

Tuần 1 - Tổng hợp kiến thức Buổi học số 1 và 2

Time-Series Team

Ngày 15 tháng 7 năm 2025

Buổi học số 1 (Thứ 3, 1/07/2025) và buổi học số 2 (Thứ 4, 2/07/2025) có nhiều nội dung tương đồng và kế thừa nhau nên nhóm mình sẽ tổng hợp lại thành 8 nội dung chính:

- *Phần 1: Motivation: Tại sao cần NumPy?*
- *Phần 2: Giới thiệu NumPy*
- *Phần 3: CRUD trên Array*
- *Phần 4: Array Transformations*
- *Phần 5: View vs Copy – Hiểu đúng để tránh bug*
- *Phần 6: Vectorization & Broadcasting*
- *Phần 7: AI Applications*
- *Phần 8: Mở rộng*

Phần 1: Motivation: Tại sao cần NumPy?

1. Đặt vấn đề: List có đủ tốt?

Trong Module 1, khi bắt đầu học Python, chúng ta đã làm quen với cấu trúc dữ liệu **List** – một trong những kiểu dữ liệu cơ bản và linh hoạt nhất. Với khả năng chứa bất kỳ kiểu phần tử nào (số, chuỗi, danh sách lồng...), cùng các thao tác thêm, xóa, duyệt, lọc khá dễ dàng, **List đặc biệt hữu ích cho các bài toán xử lý dữ liệu đơn giản, quy mô nhỏ.**

Ví dụ: xử lý chuỗi, duyệt danh sách điểm thi, lọc dữ liệu từ file, hay thậm chí là thao tác sơ bộ với dữ liệu từ csv – tất cả đều có thể xử lý gọn gàng bằng List:

```
1 scores = [7, 8.5, 9, 6.5, 10]
2 # Filter scores >= 8
3 high_scores = [s for s in scores if s >= 8]
4 print(high_scores) # [8.5, 9, 10]
```

Tuy nhiên, khi xét một bài toán khác:

Ta có một dãy số $[1, 2, 3, 4]$ và muốn tính kết quả $\text{output} = x^2 + 1$ cho từng phần tử. **Với List**, cách viết thường thấy sẽ là:

```
1 x = [1, 2, 3, 4]
2 y = [xi**2 + 1 for xi in x]
3 print(y) # [2, 5, 10, 17]
```

Cách viết này tuy vẫn gọn, nhưng:

- Vẫn cần **duyệt từng phần tử thủ công** (list comprehension).
- Khi mở rộng lên 2D/3D, hoặc hàng triệu phần tử – **hiệu suất giảm mạnh**.

List bắt đầu bộc lộ giới hạn khi nào?

Khi bài toán bắt đầu bước vào vùng:

- **Tính toán số học quy mô lớn** (hàng triệu phần tử).
- **Mảng nhiều chiều** (ảnh, video, dữ liệu tensor).
- **Cần hiệu suất cao** (AI, khoa học dữ liệu, thống kê).

Lúc này, List nhanh chóng bộc lộ các nhược điểm nghiêm trọng:

- **Không hỗ trợ vector hóa (no vectorization)**
Giả sử muốn tính toán: $\text{output} = \text{input}^2 + 1$ với $\text{input} = [1, 2, 3, 4]$.

Nếu chúng ta muốn thử:

```
1 input = [1, 2, 3, 4]
2 output = input**2 + 1 # Error: unsupported operand
```

Python sẽ báo lỗi vì List **không hỗ trợ toán học dạng vector**.

Mà với list thuần, ta phải dùng **List comprehension** để xử lý từng phần tử:

```
1 input = [1, 2, 3, 4]
2 output = [x**2 + 1 for x in input] # use for => loop
```

⇒ Không ngắn gọn, khó mở rộng, và **không tận dụng được khả năng tối ưu phần cứng**.

- **Dữ liệu không đồng nhất (heterogeneous types)**

Python List cho phép chứa mọi loại dữ liệu:



Điều này đồng nghĩa:

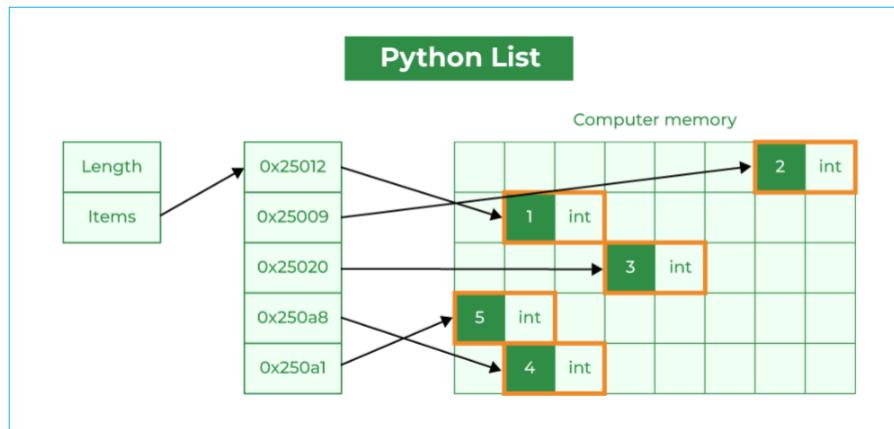
- Không thể tối ưu truy xuất dữ liệu.
- Mỗi phần tử là một object độc lập ⇒ **tốn bộ nhớ hơn rất nhiều**.

- **Bộ nhớ phân mảnh (non-contiguous memory)**

Các phần tử trong List được lưu ở các vị trí rời rạc trong bộ nhớ:

- Một int có thể nằm ở địa chỉ A.
- Một float tiếp theo ở địa chỉ B hoàn toàn khác.

Khi xử lý hàng triệu phần tử, việc **bộ nhớ không liên tục** khiến **CPU cache không thể tối ưu hóa** ⇒ tốc độ chậm.



Hình 1: Minh họa sự phân mảnh trong bộ nhớ của 1 List gồm các phần tử kiểu int

- **Không có reshape, broadcasting**

- Không thể dễ dàng thay đổi chiều dữ liệu hay thực hiện phép toán giữa mảng khác shape.

Và đó là lúc chúng ta cần đến **NumPy** – một thư viện giúp Python **chuyển mình từ ngôn ngữ “general purpose” thành công cụ mạnh cho khoa học dữ liệu và AI**.

2. NumPy Array: Giải pháp tối ưu

NumPy ra đời để khắc phục các hạn chế trên bằng cách cung cấp **mảng đa chiều hiệu suất cao – ndarray**. Dưới đây là vài ưu điểm then chốt:

- **Dữ liệu đồng nhất (homogeneous)**

NumPy yêu cầu tất cả phần tử phải có cùng kiểu dữ liệu (`int32`, `float64`, ...).
⇒ Dễ tối ưu hóa và thao tác toán học.

- **Bộ nhớ liên tục (contiguous)**

Dữ liệu được lưu liên tiếp trong vùng nhớ. Điều này giúp:

- Tận dụng tốt CPU cache.
- Tốc độ tính toán nhanh hơn nhiều lần so với List.

- **Vector hóa và broadcasting**

Các phép toán hoạt động trực tiếp trên toàn bộ array – không cần vòng lặp, cụ thể:
Cụ thể với NumPy, ví dụ trên trở nên gọn và mạnh mẽ hơn rất nhiều:

```
1 import numpy as np
2
3 x = np.array([1, 2, 3, 4])
4 y = x**2 + 1
5 print(y) # [ 2  5 10 17 ]
```

Điều kỳ diệu là:

- Cú pháp ngắn hơn.
- Nhờ tận dụng cấu trúc bộ nhớ liên tục như C và gọi các thư viện tính toán cấp thấp, giúp thao tác trên array được vector hóa và **chạy nhanh gấp nhiều lần** so với List thuần.
- Có khả năng mở rộng tốt cho array 2D, 3D, n-D
⇒ Cực kỳ hữu dụng cho nhiều tác vụ trong thực tế như: AI, xử lý ảnh, thống kê, vật lý, kỹ thuật...

3. “Implement View” vs “Execution View”

Trở lại vấn đề quenn thuộc: Cho $L = [1, 2, 3, 4]$, yêu cầu tính $L^2 + 1$.

Với List:

```
1 result = [x**2 + 1 for x in L]
```

⇒ Thao tác từng phần tử – “*Implement View*”: phải viết cách làm cụ thể.

Với NumPy:

```
1 arr = np.array([1, 2, 3, 4])
2 result = arr**2 + 1
```

⇒ Thao tác toàn mảng – “*Execution View*”: chỉ cần nói “muốn gì” – NumPy tự xử lý.

4. Benchmark: So sánh hiệu suất giữa list và ndarray

Benchmark là gì?

Benchmark (tạm dịch: “kiểm chuẩn tốc độ”) là quá trình đo và so sánh hiệu suất thực thi của các đoạn chương trình, thường dùng để:

- So sánh 2 hoặc nhiều cách giải cùng một bài toán.
- Đánh giá chi phí về thời gian, bộ nhớ hoặc hiệu suất.
- Chọn phương pháp tối ưu cho bài toán thực tế.

Trong ngữ cảnh này, ta sẽ benchmark tốc độ giữa:

- Python `list` – với vòng lặp (list comprehension).
- NumPy `ndarray` – với phép toán vector hóa.

Bài toán benchmark

Ta sẽ thực hiện bài toán đơn giản: tính bình phương của một triệu số nguyên từ 0 đến 999 999. Tuy đơn giản, nhưng bài toán này rất phù hợp để thấy sự khác biệt lớn về hiệu suất khi dữ liệu đủ lớn.

```

1 import numpy as np
2 import time
3
4 # Tạo một danh sách hàng
5 L = list(range(1_000_000))      # List
6 A = np.array(L)                 # ndarray NumPy
7
8 # Benchmark với List
9 start = time.time()
10 L_result = [x**2 for x in L]
11 list_time = time.time() - start
12 print("List time:", list_time)
13
14 # Benchmark với NumPy
15 start = time.time()
16 A_result = A**2
17 numpy_time = time.time() - start
18 print("NumPy time:", numpy_time)
19
20 # Kết quả
21 print("Speedup:", list_time / numpy_time)

```

===== Output =====

```

List time: 0.3032 s
NumPy time: 0.0155 s
Speedup: 19.55x

```

Phân tích kết quả

- Thời gian xử lý với `list` là hơn 0.3 giây — do phải lặp qua từng phần tử và tính toán từng `x**2` trong Python.
- Với NumPy, chỉ mất khoảng 0.015 giây — nhanh hơn **gần 20 lần**.

- Nguyên nhân chính:
 - ndarray lưu trữ dữ liệu **liên tục trong bộ nhớ**, giúp CPU cache xử lý hiệu quả hơn nhiều.
 - Các phép toán như **A**2** được gọi trực tiếp vào các hàm viết bằng **C/Fortran** tối ưu hóa (tận dụng SIMD, vectorization, multi-thread).
 - Không cần vòng lặp Python (rất chậm so với vòng lặp cấp thấp).

Lưu ý:

- Với bài toán lớn (tính toán ma trận, xử lý ảnh, ML), mức chênh lệch có thể lên tới **hàng trăm lần**.
- Benchmark là minh chứng rõ ràng cho lý do vì sao NumPy trở thành thư viện nền tảng cho khoa học dữ liệu, ML, AI, xử lý ảnh/video...

5. Khi nào nên dùng NumPy?

Bảng dưới đây so sánh giữa Python list và NumPy ndarray trên nhiều khía cạnh quan trọng:

Tính năng	Python list	NumPy ndarray
Kiểu dữ liệu	Linh hoạt (không đồng nhất)	Đồng nhất (fixed dtype)
Cấu trúc bộ nhớ	Phân mảnh (non-contiguous)	Liên tục (contiguous)
Toán học vector	Không hỗ trợ trực tiếp	Hỗ trợ mạnh mẽ
Vectorization	Không có	Có
Broadcasting	Không có	Có
Tốc độ thực thi	Thấp	Rất cao
Khả năng tối ưu phần cứng	Không có	Có (C-level, đa luồng)
Thao tác reshape, indexing nâng cao	Rất hạn chế	Đầy đủ

Kết luận:

- Khi làm việc với dữ liệu khoa học, xử lý ảnh, âm thanh, video, dữ liệu lớn hoặc huấn luyện mô hình AI/ML, thì **NumPy là sự lựa chọn bắt buộc**.
- ndarray không chỉ giúp code ngắn gọn, dễ đọc hơn mà còn tăng hiệu suất lên hàng chục lần.
- Nhiều thư viện quan trọng như Pandas, scikit-learn, TensorFlow đều dựa trên nền tảng NumPy.

Phần 2: Giới thiệu NumPy

1. Làm quen với ndarray

Từ những phân tích ở phần trước, ta thấy rõ rằng Python `list` không phù hợp với các bài toán cần hiệu suất cao, vector hóa hay xử lý mảng nhiều chiều. NumPy cung cấp kiểu dữ liệu `ndarray` để giải quyết vấn đề này.

Tạo ndarray từ list Python:

```

1 import numpy as np
2
3 L = [1, 2, 3, 4, 5]
4 A = np.array(L)
5 print(type(L), type(A))
6 # <class 'list'> <class 'numpy.ndarray'>

```

Sau khi khởi tạo, `A` trở thành một mảng có thể tham gia các phép toán như `A + 1`, `A ** 2` một cách tự nhiên.

So sánh nhanh giữa list và ndarray:

	Python list	NumPy ndarray
Kiểu phần tử	Hỗn hợp	Đồng nhất (dtype)
Toán học mảng	Không hỗ trợ trực tiếp	Vectorized (<code>A + 1</code> , <code>A**2</code>)
Bộ nhớ	Phân mảnh	Liên tục (contiguous)
Hiệu suất	Thấp	Cao (C-level, SIMD)

Ghi chú: Với `list`, mỗi phần tử là một object độc lập trong bộ nhớ — Python phải lưu địa chỉ từng phần tử rời rạc. Còn với `ndarray`, toàn bộ dữ liệu được lưu liên tiếp trong RAM dưới cùng một kiểu dữ liệu (`int32`, `float64`, ...), giúp tận dụng CPU cache và thư viện C phía sau NumPy để xử lý cực nhanh.

Việc chuyển đổi sang `ndarray` không chỉ là “ép kiểu” mà là bước mở cửa vào thế giới của vectorization, broadcasting, và tính toán số học ở quy mô lớn.

⇒ Từ đây, ta sẽ tiếp tục khám phá cách NumPy mở rộng khái niệm vector một chiều thành các **mảng đa chiều** có cấu trúc linh hoạt hơn rất nhiều.

2. Từ vector đến mảng đa chiều (ndarray)

Trong Toán học, **vector** là một dãy số có thứ tự $\mathbf{v} = (v_0, v_1, \dots, v_{n-1})$ — tức là một chiều dữ liệu. Khi xử lý dữ liệu trong thực tế (ảnh, video, tín hiệu), chúng ta thường gặp không chỉ vector, mà là các cấu trúc **đa chiều** hơn rất nhiều.

NumPy mở rộng khái niệm vector thành `ndarray` — một kiểu dữ liệu có thể biểu diễn **mọi chiều không gian**:

- **1D** (vector): v_i
- **2D** (ma trận): $A_{i,j}$
- **3D** (tensor): $B_{i,j,k}$
- **4D+** (video, batch, mô hình vật lý): $T_{b,c,h,w}$

Minh họa các chiều mảng trong NumPy:

```

1 import numpy as np
2
3 v = np.array([1,2,3])
4 print(f"v.ndim D: shape = {v.shape}")
5 M = np.array([[1, 2, 3], [4, 5, 6]])
6 print(f"M.ndim D: shape = {M.shape}")
7 T = np.zeros((3,3,2))
8 print(f"T.ndim D: shape = {T.shape}")

```

===== Output =====

```

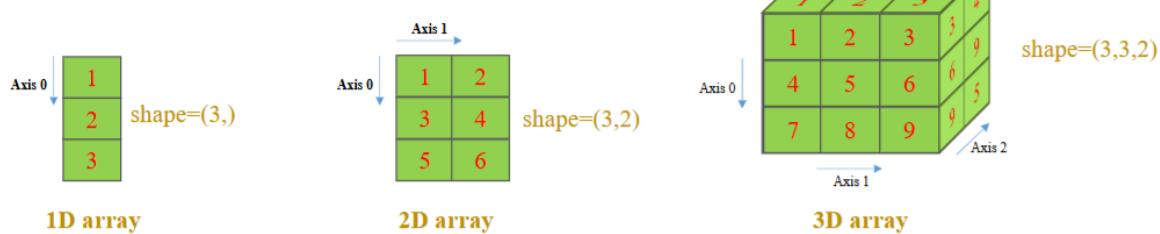
1D: shape = (3,)
2D: shape = (2, 3)
3D: shape = (3, 3, 2)

```

Mỗi chiều trong mảng được gọi là một **trục** (axis). Ví dụ, với mảng có shape (2, 3, 4):

- Trục 0 (axis 0): có 2 phần tử — tương ứng 2 “khối”.
- Trục 1 (axis 1): mỗi khối có 3 hàng.
- Trục 2 (axis 2): mỗi hàng có 4 cột.

❖ Numpy arrays are multi-dimensional arrays



Hình 2: Minh họa kích thước và chiều axis

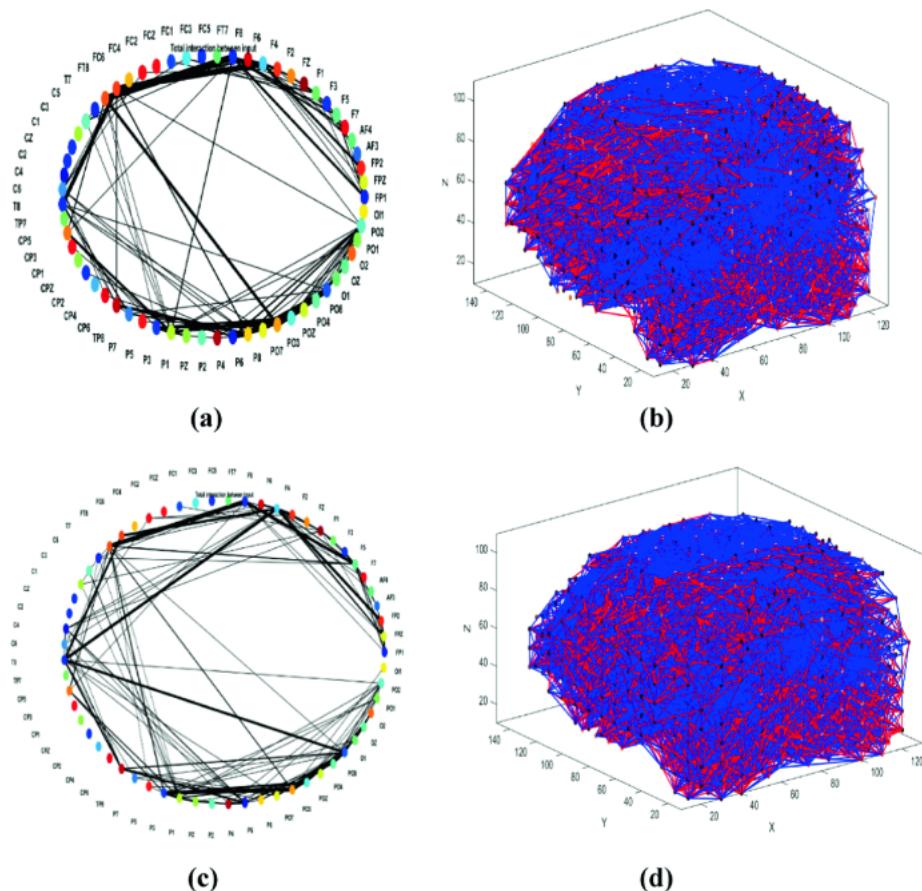
Một số ví dụ thực tế:

- Ảnh xám (2D): kích thước (H, W)
- Ảnh RGB (3D): kích thước ($3, H, W$) hoặc ($H, W, 3$) tùy thư viện
- Video (4D): ($T, 3, H, W$) — số frame, kênh màu, chiều cao, chiều rộng
- Dữ liệu batch (5D): (B, T, C, H, W) — batch size, thời gian, kênh, hình

Một số lĩnh vực ứng dụng theo chiều dữ liệu:

- **1D:** chuỗi thời gian, âm thanh, văn bản token hóa.

- **2D:** ảnh xám, dữ liệu dạng bảng, ma trận.
- **3D:** ảnh màu RGB, dữ liệu ảnh y tế (CT/MRI).
- **4D trờ lên:** video, batch training trong deep learning, mô hình mô phỏng vật lý (fluid, khí động lực...).



Hình 3: Mô tả mạng kết nối não trong không gian 2D và 3D

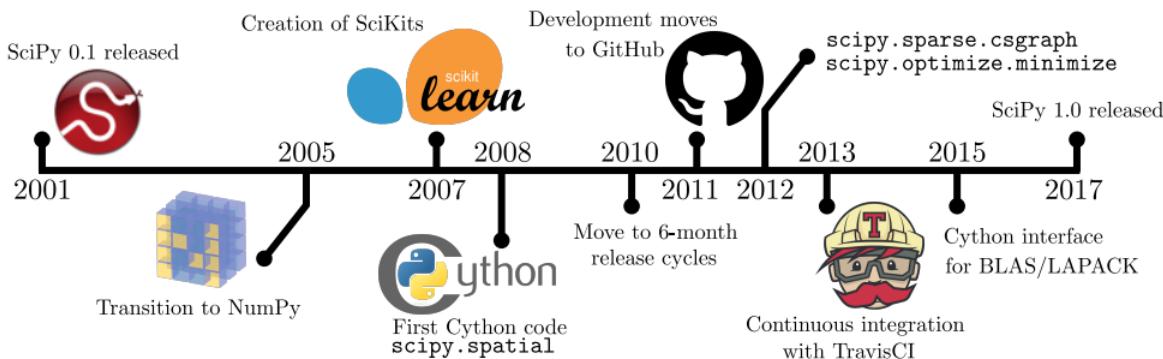
Mảng nhiều chiều chính là trung tâm của mọi tính toán khoa học bằng NumPy. Hiểu rõ cách NumPy tổ chức và xử lý ndarray là nền tảng để bước vào vectorization, broadcasting, hoặc deep learning sau này.

