Part2 Tensor

파트 2: PyTorch 의 심장, 텐서(Tensor) 완벽 정복

PyTorch 로 딥러닝 모델을 구축하고 훈련하는 모든 과정은 '텐서(Tensor)'라는 특별한 데이터 구조 위에서 이루어집니다. "PyTorch 의 모든 것은 텐서로 시작해서 텐서로 끝난다"고 해도 과언이 아닐 정도로, 텐서는 PyTorch 의 가장 근본적이고 핵심적인 구성 요소입니다. 텐서는 PyTorch 의 심장과도 같습니다.

우리가 모델에 입력하는 데이터(이미지, 텍스트, 소리), 모델이 학습을 통해 최적화하는 가중치(weight)와 편향(bias), 심지어 모델의 예측이 얼마나 틀렸는지를 나타내는 손실(loss) 값까지, PyTorch 세계의 모든 것은 텐서라는 형태로 표현되고 처리됩니다. 따라서 텐서를 얼마나 능숙하게 다루는지가 PyTorch 를 활용한 딥러닝 개발의 생산성과 효율성을 결정짓는 중요한 척도가 됩니다.

이 파트에서는 텐서의 개념을 명확히 이해하는 것부터 시작하여, 데이터를 자유자재로 다룰 수 있는 견고한 기초를 다지는 것을 목표로 합니다. 80 개 이상의 풍부한 예제와 각 코드 라인에 대한 상세한 해설을 통해, 다음과 같은 내용들을 깊이 있게 마스터하게 될 것입니다.

- **텐서의 개념:** 텐서가 수학적으로 무엇을 의미하며, NumPy 배열과 비교하여 PyTorch 텐서가 갖는 특별한 능력은 무엇인지 알아봅니다.
- **텐서 생성:** 기존 데이터로부터 텐서를 만들거나, 특정 모양과 값으로 초기화하고, 다른 텐서의 속성을 상속받는 등 40 가지가 넘는 다양한 생성 방법을 배웁니다.
- **텐서의 속성:** 텐서의 모양(shape), 데이터 타입(dtype), 장치(device) 등 텐서의 상태를 나타내는 중요한 속성들을 확인하고 디버깅에 활용하는 방법을 익힙니다.
- **텐서 연산:** 데이터를 자르고, 붙이고, 모양을 바꾸는 기본적인 조작부터, 딥러닝 모델의 순전파를 구성하는 핵심적인 수학 연산까지 70 가지 이상의 연산을 마스터합니다.
- **브로드캐스팅:** 모양이 다른 텐서 간의 연산을 가능하게 하는 강력한 메커니즘인 브로드캐스팅의 원리를 이해하고 활용합니다.
- **NumPy 호환성:** 데이터 분석 생태계의 표준인 NumPy 와 PyTorch 텐서 간의 효율적인 데이터 교환 방법을 배웁니다.

이 파트가 끝날 때쯤, 여러분은 텐서라는 강력한 도구를 손에 쥐고, 어떤 형태의 데이터든 자신감 있게 처리하며 복잡한 딥러닝 모델을 구현할 준비를 마치게 될 것입니다. 이제 PyTorch 의 심장부로 함께 들어가 보겠습니다.

2.1 텐서(Tensor)란?

텐서의 개념을 처음 접하면 다소 추상적으로 느껴질 수 있습니다. 하지만 딥러닝의 맥락에서 텐서는 매우 직관적인 개념으로 이해할 수 있습니다. 간단히 말해, 텐서는 **다차원 배열(Multi-dimensional Array)**입니다. 만약 여러분이 Python 의 리스트나 NumPy 의 `ndarray`에 익숙하다면, 이미 텐서의 기본적인 형태에 익숙한 것입니다.

수학적으로 텐서는 스칼라(scalar), 벡터(vector), 행렬(matrix)을 더 높은 차원으로 일반화한 개념입니다. 물리학이나 공학에서는 물리량을 기술하는 데 사용되지만, 딥러닝에서는 데이터를 담는 컨테이너로 사용됩니다.

NumPy 의 `ndarray`와 거의 동일한 개념이지만, PyTorch 의 텐서는 딥러닝 연산을 위해 최적화된 두 가지 강력한 기능을 추가로 가지고 있습니다. 이것이 바로 PyTorch 텐서를 특별하게 만드는 핵심입니다.

1. GPU 가속 (GPU Acceleration): 텐서는 CPU 뿐만 아니라 NVIDIA GPU 상에서도 연산이 가능합니다. 딥러닝은 대규모 행렬 곱셈과 같은 병렬 연산이 매우 빈번하게 발생하는데, 수천 개의 코어를 가진 GPU 는 이러한 연산을 CPU 대비 수십 배에서 수백 배까지 빠르게 처리할 수 있습니다. PyTorch 텐서는 `.to('cuda')`와 같은 간단한 명령어로 GPU 메모리로 이동하여 이러한 가속 효과를 누릴 수 있습니다.

2. 자동 미분 (Automatic Differentiation): 텐서는 자신의 연산 기록을 계산 그래프(Computational Graph)라는 구조에 저장할 수 있습니다. 이 기능을 통해, PyTorch 의 `autograd` 엔진은 복잡한 신경망의 손실 함수에 대한 각 파라미터의 기울기(gradient)를 자동으로 계산할 수 있습니다. 이는 모델 학습의 핵심인 경사하강법(Gradient Descent)을 가능하게 하는 마법 같은 기능입니다. (파트 3 에서 자세히 다룹니다.)

NumPy 의 `ndarray`에 익숙하다면 PyTorch 텐서를 배우는 것은 매우 쉽습니다. 대부분의 연산과 개념이 유사하기 때문입니다. 하지만 결정적인 차이점인 GPU 지원과 자동 미분 기능이 PyTorch 를 딥러닝의 강력한 도구로 만들어줍니다.

데이터 유형에 따른 텐서의 차원별 이해

텐서는 차원(dimension 또는 rank)의 수에 따라 다른 이름으로 불리기도 합니다. 각 차원의 텐서가 실제 딥러닝 프로젝트에서 어떤 종류의 데이터를 표현하는지 이해하는 것은 데이터의 구조를 파악하고 모델을 설계하는 데 매우 중요합니다.

- **0D 텐서 (스칼라, Scalar):** 차원이 없는 하나의 숫자 값입니다. 텐서의 가장 기본적인 형태로, 모양(shape)은 비어 있습니다 (`torch.Size([])`). **예시:** 모델의 손실(loss) 값, 정확도(accuracy) 값, 특정 뉴런의 활성화 값 하나. `torch.tensor(7)`
- **1D 텐서 (벡터, Vector):** 하나의 축을 가진 숫자의 배열입니다. Python 의 리스트와 같습니다. **예시:** 선형 계층의 편향(bias) 벡터, 하나의 데이터 샘플에 대한 피처(feature) 벡터. `torch.tensor([1, 2, 3])`
- **2D 텐서 (행렬, Matrix):** 두 개의 축(행과 열)을 가진 숫자의 배열입니다. 표 형태의 데이터와 같습니다. **예시:** 선형 계층의 가중치(weight) 행렬, 여러 데이터 샘플로 구성된 미니배치(mini-batch) 데이터 (샘플 수, 피처 수). `torch.tensor([[1, 2], [3, 4]])`
- **3D 텐서:** 세 개의 축을 가진 배열입니다. 여러 개의 행렬이 쌓여있는 형태로 볼 수 있습니다. **예시:**- **시계열 데이터:** (배치 크기, 시퀀스 길이, 피처 수) - 예: 주가 데이터 - **자연어 처리(NLP):** (배치 크기, 문장 길이, 단어 임베딩 차원) - 예: 문장들의 배치 - **흑백 이미지 배치:** (배치 크기, 높이, 너비)
- **4D 텐서:** 네 개의 축을 가진 배열입니다. 컴퓨터 비전 분야에서 가장 흔하게 사용됩니다. **예시:**- **컬러 이미지 배치:** (배치 크기, 채널 수, 높이, 너비) - 예: (64, 3, 224, 224)는 64 개의 224x224 크기 RGB 이미지를 의미합니다.
- **5D 텐서:** 다섯 개의 축을 가진 배열입니다. **예시:**- **비디오 데이터 배치:** (배치 크기, 프레임 수, 채널 수, 높이, 너비) - 예: (16, 30, 3, 128, 128)는 16 개의 비디오 클립, 각 클립은 30 프레임으로 구성됨을 의미합니다.

이처럼 텐서의 차원은 우리가 다루는 데이터의 본질적인 구조를 반영합니다. 모델의 각 레이어를 통과하면서 이 텐서의 모양이 어떻게 변하는지를 이해하고 추적하는 것이 딥러닝 모델링과 디버깅의 핵심 기술 중 하나입니다.

2.2 텐서 생성하기: 데이터로부터 직접 생성: `torch.tensor()` ① - 기본

PyTorch 에서 텐서를 생성하는 가장 기본적이고 일반적인 방법은 `torch.tensor()` 함수를 사용하는 것입니다. 이 함수는 Python 의 리스트나 튜플과 같은 기존 데이터 구조를 인자로 받아, 그 데이터를 담은 새로운 PyTorch 텐서를 생성합니다.

이 방법의 가장 중요한 특징 중 하나는 데이터를 **복사(copy)**한다는 점입니다. 즉, 원본 Python 리스트와는 완전히 독립적인 새로운 메모리 공간에 텐서가 만들어집니다. 따라서 텐서를 생성한 후에 원본 리스트의 값을 변경하더라도 생성된 텐서에는 아무런 영향이 없어, 데이터의 안정성을 보장하고 예측 불가능한 부작용(side effect)을 방지할 수 있습니다.

예제 2-1: Python 리스트로 다양한 차원의 텐서 생성하기

다양한 차원의 Python 리스트를 사용하여 텐서를 생성하고, 그 구조와 속성을 확인하는 예제입니다.

```
import torch
from typing import List

# 1. 0 차원 텐서 (스칼라) 생성
# 단일 숫자 값을 갖는 텐서로, 주로 손실(loss) 값 등을 표현할 때 사용됩니다.
scalar_tensor = torch.tensor(7)
print("--- 0D Tensor (Scalar) ---")
print(f"Tensor: {scalar_tensor}")
print(f"Shape: {scalar_tensor.shape}")
print(f"Dimensions: {scalar_tensor.ndim}₩n")

# 2. 1 차원 텐서 (벡터) 생성
# 숫자로 이루어진 리스트로부터 생성됩니다.
list_1d: List[int] = [1, 2, 3, 4]
tensor_1d = torch.tensor(list_1d)
print("--- 1D Tensor (Vector) ---")
print(f"Tensor: {tensor_1d}")
print(f"Shape: {tensor_1d.shape}")
print(f"Dimensions: {tensor_1d.ndim}₩n")

# 3. 2 차원 텐서 (행렬) 생성
# 중첩된 리스트로부터 생성됩니다.
list_2d: List[List[int]] = [[1, 2, 3], [4, 5, 6]]
```

```
tensor_2d = torch.tensor(list_2d)
print("--- 2D Tensor (Matrix) ---")
print(f"Tensor:₩n{tensor_2d}")
print(f"Shape: {tensor_2d.shape}")
print(f"Dimensions: {tensor_2d.ndim}₩n")

# 4. 3 차원 텐서 생성
# 삼중으로 중첩된 리스트로부터 생성됩니다.
list_3d: List[List[List[int]]] = [[[1, 2], [3, 4]], [[5, 6], [7, 8]]]
tensor_3d = torch.tensor(list_3d)
print("--- 3D Tensor ---")
print(f"Tensor:₩n{tensor_3d}")
print(f"Shape: {tensor_3d.shape}")
print(f"Dimensions: {tensor_3d.ndim}")
```

코드 라인별 상세 설명:

- scalar_tensor = torch.tensor(7): 숫자 7 하나를 `torch.tensor()` 함수에 전달하여 0 차원 텐서, 즉 스칼라를 생성합니다. 이 텐서는 모양(shape)이 `torch.Size([])`로 비어 있으며, 차원 수(`ndim`)는 0 입니다.
- list_1d: List[int] = [1, 2, 3, 4]: 타입 힌트(`List[int]`)를 사용하여 정수로 이루어진 1 차원 리스트임을 명시합니다. 이는 코드의 가독성을 높여주는 좋은 습관입니다.
- tensor_1d = torch.tensor(list_1d): `torch.tensor()` 함수는 입력된 Python 리스트의 구조를 그대로 반영하여 텐서를 생성합니다. 1 차원 리스트는 1D 텐서(벡터)로 변환됩니다. 이 텐서의 모양은 `torch.Size([4])`이며, 차원 수는 1 입니다.
- list_2d: List[List[int]] = [[1, 2, 3], [4, 5, 6]]: 2x3 형태의 중첩된 Python 리스트를 정의합니다. 바깥 리스트의 각 요소가 안쪽 리스트(행)가 됩니다.
- tensor_2d = torch.tensor(list_2d): 중첩 리스트를 2 차원 텐서(행렬)로 변환합니다. PyTorch 는 중첩 구조를 분석하여 자동으로 2 개의 행과 3 개의 열을 가진 `torch.Size([2, 3])` 모양의 텐서를 생성합니다. 차원 수는 2 입니다.

- tensor_3d = torch.tensor(list_3d): 삼중으로 중첩된 리스트를 3 차원 텐서로 변환합니다. 모양은 `torch.Size([2, 2, 2])`가 되며, 이는 2 개의 2x2 행렬이 쌓여있는 형태로 해석할 수 있습니다. 차원 수는 3 입니다.

실행 결과:

--- 0D Tensor (Scalar) ---

Tensor: 7

Shape: torch.Size([])

Dimensions: 0

--- 1D Tensor (Vector) ---

Tensor: tensor([1, 2, 3, 4])

Shape: torch.Size([4])

Dimensions: 1

--- 2D Tensor (Matrix) ---

Tensor:

tensor([[1, 2, 3],

[4, 5, 6]])

Shape: torch.Size([2, 3])

Dimensions: 2

--- 3D Tensor ---

Tensor:

tensor([[[1, 2],

[3, 4]],

[[5, 6],

[7, 8]]])

Shape: torch.Size([2, 2, 2])

Dimensions: 3

실행 결과에 대한 심층 분석:

- **차원과 모양(Shape):** 결과에서 `shape`과 `ndim` 속성을 통해 각 텐서의 구조를 명확히 확인할 수 있습니다. 스칼라는 모양이 없고, 벡터는 길이, 행렬은 (행, 열)의 크기를 가집니다.

