29. Ma trận và các phép toán đại số khi làm việc với numpy array

Lớp matrix trong numpy

Các đối tượng ma trận là một phân lớp của các mảng numpy (ndarray). Các đối tượng ma trận kế thừa tất cả các thuộc tính và phương pháp của ndarray. Một khác biệt nữa là các ma trận numpy đúng 2 chiều, trong khi các mảng numpy có thể có kích thước bất kỳ, nghĩa là chúng có thể có n chiều.

Ưu điểm quan trọng nhất của ma trận là cung cấp các ký hiệu thuận tiện cho việc nhân ma trận. Nếu X và Y là hai ma trận, X * Y xác định phép nhân ma trận. Mặt khác, nếu X và Y là ndarrays, X * Y xác định một phép nhân giữa các phần tử.

Trong bài này tôi sẽ trình bầy các bài hướng dẫn qua cửa sổ terminal(CMD).

Ví dụ sau:

```
>>> x = np.matrix(((9,3), (7, 5)))
>>> y = np.matrix(((1,9), (-5, -1)))
>>> x*y
matrix([[ -6,  78],
        [-18,  58]])
>>>
```

Ma trận chuyển vị, dùng phương thức transpose() theo hai cách sau:

```
>>> x.transpose()
matrix([[9, 7],
        [3, 5]])
>>> np.transpose(x)
matrix([[9, 7],
        [3, 5]])
>>>
```

Tạo ma trận từ các khối a,b,c,d qua np.vstack() và np.hstack()

```
>>> a = np.matrix([[1,2],[3,4]])
>>> b = np.matrix([[10,20],[3,4]])
>>> c = np.matrix([[1,20],[30,4]])
>>> d = np.matrix([[10,20],[30,40]])
>>> np.vstack([hstack([a,b]),hstack([c,d])])
>>> np.vstack([np.hstack([a,b]),np.hstack([c,d])])
matrix([[ 1,  2, 10, 20],
        [ 3,  4,  3,  4],
        [ 1, 20, 10, 20],
        [30,  4, 30, 40]])
>>>
```

Tìm vector đường chéo, sử dụng diag(a,start), trong đó a là ma trận, còn start là chỉ số cột bắt đầu của vector đường chéo. Tiếp ví dụ phía trên.

```
>>> np.diag(x)
array([ 1,  4, 10, 40])
>>> np.diag(x,3)
array([20])
>>> np.diag(x,2)
array([10,  4])
>>>
```

Tìm ma trận nghịch đảo dùng linalg.inv(a)

```
>>> np.linalg.inv(x)
matrix([[  1.49718060e+15,   7.48590300e+15,  -1.49718060e+15,-7.48590300e+14],
        [ -5.03355705e-02,   0.00000000e+00,   2.76845638e-01,2.51677852e-02],
        [ -4.19210568e+15,  -2.09605284e+16,   4.19210568e+15,2.09605284e+15],
        [  2.02119381e+15,   1.01059690e+16,  -2.02119381e+15, -1.01059690e+15]])
>>>
```

Tìm ma trận giả nghịch đảo, (Moore-Penrose) pseudo-inverse matrix dùng linalg.pinv(a)

```
>>> a = np.random.randn(4, 5)
>>> a
array([[-0.90607354, -2.26844462,  0.77482667,  1.85008866, -1.01894807],
       [-1.53054814,  0.3192559 ,  0.26676743,  2.6635191 , -0.23887169],
       [ 0.70063524, -0.65967053, -2.4837309 ,  0.00595501, -2.82667483],
       [-1.43172657, -0.92801089, -0.94395084, -0.9834375 , -1.18500965]])
>>> B = np.linalg.pinv(a)
>>> B
array([[ 0.09529477, -0.22712184,  0.16826839, -0.43358596],
       [-0.33888397,  0.24615403,  0.03035296, -0.03570538],
       [ 0.17093252, -0.10647897, -0.16647028, -0.07902336],
       [ 0.07401431,  0.22432364,  0.09424145, -0.23966783],
       [-0.04733158, -0.01970838, -0.17267625, -0.03020732]])
>>> # check a * a+ * a == a
…
>>> np.allclose(a, np.dot(a, np.dot(B, a)))
True
>>> # check a+ * a * a+ == a+
…
>>> np.allclose(B, np.dot(B, np.dot(a, B)))
True
>>>
```