3. NumPy là gì? Lịch sử và vai trò

NumPy (viết tắt của “Numerical Python”) là một thư viện mã nguồn mở nổi bật trong hệ sinh thái Python, cung cấp một kiểu dữ liệu mảng đa chiều hiệu suất cao gọi là **ndarray**, cùng với hàng trăm hàm toán học có thể hoạt động trực tiếp trên toàn mảng.

Lịch sử ra đời của NumPy:

- Trước năm 2005, cộng đồng Python khoa học bị chia rẽ giữa hai thư viện: Numeric (ra đời năm 1995, nhanh nhưng hạn chế) và numarray (linh hoạt hơn nhưng chậm).
- Năm 2005, Travis Oliphant — một giáo sư ngành điện tử và kỹ thuật y sinh — đã đứng ra hợp nhất hai thư viện này thành NumPy, bằng cách mở rộng Numeric và tích hợp ý tưởng từ numarray.
- Việc chuyển đổi toàn bộ hệ sinh thái sang NumPy bắt đầu từ năm 2005, và NumPy 1.0 chính thức được phát hành vào năm 2006.
- SciPy — một thư viện cấp cao hơn ra đời từ 2001 — ban đầu được xây dựng trên Numeric, sau đó chuyển sang dùng NumPy làm nền tảng chính.
- Từ năm 2010 trở đi, hệ sinh thái SciPy/NumPy ngày càng ổn định: chuyển lên GitHub (2011), tích hợp TravisCI (2013), và đến năm 2017 phát hành SciPy 1.0.
- Cả NumPy và SciPy đều là phần cốt lõi của hệ sinh thái Python khoa học, được phát triển cộng đồng và bảo trợ bởi tổ chức phi lợi nhuận NumFOCUS.



Hình 4: Timeline phát triển của thư viện SciPy, thể hiện rõ sự phụ thuộc vào NumPy

Vai trò cốt lõi của NumPy:

- Cung cấp kiểu dữ liệu `ndarray` — mảng đa chiều đồng nhất, hỗ trợ slice, reshape, broadcasting...
- Tích hợp hàng trăm hàm toán học vector hóa (vectorized operations), tận dụng:
 - C-level speed (viết bằng C, Fortran)
 - SIMD instructions (tăng tốc qua phần cứng)
 - Hỗ trợ đa luồng (multi-threaded operations)
- Là nền tảng cho các thư viện nổi tiếng như:
 - Pandas – xử lý dữ liệu dạng bảng (DataFrame)
 - SciPy – thuật toán khoa học (linear algebra, signal...)
 - scikit-learn – học máy cổ điển

- TensorFlow, PyTorch – cảm hứng từ cú pháp và broadcasting của NumPy

So sánh triết lý thiết kế:

	Python gốc (List)	NumPy (ndarray)
Tối ưu cho	Tính linh hoạt, dễ dùng	Tính toán hiệu suất cao
Kiểu dữ liệu	Đa kiểu (heterogeneous)	Đồng nhất (homogeneous)
Tính toán	Duyệt thủ công, vòng lặp	Vector hóa toàn mảng
Backend thực thi	Thuần Python	C/Fortran + SIMD + cache
Broadcasting	Không hỗ trợ	Có
Reshape/Index nâng cao	Hạn chế	Đầy đủ

NumPy không thay thế Python, mà là phần mở rộng giúp Python trở thành một ngôn ngữ mạnh mẽ cho khoa học dữ liệu, thống kê, AI, xử lý tín hiệu và mô phỏng vật lý.

Từ các phép toán đơn giản như cộng mảng, đến các bài toán deep learning phức tạp — mọi thứ đều khởi đầu từ một ndarray.

4. Định nghĩa ndarray và các thuộc tính cơ bản

Trong NumPy, mọi mảng đều thuộc kiểu dữ liệu `numpy.ndarray` — một cấu trúc chuyên biệt để lưu trữ dữ liệu số học theo dạng mảng nhiều chiều (vector, ma trận, tensor...).

Đặc điểm chính của ndarray:

- **Đồng nhất (homogeneous):** tất cả phần tử có cùng kiểu dữ liệu (`dtype`), ví dụ `int32`, `float64`...
- **Liên tục trong bộ nhớ (contiguous memory):** các phần tử được lưu liền mạch, giúp truy xuất nhanh hơn và dễ tối ưu hóa bằng C/Fortran.
- **Hỗ trợ đa chiều (multidimensional):** có thể là 1D, 2D, 3D, thậm chí nD.

Một số thuộc tính cơ bản:

- `arr.shape` – kích thước từng chiều (tuple).
- `arr.ndim` – số chiều (int).
- `arr.dtype` – kiểu dữ liệu phần tử.
- `arr.itemsize` – số byte mỗi phần tử.
- `arr.size` – tổng số phần tử trong array.
- `arr.nbytes` – tổng dung lượng bộ nhớ = `itemsize × size`.

Ví dụ: mảng 3 chiều

```

1 import numpy as np
2
3 X = np.zeros((3, 3, 2))
4 print("shape   :", X.shape)
5 print("ndim   :", X.ndim)
6 print("dtype   :", X.dtype)
7 print("itemsize :", X.itemsize)
8 print("size    :", X.size)
9 print("nbytes  :", X.nbytes)

```

```

=====
Output =====
shape      : (3, 3, 2)
ndim       : 3
dtype      : float64
itemsize   : 8
size       : 18
 nbytes    : 144

```

Ví dụ trực quan hơn với mảng 2D:

```

1 A = np.array([[1, 2, 3],
2              [4, 5, 6]])
3 print("shape   :", A.shape)
4 print("ndim   :", A.ndim)
5 print("size    :", A.size)
6 print("dtype   :", A.dtype)

```

```

=====
Output =====
shape      : (2, 3)
ndim       : 2
size       : 6
dtype      : int64

```

Ghi chú:

- list của Python không có bất kỳ thuộc tính nào ở trên, do đó ta không thể gọi `shape`, `dtype`, hay `size` trên list.
- Vì vậy, khi xử lý dữ liệu số học, đặc biệt với mảng nhiều chiều hoặc kích thước lớn, dùng `ndarray` là bắt buộc nếu muốn tối ưu hiệu suất.

Tổng kết: bảng thuộc tính của `ndarray`

Thuộc tính	Kiểu dữ liệu	Ý nghĩa
<code>.shape</code>	<code>tuple</code>	Kích thước từng chiều
<code>.ndim</code>	<code>int</code>	Số chiều
<code>.dtype</code>	<code>dtype</code>	Kiểu dữ liệu phần tử
<code>.itemsize</code>	<code>int</code>	Byte trên mỗi phần tử
<code>.size</code>	<code>int</code>	Tổng số phần tử
<code>.nbytes</code>	<code>int</code>	Tổng số byte trong mảng

5. Cài đặt và import

```

1 # pip
2 pip install numpy
3
4 # conda
5 conda install numpy

```

```

1 import numpy as np

```

Alias `np` là chuẩn quốc tế để gọi mọi hàm và lớp trong NumPy.

Phần 3: CRUD trên Array

1. Create – Khởi tạo ndarray trong NumPy

Khi làm việc với dữ liệu số học, ta thường cần khởi tạo các mảng số (arrays) ban đầu để thao tác, xử lý hoặc tính toán. NumPy cung cấp một tập hợp phong phú các hàm khởi tạo ndarray với cú pháp đơn giản nhưng mạnh mẽ. Việc này giúp loại bỏ vòng lặp tay, tăng tốc độ và tăng tính rõ ràng cho code.

1.1. Chuyển đổi từ list sang NumPy array

```

1 import numpy as np
2
3 lst = [1, 2, 3, 4]
4 arr = np.array(lst) # ndarray 1D

```

`np.array(...)` là cách đơn giản nhất để chuyển Python list hoặc tuple thành NumPy array.

1.2. Một số hàm tạo mảng phổ biến

Các hàm khởi tạo cơ bản:

- `np.zeros(shape)` – tạo mảng toàn số 0.
- `np.ones(shape)` – tạo mảng toàn số 1.
- `np.full(shape, value)` – tạo mảng với tất cả phần tử bằng `value`.
- `np.empty(shape)` – cấp phát mảng nhưng không khởi tạo giá trị (có thể chứa "rác").
- `np.eye(N)` – ma trận đơn vị (1 trên đường chéo).
- `np.identity(N)` – tương tự `eye`, dùng cho square matrix.

Code minh họa:

```

1 import numpy as np
2
3 arr1 = np.zeros((2, 3))
4 print("np.zeros((2, 3)): \n", arr1)
5 arr2 = np.ones((3,))
6 print("np.ones((3,)): \n", arr2)
7 arr3 = np.full((2, 2), 7)
8 print("np.full((2, 2), 7): \n", arr3)
9 arr4 = np.empty((2, 2))
10 print("np.empty((2, 2)): \n", arr4)
11 arr5 = np.eye(3) # 3x3 identity matrix
12 print("np.eye(3): \n", arr5)

=====
Output =====
np.zeros((2, 3)):
[[0. 0. 0.]
 [0. 0. 0.]]
np.ones((3,)): [1. 1. 1.]
np.full((2, 2), 7): [[7 7]
 [7 7]]
np.empty((2, 2)):
[[3.5e-323 3.5e-323]
 [3.5e-323 3.5e-323]]
np.eye(3):
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]

```

1.3. Sinh dãy số và ứng dụng trong làm mượt đồ thị

1.3.1. Đặt vấn đề – `range()` và giới hạn của List

Đầu tiên, chúng ta hãy xét bài toán quen thuộc:

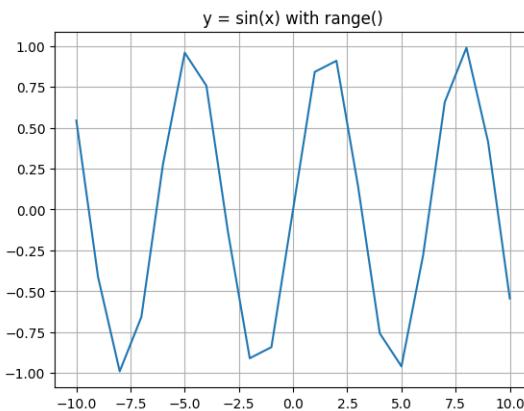
Bài toán: Vẽ đồ thị hàm số $y = \sin(x)$ trên đoạn $[-10, 10]$

Cách đơn giản nhất trong Python là dùng `range()` để sinh dãy x:

```

1 import matplotlib.pyplot as plt
2 import math as m    # use math.sin because range() generates integers
3
4 x = range(-10, 11, 1)      # Step = 1 (integer)
5 y = [m.sin(i) for i in x]
6
7 plt.plot(x, y)
8 plt.title("y = sin(x) with range()")
9 plt.grid(True)
10 plt.show()

```



Kết quả: Đồ thị bị "gãy khúc", rời rạc, không mượt. Đường cong dạng sóng $\sin(x)$ gần như biến thành các đoạn thẳng zigzag, gây hiểu nhầm về bản chất hàm.

Lý do: `range()` chỉ sinh được các số nguyên.

Chúng ta không thể làm mịn đồ thị bằng cách giảm bước xuống 0.1:

```
1 range(-10, 10, 0.1)  # TypeError!
```

⇒ Đây là một trong những **hạn chế lớn của list + range()** khi xử lý dữ liệu số học và trực quan hóa.

Trong các bài toán thực tế, chúng ta rất cần một công cụ cho phép tạo ra dãy số **với bước nhỏ bất kỳ** (kể cả số thực) — để làm mượt đồ thị, phân tích sâu hơn. ⇒ Đó chính là lúc cần đến `np.arange()`

1.3.2. Giới thiệu `np.arange()`

Như đã thấy, `range()` kết hợp với list không thể tạo ra các dãy số thực — điều này khiến việc trực quan hóa hàm liên tục trở nên thô cứng. ⇒ Chúng ta cần một công cụ mạnh hơn, đó là `np.arange()`.

np.arange() là gì

Là một hàm dùng để tạo một mảng chứa các số nguyên liên tiếp trong một khoảng xác định

Cú pháp

```
np.arange(start, stop, step)
```

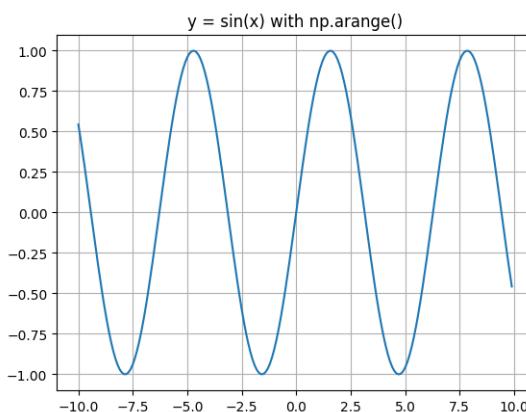
Trong đó:

- start: Giá trị bắt đầu của dãy số (giá trị này bao gồm trong mảng được tạo).
- stop: Giá trị kết thúc của dãy số (giá trị này không bao gồm trong mảng được tạo)
- step: Khoảng cách giữa các giá trị trong dãy số (mặc định là 1).

⇒ Tạo dãy số cách đều từ **start** đến nhỏ hơn **stop**, cách nhau **step** (có thể là số thực).

Ví dụ: Áp dụng lại bài toán ban đầu, nhưng lần này dùng NumPy:

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 x = np.arange(-10, 10, 0.1) # small step → 200 points
5 y = np.sin(x)               # This time use np.sin because x is an array
6
7 plt.plot(x, y)
8 plt.title("y = sin(x) with np.arange()")
9 plt.grid(True)
10 plt.show()
```



Kết quả: Đồ thị giờ đây trơn tru, mượt mà, thể hiện đúng hình dạng sóng $\sin(x)$ — không còn zigzag hay gãy khúc như trước.

Vì sao? Vì ta đang dùng **ndarray** thay vì **list**, và đặc biệt:

- Có thể chọn bước nhỏ (**step = 0.1, 0.01, 0.001** tùy ý)
- Tận dụng được khả năng vector hóa của NumPy

Ghi nhớ

`np.arange()` không chỉ là bản mở rộng của `range()` — nó là một công cụ được thiết kế đặc biệt cho tính toán số học, xử lý dữ liệu hiệu suất cao, và trực quan hóa chuyên sâu.

Trong phần tiếp theo, ta sẽ phân tích kỹ hơn: **Tại sao việc làm mượt đồ thị lại quan trọng ?**

1.3.3. Ứng dụng: Vì sao phải làm mượt đồ thị?

Sau khi đã có công cụ mạnh như `np.arange()`, câu hỏi tiếp theo là:

Làm mượt đồ thị có thật sự cần thiết, hay chỉ để... nhìn cho đẹp?

Câu trả lời là: **Không chỉ để đẹp** — mà là để:

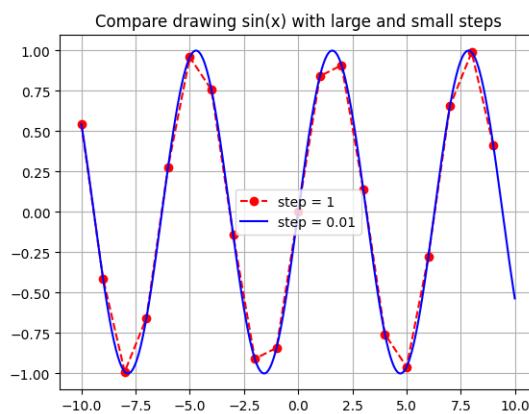
- Thấy được bản chất liên tục của hàm số hoặc dữ liệu.
- Tránh hiểu nhầm do lấy mẫu thưa (under-sampling).
- Phân tích, kiểm chứng mô hình AI chính xác hơn.
- Truyền đạt ý tưởng một cách rõ ràng và thuyết phục.

Ví dụ:

```

1 x1 = np.arange(-10, 10, 1)      # 21 points
2 x2 = np.arange(-10, 10, 0.01)    # 2000 points
3 y1 = np.sin(x1)
4 y2 = np.sin(x2)
5
6 plt.plot(x1, y1, 'ro--', label='step = 1')
7 plt.plot(x2, y2, 'b-', label='step = 0.01')
8 plt.title("Compare drawing sin(x) with large and small steps")
9 plt.legend()
10 plt.grid(True)
11 plt.show()

```



Quan sát: Khi bước là 1, sóng sin trông như răng cưa — gần như mất tính liên tục. Chỉ khi sampling dày (step nhỏ), đồ thị mới "trở về đúng bản chất".

Từ gốc nhìn học thuật: Sampling density và Định lý Nyquist

Theo định lý Nyquist–Shannon trong xử lý tín hiệu:

Định lý Nyquist

Một hàm số liên tục (band-limited signal) chỉ có thể được tái tạo chính xác nếu được lấy mẫu với tần số ít nhất gấp đôi tần số cao nhất của nó.

⇒ Nếu lấy quá ít điểm (sampling thưa), tín hiệu sẽ bị **aliasing** — tức là nhầm sai bản chất. Điều này cũng xảy ra khi chúng ta "plot sai" hàm số.

Ví dụ:

- Hàm sin có tính tuần hoàn — nếu vẽ thiếu điểm, ta không thấy sóng, chỉ thấy zigzag.
- Trong học máy, loss curve vẽ thưa có thể làm ta nghĩ mô hình dao động, trong khi thực ra nó đang học tốt.

Trong AI/ML, đồ thị mượt = hiểu sâu mô hình

Làm mượt đồ thị không chỉ giúp trực quan hóa đẹp hơn — mà còn đóng vai trò then chốt trong việc:

- Phân tích hành vi học của mô hình.
- Phát hiện lỗi mô hình (bug, overfitting, underfitting).
- Đưa ra quyết định như tuning, early stopping, chọn threshold, v.v.

Dưới đây là các tình huống cụ thể mà một đồ thị mượt có thể thay đổi toàn bộ cách chúng ta đánh giá mô hình:

Ví dụ 1: Vẽ Loss Curve

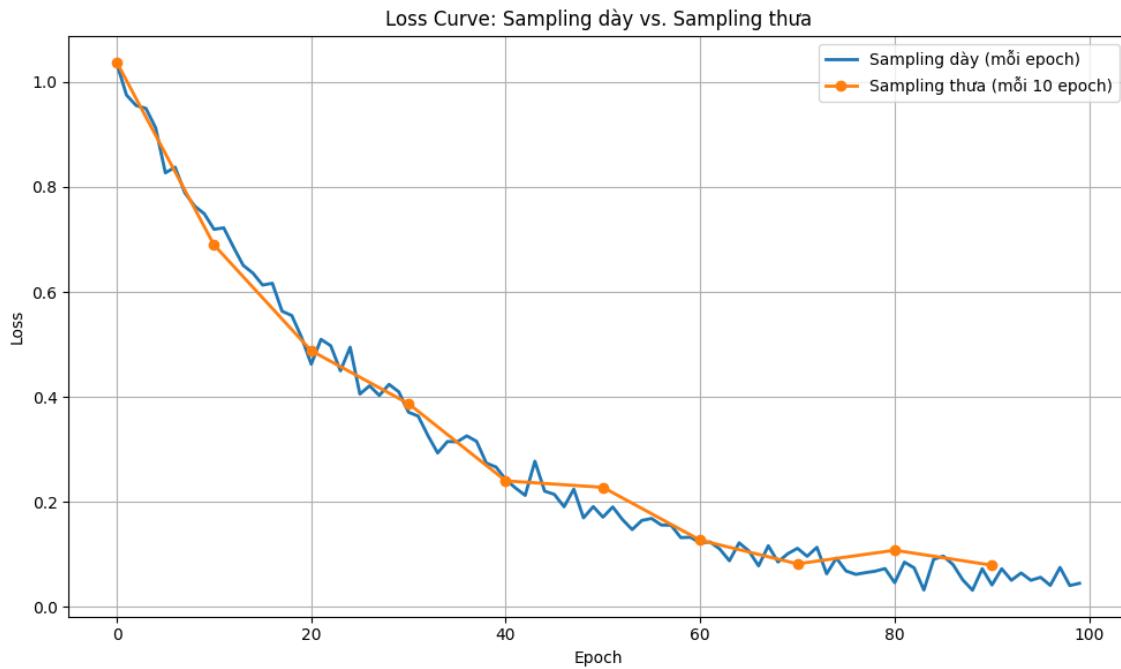
Giả sử chúng ta đang train một mô hình neural network. Nếu chỉ vẽ loss mỗi 10 epoch, sampling thưa:

```
1 epochs = range(0, 100, 10)
```

→ Đường loss trông như đang nhảy loạn. Ta có thể tưởng là mô hình đang học tệ và dừng sớm. Nhưng nếu vẽ từng epoch, sampling dày hơn:

```
1 epochs = range(0, 100, 1)
```

→ Ta sẽ thấy loss giảm đều đặn → mô hình đang học rất ổn.



Loss Curve: Sampling thưa vs. Sampling dày.

Nhận xét:

- Sampling thưa (mỗi 10 epoch) khiến đường loss bị giật cục, có thể hiểu sai rằng mô hình học không ổn định.
- Sampling dày (mỗi epoch) thể hiện rõ mô hình đang học tốt với loss giảm đều theo thời gian.

Ví dụ 2: Biên phân lớp (Decision Boundary)

Giả sử chúng ta dùng SVM hoặc MLP để phân loại dữ liệu 2 chiều. Để minh họa ranh giới phân lớp, ta cần tạo lưới điểm để predict. Nếu tạo lưới quá thưa:

