2 / (norma_v1 * norma_v2)
print("Cosseno do ângulo entre v1 e v2:", cos_theta.item())
```

```
 tensor([[ 1.0984,  0.0939,  1.2362],
        [-0.9964,  0.1367,  1.8188]])

Shape: torch.Size([3, 4])
Número de dimensões (ndim): 2
Número de elementos (numel): 12
Tipo (dtype): torch.float32
Bytes por elemento (element_size): 4

Int64: tensor([1, 2, 3])
Float32: tensor([1., 2., 3.])
Complex64: tensor([1.+2.j, 3.+4.j])

Original: tensor([0, 1, 2, 3, 4, 5])
Reshape 2x3:
 tensor([[0, 1, 2],
        [3, 4, 5]])
Transposta:
 tensor([[0, 3],
        [1, 4],
        [2, 5]])

Soma: tensor([5, 7, 9])
```

```
        [0.4787, 0.6303]])
```

```
Matriz identidade 3x3:
 tensor([[1., 0., 0.],
        [0., 1., 0.],
        [0., 0., 1.]])

Matriz M:
 tensor([[0.5166, 0.3791, 0.9449],
        [0.8597, 0.7151, 0.7638],
        [0.3361, 0.2550, 0.2228]])
Inversa de M:
 tensor([[  2.6150, -11.5407,  28.4773],
        [ -4.8076,  14.9344, -30.8124],
        [  1.5576,   0.3176,  -3.2067]])
```

Regressão linear com pytorch

Utilize o **dataset Diabetes** do `scikit-learn` para resolver um problema de **regressão linear** em **PyTorch**.

1. Carregue e normalize os dados.
2. Resolva o problema de duas formas:
   - **(a)** Usando a solução analítica (equação normal ou pseudo-inversa) apenas com tensores.
   - **(b)** Treinando um modelo `nn.Linear` com gradiente descendente.
3. Compare os resultados obtidos (MSE e R^2) e discuta a diferença entre os pesos encontrados nos dois métodos.

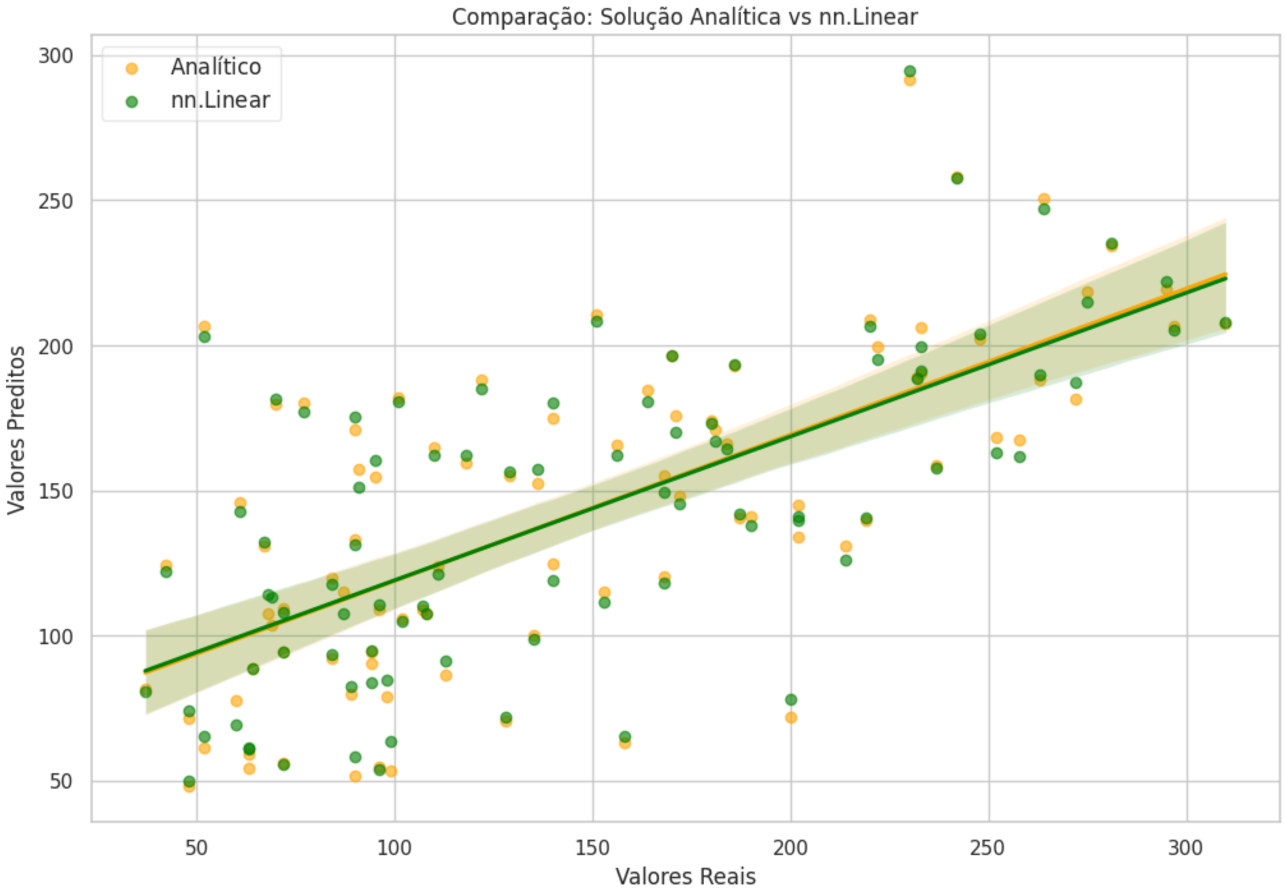
Sugestão de leitura: [Capítulo 3 do livro texto](#).