딥러닝에서 발생하는 대부분의 오류는 이 '모양'이 맞지 않아 발생하므로, 텐서의 모양을 이해하고 확인하는 습관은 매우 중요합니다.

- **데이터 타입 자동 추론:** 모든 예제에서 입력 데이터가 정수였기 때문에, PyTorch 는 자동으로 데이터 타입을 `torch.int64`(64 비트 정수)로 추론하여 텐서를 생성했습니다. 만약 입력 리스트에 단 하나의 실수라도 포함되어 있었다면(예: `[1, 2, 3.0]`), 전체 텐서의 데이터 타입은 `torch.float32`로 생성되었을 것입니다. 이는 데이터 손실을 막기 위한 PyTorch 의 동작 방식입니다.
- **데이터 복사(독립성):** 이 방법으로 생성된 텐서는 원본 Python 리스트와는 완전히 독립적인 메모리 공간을 가집니다. 따라서 텐서 생성 후에 원본 리스트의 값을 변경해도 텐서에는 아무런 영향이 없습니다. 이는 예측 불가능한 부작용을 방지하는 안전한 방법입니다.

2.2 텐서 생성하기: 데이터로부터 직접 생성: `torch.tensor()` ② - NumPy 연동

데이터 과학 생태계에서 NumPy 는 수치 연산의 표준 라이브러리입니다. Pandas, Scikit-learn, Matplotlib 등 수많은 라이브러리가 NumPy 배열을 기반으로 작동합니다. 따라서 이러한 라이브러리들과 PyTorch 를 연동하기 위해서는 NumPy 배열과 PyTorch 텐서 간의 변환이 필수적입니다.

`torch.tensor()` 함수는 NumPy 배열 또한 인자로 받아, 데이터를 '복사'하여 새로운 PyTorch 텐서를 생성합니다. 이는 원본 NumPy 데이터를 안전하게 보존하면서 PyTorch 연산을 수행하고 싶을 때 유용한 방법입니다.

예제 2-2: NumPy 배열과 PyTorch 텐서 간의 변환 (데이터 복사)

NumPy 배열로부터 텐서를 생성하고, 원본 배열을 수정했을 때 텐서에 영향이 없는지 확인하여 데이터 복사 메커니즘을 증명하는 예제입니다.

```
import torch
import numpy as np

# 1. 2 차원 NumPy 배열 생성
# np.float64 는 64 비트 부동소수점으로, PyTorch 의 torch.float64 와 대응됩니다.
numpy_array = np.array([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]], dtype=np.float64)
print("--- Original NumPy Array ---")
```

```
print(f"NumPy Array:₩n{numpy_array}")
print(f"NumPy Array Dtype: {numpy_array.dtype}")
print(f"Memory Address of NumPy Array: {numpy_array.ctypes.data}₩n")


# 2. torch.tensor()를 사용하여 NumPy 배열로부터 텐서 생성 (데이터 복사)
tensor_from_numpy = torch.tensor(numpy_array)
print("--- Tensor from NumPy (Data Copied) ---")
print(f"Tensor:₩n{tensor_from_numpy}")
print(f"Tensor Dtype: {tensor_from_numpy.dtype}")
# 메모리 주소가 NumPy 배열과 다른 것을 확인하여 데이터가 복사되었음을 증명합니다.
print(f"Memory Address of Tensor: {tensor_from_numpy.data_ptr()}₩n")


# 3. 원본 NumPy 배열의 값을 변경
# 데이터가 복사되었으므로, 이 변경은 텐서에 영향을 주지 않아야 합니다.
print("--- Modifying Original NumPy Array ---")
numpy_array[0, 0] = 99.0
print(f"Modified NumPy Array:₩n{numpy_array}")
print(f"Tensor remains unchanged:₩n{tensor_from_numpy}")
```

코드 라인별 상세 설명:

- import numpy as np: Python 에서 과학 및 수치 계산을 위한 핵심 라이브러리인 NumPy 를 `np`라는 표준 별칭으로 가져옵니다. PyTorch 는 NumPy 와 매우 유사한 API 를 제공하여 NumPy 사용자들이 쉽게 적응할 수 있도록 설계되었습니다.
- numpy_array = np.array(...): 2x3 모양의 2 차원 NumPy 배열을 생성합니다. `dtype=np.float64`를 명시하여 각 요소를 64 비트 부동소수점 숫자로 저장합니다.

- print(f"Memory Address...: {numpy_array.ctypes.data}"): NumPy 배열이 실제로 저장된 메모리의 시작 주소를 출력합니다. 이는 나중에 생성될 텐서의 메모리 주소와 비교하여 데이터 복사 여부를 확인하기 위함입니다.
- tensor_from_numpy = torch.tensor(numpy_array): `torch.tensor()` 함수에 NumPy 배열을 전달하여 PyTorch 텐서를 생성합니다. 이 함수는 기본적으로 **새로운 메모리 공간을 할당**하고 NumPy 배열의 모든 데이터를 그곳으로 **복사**합니다.
- print(f"Tensor Dtype: {tensor_from_numpy.dtype}"): 생성된 텐서의 데이터 타입을 확인합니다. `torch.tensor()`는 원본 NumPy 배열의 데이터 타입(`float64`)을 그대로 유지하여 `torch.float64` 타입의 텐서를 생성합니다.
- print(f"Memory Address...: {tensor_from_numpy.data_ptr()}"): PyTorch 텐서가 저장된 메모리의 시작 주소를 출력합니다. `.data_ptr()` 메소드를 사용합니다. 이 주소는 원본 NumPy 배열의 주소와 다를 것입니다.
- numpy_array[0, 0] = 99.0: 원본 NumPy 배열의 첫 번째 행, 첫 번째 열의 요소를 99.0 으로 변경합니다.
- print(f"Tensor remains unchanged..."): 수정된 NumPy 배열과 이전에 생성된 텐서를 함께 출력하여, 원본 데이터의 변경이 텐서에 영향을 미치지 않았음을 보여줍니다.

실행 결과:

--- Original NumPy Array ---

NumPy Array:

[[1. 2. 3.]

 [4. 5. 6.]]

NumPy Array Dtype: float64

Memory Address of NumPy Array: 22... (실행 시마다 다름)

--- Tensor from NumPy (Data Copied) ---

Tensor:

tensor([[1., 2., 3.],

        [4., 5., 6.]], dtype=torch.float64)

Tensor Dtype: torch.float64

Memory Address of Tensor: 25... (실행 시마다 다름, 위 주소와 다름)

--- Modifying Original NumPy Array ---

Modified NumPy Array:

```
[[99.  2.  3.]
 [ 4.  5.  6.]]
Tensor remains unchanged:
tensor([[1., 2., 3.],
        [4., 5., 6.]], dtype=torch.float64)
```

실행 결과에 대한 심층 분석:

- **메모리 주소의 차이:** 실행 결과에서 NumPy 배열의 메모리 주소와 PyTorch 텐서의 메모리 주소가 다른 것을 명확히 확인할 수 있습니다. 이는 `torch.tensor()`가 데이터를 복사하여 완전히 새로운 객체를 생성했음을 증명하는 결정적인 증거입니다.
- **데이터 독립성(안정성):** 원본 `numpy_array`의 [0, 0] 위치 값이 99.0 으로 변경되었음에도 불구하고, `tensor_from_numpy`의 값은 초기 상태 그대로 `1.0`을 유지하고 있습니다. 이는 데이터 복사로 인해 두 객체가 서로 독립적이기 때문입니다. 이 방식은 한쪽의 수정이 다른 쪽에 예기치 않은 영향을 미치는 부작용(side effect)을 방지하므로, 코드의 안정성을 높여줍니다.
- **언제 사용해야 하는가?:** 원본 데이터를 보존하면서 안전하게 텐서 작업을 시작하고 싶을 때 `torch.tensor()`를 사용하는 것이 좋습니다. 데이터 복사에 따른 약간의 오버헤드가 있지만, 대부분의 경우 이 오버헤드는 무시할 수 있는 수준이며 안정성이라는 큰 이점을 제공합니다. 메모리 공유를 통한 더 빠른 변환이 필요한 경우(파트 2.6 에서 다룰 `torch.from_numpy()`)에는 데이터가 공유된다는 점을 명확히 인지하고 사용해야 합니다.

2.2 텐서 생성하기: 데이터로부터 직접 생성: `torch.tensor()` ③ - `dtype`과 `device` 명시

딥러닝에서는 연산의 정밀도와 메모리 효율성, 그리고 연산 속도를 고려하여 텐서의 데이터 타입과 위치(CPU/GPU)를 신중하게 결정해야 합니다. `torch.tensor()` 함수는 `dtype`과 `device` 인자를 통해 이를 생성 시점에 직접 제어할 수 있는 강력한 기능을 제공합니다.

딥러닝 모델의 가중치나 입력 데이터는 대부분 `float32`를 사용하며, 이는 정밀도와 메모리 사용량, 연산 속도 사이의 가장 좋은 균형을 제공하는 표준 타입입니다. 레이블이나 인덱스 데이터는 보통 `int64`를 사용합니다. `device` 인자를 사용하면 텐서를 처음부터 목표 장치(예: GPU)에 생성하여 불필요한 CPU-GPU 간 데이터 복사를 피할 수 있어 효율적입니다.

예제 2-3: 데이터 타입(dtype)과 장치(device)를 명시적으로 지정하여 생성

다양한 타입의 Python 리스트를 원하는 `dtype`과 `device`를 가진 텐서로 변환하는 예제입니다.

```
import torch

# 사용할 장치를 동적으로 설정 (GPU 가 있으면 cuda, 없으면 cpu)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Target device is set to: {device}₩n")

# 1. 정수 리스트를 float32 타입의 GPU 텐서로 생성
# 딥러닝 모델의 입력이나 가중치는 대부분 float32 를 사용합니다.
data_int = [10, 20, 30]
print("--- Integer data to float32 GPU tensor ---")
tensor_gpu_float32 = torch.tensor(data_int, dtype=torch.float32, device=device)
print(f"Original data: {data_int}")
print(f"Tensor: {tensor_gpu_float32}")
print(f"Dtype: {tensor_gpu_float32.dtype}")
print(f"Device: {tensor_gpu_float32.device}₩n")


# 2. 실수 리스트를 int32 타입의 CPU 텐서로 생성 (소수점 이하 버림)
# 레이블이나 인덱스 등 정수형 데이터에 사용될 수 있습니다.
data_float = [1.1, 2.9, 3.5]
print("--- Float data to int32 CPU tensor ---")
tensor_cpu_int32 = torch.tensor(data_float, dtype=torch.int32, device="cpu")
print(f"Original data: {data_float}")
print(f"Tensor: {tensor_cpu_int32}")
print(f"Dtype: {tensor_cpu_int32.dtype}")
print(f"Device: {tensor_cpu_int32.device}₩n")

# 3. Boolean 텐서를 GPU 에 생성
# 마스킹(masking) 연산 등에서 GPU 가속을 활용할 수 있습니다.
data_bool = [True, False, True]
```

```
print("--- Boolean data to GPU tensor ---")
tensor_gpu_bool = torch.tensor(data_bool, device=device) # dtype=torch.bool 은 자동 추론됨
print(f"Original data: {data_bool}")
print(f"Tensor: {tensor_gpu_bool}")
print(f"Dtype: {tensor_gpu_bool.dtype}")
print(f"Device: {tensor_gpu_bool.device}")
```

코드 라인별 상세 설명:

- device = torch.device(...): 파트 1 에서 배운, 코드의 이식성을 높이는 핵심 패턴입니다. 이 `device` 객체를 `torch.tensor()`의 인자로 전달하여 텐서가 생성될 위치를 지정합니다.
- dtype=torch.float32: 생성될 텐서의 데이터 타입을 32 비트 부동소수점(single-precision)으로 명시적으로 지정합니다. PyTorch 는 자동으로 정수 리스트 `[10, 20, 30]`을 실수 `[10., 20., 30.]`으로 변환합니다.
- device=device: 텐서를 생성할 장치를 지정합니다. 위에서 설정한 `device` 객체를 전달하여, GPU 가 사용 가능한 환경에서는 이 텐서가 처음부터 GPU 메모리에 할당되도록 합니다.
- dtype=torch.int32: 실수형 데이터 `[1.1, 2.9, 3.5]`를 32 비트 정수 타입으로 강제 변환합니다. 이 과정에서 소수점 이하 값은 반올림 없이 **버려집니다(truncation)**. 따라서 `1.1`은 `1`로, `2.9`는 `2`로 변환됩니다. 이러한 데이터 손실 가능성을 항상 인지해야 합니다.
- device="cpu": `torch.device` 객체 대신 문자열 "cpu" 또는 "cuda"를 직접 전달하여 장치를 지정할 수도 있습니다.
- torch.tensor(data_bool, device=device): `dtype`을 명시하지 않으면 PyTorch 가 입력 데이터로부터 타입을 추론합니다. `[True, False, True]`는 명백히 불리언(boolean) 데이터이므로, `torch.bool` 타입으로 자동 설정됩니다.
-

실행 결과 (GPU 사용 가능 시):

```
Target device is set to: cuda


--- Integer data to float32 GPU tensor ---
Original data: [10, 20, 30]
Tensor: tensor([10., 20., 30.], device='cuda:0')
```

Dtype: torch.float32

Device: cuda:0

--- Float data to int32 CPU tensor ---

Original data: [1.1, 2.9, 3.5]

Tensor: tensor([1, 2, 3], dtype=torch.int32)

Dtype: torch.int32

Device: cpu

--- Boolean data to GPU tensor ---

Original data: [True, False, True]

Tensor: tensor([True, False, True], device='cuda:0')

Dtype: torch.bool

Device: cuda:0

실행 결과에 대한 심층 분석:

- **명시적 제어의 결과:** 각 텐서가 `dtype`과 `device` 인자에 지정된 대로 정확하게 생성되었음을 확인할 수 있습니다. `tensor_gpu_float32`는 `float32` 타입과 `cuda:0` 장치를, `tensor_cpu_int32`는 `int32` 타입과 `cpu` 장치를 가집니다.
- **타입 변환과 데이터 손실:** 두 번째 예제에서 `[1.1, 2.9, 3.5]`가 `[1, 2, 3]`으로 변환된 것은 타입 캐스팅(type casting) 시 발생할 수 있는 정보 손실을 명확하게 보여줍니다. 모델의 레이블이나 카테고리 인덱스를 다룰 때는 정수형이 적합하지만, 연속적인 값을 다룰 때 실수형을 정수형으로 변환하는 것은 모델 성능에 치명적일 수 있습니다.
- **실무에서의 중요성:** 모델의 파라미터와 입력 데이터는 모두 동일한 `device`에 있어야 연산이 가능합니다. 또한, 대부분의 딥러닝 연산은 `float` 타입의 입력을 기대합니다. 따라서 텐서를 생성하거나 모델에 입력하기 전에 `dtype`과 `device`를 원하는 상태로 맞추는 것은 디버깅 시간을 줄이고 안정적인 코드를 작성하는 데 매우 중요합니다. `torch.tensor()` 함수는 이 두 가지를 생성 시점에 한 번에 처리할 수 있는 편리한 방법을 제공합니다.

