Lab 0 - PCC177/BCC406

REDES NEURAIS E APRENDIZAGEM EM PROFUNDIDADE

Profs. Eduardo e Pedro

Aluna: Daniela Costa Terra

Data da entrega : 23/03

- Complete o código (marcado com ToDo) e escreva os textos diretamente nos notebooks.
- Execute todo notebook e salve tudo em um PDF nomeado como "NomeSobrenome-Lab0.pdf"
- Envie o PDF para via Google FORM.

Configure seu ambiente, seguindo os passos da seção de instalação do livro ou instale a distribuição Anaconda.

Se preferir usar Google CoLab, lembre-se de atualizar o drive do compilador CUDA da NVIDIA para que ele funcione com o MXNet. Veja exemplo no link

**Antes de realizar o notebook, leia a seção 2.1 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

```
# 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.]])
```

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.

```
a = np.zeros((3,4))
a
```

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

```
a.shape
```

```
(3, 4)
```

```
a.ndim
```

```
2
```

```
a.size
```

```
12
```

```
# ToDo : Criar um array de 3 dimensões, de shape (2,3,4) e repetir as operações aci
```

```
m3D = np.zeros((2,3,4))
m3D
```

```
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., 0.]]])
```

```
m3D.shape
```

```
(2, 3, 4)
```

```
m3D.ndim
```

```
3
```

```
m3D.size
```

```
24
```

```
# ToDo : repita as operações acima trocando a função zeros por : ones, full, empty
```

```
m3D2 = np.ones((2,3,4))
print(f'-m3D2 = np.ones((2,3,4)):\n{m3D2} \n-m3D2.shape:{m3D2.shape} \n-m3D2.ndim:{

m3D3 = np.full((2,3,4), [np.nan, np.inf, 0, 0])
print(f'\n****\n-m3D3 = np.full((2,3,4)):\n{m3D3} \n-m3D3.shape:{m3D3.shape} \n-m3D

m3D4 = np.empty((2,3,4))
print(f'\n****\n-m3D4 = np.empty((2,3,4)):\n{m3D4} \n-m3D4.shape:{m3D4.shape} \n-m3
```

```
-m3D2 = np.ones((2,3,4)):
[[[1. 1. 1. 1.]
  [1. 1. 1. 1.]
  [1. 1. 1. 1.]]

 [[1. 1. 1. 1.]
  [1. 1. 1. 1.]
  [1. 1. 1. 1.]]]
-m3D2.shape:(2, 3, 4)
-m3D2.ndim:3
-m3D2.size:24

****
-m3D3 = np.full((2,3,4)):
[[[nan inf  0.  0.]
  [nan inf  0.  0.]
  [nan inf  0.  0.]]

 [[nan inf  0.  0.]
  [nan inf  0.  0.]
```

```
  [nan inf  0.  0.]]]
-m3D3.shape:(2, 3, 4)
-m3D3.ndim:3
-m3D3.size:24

****
-m3D4 = np.empty((2,3,4)):
[[[1. 1. 1. 1.]
  [1. 1. 1. 1.]
  [1. 1. 1. 1.]]

 [[1. 1. 1. 1.]
  [1. 1. 1. 1.]
  [1. 1. 1. 1.]]]
-m3D4.shape:(2, 3, 4)
-m3D4.ndim:3
-m3D4.size:24
```

np.arange

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

```
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, 5, 0.5)
```

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

`np.rand` and `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([[0.97854292, 0.16669455, 0.86478053, 0.77477359],
       [0.76646924, 0.04017001, 0.39126673, 0.92423248],
       [0.2181957 , 0.47362675, 0.2509002 , 0.44496316]])
```

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

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

```
array([[ 0.71709608, -0.280766  , -0.34485566, -0.16323784],
       [-0.31538333,  0.02058291,  1.56656185, -0.42337715],
       [ 0.28520354, -0.61364492,  0.91582168,  0.61552854]])
```

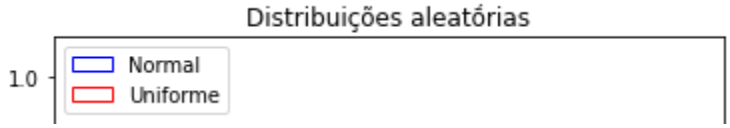
ToDo : Vamos usar a biblioteca matplotlib (para mais detalhes veja matplotlib tutorial) para plotar dois arrays de tamanho 10000, um inicializado com distribuição normal e o outro com uniforme

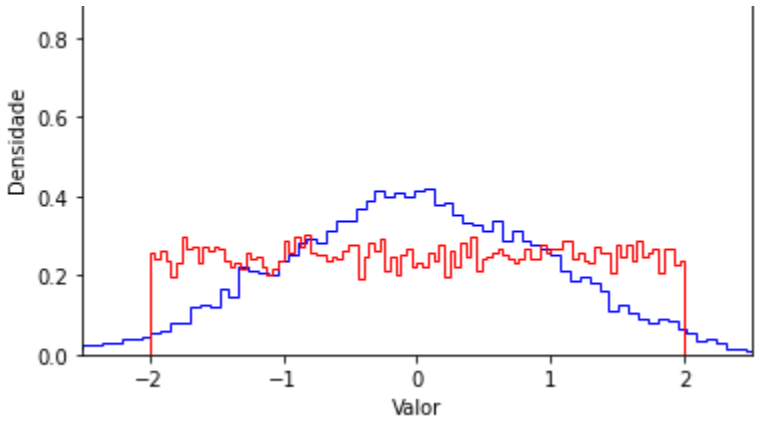
```
%matplotlib inline
import matplotlib.pyplot as plt
```

