

ChatGPT - math



[파이토치] 아래 내용 설명좀

텐서(tensor)는 배열(array)이나 행렬(matrix)과 매우 유사한 특수한 자료구조입니다.

PyTorch에서는 텐서를 사용하여 모델의 입력(input)과 출력(output), 그리고 모델의 매개변수들을 부호화(encode)합니다.

텐서는 GPU나 다른 하드웨어 가속기에서 실행할 수 있다는 점만 제외하면 [NumPy](https://numpy.org)의 ndarray와 유사합니다.

실제로 텐서와 NumPy 배열(array)은 종종 동일한 내부(underly) 메모리를 공유할 수 있어 데이터를 복사할 필요가 없습니다. (`bridge-to-np-label` 참고)

텐서는 또한 ([Autograd](autogradqs_tutorial.html))_ 장에서 살펴볼) 자동 미분(automatic differentiation)에 최적화되어 있습니다.

ndarray에 익숙하다면 Tensor API를 바로 사용할 수 있을 것입니다. 아니라면, 아래 내용을 함께 보시죠!

아래 글에서 말하는 내용을 단계별로 풀어서 설명하겠습니다.

1. 텐서(Tensor)란 무엇인가?

• 정의

텐서는 다차원 배열(multi-dimensional array) 또는 행렬(matrix)의 일반화된 개념입니다.

- 0차원 스칼라(scalar)
- 1차원 벡터(vector)
- 2차원 행렬(matrix)
- 3차원 이상의 텐서(tensor)

• 파이토치에서의 역할

- 모델의 **입력(input)**: 예를 들어 이미지나 연속된 시계열 데이터를 텐서로 표현
- 모델의 **출력(output)**: 분류 결과 확률, 회귀 예측값 등
- 모델의 **매개변수(parameter)**: 신경망 가중치(weight)와 편향(bias)

python

```
import torch
```

```
# 2x3 행렬을 표현하는 2차원 텐서 예시
x = torch.tensor([[1.0, 2.0, 3.0],
                  [4.0, 5.0, 6.0]])
print(x.shape) # torch.Size([2, 3])
```

2. NumPy 의 ndarray 와의 유사성

- API 유사성

대부분의 생성 함수(`torch.zeros`, `torch.randn` 등)와 연산(덧셈·곱셈·전치 등)이 NumPy의 `ndarray`와 형태와 이름이 비슷해, NumPy에 익숙하다면 금방 적응할 수 있습니다.

- 장비 가속(Device Acceleration)

- NumPy 배열은 CPU 상에서만 동작하는 반면, 파이토치 텐서는 **CPU**와 **GPU(CUDA)**, 나아가 **TPU** 등 다양한 하드웨어에서 동일한 API로 실행할 수 있습니다.
- GPU를 쓸 때는 `.to('cuda')` 또는 `.cuda()` 만으로 간단히 옮겨 실행 속도를 크게 높일 수 있습니다.

```
python

# CPU 텐서 생성
cpu_tensor = torch.randn(1000, 1000)

# GPU로 복사
if torch.cuda.is_available():
    gpu_tensor = cpu_tensor.to('cuda')
# 또는 gpu_tensor = cpu_tensor.cuda()
```

3. 메모리 공유(Memory Sharing) — NumPy ↔ PyTorch

- Zero-copy 인터페이스

`torch.from_numpy()` 또는 `ndarray.numpy()` 를 사용하면, 데이터 복사 없이 동일한 메모리 버퍼를 공유합니다.

- 변경이 한 쪽에서 일어나면 다른 쪽에도 즉시 반영
- 불필요한 복사를 방지하여 메모리·성능 이점

```
python

import numpy as np

a = np.ones((2,2), dtype=np.float32)
t = torch.from_numpy(a) # a 와 t 가 같은 메모리 공유
t[0,0] = 42
print(a) # [[42.  1.]
          # [ 1.  1.]
```

