

NumPy

☑ Trắc nghiệm 26 ☒ Thực hành 1 ⬇ PDF bài

NumPy là thư viện tính toán trên mảng số. Thay vì viết vòng lặp qua từng ô như trong C++, ta viết một phép tính cho cả mảng; ví dụ trừ điểm trung bình của từng môn khỏi cả bảng điểm chỉ cần một dòng. Ba khái niệm cần nắm là **shape** (kích thước mỗi chiều), **axis** (tính theo hàng hay theo cột) và **broadcasting** (cách NumPy tính trên hai mảng khác kích thước). Bài dùng một bảng điểm ba môn nhỏ để mọi kết quả đều kiểm được bằng tay. pandas và PyTorch ở các bài sau đều dựa trên các khái niệm này.

MỤC TIÊU

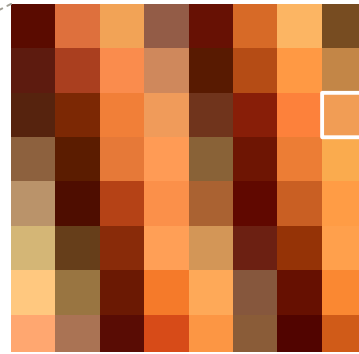
- Thay vòng lặp bằng phép tính trên cả mảng.
- Đoán shape kết quả khi tính theo axis và khi broadcasting.
- Phân biệt view và copy.
- Tránh các lỗi hay gặp: trừ mảng $(n,)$ với $(n, 1)$, tràn số uint8.

Mảng và shape

Mảng NumPy (ndarray) giống mảng của C++ hơn là list của Python: mọi phần tử cùng một kiểu số, gọi là dtype, và mảng có kích thước cố định theo từng chiều, gọi là shape. Vì mọi phần tử cùng kiểu, NumPy chạy vòng lặp bên trong bằng code C, nhanh hơn nhiều so với vòng lặp Python.

Ảnh là ví dụ gặp nhiều nhất. Ảnh màu là mảng shape (cao, rộng, 3): ảnh (480, 640, 3) cao 480 và rộng 640 điểm ảnh. Mỗi điểm ảnh có ba kênh đỏ, lục, lam; mỗi kênh lưu bằng 8 bit kiểu uint8, tức $2^8 = 256$ mức từ 0 tới 255. Số càng lớn càng sáng: (0, 0, 0) là đen, (255, 255, 255) là trắng.

(427, 640, 3)



R 241
G 157
B 85

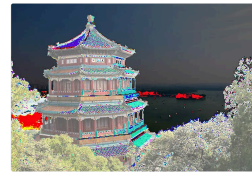
Ảnh màu là mảng (cao, rộng, 3): mỗi điểm ảnh là ba số R, G, B.



`img[:, ::-1] → (320, 480, 3):`
đảo thứ tự cột: lật ngang



`img[::8, ::8] → (40, 60, 3):`
lấy mỗi 8 hàng, 8 cột một điểm



`img + 100 → (320, 480, 3),`
uint8 tràn vòng: $200 + 100$
trong uint8 thành 44: màu sáng
bị vỡ



`np.clip(img.astype(int) + 100, 0, 255) → (320, 480, 3):`
đổi sang int rồi kẹp về 0-255

Một ảnh thật (320, 480, 3) qua mười dòng NumPy: lát cắt, chọn kênh, gộp trục, broadcast, và cái bẫy tràn số uint8.

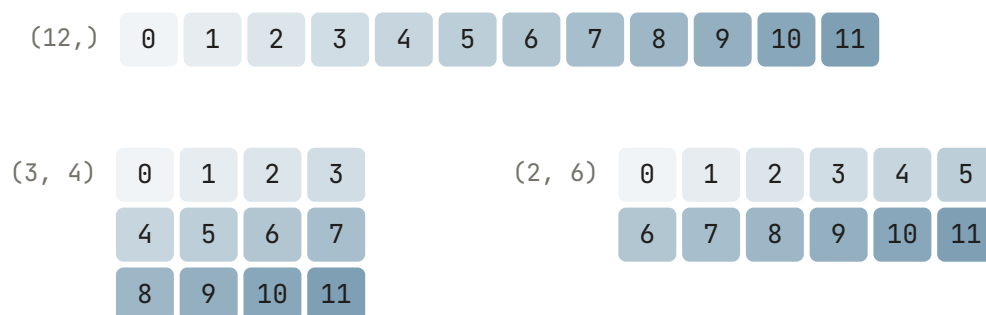
shape trả lời “có bao nhiêu phần tử trên mỗi trục”, ndim là số trục, size là tích các độ dài trục. `np.zeros((3, 4))` tạo mảng shape (3,4) toàn số 0, size là 12. `np.arange(0, 10, 2)` giống range: được [0 2 4 6 8], không lấy 10. Với bảng hai hàng và ba cột, phần tử `a[1, 2]` nằm ở hàng thứ hai, cột thứ ba:

Python

```
import numpy as np

a = np.array([[1, 2, 3], [4, 5, 6]], dtype=np.int64)
print(a.shape, a.ndim, a.size, a.dtype)
print(np.zeros((2, 3)).dtype, np.arange(6).reshape(2, 3).tolist())
print(np.linspace(0, 1, 5))
print(np.arange(12).reshape(3, -1).shape)    # -1: NumPy tự tính trục còn lại
```

reshape giữ nguyên dãy giá trị và thứ tự, chỉ đổi cách chia hàng: 12 số có thể đọc thành 3×4 hay 2×6 , miễn tích các trục bằng 12. Chỉ được viết -1 ở một trục; NumPy tự tính độ dài trục đó.



Cùng 12 số, cùng thứ tự; reshape chỉ đổi cách chia hàng.

