Introdução ao NumPy

Prof. Dr. Anselmo R. Pitombeira Neto

OPL - Pesquisa Operacional em Produção e Logística, Departamento de Engenharia de Produção, Universidade Federal do Ceará

ver 0.2

O Numerical Python (NumPy) é uma biblioteca para a linguagem Python com funções para se trabalhar com computação numérica. Seu principal objeto é o vetor n-dimensional, ou `ndarray`. Um vetor n-dimensional também é conhecido pelo nome *tensor*. A principal característica do `ndarray` é que ele deve ser homogêneo, ou seja, diferentemente do objeto lista, todos os seus elementos devem ser do mesmo tipo.

Criação de tensores

Pode-se criar um `ndarray` passando um container como inicializador para um objeto da classe `array`:

```
import numpy as np
v = np.array([1,2,3,4])
print(v)
```

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

O tipo dos elementos de um `ndarray` pode ser acessado por meio do atributo `dtype`

```
print(v.dtype)
```

```
int64
```

Neste caso, ao criar o vetor, o tipo *default* assumido foi `int64`, mas o tipo desejado pode ser especificado na criação do tensor:

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

```
float64
```

No caso, se utilizarmos números com pontos decimais, o tipo será automaticamente `float`:

```
v = np.array([1.0,2.0,3.0,4.0])
print(v.dtype)
```

```
float64
```

Os tensores NumPy também possuem um atributo chamado `shape`. Esse atributo indica a forma do tensor, por exemplo:

```
print(v.shape)
```

```
(4,)
```

Neste caso, o tensor v possui 1 dimensão (ou eixo) com 4 elementos. Um tensor unidimensional corresponde a um vetor. Podemos também criar um tensor bidimensional (uma matriz) usando o atributo `shape`:

```
v = np.array([1,2,3,4])
v.shape = (2,2)
print(v)
```

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

Outra forma útil de mudar o shape de um tensor é simplesmente utilizando a função reshape:

```
v = np.array([1,2,3,4]).reshape(2,2)
print(v)
```

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

A função `reshape` aceita o uso de -1 como dimensão na especificação do shape. Neste caso, a função reshape determinará automaticamente o número de elementos em cada dimensão de forma a manter inalterado o número total de elementos no tensor. No exemplo abaixo, especificamos o tamanho da primeira dimensão como igual a 2. A função reshape então determinará automaticamente qual o tamanho da segunda dimensão:

```
v = np.array([1,2,3,4,5,6,7,8])
v = v.reshape(2, -1)
print(v)
```

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

O número de eixos (ou dimensões) de um tensor é dado pelo atributo `ndim`, enquanto o número total de elementos é dado por `size`:

```
v = np.array(range(50)).reshape(2,5,5)

print('Shape = ', v.shape)
print('Número de dimensões = ', v.ndim)
print('Número de elementos = ', v.size)
print('Tensor v = \n', v)
```

```
Shape =  (2, 5, 5)
Número de dimensões =  3
Número de elementos =  50
Tensor v =
 [[[ 0  1  2  3  4]
  [ 5  6  7  8  9]
  [10 11 12 13 14]
```

As funções `zeros`, `ones` e `diag` são muito convenientes para a criação de tensores:

```
V = np.zeros((3,3))
print('V = \n', V)
U = np.ones((3,3))
print('U = \n', U)
D = np.diag([10, 10, 10])
print('D = \n', D)
```

```
V =
 [[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
U =
 [[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
D =
 [[10  0  0]
 [ 0 10  0]
 [ 0  0 10]]
```

As funções `arange` e `linspace` permitem a criação de sequências como tensores NumPy:

```
v = np.arange(0, 5, 0.5)
u = np.linspace(0, 5, 10)
print("v =", v)
print("u =", u)
```

```
v = [0.  0.5 1.  1.5 2.  2.5 3.  3.5 4.  4.5]
u = [0.         0.55555556 1.11111111 1.66666667 2.22222222 2.77777778
 3.33333333 3.88888889 4.44444444 5.        ]
```

O método `flatten` retorna uma cópia do tensor com todos os elementos em apenas uma dimensão:

```
v = np.array(range(50)).reshape(2,5,5)
print(v.flatten())
```

```
[ 0  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 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 48 49]
```

O atrubuto `flat` retorna um iterador nos elementos de um tensor:

```
v_iter = v.flat
for i in v_iter:
  print(i, end=' ')
```

```
0 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 26 27 28 29 30
```

Indexação de tensores

O acesso aos elementos de um tensor pode ser feito utilizando colchetes e o índice do elemento:

```
print(v[1])
```

```
[[25 26 27 28 29]
 [30 31 32 33 34]
 [35 36 37 38 39]
 [40 41 42 43 44]
 [45 46 47 48 49]]
```

