语法

```
a.shape :torch.Size([2, 3, 4])
b.shape :torch.Size([100, 100])
c = a[b]
c.shape :torch.Size([100, 100, 3, 4])
c = a[:,b]
c.shape :torch.Size([2, 100, 100, 4])
```

String objects have a bunch of useful methods; for example:

```
s = "hello"
print(s.capitalize())  # Capitalize a string; prints "Hello"
print(s.upper())       # Convert a string to uppercase; prints "HELLO"
print(s.rjust(7))      # Right-justify a string, padding with spaces; prints "  hello"
print(s.center(7))     # Center a string, padding with spaces; prints " hello "
print(s.replace('l', '(ell)'))  # Replace all instances of one substring with another;
                                # prints "he(ell)(ell)o"
print('  world '.strip())  # Strip leading and trailing whitespace; prints "world"
```

Dictionaries

A dictionary stores (key, value) pairs, similar to a `Map` in Java or an object in Javascript. You can use it like this:

```
d = {'cat': 'cute', 'dog': 'furry'}  # Create a new dictionary with some data
print(d['cat'])       # Get an entry from a dictionary; prints "cute"
print('cat' in d)     # Check if a dictionary has a given key; prints "True"
d['fish'] = 'wet'     # Set an entry in a dictionary
print(d['fish'])      # Prints "wet"
# print(d['monkey'])  # KeyError: 'monkey' not a key of d
print(d.get('monkey', 'N/A'))  # Get an element with a default; prints "N/A"
print(d.get('fish', 'N/A'))    # Get an element with a default; prints "wet"
del d['fish']         # Remove an element from a dictionary
print(d.get('fish', 'N/A')) # "fish" is no longer a key; prints "N/A"
```

Loops: It is easy to iterate over the keys in a dictionary:

```
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal in d:
    legs = d[animal]
    print('A %s has %d legs' % (animal, legs))
# Prints "A person has 2 legs", "A cat has 4 legs", "A spider has 8 legs"
```

If you want access to keys and their corresponding values, use the `items` method:

```
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal, legs in d.items():
    print('A %s has %d legs' % (animal, legs))
# Prints "A person has 2 legs", "A cat has 4 legs", "A spider has 8 legs"
```

zip()

2. `zip` 函数详解

基本功能

- **输入**：多个可迭代对象（如列表、字符串、元组等）。
- **输出**：一个迭代器，每次迭代返回一个元组，包含来自各输入序列的对应元素。
- **对齐规则**：以最短的输入序列长度为准，超长部分被截断。

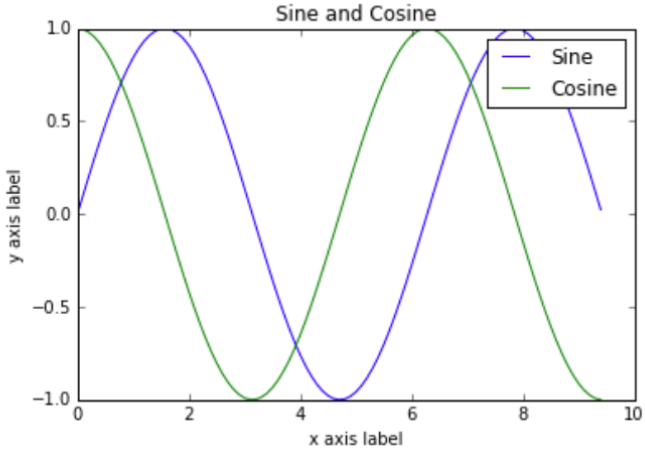
示例代码

python 复制

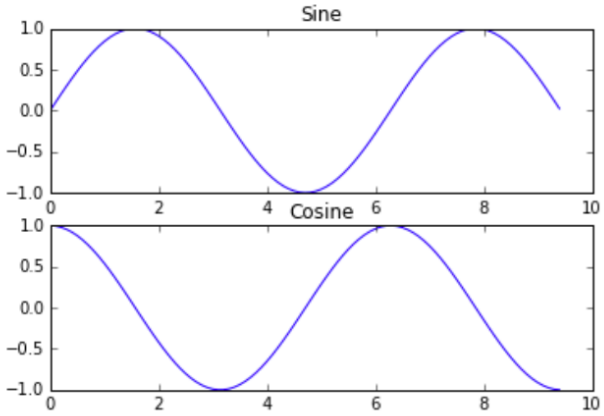
```python
# 示例 1：基础用法
names = ["Alice", "Bob", "Charlie"]
ages = [25, 30, 35]

zipped = zip(names, ages)
print(list(zipped))  # 输出：[('Alice', 25), ('Bob', 30), ('Charlie', 35)]

# 示例 2：不同长度的输入
heights = [165, 180]
zipped = zip(names, heights)
print(list(zipped))  # 输出：[('Alice', 165), ('Bob', 180)]（Charlie 被截断）

# 示例 3：解压已压缩的数据
zipped = zip(names, ages)
unzipped = zip(*zipped)  # 解压为两个独立元组
print(list(unzipped))    # 输出：[('Alice', 'Bob', 'Charlie'), (25, 30, 35)]
```