Kiểu dữ liệu là chỗ NumPy khác Python nhiều nhất. `int` Python có độ chính xác tùy ý, nhưng số nguyên NumPy có độ rộng cố định. Phép toán trên mảng nguyên có thể quay vòng khi vượt miền biểu diễn, không nhất thiết báo lỗi:

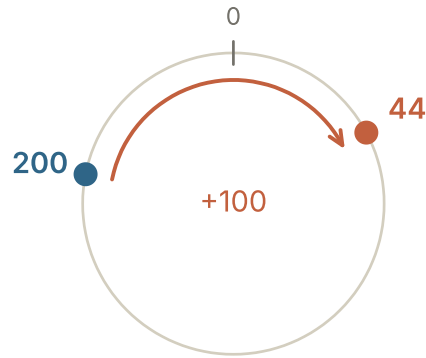
Python

```
big = np.array([2**62], dtype=np.int64)
print(big * 4, big.dtype)

pixels = np.array([200, 100], dtype=np.uint8) # ảnh thường lưu bằng uint8, giá trị 0..255
print(pixels + 100)

x = np.array([1, 2, 3])
x[0] = 2.7 # gán số thực vào mảng nguyên: bị cắt
print(x)
```

$2^{62} \cdot 4 = 2^{64}$ vượt `int64` và quay vòng về 0 trong phép toán này. Điểm ảnh $200 + 100 = 300$ vượt 255 và thành $300 - 256 = 44$. Đổi sang `float32` trước khi xử lý rồi kẹp về miền hợp lệ bằng `np.clip` sẽ tránh phép cộng `uint8` bị quay vòng. Dòng cuối cho thấy gán số thực vào mảng nguyên làm mất phần lẻ.



uint8 chạy trên một vòng 256 giá trị: $200 + 100$ thành 44.

Với số thực, float64 dùng 8 byte mỗi số, float32 dùng 4 byte. float64 chính xác hơn, nhưng mạng neural thường dùng float32 vì tốn nửa bộ nhớ và chạy nhanh hơn. Nhân lớp và chỉ số thì vẫn để kiểu nguyên.

HỎI NHANH

`np.array([250, 10], dtype=np.uint8) + 10` cho ra gì?

A [260, 20]

B [4, 20]

C [255, 20]

D Báo lỗi tràn số

Tính trên cả mảng

Phép toán số học, so sánh và các hàm như `np.exp`, `np.sqrt`, `np.log` áp dụng lên từng phần tử của mảng mà không cần vòng lặp. Cách viết này gọi là vector hóa (vectorization):

Python

```
z = np.array([-2.0, 0.0, 1.0, 3.0])
print(1 / (1 + np.exp(-z)))
```

sigmoid của bài 2.3, cho cả mảng

```
print(z > 0, (z > 0).sum(), (z > 0).mean())
print(np.where(z > 0, z, 0.0)) # ReLU: giữ số dương, số âm thành 0
print(np.clip(z, -1, 1), z.max(), z.argmax())
```

So sánh cho ra mảng bool; cộng mảng bool thì True tính là 1, nên `(z > 0).sum()` đếm số phần tử dương và `(z > 0).mean()` là tỉ lệ của chúng. Đây là cách tính accuracy: `(pred == y).mean()`.

HỎI NHANH

Với `z = np.array([-2.0, 0.0, 1.0, 3.0])`, dòng `if z > 0: print("dương")` làm gì?

- A In "dương", vì z có ít nhất một phần tử dương
- B Không in gì, vì không phải mọi phần tử đều dương
- C Báo ValueError, vì if không tự chọn any hay all
- D In "dương" hai lần, mỗi phần tử dương một lần

Dòng đầu tính hàm sigmoid $1/(1 + e^{-z})$ cho cả bốn số cùng lúc; bài 0.4 sẽ học hàm này. Ở đây chỉ cần thấy một công thức được áp vào từng phần tử mà không cần vòng lặp. Thử so tốc độ với vòng lặp Python trên 200 000 số. Như ở bài 0.1, kiểm hai cách ra cùng kết quả (bằng `np.allclose`) rồi mới đo:

Python

```
import math
from timeit import repeat

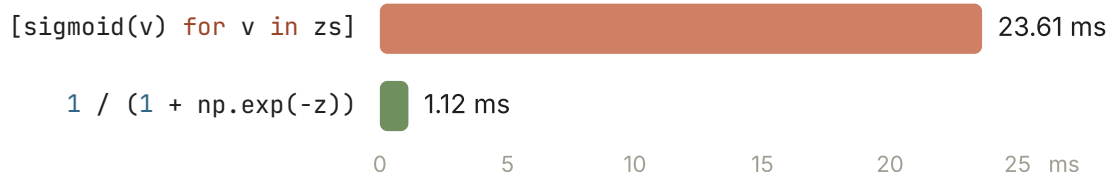
rng = np.random.default_rng(0)
z = rng.normal(size=200_000)
zs = z.tolist()

def sigmoid_loop(values):
    return [1 / (1 + math.exp(-value)) for value in values]
```

```
def sigmoid_numpy(values):
    return 1 / (1 + np.exp(-values))

p_loop = sigmoid_loop(zs)
p_vec = sigmoid_numpy(z)
assert np.allclose(p_loop, p_vec)
t_loop = min(repeat(lambda: sigmoid_loop(zs), repeat=5, number=3)) / 3
t_vec = min(repeat(lambda: sigmoid_numpy(z), repeat=5, number=3)) / 3
print(np.allclose(p_loop, p_vec), p_vec.shape)
print("NumPy nhanh hơn vòng lặp:", t_vec < t_loop)
```

Hai cách cho cùng kết quả, chỉ lệch ở chữ số cuối, và NumPy nhanh hơn. Số giây trên máy đang chạy nằm trong `t_loop` và `t_vec`; hình dưới vẽ từ một lần đo như vậy.



Sigmoid của 200 000 số, đo trên một máy CPU.