Tensores com rank maior ou igual a 2, por exemplo, acessar o elemento com índice 0 no primeiro eixo e índice 1 no segundo eixo:

```
v = np.array([10, 20, 30, 40]).reshape(2,2)
print(v[0,1])
```

```
20
```

Tensor de rank 3:

```
v = np.arange(8).reshape(2,2,2)
print("v =\n", v)
print("\n v[0,0,1] =", v[0,0,1])
```

```
v =
 [[[0 1]
  [2 3]]

 [[4 5]
  [6 7]]]

 v[0,0,1] = 1
```

Subtensores contíguos de um tensor podem ser acessados utilizando o operador " : ". Note que neste caso o NumPy não faz uma cópia dos dados, mas apenas cria uma *view* dos dados:

```
v = np.arange(10)
u = v[1:3]
print(u)
```

```
[1 2]
```

No caso de uma matriz (um tensor de rank 2), pode-se acessar submatrizes:

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

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

Vamos selecionar somente a submatriz correspondente às linhas 0 e 1, e colunas 2 e 3:

```
subA = A[0:2, 2:4]
print(subA)
```

```
[[2 3]
 [7 8]]
```

A indexação pode ser realiza também passando uma lista ou outro tensor que contém os índices dos elementos que queremos selecionar, por exemplo, abaixo queremos selecionar somente os elementos com índice 2 e 3. Observe que, neste caso, o tensor v resultante é um novo objeto criado a partir do tensor u e alocado na memória. Ocorre cópia dos dados.

```
u = np.array([2.0, 3.5, 4.0, -10.1])
v = u[[2,3]]
print(v)
```

```
[  4.  -10.1]
```

Filtrando as colunas 2 e 4 de uma matriz:

```
A = np.arange(10).reshape(2,5)
B = A[:, [2,4]]
print("A =\n", A)
print("B =\n", B)
```

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

Operações sobre tensores

As operações sobre tensores NumPy são realizada elemento a elemento, por exemplo:

```
v = np.array([10,20,30])
u = np.array([2,2,2])
w = u+v
print(w)
```

```
[12 22 32]
```

Note que no caso dos operadores * e / a operação também ocorre elemento a elemento:

```
w = u*v
```

```
print("w =", w)
x = u/v
print("x =", x)
```

```
w = [20 40 60]
x = [0.2        0.1        0.06666667]
```

Elevar todos os elementos de um vetor a uma potência:

```
x = np.array([10, 20])
y = x**2
print(y)
```

```
[100 400]
```

Alguns operadores unários como a média, menor valor, maior valor, etc possuem funções específicas:

```
x = np.arange(10)
media = x.mean()
menor_valor = np.min(x)
arg_max = np.argmax(x)
print("Média =", media)
print("Menor valor =", menor_valor)
print("Arg max =", arg_max)
```

```
Média = 4.5
Menor valor = 0
Arg max = 9
```

Em tensores com rank maior ou igual a 2 pode-se identifcar o eixo em que se deseja aplicar as funções min, max, etc. Caso o eixo não seja especificado, a funções retornará o menor/maior valor entre todos incluindo todos os eixos:

```
A = np.array([10, 30, 40, 20]).reshape(2,2)
menor= A.min()
menor_colunas = A.min(axis=0)
print("A = \n", A)
print("Menor valor = ", menor)
print("Menor valor em cada coluna =", menor_colunas)
```

```
A = 
 [[10 30]
 [40 20]]
Menor valor =  10
Menor valor em cada coluna = [10 20]
```

A operação de produto interno (ou produto escalar) pode ser realizada pela função `dot`:

```
w = np.dot(u, v)
print("w =", w)

x = u.dot(v)
print("x =", x)
```

```
w = 120
x = 120
```

Multiplicação de um escalar por um vetor:

```
print("w =", w)
x = 10*w
print("x =", x)
```

```
w = 120
x = 1200
```

Um erro muito comum no uso do NumPy é não explorar sua eficiência computacional por meio das operações vetoriais. Dentro do possível, deve-se sempre organizar implementação de um algoritmo utilizando somente operações sobre tensores. Por exemplo, abaixo é feita a comparação entre o tempo computacional em usar a operação `dot` e uma implementação da mesma operação utilizando loops:

```
def produto_interno(u, v):

  prod = 0

  for i in range(u.size):
    prod+=u[i]*v[i]

  return prod

u = np.random.rand(10000)
v = np.random.rand(10000)

%timeit produto_interno(u,v)
%timeit np.dot(u,v)
```

```
100 loops, best of 3: 3.31 ms per loop
The slowest run took 15.63 times longer than the fastest. This could mean that an i
100000 loops, best of 3: 3.52 µs per loop
```

Observe que a operação `dot` é muito mais rápida. Isso ocorre porque a operação `dot` é implementada em linguagem C e realiza a operação de maneira vetorizada.