2.2 텐서 생성하기: 특정 크기와 값으로 생성 ①: `torch.zeros()`, `torch.ones()`

데이터 없이, 원하는 모양(shape)과 특정 값(0, 1 등)으로 채워진 텐서를 생성해야 하는 경우가 많습니다. 이는 모델의 가중치를 특정 값으로 초기화하거나, 특정 형태의 마스크(mask) 또는 플레이스홀더(placeholder) 텐서를 만들 때 매우 유용합니다. `torch.zeros()`와 `torch.ones()`는 각각 모든 요소가 0 또는 1 로 채워진 텐서를 생성하는 가장 기본적인 함수입니다.

이 함수들은 `device`와 `dtype` 인자를 모두 지원하여, 텐서를 생성하는 동시에 장치 및 타입 설정을 효율적으로 할 수 있습니다.

예제 2-4: `torch.zeros()`와 `torch.ones()` - 기본 및 고급 사용법

`torch.zeros()`와 `torch.ones()`의 기본 사용법과, `dtype` 및 `device`를 지정하는 고급 활용법, 그리고 실무 예시를 통해 이 함수들을 깊이 있게 살펴봅니다.

```
import torch

# 사용할 장치를 동적으로 설정
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Target device: {device}₩n")

# 1. 기본 사용법: 3x4 크기의, 모든 요소가 0 인 float32 텐서 생성
# dtype 을 명시하지 않으면 기본값인 torch.float32 로 생성됩니다.
print("--- Basic torch.zeros() ---")
zeros_tensor_float = torch.zeros(3, 4)
print(f"Tensor:₩n{zeros_tensor_float}")
print(f"Shape: {zeros_tensor_float.shape}")
print(f"Dtype: {zeros_tensor_float.dtype}₩n")

# 2. 고급 사용법: 2x2x3 크기의, 모든 요소가 1 인 int8 타입의 GPU 텐서 생성
# dtype 과 device 를 명시하여 메모리 사용량과 연산 장치를 제어합니다.
print("--- Advanced torch.ones() with dtype and device ---")
try:
```

```
    ones_tensor_gpu_int = torch.ones(2, 2, 3, dtype=torch.int8, device=device)
    print(f"Tensor:₩n{ones_tensor_gpu_int}")
    print(f"Shape: {ones_tensor_gpu_int.shape}")
    print(f"Dtype: {ones_tensor_gpu_int.dtype}")
    print(f"Device: {ones_tensor_gpu_int.device}₩n")
except Exception as e:
    print(f"Error creating tensor on GPU: {e}")
    print("This might happen if no compatible GPU is available.₩n")

# 3. 실무 활용 예시: 이미지에 대한 마스크(mask) 생성
# 1x3x256x256 크기의 가상 이미지 텐서가 있다고 가정합니다.
# (배치크기, 채널, 높이, 너비)
image_tensor = torch.randn(1, 3, 256, 256)
# 이 이미지의 특정 영역을 가리기 위한 마스크를 생성할 때,
# torch.zeros()를 사용하여 동일한 높이와 너비를 가진 마스크를 만듭니다.
mask = torch.zeros(256, 256, dtype=torch.bool)
# 마스크의 중앙 영역을 True 로 설정 (가리지 않을 영역)
mask[100:150, 100:150] = True
print("--- Practical Example: Creating a Mask ---")
print(f"Shape of original image tensor: {image_tensor.shape}")
print(f"Shape of created mask: {mask.shape}")
print(f"Dtype of mask: {mask.dtype}")
print(f"A part of the mask (center):₩n{mask[98:102, 98:102]}")
```

코드 라인별 상세 설명:

- zeros_tensor_float = torch.zeros(3, 4): 3 행 4 열의 모양을 가지며 모든 요소가 `0.`으로 채워진 텐서를 생성합니다. 인자로 전달된 `3, 4`는 텐서의 모양을 정의하는 가변 인자입니다. `torch.zeros((3, 4))`와 같이 튜플로 전달할 수도 있습니다. `dtype`을 지정하지 않았으므로,

PyTorch 의 전역 기본 데이터 타입(보통 `torch.float32`)이 사용됩니다. 출력에서 `0.`과 같이 소수점이 보이는 것은 부동소수점 타입임을 의미합니다.

- ones_tensor_gpu_int = torch.ones(2, 2, 3, dtype=torch.int8, device=device): 2x2x3 모양의 3D 텐서를 생성하고, 모든 요소를 `1`로 채웁니다. `dtype=torch.int8`을 지정하여 각 요소를 8 비트 정수로 저장하도록 했습니다. 이는 `float32` 대비 1/4 의 메모리만 사용하므로, 0 과 1 만 필요한 마스크 등에서 메모리를 크게 절약할 수 있습니다. `device=device`를 통해 이 텐서를 GPU 메모리에 직접 생성하여 CPU-GPU 간의 불필요한 데이터 전송을 방지합니다.
- image_tensor = torch.randn(1, 3, 256, 256): 딥러닝에서 일반적인 4D 이미지 텐서를 시뮬레이션하기 위해 랜덤 텐서를 생성합니다. 모양은 (배치 크기=1, 채널=3(RGB), 높이=256, 너비=256)을 의미합니다.
- mask = torch.zeros(256, 256, dtype=torch.bool): 이미지의 높이, 너비와 동일한 256x256 크기의 2D 텐서를 생성합니다. `dtype=torch.bool`로 지정하여 `True` 또는 `False` 값을 저장하는 불리언 텐서로 만듭니다. 초기값은 모두 `False`입니다.
- mask[100:150, 100:150] = True: 텐서 슬라이싱을 사용하여 마스크의 특정 영역(y 축 100~149, x 축 100~149)을 선택하고, 그 값을 `True`로 변경합니다. 이는 이미지의 중앙 사각 영역에 해당합니다.

실행 결과 (GPU 사용 가능 시):

```
Target device: cuda
--- Basic torch.zeros() ---
Tensor:
tensor([[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]])
Shape: torch.Size([3, 4])
Dtype: torch.float32
--- Advanced torch.ones() with dtype and device ---
Tensor:
tensor([[[1, 1, 1],
         [1, 1, 1]],
        [[1, 1, 1],
```

```
        [1, 1, 1]]], device='cuda:0', dtype=torch.int8)
Shape: torch.Size([2, 2, 3])
Dtype: torch.int8
Device: cuda:0
--- Practical Example: Creating a Mask ---
Shape of original image tensor: torch.Size([1, 3, 256, 256])
Shape of created mask: torch.Size([256, 256])
Dtype of mask: torch.bool
A part of the mask (center):
tensor([[False, False, False, False],
        [False,  True,  True, False],
        [False,  True,  True, False],
        [False, False, False, False]])
```

실행 결과에 대한 심층 분석:

- **기본값과 명시적 지정의 차이:** `zeros_tensor_float`는 `dtype`과 `device`가 명시되지 않아 기본값(`float32`, `cpu`)으로 생성된 반면, `ones_tensor_gpu_int`는 `int8` 타입과 `cuda:0` 장치가 명시적으로 지정되어 출력에 그대로 반영되었습니다. 이를 통해 텐서 생성 시 원하는 속성을 완벽하게 제어할 수 있음을 알 수 있습니다.
- **실용적 마스크 생성:** 세 번째 예제는 `torch.zeros`가 단순히 0 으로 채워진 텐서를 만드는 것을 넘어, 특정 구조(이미지 크기)에 맞는 '틀'을 만들고, 그 내용을 채워나가는 방식으로 활용될 수 있음을 보여줍니다. 생성된 `mask` 텐서는 나중에 `masked_image = image_tensor * mask`와 같은 연산을 통해 이미지의 특정 부분만 남기거나(True 영역) 지우는(False 영역) 데 사용될 수 있습니다.
- **주의사항:** `torch.zeros`나 `torch.ones`로 생성된 텐서는 `autograd` 기록을 추적하지 않습니다(`requires_grad=False`가 기본값). 만약 이 텐서들이 학습 가능한 파라미터의 일부로 사용되어야 한다면, 생성 시 `requires_grad=True` 인자를 추가해야 합니다.

2.2 텐서 생성하기: 특정 크기와 값으로 생성 ②: `torch.rand()`, `torch.randn()`, `torch.randint()`

랜덤 텐서는 신경망의 가중치를 초기화하거나, 데이터 증강(Data Augmentation)에 노이즈를 추가하거나, 드롭아웃(Dropout)을 구현하는 등 딥러닝의 여러 단계에서 필수적으로 사용됩니다. 각기 다른 분포를 따르는 랜덤 생성 함수의 차이점과 용도를 명확히 이해하는 것이 중요합니다.

- `torch.rand()`: 0 과 1 사이(`[0, 1)`)의 **균등 분포(Uniform Distribution)**에서 값을 추출합니다. 모든 숫자가 선택될 확률이 동일합니다.
- `torch.randn()`: 평균 0, 표준편차 1 의 **표준 정규 분포(Standard Normal Distribution)**에서 값을 추출합니다. 0 주변의 값이 더 높은 확률로 선택됩니다. 신경망 가중치 초기화에 가장 널리 사용됩니다.
- `torch.randint()`: 주어진 범위 내의 **정수(Integer)**를 균등 분포로 추출합니다. 인덱스를 랜덤하게 샘플링할 때 유용합니다.

예제 2-5: 랜덤 텐서 생성 함수 비교 및 시각화

세 가지 랜덤 생성 함수로 대량의 데이터를 생성하고, Matplotlib 을 사용하여 각 데이터의 분포를 히스토그램으로 시각화하여 그 차이를 직관적으로 확인합니다.

```
import torch
import matplotlib.pyplot as plt

# 재현 가능한 결과를 위해 시드 고정
torch.manual_seed(42)

# 1. torch.rand(): [0, 1) 범위의 균등 분포
print("--- 1. torch.rand() ---")
rand_tensor = torch.rand(10000)
print(f"Shape: {rand_tensor.shape}")
print(f"Min value: {rand_tensor.min():.4f}, Max value: {rand_tensor.max():.4f}")
print(f"Mean: {rand_tensor.mean():.4f}, Std: {rand_tensor.std():.4f}₩n")

# 2. torch.randn(): 평균 0, 표준편차 1 의 표준 정규 분포
print("--- 2. torch.randn() ---")
```

```
randn_tensor = torch.randn(10000)
print(f"Shape: {randn_tensor.shape}")
print(f"Min value: {randn_tensor.min():.4f}, Max value: {randn_tensor.max():.4f}")
print(f"Mean: {randn_tensor.mean():.4f}, Std: {randn_tensor.std():.4f}₩n")

# 3. torch.randint(): 특정 범위의 정수를 균등 분포로 생성
print("--- 3. torch.randint() ---")
# 0(포함)부터 10(미포함)까지의 정수 중 랜덤하게 10000 개 선택
randint_tensor = torch.randint(low=0, high=10, size=(10000,))
print(f"Shape: {randint_tensor.shape}")
print(f"Unique values: {randint_tensor.unique()}")

# 2x3 형태의 랜덤 정수 텐서 생성
randint_matrix = torch.randint(0, 100, (2, 3))
print(f"₩nRandom integer matrix (0-99):₩n{randint_matrix}")

# 시각화
fig, axes = plt.subplots(1, 3, figsize=(15, 4))
axes[0].hist(rand_tensor.numpy(), bins=50, color='skyblue', edgecolor='black')
axes[0].set_title("torch.rand() - Uniform")
axes[1].hist(randn_tensor.numpy(), bins=50, color='salmon', edgecolor='black')
axes[1].set_title("torch.randn() - Normal")
axes[2].hist(randint_tensor.numpy(), bins=10, rwidth=0.8, color='lightgreen', edgecolor='black')
axes[2].set_title("torch.randint() - Integer")
fig.tight_layout()
# plt.show() # 로컬 환경에서 실행 시 주석 해제
```

코드 라인별 상세 설명:

- torch.manual_seed(42): 이 코드를 실행하는 모든 사람이 동일한 랜덤 결과를 얻을 수 있도록 난수 생성기의 시드를 42 로 고정합니다. 이는 실험의 재현성을 위해 매우 중요합니다.
- rand_tensor = torch.rand(10000): 0 이상 1 미만(`[0, 1)`) 범위의 균등 분포에서 10,000 개의 실수를 랜덤하게 추출하여 1D 텐서를 생성합니다. 각 숫자가 뽑힐 확률이 이론적으로 동일하므로, 히스토그램이 평평한 모양에 가깝게 나타납니다.
- randn_tensor = torch.randn(10000): 평균이 0 이고 표준편차가 1 인 표준 정규 분포(가우시안 분포)에서 10,000 개의 실수를 랜덤하게 추출합니다. 평균값인 0 주변의 숫자들이 가장 많이 뽑히고, 0 에서 멀어질수록 뽑힐 확률이 급격히 낮아집니다. 히스토그램이 종 모양(bell curve)을 띕니다.
- randint_tensor = torch.randint(low=0, high=10, size=(10000,)): `low`(포함)부터 `high`(미포함) 사이의 정수들로 랜덤하게 채워진 텐서를 지정된 `size`로 생성합니다. 이 경우 0 부터 9 까지의 정수 10 개 중 하나를 랜덤하게 10,000 번 선택합니다. 각 정수가 선택될 확률은 균등합니다.
- plt.hist(...): Matplotlib 라이브러리를 사용하여 텐서의 값 분포를 히스토그램으로 시각화합니다. 시각화는 데이터의 분포를 직관적으로 이해하는 데 매우 효과적인 방법입니다. `.numpy()`는 PyTorch 텐서를 NumPy 배열로 변환하는 메소드로, Matplotlib 과 같은 다른 라이브러리와 연동할 때 필요합니다.