Chỉ số và lát cắt

Mảng nhiều chiều dùng một cặp ngoặc với các chỉ số ngăn bởi dấu phẩy: `a[i, j]`, không phải `a[i][j]` (cách sau vẫn chạy nhưng chậm hơn và dễ sai với lát cắt). Mỗi trục nhận một chỉ số, một lát cắt `start:stop:step`, một list chỉ số, hoặc một mặt nạ bool:

Python

```
a = np.arange(12).reshape(3, 4)
print(a)
print(a[1, 2], a[:, 0], a[-1, ::2])
print(a[[0, 2]].shape, a[a % 5 == 0])
```

`a[0]` (hay `a[0, :]`) là hàng đầu, `a[:, 0]` là cột đầu, `(3,)`. `a[[0, 2]]` lấy hàng 0 và hàng 2 (fancy indexing). `a[a % 5 == 0]` lấy mọi phần tử chia hết cho 5 và trả về mảng một chiều.

<code>a[1, 2]</code>	<code>a[:, 0]</code>	<code>a[-1, ::2]</code>	<code>a[a % 5 == 0]</code>
0 1 2 3	0 1 2 3	0 1 2 3	0 1 2 3
4 5 6 7	4 5 6 7	4 5 6 7	4 5 6 7
8 9 10 11	8 9 10 11	8 9 10 11	8 9 10 11

Bốn cách lấy dữ liệu từ cùng một mảng 3×4 .

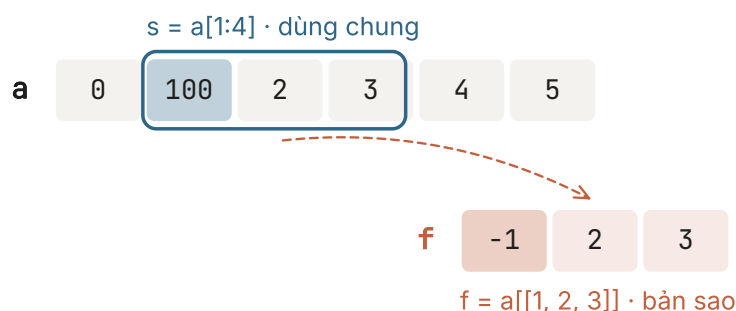
Có một khác biệt quan trọng với list. Lát cắt của list Python tạo bản sao; lát cắt của mảng NumPy là một **view**: một cách nhìn khác vào cùng vùng nhớ. Sửa view thì mảng gốc đổi theo:

Python

```
a = np.arange(6)
s = a[1:4]          # view
s[0] = 100
print(a)

f = a[[1, 2, 3]]    # fancy indexing: luôn là bản sao
f[0] = -1
print(a, np.shares_memory(a, s), np.shares_memory(a, f))
```

Quy tắc: lát cắt bằng `start:stop:step` cho view; chỉ số bằng list hay mặt nạ bool cho bản sao. View giúp NumPy không tốn bộ nhớ khi cắt mảng lớn, nhưng nếu định sửa lát cắt mà không muốn đụng tới gốc thì phải gọi `.copy()`. `reshape` cũng thường trả về view; riêng khi mảng đã chuyển vị hay cắt cách quãng, NumPy phải chép ra bản sao. Muốn biết hai mảng có dùng chung dữ liệu không thì gọi `np.shares_memory(a, b)`.



Lát cắt là view của cùng vùng nhớ; fancy indexing chép ra bản sao.

Lưu ý: “mặt nạ cho bản sao” chỉ đúng khi **lấy ra** một mảng mới, như `b = a[a > 5]`; sửa `b` sau đó không đụng tới `a`. Còn **gán thẳng** `a[a > 5] = 0` thì sửa đúng các ô đó của `a`.

HỎI NHANH

Chạy `a = np.zeros(5)`, `b = a[2:]`, `b += 1`. Lúc này `a` là gì?

A `[0, 0, 0, 0, 0]`

B `[1, 1, 1, 1, 1]`

C `[0, 0, 1]`

D `[0, 0, 1, 1, 1]`

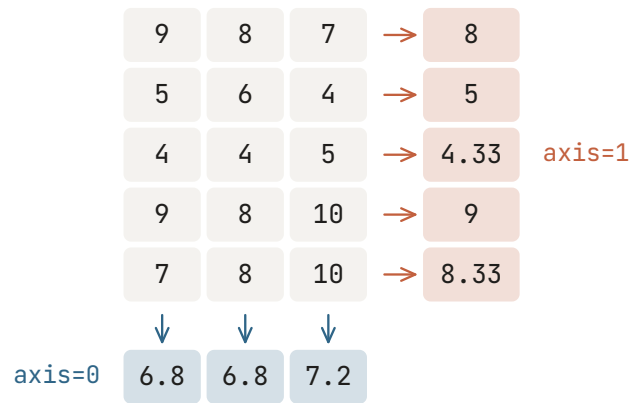
axis

Giờ đến bảng điểm: 5 học sinh, 3 môn. Mỗi hàng là một học sinh, mỗi cột là một môn. Ý nghĩa của từng trục do ta quy ước; trong học máy, quy ước gần như luôn là **trục 0 là các mẫu**.

Python

```
scores = np.array([[9, 8, 7],
                  [5, 6, 4],
                  [4, 4, 5],
                  [9, 8, 10],
                  [7, 8, 10]], dtype=float)    # (5 học sinh, 3 môn)
print(scores.mean(axis=0))                  # trung bình từng môn
print(scores.mean(axis=1).round(2))         # trung bình từng học sinh
print(scores.argmax(axis=1), scores.max())
```

Cách nhớ: **trục nào được ghi trong axis thì trục đó biến mất**. `scores` có shape `(5, 3)`; `mean(axis=0)` gộp theo trục 0 (gộp 5 học sinh lại) nên còn `(3,)`, một số cho mỗi môn. `mean(axis=1)` gộp 3 môn nên còn `(5,)`, một số cho mỗi học sinh. Không ghi `axis` thì gộp tất cả thành một số.