Tìm rank của ma trận dùng linalg.matrix_rank(a), ví dụ phía trên tìm rank của ma trận a.

```
>>> np.linalg.matrix_rank(a)
4
```

Tìm trị riêng và vector riêng của ma trận dùng linalg.eig(a), hàm này sẽ trả về trị riêng và vector riêng tương ứng, ví dụ:

```
>>> a = np.random.randn(4, 4)
>>> a
array([[-0.23636141,  1.15896826,  0.52769395, -0.98600561],
       [-0.6763387 ,  0.288351  , -0.15680697,  0.24142735],
       [ 0.73648075, -0.98730241, -0.21491164,  0.46732308],
       [-0.30138502,  0.65594634,  0.04926585,  0.02432945]])
>>> D,V  = np.linalg.eig(a)
>>> D
array([ 0.72657338+0.j        ,  0.00520488+0.j        ,
       -0.43518542+0.34972468j, -0.43518542-0.34972468j])
>>> V
array([[ 0.09567501+0.j        , -0.14708685+0.j        ,
         0.59508907-0.12882044j,  0.59508907+0.12882044j],
       [-0.64498080+0.j        , -0.14639436+0.j        ,
         0.33287198+0.08588877j,  0.33287198-0.08588877j],
       [ 0.44736688+0.j        ,  0.91256872+0.j        ,
        -0.69560703+0.j        , -0.69560703-0.j        ],
       [-0.61213473+0.j        ,  0.35235277+0.j        ,
         0.09329232-0.1360921j ,  0.09329232+0.1360921j ]])
>>>
```

Các bạn có thể tham khảo nhiều phép toán trong ma trận tại địa chỉ: https://docs.scipy.org/doc/numpy-dev/user/numpy-for-matlab-users.html

Các phép toán trong numpy array

Chúng ta đã thấy rất nhiều toán tử trong khóa học python này. Dĩ nhiên, chúng ta cũng thấy nhiều trường hợp “overloading” của các toán tử, ví dụ: “+” vừa có ý nghĩa cộng đại số với một số hay mang ý nghĩa ghép hai chuỗi.

Chúng ta sẽ học trong phần giới thiệu này rằng các kí hiệu của các toán tử cũng bị ”overloaded” trong Numpy, để chúng có thể được sử dụng theo cách "tự nhiên".

Chẳng hạn ta có thể thực hiện phép cộng một mảng ndarrays tới một số như sau:

```
>>> v = np.random.randint(100,size=10)
>>> v
array([17, 30, 24, 68, 40, 20,  0, 30,  7, 37])
>>> v + 1000
array([1017, 1030, 1024, 1068, 1040, 1020, 1000, 1030, 1007, 1037])
```

```
>>>
```

Ta cũng có thể sử dụng cách trên cho các phép sau: “trừ”, “nhân”, “chia”, “sin”,”cos” … Ngoài thực hiện với một số vô hướng, ta có thể sử dụng giữa hai mảng cùng kích thước. Tôi sẽ đưa ra một vài ví dụ sau cho cả hai trường hợp:

Trường hợp với một số:

```
>>> v = np.random.randint(100,size=10)
>>> v
array([96, 44, 23, 37, 12, 42, 79, 26, 56, 35])
>>> v**2
array([9216, 1936,  529, 1369,  144, 1764, 6241,  676, 3136, 1225])
>>> v - 1000
array([-904, -956, -977, -963, -988, -958, -921, -974, -944, -965])
>>> v/345
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
>>> v/345.0
array([ 0.27826087,  0.12753623,  0.06666667,  0.10724638,  0.03478261,
        0.12173913,  0.22898551,  0.07536232,  0.16231884,  0.10144928])
>>> v*100
array([9600, 4400, 2300, 3700, 1200, 4200, 7900, 2600, 5600, 3500])
```

Trường hợp với hai mảng

```
>>> v1 = np.random.randint(10,size=5)
>>> v2 = np.random.randint(10,size=5)
>>> v1
array([4, 2, 9, 3, 7])
>>> v2
array([4, 1, 4, 7, 8])
>>> v1-v2
array([ 0,  1,  5, -4, -1])
>>> v1+v2
array([ 8,  3, 13, 10, 15])
>>> v1*v2
array([16,  2, 36, 21, 56])
>>> v1**v2
array([    256,       2,    6561,    2187, 5764801])
>>>
```