4. 자동 미분(Automatic Differentiation) 최적화

- 파이토치의 **Autograd** 엔진은 텐서 연산을 기록(연산 그래프)한 뒤, 역전파(backpropagation)를 통해 **미분 (gradient)** 을 자동으로 계산합니다.
- 텐서 객체의 `requires_grad=True` 설정만으로, 모든 연산에 대한 기울기 추적이 가능해집니다.

```
python

# requires_grad=True 로 설정하면 기울기 추적 시작
x = torch.tensor([2.0, 3.0], requires_grad=True)
```

```

y = x**2 + 3*x + 1    # 연산 그래프 구성
z = y.sum()           # 스칼라 값

z.backward()          # 역전파 수행
print(x.grad)         #  $\partial z / \partial x = 2x + 3 \rightarrow [7.0, 9.0]$ 

```

5. 요약 및 활용 팁

1. **ndarray 경험 그대로:** NumPy 코드를 거의 그대로 옮겨와도 되고, ``.to('cuda')`` 만 추가하면 GPU 가속.
2. **메모리 효율:** ``.torch.from_numpy`` / ``.ndarray.numpy`` 로 복사 없이 서로 변환.
3. **자동 미분:** 딥러닝 모델 학습에 필요한 기울기 계산을 ``.requires_grad`` 한 줄로 해결.

이제 텐서의 기본 개념과 파이토치 텐서가 제공하는 주요 장점(장비 가속, 메모리 공유, 자동 미분 최적화)을 이해하셨을 겁니다. NumPy와 비슷한 문법으로 시작해, GPU 활용과 Autograd 기능을 차근차근 적용해 보세요!



```

-----
RuntimeError                                Traceback (most recent call last)
Cell In[30], line 4
      1 x_ones = torch.ones_like(x_data) # x_data의 속성을 유지합니다.
      2 print(f"Ones Tensor: \n {x_ones} \n")
----> 4 x_rand = torch.rand_like(x_data) # x_data의 속성을 덮어씁니다.
      5 print(f"Random Tensor: \n {x_rand} \n")

RuntimeError: "check_uniform_bounds" not implemented for 'Long'

```

해당 에러 메시지는 ``.torch.rand_like``가 **정수형(LongTensor)** 텐서에 대해서는 균일분포 난수를 생성하는 기능을 지원하지 않기 때문에 발생합니다. 조금 더 구체적으로 살펴보면:

원인

- ``.torch.rand_like(tensor)``는 0과 1 사이의 실수형 난수를 텐서와 동일한 **크기(size)**와 **속성(dtype, device)**으로 생성합니다.
- 내부적으로 균일분포 난수 생성 알고리즘은 **floating point** 타입 주변에서 동작하도록 구현되어 있어, ``.Long`` (64-bit 정수) 타입에는 ``.check_uniform_bounds`` 연산이 정의되어 있지 않습니다.
- 그래서 ``.x_data.dtype``가 ``.torch.long`` (또는 ``.torch.int64``) 일 때 위와 같은 ``.RuntimeError``가 발생합니다.

해결 방법

1. 실수형 텐서로 변환 후 ``.rand_like`` 사용

가장 간단한 방법은 원본 텐서를 실수형으로 변환해서 (``.float32`` 등) 난수를 생성하는 것 입니다.

python

```
# 예시: x_data가 LongTensor라면
x_data_float = x_data.to(torch.float32)

# 동일한 크기·속성의 실수형 균일분포 난수 생성
x_rand = torch.rand_like(x_data_float)
print(x_rand.dtype) # torch.float32
```

⚠ 이 경우 `x_rand`는 `float32` 타입이므로, 이후 정수 연산이나 정수 인덱싱에 쓰고 싶다면 다시 `long()` 또는 `to(torch.int64)`로 변환해야 합니다.

2. 정수 범위 내 난수가 필요할 때: `torch.randint_like`

정수형 텐서 크기 그대로 “특정 범위”의 정수형 난수를 채우고 싶다면 `torch.randint_like`를 사용하세요.

python

```
# x_data와 동일한 크기의 정수형 난수(0 ≤ rand < high) 생성
high = 10
x_rand_int = torch.randint_like(x_data, high=high)
print(x_rand_int) # 예: tensor([[3, 7, 1], [0, 9, 4]], dtype=torch.int64)
```

- `high` 인자 외에 `low`, `dtype`, `device` 등도 지정할 수 있습니다.