Trục ghi trong axis là trục bị gộp và biến mất.

Quy tắc giữ nguyên với mảng nhiều chiều hơn. Một batch 100 ảnh xám 28×28 có shape (100, 28, 28):

Python

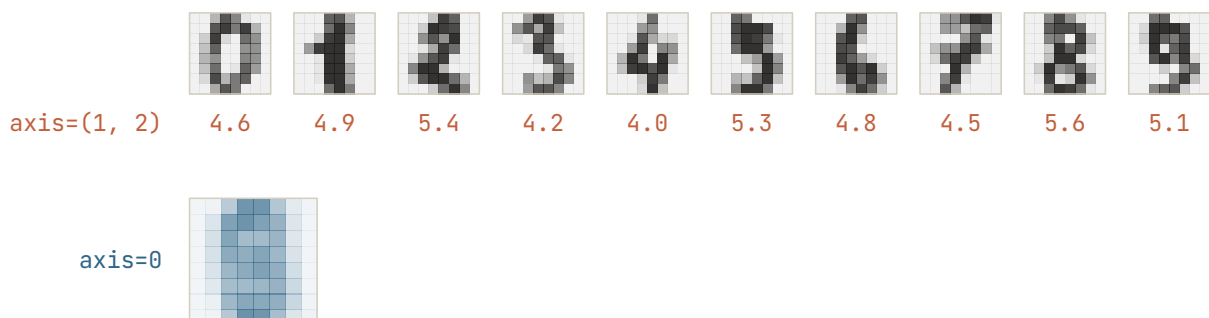
```

imgs = rng.random((100, 28, 28))
print(imgs.mean(axis=0).shape)           # ảnh trung bình
print(imgs.mean(axis=(1, 2)).shape)      # độ sáng trung bình của từng ảnh
print(imgs.mean(axis=(1, 2), keepdims=True).shape)

```

keepdims=True giữ lại trục bị gộp với độ dài 1 thay vì xóa nó. Mục sau sẽ cho thấy vì sao điều đó hữu ích.

Hình dưới làm hai phép gộp đó trên ảnh thật: 1797 ảnh chữ số viết tay 8×8 có sẵn trong scikit-learn. Gộp theo trục 0 được một ảnh trung bình; gộp theo trục 1 và 2 được độ sáng của từng ảnh.



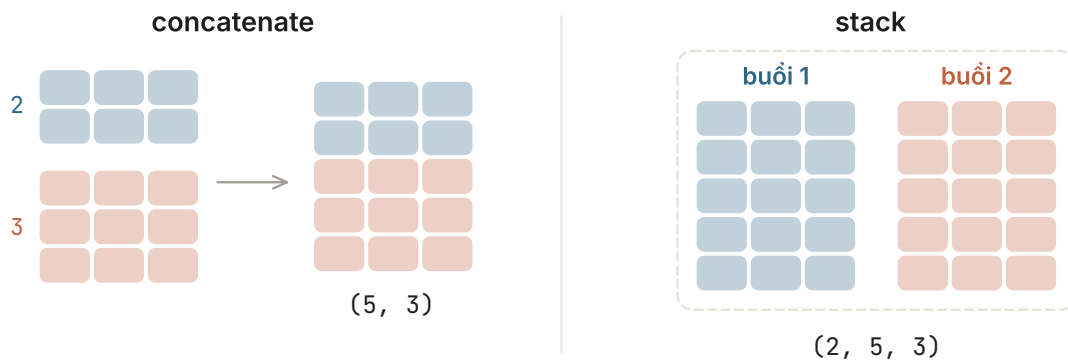
1797 chữ số viết tay 8×8 : độ sáng từng ảnh và ảnh trung bình.

Ghép hai mảng cũng phải chọn trục. concatenate nối hai mảng dọc theo một trục đã có, ví dụ ghép 2 học sinh đầu với 3 học sinh sau thành bảng 5 học sinh. stack thêm một trục mới, ví dụ xếp bảng điểm của hai buổi kiểm tra chồng lên nhau:

Python

```
joined = np.concatenate([scores[:2], scores[2:]], axis=0)
two_sessions = np.stack([scores, scores], axis=0)
print(joined.shape, np.array_equal(joined, scores))
print(two_sessions.shape)    # (buổi, học sinh, môn)
```

Shape (2, 5, 3) có thêm một trục mới ở đầu: trục buổi kiểm tra. Ghép sai trục thì code vẫn chạy, nhưng hàng và cột không còn mang ý nghĩa như ta nghĩ.



concatenate nối trục đã có; stack thêm một trục mới.

HỎI NHANH

X có shape (32, 3, 64, 64), tức là 32 ảnh màu, mỗi ảnh 3 kênh, 64 × 64 điểm ảnh.

X.mean(axis=(0, 2, 3)) có shape gì?

A (3,)

B (32,)

C (64, 64)

D (32, 3)

Broadcasting

Chuẩn hóa từng môn: lấy scores (5, 3) trừ trung bình từng môn (3,). Hai mảng khác shape, nhưng phép trừ vẫn chạy:

Python

```
mu = scores.mean(axis=0)
sigma = scores.std(axis=0)
sigma_safe = np.where(sigma == 0, 1.0, sigma)
Z = (scores - mu) / sigma_safe
print(Z.shape, Z.mean(axis=0).round(6), Z.std(axis=0))
```