matplotlib

```
# Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Plot the points using matplotlib
plt.plot(x, y_sin)
plt.plot(x, y_cos)
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.title('Sine and Cosine')
plt.legend(['Sine', 'Cosine'])
plt.show()
```

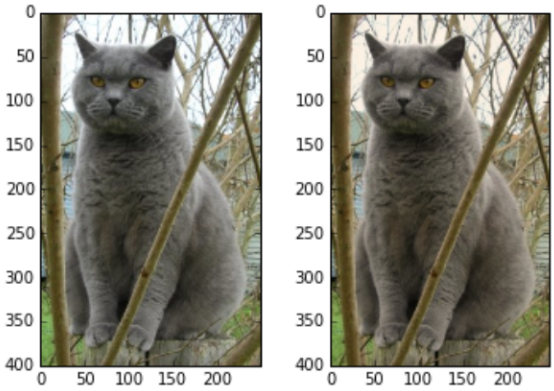


```
# Set up a subplot grid that has height 2 and width 1,
# and set the first such subplot as active.
plt.subplot(2, 1, 1)

# Make the first plot
plt.plot(x, y_sin)
plt.title('Sine')

# Set the second subplot as active, and make the second plot.
plt.subplot(2, 1, 2)
plt.plot(x, y_cos)
plt.title('Cosine')

# Show the figure.
plt.show()
```



imshow

```
import numpy as np
from scipy.misc import imread, imresize
import matplotlib.pyplot as plt

img = imread('assets/cat.jpg')
img_tinted = img * [1, 0.95, 0.9]

# Show the original image
plt.subplot(1, 2, 1)
plt.imshow(img)

# Show the tinted image
plt.subplot(1, 2, 2)

# A slight gotcha with imshow is that it might give strange results
# if presented with data that is not uint8. To work around this, we
# explicitly cast the image to uint8 before displaying it.
plt.imshow(np.uint8(img_tinted))
plt.show()
```



figure,xticks

```
import matplotlib.pyplot as plt

# 方式1：创建画布时直接指定
plt.figure(figsize=(宽度, 高度))  # 单位是英寸
```

```
# 绘制曲线（仅传递 y 数据，x 轴默认为索引）
plt.plot(train_loss, label='Loss')

# 单独设置 x 轴标签
plt.xlabel('epoch')  # ✅ 正确方法

plt.legend()
plt.show()
```

```python
# 设置横坐标间隔为 1（显示所有 epoch）
plt.xticks(np.arange(0, len(train_loss), step=1))  # 关键设置
```

基本功能

`imshow()` 用于将数组数据可视化为图像，支持多种数据类型（如灰度图、RGB 彩色图、热力图等）。

python 复制

```python
plt.imshow(X, cmap=None, vmin=None, vmax=None, ...)
```

核心参数详解

参数	说明	默认值	典型应用场景
** `X` **	输入数据，可以是： - 2D 数组（灰度图） - 3D 数组（H×W×C，C为通道数）	-	MNIST 手写数字（2D）
** cmap **	颜色映射表（colormap） **单通道必须指定**（如 `'gray'`，`'binary'`）	`'viridis'`	灰度图用 `'gray'`，热
** `vmin/vmax` **	数据范围，用于归一化颜色映射	数据最小/最大值	数据范围异常时手动指
** origin **	坐标原点位置 `'upper'`（左上角，Matplotlib 默认）或 `'lower'`	`'upper'`	数学坐标系需设为 `'lo`
** `interpolation` **	插值方法，控制缩放时的显示效果 常用：`'nearest'`（像素化）、`'bilinear'`（平滑）	`'nearest'`	缩小图像用 `'nearest'`
** alpha **	透明度（0~1） 叠加显示时使用（如与背景图混合）	1.0	半透明叠加效果

scatter

1. `plt.scatter()` ：绘制散点图

用途：展示两个变量之间的分布关系，常用于观察数据点的相关性、聚类或离群值。

基本语法

python 复制

```python
plt.scatter(x, y, s=None, c=None, marker=None, cmap=None, norm=None,
            vmin=None, vmax=None, alpha=None, linewidths=None, *,
            edgecolors=None, plotnonfinite=False, **kwargs)
```