3. `dtype` 파라미터로 오버라이드

`torch.rand_like`에 `dtype` 인자를 직접 넘겨서 실수형으로 지정할 수도 있습니다.

python

```
# x_data는 LongTensor
x_rand = torch.rand_like(x_data, dtype=torch.float32, device=x_data.device)
print(x_rand.dtype) # torch.float32
```

요약

- **에러 원인:** `rand_like`는 정수형 텐서에 대한 균일분포 생성이 구현되어 있지 않음.
- **실수 난수가 필요하면**
 1. 텐서를 실수형으로 변환한 뒤 `rand_like`
 2. 또는 `dtype` 파라미터로 실수형 지정
- **정수 난수가 필요하면**
 - `torch.randint_like`

이제 `torch.rand_like`를 어떤 상황에, 어떤 dtype으로 써야 할지 이해되셨을 거예요. 필요에 맞춰 위 세 가지 방법 중 하나를 골라 적용해 보세요!



torch.cat 이랑 torch.stack 이 뭐가 다른거야 다음 내용을 참조해서 정리해봐

<https://pytorch.org/docs/stable/generated/torch.cat.html#torch.cat>

<https://pytorch.org/docs/stable/generated/torch.stack.html>

아래 표는 `torch.cat` 과 `torch.stack` 의 핵심 차이점을 정리한 것입니다.

기능	<code>torch.cat</code>	<code>torch.stack</code>
기본 동작	기존 차원(existing dimension)을 따라 텐서들을 이어 붙임	새로운 차원(new dimension)을 삽입한 뒤 텐서들을 이어 붙임
주요 제약	모든 입력 텐서는 이어붙이는(dim) 차원을 제외한 나머지 차원들이 동일해야 함 PyTorch	모든 입력 텐서는 모두 동일한 크기(shape) 여야 함 PyTorch
출력 형태 변화	입력 텐서들의 해당 차원 크기들이 합쳐져 늘어남	입력 텐서 개수만큼 새로운 차원 크기가 생기며, 기존 차원들은 그대로 유지됨
차원 지정 인자	<code>dim</code> (int): 이어붙일 기존 차원의 인덱스 (default=0)	<code>dim</code> (int): 새로 추가 할 차원의 인덱스 (0부터 <code>ndim</code> 까지 가능, default=0)
대표 예시	<pre>python
 x = torch.randn(2,3)
 torch.cat((x,x), dim=0).shape # →
 torch.Size([4,3])
</pre>	<pre>python
 x = torch.randn(2,3)
 torch.stack((x,x), dim=0).shape # →
 torch.Size([2,2,3])
</pre>

예시 코드 비교

```
python

import torch

x = torch.tensor([[1,2,3],
                  [4,5,6]])
# torch.cat: 같은 2차원 공간에서 이어 붙이기
# - dim=0 축(행)을 따라 두 배 늘어난 4x3 행렬 생성
cat_out = torch.cat((x, x), dim=0)
print(cat_out.shape) # torch.Size([4,3])

# torch.stack: 새로운 축을 도입하여 3차원 텐서 생성
# - dim=0 위치에 새로운 축이 생겨 2x2x3 텐서 생성
stack_out = torch.stack((x, x), dim=0)
print(stack_out.shape) # torch.Size([2,2,3])
```

• 언제 쓰나?

- 서로 다른 배치(batch)나 시퀀스 길이를 유지하면서 단순히 이어 붙이고 싶을 때는 `torch.cat`
- 서로 다른 모델 출력 혹은 반복된 결과를 하나의 추가 차원으로 묶어 관리하고 싶을 때는 `torch.stack`

이 차이를 이해하면, 데이터 전처리나 모델 출력 합치기 작업에서 원하는 형태로 텐서를 구성할 수 있습니다.



출처



2차원넘어가니까 시각화가 어렵네 방법 없나

3차원 이상의 텐서를 화면에 ‘썩’ 펼쳐서 보는 건 당연히 어렵습니다. 대신, 다음과 같은 기법들을 조합해서 시각화할 수 있습니다.