```
# %% ------------------------------------------------------------
# Regressão no Diabetes (scikit-learn) com PyTorch
# A) Solução analítica (pseudo-inversa)
# B) Treinamento com gradiente (nn.Linear)
# -------------------------------------------------------------
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.datasets import load_diabetes
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, r2_score
import matplotlib.pyplot as plt
import seaborn as sns

# Dispositivo (usa GPU se disponível)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# 1) Carregar dados do dataset Diabetes
X_np, y_np = load_diabetes(return_X_y=True)  # X: (442,10), y: (442,)

# 2) Dividir dados em treino e teste
X_tr, X_te, y_tr, y_te = train_test_split(X_np, y_np, test_size=0.2, random_state=42)

# 3) Normalização dos dados (média 0 e desvio padrão 1)
scaler = StandardScaler()
X_tr = scaler.fit_transform(X_tr)
X_te = scaler.transform(X_te)

# 4) Conversão para tensores do PyTorch (e envio para GPU se disponível)
X_tr_t = torch.tensor(X_tr, dtype=torch.float32, device=device)
X_te_t = torch.tensor(X_te, dtype=torch.float32, device=device)
y_tr_t = torch.tensor(y_tr, dtype=torch.float32, device=device).view(-1, 1)
y_te_t = torch.tensor(y_te, dtype=torch.float32, device=device).view(-1, 1)

# =======================================================
# A) Solução analítica (pseudo-inversa)
# =======================================================
# Adiciona coluna de 1s para o bias (intercepto)
X_tr_bias = torch.cat([torch.ones(X_tr_t.shape[0], 1, device=device), X_tr_t], dim=1)
X_te_bias = torch.cat([torch.ones(X_te_t.shape[0], 1, device=device), X_te_t], dim=1)

# Calcula os pesos usando pseudo-inversa (equação normal)
w_analytical = torch.linalg.pinv(X_tr_bias) @ y_tr_t

# Faz predição nos dados de teste
y_pred_analytical = X_te_bias @ w_analytical
```

```
# Calcula métricas de erro e ajuste
mse_a = mean_squared_error(y_te, y_pred_analytical.cpu().detach().numpy())
r2_a = r2_score(y_te, y_pred_analytical.cpu().detach().numpy())

# ========================================================
# B) Regressão Linear com nn.Linear + SGD
# ========================================================
# Cria modelo linear
model = nn.Linear(X_tr_t.shape[1], 1).to(device)

# Define função de perda e otimizador
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

# Treinamento do modelo
epochs = 1000
for epoch in range(epochs):
    # Predição nos dados de treino
    y_pred = model(X_tr_t)
    # Calcula a perda
    loss = criterion(y_pred, y_tr_t)