核心参数

参数	说明
`x` ， `y`	数据点的横纵坐标（数组或列表）。
`s`	点的大小（标量或与数据点数量相同的数组，默认 `20` ）。
`c`	点的颜色（颜色名称、十六进制码或数值数组，默认蓝色）。
`marker`	点的形状（如 `'o'` 圆形、 `'s'` 方形、 `'^'` 三角形，默认 `'o'` ）。
`alpha`	透明度（0~1，默认 `1` ）。
`edgecolors`	点边缘颜色（默认 `'face'` 使用 `c` 参数颜色， `'none'` 无边缘）。
`linewidths`	点边缘线宽（默认 `None` ）。

numpy

方法 1：直接布尔索引赋值

python 复制

```python
import numpy as np

# 示例数组
arr = np.array([[-1, 2, 0], [3, -4, 5], [0.5, -0.3, 0]])

# 将所有大于 0 的元素设为 1
arr[arr > 0] = 1

print(arr)
```

输出：

```
[[-1  1  0]
 [ 1 -4  1]
 [ 1 -0.3  0]]
```

关键步骤

1. **条件生成布尔数组**： `arr > 0` 会生成一个与 `arr` 形状相同的布尔数组，标识每个元素是否大于 0。
2. **布尔索引定位元素**： `arr[arr > 0]` 选中所有满足条件的元素。
3. **批量赋值**：直接对选中的元素赋值为 `1` 。

random.*

1. `np.random.rand` ：均匀分布随机数

功能

生成指定维度的 [0, 1) **均匀分布**随机数（左闭右开区间）。

语法

python

```
np.random.rand(d0, d1, ..., dn)
```

- **参数**： `d0, d1, ..., dn` 为整数，表示每个维度的大小。
- **返回值**：指定维度的数组，元素范围在 `[0, 1)` 。

示例

python

```
import numpy as np

# 生成 2x3 的均匀分布数组
arr = np.random.rand(2, 3)
print(arr)
```

2. `np.random.randn` ： 标准正态分布随机数

功能

生成指定维度的 **标准正态分布（均值=0，标准差=1）** 随机数。

语法

```python
np.random.randn(d0, d1, ..., dn)
```

- **参数**： `d0, d1, ..., dn` 为整数，表示每个维度的大小。
- **返回值**：指定维度的数组，元素服从标准正态分布。

示例

```python
# 生成 2x3 的标准正态分布数组
arr = np.random.randn(2, 3)
print(arr)
```

输出示例：

```
[[ 0.49671415 -0.1382643   0.64768854]
 [ 1.52302986 -0.23415337 -0.23413696]]
```

3.1 `np.random.randint` ： 生成随机整数

```python
np.random.randint(low, high=None, size=None, dtype=int)
```

- **参数**：
  - `low` ：最小值（包含）。
  - `high` ：最大值（不包含，若未指定则范围为 `[0, low)` ）。
  - `size` ：输出形状。
- **示例**：

```python
# 生成 2x2 的 0~9 之间的整数
arr = np.random.randint(0, 10, (2, 2))
print(arr)
```

3.2 `np.random.uniform` ：均匀分布（自定义范围）

```python
np.random.uniform(low=0.0, high=1.0, size=None)
```

- **参数**：
  - `low` ：最小值（包含）。
  - `high` ：最大值（不包含）。
  - `size` ：输出形状。
- **示例**：

```python
# 生成 2x2 的 1~2 之间的均匀分布
arr = np.random.uniform(1.0, 2.0, (2, 2))
print(arr)
```

3.3 `np.random.normal` ：正态分布（自定义均值和标准差）

```python
np.random.normal(loc=0.0, scale=1.0, size=None)
```

- **参数**：
  - `loc` ：均值（默认 0）。
  - `scale` ：标准差（默认 1）。
  - `size` ：输出形状。
- **示例**：

```python
# 生成均值为 5，标准差为 2 的正态分布
arr = np.random.normal(5, 2, (2, 2))
print(arr)
```