1. 슬라이스(slice)별로 떼어서 보기

높은 차원의 텐서는 “하고 싶은 만큼” 슬라이스를 잘라서 2D로 보여주는 방식이 가장 직관적입니다.

- **3D 텐서** ($C \times H \times W$ 혹은 $D \times H \times W$)
 - 채널별(또는 깊이별)로 `matplotlib.pyplot.imshow`
 - Jupyter Notebook의 `ipywidgets` 슬라이서로 인덱스 조정
- **4D 텐서** ($B \times C \times H \times W$; 배치(batch) 단위)
 - `torchvision.utils.make_grid`로 배치 내부 여러 이미지를 한 번에 격자(grid) 형태로 출력

```
python

import torch, torchvision, matplotlib.pyplot as plt

# 예: 4D 텐서 (batch, channel, height, width)
x = torch.randn(8, 3, 64, 64)

# 배치 내 8장 이미지를 2x4 격자로 묶기
grid = torchvision.utils.make_grid(x, nrow=4, normalize=True, pad_value=1)

plt.figure(figsize=(6,3))
plt.imshow(grid.permute(1,2,0)) # (H, W, C) 순서로 바꿔줘야 plt가 읽음
plt.axis('off')
```

2. 채널(channel) 합성

컬러 이미지를 다루는 `c=3` 일 때만 유효한 게 아닙니다.

- 1채널 데이터 여러 장
 - R/G/B 채널에 각각 다른 슬라이스를 매핑하거나
 - heatmap(색상 지도로)을 겹쳐서 표시
- `seaborn.heatmap` 혹은 `plt.pcolormesh` 로 2D 매트릭스를 컬러맵으로 보기

3. 인터랙티브 시각화 도구

- **TensorBoard**
 - `SummaryWriter.add_image` / `add_images` 로 배치 시각화

- ``add_embedding`` 으로 벡터(1D) → 2D/3D 프로젝션
- **Plotly / Bokeh / Napari**
 - 슬라이드바, 줌 기능 포함한 뷰어
 - 3D 부피(volume) 렌더링(Plotly의 ``Volume`` trace 등)

4. 차원 축소 (Dimensionality Reduction)

(채널 수나 피쳐(feature) 수가 너무 많아 슬라이스가 많다면)

- **PCA / t-SNE / UMAP**
 1. ``torch.flatten(x, start_dim=1)`` 등으로 ``(N, D)`` 형태로 바꾼 뒤
 2. ``sklearn.decomposition.PCA`` 혹은 ``TSNE(n_components=2)`` 적용
 3. 2D 산점도(scatter plot)로 점군 시각화

```
python

from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

# 예: (batch, features) 형태로 플래튼
feat = x.view(x.size(0), -1).cpu().numpy()

# PCA로 2차원 프로젝션
proj = PCA(n_components=2).fit_transform(feat)

plt.scatter(proj[:,0], proj[:,1], s=30, cmap='viridis')
plt.title("PCA Projection of Batch Features")
plt.show()
```

5. 3D 플롯

- ``matplotlib`` 의 ``Axes3D`` 모듈로 ``(i, j, 값)`` 형태의 3D surface나 scatter
- ``plotly.graph_objects.Surface`` 등으로 풍부한 인터랙티브 그래프

결론

1. 슬라이스 → 2D 이미지로 분할
2. 그리드 → 배치/채널 묶음
3. 인터랙티브 툴 → TensorBoard, Plotly, Napari
4. 차원 축소 → PCA/t-SNE 로 전체 구조 파악

이 방법들을 상황에 맞게 조합하면, 3차원 이상 텐서도 충분히 '눈에 보이도록' 분석하고 디버깅할 수 있습니다. 필요하신 예제나 코드가 더 있으면 말씀해 주세요!