```
1 xx, yy = np.meshgrid(np.arange(-5, 5, 1),
2                         np.arange(-5, 5, 1)) # sparse grid
```

→ Đường ranh giới sẽ răng cưa, sai lệch, thậm chí mất kết nối.

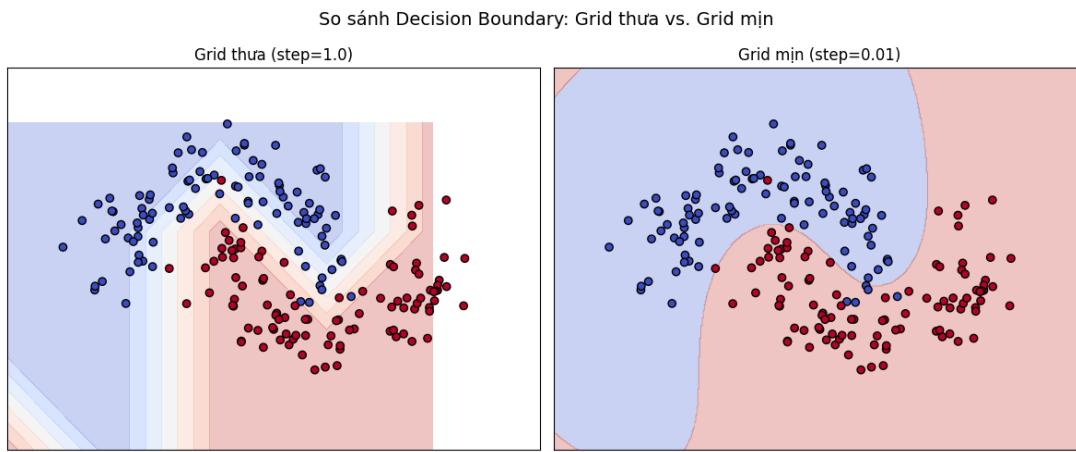
Nếu làm mượt:

```
1 xx, yy = np.meshgrid(np.arange(-5, 5, 0.01),
2                         np.arange(-5, 5, 0.01)) # smooth grid
```

→ Biên phân lớp hiện rõ, chúng ta hiểu cách mô hình "suy nghĩ" như thế nào :v

Nhận xét:

- *Trái*: Grid thưa (step = 1.0) khiến đường ranh giới bị răng cưa, mất chi tiết.
- *Phải*: Grid mịn (step = 0.01) cho thấy biên phân lớp mượt mà, giúp ta hiểu rõ cách mô hình phân biệt hai lớp.



Biên phân lớp với grid thưa và grid mịn

Ví dụ 3: Hàm sigmoid bị hiểu nhầm

Hàm sigmoid:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Nếu chúng ta vẽ chỉ với 5 điểm:

```
1 x = np.linspace(-5, 5, 5)
2 y = 1 / (1 + np.exp(-x))
```

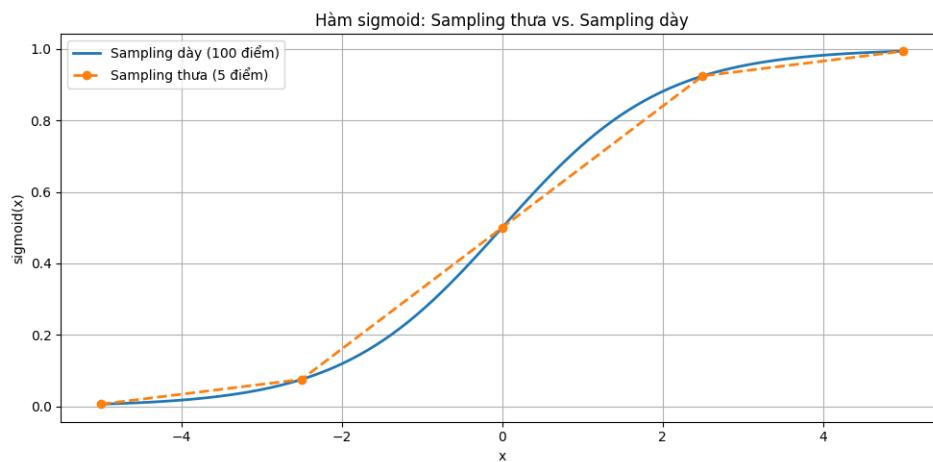
→ Đường cong trông gần như tuyến tính!

Nếu vẽ 100 điểm:

```
1 x = np.linspace(-5, 5, 100)
```

→ Chúng ta mới thấy được:

- Vùng tuyến tính ở giữa
- Vùng bao hoà hai bên



Hàm sigmoid: Sampling thưa vs. Sampling dày

Nhận xét:

- Sampling thừa (5 điểm) khiến đường cong trông gần như tuyến tính → dễ hiểu nhầm bản chất hàm sigmoid.
- Sampling dày (100 điểm) thể hiện rõ vùng tuyến tính ở giữa và vùng bao hòa hai bên.

Nguyên tắc

Đồ thị là tiếng nói của mô hình. Nếu bạn không sampling đủ dày, bạn đang bịt miệng nó.

Tóm lại:

- Làm mượt đồ thị không chỉ giúp đồ thị "đẹp", mà là một bước **thiết yếu** trong phân tích dữ liệu hiện đại.
- Nhờ `np.arange()`, ta có thể sampling với độ phân giải cao, từ đó nhìn ra những điều mà `list` và `range()` không thể.
- Đây là lý do vì sao bất cứ ai làm AI, khoa học dữ liệu, mô phỏng vật lý, v.v. đều xem `np.arange()` như vũ khí đầu tay.

1.4. Tạo mảng dựa trên mảng khác

Khi muốn tạo mảng giống `shape` và `dtype` của một mảng đã có:

- `np.zeros_like(array)`, `np.ones_like(array)`
- `np.full_like(array, fill_value)`

Code minh họa:

```

1 a = np.array([[1, 2, 3], [4, 5, 6]])
2
3 np.zeros_like(a)      # [[0, 0, 0], [0, 0, 0]]
4 np.full_like(a, -1)   # [[-1, -1, -1], [-1, -1, -1]]
```

Ứng dụng trong xử lý ảnh

Trong xử lý ảnh, một tác vụ rất thường gặp là tạo một **mặt nạ (mask)** — thường là mảng 0–1 có cùng `shape` với ảnh gốc — để đánh dấu các vùng cần xử lý (ví dụ: vùng sáng, vật thể, biên cảnh...).

Khởi tạo mask cùng shape với ảnh

- `np.zeros_like(image)` — tạo mặt nạ toàn 0
- `np.ones_like(image)` — tạo mặt nạ toàn 1
- `np.full_like(image, value)` — tạo mặt nạ với giá trị tùy ý

Ví dụ: Tạo mask vùng sáng trong ảnh grayscale

```

1 import numpy as np
2 import cv2
3
4 # grayscale image
5 image = cv2.imread('/content/
    NewBackground.jpg', 0)
6
7 # Tao mask: vung sang (pixel > 180)
8 mask = np.zeros_like(image)
9 mask[image > 180] = 1
10
11 # Ap dung mask: giam do sang vung sang di
    50 don vi
12 output = image.copy()
13 output[mask == 1] = \
14 np.clip(output[mask == 1] - 50, 0, 255)
15
16 print(mask.shape == image.shape)
17 print(mask)

```

===== Output =====

```

True
[[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 ... 1 0 0]
 [0 0 0 ... 0 0 0]]

```

- Bây giờ mask là ma trận 0–1 có cùng kích thước với ảnh gốc.
- Có thể dùng để tô vùng sáng, đếm pixel, làm attention mask, hoặc tách foreground (đối tượng).



Minh họa ba bước xử lý ảnh:

(1) ảnh gốc, (2) tạo mask vùng sáng (pixel > 180), và (3) làm tối vùng sáng bằng mask.

Cách làm này thường dùng trong:

- Làm mờ hậu cảnh, làm nổi bật foreground (đối tượng).
- Data augmentation trong segmentation.
- Xử lý ảnh y tế hoặc ảnh vệ tinh RGB.

Tóm lại:

Tính năng	Ứng dụng cụ thể
np.zeros_like(image)	Tạo mask nhanh
Dùng trong	Segment ảnh, xử lý vùng sáng, masking trong CNN
Ưu điểm	Ngắn gọn, đúng kiểu dữ liệu, không cần chỉ định shape

1.5. Tạo array bằng hàm sinh nội dung

- `np.fromiter(iterable, dtype)` – tạo array từ iterable.
- `np.fromfunction(func, shape)` – sinh array từ hàm nhận index (i,j,...).

Code minh họa:

```

1 arr1 = np.fromiter((i**2 for i in range
2   (5)), dtype=int)
3 print('arr1: ', arr1)
4 arr2 = np.fromfunction(lambda i, j: i + j
5   , (3, 3), dtype=int)
6 print('arr2: \n', arr2)

```

===== Output =====

```

arr1: [ 0  1  4  9 16]
arr2:
 [[0 1 2]
 [1 2 3]
 [2 3 4]]

```

1.6. Tạo mảng ngẫu nhiên với `np.random`

NumPy cung cấp một module mạnh mẽ là `np.random` để sinh mảng ngẫu nhiên theo nhiều phân phối khác nhau: đồng đều, chuẩn (Gaussian), rời rạc (int), nhị phân... Việc sinh mảng ngẫu nhiên đặc biệt hữu ích trong:

- Sinh dữ liệu giả để test mô hình.
- Khởi tạo trọng số ban đầu cho mô hình AI.
- Mô phỏng xác suất – thống kê.

1.6.1. Sinh mảng phân phối đều

- `np.random.rand(d0, d1, ...)` → sinh các số thực phân phối đều trên [0, 1), dạng float.
- `np.random.uniform(low, high, size)` → sinh số thực trong đoạn [low, high).

Code minh họa:

```

1 np.random.rand(2, 3)           # [[0.54, 0.71, 0.60], ...]
2 np.random.uniform(10, 20, 5)    # [11.23, 17.88, 15.12, ...]

```

1.6.2. Sinh mảng phân phối chuẩn (Gaussian)

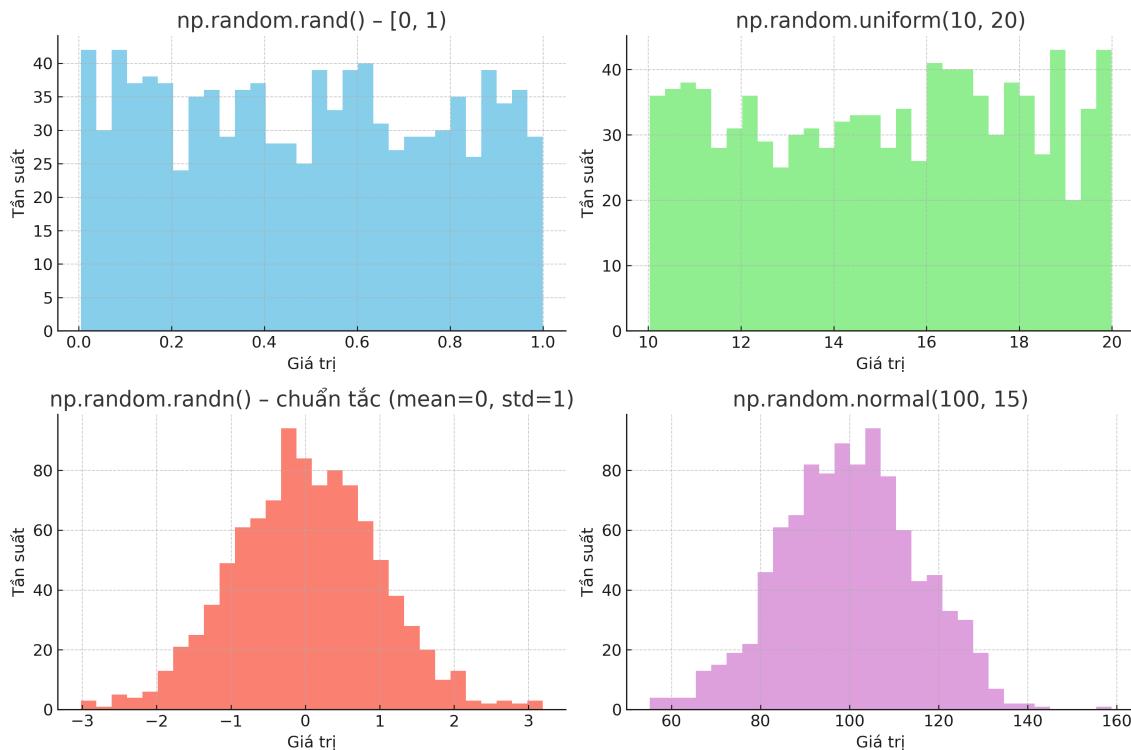
- `np.random.randn(d0, d1, ...)` → phân phối chuẩn chuẩn tắc (mean = 0, std = 1).
- `np.random.normal(loc, scale, size)` → phân phối chuẩn tổng quát.

Code minh họa:

```

1 np.random.randn(3)             # [ 0.21, -0.89, 1.45]
2 np.random.normal(loc=100, scale=15, size=(2, 2))
3 # [[92.1, 108.4], [103.5, 96.7]]

```



Hàng trên: Mô phỏng phân phối đều, dữ liệu có dạng hình chữ nhật

Hàng dưới: Mô phỏng phân phối chuẩn, dữ liệu có dạng hình chuông

1.6.3. Sinh số ngẫu nhiên

`np.random.randint(low, high, size)` → Sinh số ngẫu nhiên rời rạc từ `low` đến `high-1`.

Code minh họa:

```

1 np.random.randint(0, 10, size=(3, 3))
2 # [[6, 2, 8],
3 #  [1, 9, 5],
4 #  [0, 4, 3]]
```

1.6.4. Sinh mảng nhị phân, chọn ngẫu nhiên

- `np.random.choice(a, size)` – lấy mẫu ngẫu nhiên từ 1D array `a`.
- `np.random.binomial(n, p, size)` – mô phỏng phân phối nhị phân (Bernoulli trials).

Code minh họa:

```

1 np.random.choice([10, 20, 30], size=5)
2 # [30, 10, 10, 20, 30]
3
4 np.random.binomial(n=10, p=0.5, size=5)
5 # [4, 5, 6, 3, 7]
```

1.6.5. Thiết lập seed để tái lập kết quả

Khi làm việc với số ngẫu nhiên, nếu muốn kết quả có thể lặp lại ở các lần chạy khác nhau (ví dụ để debug), chúng ta nên đặt “hạt giống” (seed):

Code minh họa:

```
1 np.random.seed(42)
2 np.random.rand(3)      # Always return the same result
```

Ghi chú:

- Kể từ NumPy 1.17, có thể dùng Generator mới: `np.random.default_rng(seed)` — ưu tiên dùng trong ứng dụng nghiêm túc.
- Ví dụ:

```
1 rng = np.random.default_rng(seed=123)
2 rng.integers(0, 10, size=4)      # Thay cho np.random.randint
```

Tổng kết:

Hàm	Mục đích
<code>rand, uniform</code>	Số thực đồng đều
<code>randn, normal</code>	Số thực chuẩn
<code>randint, integers</code>	Số nguyên
<code>choice, binomial</code>	Lấy mẫu, mô phỏng
<code>seed, default_rng</code>	Điều khiển tính ngẫu nhiên

1.6.6. Ứng dụng thường gặp:

- Sinh dữ liệu test cho thuật toán.
- Sinh trọng số ban đầu trong mạng nơ-ron.
- Mô phỏng rút thăm, xác suất, thống kê Monte Carlo.

Tiếp theo là phần thú vị nhất của mục này, chúng ta sẽ cùng đi qua một số ví dụ áp dụng những lý thuyết khá lâng nhàng ở trên.

Ví dụ 1: Sinh dữ liệu giả để kiểm thử mô hình phân loại ung thư

Giả sử ta đang phát triển một mô hình học máy phân loại ung thư tuyến vú (benign vs. malignant), nhưng chưa có dữ liệu thật. Ta có thể sử dụng `np.random` để mô phỏng dữ liệu đầu vào và nhãn mục tiêu.

Tạo dữ liệu giả cho 100 bệnh nhân, mỗi người có 2 đặc trưng (feature):

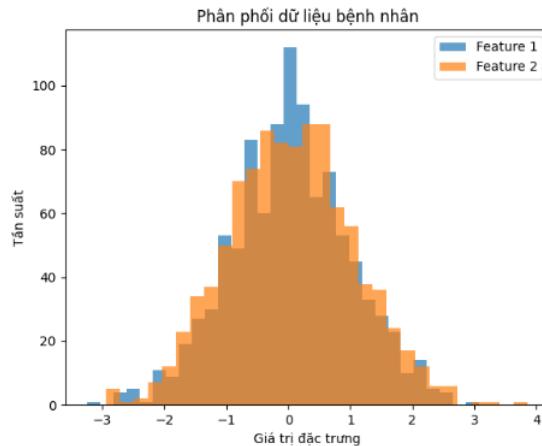
```
1 X = np.random.normal(loc=0, scale=1, size=(100, 2))  # Input Data
2 y = np.random.randint(0, 2, size=100)                 # Label
```

Trong đó:

- `X` là mảng $(100, 2)$ mô phỏng đặc trưng chuẩn hóa (ví dụ: kích thước khối u, mật độ mô).

- y là mảng nhän nhị phân, cho biết mỗi bệnh nhân là lành hay ác tính: 0 = lành tính, 1 = ác tính.

Mô hình học máy (như Logistic Regression hoặc SVM) có thể được huấn luyện trên dữ liệu này để kiểm tra pipeline, debugging, hoặc đánh giá thuật toán trước khi áp dụng vào dữ liệu thật.



Hình 5: Phân phối dữ liệu bệnh nhân: hai đặc trưng mô phỏng theo phân phối chuẩn

Ví dụ 2: Khởi tạo trọng số ban đầu cho mô hình mạng nơ-ron chẩn đoán võng mạc

Trong deep learning, việc khởi tạo trọng số ban đầu có ảnh hưởng rất lớn đến khả năng hội tụ của mô hình. Ví dụ dưới đây mô phỏng việc khởi tạo trọng số cho một lớp fully-connected trong mô hình phát hiện bệnh võng mạc tiểu đường từ ảnh mắt.

```

1 # Dense layer: 512 input (from image) → 128 output neurons
2 W = np.random.normal(loc=0.0, scale=0.01, size=(512, 128))    # Initial weight
3 b = np.zeros(128)                                              # Bias initialized to 0

```

Trong đó:

- W là ma trận trọng số với các giá trị nhỏ quanh 0 (theo phân phối chuẩn), giúp tránh gradient quá lớn/nhỏ.
- b là vector bias khởi tạo bằng 0, một cách làm phổ biến.

Các trọng số này sẽ được tối ưu trong quá trình huấn luyện để mô hình học cách phân biệt bệnh – không bệnh dựa trên đặc trưng hình ảnh.

Ví dụ 3: Mô phỏng hiệu quả vaccine bằng thống kê Monte Carlo

Trong dịch tễ học, ta thường mô phỏng hiệu quả vaccine thông qua phương pháp thống kê. Ví dụ: nếu một loại vaccine có hiệu quả 70%, thì mỗi người có 70% cơ hội được bảo vệ.

Ta có thể mô phỏng kết quả tiêm cho 10.000 người như sau:

```

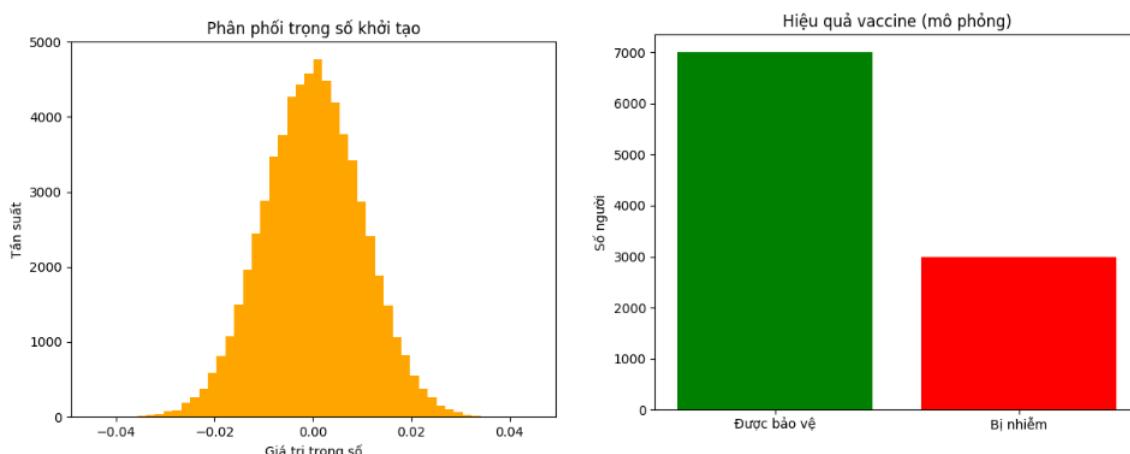
1 # 1 = not infected after vaccination, 0 = still infected
2 outcome = np.random.binomial(n=1, p=0.7, size=10_000)
3
4 # Calculate the actual protection rate
5 protection_rate = np.mean(outcome)
6 print(f"Protection rate: {protection_rate:.2%}")

```

Giải thích:

- `np.random.binomial(n=1, p=0.7)` mô phỏng 1 thử nghiệm Bernoulli: 70% khả năng thành công.
- Lặp lại 10.000 lần sẽ cho phân phối kết quả tiêm thực tế.
- Lấy trung bình để ước lượng hiệu quả vaccine trên quy mô lớn.

Phương pháp này gọi là **Monte Carlo simulation** và được dùng rộng rãi trong thống kê, khoa học dữ liệu và nghiên cứu chính sách y tế.



Trái: Phân phối trọng số mạng nơ-ron: trọng số khởi tạo nhỏ quanh 0

Phải: Mô phỏng hiệu quả vaccine: gần 70% người được bảo vệ theo giả định

6.6.7. Tổng kết:

Hàm	Mô tả ngắn
<code>np.array</code>	Tạo từ list / tuple
<code>np.zeros</code>	Mảng toàn 0
<code>np.ones</code>	Mảng toàn 1
<code>np.full</code>	Mảng toàn giá trị bất kỳ
<code>np.arange</code>	Dãy cách đều, bước tùy ý
<code>np.linspace</code>	Dãy chia đều số lượng phần tử
<code>np.empty</code>	Cấp phát nhưng không khởi tạo
<code>np.eye</code>	Ma trận đơn vị
<code>np.fromiter</code>	Từ iterable
<code>np.fromfunction</code>	Sinh từ chỉ số hàm
<code>*_like()</code>	Khởi tạo theo mảng đã có

Ghi chú: Việc dùng đúng hàm tạo phù hợp không chỉ giúp viết code “Pythonic” mà còn đảm bảo hiệu suất xử lý tốt nhất khi làm việc với dữ liệu lớn.

2. Read

Trong bất kỳ thao tác xử lý dữ liệu nào, sau khi tạo và lưu trữ mảng (Create), bước tiếp theo tự nhiên là đọc và truy xuất dữ liệu từ đó. Ở NumPy, việc “đọc” không đơn thuần là in ra một giá trị — mà là cả một hệ thống kỹ thuật mạnh mẽ giúp ta lấy ra phần tử, lát cắt, hoặc cấu trúc con trong mảng nhiều chiều. Việc truy xuất dữ liệu hiệu quả sẽ quyết định độ mượt của toàn bộ pipeline xử lý.

Phần này sẽ trình bày chi tiết về cách **Read – truy xuất dữ liệu** trong ndarray, mở đầu với một kỹ thuật cực kỳ quan trọng: **Indexing**.

2.1. Kỹ thuật Indexing

Khi muốn lấy ra một phần dữ liệu trong mảng — dù là một số, một dòng, một cột hay cả một khối 3D — ta phải biết cách “**chỉ tay vào chỗ cần thiết**”.

Đó chính là nhiệm vụ của **Indexing**.

Hiểu đơn giản, Indexing là **ngôn ngữ để truy cập dữ liệu trong ndarray**, giống như cách ta dùng địa chỉ để tìm đến một ngôi nhà.

Trong phần này, ta sẽ đi sâu vào **các dạng Indexing** trong NumPy – được chia thành 3 nhánh chính:

- **Scalar**
- **Basic Slicing**
 - Dấu `:`
 - `None / np.newaxis`
 - Số nguyên đơn
- **Advanced / Fancy Indexing**
 - `int-array`
 - `boolean-mask`

I. Scalar Indexing

Định nghĩa

Scalar indexing là kỹ thuật truy xuất một phần tử đơn lẻ trong mảng NumPy bằng cách sử dụng các **chỉ số nguyên đơn** (`int`) ứng với từng chiều của mảng.

- Với mảng 1 chiều: chỉ cần 1 chỉ số.
- Với mảng 2 chiều trở lên: cần **1 chỉ số cho mỗi chiều**.

Cú pháp

- `arr[i]`: Truy xuất phần tử thứ i của mảng 1D
- `arr[i, j]`: Truy xuất phần tử tại dòng i, cột j của mảng 2D
- `arr[i, j, k]`: Truy xuất phần tử tại (i, j, k) trong mảng 3D

Giải thích:

- `arr[3]` với mảng 1 chiều là truy xuất phần tử tại chỉ số 3.
- `arr[1, 2]` với mảng 2 chiều là truy xuất phần tử dòng 1, cột 2.
- Các chỉ số bắt đầu từ 0, như trong Python nói chung.

Đặc điểm:

- Trả về một bản sao chép (copy) của phần tử, không phải view.
- Việc gán lại giá trị cho biến sau khi scalar indexing **không ảnh hưởng đến mảng gốc**.
- Truy xuất rất nhanh, hiệu quả trong các phép toán đơn giản.

Ví dụ minh họa:

```

1 import numpy as np
=====
2
3 a = np.array([[1, 2, 3],
4             [4, 5, 6]])
5 print('a before:\n', a)
6
7 # Get element at row 0, column 1 → x = 2
8 x = a[0, 1]
9 # Reassign x value, does not affect 'a'
10 x = 100
11
12 # Array a remains the same
13 print('a after:\n', a)
===== Output =====
a before:
[[1 2 3]
 [4 5 6]]
a after:
[[1 2 3]
 [4 5 6]]
```

Giải thích:

- Biến `a` là một mảng NumPy 2 chiều, kích thước (2, 3).
- Lệnh `a[0, 1]` truy xuất phần tử tại dòng 0, cột 1, tức là giá trị 2.
- Biến `x` nhận bản sao (copy) của phần tử đó. Khi viết `x = 100`, chỉ có `x` thay đổi, còn mảng `a` hoàn toàn không bị ảnh hưởng.
- Việc này chứng minh rằng **scalar indexing trả về copy**, không phải là view liên kết với mảng gốc.

Nếu chỉ cần lấy một phần tử cụ thể từ mảng mà không muốn ảnh hưởng đến mảng gốc, scalar indexing là lựa chọn đơn giản và hiệu quả nhất.

II. Basic Slicing – Truy xuất lát cắt

Slicing trong NumPy là một trong những kỹ thuật cơ bản nhưng vô cùng mạnh mẽ giúp ta truy xuất từng phần trong mảng (array) theo nhiều chiều. Nó không chỉ hỗ trợ thao tác lấy dòng/cột, mà còn cho phép tạo ra **view** linh hoạt, giúp thay đổi dữ liệu gốc một cách hiệu quả.

1. Cú pháp slicing cơ bản

Cú pháp

```
arr[for_axis_0, for_axis_1, ...]
    • : – lấy tất cả phần tử trên chiều đó
    • a:b – lấy từ phần tử thứ a đến b-1
    • Có thể áp dụng slicing cho nhiều chiều cùng lúc
```

2. Slicing nhiều chiều: ví dụ minh họa

Giả sử ta có mảng 2 chiều:

```
1 data = np.array([[1, 2],
2                  [3, 4],
3                  [5, 6]])
```

Một số ví dụ truy xuất:

- `data[0,1]` ⇒ phần tử dòng 0, cột 1 (giá trị: 2)
- `data[1:3]` ⇒ hai dòng cuối:

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

- `data[0:1, 0]` ⇒ dòng 0, cột 0: `[[1]]`
- `data[:, :]` ⇒ toàn bộ mảng

3. View hay Copy? Mutable hay Immutable?

Khi slicing, bạn sẽ nhận được một **view** chứ không phải bản sao. Nghĩa là nếu bạn thay đổi phần tử trong slice, dữ liệu gốc cũng sẽ thay đổi.

```
1 a_arr = np.array([[1,2,3],
2                   [5,6,7]])
3 b_arr = a_arr[:, 1:3]      # lấy ôct 1 và 2 ủca օmi dòng
4 b_arr[0,0] = 99            # thay đổi պhn ստ đầu tiên ւca b_arr
5 print(a_arr)
```

Kết quả:

```
1 [[ 1  99  3]
2  [ 5   6  7]]
```

⇒ `b_arr` là view, nên thay đổi ảnh hưởng tới `a_arr`.

4. Truy xuất dòng: vector vs matrix

```

1 arr = np.array([[1,2,3],
2                 [5,6,7],
3                 [9,10,11]])
4
5 row_m1 = arr[1, :]      # vector 1 chiều, shape = (3,)
6 row_m2 = arr[1:2, :]    # ma trận 2D, shape = (1,3)

```

- `arr[1, :]` trả về vector: mảng 1 chiều
- `arr[1:2, :]` giữ nguyên chiều: trả về ma trận (2D)

5. Truy xuất cột: vector vs matrix

```

1 col_m1 = arr[:, 1]      # vector shape (3,)
2 col_m2 = arr[:, 1:2]    # ma trận shape (3,1)

```

- `arr[:, 1]` ⇒ vector 1 chiều (flattened)
- `arr[:, 1:2]` ⇒ giữ nguyên ma trận 2 chiều

6. Fancy indexing: dùng list để truy xuất

```

1 arr = np.array([[1,2],
2                 [3,4],
3                 [5,6]])
4
5 out1 = arr[[0,1,2], [0,1,0]]  # lấy (0,0), (1,1), (2,0)
6 out2 = arr[[0,0], [1,1]]      # lấy (0,1), (0,1)

```

out1: [1 4 5] – lấy các phần tử theo index tương ứng
out2: [2 2] – có thể truy xuất lặp lại cùng 1 phần tử

Tổng kết nhanh:

- Slicing là cú pháp linh hoạt để truy xuất mảng theo chiều
- Cần phân biệt kỹ giữa việc trả về **view** hay **copy**
- Fancy indexing **luôn tạo bản sao** (copy), khác với slicing
- Để giữ nguyên shape 2D khi lấy dòng/cột, dùng `a[1:2, :]` hoặc `a[:, 1:2]`

1. Dấu ":" (colon)

Cú pháp:

```
1 arr[start:stop:step]
```

Ví dụ:

```

1 a = np.array([0, 1, 2, 3, 4, 5])
2 b = a[1:4]          # b = [1, 2, 3]
3 b[0] = 100
4 print(a)           # [0, 100, 2, 3, 4, 5]

```

⇒ `b` là view, thay đổi `b` sẽ ảnh hưởng tới `a`.

2. Số nguyên đơn

Dùng để giữ nguyên một chiều (co lại 1 chiều trong mảng nhiều chiều).

```

1 a = np.array([[10, 20, 30],
2                 [40, 50, 60]])
3
4 print(a[1])      # [40 50 60]
5 print(a[1, 2])   # 60

```

3. None hoặc np.newaxis – thêm chiều mới

```

1 a = np.array([1, 2, 3])
2 print(a.shape)          # (3,)
3
4 a2 = a[:, np.newaxis]   # thêm chiều ôm
5 print(a2.shape)         # (3, 1)
6
7 a3 = a[:, None]         # ướtng đương

```

⇒ vẫn là view, không phải copy.

III. Advanced (Fancy) Indexing – Truy xuất nâng cao

1. Integer array indexing

```

1 a = np.array([10, 11, 12, 13])
2 print(a[[0, 3]])        # [10 13]
3
4 a[[0, 3]] = [100, 200]
5 print(a)                # [100 11 12 200]

```

Mảng 2D:

```

1 a = np.array([[1, 2],
2                 [3, 4],
3                 [5, 6]])
4 print(a[[0, 2], [1, 0]]) # [2, 5]

```

2. Boolean Mask

```

1 a = np.array([5, 10, 15, 20, 25])
2 mask = a > 15
3 print(mask)           # [False False False  True  True]
4 print(a[mask])        # [20 25]
5
6 # Ông già:
7 print(a[a > 15])    # [20 25]

```

⇒ Fancy Indexing luôn trả về copy.

Bảng so sánh View vs Copy

Loại Indexing	Cú pháp tiêu biểu	View hay Copy
Scalar Indexing	<code>arr[1, 2]</code>	Copy
Basic Slicing	<code>arr[1:4], arr[:, 1]</code>	View
<code>np.newaxis</code>	<code>arr[:, None]</code>	View
Fancy Indexing (int)	<code>arr[[1,3]]</code>	Copy
Boolean Mask	<code>arr[arr > 5]</code>	Copy

Tổng kết

- **Indexing** là kỹ thuật cốt lõi giúp ta làm chủ dữ liệu trong mảng.
- **Basic Slicing** và `np.newaxis` thường trả về view (cùng vùng nhớ).
- **Fancy Indexing** (int-array, boolean mask) luôn trả về copy.
- Cần phân biệt kỹ giữa view và copy để tránh lỗi khi thao tác dữ liệu.

3. Update

Update trong ndarray nghĩa là thay đổi giá trị tại một vị trí cụ thể. Do NumPy mảng có chỉ số, ta có thể cập nhật dễ dàng bằng cách truy cập đúng chỉ mục và sử dụng phép gán trực tiếp.

```

1 import numpy as np
2
3 a = np.array([
4     [1, 2, 3],
5     [4, 5, 6]])
6
7 # assign new value
8 a[1,2] = 9999
9
10 print(a)

```

Một tính năng tuyệt vời của NumPy là khả năng cập nhật giá trị có điều kiện bằng cách sử dụng Boolean Indexing . Boolean Indexing cho phép bạn chọn các phần tử từ một mảng dựa trên các điều kiện. Để thực hiện như vậy, trước tiên bạn phải tạo một mảng Boolean bằng cách áp dụng điều kiện cho một mảng NumPy.

```

1 import numpy as np
2
3 a = np.array([
4     [1, 2, 3],
5     [4, 5, 6]
6 ])
7
8 # create boolean array
9 bool_arr = a < 5
10
11 print(bool_arr)

```

Như bạn có thể thấy, chúng ta có được một mảng Boolean có cùng hình dạng với mảng ban đầu. Sau đó, bạn sử dụng mảng Boolean này để lập chỉ mục cho mảng gốc. Chỉ các phần tử có mảng Boolean True mới được chọn. Theo cách này, bạn có thể lọc các phần tử từ một mảng:

```

1 import numpy as np
2
3 a = np.array([
4     [1, 2, 3],
5     [4, 5, 6]
6 ])
7
8 # create boolean array
9 bool_arr = a < 5
10
11 # index 'a' using boolean array
12 print(a[bool_arr])

```

Và tất nhiên, bạn cũng có thể gán các giá trị mới cho các phần tử có điều kiện True. Hãy cập nhật tất cả các giá trị nhỏ hơn 5:

```

1 import numpy as np
2
3 a = np.array([
4     [1, 2, 3],
5     [4, 5, 6]
6 ])
7
8 # create boolean array
9 bool_arr = a < 5
10
11 # update values
12 a[bool_arr] = 9999
13
14 print(a)

```

Tóm gọn lại, Thay vì thực hiện theo 2 bước, chúng ta có thể lập chỉ mục và áp dụng điều kiện cùng lúc:

```

1 import numpy as np
2
3 a = np.array([
4     [1, 2, 3],
5     [4, 5, 6]
6 ])
7
8 # conditionally update 'a'
9 a[a < 5] = 9999
10
11 print(a)

```

Một cách khác để cập nhật nhiều phần tử trong mảng NumPy là sử dụng slicing. Ví dụ, ở đây chúng ta cập nhật toàn bộ hàng đầu tiên bằng cách cắt lát:

```

1 import numpy as np
2
3 a = np.array([
4     [1, 2, 3],
5     [4, 5, 6]
6 ])
7
8 # update first row
9 a[0, :] = [100, 200, 300]
10
11 print(a)
12 # [[100 200 300] [4 5 6]]

```

Hoặc cập nhập 2 hàng đầu tiên và 2 cột đầu tiên:

```

1 import numpy as np
2
3 a = np.array([
4     [1, 2, 3],
5     [4, 5, 6]
6 ])
7
8 # update first 2 rows and first 2 cols
9 a[:2,:2] = [[9999, 9999], [9999, 9999]]
10
11 print(a)

```

Lưu ý: Đảm bảo rằng hình dạng của mảng bạn gán khớp với hình dạng của lát cắt đã chọn, nếu không, bạn sẽ gặp lỗi ValueError: could not broadcast input array from shape (...) into shape (...).

Kết luận: Thay đổi giá trị giữa hai giá trị cụ thể trong một mảng Numpy là một tác vụ phổ biến trong quá trình xử lý dữ liệu. Bằng cách sử dụng chỉ mục Boolean, bạn có thể thực hiện việc này một cách hiệu quả và theo cách dễ hiểu. Hãy nhớ luôn cẩn thận khi thay đổi giá trị trong một mảng, vì nó có thể thay đổi dữ liệu của bạn theo những cách mà có thể không nhận thấy ngay lập tức.

4. Delete

Hàm numpy.delete() trong Python là một công cụ quan trọng để thao tác mảng. Nó cho phép bạn dễ dàng xóa các phần tử khỏi ndarray, đặc biệt hữu ích trong quá trình xử lý dữ liệu trước, khi bạn có thể cần bỏ qua các điểm dữ liệu hoặc điều chỉnh kích thước tập dữ liệu để tương thích với phân tích.

numpy.delete() với Mảng một chiều

- Xóa các phần tử cụ thể theo chỉ mục

```

1 import numpy as np
2 array_1d = np.array([1, 2, 3, 4, 5])
3 result = np.delete(array_1d, 2)
4 print(result) #3

```

- Xóa nhiều chỉ mục

```

1 array_md = np.array([0, 1, 2, 3, 4, 5, 6])
2 result = np.delete(array_md, [1, 3, 5])
3 print(result) #0,2,4,5,6

```

numpy.delete() với Mảng đa chiều

- Xóa toàn bộ một hàng hoặc một cột

```

1 array_2d = np.array([[1, 2], [3, 4], [5, 6]])
2 result_row = np.delete(array_2d, 1, axis=0)
3 result_column = np.delete(array_2d, 0, axis=1)
4 print("After deleting row:", result_row)
5 print("After deleting column:", result_column)

```

Trong ví dụ này, hàng 1 và cột 0 được xóa khỏi mảng 2D array_2d. Hàm này sửa đổi cấu trúc mảng theo trực tiếp được chỉ định.

- Xóa tuần tự từ nhiều trục

```
1 array_3d = np.array([[1, 2], [3, 4]], [[5, 6], [7, 8]], [[9, 10], [11, 12]]])
2 step1 = np.delete(array_3d, 0, axis=0) # Deletes the first block of 2D array
3 result = np.delete(step1, 1, axis=1) # Deletes the second column from each 2D
   block
4 print(result)
```

Trong đoạn mã này, việc xóa diễn ra tuần tự trên các trục khác nhau trong một mảng 3D. Ban đầu, toàn bộ mảng con 2D đầu tiên bị xóa, sau đó là cột thứ hai của mảng con còn lại.

Hàm numpy.delete() trong Python có thể ảnh hưởng sâu sắc đến cách bạn thao tác và xử lý các tập dữ liệu, đặc biệt là với các mảng numpy. Bằng cách làm theo các ví dụ được đề cập ở trên, việc xóa một hoặc nhiều phần tử, cũng như toàn bộ các hàng hoặc cột khỏi các mảng có nhiều chiều khác nhau trở thành một quy trình hợp lý. Sử dụng các chiến lược này để nâng cao khả năng xử lý dữ liệu của bạn trong các dự án máy tính khoa học và học máy, đảm bảo các tập dữ liệu của bạn được điều chỉnh hoàn hảo theo yêu cầu phân tích của bạn.

Phần 4: Array Transformations

Trong NumPy, thao tác với mảng không chỉ dừng lại ở việc truy xuất hay tính toán — mà còn bao gồm cả việc **biến đổi hình dạng, kết cấu và cấu trúc dữ liệu**. Những biến đổi này được gọi chung là **Array Transformations**, và chúng đóng vai trò then chốt trong:

- Tái định dạng dữ liệu (`reshape`),
- Nối, xếp, lặp mảng (`concatenate`, `stack`, `repeat`, `tile`),
- Chuẩn hoá đầu vào cho mô hình Deep Learning,
- Sắp xếp và lọc dữ liệu theo điều kiện.

Lưu ý: Đặc biệt trong lĩnh vực Trí tuệ Nhân tạo (AI), việc **hiểu rõ cách biến đổi mảng** là điều kiện tiên quyết để xây dựng pipeline hiệu quả, tránh lỗi `shape` và tối ưu hiệu suất tính toán.

Trong phần này, chúng ta sẽ lần lượt khám phá các thao tác biến đổi mảng quan trọng trong NumPy:

- **Reshape:** tái định hình mảng
- **Flatten:** làm phẳng toàn bộ dữ liệu
- **Transpose:** hoán đổi trực
- **Repeat, Tile, Stack, Concatenate**
- **Sorting & Aggregation:** sắp xếp và thống kê
- **Filtering bằng mask logic**

4.1. Reshape

4.1.1. Đặt vấn đề: Khi nào ta cần `reshape()`?

Trong quá trình làm việc với dữ liệu (đặc biệt là ảnh, âm thanh, văn bản...), bạn sẽ không tránh khỏi tình huống:

“Mảng dữ liệu có kích thước không giống như mô hình yêu cầu.”

Lúc này, ta cần một công cụ giúp **biến đổi kích thước mảng mà vẫn giữ nguyên dữ liệu gốc**. Đó chính là hàm `reshape()` của NumPy.

Tại sao phải `reshape`?

Hãy tưởng tượng chúng ta có một bức ảnh trắng đen 28×28 pixel. Máy tính sẽ lưu nó dưới dạng một mảng 1 chiều gồm 784 phần tử. Nhưng nếu ta muốn xử lý nó như một bức ảnh 2D, thì phải “gấp lại” nó về dạng 28×28 . Đó chính là `reshape`!

Một vài tình huống thường gặp

Bài toán	Dạng dữ liệu ban đầu	Cần reshape thành
Ảnh 1D	[784]	[28, 28]
Chuỗi 100 từ 5 ảnh $28 \times 28 \times 1$	[100, 300] (embedding) [5, 784]	[1, 100, 300] (thêm chiều batch) [5, 28, 28, 1]

Một vài mô hình AI yêu cầu dữ liệu như sau

- **CNN (Mạng tích chập):**
 $\text{shape} = (\text{batch}, \text{height}, \text{width}, \text{channel})$
⇒ Dùng cho xử lý ảnh.
- **RNN (Mạng hồi tiếp):**
 $\text{shape} = (\text{batch}, \text{time steps}, \text{features})$
⇒ Dùng cho xử lý chuỗi thời gian, văn bản.
- **Transformer (Mạng attention):**
 $\text{shape} = (\text{batch}, \text{seq_len}, \text{embedding_dim})$
⇒ Dùng cho NLP và cả thị giác máy tính.

Nói tóm lại, chúng ta có thể hiểu qua một chút về tác dụng của `reshape`

- `reshape()` giúp biến đổi hình dạng mảng, rất hữu ích khi xử lý dữ liệu thực tế.
- Nó không làm thay đổi nội dung, chỉ thay đổi cách ta “nhìn” và “chia khối” dữ liệu.

4.1.2. Cú pháp

`reshape` là một hàm được sử dụng để thay đổi hình dạng của một mảng mà không làm thay đổi dữ liệu bên trong nó. Kích thước mảng mới được chỉ định thông qua tham số của hàm `reshape`

Cú pháp 1 – hàm của NumPy

```
1 np.reshape (input, newshape)
```

Cú pháp 2 – phương thức của ndarray

```
1 arr.reshape(newshape, order='C')
```

Trong đó:

- input: là mảng đầu vào.
- newshape là hình dạng mới ta muốn chuyển đổi, là một tuple thể hiện kích thước mới (ví dụ: (3, 4)).
- order: chọn cách duyệt bộ nhớ:

- 'C' (mặc định): đọc theo hàng (row-major),
- 'F': đọc theo cột (column-major).

Ví dụ:

```

1 import numpy as np
2
3 a = np.arange(6)
4 a_reshaped = a.reshape((2, 3))
5 print('a reshape:\n', a_reshaped)

```

```
=====
Output =====
a reshape:
[[0 1 2]
 [3 4 5]]
```

Minh họa:

$$\text{Vector: } [0, 1, 2, 3, 4, 5] \xrightarrow{\text{reshape}(2,3)} \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{bmatrix}$$

Giải thích: Mảng ban đầu có shape (6,), sau khi reshape thành (2, 3) thì:

- Dữ liệu không đổi.
- Cách chia khối (2 hàng, 3 cột) được cập nhật lại.

Nguyên tắc quan trọng khi reshape

1. Tổng số phần tử phải giữ nguyên:

```

1 np.arange(12).reshape(3, 4) # OK
2 np.arange(12).reshape(2, 5) # ValueError!
3

```

Nếu chúng ta reshape sang kích thước khác số phần tử ban đầu, NumPy sẽ báo lỗi.

2. Dùng -1 để tự động suy ra chiều còn lại

```

1 a = np.arange(12)
2 print(a.reshape(3, -1))    # → (3, 4)
3 print(a.reshape(-1, 6))    # → (2, 6)

```

```
=====
Output =====
[ 4  5  6  7]
[ 8  9 10 11]

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

Chỉ được dùng một -1 mỗi lần reshape — nếu không, NumPy sẽ không biết chiều nào cần suy luận.