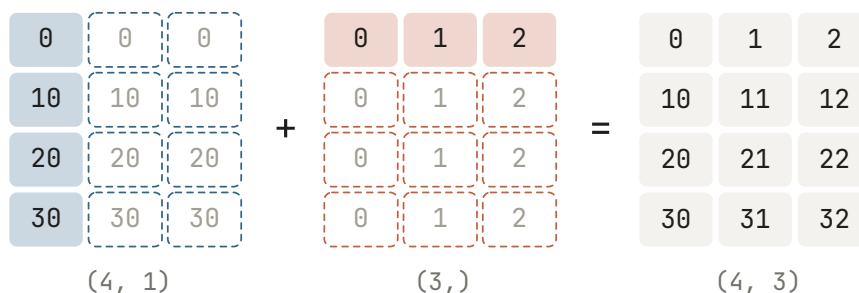
sigma là độ lệch chuẩn của từng môn: điểm của môn càng trải xa trung bình thì sigma càng lớn (bài 0.7 sẽ học kỹ). Mỗi ô của Z cho biết điểm đó cao hơn hay thấp hơn trung bình môn bao nhiêu lần độ lệch chuẩn, nên so được giữa các môn. Output cho thấy mỗi cột của Z có trung bình 0 và độ lệch chuẩn 1 (-0. là một số âm rất nhỏ làm tròn thành 0). sigma_safe thay độ lệch chuẩn bằng 1 ở môn nào mọi điểm giống nhau, để không chia cho 0.

Phép trừ scores - mu chạy được dù hai mảng khác shape: NumPy lấy mu (3,) trừ vào từng hàng của bảng (5, 3), như thể mu được chép thành 5 hàng. Cơ chế này gọi là broadcasting. NumPy không chép thật mu ra 5 lần, nhưng mảng kết quả vẫn đủ 5×3 ô. Có hai quy tắc:

1. Xếp shape của hai mảng thẳng hàng **từ bên phải**. Mảng nào ít trục hơn thì coi như có thêm các trục độ dài 1 ở bên trái.
2. Ở mỗi trục, hai độ dài phải bằng nhau, hoặc một trong hai bằng 1. Trục độ dài 1 được kéo dẫn cho khớp trục kia.

A	B	Kết quả	Vi sao
(5, 3)	(3,)	(5, 3)	(3,) thành (1, 3), kéo dẫn trục 0
(5, 3)	(5, 1)	(5, 3)	Kéo dẫn trục 1 của B
(4, 1)	(3,)	(4, 3)	Cả hai cùng được kéo dẫn

A	B	Kết quả	Vì sao
(5,3)	(5,)	Lỗi	(5,) thành (1,5); trục cuối 3 $\neq$ 5



Trục độ dài 1 được kéo dẫn cho khớp; ô nét đứt không bị chép thật.

Dòng cuối của bảng là lỗi thường gặp khi muốn trừ trung bình **của từng học sinh**:

`scores.mean(axis=1)` có shape (5,), xếp thẳng từ phải với (5, 3) thì không khớp. Đây là lúc dùng `keepdims`:

Python

```
row_mean = scores.mean(axis=1)
try:
    scores - row_mean
except ValueError as e:
    print("ValueError:", e)
print((scores - scores.mean(axis=1, keepdims=True)).shape)
```

Với `keepdims=True`, trung bình có shape (5, 1) và kéo dẫn đúng theo trục môn học.

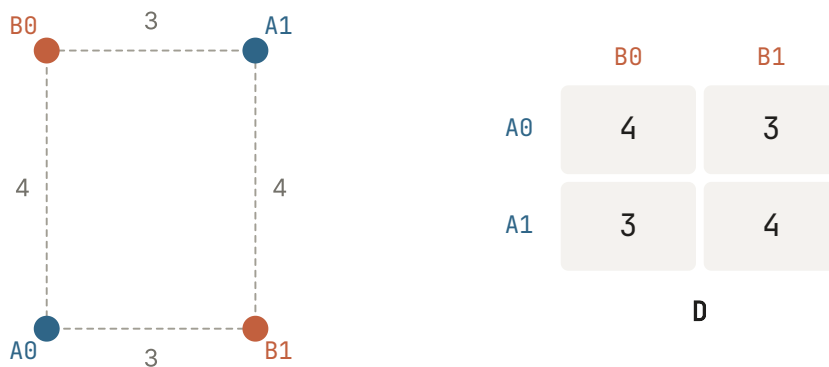
Broadcasting còn thay được cả vòng lặp lồng nhau. Tính khoảng cách giữa mọi cặp điểm của hai tập A (n điểm) và B (m điểm) trong d chiều, việc mà bài 2.4 (k-nearest neighbors) cần, bằng cách chèn thêm trục độ dài 1:

Python

```
A = np.array([[0.0, 0.0], [3.0, 4.0]])
B = np.array([[0.0, 4.0], [3.0, 0.0]])
diff = A[:, None, :] - B[None, :, :] # (2, 1, 2) - (1, 2, 2) -> (2, 2, 2)
D = np.sqrt((diff ** 2).sum(axis=-1))
print(diff.shape, D.shape)
```

```
print(D)
print(diff.nbytes)
```

None (hoặc `np.newaxis`) chèn một trục độ dài 1. Trục cuối chứa tọa độ; gộp nó để lấy một khoảng cách cho mỗi cặp, shape (n, m) . Cặp đầu là từ $(0, 0)$ đến $(0, 4)$, khoảng cách bằng 4; cặp hàng 1, cột 0 là từ $(3, 4)$ đến $(0, 4)$, khoảng cách bằng 3. Bốn ô đều tính tay được, nên ví dụ nhỏ này là phép thử cho code trước khi chạy trên dữ liệu thật.



Ô (i, j) của D là khoảng cách từ A_i tới B_j .

Cách này tiện nhưng tốn bộ nhớ: mảng `diff` có $n \cdot m \cdot d$ số, mỗi số 8 byte. Ở đây chỉ 64 byte, nhưng với $n = m = 10\,000$ điểm, $d = 64$ chiều thì `diff` cần 51.2 GB, chưa kể `diff` ** 2. Vì vậy trước khi chạy trên dữ liệu lớn, hãy tính thử kích thước các mảng trung gian; bài 2.4 sẽ có cách tính khoảng cách mà không cần mảng `diff`.