실행 결과 (출력 값):

--- 1. torch.rand() ---

Shape: torch.Size([10000])

Min value: 0.0001, Max value: 0.9999

Mean: 0.5001, Std: 0.2884

--- 2. torch.randn() ---

Shape: torch.Size([10000])

Min value: -4.0219, Max value: 3.9339

Mean: 0.0019, Std: 0.9989

--- 3. torch.randint() ---

Shape: torch.Size([10000])

Unique values: tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

Random integer matrix (0-99):

```
tensor([[83, 37, 58],
        [82, 32, 86]])
```

실행 결과에 대한 심층 분석:

- **분포의 통계적 특성:** `torch.rand`의 평균은 이론값인 0.5 에, 표준편차는 이론값인 `1/sqrt(12) ≈ 0.2887`에 매우 가깝습니다. `torch.randn`의 평균은 0 에, 표준편차는 1 에 매우 가깝습니다. 이는 함수들이 이론적 분포를 잘 따르고 있음을 보여줍니다.
- **시각화를 통한 이해:** 위의 히스토그램은 세 함수의 차이를 명확하게 보여줍니다. `rand`는 평평한 분포, `randn`은 중앙이 솟은 종 모양 분포, `randint`는 각 정수마다 막대가 있는 분포를 보입니다.
- **실무에서의 선택 기준:**
  - **가중치 초기화:** `torch.randn`이 가장 일반적으로 사용됩니다. 이는 학습 초기에 모든 뉴런이 비슷한 값으로 활성화되는 것을 방지(Symmetry Breaking)하고, 기울기 소실/폭주(Vanishing/Exploding Gradients) 문제를 완화하는 데 도움이 되기 때문입니다.
  - **드롭아웃 마스크 생성:** `torch.rand`를 사용하여 0 과 1 사이의 랜덤 텐서를 생성한 뒤, 특정 임계값(예: 0.5)보다 작은 요소는 0, 큰 요소는 1 로 만드는 방식으로 드롭아웃 마스크를 구현할 수 있습니다.
  - **데이터 샘플링:** `torch.randint`는 데이터셋에서 특정 인덱스의 데이터를 랜덤하게 추출(random sampling)할 때 유용하게 사용됩니다.

2.2 텐서 생성하기: 특정 크기와 값으로 생성 ③: 재현성 확보와 시퀀스 생성

랜덤성을 사용하는 연구나 개발에서는 실험 결과를 재현하는 것이 매우 중요합니다. 다른 사람이 내 코드를 실행했을 때, 또는 내가 나중에 동일한 코드를 다시 실행했을 때 정확히 동일한 결과를 얻을 수 있어야 합니다. `torch.manual_seed()`는 이를 가능하게 하는 핵심 도구입니다. 또한, 규칙적인 순서를 갖는 시퀀스 텐서를 생성하는 함수들도 다양한 용도로 활용됩니다.

예제 2-6: `torch.manual_seed()`로 재현성 확보하기

`torch.manual_seed()`를 사용하여 난수 생성기의 시드(seed)를 고정하면, 이후에 생성되는 랜덤 텐서의 수열이 항상 동일하게 유지됩니다.

```
import torch

# 1. 시드(seed)를 42 로 설정
torch.manual_seed(42)
rand1 = torch.rand(2, 2)
print(f"Random tensor with seed 42:₩n{rand1}")

# 2. 시드 설정 없이 다시 생성 (이전 상태에 이어서 생성)
rand2 = torch.rand(2, 2)
print(f"₩nNext random tensor:₩n{rand2}")

# 3. 시드를 다시 42 로 설정
torch.manual_seed(42)
rand3 = torch.rand(2, 2)
print(f"₩nRandom tensor with seed 42 again:₩n{rand3}")
```

실행 결과:

```
Random tensor with seed 42:
tensor([[0.8823, 0.9150],
        [0.3829, 0.9593]])
Next random tensor:
tensor([[0.3904, 0.6009],
        [0.2566, 0.7936]])
Random tensor with seed 42 again:
tensor([[0.8823, 0.9150],
        [0.3829, 0.9593]])
```

결과 해석:

`rand1`과 `rand3`의 값이 정확히 일치하는 것을 볼 수 있습니다. 이는 `torch.manual_seed()`가 난수 생성을 제어하여 재현 가능한 결과를 보장함을 보여줍니다. 연구 논문을 작성하거나, 버그를 재현하거나, 다른 사람과 코드를 공유할 때 스크립트 시작 부분에 `manual_seed`를 설정하는 것은 매우 중요한 습관입니다.

예제 2-7: `torch.arange()`, `torch.linspace()`, `torch.logspace()`

규칙적인 시퀀스 텐서는 반복문 제어, 좌표 생성, 하이퍼파라미터 탐색 등 다양한 용도로 활용됩니다.

```
import torch

# 1. torch.arange(): 주어진 간격(step)으로 균일하게 증가하는 정수 시퀀스 생성
# arange(start, end, step) -> [start, end)
arange_tensor = torch.arange(0, 10, 2)
print(f"Arange (0 to 9, step 2): {arange_tensor}₩n")

# 2. torch.linspace(): 주어진 구간을 원하는 개수(steps)로 균등하게 분할 (끝점 포함)
# linspace(start, end, steps) -> [start, ..., end]
linspace_tensor = torch.linspace(0, 10, steps=5)
print(f"Linspace (0 to 10, 5 steps): {linspace_tensor}₩n")

# 3. torch.logspace(): 주어진 구간을 로그 스케일로 균등하게 분할
# logspace(start, end, steps) -> [10^start, ..., 10^end]
logspace_tensor = torch.logspace(start=-2, end=2, steps=5)
print(f"Logspace (10^-2 to 10^2, 5 steps): {logspace_tensor}")
```

실행 결과:

Arange (0 to 9, step 2): tensor([0, 2, 4, 6, 8])

Linspace (0 to 10, 5 steps): tensor([0.0000, 2.5000, 5.0000, 7.5000, 10.0000])

Logspace (10^-2 to 10^2, 5 steps): tensor([1.0000e-02, 1.0000e-01, 1.0000e+00, 1.0000e+01, 1.0000e+02])

결과 해석:

- `arange`는 정수 인덱스나 반복 횟수를 생성할 때 유용합니다.
- `linspace`는 특정 범위 내에서 균일한 샘플링이 필요할 때, 예를 들어 함수를 그래프로 그리기 위한 x 좌표를 생성할 때 사용됩니다.
- `logspace`는 학습률(learning rate) 범위 탐색과 같이 크기 단위(order of magnitude)가 중요한 값을 탐색해야 할 때 매우 유용합니다.

2.2 텐서 생성하기: 다른 텐서의 속성을 본떠 생성: `_like` 함수들

코드를 작성하다 보면, 기존 텐서의 모양(`shape`), 데이터 타입(`dtype`), 장치(`device`) 등의 속성은 그대로 유지하면서 값만 다르게 채운 새로운 텐서를 만들어야 할 때가 많습니다. 예를 들어, 입력 이미지와 똑같은 크기의 마스크 텐서를 만들거나, 특정 레이어의 출력과 동일한 모양의 노이즈 텐서를 더하는 경우가 그렇습니다.

이때 `torch.zeros_like()`, `torch.ones_like()`, `torch.rand_like()`와 같은 `_like` 함수들을 사용하면 매우 편리합니다. 이 함수들은 입력으로 받은 텐서의 속성을 자동으로 상속받아 새로운 텐서를 생성해 줍니다. 이는 `shape`, `dtype`, `device`를 일일이 하드코딩하는 것을 피하고, 코드의 가독성과 안정성을 높여주는 최고의 실천 방법(best practice) 중 하나입니다.

예제 2-8: `torch.zeros_like()`, `torch.ones_like()`, `torch.rand_like()`

원본 텐서의 속성을 상속받아 새로운 텐서들이 어떻게 생성되는지 확인하는 예제입니다.

```
import torch

# 1. 원본 텐서 생성 (GPU 가 사용 가능하다면 GPU 에 생성)
device = "cuda" if torch.cuda.is_available() else "cpu"
```

```
original_tensor = torch.rand(2, 3, dtype=torch.float64, device=device)
print("--- Original Tensor ---")
print(f"Tensor:₩n{original_tensor}")
print(f"Shape: {original_tensor.shape}, Dtype: {original_tensor.dtype}, Device:
{original_tensor.device}₩n")

# 2. original_tensor 와 같은 속성(shape, dtype, device)을 가지지만 값은 0 으로 채움
# torch.zeros(original_tensor.shape, dtype=original_tensor.dtype, device=original_tensor.device)
와 동일
zeros_like_tensor = torch.zeros_like(original_tensor)
print("--- Zeros_like Tensor ---")
print(f"Tensor:₩n{zeros_like_tensor}")
print(f"Shape: {zeros_like_tensor.shape}, Dtype: {zeros_like_tensor.dtype}, Device:
{zeros_like_tensor.device}₩n")

# 3. original_tensor 와 같은 속성을 가지지만 값은 1 로 채움
ones_like_tensor = torch.ones_like(original_tensor)
print("--- Ones_like Tensor ---")
print(f"Tensor:₩n{ones_like_tensor}")
print(f"Shape: {ones_like_tensor.shape}, Dtype: {ones_like_tensor.dtype}, Device:
{ones_like_tensor.device}₩n")

# 4. original_tensor 와 같은 속성을 가지지만 값은 [0, 1) 범위의 랜덤 값으로 채움
rand_like_tensor = torch.rand_like(original_tensor)
print("--- Rand_like Tensor ---")
print(f"Tensor:₩n{rand_like_tensor}")
print(f"Shape: {rand_like_tensor.shape}, Dtype: {rand_like_tensor.dtype}, Device:
{rand_like_tensor.device}₩n")
```

코드 라인별 상세 설명:

- original_tensor = torch.rand(...): 2x3 모양, `float64` 데이터 타입, 그리고 사용 가능한 장치(CPU 또는 GPU)에 원본 텐서를 생성합니다. 이 텐서가 `_like` 함수들의 기준이 됩니다.
- zeros_like_tensor = torch.zeros_like(original_tensor): `original_tensor`를 인자로 받아, 그 텐서의 `shape`, `dtype`, `device` 속성을 그대로 복사하여 새로운 텐서를 만듭니다. 값만 0 으로 채워집니다. 이 한 줄은 `torch.zeros(original_tensor.shape, dtype=original_tensor.dtype, device=original_tensor.device)`와 같이 길고 오류가 발생하기 쉬운 코드를 대체합니다.
- ones_like_tensor = torch.ones_like(original_tensor): `zeros_like`와 동일하게 작동하지만, 값을 1 로 채웁니다.
- rand_like_tensor = torch.rand_like(original_tensor): `original_tensor`와 동일한 속성을 가지지만, 값은 `[0, 1)` 범위의 균등 분포에서 랜덤하게 채웁니다. `torch.randn_like`도 동일한 방식으로 사용할 수 있습니다.

실행 결과 (GPU 사용 가능 시):

--- Original Tensor ---

Tensor:

tensor([[0.8823, 0.9150, 0.3829],

[0.9593, 0.3904, 0.6009]], device='cuda:0', dtype=torch.float64)

Shape: torch.Size([2, 3]), Dtype: torch.float64, Device: cuda:0

--- Zeros_like Tensor ---

Tensor:

tensor([[0., 0., 0.],

[0., 0., 0.]], device='cuda:0', dtype=torch.float64)

Shape: torch.Size([2, 3]), Dtype: torch.float64, Device: cuda:0

--- Ones_like Tensor ---

Tensor:

tensor([[1., 1., 1.],

[1., 1., 1.]], device='cuda:0', dtype=torch.float64)

Shape: torch.Size([2, 3]), Dtype: torch.float64, Device: cuda:0

--- Rand_like Tensor ---

```
Tensor:
tensor([[0.2566, 0.7936, 0.9408],
        [0.1332, 0.9346, 0.5936]], device='cuda:0', dtype=torch.float64)
Shape: torch.Size([2, 3]), Dtype: torch.float64, Device: cuda:0
```

실행 결과에 대한 심층 분석:

- **속성의 완벽한 상속:** `_like` 함수들로 생성된 모든 텐서들이 원본 텐서와 동일한 모양(torch.Size([2, 3])), 데이터 타입(`torch.float64`), 그리고 장치(`cuda:0`)를 가지는 것을 명확히 확인할 수 있습니다.
- **코드의 견고함과 유연성:** 만약 `original_tensor`의 모양이나 데이터 타입을 변경해야 할 경우(예: 모델 입력 크기 변경, `float32`에서 `float16`으로 혼합 정밀도 훈련 변경), `_like` 함수를 사용한 코드는 전혀 수정할 필요가 없습니다. 자동으로 새로운 속성을 상속받아 올바르게 동작합니다. 이는 코드의 유지보수성을 극대화합니다.
- **실무 활용:** 신경망의 `forward` 메소드 내에서 입력 텐서(`x`)와 동일한 속성을 가진 중간 결과물이나 마스크를 생성해야 할 때 `torch.zeros_like(x)`와 같은 패턴은 거의 표준처럼 사용됩니다.