Lưu ý

Chỉ được dùng **một** -1 trong mỗi lệnh `reshape()`. Dùng nhiều hơn sẽ gây lỗi vì không thể xác định chiều nào cần tính.

4.1.3. Kiểu C-like vs Fortran-like

Khi ta `reshape` hoặc `flatten` mảng nhiều chiều, NumPy cần biết nên ưu tiên duyệt dữ liệu theo chiều nào: **hàng trước** hay **cột trước**. Đây được gọi là cách *truy xuất bộ nhớ*:

- **C-like (row-major)** – theo ngôn ngữ C.
- **Fortran-like (column-major)** – theo ngôn ngữ Fortran.

Ví dụ:

Giả sử ta có một mảng 1 chiều gồm 8 phần tử:

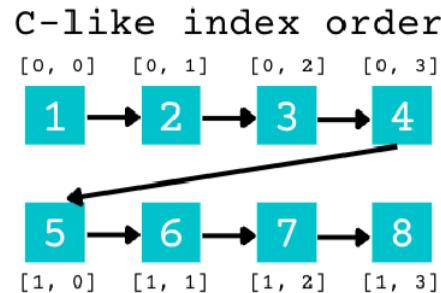
```
1 import numpy as np
2 numbers = np.array([1, 2, 3, 4, 5, 6, 7, 8])
```

- `numbers.shape` (8,)
- `numbers.ndim` 1 (1 chiều)

Giờ ta `reshape` nó thành mảng (2, 4) theo hai cách truy xuất bộ nhớ:

C-style (row-major) — hàng trước

```
1 numbers.reshape((2, 4), order='C')
```



Reshape theo order='C'

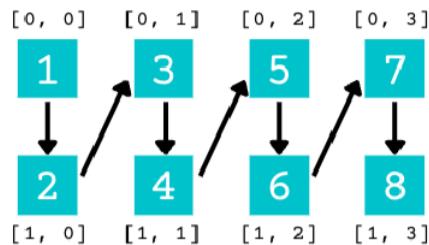
Giải thích:

- NumPy duyệt dữ liệu gốc từ trái sang phải.
- Mỗi dòng (row) được lấp đầy lần lượt:
 - Dòng 1: 1, 2, 3, 4
 - Dòng 2: 5, 6, 7, 8

Fortran-style (column-major) — cột trước

```
1 numbers.reshape((2, 4), order='F')
```

Fortran-like index order



Reshape the order='F'

Giải thích:

- NumPy duyệt dữ liệu theo từng cột (column).
- Mỗi cột được lấp đầy từ trên xuống dưới:
 - Cột 1: 1, 2
 - Cột 2: 3, 4
 - Cột 3: 5, 6
 - Cột 4: 7, 8

Ảnh hưởng đến hiệu suất

Cách truy xuất bộ nhớ phù hợp sẽ giúp:

- Tăng tốc độ xử lý dữ liệu lớn, vì CPU truy cập vùng nhớ liên tục hơn.
- Tránh lỗi shape ngầm khi tương tác với thư viện khác (như MATLAB dùng kiểu Fortran).

4.1.4. Các dạng reshape phổ biến

Khi mới học NumPy, chúng ta sẽ gặp nhiều tình huống cần thay đổi hình dạng mảng (số chiều, số phần tử trên mỗi chiều). Dưới đây là 6 dạng `reshape()` phổ biến, dễ nhớ và hay dùng nhất.

1. Flat → 2D (tạo mảng 2 chiều từ 1 chiều)

Cú pháp

```
1 a.reshape(1, -1)
```

Dùng khi: Ta có 1 dãy số (1 chiều), muốn biến nó thành 1 hàng 2 chiều.

```
1 a = np.arange(10)
2 a_2d = a.reshape(1, -1)
3 print(a_2d.shape)
```

```
===== Output =====
(1, 10)
```

Mảng (10,) đã biến thành (1, 10) — tức là 1 hàng, 10 cột.

2. 2D → Flat (làm phẳng mảng 2 chiều)

Cú pháp

```
1 a.reshape(-1)
```

Dùng khi: Ta muốn đưa mảng 2D về 1 chiều.

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

===== Output =====

Đây là cách "nén" toàn bộ mảng về dạng 1 dãy duy nhất.

3. Thêm trực (từ (n,) → (n, 1))

Cú pháp

```
1 a.reshape(-1, 1)
```

Dùng khi: Ta có mảng 1 chiều và muốn biến nó thành nhiều dòng, 1 cột.

```
1 a = np.array([1, 2, 3])
2 print(a.reshape(-1, 1))
```

===== Output =====

```
[2]
[3]
```

Dạng này hay dùng khi cần mảng dọc (vector cột).

4. Thêm chiều đầu (từ (n,) → (1, n))

Cú pháp

```
1 a[np.newaxis, :]
```

hoặc:

```
1 a.reshape(1, -1)
```

Dùng khi: Ta có 1 dãy số, muốn biến thành 1 hàng trong mảng 2D.

```
1 a = np.arange(8)
2 b = a[np.newaxis, :]
3 print(b.shape)
```

===== Output =====

```
(1, 8)
```

Cách này giống ví dụ 1, nhưng dùng thêm `np.newaxis`.

5. Biến ảnh từ 1D → 2D (giống ảnh 28x28)

Cú pháp

```
1 a.reshape(28, 28)
```

Dùng khi: Mảng có 784 phần tử (flatten), muốn đưa về ma trận 28x28.

```
1 a = np.random.rand(784)
2 img = a.reshape(28, 28)
3 print(img.shape)
```

```
===== Output =====
(28, 28)
```

Thường gặp khi học với dữ liệu ảnh trắng đen nhỏ (MNIST).

6. Chia dữ liệu theo số đặc trưng (feature)

Cú pháp

```
1 a.reshape(-1, feature\_dim)
```

Dùng khi: Ta có 1 dãy dài, muốn chia thành nhiều dòng có số cột bằng `feature_dim`.

```
1 a = np.random.rand(1000)
2 x = a.reshape(-1, 100)
3 print(x.shape)
```

```
===== Output =====
(10, 100)
```

Ví dụ trên: chia mảng 1000 phần tử thành 10 dòng, mỗi dòng 100 phần tử.

Bảng tổng hợp các dạng reshape

Tên gọi	Cú pháp	Ý nghĩa
Flat → 2D	a.reshape(1, -1)	Biến mảng 1 chiều thành 1 hàng
2D → Flat	a.reshape(-1)	Làm phẳng toàn bộ mảng
Thêm trực cột	a.reshape(-1, 1)	Biến vector thành cột dọc
Thêm chiều hàng	a[np.newaxis, :]	Thêm 1 chiều ở trên cùng
Flatten ảnh	a.reshape(28, 28)	Đưa ảnh 784 phần tử về ma trận 2D
Chia theo cột cố định	a.reshape(-1, 100)	Tự động chia dòng nếu biết số cột

- `reshape()` không thay đổi dữ liệu, chỉ thay đổi cách tổ chức mảng.
- Dùng `-1` để NumPy tự tính chiều còn lại.
- Bạn nên in `.shape` sau mỗi `reshape` để kiểm tra kết quả.

1.5. Ứng dụng trong AI

Trong thực tế, dữ liệu đầu vào của mô hình học sâu không phải lúc nào cũng có đúng `shape` như mô hình yêu cầu. Vì vậy, việc sử dụng `reshape()` là điều **bắt buộc phải thành thạo** khi làm việc với AI.

Dưới đây là các ứng dụng thường gặp:

1. Chuẩn hóa ảnh đầu vào cho CNN (Convolutional Neural Network)

Bạn có một ảnh trắng đen kích thước 28×28 pixels, nhưng CNN yêu cầu đầu vào có dạng:

(`batch_size, height, width, channel`)

```
1 import numpy as np
2
3 img = np.random.rand(28, 28) # ảnh trắng đen
4 img_batch = img.reshape(1, 28, 28, 1) # 1 ảnh, 1 channel
5 print(img_batch.shape)
```

Shape ban đầu: (28, 28)

⇒ Sau `reshape`: (1, 28, 28, 1)

Đây là điều kiện bắt buộc khi đưa ảnh vào CNN như LeNet, VGG, ResNet...

2. Định dạng chuỗi embedding cho NLP

Trong NLP, mỗi từ thường được mã hoá thành một vector (embedding). Ví dụ: bạn có 100 từ, mỗi từ là vector 300 chiều, tạo thành ma trận 100×300 .

Tuy nhiên, model yêu cầu đầu vào có shape:

(`batch_size, sequence_length, embedding_dim`)

```
1 embedding = np.random.rand(100, 300) # 100 tu, moi tu 300 chieu
2 input_tensor = embedding.reshape(1, 100, 300)
3 print(input_tensor.shape)
```

Sau `reshape`: (1, 100, 300)

Áp dụng cho mô hình như LSTM, GRU, Transformer...

3. Tự động chia batch từ vector lớn

Bạn có 5000 giá trị đầu vào, mỗi mẫu có 100 đặc trưng (feature). Không cần biết số mẫu, chỉ cần:

```
1 features = np.random.rand(5000)
2 input_batch = features.reshape(-1, 100)
3 print(input_batch.shape)
```

⇒ Kết quả: (50, 100) nghĩa là 50 mẫu, mỗi mẫu 100 đặc trưng.

Cách này thường dùng trong bước tiền xử lý dữ liệu trước khi đưa vào model.

4. Dữ liệu thời gian — reshape cho RNN

Ví dụ: bạn có chuỗi dữ liệu cảm biến gồm 60 thời điểm, mỗi thời điểm có 6 đặc trưng. Đầu vào RNN cần dạng:

(`batch_size, time_steps, features`)

```

1 signal = np.random.rand(60 * 6) # du lieu phang
2 rnn_input = signal.reshape(1, 60, 6) # batch = 1

```

Áp dụng trong bài toán: dự đoán chuỗi thời gian, nhận dạng chữ viết tay, phân tích âm thanh...

Tóm lại

Tình huống	Trước reshape	Sau reshape
Ảnh trắng đen	(28, 28)	(1, 28, 28, 1)
Chuỗi 100 từ (300-dim)	(100, 300)	(1, 100, 300)
5000 giá trị → batch	(5000,)	(50, 100)
Chuỗi thời gian 60×6	(360,)	(1, 60, 6)

4.2. Flatten – Làm phẳng mảng nhiều chiều thành 1 chiều

4.2.1. Tại sao cần flatten?

Trong nhiều bài toán, đặc biệt là **AI, học sâu**, ta sẽ cần:

- Biến mảng nhiều chiều (2D, 3D) thành **mảng 1 chiều (vector)**.
- Vì nhiều hàm hoặc mô hình chỉ chấp nhận **mảng 1 chiều** làm đầu vào.

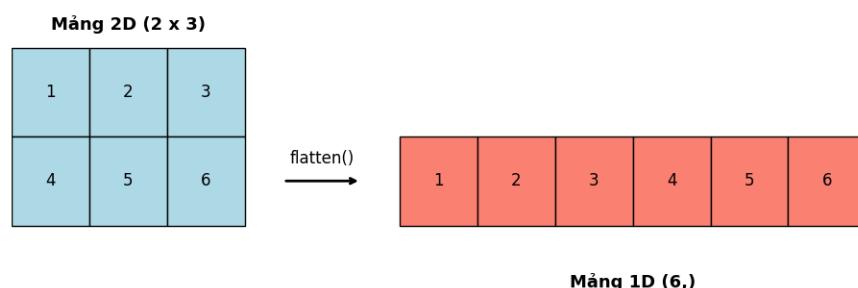
Việc này gọi là **flatten – làm phẳng mảng**.

Ví dụ:

```

1 a = np.array([[1, 2, 3],
2                 [4, 5, 6]])
3 print('a =', a)
=====
===== Output =====
a = [1, 2, 3, 4, 5, 6]

```



Hình 6: Minh họa quá trình flatten mảng 2D thành mảng 1D

4.2.2. Cách làm phẳng mảng (flatten)

NumPy có 2 cách chính để flatten:

Cách dùng	Ý nghĩa
<code>a.flatten()</code>	Luôn trả về mảng mới , tách biệt hoàn toàn với gốc.
<code>a.ravel()</code>	Trả về view nếu có thể (nhanh hơn, chia sẻ bộ nhớ).

Ví dụ:

```

1 import numpy as np
2
3 a = np.array([[1, 2],
4               [3, 4]])
5
6 print('a flatten:', a.flatten())
7 print('a ravel:', a.ravel())

```

```
=====
Output =====
a flatten: [1 2 3 4]
a ravel: [1 2 3 4]
```

Kết quả giống nhau, nhưng bên trong hoạt động khác nhau.

4.2.3. `flatten()` có ảnh hưởng mảng gốc không?

```

1 a = np.array([[1, 2],
2               [3, 4]])
3
4 b = a.flatten()
5 b[0] = 999
6
7 print("a:\n", a) # a khong doi
8 print("b:", b)   # b thay doi

```

```
=====
Output =====
a:
[[1 2]
 [3 4]]
b: [999 2 3 4]
```

Vì `flatten()` tạo bản sao \Rightarrow mảng `a` không bị thay đổi.

4.2.4. `ravel()` thì sao?

```

1 a = np.array([[1, 2],
2               [3, 4]])
3
4 b = a.ravel()
5 b[0] = 999
6
7 print("a:\n", a) # a bi thay doi
8 print("b:", b)

```

```
=====
Output =====
a:
[[999 2]
 [3 4]]
b: [999 2 3 4]
```

Vì `ravel()` trả về **view** (chung vùng nhớ) nên thay đổi `b` cũng làm thay đổi `a`.

4.2.5. Duyệt theo hàng hay theo cột?

ta có thể chọn cách NumPy "đọc" dữ liệu:

Kiểu đọc	Dùng gì	Ý nghĩa
Theo hàng (C-style)	order='C'	Duyệt từng hàng trước (mặc định).
Theo cột (F-style)	order='F'	Duyệt từng cột trước (ít dùng hơn).

Ví dụ:

```
1 a = np.array([[1, 2],
   [3, 4]])
2
3
4 print(a.flatten(order='C')) # [1 2 3 4]
5 print(a.flatten(order='F')) # [1 3 2 4]
```

4.2.6. Khi nào dùng flatten trong AI?

Flatten thường được dùng trước khi đưa dữ liệu vào **Dense Layer** trong mạng học sâu (CNN):

- Ảnh đầu vào: (28, 28, 1)
- Sau khi qua Conv2D → (7, 7, 64)
- Cần flatten: (7 × 7 × 64 = 3136,)

```
1 x = np.random.rand(7, 7, 64)
2 x_flat = x.flatten()
3 print(x_flat.shape) # (3136,)
```

4.3. Transpose – Hoán vị trực mảng

4.3.1. Đặt vấn đề

Khi xử lý dữ liệu nhiều chiều, ta sẽ cần **hoán vị các trực (axes)** để đưa mảng về đúng hình dạng mô hình yêu cầu.

Ví dụ:

- Dữ liệu ảnh trong Deep Learning thường ở dạng (N, H, W, C):
 - N – số lượng ảnh (batch size)
 - H – chiều cao ảnh
 - W – chiều rộng ảnh
 - C – số kênh màu (channels)
 - Nhưng PyTorch yêu cầu định dạng: (N, C, H, W)
- ⇒ Lúc này, ta cần dùng **transpose** để đổi thứ tự trực.

4.3.2. Cú pháp

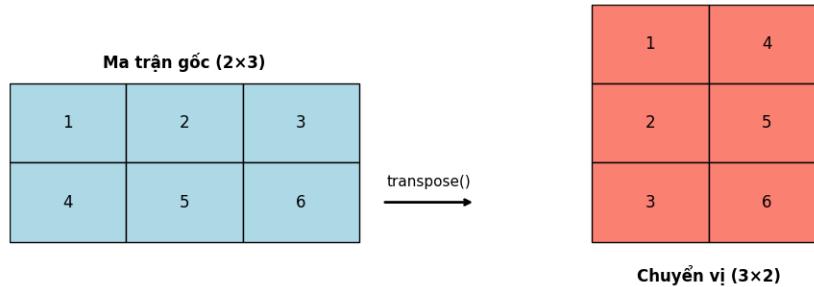
Cú pháp

```
1 np.transpose(a, axes)
  • a là mảng nhiều chiều (tensor).
  • axes là một tuple (hoặc list) chứa thứ tự mới của các trực.
```

Nếu là mảng 2 chiều: Ta có thể dùng cú pháp ngắn gọn hơn là `a.T` — tương tự chuyển vị trong toán học:

```
1 a = np.array([[1, 2, 3],
2                 [4, 5, 6]])
3
4 print('a.T: \n', a.T)
```

```
===== Output =====
a.T:
[[1 4]
 [2 5]
 [3 6]]
```



Minh họa quá trình transpose (chuyển vị) một ma trận 2D

Nếu là mảng nhiều chiều ($\geq 3D$): Khi muốn hoán đổi nhiều trực, hãy liệt kê thứ tự mới theo ý mình:

```
1 a = np.random.rand(10, 28, 28, 3) # (N, H, W, C)
2
3 # Đổi vị (N, C, H, W)
4 a_transposed = np.transpose(a, (0, 3, 1, 2))
```

Giải thích: Mảng ban đầu có 4 trực:

$$(0 \rightarrow N, 1 \rightarrow H, 2 \rightarrow W, 3 \rightarrow C)$$

Sau transpose, trực C (3) được đưa lên vị trí thứ 2 \Rightarrow định dạng mới: (N, C, H, W)

4.4. Repeat, Tile, Concatenate, Stack

Khi làm việc với dữ liệu nhiều chiều (*tensor*), đôi lúc ta cần:

- **Nhân bản phần tử hoặc cả mảng** – để tạo dữ liệu huấn luyện, broadcasting.
- **Ghép nhiều mảng lại** – kết hợp dữ liệu từ nhiều nguồn, ghép batch, v.v.

4.4.1. `np.repeat()` vs `np.tile()`

Hàm	Mục đích	Kiểu lặp
repeat	Lặp từng phần tử theo một chiều	<i>element-wise</i>
tile	Lặp toàn bộ mảng theo pattern	<i>block-wise</i>

Cú pháp

```
1 np.repeat(a, repeats, axis=None)
2 np.tile(a, reps)
```

Trong đó:

- **a** — Mảng gốc (`np.array`) cần được lặp lại.
- **repeats** — Số lần lặp:
 - Nếu là một `int`, thì tất cả phần tử đều được lặp cùng số lần.
 - Nếu là một mảng (list), có thể chỉ định số lần lặp khác nhau cho từng phần tử (áp dụng theo trực).
- **axis** — Trục để thực hiện lặp:
 - Nếu `None` (mặc định), mảng sẽ được làm phẳng trước rồi mới lặp.
 - Nếu xác định rõ trực (`axis=0, axis=1, ...`), thì lặp theo chiều cụ thể (ví dụ: hàng hoặc cột).