HỎI NHANH

A có shape $(8, 1, 6)$, B có shape $(7, 1)$. $(A + B).shape$ là gì?

A Báo lỗi, vì hai mảng khác số trục

B $(8, 7, 1)$

C $(8, 1, 6)$

D $(8, 7, 6)$

$(n,)$ và $(n, 1)$

Hai shape này trông giống nhau và in ra gần giống nhau, nhưng là hai thứ khác hẳn. $(n,)$ là vector một trục. $(n, 1)$ là ma trận n hàng, một cột. Trộn chúng với nhau là lỗi âm thầm nguy hiểm nhất của NumPy, vì broadcasting biến nó thành một phép tính hợp lệ nhưng sai:

Python

```
y = np.array([1.0, 2.0, 3.0])          # (3,): nhãn thật
y_hat = np.array([[1.0], [2.0], [3.0]]) # (3, 1): dự đoán, đúng hoàn toàn
print((y_hat - y).shape)
print("MSE sai:", np.mean((y_hat - y) ** 2), " MSE đúng:", np.mean((y_hat.ravel() - y) ** 2))
```

Dự đoán đúng hoàn toàn mà MSE ra 1.33 thay vì 0. Phép trừ $(3, 1) - (3,)$ bị broadcast thành ma trận 3×3 chứa mọi hiệu $\hat{y}_i - y_j$, rồi `np.mean` lấy trung bình cả 9 số. NumPy không báo lỗi gì. Với 10^5 mẫu, ma trận này có 10^{10} số, lúc đó notebook mới báo hết bộ nhớ.

		y (3,)			
		1	2	3	
y_hat (3, 1)	1	0	-1	-2	MSE sai 1.333 MSE đúng 0
	2	1	0	-1	
	3	2	1	0	

$(3, 1) - (3,)$ thành bảng 3×3 ; chỉ đường chéo là phần dư thật.

Lỗi này hay gặp khi model trả dự đoán dạng $(n, 1)$, ví dụ mạng neural có một đầu ra. Cách sửa là đưa về $(n,)$ bằng `.ravel()` hoặc `.reshape(-1)`. Cách phòng là kiểm shape ngay trong hàm tính loss:

Python

```
def mse(y_hat, y):
    assert y_hat.shape == y.shape, (y_hat.shape, y.shape)
```

```
return np.mean((y_hat - y) ** 2)

try:
    mse(y_hat, y)
except AssertionError as e:
    print("AssertionError:", e)
```

HỎI NHANH

`y_hat` có shape `(1000, 1)`, còn `y` có shape `(1000,)`. Ta tính `np.mean((y_hat - y) ** 2)`. Kết quả ra sao?

- A Ra một con số sai, và không báo lỗi gì
- B Báo lỗi, vì hai shape khác nhau
- C Ra đúng MSE, vì NumPy tự khớp hai shape
- D Chỉ tính trên mẫu đầu tiên của mỗi mảng

Đại số tuyến tính và số ngẫu nhiên

Phép nhân ma trận dùng `@`. Phép `*` là nhân từng phần tử, không phải nhân ma trận:

Python

```
M = np.array([[2.0, 1.0], [1.0, 3.0]])
v = np.array([1.0, 2.0])
print(M @ v, M * v)
print(M.T @ M)
print(np.linalg.solve(M, v), np.linalg.norm(v))
```

- `M @ v`: mỗi hàng của `M` nhân từng cặp với `v` rồi cộng lại. Hàng đầu: $2 \cdot 1 + 1 \cdot 2 = 4$; hàng sau: $1 \cdot 1 + 3 \cdot 2 = 7$. Kết quả có shape `(2,)`.
- `M * v`: nhân từng ô, theo broadcasting: cột đầu nhân 1, cột sau nhân 2.

- M.T: chuyển vị, đổi hàng thành cột; ma trận $(2, 3)$ thành $(3, 2)$.

Nhân hai ma trận theo quy tắc shape $(n, k) @ (k, m) \rightarrow (n, m)$: số cột bên trái phải bằng số hàng bên phải. Với hai vector cùng độ dài, $a @ b$ là tích vô hướng: $(1, 2, 3) \cdot (4, 5, 6) = 4 + 10 + 18 = 32$. Bài 0.5 sẽ giải thích phép nhân này kỹ hơn.

M @ v

$$\begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix} @ \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 4 \\ 7 \end{bmatrix}$$

$$2 \cdot 1 + 1 \cdot 2 = 4$$

M * v

$$\begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix} * \begin{bmatrix} 1 & 2 \end{bmatrix} = \begin{bmatrix} 2 & 2 \\ 1 & 6 \end{bmatrix}$$

nhân từng ô

@ là tích vô hướng từng hàng; * nhân từng ô.

`np.linalg.solve(M, v)` giải hệ phương trình $M\mathbf{x} = \mathbf{v}$, tức $2x_0 + x_1 = 1$ và $x_0 + 3x_1 = 2$. Kết quả $(0.2, 0.6)$; kiểm tay: $2 \cdot 0.2 + 0.6 = 1$ và $0.2 + 3 \cdot 0.6 = 2$. `np.linalg.norm(v)` là độ dài của vector: $\sqrt{1^2 + 2^2} \approx 2.236$.

HỎI NHANH

Ma trận A có 3 hàng, 4 cột. Với mọi $\mathbf{x} = (x_0, x_1, x_2, x_3)$, ta có $A\mathbf{x} = (x_2 - x_3, x_3, x_0 + 2x_1)$. Hàng đầu của A là gì?