3.5 np.random.seed ：设置随机种子

python

```
np.random.seed(seed)
```

- **参数**： seed 为整数。
- **作用**：固定随机数生成器的种子，确保结果可重复。
- **示例**：

python

```
np.random.seed(42)
print(np.random.rand(3))  # 每次运行结果相同
```

full,eye,slices

```
c = np.full((2,2), 7)  # Create a constant array
print(c)               # Prints "[[ 7.  7.]
                       #          [ 7.  7.]]"

d = np.eye(2)          # Create a 2x2 identity matrix
print(d)               # Prints "[[ 1.  0.]
                       #          [ 0.  1.]]"

e = np.random.random((2,2))  # Create an array filled with random values
print(e)                     # Might print "[[ 0.91940167  0.08143941]
                             #               [ 0.68744134  0.87236687]]"
```

```
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]

# A slice of an array is a view into the same data, so modifying it
# will modify the original array.
print(a[0, 1])   # Prints "2"
b[0, 0] = 77     # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])   # Prints "77"
```

```python
# Two ways of accessing the data in the middle row of the array.
# Mixing integer indexing with slices yields an array of lower rank,
# while using only slices yields an array of the same rank as the
# original array:
row_r1 = a[1, :]    # Rank 1 view of the second row of a
row_r2 = a[1:2, :]  # Rank 2 view of the second row of a
print(row_r1, row_r1.shape)  # Prints "[5 6 7 8] (4,)"
print(row_r2, row_r2.shape)  # Prints "[[5 6 7 8]] (1, 4)"
```

add.at()

三、与普通索引赋值的区别

特性	np.add.at	普通索引赋值 a[indices] += b
重复索引处理	累加所有重复索引的值	后续赋值覆盖前面的结果
广播支持	支持，自动扩展 b 的形状以匹配索引区域	需手动确保形状匹配
性能	略慢于原地操作，但功能更灵活	更快，但功能受限

示例对比:

python 复制

```python
a = np.array([1, 2, 3, 4])
# np.add.at
np.add.at(a, [0,0], 1)  # 索引0被加两次
print(a)  # 输出: [3,2,3,4]

# 普通索引赋值
a = np.array([1, 2, 3, 4])
a[[0,0]] += 1  # 仅最后一次赋值生效
print(a)  # 输出: [2,2,3,4]
```

array,ndarray,asarray

```python
# ------------------------------
# 示例 2: np.array 处理已有 ndarray
# ------------------------------
original = np.array([1, 2, 3])
arr_copy = np.array(original)  # 即使输入是 ndarray，也生成新数组
original[0] = 99
print("
np.array 处理已有 ndarray:")
print("原数组:", original)        # [99  2  3]
print("np.array 生成的数组:", arr_copy)  # [1 2 3]（数据被复制，不受原数组修改影响）
```

```python
# ------------------------------
# 示例 4: np.asarray 处理 ndarray（无复制）
# ------------------------------
original = np.array([1, 2, 3])
arr_asarray = np.asarray(original)  # 直接返回原数组的视图（无复制）
original[0] = 99
print("
np.asarray 处理 ndarray:")
print("原数组:", original)          # [99  2  3]
print("np.asarray 结果:", arr_asarray)  # [99  2  3]（与原数组共享内存）
```

```python
# ------------------------------
# 示例 6: np.ndarray 直接创建数组
# ------------------------------
# 直接通过底层构造函数创建（不常用！）
raw_arr = np.ndarray(shape=(2, 2), dtype=int)  # 未初始化，内容为随机值
print("
np.ndarray 直接创建:")
print(raw_arr)
# 输出示例（每次运行结果不同）：
# [[         0 1072693248]
#  [1073741824 1074266112]]
```

argpartition,nsmallest

方法 3：使用 `np.argpartition()` 获取索引

如果需要同时获取最小 `k` 个数的索引：

python

```python
# 获取前 k 小元素的索引
indices = np.argpartition(arr, k)[:k]
smallest_k_with_indices = arr[indices]

print("最小 k 个数的索引:", indices)
print("对应的值:", smallest_k_with_indices)
```

输出：

最小 k 个数的索引: [1 3 6]
对应的值: [1 1 2]

方法 4：使用 `np.nsmallest()` （直接返回结果）

NumPy 提供了更直观的 `nsmallest()` 函数，直接返回最小的 `k` 个数（有序）：

python 复制

```python
smallest_k_nsmallest = np.nsmallest(k, arr)
print("最小的 k 个数（nsmallest）:", smallest_k_nsmallest)
```

输出：

最小的 k 个数（nsmallest）: [1 1 2]

性能对比

方法	时间复杂度	是否有序	适用场景
`np.argpartition()`	O(n)	无序	大数据，仅需值
`np.nsmallest()`	O(n log k)	有序	需要有序结果
`np.sort()`	O(n log n)	有序	小数据，需全排序