    # Backpropagation
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

# Predição nos dados de teste após treinamento
y_pred_model = model(X_te_t).detach().cpu().numpy()

# Calcula métricas de erro e ajuste
mse_b = mean_squared_error(y_te, y_pred_model)
r2_b = r2_score(y_te, y_pred_model)

# ========================================================
# Comparação dos resultados
# ========================================================
print("=== Solução Analítica ===")
print(f"MSE: {mse_a:.4f}")
print(f"R² : {r2_a:.4f}")

print("\n=== Regressão Treinada ===")
print(f"MSE: {mse_b:.4f}")
print(f"R² : {r2_b:.4f}")

print("\n=== Comparação dos pesos ===")
# Mostra os primeiros 5 pesos e bias para comparação
print("Pesos analíticos (primeiros 5):", w_analytical.view(-1)[:5].cpu().numpy())
print("Pesos treinados  (primeiros 5):", list(model.parameters())[0].detach().cpu().numpy()[0][:5])
print("Bias analítico:", w_analytical[0].item())
print("Bias treinado :", list(model.parameters())[1].item())

# ========================================================
# Plot comparativo usando seaborn regplot com parâmetros em LaTeX
# ========================================================
# ========================================================
# Plot comparativo usando seaborn regplot com parâmetros em LaTeX
# ========================================================
sns.set(style="whitegrid")
plt.figure(figsize=(12, 8))

# Pega os pesos e bias do modelo nn.Linear
w_nn = list(model.parameters())[0].detach().cpu().numpy().flatten()
b_nn = list(model.parameters())[1].item()
w_anal = w_analytical.view(-1).cpu().numpy()

# Cria strings LaTeX para os labels apenas com método (sem listar pesos)
label_analytical = r"$\text{Analítico}$"
label_nn = r"$\text{nn.Linear}$"

# Ponto real vs predito (analítico)
sns.regplot(
    x=y_te,
    y=y_pred_analytical.cpu().detach().numpy(),
```

```
        color='orange',
        label=label_analytical,
        scatter_kws={'alpha':0.6}
    )

    # Ponto real vs predito (modelo nn.Linear)
    sns.regplot(
        x=y_te,
        y=y_pred_model,
        color='green',
        label=label_nn,
        scatter_kws={'alpha':0.6}
    )

    # Configuração dos eixos e título
    plt.xlabel("Valores Reais")
    plt.ylabel("Valores Preditos")
    plt.title("Comparação: Solução Analítica vs nn.Linear")

    # Legenda com fonte menor e caixa semi-transparente
    plt.legend(fontsize=12, framealpha=0.4, loc='upper left')
    plt.show()
```

```
=== Solução Analítica ===
MSE: 2900.1936
R² : 0.4526

=== Regressão Treinada ===
MSE: 2885.7159
R² : 0.4553

=== Comparação dos pesos ===
Pesos analíticos (primeiros 5): [153.73654     1.7537557 -11.511811   25.607115   16.828873 ]
Pesos treinados  (primeiros 5): [  1.9378772 -11.433249   26.268246   16.607763   -9.853201 ]
Bias analítico: 153.73654174804688
Bias treinado : 153.73617553710938
```



No experimento de regressão linear utilizando o dataset Diabetes do scikit-learn, os dados foram carregados, normalizados e tratados como tensores do PyTorch. O problema foi resolvido de duas formas: pela solução analítica, com cálculo direto dos pesos via pseudo-inversa da equação normal, e pelo treinamento de um modelo nn.Linear otimizado com gradiente descendente estocástico. Os resultados foram bastante próximos: a solução analítica apresentou MSE de 2900.1936 e R^2 de 0.4526, enquanto o modelo treinado obteve MSE de 2885.8741 e R^2 de 0.4553. A comparação dos pesos mostra diferenças pequenas nos coeficientes, mas ambos os métodos convergem para valores semelhantes, com bias praticamente idêntico. Isso evidencia que a otimização iterativa consegue recuperar resultados equivalentes à solução exata, ainda que sujeita a pequenas variações numéricas.