`stack_out = torch.stack((x, x), dim=0)` 이랑 `stack_out = torch.stack((x, x), dim=1)` 이 직관적으로 안 와닿네

위 예제는 NumPy의 `np.stack`을 사용했지만, `torch.stack`도 **동일한 방식**으로 동작합니다. 결과를 한눈에 보면:

1. `axis=0` (또는 `dim=0`)

```
python
np.stack((x, x), axis=0).shape # → (2, 2, 3)
```

- 첫 번째 축(0번 축)에 새로운 차원이 생깁니다.
- 결과 텐서는 `[copy_index, row, col]` 순서로 색인이 됩니다.
- 즉, `stack0[0]`은 원본 `x` 전체, `stack0[1]` 역시 `x` 전체를 담고 있습니다.

```
csharp
stack0 =
[
    [ # copy 0: x 전체
      [1,2,3],
      [4,5,6]
    ],
    [ # copy 1: x 전체
      [1,2,3],
      [4,5,6]
    ]
]
```

2. `axis=1` (또는 `dim=1`)

```
python
np.stack((x, x), axis=1).shape # → (2, 2, 3)
```

- 두 번째 축(1번 축)에 새로운 차원이 생깁니다.
- 결과 텐서는 `[row, copy_index, col]` 순서로 색인이 됩니다.
- 즉, 각 **행(row)** 별로 복사본이 쌓여 있습니다:
 - `stack1[0]`은 원본 `x`의 첫 번째 행 `[1,2,3]`이 복사본 2개로 묶인 것,
 - `stack1[1]`은 두 번째 행 `[4,5,6]`이 복사본 2개로 묶인 것.

```
csharp
stack1 =
[
    [ # row 0
      [1,2,3], # copy 0 of row 0
      [1,2,3]  # copy 1 of row 0
    ],
    [ # row 1
      [4,5,6], # copy 0 of row 1
      [4,5,6]  # copy 1 of row 1
    ]
]
```



```
]
]
```

직관적인 정리

- `dim=0`:
 - “행렬(혹은 텐서) 전체를 위아래로 쌓는다.”
 - 결과의 `0` 번 축이 **몇 개를 쌓았는지**(여기선 2) 를 나타냄.
- `dim=1`:
 - “각 행(row)을 안쪽으로 하나씩 복제해 쌓는다.”
 - 결과의 `1` 번 축이 **몇 개를 쌓았는지**(여기선 2) 를 나타냄.

즉, ‘**새로운 축(new dimension)**’이 어디에 생기느냐가 바로 `dim` (또는 NumPy의 `axis`) 인자에 따른 차이입니다. 이 위치에 따라 데이터 구조(인덱싱 순서)가 바뀌는 것이죠.



시각화 없어?

아래는 `torch.stack((x,x), dim=0)` 과 `torch.stack((x,x), dim=1)` 의 결과를 **2D 형태로** 풀어쓴 시각화입니다. (값은 모두 동일하기 때문에 숫자만 보고도 어느 축에 쌓였는지 구분할 수 있습니다.)

1) `stack(dim=0)`

- 결과 shape: `(2, 2, 3)` → **첫 번째 축(0번 축)**에 “전체 행렬” 2개를 쌓음

slice\row	row 0	row 1
slice 0	[1, 2, 3]	[4, 5, 6]
slice 1	[1, 2, 3]	[4, 5, 6]

이 표에서 **slice 0** 과 **slice 1** 이 각각 `x` 전체(2×3)를 담고 있는 모습입니다.

2) `stack(dim=1)`

- 결과 shape: `(2, 2, 3)` → **두 번째 축(1번 축)**에 “각 행(row)” 단위로 2개 복제본을 쌓음

row\copy	copy 0	copy 1
row 0	[1, 2, 3]	[1, 2, 3]
row 1	[4, 5, 6]	[4, 5, 6]

이 표에서 **row 0** 과 **row 1** 각각이 내부적으로 복제본(copy 0, copy 1) 2개를 갖고 있음을 볼 수 있습니다.

요약

- ``dim=0`` → “행렬 전체”를 위아래로 쌓는다 → slice 단위로 구분
- ``dim=1`` → “행” 단위로 안쪽에 복제본을 쌓는다 → row 단위로 구분

이렇게 표로 풀어보면 새로운 차원이 어디에 생겼는지 더 직관적으로 파악할 수 있습니다.