Matrizes

Em NumPy, matrizes são tensores com 2 dimensões (tensores de rank 2). Embora exista um objeto `matrix`, em geral é mais prático tratar matrizes como tensores de rank 2:

```
A = np.array([[10, 20], [30, 40]])
print(A)
```

```
[[10 20]
 [30 40]]
```

Criar uma matriz identidade:

```
I = np.eye(5)
print(I)
```

```
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
```

Matrizes diagonais:

```
D = np.diag(np.arange(5))
print(D)
```

```
[[0 0 0 0 0]
 [0 1 0 0 0]
 [0 0 2 0 0]
 [0 0 0 3 0]
 [0 0 0 0 4]]
```

Operações sobre matrizes

Multilplicação de vetores e matrizes é realizada por meio da função `dot`:

```
v = np.array([10, 10])
A = np.arange(4).reshape(2,2)
u = A.dot(v)
print(u)
```

```
[10 50]
```

Multiplicação de matrizes:

```
A = np.ones((2,2))
B = 10*np.ones((2,2))
C= np.dot(A,B)
print(C)
```

```
[[20. 20.]
 [20. 20.]]
```

A multiplicação de matrizes também pode ser realizada por meio do operador @:

```
C = A @ B
print(C)
```

```
[[20. 20.]
 [20. 20.]]
```

Transposição de matrizes pode ser feita com a função `transpose` ou simplesmente com a letra " `T` ":

```
A = np.arange(4).reshape(2,2)
print("Transposta de A =\n", A.transpose())
print("Transposta de A =\n", A.T)
```

```
Transposta de A =
 [[0 2]
 [1 3]]
```

Funções universais

O NumPy oferece diversas funções matemáticas clássicas, como exponencial, logaritmo, etc. Essas funções são aplicadas a todos os elementos de um tensor. Exemplo de aplicação da função exponencial:

```
u = np.arange(5)
v = np.exp(u)
print(v)
```

```
[ 1.          2.71828183  7.3890561  20.08553692 54.59815003]
```

Função seno:

```
v = np.sin(u)
print(v)
```

```
[ 0.          0.84147098  0.90929743  0.14112001 -0.7568025 ]
```

Operadores lógicos

As operações lógicas entre tensores ocorrem elemento a elemento. O resultado da operação é um tensor booleano com valores `True` ou `False` que indicam se a condição lógica é verdadeira ou falsa:

```
u = np.arange(4).reshape(2,2)
v = 2*np.ones((2,2))
w = u > v
print(w)
```

```
[[False False]
 [False  True]]
```

Os tensores booleanos podem ser usados para filtrar valores em um tensor, por exemplo:

```
u = np.array([-1, 2, -3])
v = np.array([True, False, True])
print(u[v])
```

```
[-1 -3]
```

Note que podemos obter o mesmo efeito anterior de filtrar somente os valores negativos fazendo:

```
w = u[u < 0]
print(w)
```

```
[-1 -3]
```

E podemos também atribuir um valor específico aos elementos de um tensor que satisfazem uma certa condições, por exemplo zerar todos os valores negativos:

```
print("u =", u)
u[u < 0] = 0
print("u =", u)
```

```
u = [-1  2 -3]
u = [0 2 0]
```

As funções `all` e `any` são muito úteis em comparações lógicas. `all` retorna `True` se todos os elementos de um tensor booleano forem `True`:

```
v = np.arange(5)
u = v >= 0
print('u =', u)
print(np.all(u))
```

```
u = [ True  True  True  True  True]
True
```

enquanto `any` retorna `True` se pelo menos um elemento de um tensor booleano for `True`:

```
v = np.array([-1, 1, 2, 3])
u = v > 0
print('u =',u)
print(np.any(u))
```

```
u = [False  True  True  True]
True
```

Geração de números aleatórios

O NumPy possui um submódulo chamado `random` que possui diversas funções para a geração de números (pseudo) aleatórios. Embora o Python possua uma biblioteca padrão também chamada `random`, a biblioteca do NumPy tem mais funcionalidas e gera diretamente tensores aleatórios.