2.2 텐서 생성하기: 텐서 생성 함수 선택 가이드

지금까지 다양한 텐서 생성 함수들을 살펴보았습니다. 실제 프로젝트에서는 상황에 맞는 최적의 함수를 선택하는 것이 중요합니다. 아래 가이드라인은 여러분이 적절한 함수를 빠르고 정확하게 선택하는 데 도움을 줄 것입니다.

실무 활용 팁: 상황별 텐서 생성 함수 선택 가이드

1. 기존 데이터가 이미 있을 때 (Python 리스트, NumPy 배열 등)

- **데이터를 복사하여 안전하게 독립적인 텐서를 만들고 싶을 때:** torch.tensor(data, dtype=..., device=...) 가장 일반적이고 안전한 방법입니다. 원본 데이터의 수정이 텐서에 영향을 주지 않기를 원할 때 사용합니다. 데이터 타입과 장치를 생성 시점에 명시할 수 있어 편리합니다.
- **NumPy 배열과 메모리를 공유하여 빠르고 효율적으로 변환하고 싶을 때:** torch.from_numpy(numpy_array) 대용량 데이터의 불필요한 복사를 피하고 싶을 때 사용합니다. 단, 한쪽의 수정이 다른 쪽에 영향을 미치는 점을 반드시 인지해야 합니다. CPU 텐서에서만 가능합니다. (자세한 내용은 2.6 절에서 다룹니다.)

2. 데이터 없이, 특정 모양과 값으로 텐서를 생성해야 할 때

- **모든 값이 0 또는 1 인 텐서가 필요할 때:** torch.zeros(shape, ...),torch.ones(shape, ...) 마스크 텐서를 만들거나, 특정 텐서의 초기 상태를 정의할 때 사용합니다.
- **랜덤 값으로 채워진 텐서가 필요할 때:** torch.rand(shape, ...): [0, 1) 균등 분포. 드롭아웃 마스크 생성 등에 활용. torch.randn(shape, ...): 평균 0, 표준편차 1 의 정규 분포. 신경망 가중치 초기화에 가장 널리 사용. torch.randint(low, high, shape, ...): 정수 랜덤 값. 데이터 랜덤 샘플링 등에 활용.
- **규칙적인 순서의 값이 필요할 때:** torch.arange(start, end, step): 정수 시퀀스 생성. torch.linspace(start, end, steps): 균일한 간격의 실수 시퀀스 생성. torch.logspace(start, end, steps): 로그 스케일의 실수 시퀀스 생성 (학습률 탐색 등).

3. 다른 텐서의 속성(모양, 타입, 장치)을 그대로 따르고 싶을 때

- **`_like` 함수들을 적극적으로 사용하세요:** torch.zeros_like(other_tensor) torch.ones_like(other_tensor) torch.rand_like(other_tensor) 이 방법은 코드의 유연성과 안정성을 극대화하는 최고의 실천 방법입니다. 원본 텐서의 속성이 바뀌더라도 코드를 수정할 필요가 없어 유지보수가 매우 용이합니다.

이 가이드라인을 바탕으로, 여러분의 코드에서 텐서 생성 부분을 더욱 간결하고, 효율적이며, 안정적으로 개선해 보시기 바랍니다.

2.3 텐서의 속성(Attributes)

텐서를 생성하고 나면, 그 텐서는 자신의 상태와 구조를 나타내는 여러 유용한 속성(attribute)들을 가지게 됩니다. 이 속성들을 확인하는 것은 모델의 레이어 간 데이터 흐름을 추적하고, 특히 딥러닝에서 가장 흔한 오류 중 하나인 차원 불일치(shape mismatch) 문제를 디버깅하는 데 필수적입니다.

주요 속성 목록

다음은 반드시 알아두어야 할 텐서의 핵심 속성들입니다.

속성	설명	예시
tensor.shape (또는 tensor.size())	텐서의 각 차원의 크기를 튜플 형태로 반환합니다. 텐서의 '모양'을 의미하며, 디버깅 시 가장 많이 사용됩니다.	`torch.Size([64, 3, 224, 224])`
tensor.dtype	텐서에 저장된 데이터의 타입을 나타냅니다. (예: `torch.float32`, `torch.int64`) 연산 시 데이터 타입이 맞지 않으면 오류가 발생할 수 있습니다.	`torch.float32`
tensor.device	텐서가 어느 장치(CPU 또는 GPU)의 메모리에 저장되어 있는지를 나타냅니다. (예: `cpu`, `cuda:0`) 서로 다른 장치에 있는 텐서끼리는 직접 연산할 수 없습니다.	`device(type='cuda', index=0)`
tensor.ndim (또는 tensor.dim())	텐서의 차원 수를 정수로 반환합니다. `len(tensor.shape)`와 같습니다.	`4`
tensor.requires_grad	이 텐서에 대한 기울기를 계산해야 하는지 여부를 나타내는 불리언(boolean) 값입니다. 모델의 학습 가능한 파라미터는 이 값이 `True`로 설정됩니다.	`True`
tensor.grad	`requires_grad`가 `True`인 텐서에 대해 `.backward()`가 호출된 후, 계산된 기울기가 저장되는 속성입니다. 기울기가 계산되기 전에는 `None`입니다.	`tensor([[...]])`
tensor.grad_fn	이 텐서를 생성한 연산(함수)을 가리킵니다. 사용자가 직접 생성한 텐서(leaf tensor)는 이 값이 `None`입니다. 계산 그래프를 추적하는 데 사용됩니다.	`<MulBackward0 object>`
tensor.is_leaf	이 텐서가 계산 그래프의 잎(leaf) 노드인지 여부를 나타냅니다. 사용자가 직접 생성했거나 `requires_grad=False`인 텐서는 `True`입니다.	`True`

예제 2-9: 텐서 속성 확인하기

`requires_grad=True`로 텐서를 생성하고, 간단한 연산을 수행한 후 원본 텐서(잎 노드)와 연산 결과 텐서(중간 노드)의 속성들이 어떻게 다른지 비교해 보겠습니다. 이를 통해 계산 그래프의 개념을 시각적으로 이해할 수 있습니다.

```
import torch

# GPU 가 사용 가능하면 'cuda', 아니면 'cpu'를 device 로 설정
device = "cuda" if torch.cuda.is_available() else "cpu"

# 3x5 크기의 float32 타입 텐서를 생성하고, 기울기 추적을 활성화하며, 지정된 device 로 이동
tensor = torch.randn(3, 5, dtype=torch.float32, device=device, requires_grad=True)

# 연산을 통해 새로운 텐서 생성
new_tensor = tensor * 2

# 텐서의 속성들 출력
print(f"--- Original Tensor (Leaf Node) ---")
print(f"Shape: {tensor.shape}")
print(f"Dtype: {tensor.dtype}")
print(f"Device: {tensor.device}")
print(f"Ndim: {tensor.ndim}")
print(f"Requires Grad?: {tensor.requires_grad}")
print(f"Is Leaf?: {tensor.is_leaf}")
print(f"Gradient: {tensor.grad}")
print(f"Gradient Function: {tensor.grad_fn}")

print(f"₩n--- New Tensor (after operation) ---")
print(f"Shape: {new_tensor.shape}")
print(f"Requires Grad?: {new_tensor.requires_grad}")
```

print(f"Is Leaf?: {new_tensor.is_leaf}")

print(f"Gradient Function: {new_tensor.grad_fn}")

코드 라인별 상세 설명:

- tensor = torch.randn(..., requires_grad=True): `requires_grad=True` 인자를 통해 이 텐서에 대한 모든 연산을 추적하고, 나중에 `.backward()`가 호출될 때 이 텐서에 대한 기울기를 계산하도록 `autograd` 엔진에 지시합니다.
- new_tensor = tensor * 2: `requires_grad=True`인 `tensor`에 연산을 수행하여 `new_tensor`를 생성합니다. 이 곱셈 연산은 계산 그래프에 기록됩니다.
- print(f"Is Leaf?: {tensor.is_leaf}"): 이 텐서가 계산 그래프의 잎(leaf) 노드인지 확인합니다. 사용자가 직접 생성한 텐서는 `True`입니다.
- print(f"Gradient: {tensor.grad}"): 아직 `.backward()`가 호출되지 않았으므로 기울기 값은 `None`입니다.
- print(f"Gradient Function: {tensor.grad_fn}"): `tensor`는 사용자가 직접 생성한 잎(leaf) 텐서이므로, 이 텐서를 생성한 함수가 없어 `grad_fn`이 `None`입니다.
- print(f"Is Leaf?: {new_tensor.is_leaf}"): `new_tensor`는 연산의 결과물이므로 잎 노드가 아니며, `False`가 출력됩니다.
- print(f"Gradient Function: {new_tensor.grad_fn}"): `new_tensor`는 곱셈 연산의 결과로 생성되었으므로, `grad_fn`이 `<MulBackward0>`와 같은 값을 가집니다. 이는 역전파 시 어떤 연산을 수행해야 하는지를 알려주는 '미분 함수'에 대한 참조입니다.

실행 결과 (GPU 사용 가능 시):

--- Original Tensor (Leaf Node) ---

Shape: torch.Size([3, 5])

Dtype: torch.float32

Device: cuda:0

Ndim: 2

Requires Grad?: True

Is Leaf?: True

Gradient: None

Gradient Function: None

--- New Tensor (after operation) ---

Shape: torch.Size([3, 5])

Requires Grad?: True

Is Leaf?: False

Gradient Function: <MulBackward0 object at 0x...>

실행 결과에 대한 심층 분석:

출력된 정보를 통해 `tensor`가 3x5 모양의 32 비트 부동소수점 텐서이며, 첫 번째 GPU(`cuda:0`)에 위치하고, 기울기 계산이 필요한 텐서임을 종합적으로 파악할 수 있습니다. 가장 중요한 차이점은 `grad_fn`과 `is_leaf` 속성입니다. 사용자가 직접 생성한 `tensor`는 계산 그래프의 시작점, 즉 잎 노드이므로 `is_leaf`가 `True`이고 `grad_fn`이 `None`입니다. 반면, 연산의 결과물인 `new_tensor`는 곱셈 연산으로부터 파생되었으므로 `is_leaf`가 `False`이고, 자신을 생성한 연산의 미분 함수(`MulBackward0`)를 `grad_fn`에 기록하고 있습니다. 이것이 바로 PyTorch 가 연산의 역사를 추적하고, 나중에 기울기를 계산할 수 있는 원리입니다.

2.4 텐서 연산(Operations): 인덱싱, 슬라이싱, 결합, 변형

텐서 연산은 PyTorch 의 핵심이며, 딥러닝 모델의 순전파(forward pass)와 역전파(backpropagation)를 구성하는 기본 단위입니다. PyTorch 는 NumPy 와 유사한 풍부한 연산 라이브러리를 제공합니다. 먼저, 데이터를 원하는 형태로 자르고, 붙이고, 바꾸는 조작 연산들을 살펴보겠습니다. 이는 데이터 전처리 및 모델 내부의 데이터 흐름을 제어하는 데 필수적입니다.

예제 2-10: 인덱싱과 슬라이싱 (NumPy 스타일)

NumPy 와 거의 동일한 방식으로 텐서의 특정 요소나 부분에 접근할 수 있습니다. 이는 매우 직관적이고 강력합니다.

```
import torch

# 4x5 크기의 텐서 생성
x = torch.arange(20).reshape(4, 5)
print("Original Tensor (4x5):₩n", x, "₩n")

# 1. 단일 요소 접근: 1 행 2 열의 요소 (0-based index)
print("Element at [1, 2]:", x[1, 2])

# 2. 행 전체 슬라이싱: 2 행 전체
print("Row 2:", x[2])
print("Row 2 (alternative):", x[2, :], "₩n")

# 3. 열 전체 슬라이싱: 3 열 전체
print("Column 3:₩n", x[:, 3], "₩n")

# 4. 부분 슬라이싱: 1~2 행, 2~3 열
sub_tensor = x[1:3, 2:4]
print("Sub-tensor (rows 1-2, cols 2-3):₩n", sub_tensor, "₩n")
```

```
# 5. Boolean 인덱싱: 10 보다 큰 요소만 선택
mask = x > 10
print("Mask (x > 10):₩n", mask, "₩n")
print("Elements > 10:", x[mask])
```

코드 라인별 상세 설명:

- x = torch.arange(20).reshape(4, 5): 0 부터 19 까지의 숫자로 1D 텐서를 만든 후, `reshape(4, 5)`를 통해 4 행 5 열의 2D 텐서로 변형합니다.
- x[1, 2]: 텐서의 1 번 행, 2 번 열에 위치한 요소를 선택합니다. 스칼라 텐서가 반환됩니다.
- x[2]또는 x[2, :]: 2 번 행의 모든 열을 선택합니다. 1D 텐서가 반환됩니다.
- x[:, 3]: 모든 행의 3 번 열을 선택합니다. 1D 텐서가 반환됩니다.
- x[1:3, 2:4]: 1 번 행부터 3 번 행 전까지(즉, 1, 2 번 행), 그리고 2 번 열부터 4 번 열 전까지(즉, 2, 3 번 열)를 선택하여 2x2 크기의 부분 텐서를 만듭니다.
- mask = x > 10: 텐서 `x`의 각 요소에 대해 `> 10` 비교 연산을 수행하여, 조건이 참이면 `True`, 거짓이면 `False`인 불리언 텐서(마스크)를 생성합니다.
- x[mask]: 생성된 마스크를 인덱스로 사용하여, 마스크에서 `True`인 위치의 요소들만 선택하여 1D 텐서로 반환합니다.