A [1, -1, 0, 0]

B [0, 0, 1, 0]

C [0, 0, 1, -1]

D [1, 2, 0, 0]

PHÒNG THEO [USA-NA-AIO 2026 · Vòng 1 · Problem 3.1](#)

Số ngẫu nhiên dùng một bộ sinh có seed, để chạy lại cho đúng kết quả cũ:

Python

```
rng = np.random.default_rng(42)
print(rng.integers(0, 10, size=5), rng.normal(size=2).round(3))
print(rng.permutation(5), rng.choice(["A", "B", "C"], size=4))
rng2 = np.random.default_rng(42)
print(rng2.integers(0, 10, size=5))
```

cùng seed: cùng dãy số

Seed cố định cả một dãy số. Hai bộ sinh cùng seed, gọi theo cùng thứ tự, cho cùng các số. Nhưng mỗi lần gọi trên một rng lại lấy phần tiếp theo của dãy, nên hai lần gọi liên tiếp ra hai kết quả khác nhau. Trong code cũ hay gặp `np.random.seed(...)`; cách này dùng một bộ sinh chung cho cả chương trình, dễ bị đoạn code khác gọi xen vào làm lệch dãy, nên nên dùng `default_rng`.



Seed cố định cả dãy; mỗi lần gọi lấy phần tiếp theo.

HỎI NHANH

```
rng = np.random.default_rng(42)
a = rng.integers(0, 10, size=5)
b = rng.integers(0, 10, size=5)
```

a có bằng b không? Chạy lại cả đoạn thì có ra đúng các số cũ không?

- A a bằng b, và chạy lại ra đúng số cũ
- B a khác b, và chạy lại ra đúng số cũ
- C a bằng b, và chạy lại ra số mới
- D a khác b, và chạy lại ra số mới

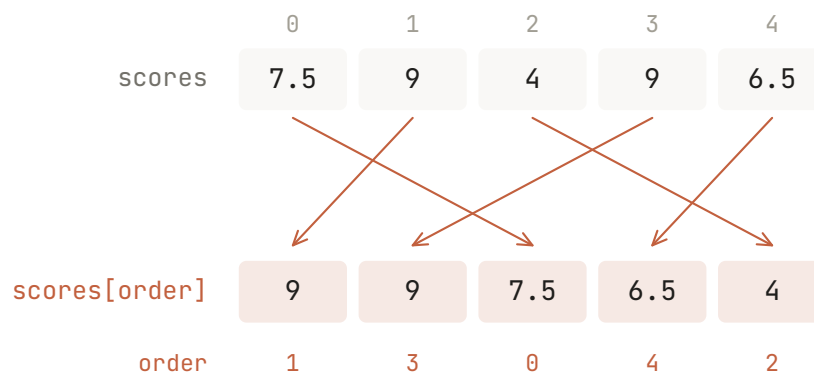
Sắp xếp, đếm, cộng dồn và lưu mảng

NumPy có sẵn các thuật toán quen thuộc từ lập trình thi đấu: sắp xếp, đếm, cộng dồn, tìm nhị phân. Học máy dùng chúng để lấy k lớp có điểm cao nhất, đếm số mẫu của mỗi nhãn, chia điểm thành các khoảng.

Python

```
scores = np.array([7.5, 9.0, 4.0, 9.0, 6.5])
order = np.argsort(-scores, kind="stable")      # chỉ số theo điểm giảm dần
print(order, scores[order])
top2 = np.argpartition(-scores, 2)[:2]         # hai chỉ số điểm cao nhất, không sắp cả
mảng
print(np.sort(top2))
labels = np.array([2, 0, 2, 1, 2, 0])
print(np.unique(labels, return_counts=True))
print(np.bincount(labels, minlength=4))
print(np.cumsum([3, 1, 4, 1, 5]))
edges = np.array([5.0, 6.5, 8.0])              # ngưỡng điểm: dưới 5, 5-6.5, 6.5-8, từ 8
print(np.searchsorted(edges, [4.0, 6.5, 9.0], side="right"))
```

`argsort` trả về **chỉ số**. Sắp `-scores` để được thứ tự giảm dần; `kind="stable"` giữ hai điểm 9.0 theo thứ tự xuất hiện. `argpartition` giống `nth_element` của C++: đưa k chỉ số lớn nhất lên đầu trong thời gian tuyến tính nhưng không sắp chúng.



Đọc scores theo order được dãy giảm dần.

- `unique`: các giá trị khác nhau và số lần xuất hiện của mỗi giá trị.

- `bincount`: ô i là số lần gặp số i ; `minlength=4` để có cả ô của nhãn 3 dù không xuất hiện.
- `cumsum`: mảng cộng dồn (prefix sum).
- `searchsorted`: tìm nhị phân trên mảng đã sắp; `side="right"` giống `upper_bound` của C++. Điểm 6.5 rơi vào khoảng 2, tức từ 6.5 đến dưới 8.

HỎI NHANH

`np.argsort(np.array([30, 10, 20]))` in ra gì?

A [2 0 1]

B [1 2 0]

C [10 20 30]

D [0 1 2]

Đặc trưng hay embedding tính lâu thì lưu lại bằng định dạng của NumPy; lần sau nạp ngay, giữ nguyên shape và dtype:

Python

```
from pathlib import Path
from tempfile import TemporaryDirectory

with TemporaryDirectory() as directory:
    path = Path(directory) / "features.npz"
    np.savez_compressed(path, X=np.arange(6, dtype=np.float32).reshape(2, 3), y=np.array([0, 1]))
    with np.load(path) as data:
        print(data.files, data["X"].shape, data["X"].dtype, data["y"])
```

`np.save` ghi một mảng ra tệp `.npy`; `np.savez_compressed` gom nhiều mảng có tên vào một tệp `.npz`. Lưu bằng CSV thì mất dtype và chậm với mảng lớn. Trên Kaggle, tệp ghi vào `/kaggle/working` được giữ lại sau khi lưu phiên bản notebook.

Khi đi thi

- In shape sau mỗi bước; đặt `assert y_hat.shape == y.shape` trước khi tính loss.
- Ảnh `uint8` thì đổi sang `float32` trước khi cộng trừ.
- Trước khi broadcast trên dữ liệu lớn, tính bộ nhớ của mảng trung gian.