Criação de um tensor segundo uma distribuição uniforme [0,1):

```
import numpy.random as rd
v = rd.rand(4,4)
print(v)
```

```
[[0.785982   0.84139878 0.09506509 0.15178495]
 [0.60974876 0.99289147 0.7401389  0.4849863 ]
 [0.26441043 0.24523774 0.38216544 0.11126995]
 [0.54827425 0.1749092  0.18705632 0.34502868]]
```

Criação de um tensor em que cada elemento segue uma distribuição normal com $\mu = 10.0$ e $\sigma = 1.0$:

```
v = rd.normal(10, 1, (4,4))
print(v)
```

```
[[12.13441239  9.75742507  9.10398105  8.52091947]
 [10.551759    9.72344626 10.41363376  8.71661515]
 [ 9.91169422  9.26378443  9.95011702  8.67414398]
 [10.46478971 11.56541705  9.6424856  10.30856245]]
```

Note que toda vez que rodarmos o código, os tensores terão valores diferentes. Podemos evitar esse comportamento, de forma que toda vez que o código é executado o tensor aleatório tenha o mesmo valor por meio da função `seed`, cujo argumento é a semente para o gerador de números aleatórios do Python:

```
rd.seed(1000)
v = rd.rand(4)
print(v)
rd.seed(1000)
v = rd.rand(4)
print(v)
```

```
[0.65358959 0.11500694 0.95028286 0.4821914 ]
[0.65358959 0.11500694 0.95028286 0.4821914 ]
```

Álgebra Linear

O NumPy possui um submódulo específico para a realização de operações típicas da álgebra linear chamado `linalg`. Com este módulo é possível resolver sistemas lineares, obter inversas de matrizes, calcular autovalores e autovetores, etc.

Solução de sistemas lineares

A solução de sistemas de equações lineares do tipo $Ax = b$ pode ser realizada por meio da função `solve`:

```
A = np.array([10, 20, 30, 40]).reshape(2,2)
b = np.array([5,10])
x = np.linalg.solve(A, b)
print(x)
```

```
[0.   0.25]
```

Caso $b = [b_1, b_2, \ldots, b_p]$em que cada $b_j, j = 1, \ldots, p$é um vetor coluna, a função `solve` retorna uma matriz $x = [x_1, x_2, \ldots, x_p]$em que cada x_j é solução do sistema linear $Ax_j = b_j$.

```
b1 = np.array([5, 10]).reshape(2,1)
b2 = np.array([10, 12]).reshape(2,1)
b = np.hstack([b1, b2])
print("b =\n", b)
x = np.linalg.solve(A, b)
print("x =\n", x)
```

Note que resolver o sistema linear simultaneamente para b_1 e b_2 é mais eficiente que resolver cada sistema individualmente:

```
     ^ -

def fun(A, b1, b2):
  x1 = np.linalg.solve(A,b1)
  x2 = np.linalg.solve(A,b2)


A = np.random.rand(1000,1000)
b1 = np.random.rand(1000).reshape(1000,1)
b2 = np.random.rand(1000).reshape(1000,1)
b = np.hstack([b1,b2])

%timeit fun(A,b1,b2)
%timeit np.linalg.solve(A,b)
```

```
10 loops, best of 3: 77.5 ms per loop
10 loops, best of 3: 40.5 ms per loop
```

Matriz inversa

O cálculo da inversa A^{-1} de uma matriz A pode ser realizado pela função `inv`:

```
A = np.array([10, 20, 30, 40]).reshape(2,2)
inv_A = np.linalg.inv(A)
print(inv_A)
```

```
[[-0.2   0.1 ]
 [ 0.15 -0.05]]
```

Rank e determinante

```
rank_A = np.linalg.matrix_rank(A)
print("Rank de A =",rank_A)
det_A = np.linalg.det(A)
print("Determinante de A = %.2f" % det_A)
```

```
Rank de A = 2
Determinante de A = -200.00
```

Broadcasting

Em princípio, operações sobre tensores com shapes incompatíveis são indefinidas. Por exemplo, podemos somar dois tensores com shapes (4, 4), resultando em um tensor com também com shape (4,4), uma vez que a operação de soma será aplicada elemento a elemento. No entanto, o que acontece se tentarmos somar um tensor com shape (4,4) e outro com shape (1,4) ? Neste caso, o NumPy realizará o *broadcasting* do vetor (1,4) para poder realizar a operação. Dentro de certas regras de broadcasting, o NumPy permite a aplicações de operações sobre tensores com shapes diferentes.

As dimensões de dois tensores são consideradas compatíveis quando elas são a mesma ou uma delas é 1. Por exemplo, as dimensões dos tensores u = (4,4) e v = (1,4) são compatíveis pois a primeira dimensão do tensor v é 1 e a segunda dimensão de ambos é 4. Neste caso, o tensor v será "estendido"

```
u = np.ones((4,4))
v = np.array([10, 20, 30, 40]).reshape(1,4)
x = u*v
print('u =\n',u)
print('v =\n',v)
print('x =\n',x)
```

```
u =
 [[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
v =
 [[10 20 30 40]]
x =
 [[10. 20. 30. 40.]
 [10. 20. 30. 40.]
 [10. 20. 30. 40.]
 [10. 20. 30. 40.]]
```