Ví dụ minh họa:

```
1 a = np.array([1, 2, 3])
2 print('a repeat:', np.repeat(a, 2))
3 print('a tile:', np.tile(a, 2))
```

```
===== Output =====
a repeat: [1 1 2 2 3 3]
a tile: [1 2 3 1 2 3]
```

Giải thích:

- `repeat`: mỗi phần tử được nhân bản riêng lẻ.
- `tile`: nhân bản toàn bộ mảng theo mẫu.

data	repeat_axis_0
4	4
7	3

repeat_axis_1
4
4
3
3
7
7
1
1

Hình 7: Minh họa repeat theo từng chiều của một Array

4.4.2. Stack và Concatenate

NumPy cung cấp nhiều cách để **ghép các mảng lại với nhau**:

Hàm	Mô tả
np.concatenate	Ghép theo chiều đã có
np.stack	Ghép và tạo thêm chiều mới
np.hstack	Ghép ngang (horizontal), tương đương axis=1
np.vstack	Ghép dọc (vertical), tương đương axis=0

np.concatenate() – Ghép theo chiều có sẵn

```
1 a = np.array([[1, 2], [3, 4]])
2 b = np.array([[5, 6]])
3 print('Concatenate:\n', np.concatenate([a, b], axis=0))
```

```
===== Output =====
Concatenate:
[[1 2]
 [3 4]
 [5 6]]
```

np.stack() – Ghép và thêm chiều mới

```
1 a = np.array([1, 2, 3])
2 b = np.array([4, 5, 6])
3
4 print('Batch stack:\n', np.stack([a, b], axis=0))
5 print('Column stack:\n', np.stack([a, b], axis=1))
```

```
===== Output =====
Batch stack:
[[1 2 3]
 [4 5 6]]
Column stack:
[[1 4]
 [2 5]
 [3 6]]
```

- concatenate không tạo chiều mới, cần các mảng cùng shape ở chiều không ghép.
- stack tạo thêm chiều mới, phù hợp khi gom mảng thành batch.

4.4.3. vstack, hstack

```
1 a = np.array([1, 2])
2 b = np.array([3, 4])
3
4 print('vstack:\n', np.vstack([a, b]))
5 print('hstack:\n', np.hstack([a, b]))
```

```
===== Output =====
vstack:
[[1 2]
 [3 4]]
hstack:
[1 2 3 4]
```

Tình huống	Hàm nên dùng
Ghép mảng cùng shape theo 1 chiều	concatenate
Ghép mảng và thêm trực mới	stack
Ghép mảng 1D/2D đơn giản	vstack, hstack

4.5. Sorting — Sắp xếp dữ liệu trong NumPy

Trong xử lý dữ liệu và học máy (*machine learning*), **sắp xếp (sorting)** là thao tác rất phổ biến: từ sắp xếp điểm số, chọn **top-k phần tử quan trọng nhất**, đến việc sort tensor đầu ra để tính **accuracy**.

NumPy cung cấp công cụ sắp xếp cực kỳ mạnh mẽ, hỗ trợ tốt cho cả mảng nhiều chiều.

4.5.1. Cú pháp cơ bản

NumPy hỗ trợ 2 phương thức sắp xếp chính:

`np.sort(a)` – Trả về bản sao đã sắp xếp

- Trả về **một** bản sao đã được sắp xếp.
- **Không** làm thay đổi mảng gốc.

```

1 import numpy as np
2
3 a = np.array([3, 1, 4, 2])
4 b = np.sort(a)
5
6 print("a:", a)  # [3 1 4 2]
7 print("b:", b)  # [1 2 3 4]
```

`a.sort()` – Sắp xếp in-place

- **Sắp xếp trực tiếp** trên mảng gốc (in-place).
- **Không** tạo ra bản sao.

```

1 a = np.array([3, 1, 4, 2])
2 a.sort()
3 print(a)  # [1 2 3 4]
```

Ghi nhớ

`np.sort(a)` phù hợp khi bạn muốn giữ nguyên mảng gốc.

`a.sort()` dùng khi bạn muốn tiết kiệm bộ nhớ và không cần giữ lại mảng ban đầu.

4.5.2. Vì sao không có `reverse=True` như trong `list.sort()`?

Với list Python:

```

1 a = [3, 1, 4, 2]
2 a.sort(reverse=True)  # [4, 3, 2, 1]
```

Với NumPy: không hỗ trợ `reverse=True`.

Thiết kế cho mảng nhiều chiều

Trong mảng 2D, 3D trở lên, việc "đảo ngược" cần xác định rõ theo chiều nào.
⇒ NumPy tách biệt rõ thao tác `sort` và `reverse`.

Cách đảo ngược thứ tự sau khi sắp xếp:

```
1 a = np.array([3, 1, 4, 2])
2 print(np.sort(a)[:-1]) # [4 3 2 1]
```

Ghi nhớ

Dùng slicing `[:-1]` để đảo ngược thứ tự sau khi sort.
NumPy không dùng `reverse=True` để đảm bảo tính tổng quát.

4.5.3. Ứng dụng Sorting trong AI và xử lý dữ liệu

Chọn top-k phần tử

```
1 a = np.array([0.1, 0.5, 0.3, 0.9])
2 top_k = np.sort(a)[-2:] # [0.5, 0.9]
```

Dùng để chọn các nhãn có xác suất cao nhất (sau softmax).

Sắp xếp toàn bộ ảnh

```
1 img = np.random.rand(28, 28)
2 sorted_pixels = np.sort(img, axis=None) # ắp xếp toàn ảnh (flatten)
```

Dùng khi muốn chuẩn hóa, tính quantile, trực quan hóa histogram.

4.5.4. Tổng kết

Hàm	In-place	Trả bản sao	Đa chiều	Ghi chú
<code>a.sort()</code>	✓		✓	Thay đổi mảng gốc
<code>np.sort(a)</code>		✓	✓	Trả về bản sao đã sắp xếp
<code>[:-1]</code>		✓	✓	Đảo ngược thứ tự sau sort

Phần 5: View vs Copy – Hiểu đúng để tránh bug

Giới thiệu

Trong NumPy, chúng ta có thể dễ dàng viết vài dòng code tưởng vô hại mà vô tình phá nát dữ liệu gốc, dẫn đến lỗi rất khó debug. Lý do? Bạn **nghĩ** mình đang làm việc trên một bản sao, nhưng hóa ra lại là một “cái nhìn” (*view*) tối mảng gốc.

Hiểu **view** và **copy** không chỉ giúp ta tránh được bug rình rập, mà còn tối ưu hóa hiệu suất và tránh cấp phát bộ nhớ không cần thiết.

5.1. View là gì?

View là một đối tượng `ndarray` không cấp phát vùng nhớ mới, mà chỉ trỏ (reference) tới vùng nhớ gốc. Mọi thao tác thay đổi giá trị trên view đều ảnh hưởng tới dữ liệu ban đầu.

Ở tầng bộ nhớ:

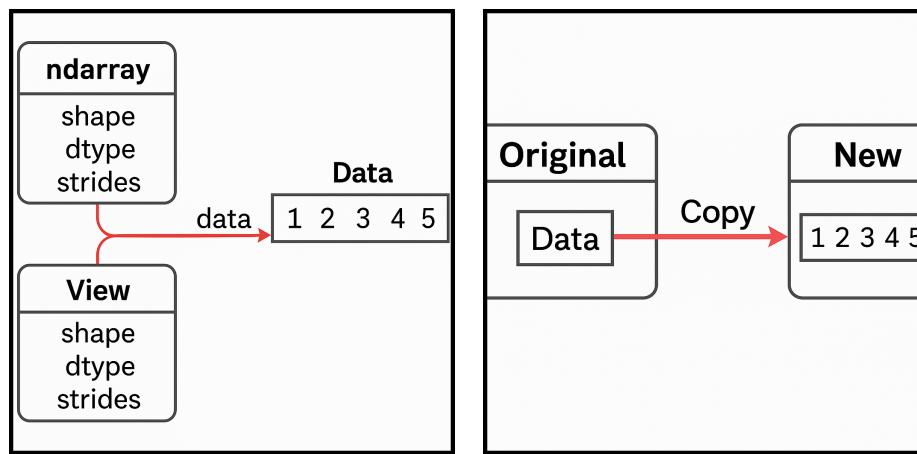
- Mỗi `ndarray` gồm:
 - `data`: con trỏ tới vùng nhớ chứa dữ liệu.
 - `shape`, `dtype`, `strides`: mô tả cách truy cập data.
- Khi tạo view:
 - `data` giữ nguyên.
 - `shape`, `strides` có thể khác (ví dụ: `reshape`, `slicing`).
 - Không copy bất kỳ bit dữ liệu nào.

5.2. Copy là gì?

Copy là một mảng mới, có vùng nhớ riêng độc lập. Mọi thay đổi trên bản copy **không ảnh hưởng** đến mảng gốc.

Ở tầng bộ nhớ:

- Cấp phát vùng nhớ mới cho `data`.
- Copy từng phần tử từ mảng gốc sang.
- `data` là địa chỉ khác hoàn toàn so với mảng ban đầu.



So sánh giữa **View** (trái) và **Copy** (phải) trong NumPy. View dùng chung vùng nhớ với mảng gốc thông qua cùng con trỏ data, trong khi Copy tạo vùng nhớ mới và độc lập.

5.3. Khi nào tạo ra View? Khi nào tạo ra Copy?

Trong NumPy, nhiều thao tác tưởng chừng đơn giản lại tạo ra **view**, còn những thao tác khác lại âm thầm tạo ra **copy**. Việc hiểu rõ điều này giúp tránh bug và kiểm soát bộ nhớ hiệu quả hơn.

Cách truy cập	View/Copy	Giải thích chi tiết
<code>a[1:5]</code> (slicing)	View	Tạo ra view vì chỉ thay đổi cách truy cập thông qua strides , không tạo vùng nhớ mới. Rất hiệu quả nhưng dễ gây bug nếu vô tình thay đổi dữ liệu gốc.
<code>a[[1, 2, 3]]</code> (fancy indexing)	Copy	Fancy indexing buộc phải tạo ra một mảng mới chứa dữ liệu được sao chép từ các vị trí chỉ định → là copy. Tốn bộ nhớ hơn nhưng an toàn.
<code>a[a > 0]</code> (boolean masking)	Copy	Cũng là một dạng fancy indexing: chọn các phần tử theo điều kiện logic → NumPy không thể dựa vào strides , nên phải tạo copy.
<code>a.T</code> (transpose)	View	Transpose chỉ thay đổi thứ tự truy cập chiều → thay đổi strides , giữ nguyên data → là view. Nhưng với mảng không liên tục trong bộ nhớ, có thể sẽ bị copy ngầm khi tính toán.
<code>a.reshape(...)</code>	View (nếu được)	Nếu memory layout còn phù hợp (liên tục và đúng strides) thì trả về view. Nếu không, NumPy sẽ ngầm tạo copy → cần dùng <code>np.shares_memory()</code> để kiểm tra.
<code>a.copy()</code>	Copy	Ép buộc tạo bản sao độc lập hoàn toàn khỏi dữ liệu gốc. Rất hữu ích khi cần chỉnh sửa mà không làm ảnh hưởng array gốc.

Ghi chú:

- Nếu không chắc **view** hay **copy**, hãy dùng: `np.shares_memory(a, b)` để kiểm tra xem hai array có dùng chung vùng nhớ hay không.
- Ngoài ra, ta cũng có thể dùng: `b.base is a`
Trả về `True` nếu `b` là **view** của `a`.

5.4. Một số ví dụ minh họa

1. Slicing → View

```

1 import numpy as np
2
3 a = np.array([1, 2, 3, 4, 5])
4 b = a[1:4]
5 b[0] = 999
6
7 print('a:', a)
8 print('b:', b)

```

```
=====
Output =====
a: [ 1 999 3 4 5]
b: [999 3 4]
```

Phân tích: `b` là **view** → dùng chung data với `a`.

2. Fancy Indexing → Copy

```

1 a = np.array([1, 2, 3, 4, 5])
2 b = a[[1, 2, 3]]
3 b[0] = 999
4
5 print('a:', a)
6 print('b:', b)

```

```
=====
Output =====
a: [1 2 3 4 5]
b: [999 3 4]
```

Giải thích: `b` sống độc lập, không ảnh hưởng `a` vì là **copy**.

Khi muốn làm việc độc lập với mảng gốc, hãy dùng `.copy()`.

5.5. Kiểm tra bằng `np.shares_memory()`

Trong NumPy, việc một mảng là **view** hay **copy** không phải lúc nào cũng dễ phân biệt chỉ bằng mắt thường. Một cách hiệu quả và đáng tin cậy để kiểm tra là dùng hàm `np.shares_memory()`.

Cú pháp:

```
1 np.shares_memory(x, y)
```

Hàm này sẽ trả về:

- `True` nếu `x` và `y` dùng chung bộ nhớ (có vùng đè nhau).
- `False` nếu `x` và `y` là hai mảng độc lập (`copy`).

Ví dụ minh họa:

```

1 import numpy as np
2
3 a = np.arange(10)
4 b = a[2:5]      # slicing → view
5 c = a[[2, 3, 4]] # fancy indexing → copy
6
7 print(np.shares_memory(a, b)) # True → b is view
8 print(np.shares_memory(a, c)) # False → c is copy

```

Phân tích:

- `b = a[2:5]` là slicing → chỉ thay đổi cách truy cập, không cấp phát mới → `np.shares_memory(a, b)` trả về True.
- `c = a[[2, 3, 4]]` dùng fancy indexing → NumPy buộc phải cấp phát vùng nhớ mới → `np.shares_memory(a, c)` trả về False.

Lưu ý:

- `np.shares_memory()` rất hữu ích để kiểm tra trong các thao tác phức tạp như `reshape()`, `transpose()`, slicing nhiều tầng, hoặc các tình huống chúng ta nghi ngờ bị aliasing bộ nhớ.
- Kết quả phụ thuộc vào việc hai mảng có vùng bộ nhớ bị chồng nhau hay không (không nhất thiết phải là cùng toàn bộ).

Ngoài lề: Nếu chúng ta chỉ muốn biết một mảng có phải là view của một mảng cụ thể không (thay vì kiểm tra bộ nhớ chung), ta có thể dùng:

```

1 b.base is a
→ Trả về True nếu b là view được tạo từ a.

```

5.6. Fancy Indexing – Kẻ thù thầm lặng

Fancy indexing (chỉ mục hóa nâng cao) là kỹ thuật truy cập phần tử của mảng bằng một danh sách (array) chỉ số – thay vì một lát cắt (slice). Tuy nhiên, đây cũng là **nguồn gốc của rất nhiều bug âm thầm** trong NumPy, bởi nó tạo ra một **bản sao (copy)** thay vì **view** như slicing.

Ví dụ:

```

1 import numpy as np
2
3 a = np.array([10, 20, 30, 40, 50])
4 b = a[[1, 2]]      # Fancy indexing → COPY
5 b[0] = 999
6
7 print('a:', a)
8 print('b:', b)

```

```

=====
Output =====
a: [10 20 30 40 50]    KHÔNG đổi!
b: [999 30]

```

Giải thích:

- Dòng `b = a[[1, 2]]` sử dụng **fancy indexing** (chỉ định index bằng mảng).

- Kết quả: NumPy buộc phải **tạo mảng mới** (copy) chứa các giá trị tại chỉ số 1 và 2.
- Vì vậy, b sống độc lập hoàn toàn với a thay đổi b| KHÔNG làm thay đổi a|.
- Nếu ta nhầm tưởng đây là view, ta sẽ rơi vào những bug cực kỳ khó debug trong các đoạn xử lý dữ liệu lớn.

Cách phát hiện: Chúng ta có thể kiểm tra bằng cách dùng:

```
1 np.shares_memory(a, b)      # False    copy
```

Fancy indexing luôn trả về copy. Vì vậy, hãy:

- Chỉ dùng khi thật sự cần bản sao độc lập.
- Tránh nhầm lẫn với slicing (view).
- Luôn dùng np.shares_memory() hoặc .base is ... khi cần chắc chắn.

5.7. View để tiết kiệm bộ nhớ

Trong thực tế, các mảng dữ liệu NumPy có thể rất lớn — đặc biệt trong các bài toán xử lý ảnh, AI, dữ liệu tín hiệu, hoặc mô phỏng khoa học. Việc tạo view thay vì copy giúp tiết kiệm một lượng lớn bộ nhớ RAM và tăng hiệu năng đáng kể.

Ví dụ:

```
1 import numpy as np
2
3 a = np.random.rand(10000, 10000)  # ~800MB if dtype=float64
4 b = a[:, :100]                  # View on top 100 columns
```

Phân tích:

- Biến a là một mảng 100 triệu số thực 64-bit → khoảng 800MB bộ nhớ.
- Khi tạo b = a[:, :100], đây là một **view**:
 - Không có dữ liệu nào được sao chép.
 - b chỉ thay đổi **shape** và **strides** để nhìn vào 100 cột đầu của a.
- Kết quả: bộ nhớ không tăng, tốc độ xử lý cao, và rất tiết kiệm tài nguyên.

Lợi ích của view trong bài toán lớn:

- **Tiết kiệm RAM:** Không cần nhân đôi dữ liệu khi chỉ cần thao tác một phần nhỏ.
- **Hiệu năng cao:** Truy cập nhanh, không tốn thời gian copy dữ liệu.
- **Thân thiện cache CPU:** Truy cập dữ liệu tuyến tính, tránh cache miss.

Cẩn trọng khi dùng:

- Mọi thay đổi trên view sẽ làm thay đổi mảng gốc.

- Nếu mảng gốc bị xóa (hoặc bị thay đổi), `view` có thể trở nên không hợp lệ.
- Khi cần thao tác độc lập, luôn dùng `.copy()` để tránh bug không rõ ràng.

- Trong các hệ thống xử lý ảnh/video (OpenCV, PyTorch), `view` thường được tận dụng để tăng tốc đáng kể.
- Với mảng lớn, việc lỡ tạo copy có thể làm tăng RAM từ vài trăm MB lên hàng GB dễ crash nếu không kiểm soát.

5.8. Benchmark: View vs Copy trong thực tế

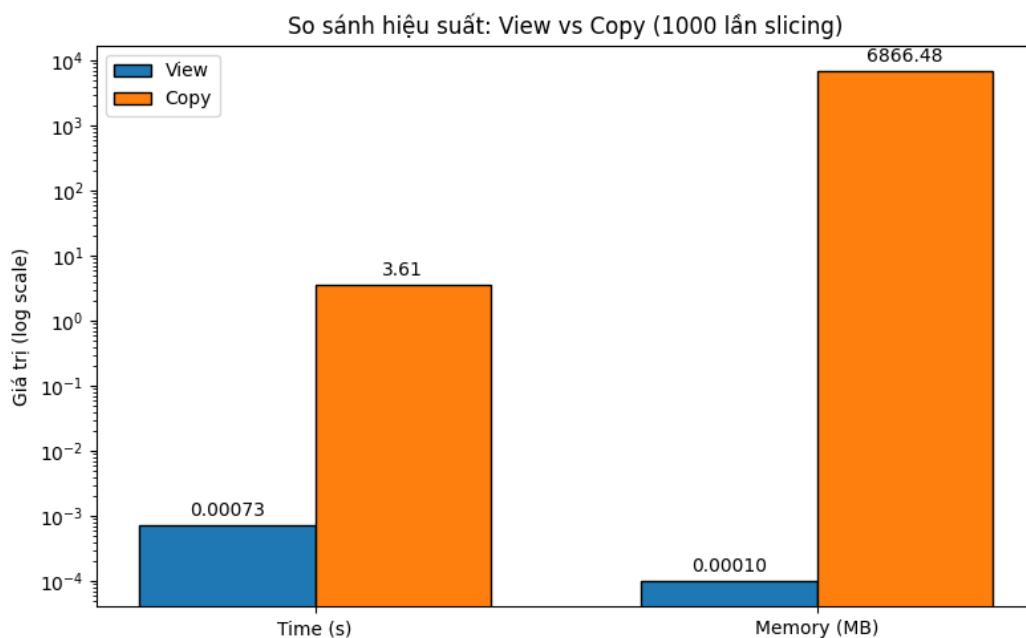
Để đánh giá hiệu suất khi làm việc với mảng lớn trong NumPy, ta thực hiện benchmark với một mảng có kích thước `a.shape = (10000, 10000)`, sau đó lặp lại thao tác slicing **1000 lần** theo hai cách:

- **View:** `a[:, :100]` — tạo view (không sao chép dữ liệu)
- **Copy:** `a[:, :100].copy()` — tạo bản sao độc lập (copy toàn bộ slice)

Sau mỗi thao tác, ta đo thời gian thực thi (tính bằng giây) và mức tiêu thụ bộ nhớ (tính bằng MB). Kết quả như sau:

- **View:** mất 0.00073s, tăng thêm chỉ 0.0001 MB RAM
- **Copy:** mất tối 3.61s, tăng đến 6866.48 MB RAM

Biểu đồ log-scale dưới đây cho thấy chênh lệch rõ rệt về hiệu suất giữa View và Copy:



Nhận xét:

- View nhanh hơn **gần 5000 lần** và tiết kiệm gần như toàn bộ bộ nhớ.
- Copy tiêu tốn tài nguyên khủng khiếp khi xử lý lượng lớn slice.
- Nếu bạn vô tình dùng **fancy indexing** hay quên dùng `.copy()` đúng lúc, hệ thống của bạn có thể bị nghẽn tài nguyên hoặc chạy cực kỳ chậm.

Khi làm việc với dữ liệu lớn (hình ảnh, tín hiệu, mảng số liệu,...), hãy ưu tiên View để tránh nhân đôi dữ liệu. Chỉ sử dụng Copy khi thực sự cần thao tác độc lập và có kiểm soát.

Tóm lại, chúng ta có:

Trường hợp	View	Copy	Ghi chú
Slicing	✓		Khi muốn độc lập → <code>.copy()</code>
Fancy indexing		✓	Luôn tạo bản sao
Reshape	✓(tùy)	Có thể	Phụ thuộc layout
Transpose	✓		Không cần copy
<code>.copy()</code>		✓	Ép tạo bản sao

Phần 6: Vectorization & Broadcasting

6.1. Broadcasting

Thuật ngữ broadcasting mô tả cách NumPy xử lý các mảng có hình dạng khác nhau trong các phép toán số học. Dưới một số điều kiện nhất định, mảng nhỏ hơn sẽ được “broadcast” qua mảng lớn hơn để chúng có hình dạng tương thích. Broadcasting cung cấp một phương thức để vector hóa các phép toán trên mảng, giúp bạn “bớt viết vòng lặp” hơn, từ đó tối ưu code và giảm bớt bug.