실행 결과:

```
Original Tensor (4x5):
 tensor([[ 0,  1,  2,  3,  4],
         [ 5,  6,  7,  8,  9],
         [10, 11, 12, 13, 14],
         [15, 16, 17, 18, 19]])
Element at [1, 2]: tensor(7)
Row 2: tensor([10, 11, 12, 13, 14])
Row 2 (alternative): tensor([10, 11, 12, 13, 14])
Column 3:
 tensor([ 3,  8, 13, 18])
Sub-tensor (rows 1-2, cols 2-3):
 tensor([[ 7,  8],
```

```
        [12, 13]])
Mask (x > 10):
 tensor([[False, False, False, False, False],
        [False, False, False, False, False],
        [False,  True,  True,  True,  True],
        [ True,  True,  True,  True,  True]])

Elements > 10: tensor([11, 12, 13, 14, 15, 16, 17, 18, 19])
```

예제 2-11: `torch.cat()` - 텐서 이어 붙이기

여러 텐서를 하나의 차원을 따라 길게 이어 붙이는 연산입니다. `torch.cat()`은 배치(batch)를 합치거나 여러 소스에서 나온 피처(feature)들을 결합할 때 필수적으로 사용됩니다. 이어 붙이려는 차원을 제외한 나머지 차원들의 크기는 모두 동일해야 합니다.

```
import torch

t1 = torch.arange(6).reshape(2, 3)
t2 = torch.arange(6, 12).reshape(2, 3)
print("t1 (2x3):₩n", t1)
print("t2 (2x3):₩n", t2, "₩n")

# 1. dim=0: 첫 번째 차원(행)을 따라 이어 붙입니다.
# (2, 3)과 (2, 3) 텐서를 행 방향으로 합쳐 (4, 3) 텐서를 만듭니다.
cat_dim0 = torch.cat([t1, t2], dim=0)
print("--- torch.cat([t1, t2], dim=0) ---")
print("Result shape:", cat_dim0.shape)
print(cat_dim0, "₩n")

# 2. dim=1: 두 번째 차원(열)을 따라 이어 붙입니다.
# (2, 3)과 (2, 3) 텐서를 열 방향으로 합쳐 (2, 6) 텐서를 만듭니다.
```

```
cat_dim1 = torch.cat([t1, t2], dim=1)
print("--- torch.cat([t1, t2], dim=1) ---")
print("Result shape:", cat_dim1.shape)
print(cat_dim1, "₩n")
```

코드 라인별 상세 설명:

- torch.cat(tensors, dim=0): 텐서들의 리스트 `tensors`를 `dim`으로 지정된 차원을 따라 이어 붙입니다.
- dim=0: 첫 번째 차원(행)을 기준으로 `t2`가 `t1`의 아래에 붙습니다. 결과 텐서의 행의 개수는 `t1.shape[0] + t2.shape[0]`이 됩니다.
- dim=1: 두 번째 차원(열)을 기준으로 `t2`가 `t1`의 오른쪽에 붙습니다. 결과 텐서의 열의 개수는 `t1.shape[1] + t2.shape[1]`이 됩니다.

실행 결과:

```
t1 (2x3):
 tensor([[0, 1, 2],
         [3, 4, 5]])
t2 (2x3):
 tensor([[ 6,  7,  8],
         [ 9, 10, 11]])
--- torch.cat([t1, t2], dim=0) ---
Result shape: torch.Size([4, 3])
tensor([[ 0,  1,  2],
         [ 3,  4,  5],
         [ 6,  7,  8],
         [ 9, 10, 11]])
--- torch.cat([t1, t2], dim=1) ---
Result shape: torch.Size([2, 6])
tensor([[ 0,  1,  2,  6,  7,  8],
         [ 3,  4,  5,  9, 10, 11]])
```

예제 2-12: `torch.stack()` - 새로운 차원으로 텐서 쌓기

`torch.stack()`은 `torch.cat()`과 비슷해 보이지만 근본적으로 다른 연산을 수행합니다. `cat`이 기존 차원을 확장하는 반면, `stack`은 **새로운 차원**을 만들어 텐서들을 쌓습니다. 따라서 `stack`을 사용하려면 모든 텐서의 모양이 완전히 동일해야 하며, 결과 텐서의 차원 수는 1 증가합니다.

이 연산은 여러 개의 단일 데이터 샘플(예: 이미지)을 묶어 하나의 배치(batch) 텐서를 만들 때 핵심적으로 사용됩니다.

```
import torch

t1 = torch.arange(6).reshape(2, 3)
t2 = torch.arange(6, 12).reshape(2, 3)
print("t1 (2x3):₩n", t1)
print("t2 (2x3):₩n", t2, "₩n")

# 1. dim=0: 새로운 0 번 차원을 만들어 쌓습니다.
# (2, 3) 텐서 두 개를 쌓아 (2, 2, 3) 텐서를 만듭니다.
stack_dim0 = torch.stack([t1, t2], dim=0)
print("--- torch.stack([t1, t2], dim=0) ---")
print("Result shape:", stack_dim0.shape)
print(stack_dim0, "₩n")

# 2. dim=1: 새로운 1 번 차원을 만들어 쌓습니다.
# (2, 3) 텐서 두 개를 쌓아 (2, 2, 3) 텐서를 만듭니다.
stack_dim1 = torch.stack([t1, t2], dim=1)
print("--- torch.stack([t1, t2], dim=1) ---")
print("Result shape:", stack_dim1.shape)
print(stack_dim1, "₩n")

# 3. dim=2: 새로운 2 번 차원을 만들어 쌓습니다.
# (2, 3) 텐서 두 개를 쌓아 (2, 3, 2) 텐서를 만듭니다.
```

```
stack_dim2 = torch.stack([t1, t2], dim=2)
print("--- torch.stack([t1, t2], dim=2) ---")
print("Result shape:", stack_dim2.shape)
print(stack_dim2)
```

`cat` vs `stack` 핵심 차이 요약:

- torch.cat(Concatenate): 텐서들을 **기존 차원**에 이어 붙입니다. 차원 수는 변하지 않습니다.
- torch.stack: 텐서들을 **새로운 차원**으로 쌓습니다. 차원 수가 1 증가합니다.

실행 결과:

```
t1 (2x3):
 tensor([[0, 1, 2],
         [3, 4, 5]])
t2 (2x3):
 tensor([[ 6,  7,  8],
         [ 9, 10, 11]])
--- torch.stack([t1, t2], dim=0) ---
Result shape: torch.Size([2, 2, 3])
tensor([[[ 0,  1,  2],
         [ 3,  4,  5]],
        [[ 6,  7,  8],
         [ 9, 10, 11]]])
--- torch.stack([t1, t2], dim=1) ---
Result shape: torch.Size([2, 2, 3])
tensor([[[ 0,  1,  2],
         [ 6,  7,  8]],
        [[ 3,  4,  5],
         [ 9, 10, 11]]])
```

```
--- torch.stack([t1, t2], dim=2) ---
Result shape: torch.Size([2, 3, 2])
tensor([[[ 0,  6],
         [ 1,  7],
         [ 2,  8]],
        [[ 3,  9],
         [ 4, 10],
         [ 5, 11]]])
```

결과 해석:

`dim` 인자에 따라 새로운 차원이 어느 위치에 삽입되는지, 그리고 데이터가 어떻게 재배열되는지 주의 깊게 살펴보세요. `stack(dim=0)`은 `t1`과 `t2`를 책처럼 쌓아, 2 개의 페이지를 가진 책을 만든 것과 같습니다. `DataLoader`가 여러 개의 `(C, H, W)` 모양의 이미지를 묶어 `(N, C, H, W)` 모양의 배치 텐서를 만드는 것이 바로 이 원리입니다.

squeeze() 와 unsqueeze()

딥러닝 모델의 레이어들은 특정한 차원의 입력을 기대합니다. 예를 들어, 컨볼루션 레이어는 배치, 채널, 높이, 너비의 4D 텐서를, 선형 레이어는 배치, 피처의 2D 텐서를 기대합니다. `squeeze()`와 `unsqueeze()`는 크기가 1 인 불필요한 차원을 제거하거나, 특정 위치에 크기가 1 인 차원을 추가하여 레이어 간 차원을 맞추는 데 매우 중요한 역할을 합니다.

```
import torch

# 1x1x3x1x4 크기의 텐서 생성
x = torch.randn(1, 1, 3, 1, 4)
print(f"Original tensor shape: {x.shape}₩n")

# 1. torch.squeeze(): 크기가 1 인 모든 차원을 제거
squeezed_all = torch.squeeze(x)
print("--- Squeeze all dimensions of size 1 ---")
print(f"Shape after squeeze(): {squeezed_all.shape}₩n")

# 2. torch.squeeze(dim): 특정 위치에서 크기가 1 인 차원만 제거
# dim=1 위치의 차원을 제거. (1, 1, 3, 1, 4) -> (1, 3, 1, 4)
squeezed_dim1 = torch.squeeze(x, dim=1)
print("--- Squeeze dimension at index 1 ---")
print(f"Shape after squeeze(dim=1): {squeezed_dim1.shape}₩n")

# 3. torch.unsqueeze(dim): 특정 위치에 크기가 1 인 차원을 추가
# squeezed_all (3, 4) 텐서를 기준으로 작업
base_tensor = squeezed_all
print(f"Base tensor shape: {base_tensor.shape}₩n")

# dim=0 위치에 차원 추가. (3, 4) -> (1, 3, 4)
# 이는 단일 샘플을 배치(batch)로 만들 때 흔히 사용됩니다.
```

```
unsqueezed_dim0 = torch.unsqueeze(base_tensor, dim=0)
print("--- Unsqueeze at dimension 0 ---")
print(f"Shape after unsqueeze(dim=0): {unsqueezed_dim0.shape}₩n")

# dim=2 위치에 차원 추가. (3, 4) -> (3, 4, 1)
# 이는 채널(channel) 차원을 추가할 때 사용될 수 있습니다.
unsqueezed_dim2 = torch.unsqueeze(base_tensor, dim=2)
print("--- Unsqueeze at dimension 2 ---")
print(f"Shape after unsqueeze(dim=2): {unsqueezed_dim2.shape}")
```

코드 라인별 상세 설명:

- torch.squeeze(x): 텐서 `x`의 모양 `(1, 1, 3, 1, 4)`에서 크기가 1 인 모든 차원(0, 1, 3 번 인덱스)을 제거하여 `(3, 4)` 모양의 텐서를 반환합니다.
- torch.squeeze(x, dim=1): `dim` 인자를 사용하여 특정 위치의 차원만 제거하도록 지정합니다. 1 번 인덱스의 차원 크기가 1 이므로 제거되어 `(1, 3, 1, 4)`가 됩니다. 만약 지정된 차원의 크기가 1 이 아니면 아무 변화도 일어나지 않습니다.
- torch.unsqueeze(base_tensor, dim=0): `(3, 4)` 모양의 텐서에 0 번 인덱스 위치에 새로운 차원을 추가하여 `(1, 3, 4)`로 만듭니다. 이는 딥러닝에서 단일 이미지(높이, 너비)나 단일 문장(시퀀스 길이, 피처)을 모델에 입력하기 위해 배치 차원(batch dimension)을 추가하는 가장 일반적인 방법입니다.
- torch.unsqueeze(base_tensor, dim=2): `(3, 4)` 모양의 텐서에 2 번 인덱스 위치(마지막)에 새로운 차원을 추가하여 `(3, 4, 1)`로 만듭니다. 흑백 이미지를 다룰 때 채널 차원을 명시적으로 추가하는 경우 등에 사용됩니다.
-

실행 결과:

```
Original tensor shape: torch.Size([1, 1, 3, 1, 4])
--- Squeeze all dimensions of size 1 ---
Shape after squeeze(): torch.Size([3, 4])
--- Squeeze dimension at index 1 ---
Shape after squeeze(dim=1): torch.Size([1, 3, 1, 4])
```

Base tensor shape: torch.Size([3, 4])

--- Unsqueeze at dimension 0 ---

Shape after unsqueeze(dim=0): torch.Size([1, 3, 4])

--- Unsqueeze at dimension 2 ---

Shape after unsqueeze(dim=2): torch.Size([3, 4, 1])

예제 2-14: `view()`, `reshape()` - 텐서 모양 변경

텐서의 모양을 바꾸는 것은 모델 내에서 데이터의 형태를 맞추기 위해 필수적입니다. 가장 대표적인 예는 컨볼루션 레이어의 다차원 출력을 선형 레이어에 입력하기 위해 1 차원 벡터로 평탄화(flatten)하는 것입니다. `view()`와 `reshape()`는 이 역할을 수행하며, 둘은 비슷하지만 중요한 차이점이 있습니다.

- `view()`: 텐서의 모양을 바꾸지만, 메모리에 저장된 데이터의 순서는 바꾸지 않습니다. 따라서 텐서가 메모리 상에서 연속적(contiguous)일 때만 작동합니다. 원본 텐서와 항상 메모리를 공유하므로 데이터 복사가 발생하지 않아 빠릅니다.
- `reshape()`: `view()`와 동일한 기능을 하지만, 텐서가 비연속적일 경우 자동으로 데이터를 복사하여 연속적으로 만든 후 모양을 변경합니다. 더 안전하고 편리하지만, 의도치 않은 데이터 복사가 발생할 수 있습니다.

```
import torch

x = torch.arange(12)
print("Original 1D tensor:", x)
print(f"Is contiguous? {x.is_contiguous()}₩n")

# 1. view(): 텐서의 모양을 변경.
x_view = x.view(3, 4)
print("--- x.view(3, 4) ---")
print("Shape:", x_view.shape)
print(x_view, "₩n")
```

```
# 2. reshape(): view()와 유사하게 작동.
x_reshaped = x.reshape(2, 6)
print("--- x.reshape(2, 6) ---")
print("Shape:", x_reshaped.shape)
print(x_reshaped, "₩n")

# 3. view()와 reshape()의 차이점: 비연속적인 텐서
# .T (transpose) 연산은 텐서를 비연속적으로 만듭니다.
transposed_x = x_view.T
print("--- Transposed Tensor ---")
print(f"Transposed shape: {transposed_x.shape}")
print(f"Is contiguous? {transposed_x.is_contiguous()}₩n")

try:
    # 비연속적인 텐서에 view()를 사용하면 오류 발생
    transposed_x.view(12)
except RuntimeError as e:
    print("Error with view():", e)

# reshape()는 비연속적인 텐서에도 작동 (내부적으로 .contiguous() 호출 후 view 실행)
reshaped_from_transposed = transposed_x.reshape(12)
print("₩nShape after reshape() on transposed tensor:", reshaped_from_transposed.shape)
print("Reshaped tensor:", reshaped_from_transposed)
```

결과 해석:

`x_view`를 전치(transpose)한 `transposed_x`는 메모리 상에서 데이터가 더 이상 순서대로 놓여있지 않은 비연속적인 텐서가 됩니다. 이 텐서에 `view()`를 사용하면 `RuntimeError`가 발생합니다. 반면,

`reshape()`는 내부적으로 데이터를 연속적인 메모리 공간에 복사하는 과정을 거치므로 정상적으로 작동합니다. 따라서, 텐서의 메모리 레이아웃을 확신할 수 없을 때는 `reshape()`를 사용하는 것이 더 안전합니다.