Nhiều người thấy rằng mảng hai chiều của Numpy giống ma trận. Điều này cơ bản là đúng, bởi vì nó hành xử trong hầu hết các khía cạnh như ý tưởng toán học của ma trận. Chúng tôi thậm chí còn thấy rằng chúng ta có thể thực hiện phép nhân ma trận trên chúng (dot). Tuy nhiên, có một sự khác biệt tinh tế. Trong Numpy thực sự có định nghĩa ma trận. Chúng là tập con của mảng hai chiều. Chúng ta có thể biến một mảng hai chiều thành

một ma trận bằng cách sử dụng hàm "mat". Sự khác biệt chính thể hiện, nếu bạn nhân hai mảng hai chiều hoặc hai ma trận. Chúng tôi nhận được phép nhân ma trận thực bằng cách nhân hai ma trận (dot), nhưng các mảng hai chiều sẽ chỉ được nhân giữa các phần tử của mảng với nhau:

```
>>> A = np.random.randint(20,size=(3,3))
>>> A
array([[ 6,  7, 10],
       [ 4,  3,  3],
       [ 4,  8, 10]])
>>> B = np.random.randint(20,size=(3,3))
>>> B
array([[10,  3, 19],
       [ 1, 13, 18],
       [11,  4, 12]])
>>> A*B
array([[ 60,  21, 190],
       [  4,  39,  54],
       [ 44,  32, 120]])
>>>
```

Ta có thể kiểm chứng kết quả của hai ma trận từ A và từ B như sau:

```
>>> MA = np.mat(A)
>>> MB = np.mat(B)
>>> MA
matrix([[ 6,  7, 10],
        [ 4,  3,  3],
        [ 4,  8, 10]])
>>> MB
matrix([[10,  3, 19],
        [ 1, 13, 18],
        [11,  4, 12]])
>>> MA * MB
matrix([[177, 149, 360],
        [ 76,  63, 166],
        [158, 156, 340]])
>>>
```

Phép nhân ma trận cũng có thể thực hiên qua hàm (dot) trực tiếp từ hai mảng A và B:

```
>>> A.dot(B)
array([[177, 149, 360],
       [ 76,  63, 166],
```

```
      [158, 156, 340]])

>>> np.dot(A,B)

array([[177, 149, 360],

      [ 76,  63, 166],

      [158, 156, 340]])

>>>
```

Chúng ta vẫn sử dụng toán tử so sánh cho các kiểu cơ bản như integer, float hay string. Tuy nhiên, numpy còn hỗ trợ việc so sánh giữa hai mảng dựa trên so sánh giữa các phần tử cùng index trong mảng với nhau và trả về một mảng các giá trị True, False.

```
>>> np.array([1,2,3]) > np.array([3,4,5])
array([False, False, False], dtype=bool)
>>>
```

Chúng ta có thể dùng các hàm logical để thực hiên việc ‘and’, ‘or’, ‘xor’ giữa hai mảng với nhau, các phép toán đó sẽ trả về mảng các giá trị True, False như phép so sánh phía trên: có thể dùng np.logical_or, np.logical_and, np.logical_xor.

```
>>> A = np.array([1,0,0,1])

>>> B = np.array([1,1,0,0])

>>> np.logical_xor(A,B)

array([False,  True, False,  True], dtype=bool)

>>> np.logical_and(A,B)

array([ True, False, False, False], dtype=bool)

>>> np.logical_or(A,B)

array([ True,  True, False,  True], dtype=bool)
```

Numpy cung cấp một cơ chế mạnh mẽ, được gọi là “Broadcasting”, cho phép thực hiện các phép tính số học trên các mảng có hình dạng khác nhau. Điều này có nghĩa là chúng ta có thể thực hiện các phép toán đại số giữa một mảng nhỏ hơn và một mảng lớn hơn. Nói cách khác: Trong điều kiện nhất định, mảng nhỏ hơn là " Broadcasting" trong một cách nào đó để nó có hình dạng giống như mảng lớn hơn. Với sự trợ giúp của “Broadcasting” chúng ta có thể tránh các vòng lặp trong chương trình Python. Chuỗi xảy ra ngầm trong triển khai Numpy. Chúng tôi cũng tránh tạo các bản sao dữ liệu không cần thiết. Chúng tôi trình bày nguyên tắc hoạt động “Broadcasting” trong các ví dụ đơn giản sau:

```
>>> A = np.random.randint(20,size=(3,3))
>>> A
array([[11, 15,  8],
      [18,  8,  3],
      [ 6,  1, 17]])
>>> B = np.random.randint(20,size=3)
>>> B
array([12, 11, 14])
>>> A*B
array([[132, 165, 112],
      [216,  88,  42],
      [ 72,  11, 238]])
>>>
```