注意事项

1. **重复值处理**：上述方法均保留重复的最小值。
2. **k 的合法性**：确保 `k <= len(arr)` ，否则会报错。
3. **多维数组**：如果输入是多维数组，可以通过 `axis` 参数指定轴（默认扁平化处理）。

linalg.norm()

`numpy.linalg.norm(x, ord=None, axis=None, keepdims=False)` 计算数组的范数，支持向量、矩阵及高维数组。以下是详细说明：

参数解析

1. **x**：输入数组（向量、矩阵或多维数组）。
2. **ord**：范数的阶数，决定计算方式。常用值如下：
   - **向量范数**（一维数组或数组的展平结果）：
     - `None` 或 `2`：欧氏范数（默认），即 $\sqrt{\sum x_i^2}$。
     - `1`：绝对值之和，$\sum |x_i|$。
     - `-1`：反向绝对值之和（等价于 `1` 的倒数，实际计算同 `1`）。
     - `inf`：最大绝对值，$\max |x_i|$。
     - `-inf`：最小绝对值，$\min |x_i|$。
     - 其他正数 `p`：L^p 范数 $\left(\sum |x_i|^p\right)^{1/p}$。
   - **矩阵范数**（二维数组）：
     - `'fro'`：Frobenius 范数，$\sqrt{\sum_{i,j} x_{ij}^2}$。
     - `2` 或 `'nuc'`：谱范数（最大奇异值）或核范数（奇异值之和）。**注意**：`ord=2` 计算谱范数，`ord='nuc'` 计算核范数。
     - `1`：列和的最大值，$\max_j \sum_i |x_{ij}|$。
     - `-1`：反向列和的最小值。
     - `inf`：行和的最大值，$\max_i \sum_j |x_{ij}|$。
     - `-inf`：反向行和的最小值。

maximum,minimum,fmax

函数	功能	示例
`np.maximum`	逐元素取最大值	`np.maximum(a, b)`
`np.minimum`	逐元素取最小值	`np.minimum(a, b)`
`np.fmax`	逐元素取最大值，但忽略 `NaN` 值	`np.fmax([1, np.nan], [3, 2])` → `[3, 2]`
`np.amax`	沿指定轴取全局最大值（返回标量或数组）	`np.amax(a, axis=0)`
`np.max`	沿指定轴取全局最大值（简写形式，等同于 `amax`）	`np.max(a)`
`np.greater`	逐元素比较是否大于（返回布尔数组）	`np.greater(a, b)` → `a > b`

关键区别

- **`np.maximum` vs `np.fmax`**：`np.fmax` 忽略 `NaN`，而 `np.maximum` 会将 `NaN` 视为无效值（结果保留另一数组的值）。
- **`np.maximum` vs `np.amax`**：`np.maximum` 是逐元素操作，而 `np.amax` 是沿轴取全局最大值。

unique()

基本功能

返回输入数组 `ar` 中的 **唯一元素**（去重后的元素），默认按升序排列。

参数详解

python 复制

```
numpy.unique(ar, return_index=False, return_inverse=False, return_counts=False, axis=None)
```

参数	说明
`ar`	输入数组（需为一维或可展平的多维数组）。
`return_index`	若为 `True`，返回唯一值在原数组中第一次出现的索引。默认 `False`。
`return_inverse`	若为 `True`，返回一个数组，可通过该数组重构原数组的唯一值映射。默认 `False`。
`return_counts`	若为 `True`，返回每个唯一值的出现次数。默认 `False`。
`axis`	沿指定轴查找唯一元素（对多维数组有效）。若为 `None`，则数组会被展平后处理。可选值： `axis=0`（按列处理）、`axis=1`（按行处理）。默认 `None`。

返回值

- 若未指定任何参数，返回 **扁平化后的唯一值数组**（升序排列）。
- 若指定多个参数（如 `return_index` 和 `return_inverse`），返回元组 `(unique_values, indices)` 或 `(unique_values, inverse_indices)` 等。

```python
import numpy as np

arr = np.array([1, 2, 2, 3, 3, 3, 4, 4, 4, 4])

# 获取唯一值和对应出现次数
unique_values, counts = np.unique(arr, return_counts=True)

# 将结果转为字典（可选）
count_dict = dict(zip(unique_values, counts))

print("唯一值:", unique_values)  # 输出: [1 2 3 4]
print("出现次数:", counts)       # 输出: [1 2 3 4]
print("字典形式:", count_dict)   # 输出: {1: 1, 2: 2, 3: 3, 4: 4}
```

array_split()