Trong trường hợp đơn giản nhất, hai mảng phải có cùng kích thước để thực hiện phép toán với các phần tử của mảng:

```

1 import numpy as np
2
3 a = np.array([1.0, 2.0, 3.0])
4 b = np.array([2.0, 2.0, 2.0])
5 result = a * b

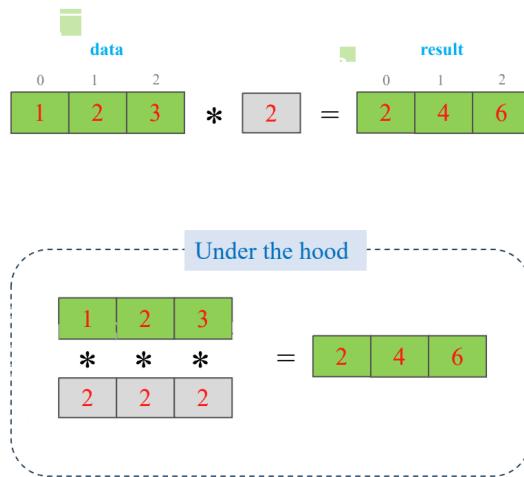
```

Broadcasting cũng cho phép tính toán từng phần tử giữa các mảng và các giá trị vô hướng (*Giải thích kỹ ở phần quy tắc phía sau*). Các giá trị vô hướng được tự động mở rộng để phù hợp với kích thước của mảng. Ví dụ:

```

1 import numpy as np
2 a = np.array([1, 2, 3])
3 b = 2
4 a * b
5 # output [2 4 6]

```



Quy tắc broadcasting tổng quát

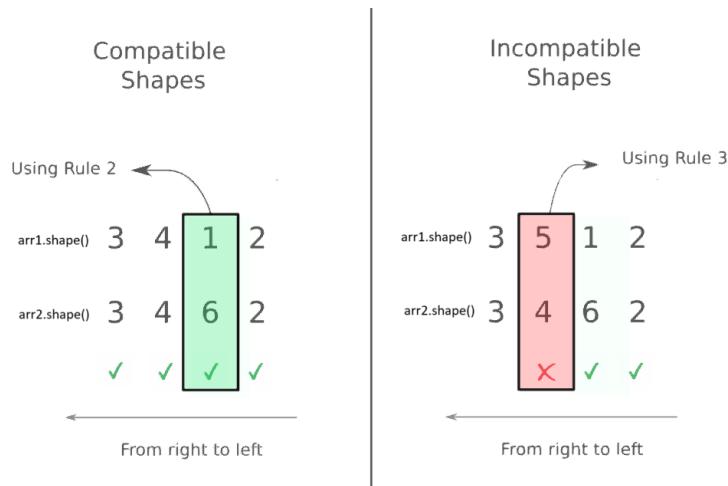
Trước khi bắt đầu, một định nghĩa quan trọng mà chúng ta cần phải biết đó là thứ hạng(rank) của mảng trong numpy. Thứ hạng(rank) là tổng số chiều mà một mảng Numpy có, ví dụ: $b.shape()$ là $(1, 6)$ có rank là 2, $b.shape()$ là $(3, 4, 3)$ có rank là 3.

Để xác định hai mảng nào phù hợp cho các phép toán, NumPy so sánh hình dạng của hai mảng theo từng chiều bắt đầu từ các chiều sau của mảng theo hướng **tiền về phía trước**. (từ phải sang trái):

- Nếu hai chiều có cùng kích thước, thì chúng tương thích.
- Một trong hai chiều có kích thước bằng 1, chúng cũng tương thích.
- Nếu cả hai chiều không bằng nhau và không có chiều nào bằng 1 thì NumPy sẽ báo lỗi và dừng lại.

Mảng có thứ hạng bằng nhau

Với 2 mảng có thứ hạng bằng nhau, chúng ta thực hiện xét từ phải qua trái và thực hiện so sánh các chiều của chúng. Ở phía bên trái, Numpy so sánh số chiều từ phải qua trái của 2 mảng và thấy rằng chúng thỏa mãn quy tắc 1 và $2 \Rightarrow \text{arr1}$ và arr2 tương thích, và có thể thực hiện tính toán. Ở phía bên phải, Numpy cũng thực hiện so sánh tương tự, nhưng phát hiện ra rằng đã vi phạm quy tắc số 3 $\Rightarrow \text{arr1}$ và arr2 không tương thích, khi thực hiện phép toán sẽ cho ra lỗi ValueError.



- Quy tắc 1

```

1 import numpy as np
2
3 a = np.array([[1, 2, 3],
4                 [4, 5, 6]])      # shape (2, 3)
5 b = np.array([[10, 20, 30],
6                 [40, 50, 60]]) # shape (2, 3)
7
8 result = a + b
9 print(result)

```

- Quy tắc 2

```

1 a = np.array([[1, 2, 3],
2                 [4, 5, 6]])      # shape (2, 3)
3 b = np.array([[10], 
4                 [20]])        # shape (2, 1)
5
6 result = a + b
7 print(result)

```

- Quy tắc 3

```

1 a = np.array([[1, 2, 3],
2                 [4, 5, 6]])      # shape (2, 3)
3 b = np.array([[10, 20],
4                 [30, 40]])      # shape (2, 2)
5
6 result = a + b    #ValueError

```

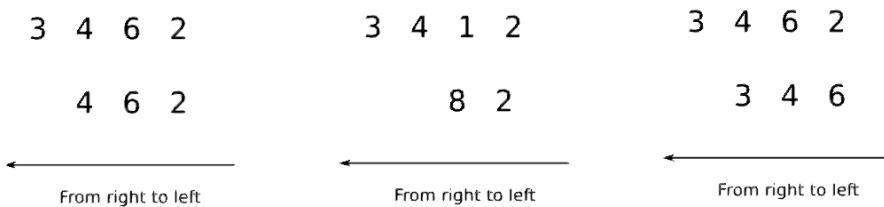
Mảng khác thứ hạng

Các mảng có thứ hạng không bằng nhau cũng có thể được vận hành theo một số điều kiện nhất định. Một lần nữa, Numpy áp dụng quy tắc di chuyển từ **phải sang trái** và so sánh hai mảng. Broadcasting chỉ xảy ra *nếu và chỉ nếu* từng chiều tương ứng thỏa mãn một trong hai điều kiện sau:

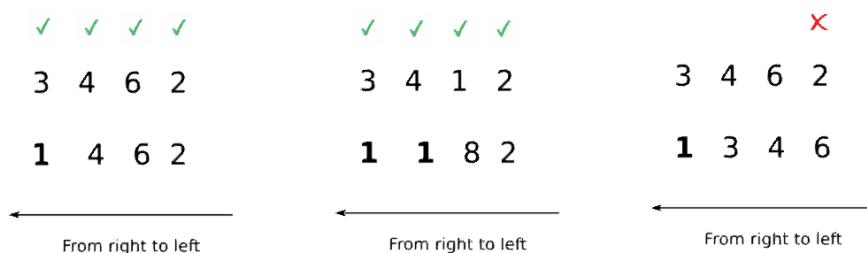
- Hai kích thước bằng nhau, hoặc
 - Một trong hai chiều có kích thước bằng 1
- Nếu **bất kì chiều nào** không thỏa, toàn bộ phép toán sẽ lỗi.

Điều này cũng tương tự như việc so sánh mảng có thứ hạng bằng nhau, chỉ khác ở chỗ Numpy tự thêm chiều 1 vào đầu mảng có shape ngắn hơn.

Hãy xem xét các ví dụ sau:



Trong hình ảnh đầu tiên, mảng đầu tiên có thứ hạng 4, trong khi mảng thứ 2 có thứ hạng 3. Để so sánh hai mảng như vậy, Numpy sẽ thêm các chiều có kích thước 1 vào mảng có shape ngắn hơn để thực hiện tính toán. Khi đó, nó được hiểu như thế này : Ở từng chiều được so sánh từ phải qua trái, cả hình 1 và hình 2, các chiều đều thỏa mãn một trong hai điều kiện trên. Tuy nhiên, đối với hình 3, các chiều đều không thỏa mãn, sinh ra lỗi. (Lưu ý rằng tôi sử dụng chữ đậm để thêm vào vì đây chỉ là cách để hình dung NumPy đang làm gì. Về mặt nội bộ, không có thêm vào)



Lưu ý: Quá trình này **không sinh ra mảng mới** đầy đủ trong RAM — NumPy chỉ trả về một view “ảo” trỏ đến dữ liệu gốc, giữ hiệu năng cao.

Giờ thì chắc hẳn bạn đã hiểu rõ vì sao numpy có thể thực hiện tính toán giữa mảng và scalar một cách mượt mà như vậy rồi đúng không?

```

1 import numpy as np
2 a = np.array([1, 2, 3])
3 b = 2
4 a * b
5 # output [2 4 6]

```

Một số ví dụ thực tế

Thêm màu vào một hình ảnh

giả sử bạn có một bức ảnh màu với 3 kênh (R,G,B). Bạn muốn với mỗi pixel tăng giá trị màu đỏ lên 10, màu xanh lá cây lên 5 và màu xanh lam lên 15. Điều này có thể dễ dàng thực hiện bằng broadcasting.

Như đã biết, một hình ảnh màu được biểu diễn dưới dạng ma trận (H,W, C) nếu chúng ta đọc chúng bằng opencv.

```

1 import numpy as np
2 import cv2
3
4 img = cv2.imread("image1.jpeg")
5 print(img.shape)
6 #output -> (768,1024,3)

```

Tiếp theo, chúng ta sẽ lần lượt thêm các màu vào các kênh màu tương ứng

```

1
2 add_color = np.array([10, 5, 15], dtype=np.uint8)
3 result = img + add_color

```

Hình dung các vòng lặp như một mảng

Giả sử ta có hai mảng [1,2,3] và [4,5]. Ta muốn lấy tổng của tích mọi phần tử của 2 mảng. Nếu không biết về broadcast, chúng ta nghĩ ngay đến vòng lặp đúng không nào?

```

1 import numpy as np
2
3 arr1 = np.array([1,2,3])
4 arr2 = np.array([4,5])
5
6 sum = 0
7 for i in arr1:
8     for j in arr2:
9         sum += i*j
10 print(sum)

```

Cũng không khó lắm đúng không? Nhưng đây là một vòng lặp lồng nhau và thử nghĩ xem, nếu kích thước của các mảng này trở lên quá lớn thì thời gian duyệt vòng lặp cũng tăng lên. Đó là khi broadcast phát huy tác dụng.

```

1 import numpy as np
2
3 arr1 = np.array([1,2,3])
4 arr2 = np.array([4,5])
5
6 result = arr1*arr2
7 sum = np.sum(result)
8 print(sum)

```

6.2. Vectorization

Vectorization là kỹ thuật “thao tác” toàn bộ mảng cùng lúc thay vì lặp qua từng phần tử bằng vòng lặp Python

Thay vì viết:

```
1 import numpy as np
2 def myfunc(a, b):
3     "Return a-b if a>b, otherwise return a+b"
4     if a > b:
5         return a - b
6     else:
7         return a + b
8
9 a_list = [1, 2, 3, 4]
10 b_value = 2
11
12 result = []
13 for a in a_list:
14     result.append(myfunc(a, b_value))
15
16 result = np.array(result)
17 print(result) # [3 4 1 2]
```

bạn dùng ngay:

```
1 import numpy as np
2 def myfunc(a, b):
3     "Return a-b if a>b, otherwise return a+b"
4     if a > b:
5         return a - b
6     else:
7         return a + b
8 vfunc = np.vectorize(myfunc)
9 vfunc([1, 2, 3, 4], 2)
10 print(vfunc) # [3,4,1,2]
```

Lưu ý: Hàm `np.vectorize()` chủ yếu được cung cấp để tiện sử dụng, chứ không nhằm mục đích tăng hiệu suất. Về bản chất, việc thực thi vẫn là một vòng lặp `for`

Phần 7: AI Applications

1. Brightness Changes – Làm sáng / tối ảnh bằng NumPy

Trong xử lý ảnh với Python, **làm sáng hoặc làm tối ảnh** là thao tác cực kỳ phổ biến — ví dụ như khi muốn tăng độ sáng ảnh trong AI, xử lý camera, hoặc đơn giản là “tăng độ đẹp trai” cho ảnh chân dung.

1.1. Ảnh số và NumPy

Một ảnh grayscale (đen trắng) được lưu trong NumPy như một ma trận 2D. Mỗi phần tử là 1 pixel, giá trị từ 0 đến 255:

- 0 = đen hoàn toàn
- 255 = trắng hoàn toàn
- các giá trị ở giữa là mức xám

```
1 import numpy as np
2
3 img = np.array([[100, 150],
4 [200, 250]], dtype=np.uint8)
```

`dtype=np.uint8` có nghĩa là:

- **unsigned**: chỉ có số dương
- **integer**: số nguyên
- **8-bit**: giá trị từ 0 đến 255

1.2. Tăng/giảm độ sáng ảnh (brightness changes)

Ý tưởng: Cộng/trừ một giá trị v vào mỗi pixel.

```
1 img = img + v # cong
2 img = img - v # tru
```

Lưu ý

Nếu không cẩn thận, sẽ xảy ra tràn số:

```
1 np.uint8(250) + 10 # bi tran!
```

Trong NumPy, ảnh dạng uint8 rất dễ bị **tràn số** khi cộng/trừ. Dưới đây là 3 cách xử lý thông minh để tránh lỗi:

Cách 1: Ép kiểu sang float, dùng clip

```

1 image = cv2.imread('image1.png', 1)
2 image = image.astype(float)
3
4 image = image + 100
5 image = np.clip(image, 0, 255)
6 image = image.astype(np.uint8)

```

Ưu điểm: An toàn tuyệt đối, dễ viết.

Nhược điểm: Tốn thêm bước ép kiểu, chiếm nhiều bộ nhớ hơn.

Cách 2: Dùng np.where

```

1 image = cv2.imread('image1.png', 1)
2 image = image.astype(float)
3
4 image = image + 100
5 image = np.where(image>255, 255, image)
6 image = image.astype(np.uint8)

```

Ưu điểm: Rất linh hoạt — có thể điều chỉnh từng vùng.

Nhược điểm: Cần hiểu rõ broadcasting và điều kiện.

Cách 3: Viết hàm tự định nghĩa dùng boolean mask

Phù hợp nếu bạn muốn áp dụng **có điều kiện** — ví dụ: chỉ làm sáng các pixel tối hơn 200.

```

1 def smart_adjust(img, delta=50, threshold=205, mode='lt'):
2     """
3         Add or subtract a delta value to pixels that satisfy a given threshold condition,
4         while preventing uint8 overflow.
5
6     Parameters:
7         img      : ndarray
8             Original image (dtype: uint8).
9         delta   : int
10            Value to add or subtract (can be negative).
11         threshold : int
12            Pixel intensity threshold for conditional modification.
13         mode    : str
14            Comparison operator to apply. Must be one of:
15            'lt' (<), 'le' (), 'gt' (>), 'ge' (), 'eq' (==), 'ne' (!=).
16
17     Returns:
18         new_img : ndarray
19             Image after brightness adjustment (dtype: uint8).
20     """
21     img = img.copy() # avoid modifying the original image
22
23     # Create a boolean mask based on the selected comparison mode
24     ops = {
25         'lt': img < threshold,
26         'le': img <= threshold,
27         'gt': img > threshold,
28         'ge': img >= threshold,

```

```

29     'eq': img == threshold,
30     'ne': img != threshold
31 }
32 mask = ops[mode]
33
34 # Convert to int16 to safely handle overflow during arithmetic
35 tmp = img.astype(np.int16)
36
37 # Apply delta only where the mask is True
38 tmp[mask] += delta
39
40 # Clip values to [0, 255] and convert back to uint8
41 tmp = np.clip(tmp, 0, 255).astype(np.uint8)
42
43 return tmp

```

Ưu điểm:

- Cho phép kiểm soát chi tiết từng vùng ảnh.
- Có thể kết hợp nhiều điều kiện nâng cao.

Nhược điểm:

- Phải ép kiểu về int16 nếu không sẽ gây tràn khi dùng với uint8.
- Dễ sai nếu quên clip() hoặc ép về uint8.

Tóm tắt

Phương pháp	Ưu điểm	Nhược điểm
Ép sang float + clip	An toàn, đơn giản	Tốn RAM, cần ép kiểu
np.where	Linh hoạt, vector hoá	Cần viết điều kiện cẩn thận
boolean mask	Tùy biến cao, có điều kiện	Phải ép kiểu tạm, dễ sai nếu quên clip()

1.3. Vì sao phải dùng clip() và uint8?

Nếu không clip:

- Giá trị có thể vượt quá 255 hoặc nhỏ hơn 0.
- Gây lỗi tràn như đồng hồ (overflow).

```

1 np.uint8(250 + 10)
2 # bi vong lai!

```

✓ Dùng clip() giúp:

Giữ mọi giá trị trong khoảng hợp lệ:

```

1 img = np.clip(img, 0, 255)

```

✓ Dùng `.astype(np.uint8)`

Đảm bảo định dạng ảnh đúng 8-bit, cần thiết khi hiển thị với OpenCV hoặc lưu file.

2. Contrast Change – Điều chỉnh độ tương phản ảnh bằng NumPy

2.1. Contrast là gì?

- **Brightness (Độ sáng):** Dịch toàn bộ ảnh lên sáng hơn hoặc tối hơn.
- **Contrast (Độ tương phản):** Làm cho phần sáng sáng hơn, phần tối tối hơn.

Ảnh có độ tương phản cao sẽ nổi bật chi tiết rõ ràng. Ngược lại, nếu độ tương phản thấp, ảnh trở nên mờ nhạt, “xám xịt”.

2.2. Công thức điều chỉnh độ tương phản

Với ảnh grayscale (giá trị pixel từ 0 đến 255), ta điều chỉnh tương phản bằng công thức:

$$I' = \alpha \cdot (I - 128) + 128$$

Trong đó:

- I : ảnh gốc
- α : hệ số tương phản
 - $\alpha > 1$ tăng tương phản
 - $\alpha < 1$ giảm tương phản
- 128: là giá trị xám trung tính (trung tâm của khoảng 0–255)

2.3. Code NumPy cho Contrast Adjustment

```

1 import numpy as np
2
3 def change_contrast(img, alpha):
4     """
5         Adjust contrast of a grayscale image using contrast factor alpha.
6
7     Parameters:
8         img    : ndarray (uint8) - input grayscale image
9         alpha : float - contrast factor (>1: increase, <1: decrease)
10
11    Returns:
12        Adjusted image (uint8)
13    """
14    img = img.astype(np.float32)
15    img = alpha * (img - 128) + 128
16    img = np.clip(img, 0, 255)
17    return img.astype(np.uint8)

```

2.4. Ví dụ thực tế với OpenCV

```

1 import cv2
2
3 img = cv2.imread("gray_image.png", cv2.IMREAD_GRAYSCALE)
4
5 high_contrast = change_contrast(img, 1.5) # tang tuong phan
6 low_contrast = change_contrast(img, 0.5) # giam tuong phan
7
8 cv2.imshow("Original", img)
9 cv2.imshow("High Contrast", high_contrast)
10 cv2.imshow("Low Contrast", low_contrast)
11 cv2.waitKey(0)
12 cv2.destroyAllWindows()

```

2.5. Tại sao phải trừ 128 rồi cộng lại?

Nếu chỉ viết:

```
1 img = alpha * img
```

...thì pixel tối (gần 0) vẫn là số dương sau khi nhân, không thay đổi bản chất tối/sáng.

Giải pháp: dịch ảnh về trung tâm (128), điều chỉnh, rồi dịch ngược lại:

$$I' = \alpha \cdot (I - 128) + 128$$

2.6. Tóm lại

Điều chỉnh	Công thức	Lưu ý
Brightness	$I' = I \pm v$	Dùng np.clip()
Contrast	$I' = \alpha(I - 128) + 128$	$\alpha > 1$: tăng, < 1 : giảm
Sau xử lý	clip & uint8	Luôn cần .astype(np.uint8)

2.7. Gợi ý hệ số tương phản thường dùng

Hệ số α	Ý nghĩa
0.5	Giảm mạnh tương phản
0.8	Giảm nhẹ tương phản
1.0	Không thay đổi
1.2–1.5	Tăng tương phản nhẹ
2.0	Tăng mạnh tương phản

2.8. Vì sao điều này quan trọng?

- Là bước tiền xử lý cơ bản trong học sâu và xử lý ảnh.
- Giúp mô hình học được tốt hơn từ dữ liệu ảnh.
- Là kỹ thuật nền tảng cho augmentation (data tăng cường) trong AI.

Phần 8: Mở rộng - Numpy in Image Data-Preprocessing

1 Giới thiệu

Trong lĩnh vực thị giác máy tính, dữ liệu đầu vào là ảnh đóng vai trò cực kỳ quan trọng. Ảnh y học (CT/MRI) không giống với ảnh RGB thông thường, mà chứa thông tin vật lý, giá trị đo lường và định dạng thể tích (3 chiều). Nếu không tiền xử lý đúng cách, mô hình học sâu có thể gặp lỗi hội tụ, kết quả sai lệch hoặc không học được đặc trưng hữu ích.(Tham khảo: <https://github.com/fitushar/3D-Medical-Imaging-Preprocessing-All-you-need>)

2 Đặc điểm riêng biệt của ảnh y học 3D

- Dữ liệu 3D dạng thể tích (Volume):** mỗi ảnh bao gồm nhiều lát cắt (slice), tạo thành khối voxel.
- Giá trị pixel là vật lý (Hounsfield Unit - HU):** phản ánh mật độ mô, không phải màu sắc như ảnh RGB.
- Kích thước voxel không đồng nhất (spacing khác nhau):** ảnh từ các máy chụp khác nhau có thể có độ phân giải không gian khác nhau.
- Nhiều (noise), artifacts:** do chuyển động, kim loại trong cơ thể, lỗi thiết bị.
- Ảnh lớn, bất đối xứng, vùng quan tâm nhỏ (ROI):** thường cần cắt hoặc đệm để đưa về kích thước chuẩn.