예제 2-15: permute - 차원의 순서 변경

`view`나 `reshape`가 텐서의 모양을 바꾸는 것이라면, `permute`는 텐서의 **차원의 순서** 자체를 바꾸는 연산입니다. 데이터 자체를 재배열하는 것이 아니라, 각 차원을 가리키는 방식(stride)을 변경합니다. 결과 텐서는 대부분 비연속적이 되며, 원본 텐서와 메모리를 공유합니다.

이 연산은 특히 이미지 데이터의 차원 순서를 바꿀 때 매우 유용합니다. PyTorch 의 컨볼루션 레이어는 `(N, C, H, W)` (배치, 채널, 높이, 너비) 순서의 입력을 기대하지만, Matplotlib 과 같은 시각화 라이브러리는 `(N, H, W, C)` 순서를 기대하기 때문입니다.

```
import torch

# (Batch, Channel, Height, Width) 모양의 가상 이미지 텐서
image_tensor_chw = torch.randn(32, 3, 28, 28)
print(f"Original image tensor shape (N, C, H, W): {image_tensor_chw.shape}₩n")

# 1. permute()를 사용하여 차원의 순서를 변경
# (0, 1, 2, 3) -> (0, 2, 3, 1)
# N, C, H, W -> N, H, W, C
image_tensor_hwc = image_tensor_chw.permute(0, 2, 3, 1)
print("--- image_tensor.permute(0, 2, 3, 1) ---")
print(f"Permuted tensor shape (N, H, W, C): {image_tensor_hwc.shape}₩n")

# 2. permute()는 메모리를 공유함
# 원본 텐서와 permute 된 텐서의 메모리 주소가 동일한지 확인
print(f"Memory address of original: {image_tensor_chw.data_ptr()}")
print(f"Memory address of permuted: {image_tensor_hwc.data_ptr()}")
```

```
print(f"Are they sharing storage? {image_tensor_chw.storage().data_ptr() ==
image_tensor_hwc.storage().data_ptr()}₩n")

# 3. permute() 결과는 비연속적(non-contiguous)임
print(f"Is original tensor contiguous? {image_tensor_chw.is_contiguous()}")
print(f"Is permuted tensor contiguous? {image_tensor_hwc.is_contiguous()}₩n")

# 4. 비연속적인 텐서에 view()를 사용하려면 .contiguous()를 먼저 호출해야 함
try:
    image_tensor_hwc.view(32, -1)
except RuntimeError as e:
    print("Error with view() on permuted tensor:", e)

# .contiguous()를 호출하여 메모리 상에서 연속적인 복사본을 만든 후 view() 사용
contiguous_tensor = image_tensor_hwc.contiguous()
flattened_tensor = contiguous_tensor.view(32, -1)
print("₩nSuccessfully flattened after .contiguous():")
print(f"Shape: {flattened_tensor.shape}")
```

결과 해석: `permute` 연산 후 텐서의 모양은 바뀌었지만, 메모리 주소는 동일하여 데이터를 공유함을 알 수 있습니다. 그러나 메모리 레이아웃이 변경되어 텐서는 비연속적이 됩니다. 따라서 `permute`된 텐서에 `view`를 사용하려면, `.contiguous()`를 호출하여 데이터를 연속적인 메모리 블록에 새로 복사한 후 사용해야 합니다. `reshape`는 이 과정을 자동으로 처리해 줍니다.

2.4 텐서 연산(Operations): 수학 및 논리 연산

딥러닝 모델의 핵심인 수학적 연산들을 다룹니다. 요소별 연산, 리덕션 연산, 비교 연산, 행렬 연산으로 나누어 살펴봅니다.

예제 2-16: 요소별(Element-wise) 연산

텐서의 같은 위치에 있는 요소들끼리 연산을 수행합니다. 연산에 참여하는 텐서들의 모양이 같거나, 브로드캐스팅이 가능해야 합니다. 덧셈, 뺄셈, 곱셈, 나눗셈 등이 포함됩니다.

```
import torch

a = torch.tensor([[1, 2], [3, 4]])
b = torch.tensor([[5, 6], [7, 8]])
print("a:₩n", a)
print("b:₩n", b, "₩n")

# 덧셈
print("--- Addition ---")
print("a + b:₩n", a + b)
print("torch.add(a, b):₩n", torch.add(a, b), "₩n")

# 뺄셈
print("--- Subtraction ---")
print("a - b:₩n", a - b, "₩n")

# 곱셈 (요소별 곱)
print("--- Element-wise Multiplication ---")
print("a * b:₩n", a * b)
print("torch.mul(a, b):₩n", torch.mul(a, b), "₩n")

# 나눗셈
print("--- Division ---")
print("b / a:₩n", b / a, "₩n")

# 거듭제곱
```

```
print("--- Power ---")
print("a ** 2:₩n", a ** 2)
```

실행 결과:

```
a:
 tensor([[1, 2],
         [3, 4]])
b:
 tensor([[5, 6],
         [7, 8]])
--- Addition ---
a + b:
 tensor([[ 6,  8],
         [10, 12]])
torch.add(a, b):
 tensor([[ 6,  8],
         [10, 12]])
--- Subtraction ---
a - b:
 tensor([[-4, -4],
         [-4, -4]])
--- Element-wise Multiplication ---
a * b:
 tensor([[ 5, 12],
         [21, 32]])
torch.mul(a, b):
 tensor([[ 5, 12],
         [21, 32]])
--- Division ---
```

```
b / a:
 tensor([[5.0000, 3.0000],
         [2.3333, 2.0000]])
--- Power ---
a ** 2:
 tensor([[ 1,  4],
         [ 9, 16]])
```

결과 해석:

- 연산자(`+`, `*`)를 사용하는 것과 PyTorch 함수(`torch.add`, `torch.mul`)를 사용하는 것은 동일한 결과를 냅니다.
- `a * b`는 행렬 곱셈이 아닌 **요소별 곱셈**이라는 점에 반드시 주의해야 합니다. 행렬 곱셈은 `torch.matmul` 또는 `@` 연산자를 사용합니다.
- PyTorch 는 연산 시 자동으로 데이터 타입을 승격(promote)합니다. 정수 텐서 간의 나눗셈 결과가 `float` 타입으로 나온 것이 그 예입니다.

예제 2-17: 리덕션(Reduction) 연산

텐서의 여러 값을 하나의 값(또는 더 적은 수의 값)으로 줄이는(reduce) 연산입니다. 텐서 전체에 대해 수행하거나, 특정 차원(`dim`)을 따라 수행할 수 있습니다. 합계, 평균, 최댓값, 최솟값 등이 포함됩니다.

```
import torch

x = torch.tensor([[1., 8., 3.], [6., 2., 9.]])
print("x (2x3):₩n", x, "₩n")

# 1. 전체 합계, 평균, 최댓값
print("--- Global Reduction ---")
print(f"Sum of all elements: {torch.sum(x)}")
```

```
print(f"Mean of all elements: {x.mean()}")
print(f"Max value of all elements: {x.max()}₩n")

# 2. 차원별 리덕션
# dim=0: 각 열(column)에 대해 연산. (2, 3) -> (3,)
sum_dim0 = x.sum(dim=0)
print("--- Reduction along dim=0 (columns) ---")
print("Sum of each column:", sum_dim0)
print("Shape:", sum_dim0.shape, "₩n")

# dim=1: 각 행(row)에 대해 연산. (2, 3) -> (2,)
mean_dim1 = x.mean(dim=1)
print("--- Reduction along dim=1 (rows) ---")
print("Mean of each row:", mean_dim1)
print("Shape:", mean_dim1.shape, "₩n")

# 3. max/min 연산의 추가 반환 값 (indices)
# dim=1 을 따라 각 행의 최댓값과 그 인덱스를 찾음
max_result_dim1 = x.max(dim=1)
print("--- max() with indices ---")
print("Max result object:", max_result_dim1)
print("Max values:", max_result_dim1.values)
print("Max indices:", max_result_dim1.indices)
```

코드 라인별 상세 설명:

- torch.sum(x)또는 x.sum(): 텐서의 모든 요소를 더하여 하나의 스칼라 텐서를 반환합니다.
- x.sum(dim=0): `dim=0`(행 방향)을 따라 요소를 더합니다. 즉, 각 열의 합계를 계산합니다. `(2, 3)` 모양의 텐서가 `(3,)` 모양의 텐서로 줄어듭니다.

- x.max(dim=1): `dim=1`(열 방향)을 따라 각 행의 최댓값을 찾습니다. 이 함수는 두 개의 텐서를 담은 명명된 튜플(named tuple)을 반환합니다:이러한 동작은 분류 모델의 출력(logits)에서 가장 높은 확률을 가진 클래스를 찾을 때(`torch.argmax` 또는 `torch.max`) 매우 유용합니다.
  - `values`: 각 행에서 찾은 최댓값들을 담은 텐서
  - `indices`: 각 행에서 최댓값이 위치한 인덱스(열 번호)를 담은 텐서

실행 결과:

x (2x3):

tensor([[1., 8., 3.],

[6., 2., 9.]])

--- Global Reduction ---

Sum of all elements: 29.0

Mean of all elements: 4.833333492279053

Max value of all elements: 9.0

--- Reduction along dim=0 (columns) ---

Sum of each column: tensor([7., 10., 12.])

Shape: torch.Size([3])

--- Reduction along dim=1 (rows) ---

Mean of each row: tensor([4.0000, 5.6667])

Shape: torch.Size([2])

--- max() with indices ---

Max result object: torch.return_types.max(

values=tensor([8., 9.]),

indices=tensor([1, 2]))

Max values: tensor([8., 9.])

Max indices: tensor([1, 2])

결과 해석:

`dim=0`으로 리덕션하면 해당 차원이 사라집니다. `(2, 3)`에서 0 번 차원이 사라져 `(3,)`이 됩니다. `x.max(dim=1)`의 결과에서, 첫 번째 행 `[1., 8., 3.]`의 최댓값은 `8.`이고 인덱스는 `1`입니다. 두 번째 행 `[6., 2., 9.]`의 최댓값은 `9.`이고 인덱스는 `2`입니다.

예제 2-18: 행렬 연산

선형대수의 핵심인 행렬 곱셈은 모든 딥러닝 모델의 기본 연산입니다. PyTorch 는 이를 위한 직관적인 방법을 제공합니다.

```
import torch

# 행렬 곱셈 (Matrix Multiplication)
mat1 = torch.randn(3, 4)
mat2 = torch.randn(4, 5)
print(f"mat1 shape: {mat1.shape}")
print(f"mat2 shape: {mat2.shape}₩n")

# 1. torch.matmul() 사용
result_matmul = torch.matmul(mat1, mat2)
print("--- torch.matmul(mat1, mat2) ---")
print("Result shape:", result_matmul.shape)
print(result_matmul, "₩n")

# 2. @ 연산자 사용 (Python 3.5+ 에서 지원, 권장)
result_at = mat1 @ mat2
print("--- mat1 @ mat2 ---")
print("Result shape:", result_at.shape)
print(result_at, "₩n")
```

```
# 3. 전치(Transpose)
x = torch.tensor([[1, 2, 3], [4, 5, 6]])
print("--- Transpose ---")
print("Original x (2x3):₩n", x)
print("x.T (3x2):₩n", x.T)
print("x.transpose(0, 1) (3x2):₩n", x.transpose(0, 1))
```

코드 라인별 상세 설명:

- torch.matmul(mat1, mat2): `(3, 4)` 모양의 `mat1`과 `(4, 5)` 모양의 `mat2`를 행렬 곱셈합니다. 내적 차원(4)이 일치해야 하며, 결과 텐서의 모양은 `(3, 5)`가 됩니다.
- mat1 @ mat2: `torch.matmul`과 완전히 동일한 연산을 수행하는 연산자입니다. 코드가 더 간결하고 가독성이 높아 선호됩니다.
- x.T: 텐서의 차원을 뒤집어 전치(transpose)합니다. 2D 텐서의 경우 행과 열을 바꿉니다. `(2, 3)` -> `(3, 2)`.
- x.transpose(0, 1): 두 차원의 순서를 명시적으로 바꿉니다. 2D 텐서의 경우 `x.T`와 동일합니다. 3D 이상의 텐서에서 특정 두 차원만 바꾸고 싶을 때 유용합니다.

`*` vs `@`: 가장 흔한 실수!

초보자들이 가장 많이 하는 실수 중 하나는 행렬 곱셈을 의도하고 `*` 연산자를 사용하는 것입니다. `*`는 요소별 곱셈이며, `@`가 행렬 곱셈입니다. 이 둘을 혼동하면 디버깅하기 어려운 차원 불일치 오류나 논리적 오류가 발생합니다.

실행 결과 (랜덤 값은 실행 시마다 다름):

```
mat1 shape: torch.Size([3, 4])
mat2 shape: torch.Size([4, 5])
--- torch.matmul(mat1, mat2) ---
Result shape: torch.Size([3, 5])
tensor([[ 0.3134, -0.0739, -0.3752, -0.1383,  0.4763],
        [-0.5113,  0.3414,  0.5317,  0.0902, -0.1987],
        [ 0.4934, -0.2148, -0.4133, -0.1032,  0.2784]])
--- mat1 @ mat2 ---
```

```
Result shape: torch.Size([3, 5])
tensor([[ 0.3134, -0.0739, -0.3752, -0.1383,  0.4763],
        [-0.5113,  0.3414,  0.5317,  0.0902, -0.1987],
        [ 0.4934, -0.2148, -0.4133, -0.1032,  0.2784]])
--- Transpose ---
Original x (2x3):
 tensor([[1, 2, 3],
        [4, 5, 6]])
x.T (3x2):
 tensor([[1, 4],
        [2, 5],
        [3, 6]])
x.transpose(0, 1) (3x2):
 tensor([[1, 4],
        [2, 5],
        [3, 6]])
```

예제 2-19: In-place 연산

PyTorch 의 많은 연산에는 `_` 접미사가 붙는 버전이 있습니다. 예를 들어, `add()`에 대응하는 `add_()`가 있습니다. 이들은 **In-place 연산**이라고 불리며, 새로운 텐서를 반환하는 대신 텐서 자기 자신을 직접 수정합니다.