函数签名

python 复制

```
numpy.array_split(a, indices_or_sections, axis=0)
```

参数解析

参数	说明
a	要分割的输入数组（可以是任意维度的数组）。
indices_or_sections	分割方式，可以是以下两种形式： - **整数**：将数组平均分成 N 份（若无法整除，余数分配到前几个子数组）。 - **列表/数组**：指定沿轴的分割位置（如 [2, 5] 表示在第2和第5元素后分割）。
axis	分割的轴方向（默认为 0 ，即沿第一个轴分割）。

核心功能

- **不均匀分割**：当数组长度无法被均分时，余数会分配到前面的子数组。
- **灵活指定分割点**：通过索引列表精确控制分割位置。
- **多维数组支持**：可沿任意轴分割（如二维数组的行或列）。

```
arr = np.arange(10)  # 创建数组 [0 1 2 3 4 5 6 7 8 9]

# 分成3份（无法整除，余数分配到前两份）
split_arrays = np.array_split(arr, 3)

print("分割后的数组：")
for sub_arr in split_arrays:
    print(sub_arr)
```

输出：

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

concatenate/vstack/hstack

1. 使用 `np.concatenate()`

`np.concatenate()` 是最通用的拼接函数，支持沿指定轴（ `axis` ）拼接多个数组。

语法

python

```
numpy.concatenate((a1, a2, ...), axis=0)
```

示例

python

```
import numpy as np

# 创建两个二维数组
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6]])

# 沿行方向拼接（axis=0）
result_row = np.concatenate([a, b], axis=0)
print("沿行拼接:
", result_row)
# 输出:
# [[1 2]
#  [3 4]
#  [5 6]]

# 沿列方向拼接（axis=1）
result_col = np.concatenate([a, b.T], axis=1)  # 注意 b 需要转置
print("沿列拼接:
", result_col)
# 输出:
# [[1 2 5]
#  [3 4 6]]
```

2. 使用 `np.vstack()` 和 `np.hstack()`

- `np.vstack()` ：垂直拼接（沿行方向，等效于 `axis=0` ）。
- `np.hstack()` ：水平拼接（沿列方向，等效于 `axis=1` ）。

示例

python

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

# 垂直拼接
vstack_result = np.vstack([a, b])
print("垂直拼接:
", vstack_result)
# 输出:
# [[1 2]
#  [3 4]
#  [5 6]]

# 水平拼接（需确保列数一致）
hstack_result = np.hstack([a, b.T])  # b 需要转置为 [[5], [6]]
print("水平拼接:
", hstack_result)
# 输出:
# [[1 2 5]
#  [3 4 6]]
```

Pytorch

PyTorch 是一个基于 Python 的开源深度学习框架，以**动态计算图**和**易用性**著称，广泛应用于机器学习研究、模型开发和生产部署。它的核心模块覆盖张量计算、自动微分、神经网络构建及优化等任务，并提供了丰富的工具库。

PyTorch 的核心模块

模块名	功能描述
** `torch` **	核心模块，提供张量操作（类似 NumPy）、自动梯度（autograd）、GPU 加速等功能。
** torch.nn **	神经网络工具包，包含层（如 `Linear` ， `Conv2d` ）、损失函数（如 `CrossEntropyLoss` ）、优化器（如 `SGD` ， `Adam` ）等。
** `torch.optim` **	优化算法库（如 `AdamW` ， `RMSprop` ），与 `nn.Module` 集成。
** torch.autograd **	自动微分引擎，支持张量的链式求导（ `backward()` ）。
** `torch.distributed` **	分布式训练工具，支持多 GPU、多节点并行计算。
** torch.utils.data **	数据加载与处理工具（如 `DataLoader` ， `Dataset` ）。
** `torchvision` **	计算机视觉工具库，包含数据集（CIFAR-10/100）、模型（ResNet, VGG）和图像变换（ `transforms` ）。
** torchtext **	自然语言处理（NLP）工具库，提供文本处理、数据集和预训练词向量。
** `torchaudio` **	音频处理工具库，支持音频 I/O、特征提取（如 MFCC）和语音识别任务。
** torch_geometric **	图神经网络（GNN）库，支持图结构数据处理。
** `torch.jit` **	模型序列化工具（TorchScript），支持模型跨平台部署。
** torch.onnx **	模型导出工具，可将 PyTorch 模型转为 ONNX 格式供其他框架使用。
** `torch.profiler` **	性能分析工具，用于优化模型推理和训练速度。

clamp()

2. `torch.clamp` 函数

功能

将张量中的每个元素限制在指定的最小值和最大值之间。

函数签名