코드 풀이좀 해봐 쓰임새랑 # 두 텐서 간의 행렬 곱(matrix multiplication)을 계산합니다. y1, y2, y3은 모두 같은 값을 갖습니다.

``tensor.T`` 는 텐서의 전치(transpose)를 반환합니다.

``# print(tensor)``

y1 = tensor @ tensor.T

y2 = tensor.matmul(tensor.T)

y3 = torch.rand_like(y1)

torch.matmul(tensor, tensor.T, out=y3)

print(y3)

요소별 곱(element-wise product)을 계산합니다. z1, z2, z3은 모두 같은 값을 갖습니다.

z1 = tensor * tensor

z2 = tensor.mul(tensor)

z3 = torch.rand_like(tensor)

torch.mul(tensor, tensor, out=z3)

아래 예제는 “행렬 곱”과 “요소별 곱”을 계산하는 세 가지 스타일(연산자, 메서드, 함수 + ``out=``)을 보여주고, 각 스타일의 쓰임새를 설명합니다.

1. 행렬 곱셈 (matrix multiplication)

python

가정: tensor 는 (N×M) 크기의 2D 텐서

tensor.T 는 (M×N) 크기로 전치(transpose)된 텐서

1) 연산자(@) 사용

y1 = tensor @ tensor.T

2) 메서드(matmul) 사용

y2 = tensor.matmul(tensor.T)

3) 함수(torch.matmul) + out= 사용

y3 = torch.rand_like(y1)

torch.matmul(tensor, tensor.T, out=y3)

print(y3)

결과를 받을 빈 텐서 생성

결과를 y3 에 바로 기록

방식	설명	쓰임새
<code>`@`</code> 연산자	파이썬 문법 차원에서 행렬 곱을 직관적으로 표현.	빠르게 읽고 쓰기 좋음. 간단한 스크립트나 수학적 표기에 유리.

방식	설명	쓰임새
<code>`tensor.matmul(...)`</code>	텐서 객체의 메서드 호출. 내부적으로 같은 연산자를 수행하되, 추가 옵션 없이 메서드 체이닝이 가능.	메서드 체인(<code>`tensor.matmul().relu()`</code>) 형태로 연산 순서를 연결할 때.
<code>`torch.matmul(..., out=...)`</code>	함수 호출 방식. <code>`out`</code> 인자를 쓰면 이미 할당된 텐서에 값을 덮어쓰므로, 메모리 재사용 (zero-allocation)에 유리.	반복 루프 안에서 불필요한 메모리 할당을 줄이고 싶을 때.

- **메모리 이점:** ``out=...`` 을 사용하면 PyTorch가 내부에서 새로운 버퍼를 할당하지 않고, 기존에 만들어 둔 ``y3`` 텐서를 재활용합니다.
- **일관성:** 함수형 API(``torch.matmul``)는 여러 변수를 동시에 처리하거나, static graph/torchscript로 내보낼 때 안정적입니다.

2. 요소별 곱(element-wise product)

python
<pre># 1) 연산자(*) 사용 z1 = tensor * tensor # 2) 메서드(mul) 사용 z2 = tensor.mul(tensor) # 3) 함수(torch.mul) + out= 사용 z3 = torch.rand_like(tensor) torch.mul(tensor, tensor, out=z3)</pre>

방식	설명	쓰임새
<code>`*`</code> 연산자	NumPy와 유사한 문법으로, 같은 위치의 원소끼리 곱셈.	간단한 수식 표현 시 가독성↑
<code>`tensor.mul(...)`</code>	객체 메서드 호출. <code>`mul_()`</code> 과 같이 뒤에 언더스코어를 붙여 쓰면 in-place 연산(<code>`z1`</code> 대신 <code>`tensor`</code> 자체가 바뀜)도 가능.	<code>`tensor.mul_(other)`</code> 로 메모리 재활당 없이 즉시 덮어쓸 때.
<code>`torch.mul(..., out=...)`</code>	함수 호출 + <code>`out=...`</code> 미리 할당해 둔 버퍼에 결과를 저장.	반복문 내부에서 불필요한 버퍼 생성을 줄여 성능 최적화할 때.

3. 언제 어떤 방식을 고를까?