```
from numpy import random
array_a = random.randn(10000) # ToDo : complete  'Distribuição Normal'
print(array_a)
array_b = random.uniform(-2, 2, 10000)  # ToDo : complete 'Conforme enunciado acima
print(array_b)
```

```
[ 0.72519323 -0.15057161  0.62925829 ...  0.26781766 -1.45761175
 -0.1795345 ]
[-0.53083167 -0.55157561 -0.24227158 ... -1.69490057  1.33285798
  1.95219423]
```

```
plt.hist(array_a, density=True, bins=100, histtype="step", color="blue", label="Nor
plt.hist(array_b, density=True, bins=100, histtype="step", color="red", label="Unif
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]
```

```
c = np.arange(1.0, 5.0)
print(c.dtype, c)
```

```
float64 [1. 2. 3. 4.]
```

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

`itemsize`

O atributo `itemsize` retorna o tamanho em bytes

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

```
8
```

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

```
<memory at 0x7fbf732a0590>
```

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
```

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]])
```

```
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]]])
```

```
#repare que o objeto 'g' foi modificado também!
```

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 aqui o conceito de broadcasting

O Broadcasting permite que operações com arrays (tensores) sejam feitas mesmo que eles tenham formatos (ou shapes) distindos. As operações serão realizada após a expansão apropriada de um ou de ambos os arrays, replicando seus elementos. Por exemplo, supondo A um array (3 x 1) e B um array (1 x 2). Sendo A = [[1],[3],[5]] e B =[0, 9], a coluna 1 da matriz 'A' será duplicada gerando [[1, 1],[3, 3],[5, 5]] de shape (3, 2). A linha 1 de B também será replicada 3 vezes, produzindo [[0, 9], [0, 9], [0, 9]], e assim igualando o formato (3x2). Depois desse ajuste a operação (elementwize) será realizada: A + B = array([[1, 10],[3, 12], [5, 14]]).

Iterando : repare que você pode iterar pelos ndarrays. Repare que a iteração é feita pelos axis.

```
c = np.arange(24).reshape(2, 3, 4)  # A 3D array (composed of two 3x4 matrices)
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)):  # Note that 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 iteirar 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
```

Concatenando arrays

```
# pode-se concatenar arrays pelos axis
q1 = np.full((3,4), 1.0)

q2 = np.full((3,4), 2.0)

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

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

q = np.concatenate((q1, q2, q3), axis=1)
print(q)
```

```
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [2. 2. 2. 2.]
 [2. 2. 2. 2.]
 [2. 2. 2. 2.]
 [3. 3. 3. 3.]
 [3. 3. 3. 3.]
 [3. 3. 3. 3.]]
[[1. 1. 1. 1. 2. 2. 2. 2. 3. 3. 3. 3.]
 [1. 1. 1. 1. 2. 2. 2. 2. 3. 3. 3. 3.]
 [1. 1. 1. 1. 2. 2. 2. 2. 3. 3. 3. 3.]]
```

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
m1t=np.transpose(m1)
print(m1t)
```

```
[[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]])
```

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]])
```

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]])
```

Comparando os objetos de array

Os objetos do tipo array das bibliotecas de deep learning (tensorflow, pytorch, mxnet) são muito