```python
torch.clamp(input, min, max, out=None) → Tensor
```

- ** `input` **: 输入张量。
- ** `min` **: 允许的最小值（标量或张量）。
- ** `max` **: 允许的最大值（标量或张量）。
- ** `out` **: 输出张量（可选）。

返回值

返回一个 **新张量**，所有元素被限制在 `[min, max]` 范围内。

核心规则

- **逐元素操作**：对张量中的每个元素单独进行裁剪。
- **广播机制**：若 `min` 或 `max` 是张量，需满足广播条件。
- **不改变原始数据**：返回新张量，原始数据不变。

transpose

方法 1: `torch.transpose()`

功能：交换张量的两个指定维度。
语法：

```python
torch.transpose(input, dim0, dim1)
```

或通过张量对象调用：

```python
x.transpose(dim0, dim1)
```

示例：

假设有一个形状为 `(2, 3, 4)` 的张量，交换第 1 维和第 2 维（索引从 0 开始）：

```python
import torch

x = torch.randn(2, 3, 4)
y = x.transpose(1, 2)  # 新形状变为 (2, 4, 3)
```

unsqueeze() / squeeze()

1. `torch.unsqueeze` 函数

功能

在张量的指定位置插入一个新的维度（维度大小为 1）。

函数签名

python 复制

```
torch.unsqueeze(input, dim) → Tensor
```

- ** `input` **: 输入张量。
- ** `dim` **: 插入新维度的位置（整数，可为负数）。

返回值

返回一个 **新张量**，形状在 `dim` 位置增加了一个维度。

核心规则

- **维度索引**：PyTorch 的维度索引从 0 开始，支持负数（如 `-1` 表示最后一个位置）。
- **不改变数据**：仅改变张量的形状，不修改原始数据。
- **与 `squeeze` 的关系**：`unsqueeze` 的逆操作是 `squeeze(dim)`，用于删除大小为 1 的维度。

numel()

`numel()` 是 PyTorch 中的一个张量方法，用于**计算张量中所有元素的总数**，无论张量的维度如何。它会将张量的所有维度相乘，返回一个标量值，表示张量中元素的个数。

基本用法

python 复制

```python
import torch

# 创建一个张量
x = torch.randn(3, 4, 5)  # 形状为 (3, 4, 5)

# 计算元素总数
total_elements = x.numel()

print(total_elements)  # 输出：3 * 4 * 5 = 60
```

shape,size()

```python
# 创建一个张量
x = torch.randn(3, 4, 5)

# 使用 size()
print(x.size())  # 输出：torch.Size([3, 4, 5])

# 使用 shape
print(x.shape)   # 输出：torch.Size([3, 4, 5]))

# 修改 shape 属性（直接改变张量形状）
x.shape = (2, 10)  # 等价于 x = x.reshape(2, 10)
print(x.size())    # 输出：torch.Size([2, 10])

# 修改 size() 无效（需通过其他方法）
x.size = (3, 5)    # 报错！AttributeError: can't set attribute
```

3. 关键细节

(1) `shape` 是属性，可直接修改

- 修改 `shape` 会直接改变张量的形状（类似 `reshape` ）：

```python
x = torch.randn(3, 4)
x.shape = (4, 3)  # 修改形状
print(x.shape)    # 输出：torch.Size([4, 3])
```

(2) `size()` 是方法，返回当前形状

torchsummary.summary()

```
from torchsummary import summary
from cnn import CNN
summary(CNN().to('cuda'), (1,28,28))    # (C, H, W)
```

[1] ✓ 1.9s

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 28, 28]             320
         MaxPool2d-2           [-1, 32, 14, 14]               0
              ReLU-3           [-1, 32, 14, 14]               0
            Conv2d-4           [-1, 64, 14, 14]          18,496
         MaxPool2d-5             [-1, 64, 7, 7]               0
              ReLU-6             [-1, 64, 7, 7]               0
           Flatten-7                 [-1, 3136]               0
            Linear-8                  [-1, 128]         401,536
              ReLU-9                  [-1, 128]               0
           Linear-10                   [-1, 10]           1,290
================================================================
Total params: 421,642
Trainable params: 421,642
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.46
Params size (MB): 1.61
Estimated Total Size (MB): 2.07
----------------------------------------------------------------
```