장점: 새로운 메모리를 할당하지 않으므로 메모리를 절약할 수 있습니다.

단점: `autograd`는 기울기 계산을 위해 순전파 시의 중간 값들을 저장해 두어야 합니다. In-place 연산은 이 중간 값들을 덮어 써버릴 수 있기 때문에, 역전파 과정에서 문제를 일으킬 수 있습니다. 따라서, 기울기 계산이 필요한 텐서에는 In-place 연산을 사용하지 않는 것이 안전합니다.

```
import torch

# 1. 일반 연산 (새로운 텐서 반환)
a = torch.ones(2, 2)
b = a.add(1) # a + 1 과 동일
print("--- Standard Operation ---")
print("Original a:₩n", a)
print("Result b:₩n", b)
print(f"Memory address of a: {a.data_ptr()}")
print(f"Memory address of b: {b.data_ptr()}₩n") # a 와 주소가 다름

# 2. In-place 연산 (자기 자신을 수정)
c = torch.ones(2, 2)
print("--- In-place Operation ---")
print("Original c:₩n", c)
print(f"Memory address of c before: {c.data_ptr()}")
c.add_(1) # c = c + 1 과 유사하지만, 메모리를 재할당하지 않음
print("Modified c:₩n", c)
print(f"Memory address of c after: {c.data_ptr()}") # 주소가 동일함
```

실행 결과:

```
--- Standard Operation ---
Original a:
 tensor([[1., 1.],
         [1., 1.]])
Result b:
 tensor([[2., 2.],
         [2., 2.]])
```

Memory address of a: ...

Memory address of b: ... (a 와 다른 주소)

--- In-place Operation ---

Original c:

tensor([[1., 1.],

[1., 1.]])

Memory address of c before: ...

Modified c:

tensor([[2., 2.],

[2., 2.]])

Memory address of c after: ... (이전 c 와 동일한 주소)

In-place 연산 사용 시 주의사항

`requires_grad=True`인 텐서에 In-place 연산을 사용하면 `RuntimeError: a view of a leaf Variable that requires grad is being used in an in-place operation.` 와 같은 오류가 발생할 수 있습니다. 기울기 계산이 필요한 파라미터나 중간 값에는 In-place 연산을 피하는 것이 좋습니다.

2.5 브로드캐스팅(Broadcasting)

브로드캐스팅은 모양이 다른 텐서 간의 산술 연산을 가능하게 하는 매우 강력하고 효율적인 메커니즘입니다. 모든 텐서를 연산을 위해 명시적으로 동일한 모양으로 확장(복사)하는 대신, PyTorch 는 내부적으로 "가상으로" 텐서를 확장하여 연산을 수행합니다. 이는 코드를 간결하게 만들고, 불필요한 메모리 사용을 줄여줍니다.

예를 들어, 이미지의 모든 픽셀에 동일한 값을 더해야 할 때, 이미지와 똑같은 크기의 텐서를 만들 필요 없이 스칼라 값 하나만 더하면 PyTorch 가 알아서 모든 픽셀에 더해줍니다. 이것이 브로드캐스팅의 가장 간단한 예입니다.

브로드캐스팅 규칙

두 텐서 `a`와 `b`가 브로드캐스팅 가능하려면, 다음 두 가지 규칙을 만족해야 합니다. (두 텐서의 차원을 **뒤에서부터 앞으로** 비교합니다.)

1. 규칙 1: 각 차원의 크기가 동일하다.
2. 규칙 2: 한쪽 텐서에서 해당 차원의 크기가 1 이거나, 또는 해당 차원이 존재하지 않는다.
이 규칙들이 모든 차원에 대해 만족되면, 두 텐서는 브로드캐스팅 가능합니다. 연산 시, 크기가 1 이거나 존재하지 않는 차원은 다른 텐서의 해당 차원 크기에 맞게 "복사"되어 가상으로 확장됩니다.

예시:

`a` (모양: `(3, 4)`) + `b` (모양: `(4,)`)

1. 뒤에서부터 비교: `a`의 마지막 차원은 4, `b`의 마지막 차원도 4. (규칙 1 만족)

2. 다음 차원 비교: `a`의 다음 차원은 3, `b`에는  . (규칙 2 만족)

=> 브로드캐스팅 가능. `b`가 `(1, 4)`로 간주된 후, `(3, 4)`로 확장되어 `a`의 각 행에 더해집니다.

예제 2-20: 브로드캐스팅 기본 예제

다양한 모양의 텐서 간 연산에서 브로드캐스팅이 어떻게 동작하는지 구체적인 예제를 통해 확인합니다.

```
import torch
# 예제 1: 행렬 + 스칼라
mat = torch.ones(3, 4)
scalar = 100
result1 = mat + scalar
print("--- Matrix (3, 4) + Scalar ---")
print(f"Shape: {mat.shape} + (scalar) -> {result1.shape}")
print(result1, "₩n")
# 설명: 스칼라(0D)가 (3, 4) 모양으로 확장되어 각 요소에 더해짐
# 예제 2: 행렬 + 벡터 (행 방향 브로드캐스팅)
mat = torch.arange(12).reshape(3, 4)
```

```
vec_row = torch.tensor([10, 20, 30, 40]) # shape: (4,)
result2 = mat + vec_row
print("--- Matrix (3, 4) + Vector (4,) ---")
print(f"Shape: {mat.shape} + {vec_row.shape} -> {result2.shape}")
print("Matrix:₩n", mat)
print("Vector:", vec_row)
print("Result:₩n", result2, "₩n")
# 설명: vec_row(4,)가 (1, 4)로 간주된 후, (3, 4) 모양으로 확장(복사)되어 mat 의 각 행에 더해짐
# 예제 3: 행렬 + 벡터 (열 방향 브로드캐스팅)
mat = torch.arange(12).reshape(3, 4)
vec_col = torch.tensor([10, 20, 30]).unsqueeze(1) # shape: (3, 1)
result3 = mat + vec_col
print("--- Matrix (3, 4) + Vector (3, 1) ---")
print(f"Shape: {mat.shape} + {vec_col.shape} -> {result3.shape}")
print("Matrix:₩n", mat)
print("Vector:₩n", vec_col)
print("Result:₩n", result3, "₩n")
# 설명: vec_col(3, 1)이 (3, 4) 모양으로 확장(복사)되어 mat 의 각 열에 더해짐
# 예제 4: 브로드캐스팅 불가 사례
vec_fail = torch.tensor([10, 20, 30]) # shape: (3,)
try:
    mat + vec_fail
except RuntimeError as e:
    print("--- Incompatible Shapes ---")
    print(f"Matrix (3, 4) + Vector (3,): Error")
    print(e)
```

실행 결과:

```
--- Matrix (3, 4) + Scalar ---
Shape: torch.Size([3, 4]) + (scalar) -> torch.Size([3, 4])
tensor([[101., 101., 101., 101.],
        [101., 101., 101., 101.],
        [101., 101., 101., 101.]])
--- Matrix (3, 4) + Vector (4,) ---
Shape: torch.Size([3, 4]) + torch.Size([4]) -> torch.Size([3, 4])
Matrix:
 tensor([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
Vector: tensor([10, 20, 30, 40])
Result:
 tensor([[10, 21, 32, 43],
        [14, 25, 36, 47],
        [18, 29, 40, 51]])
--- Matrix (3, 4) + Vector (3, 1) ---
Shape: torch.Size([3, 4]) + torch.Size([3, 1]) -> torch.Size([3, 4])
Matrix:
 tensor([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
Vector:
 tensor([[10],
        [20],
        [30]])
Result:
 tensor([[10, 11, 12, 13],
        [24, 25, 26, 27],
```

[38, 39, 40, 41]])

--- Incompatible Shapes ---

Matrix (3, 4) + Vector (3,): Error

The size of tensor a (4) must match the size of tensor b (3) at non-singleton dimension 1

결과 해석:

예제 4 에서 `(3, 4)`와 `(3,)` 모양의 텐서는 브로드캐스팅이 불가능합니다. 뒤에서부터 비교하면, `a`의 마지막 차원은 4, `b`의 마지막 차원은 3 으로 크기가 다르고 어느 쪽도 1 이 아니므로 규칙을 위반합니다. 오류 메시지는 `non-singleton dimension 1` (크기가 1 이 아닌 1 번 차원)에서 크기(4 와 3)가 일치하지 않음을 명확히 알려줍니다.

2.6 NumPy 와 호환성

PyTorch 는 데이터 과학 생태계의 핵심인 NumPy 와 매우 긴밀하게 통합되어 있습니다. 앞서 `torch.tensor()`가 NumPy 배열을 '복사'하여 텐서를 만드는 것을 보았습니다. 이번에는 데이터를 복사하지 않고 메모리를 '공유'하여 NumPy 배열과 PyTorch 텐서 간에 매우 빠르고 효율적인 변환을 수행하는 방법을 알아봅니다.

`torch.from_numpy()`와 `tensor.numpy()`: 메모리 공유

이 방식은 CPU 텐서와 NumPy 배열이 **동일한 메모리 공간을 공유**하게 만듭니다. 이는 대용량 데이터의 불필요한 복사를 피할 수 있어 매우 효율적이지만, 한쪽에서의 수정이 다른 쪽에 즉시 반영되므로 사용 시 주의가 필요합니다.

예제 2-21: 메모리 공유를 통한 상호 변환

```
import torch
import numpy as np

# 1. NumPy 배열 생성
numpy_array = np.ones((2, 3), dtype=np.float64)
print("--- Original NumPy Array ---")
```

```
print(f"NumPy Array:₩n{numpy_array}")
print(f"Memory Address of NumPy Array: {numpy_array.ctypes.data}₩n")

# 2. torch.from_numpy()로 텐서 생성 (메모리 공유)
tensor_from_numpy = torch.from_numpy(numpy_array)
print("--- Tensor from NumPy (Memory Shared) ---")
print(f"Tensor:₩n{tensor_from_numpy}")
print(f"Memory Address of Tensor: {tensor_from_numpy.data_ptr()}₩n")

# 3. 원본 NumPy 배열의 값을 변경
print("--- Modifying Original NumPy Array ---")
numpy_array[0, 0] = 99.0
print(f"Modified NumPy Array:₩n{numpy_array}")
print(f"Tensor is also changed:₩n{tensor_from_numpy}₩n")

# 4. 텐서의 값을 변경
print("--- Modifying Tensor ---")
tensor_from_numpy.mul_(2) # In-place multiplication
print(f"Modified Tensor:₩n{tensor_from_numpy}")
print(f"NumPy array is also changed:₩n{numpy_array}₩n")

# 5. 텐서를 다시 NumPy 배열로 변환 (메모리 공유)
numpy_from_tensor = tensor_from_numpy.numpy()
print("--- NumPy from Tensor (Memory Shared) ---")
print(f"NumPy Array:₩n{numpy_from_tensor}")
print(f"Memory Address of new NumPy Array: {numpy_from_tensor.ctypes.data}")
```

코드 라인별 상세 설명:

- torch.from_numpy(numpy_array): NumPy 배열을 PyTorch 텐서로 변환합니다. 이 함수는 새로운 메모리를 할당하지 않고, `numpy_array`가 사용하는 메모리를 그대로 가리키는 텐서를 생성합니다.
- numpy_array[0, 0] = 99.0: NumPy 배열의 값을 변경합니다. 메모리를 공유하므로, `tensor_from_numpy`의 해당 값도 99.0 으로 변경됩니다.
- tensor_from_numpy.mul_(2): 텐서의 모든 요소에 2 를 곱하는 **in-place** 연산입니다. 이 변경 사항은 `numpy_array`에도 즉시 반영됩니다.
- tensor_from_numpy.numpy(): PyTorch 텐서를 NumPy 배열로 변환합니다. 이 역시 메모리를 공유합니다.

실행 결과:

--- Original NumPy Array ---

NumPy Array:

[[1. 1. 1.]

 [1. 1. 1.]]

Memory Address of NumPy Array: ...1520

--- Tensor from NumPy (Memory Shared) ---

Tensor:

tensor([[1., 1., 1.],

        [1., 1., 1.]], dtype=torch.float64)

Memory Address of Tensor: ...1520

--- Modifying Original NumPy Array ---

Modified NumPy Array:

[[99. 1. 1.]

 [1. 1. 1.]]

Tensor is also changed:

tensor([[99., 1., 1.],

        [1., 1., 1.]], dtype=torch.float64)

```
--- Modifying Tensor ---
Modified Tensor:
tensor([[198.,   2.,   2.],
        [  2.,   2.,   2.]], dtype=torch.float64)
NumPy array is also changed:
[[198.   2.   2.]
 [  2.   2.   2.]]

--- NumPy from Tensor (Memory Shared) ---
NumPy Array:
[[198.   2.   2.]
 [  2.   2.   2.]]
Memory Address of new NumPy Array: ...1520
```

메모리 공유의 제약사항 및 주의점

- 메모리 공유는 **CPU 텐서**에서만 가능합니다. GPU 에 있는 텐서를 NumPy 배열로 변환하려면, 먼저 `.cpu()`를 호출하여 CPU 로 가져와야 합니다: `gpu_tensor.cpu().numpy()`.
- `torch.from_numpy()`는 NumPy 배열의 데이터 타입을 그대로 유지합니다. PyTorch 는 `np.float64`를 기본으로 사용하지 않으므로, `np.array(..., dtype=np.float32)`와 같이 타입을 맞춰주는 것이 좋습니다.
- 메모리 공유는 매우 강력하지만, 의도치 않은 값 변경을 유발할 수 있습니다. 데이터의 독립성이 중요하다면, `torch.tensor(numpy_array)` (복사) 또는 `torch.from_numpy(numpy_array).clone()` (공유 후 복제)를 사용해야 합니다.