3 Đọc ảnh NIfTI và chuyển sang NumPy

```

1 import SimpleITK as sitk
2
3 img_sitk = sitk.ReadImage("img0001.nii.gz", sitk.sitkFloat32)
4 image = sitk.GetArrayFromImage(img_sitk)

```

Ảnh CT thường ở định dạng .nii.gz. SimpleITK giúp đọc ảnh và giữ lại metadata như spacing, origin, direction. Sau đó, ta chuyển sang dạng np.ndarray để dễ xử lý.

4 Hounsfield Unit (HU) và lý do cần cắt ngưỡng

Trong ảnh CT, mỗi pixel mang một giá trị đo gọi là Hounsfield Unit (HU). Giá trị này phản ánh mức độ hấp thụ tia X của mô:

- Không khí: -1000 HU
- Mỡ: khoảng -100 đến -50 HU
- Mô mềm: 30–60 HU
- Xương: 400 đến > 1000 HU
- Kim loại: > 2000 HU

Các giá trị này có thể trải dài rất rộng, gây khó khăn cho mạng học sâu nếu không xử lý. Do đó, ta dùng hàm `np.clip` để giới hạn ảnh trong một khoảng HU hợp lý.

Ví dụ: cắt ngưỡng HU từ -1000 đến 800

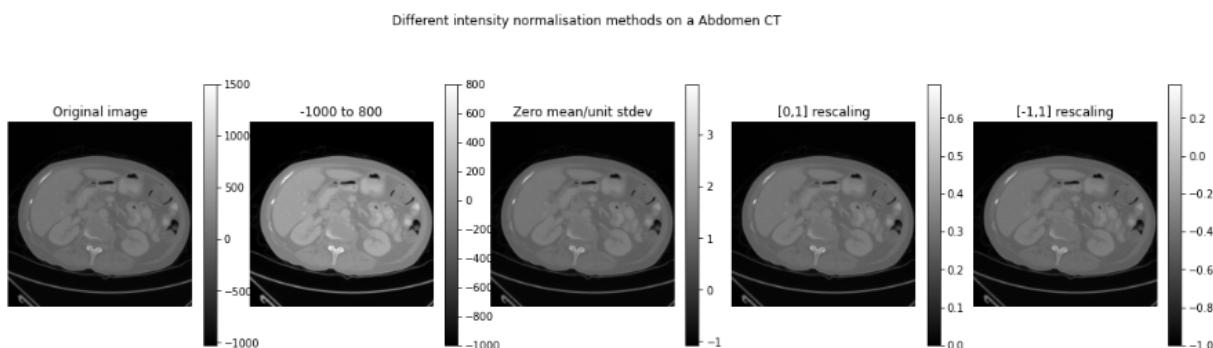
```
1 image = np.clip(image, -1000., 800.).astype(np.float32)
```

Việc cắt ngưỡng giúp loại bỏ các điểm nhiễu (vd: không khí, kim loại) và chỉ giữ lại vùng mô có ý nghĩa lâm sàng. Giá trị cắt có thể thay đổi theo bài toán, ví dụ:

- CT phổi: từ -1000 đến 400 HU
- CT gan: từ -100 đến 250 HU
- CT toàn thân: từ -1000 đến 800 HU

5 Chuẩn hoá ảnh

Sau khi cắt ngưỡng, ta cần đưa ảnh về phạm vi giá trị chuẩn để mô hình dễ học.



Hình 8: Ví dụ chuẩn hóa ảnh về từng khoảng

1 Chuẩn hóa về khoảng $[0, 1]$ – Min-Max Scaling

```
1 def normalize_zero_one(image):
2     return (image - np.min(image)) / (np.max(image) - np.min(image))
```

Lý do sử dụng:

Việc chuẩn hóa ảnh về khoảng $[0, 1]$ giúp đưa tất cả giá trị pixel về cùng một phạm vi, bất kể ảnh gốc có dải giá trị như thế nào. Cách này đặc biệt quan trọng khi:

- Mô hình sử dụng các hàm kích hoạt như `sigmoid`, vốn có đầu ra nằm trong khoảng $[0, 1]$.
- Tránh hiện tượng ảnh quá tối hoặc quá sáng do dải giá trị lớn nhỏ không đồng đều giữa các ảnh.
- Giúp tăng tính nhất quán khi huấn luyện nhiều ảnh từ các nguồn máy chụp khác nhau.

Ảnh hưởng đến gradient: Nếu không chuẩn hóa, đầu vào quá lớn sẽ gây ra gradient rất nhỏ (vanishing gradient), làm cho mạng học chậm hoặc không học được.

2 Chuẩn hoá về khoảng [-1, 1]

```
1 def normalize_one_one(image):
2     return 2.0 * normalize_zero_one(image) - 1.0
```

Lý do sử dụng:

Khi ta chuẩn hoá ảnh về $[-1, 1]$, ta đưa dữ liệu vào trung tâm gốc toạ độ (0), giúp tăng tính đối xứng của phân phối dữ liệu. Điều này rất quan trọng trong các mô hình sử dụng:

- Hàm kích hoạt `tanh`, vốn đầu ra nằm trong khoảng $[-1, 1]$.
- Mạng GAN (Generative Adversarial Networks), nơi generator và discriminator thường hoạt động hiệu quả hơn khi input được phân bố xung quanh 0.
- Các bài toán có dữ liệu cân bằng giữa giá trị âm và dương (vd: ảnh đã được cắt ngưỡng HU về $[-1000, 800]$).

So sánh: Chuẩn hoá về $[-1, 1]$ giúp mô hình học được đặc trưng theo cả chiều tăng và giảm của cường độ ảnh, trong khi $[0, 1]$ chỉ phản ánh độ sáng tương đối.

3 Whitening – Chuẩn hoá về Zero Mean, Unit Variance

```
1 def whitening(image):
2     return (image - np.mean(image)) / np.std(image)
```

Khái niệm: Đây là dạng chuẩn hoá phổ biến nhất trong học sâu, đưa ảnh về trung bình bằng 0 và độ lệch chuẩn bằng 1.

Lý do sử dụng:

- Dữ liệu có trung bình quá xa 0 sẽ gây lệch gradient trong quá trình lan truyền ngược (backpropagation).
- Đưa dữ liệu về phân phối chuẩn hoá giúp mạng hội tụ nhanh hơn, đặc biệt với các optimizer như SGD hoặc Adam.
- Tránh hiện tượng "bias" do một số ảnh có độ sáng cao, một số ảnh lại rất tối.
- Được dùng phổ biến trong ResNet, EfficientNet, và hầu hết mô hình SOTA hiện nay.

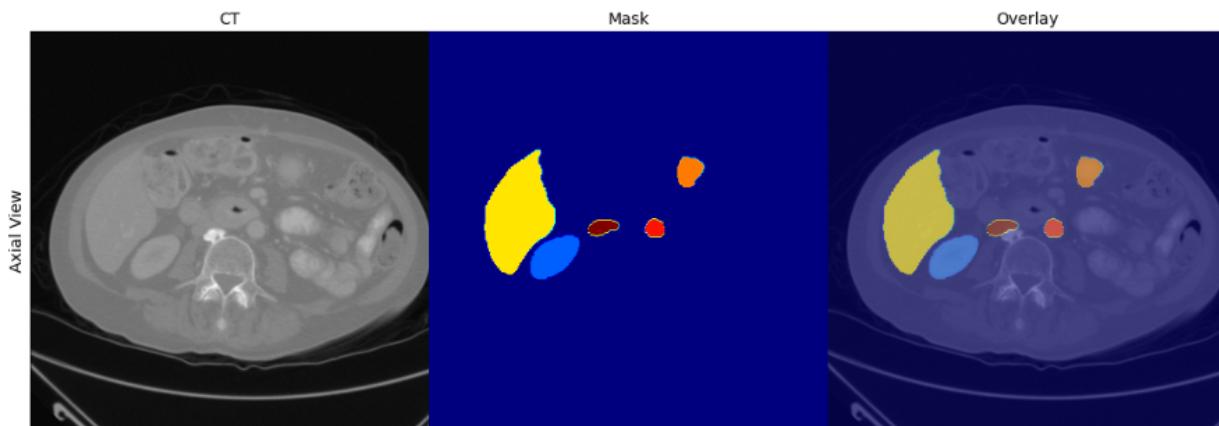
Lưu ý khi dùng:

- Whitening nên được áp dụng sau khi cắt ngưỡng HU để tránh ảnh hưởng bởi outlier.
- Không áp dụng cho ảnh mask (nhị phân 0/1), vì sẽ mất cấu trúc.

6 Resampling ảnh (Chuẩn hoá Spacing Voxel)

Trong ảnh CT hoặc MRI 3D, dữ liệu không chỉ là một ma trận các giá trị, mà còn chứa thông tin về kích thước vật lý của mỗi voxel — gọi là **spacing**.

CT Shape Original=(147, 512, 512), Resampled to 1mm=(441, 342, 342)
 CT Mask Shape=(147, 512, 512), Resampled to 1mm=(441, 342, 342)



Hình 9: Ví dụ về resampling ảnh

Spacing là gì?

Spacing (hay còn gọi là độ phân giải vật lý) mô tả kích thước thực tế (theo milimet) của mỗi voxel theo ba trục: (`spacing_x`, `spacing_y`, `spacing_z`).

- Một voxel có spacing (0.5, 0.5, 5.0) có nghĩa là:
 - Mỗi pixel trong mặt phẳng XY có kích thước 0.5×0.5 mm
 - Khoảng cách giữa hai lát cắt (slice) theo trục Z là 5.0 mm
- Một ảnh khác có thể có spacing (1.0, 1.0, 1.0) → voxel là khối lập phương.

Vấn đề: Nếu không xử lý, hai ảnh có cùng kích thước số học (vd: $128 \times 128 \times 64$) nhưng spacing khác nhau sẽ có tỷ lệ vật lý khác hoàn toàn — dẫn đến hình dạng cấu trúc giải phẫu bị méo lệch trong mắt mô hình học sâu.

Ví dụ:

- Cùng là gan, nhưng một ảnh có lát cắt dày 5mm, một ảnh khác dày 1mm → khối gan trong ảnh đầu sẽ “mỏng” hơn trong không gian 3D thật.
- Nếu không resample, mạng CNN 3D có thể học nhầm kích thước và hình dạng thực của mô.

Giải pháp: Resampling ảnh về spacing chuẩn

Để đồng nhất tỷ lệ vật lý giữa các ảnh, ta cần **nội suy (interpolation)** lại ảnh về cùng một spacing chuẩn — ví dụ: (1.0, 1.0, 1.0) mm hoặc (2.0, 2.0, 2.0) mm.

```

1 def resample_img(itk_image, out_spacing=[2.0, 2.0, 2.0]):
2     original_spacing = itk_image.GetSpacing()
3     original_size = itk_image.GetSize()
4
5     out_size = [
6         int(np.round(original_size[0] * (original_spacing[0] / out_spacing[0]))),
7         int(np.round(original_size[1] * (original_spacing[1] / out_spacing[1]))),

```

```

8     int(np.round(original_size[2] * (original_spacing[2] / out_spacing[2])))
9 ]
10
11 resample = sitk.ResampleImageFilter()
12 resample.SetOutputSpacing(out_spacing)
13 resample.SetSize(out_size)
14 ...
15 return resample.Execute(itk_image)

```

Ý nghĩa của nội suy lại spacing

- **Đảm bảo đồng bộ vật lý:** Dữ liệu sau nội suy sẽ cùng tỷ lệ voxel, tránh sai lệch về hình học.
- **Tăng tính tổng quát của mô hình:** Mạng học sâu không phải tự điều chỉnh theo từng spacing khác nhau.
- **Tương thích với kiến trúc mạng cố định kích thước (input shape):** ví dụ các mô hình U-Net, DenseVNet cần ảnh đầu vào có kích thước thống nhất.

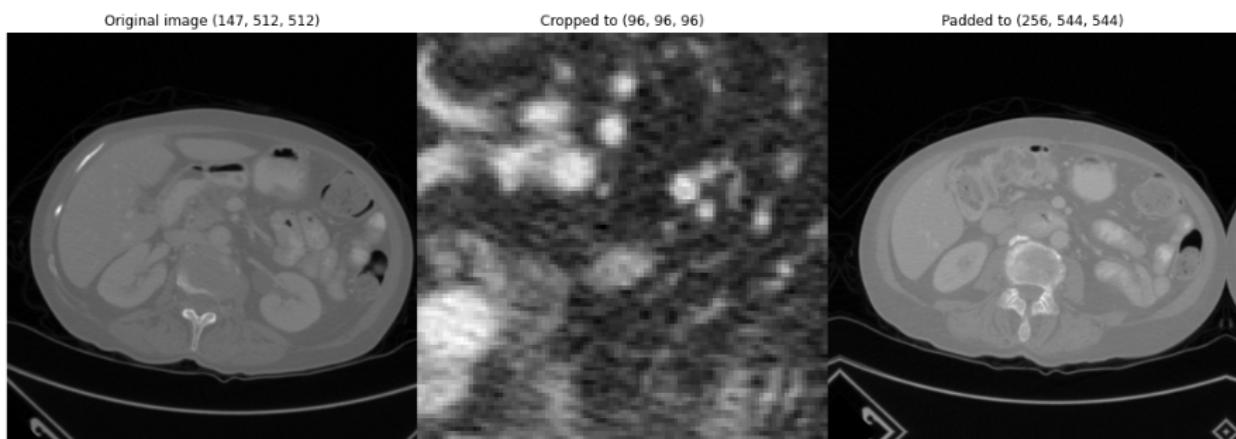
Chọn spacing bao nhiêu là phù hợp?

- Với ảnh toàn thân hoặc ổ bụng: (2.0, 2.0, 2.0) mm là phổ biến
- Với ảnh não hoặc chi tiết cao: có thể dùng (1.0, 1.0, 1.0) mm hoặc nhỏ hơn
- Spacing nhỏ hơn → độ chi tiết cao hơn → yêu cầu GPU lớn hơn

Lưu ý:

- `sitkBSpline`: dùng để nội suy ảnh liên tục (CT, MRI).
- `sitkNearestNeighbor`: dùng cho ảnh nhãn (mask segmentation) để tránh mất giá trị nhãn.

7 Crop hoặc Pad ảnh về kích thước cố định



Hình 10: Ví dụ về crop và padding ảnh

Vì sao cần crop/pad ảnh 3D?

Một đặc điểm phổ biến của ảnh y học 3D (CT, MRI) là kích thước không đồng nhất giữa các bệnh nhân:

- **Về kích thước voxel:** ảnh có thể là $(512 \times 512 \times 50)$ hoặc $(256 \times 256 \times 200)$
- **Về vùng chụp (FOV - field of view):** có người chụp toàn thân, có người chỉ chụp gan

Tuy nhiên, các mô hình học sâu (như 3D U-Net, VNet) yêu cầu kích thước ảnh **đầu vào cố định** để có thể huấn luyện theo batch hoặc tính toán gradient trên GPU.

Nếu kích thước ảnh thay đổi, mô hình sẽ lỗi ngay từ bước forward hoặc backward do shape không thống nhất.

Giải pháp: Cắt (Crop) hoặc đệm thêm (Pad)

Ý tưởng là đưa mọi ảnh về cùng kích thước mục tiêu (vd: $64 \times 64 \times 64$):

- Nếu ảnh quá lớn → ta **cắt bớt** phần biên, giữ lại vùng trung tâm chứa ROI (Region of Interest)
- Nếu ảnh quá nhỏ → ta **đệm thêm số 0** (zero-padding) ở các chiều thiếu để đủ kích thước

```

1 def resize_image_with_crop_or_pad(image, img_size=(64, 64, 64), **kwargs):
2     """Image resizing. Resizes image by cropping or padding dimension
3         to fit specified size.
4     Args:
5         image (np.ndarray): image to be resized
6         img_size (list or tuple): new image size
7         kwargs (): additional arguments to be passed to np.pad
8     Returns:
9         np.ndarray: resized image
10    """
11
12    assert isinstance(image, (np.ndarray, np.generic))
13    assert (image.ndim - 1 == len(img_size) or image.ndim == len(img_size)), \
14        'Example size doesnt fit image size'
15
16    # Get the image dimensionality
17    rank = len(img_size)
18
19    # Create placeholders for the new shape
20    from_indices = [[0, image.shape[dim]] for dim in range(rank)]
21    to_padding = [[0, 0] for dim in range(rank)]
22
23    slicer = [slice(None)] * rank
24
25    # For each dimensions find whether it is supposed to be cropped or padded
26    for i in range(rank):
27        if image.shape[i] < img_size[i]:
28            to_padding[i][0] = (img_size[i] - image.shape[i]) // 2
29            to_padding[i][1] = img_size[i] - image.shape[i] - to_padding[i][0]
30        else:
31            from_indices[i][0] = int(np.floor((image.shape[i] - img_size[i]) / 2.))
32            from_indices[i][1] = from_indices[i][0] + img_size[i]
33

```

```

34     # Create slicer object to crop or leave each dimension
35     slicer[i] = slice(from_indices[i][0], from_indices[i][1])
36
37     # Pad the cropped image to extend the missing dimension
38     return np.pad(image[slicer], to_padding, **kwargs)

```

Vì sao cách này hiệu quả?

- **Tập trung vào ROI:** phần lớn thông tin lâm sàng nằm ở trung tâm (gan, phổi, não), nên việc cắt biên không gây mất mát nghiêm trọng.
- **Tăng tốc độ training:** ảnh nhỏ hơn giúp giảm chi phí bộ nhớ và tăng batch size.
- **Tránh lỗi out-of-memory trên GPU:** nếu ảnh lớn quá mà không crop, mô hình dễ bị tràn RAM GPU.
- **Tương thích với DataLoader:** các framework như PyTorch yêu cầu tensor đầu vào có shape đồng nhất.

Một số chiến lược crop/pad phổ biến:

- **Center crop:** cắt từ trung tâm ảnh
- **Random crop:** dùng cho augment, tăng đa dạng dữ liệu
- **Symmetric padding:** đệm đều 2 bên mỗi chiều
- **Asymmetric padding:** tuỳ theo hướng thiều nhiều hay ít

8 Maximum Intensity Projection (MIP)

MIP là gì trong thực tế lâm sàng?

Maximum Intensity Projection (MIP) là một kỹ thuật dựng ảnh 2D từ ảnh 3D bằng cách chọn giá trị voxel có **cường độ sáng nhất** (intensity lớn nhất) trên mỗi “tia” dọc theo một trục nhất định (thường là trục Z).

Cụ thể:

- Mỗi pixel trong ảnh MIP là giá trị lớn nhất trên cột voxel tương ứng trong không gian 3D.
- Được xem như việc “chiếu tia X ảo” xuyên qua khối 3D và chỉ giữ lại phần “sáng nhất”.

```

1 mip = np.max(image, axis=0) # Chiu theo truc Z

```

Ứng dụng của MIP trong y học

MIP được sử dụng rộng rãi trong chẩn đoán hình ảnh, đặc biệt trong các trường hợp sau:

- **Chẩn đoán mạch máu (CTA, MRA):** Do lòng mạch được tiêm thuốc cản quang có cường độ rất cao, MIP giúp làm nổi bật cấu trúc mạch máu nhỏ nhất trong khối 3D.
- **Tìm tổn thương có mật độ cao:** như nốt vôi hoá, sỏi, khối u rắn.

- **Hiển thị xương trong CT toàn thân hoặc CT sọ não:** vì xương có HU rất cao (> 1000), luôn là điểm sáng nhất trong các lát cắt.
- **Phát hiện di căn xương, vỡ xương, khối u phổi...** qua các lát chiếu toàn vùng.

Tính chất đặc biệt của ảnh CT/MRI khiếu MIP hữu ích

Ảnh CT/MRI 3D có các đặc điểm:

- **Cấu trúc lồng ghép theo chiều sâu:** Có nhiều mô khác nhau nằm chồng lên nhau trong không gian 3D, khó thấy bằng lát cắt đơn lẻ.
- **ROI nhỏ và mờ:** Tổn thương cần phát hiện thường rất nhỏ và dễ bị bỏ sót trong một lát cắt đơn.
- **Cường độ không đều:** Một số cấu trúc sáng rõ, nhưng cũng có thể bị che khuất nếu xem từng lát.

MIP giúp “tổng hợp” toàn bộ các cấu trúc sáng nhất, làm nổi bật vùng nghi ngờ mà bác sĩ (hoặc mô hình AI) cần chú ý.

MIP giúp mô hình học sâu như thế nào?

Mặc dù ảnh 3D chứa nhiều thông tin hơn, nhưng có những lý do khiến ta dùng MIP trong mô hình học sâu:

- **Chuyển ảnh 3D thành 2D:** giúp tận dụng các mô hình 2D CNN mạnh mẽ (như ResNet, EfficientNet) mà không cần kiến trúc 3D phức tạp.
- **Tiết kiệm tài nguyên:** mô hình 2D có ít tham số hơn, dễ huấn luyện và phù hợp hơn với máy tính có GPU hạn chế.
- **Tăng độ tương phản vùng ROI:** vì MIP loại bỏ các voxel tối, giúp mô hình tập trung học đặc trưng nổi bật nhất.
- **Tăng tính khái quát hoá (generalization):** mô hình học đặc trưng theo toàn vùng chiếu, thay vì học lệch theo lát cắt cụ thể.
- **Phù hợp với học tự giám sát (self-supervised):** ví dụ dùng MIP làm augment hoặc target trong pretext tasks.

So sánh với ảnh gốc hoặc slice 2D

Loại ảnh	Ưu điểm	Hạn chế
Ảnh lát cắt (slice)	Đầy đủ thông tin theo lớp	ROI nhỏ dễ bị bỏ sót
Ảnh MIP	ROI sáng rõ, toàn vùng	Mất thông tin chiều sâu
Ảnh 3D gốc	Chính xác vật lý, giàu thông tin	Cần mạng 3D, chi phí lớn