torch.mm / bmm / matmul

操作	功能简述	输入形状要求	输出形状	示例代码
`torch.mm`	**二维矩阵乘法**（仅限二维矩阵）	两个二维张量 `(m, n)` 和 `(n, p)`	`(m, p)`	`python  a = torch.randn(2, 3) b = torch.randn(3, 4) c = torch.mm(a, b) # 形状 (2, 4)`
`torch.bmm`	**批量矩阵乘法**（严格三维张量，对齐批量维度）	两个三维张量 `(B, M, N)` 和 `(B, N, P)`	`(B, M, P)`	`python  a = torch.randn(2, 3, 4) b = torch.randn(2, 4, 5) c = torch.bmm(a, b) # 形状 (2, 3, 5)`
`torch.matmul`	**通用矩阵乘法**（支持广播和高维张量，自动对齐最后两个维度）	任意维度，但最后两个维度需满足矩阵乘法规则：`(..., M, N)` 和 `(..., N, P)`	`(..., M, P)`	`python  a = torch.randn(2, 3, 4) b = torch.randn(2, 4, 5) c = torch.matmul(a, b) # 形状 (2, 3, 5)`
`@`（运算符）	**语法糖**，等价于 `torch.matmul`	同 `torch.matmul`	同 `torch.matmul`	`python  a = torch.randn(2, 3, 4) b = torch.randn(2, 4, 5) c = a @ b # 形状 (2, 3, 5)`

register_buffer / nn.Parameter

```python
import torch
import torch.nn as nn

class CustomLayer(nn.Module):
    def __init__(self):
        super().__init__()
        # 注册一个缓冲区，保存移动均值（不需要梯度）
        self.register_buffer("running_mean", torch.zeros(10))

        # 注册一个参数（需要梯度，会被优化器更新）
        self.weight = nn.Parameter(torch.randn(10))

    def forward(self, x):
        # 使用缓冲区和参数
        return x * self.weight + self.running_mean

# 创建模块实例
layer = CustomLayer()

# 查看参数和缓冲区
print(layer.named_parameters())  # 输出: ('weight', Parameter(...))
print(layer.named_buffers())     # 输出: ('running_mean', tensor([0., 0., ...]))
```

nn.optim

四、优化器（ `torch.nn.optim` ）

1. 随机梯度下降（ `SGD` ）

python

```python
class torch.optim.SGD(parameters, lr=1e-3, momentum=0, weight_decay=0)
```

- **参数**:
  - `lr` ：学习率
  - `momentum` ：动量系数（用于加速收敛）
  - `weight_decay` ：L2 正则化强度

2. Adam 优化器

python

```python
class torch.optim.Adam(parameters, lr=1e-3, betas=(0.9, 0.999), eps=1e-8)
```

- **参数**:
  - `betas` ：一阶矩和二阶矩衰减率
  - `eps` ：防止除零的小常数

nn.ModelList

2. **替代普通列表**
   如果用普通列表存储子模块，PyTorch 不会将其参数注册到模型中，导致训练时无法更新参数。

```python
# ❌ 错误：普通列表不会注册参数
self.layers = [nn.Linear(10, 10) for _ in range(3)]

# ✅ 正确：使用 ModuleList
self.layers = nn.ModuleList([nn.Linear(10, 10) for _ in range(3)])
```

3. **访问子模块**
   支持通过索引（如 `self.layers[0]` ）或名称（需自定义）访问子模块。

关键方法与属性

方法/属性	说明
append(module)	在列表末尾添加一个子模块
insert(index, module)	在指定位置插入子模块
extend(modules)	扩展列表，添加多个子模块
__getitem__(index)	通过索引访问子模块
named_children()	返回子模块的名称和模块本身的迭代器

Scipy

SciPy 的主要模块

模块名	功能示例
scipy.cluster	聚类分析（K-means、层次聚类）
scipy.constants	物理常数（如光速、普朗克常数）
scipy.fftpack	快速傅里叶变换（FFT）
scipy.integrate	积分、常微分方程求解（如 quad , odeint ）
scipy.interpolate	数据插值（样条插值、多项式插值）
scipy.io	文件读写（如 MATLAB .mat 文件）
scipy.linalg	线性代数运算（矩阵分解、特征值）
scipy.ndimage	多维图像处理（滤波、形态学操作）
scipy.optimize	优化算法（最小化、根求解，如 minimize ）
scipy.signal	信号处理（滤波器设计、频谱分析）
scipy.sparse	稀疏矩阵操作
scipy.spatial	空间分析（KD树、距离计算）
scipy.special	特殊数学函数（伽马函数、误差函数等）
scipy.stats	统计分布、假设检验、概率密度函数