parecedios com o do NumPy. Porém, otimizados. Crie um objeto do tipo NDArray do MXNet e compare a performance contra o objeto do NumPy.

```
!pip install -U mxnet-cu101==1.7.0
from mxnet import nd, gpu, gluon, autograd
import mxnet as mx
from mxnet.gluon import nn
import time
```

```
Collecting mxnet-cu101==1.7.0
  Downloading mxnet_cu101-1.7.0-py2.py3-none-manylinux2014_x86_64.whl (846.0 M
     |████████████████████████████████| 834.1 MB 1.2 MB/s eta 0:00:10tcmalloc:
     |████████████████████████████████| 846.0 MB 22 kB/s
Collecting graphviz<0.9.0,>=0.8.1
  Downloading graphviz-0.8.4-py2.py3-none-any.whl (16 kB)
Requirement already satisfied: requests<3,>=2.20.0 in /usr/local/lib/python3.7
Requirement already satisfied: numpy<2.0.0,>1.16.0 in /usr/local/lib/python3.7
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /usr
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.7/
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.7/d
Installing collected packages: graphviz, mxnet-cu101
  Attempting uninstall: graphviz
    Found existing installation: graphviz 0.10.1
    Uninstalling graphviz-0.10.1:
      Successfully uninstalled graphviz-0.10.1
Successfully installed graphviz-0.8.4 mxnet-cu101-1.7.0
```

```
#criando um array com ndarray do mxnet
nd.array(( (1,2,3), (4,5,6) ))
```

```
[[1. 2. 3.]
 [4. 5. 6.]]
<NDArray 2x3 @cpu(0)>
```

```
# crianod-se uma matriz
x = nd.ones(shape = (2,3))
x
```

```
[[1. 1. 1.]
 [1. 1. 1.]]
<NDArray 2x3 @cpu(0)>
```

O objeto NDArray do MXNet também possui as funções de criação de números aleatórios e de algebra, feito as do NumPy.

```
#criando uma matrix uniforme aleatoria, com valores entre -1 e 1
y = nd.random.uniform(low=-1, high=1, shape = (2,3))
y
```

```
[[-0.8257414   0.2963438  -0.9595632 ]
 [-0.2635169   0.6652397   0.91431034]]
<NDArray 2x3 @cpu(0)>
```

Diferentemente do numpy, NDArray me permite colocar os dados em alguma CPU específica ou em alguma GPU : repare no contexto!

```
# shape e tamanho da matriz
# os outros parâmetros são o tipo de dados e contexto

(x.shape, x.size, x.dtype, x.context)
```

```
((2, 3), 6, numpy.float32, cpu(0))
```

```
# pode-se definir o tipo de dados em tempo de criação
nd.ones(shape = (2,3), dtype = np.uint8)
```

```
[[1 1 1]
 [1 1 1]]
<NDArray 2x3 @cpu(0)>
```

```
# ou pode-se alterar dinamicamente
y.astype(np.float16)
```

```
[[-0.8257  0.2964 -0.9595]
 [-0.2634  0.665   0.9146]]
<NDArray 2x3 @cpu(0)>
```

Alocando na CPU

```
#NDArray permite alocar me CPU e GPU
nd.ones(shape = (2,3), ctx=mx.cpu())
```

```
[[1. 1. 1.]
 [1. 1. 1.]]
<NDArray 2x3 @cpu(0)>
```

Alocando na GPU

```
nd.ones(shape = (2,3), ctx=mx.gpu())
```

```
[[1. 1. 1.]
 [1. 1. 1.]]
<NDArray 2x3 @gpu(0)>
```

ToDo:

Crie 6 matrizes, com a função ones:

- Uma de shape (10000, 5000) e outra com shape (5000, 10000), usando-se o objeto do NumPy.
- Uma de shape (10000, 5000) e outra com shape (5000, 10000), usando-se o objeto do NDArray do MXNets, porém alocada em CPU.
- Uma de shape (10000, 5000) e outra com shape (5000, 10000), usando-se o objeto do NDArray do MXNets, porém alocada em GPU.

```
#x_np, y_np = np.ones(shape = (10000,5000)) # ToDo : complete
x_np = np.ones(shape = (10000,5000)) # ToDo : complete
y_np = np.ones(shape = (5000,10000)) # ToDo : complete

#x_nd_cpu , y_nd_cpu = nd.ones(shape = (10000,5000), ctx= mx.cpu()) # ToDo : comple
x_nd_cpu = nd.ones(shape = (10000,5000), ctx= mx.cpu()) # ToDo : complete
y_nd_cpu = nd.ones(shape = (5000, 10000), ctx= mx.cpu()) # ToDo : complete

#x_nd_gpu , y_nd_gpu = nd.ones(shape = (10000,5000), ctx= mx.gpu())# ToDo : complet
x_nd_gpu = nd.ones(shape = (10000,5000), ctx= mx.gpu())# ToDo : complete
y_nd_gpu = nd.ones(shape = (5000, 10000), ctx= mx.gpu())# ToDo : complete
```

Execute as multiplicações e verifique o tempo de execução

```
tic = time.time()
np.dot(x_np, y_np)
print("NumPy time : {:.4f}s".format(time.time()-tic))
```

```
NumPy time : 29.7072s
```

```
tic = time.time()
nd.dot(x_nd_cpu, y_nd_cpu)
print("MXNet CPU time : {:.4f}s".format(time.time()-tic))
```

```
MXNet CPU time : 0.0068s
```

```
tic = time.time()
nd.dot(x_nd_gpu, y_nd_gpu)
print("MXNet GPU time : {:.4f}s".format(time.time()-tic))
```

```
MXNet GPU time : 0.0166s
```