1. 가독성 우선

- 한두 줄의 연산이고, 메모리/성능이 크게 문제되지 않을 때는 ``@``, ``*`` 연산자.
- 예) 수식처럼 직관적인 코드를 빠르게 작성할 때.

2. 메서드 체이닝

- 여러 연산을 연결해서 한 줄로 쓰고 싶다면 ``matmul()``, ``mul()``, ``add()``, ``relu()`` 등.
- 예) ``out = tensor.matmul(W).add(b).relu()``

3. 메모리/성능 최적화

- 대규모 반복 학습 루프 안에서 같은 크기의 결과를 여러 번 생성해야 할 때: 함수형 API + `out=`.
- 예) 커스텀 학습 루프에서 매 배치마다 새로운 텐서를 할당하지 않으려면

```
python

tmp = torch.empty_like(prev)
torch.matmul(prev, W, out=tmp)
prev = tmp
```

4. In-place 연산

- 메모리 사용량을 극도로 줄이려면 `.mul_()`, `.add_()` 처럼 뒤에 언더스코어가 붙은 in-place 메서드.
- 주의: Autograd 추적(gradient computation)을 방해할 수 있으므로, 필요할 때만 신중히 사용.

정리

- 연산자: 간단·직관적.
- 메서드: 체이닝, in-place 옵션(`_`) 지원.
- 함수 (`torch.*`): `out=` 으로 메모리 재할당 최소화, static API 호환성.

필요에 따라 세 방식을 적절히 조합하면, **가독성**, **성능**, **메모리 효율**을 모두 잡을 수 있습니다.



NumPy 변환(Bridge)

CPU 상의 텐서와 NumPy 배열은 메모리 공간을 공유하기 때문에, 하나를 변경하면 다른 하나도 변경됩니다.
라고 하는데 그럼 GPU 상에선 분리야? 아니면 넘파이는 gpu로 어차피 못 넘어와?

NumPy 배열은 **호스트(CPU) 메모리** 상에서만 동작하기 때문에, GPU에 올라간 텐서와는 직접 메모리를 공유할 수 없습니다. 정리하면 다음과 같습니다.

1. CPU 텐서 ↔ NumPy 배열: 메모리 공유

```
python

import torch, numpy as np

t_cpu = torch.ones(3, dtype=torch.float32)          # CPU 텐서
a = t_cpu.numpy()                                   # 뷰(view) 형태로 공유
t_cpu[0] = 42
print(a)      # [42.  1.  1.]
```

- `torch.from_numpy()` 와 `ndarray.numpy()` 는 **zero-copy** 뷰를 반환
- 한쪽을 바꾸면 다른 쪽에도 즉시 반영

2. GPU 텐서 ↔ NumPy 배열: 별도 복사

python

```
t_gpu = torch.ones(3, device='cuda') # GPU 텐서
# a = t_gpu.numpy() # ❌ 오류! CUDA 텐서는 .numpy() 지원 불가

# CPU로 가져오면서 NumPy 변환 (메모리 복사 발생)
a = t_gpu.cpu().numpy()
```

- ``.cpu()``를 거치지 않고는 NumPy 변환이 불가능 → GPU → CPU 메모리 복사
- 따라서 NumPy 배열과는 메모리 공유하지 않음

3. GPU 상에서 NumPy처럼 쓰고 싶다면?

- **CuPy**: NumPy 호환 API를 갖춘 GPU 배열 라이브러리

python

```
import cupy as cp
t_gpu = torch.randn(3, device='cuda')
# DLPack 브리지로 공유 가능 (zero-copy)
a_cp = cp.from_dlpack(torch.utils.dlpack.to_dlpack(t_gpu))
```

- CuPy 배열은 GPU 메모리를 바로 쓸 수 있고, NumPy와 유사한 문법을 제공합니다.

요약

- CPU 텐서 ↔ NumPy: 메모리 공유 (zero-copy)
- GPU 텐서 ↔ NumPy: 메모리 공유 불가 → ``.cpu().numpy()``로 복사
- GPU에서 NumPy 스타일로 작업하려면 **CuPy** + DLPack 브리지를 활용하세요.