Tóm tắt

- Mảng NumPy có shape và dtype. Số nguyên NumPy có giới hạn và có thể tràn, như `uint8` của ảnh.
- Phép toán, so sánh và hàm toán học áp dụng lên cả mảng, nhanh hơn vòng lặp Python.
- Lát cắt `start:stop` là view; list chỉ số và mặt nạ `bool` cho bản sao.
- Gộp theo axis làm trục đó biến mất; `keepdims=True` giữ nó lại với độ dài 1.
- Broadcasting xếp shape từ phải, kéo dẫn trục độ dài 1. Cẩn thận với $(n,)$ và $(n, 1)$.
- `@` là nhân ma trận, `*` là nhân từng phần tử; số ngẫu nhiên qua `np.random.default_rng(seed)`.

Kiểm tra cuối bài

Mười câu dưới đây bao quát cả bài. Làm hết rồi xem điểm; câu nào sai thì đọc lại mục tương ứng.

CÂU 1

`np.arange(24).reshape(2, 3, -1).shape` là gì?

A (2, 3, 1)

B (2, 3, 24)

C (2, 3, 4)

D Báo lỗi

CÂU 2

Có hai mảng NumPy `pred = [1, 0, 1, 1, 0]` và `y = [1, 1, 1, 0, 0]`. `(pred == y).mean()` bằng bao nhiêu?

A 1

B 0.6

C 0.4

D 3

CÂU 3

Chạy `a = np.arange(10)`, `b = a[a > 5]`, `b[:] = 0`. Mảng `a` thay đổi thế nào?

A Bốn phần tử cuối thành 0

B Mọi phần tử thành 0

C Báo lỗi

D Không đổi

CÂU 4

X có shape (1000, 20): 1000 mẫu, 20 đặc trưng, cột nào cũng có độ lệch chuẩn khác 0. Muốn chuẩn hóa riêng từng đặc trưng, ta viết thế nào?

A `(X - X.mean(axis=1)) / X.std(axis=1)`

B `(X - X.mean(axis=0)) / X.std(axis=0)`

C `(X - X.mean()) / X.std()`

D `X / X.max()`

CÂU 5

imgs có shape $(N, H, W, 3)$. Muốn lấy độ sáng trung bình của từng ảnh, kết quả có shape $(N,)$, thì gộp theo axis nào?

A `axis=0`

B `axis=3`

C `axis=(0, 1, 2)`

D `axis=(1, 2, 3)`

CÂU 6

Cho $A = \text{np.array}([[1, 2], [3, 4]])$ và $B = \text{np.array}([[0, 1], [1, 0]])$. $A * B$ và $A @ B$ lần lượt bằng gì?

A Cả hai đều bằng $[[2, 1], [4, 3]]$

B $[[0, 2], [3, 0]]$ và $[[2, 1], [4, 3]]$

C $[[2, 1], [4, 3]]$ và $[[0, 2], [3, 0]]$

D $[[0, 2], [3, 0]]$ và $[[3, 4], [1, 2]]$

PHÒNG THEO [NOAI 2026 Singapore](#) · Vòng sơ loại · Câu 113

CÂU 7

A có shape $(6, 4)$. Muốn trừ đi trung bình của từng hàng mà kết quả vẫn có shape $(6, 4)$, ta viết thế nào?

A $A - A.\text{mean}(\text{axis}=0, \text{keepdims}=\text{True})$

B $A - A.\text{mean}(\text{axis}=0)$

C $A - A.\text{mean}(\text{axis}=1, \text{keepdims}=\text{True})$

D $A.T - A.\text{mean}(\text{axis}=1)$

CÂU 8

Cho `pred = np.array([[2], [5], [8]])` và `y = np.array([1, 5, 10])`. Cần tính trung bình sai số tuyệt đối (MAE) của ba cặp tương ứng. Dòng nào đúng?

A `np.abs(pred.mean(axis=0) - y).mean()`

B `np.abs(pred - y).mean()`

C `np.abs(pred - y).sum() / len(y)`

D `np.abs(pred.ravel() - y).mean()`

CÂU 9

Với `A = np.array([[2, 1], [0, 3]])`, `A @ A` bằng gì?

A `[[4, 5], [0, 9]]`

B `[[4, 3], [0, 9]]`

C `[[2, 1], [0, 3]]`

D `[[4, 1], [0, 9]]`

CÂU 10

A và B đều có shape (20 000, 128), kiểu `float64`. Tính `A[:, None, :] - B[None, :, :]` thì mảng trung gian tốn khoảng bao nhiêu bộ nhớ?

A 400 GB

B 20 MB

C 400 MB

D 4 TB

Đọc thêm

- NumPy — [NumPy: the absolute basics for beginners](#). Hướng dẫn chính thức, có hình minh họa cho shape, axis và lát cắt.
- NumPy — [Broadcasting](#). Quy tắc broadcasting đầy đủ, kèm hình các mảng được kéo dẫn.
- VanderPlas — [Python Data Science Handbook](#), chương 2. Sách miễn phí, giải thích vì sao NumPy nhanh và các kỹ thuật vector hóa.
- Rougier — [From Python to NumPy](#). Sách miễn phí về cách nghĩ vector hóa, có phần view và copy rất kỹ.
- Stanford CS231n — [Python Numpy Tutorial](#). Bản tóm tắt của khóa thị giác máy tính Stanford; ôn phần Python và NumPy dùng ở các chương sau.
- Rougier — [100 NumPy exercises](#). Một trăm bài tập ngắn từ dễ đến khó để luyện vector hóa.
- NumPy — [Copies and views](#), [Data type objects](#) và [numpy.vectorize](#). Đọc để hiểu vùng nhớ dùng chung, dtype và vì sao tên “vectorize” không bảo đảm tăng tốc.