Qua ví dụ trên bạn sẽ thấy A có shape = (3,3) nhưng lại có thể nhân với B có shape = 3.

Trong phép nhân phía trên B sẽ được hiểu như là: np.array([[12, 11, 14],] * 3)

Nếu B là một mảng cột thì phép tính sẽ được hiểu như sau:

```
>>> B = np.random.randint(20,size=(3,1))
>>> B
array([[ 7],
       [ 3],
       [19]])
>>> A*B
array([[ 77, 105,  56],
       [ 54,  24,   9],
       [114,  19, 323]])
>>>
```

Ta còn có thể thực hiện “Broadcasting” với mảng nhiều chiều hơn 3 chiều như ví dụ dưới đây.
Sẽ được sử dụng nhiều trong xử lý ảnh!

```
>>> A = np.array([ [[3, 4, 7], [5, 0, -1] , [2, 1, 5]],
...       [[1, 0, -1], [8, 2, 4], [5, 2, 1]],
...       [[2, 1, 3], [1, 9, 4], [5, -2, 4]]])
>>> B = np.array([ [[3, 4, 7], [1, 0, -1], [1, 2, 3]] ])
>>> B * A
array([[[ 9, 16, 49],
        [ 5,  0,  1],
        [ 2,  2, 15]],
```

Trong ví dụ sau sẽ chứng minh sự hữu ích trong “Broadcasting”:
Trong toán học, khoa học máy tính và đặc biệt là lý thuyết đồ thị, một ma trận khoảng cách là một ma trận hoặc một mảng hai chiều, trong đó có khoảng cách giữa các phần tử của tập, được ghép đôi. Kích thước của mảng hai chiều này là n x n, nếu tập hợp bao gồm n phần tử. Một ví dụ thực tế về ma trận khoảng cách là một ma trận khoảng cách giữa các vị trí địa lý, ví dụ tại các thành phố Eurpean của chúng ta.

```
>>> cities = ["Ha Noi","Thanh Hoa","Nghe An","Sai Gon"]
>>> dist2Hanoi = [0,100,300,3000]
>>> dist2Hanoi = np.array([0,100,300,3000])
>>> dist2Hanoi
array([   0,  100,  300, 3000])
>>> dist2Hanoi[:,np.newaxis]
array([[   0],
       [ 100],
       [ 300],
       [3000]])
>>> np.abs(dist2Hanoi - dist2Hanoi[:,np.newaxis])
array([[   0,  100,  300, 3000],
       [ 100,    0,  200, 2900],
       [ 300,  200,    0, 2700],
       [3000, 2900, 2700,    0]])
>>>
```

Kết luận

Trong bài này chúng tôi đề cập đến hai nội dung chính.

Thứ nhất là về các đối tượng ma trận và các phép toán trên lớp ma trận trong thư viện numpy; bao gồm phép nhân hai ma trận, tạo ma trận chuyển vị với np.transpose(), tạo ma trận qua np.vstack() và np.hstack(), tìm vector đường chéo sử dụng diag(), tìm ma trận nghịch đảo dùng linalg.inv(a) …

Thứ hai là về các phép toán trên đối tượng mảng trong thư viện numpy. Ta có thể thực hiện rất nhiều phép toán (phép toán số học, logic, so sánh, …) trên các đối tượng mảng có cùng kích thước. Numpy cũng cung cấp cơ chế “Broadcasting”, cho phép thực hiện các phép tính số học trên các mảng có hình dạng khác nhau.

Một điểm cần lưu ý nữa đó là mảng hai chiều của Numpy về cơ bản khá giống ma trận tuy nhiên chúng có sự

khác biệt trong phép nhân hai mảng hai chiều hoặc hai ma trận.