Chapter 2

텐서(Tensor)

학습 목표

- 파이토치 텐서의 기본 개념을 이해하고 텐서를 생성 및 활용할 수 있다.
- 텐서의 기본 연산 및 조작 방법을 익히고, 텐서의 속성과 특성에 대해 이해할 수 있다.
- 텐서 데이터 타입과 장치 이동 방식을 이해하고, 이를 활용하여 효율적인 프로그래밍을 할 수 있다.
- 텐서 변환과 브로드캐스팅 개념을 이해하고, 이를 활용하여 수치 계산 및 데이터 분석을 수행할 수 있다.

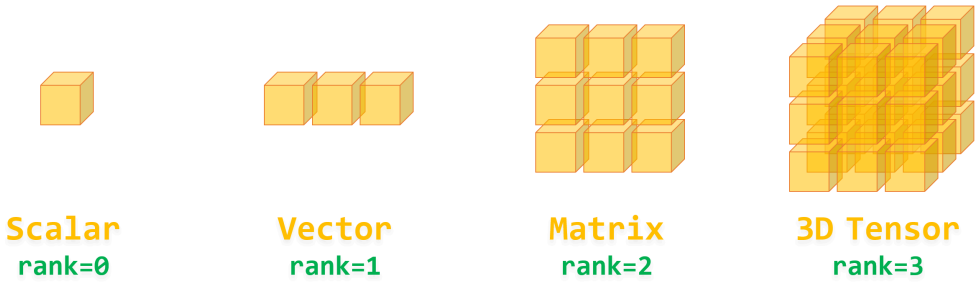
2.1 텐서(Tensor) 소개

- 최근의 머신 러닝 시스템은 일반적으로 **텐서를 기본 데이터 구조**로 사용한다.
- 데이터를 위한 컨테이너라고 생각하면 쉽게 이해할 수 있다.
- NumPy는 훌륭한 프레임워크지만, GPU를 사용하여 수치 연산을 가속화할 수는 없기 때문에 GPU연산이 가능한 PyTorch에서의 Tensor를 사용한다. (디바이스 선택이 수월함.)

  심층 신경망에서 GPU는 종종 50배 또는 그 이상의 속도 향상을 제공하기 때문에, 안타깝게도 NumPy는 딥러닝 프로그래밍에는 충분치 않다.
- **PyTorch 텐서(Tensor)는 개념적으로 NumPy 배열과 동일** : 텐서(Tensor)는 n-차원 배열이며, PyTorch는 이러한 텐서들의 연산을 위한 다양한 기능들을 제공한다.

텐서의 특성

- 축의 개수:
  랭크(Rank)라고도 부르며, 넘파이 배열에서는 `ndim`을 통해 확인할 수 있다.
- 크기:
  텐서의 각 축을 따라 얼마나 많은 차원이 있는지를 나타낸 튜플 넘파이 배열에서는 `shape`을 통해 확인할 수 있다.
- 데이터 타입:
  텐서에 포함된 데이터 타입이다.
  타입은 float32, unit8, float64 등이 될 수 있으며 문자열은 지원하지 않는다. `dtpye` 속성으로 데이터를 확인할 수 있다.



출처 : https://codetorial.net/tensorflow/basics_of_tensor.html

In [2]:
```
# 스칼라(0D Tensor)
# 하나의 숫자만을 담고 있는 텐서를 스칼라라고 하며 0차원텐서 라고 한다.
# 스칼라의 축의 개수는 0개이다.
import torch
import numpy as np

scalar = np.array(10)
print(scalar)
print(scalar.ndim)
print(scalar.shape) # 차원이 없으므로 빈 튜플을 출력
```

```
10
0
()
```

In [ ]:
```
# 벡터(1D Tensor)
# 숫자들의 배열을 벡터라고 하며 1차원 텐서라고 한다.
# 벡터의 축의 개수는 1개이다.
import torch
import numpy as np

vector = np.array([1, 2, 3, 4, 5])
print(vector)
print(vector.ndim) # 차원
print(vector.shape) # 배열의 크기
```

```
[1 2 3 4 5]
1
(5,)
```

In [3]:
```
# 행렬(2D Tensor)
# 벡터들의 배열을 행렬 이라고 하며 2차원 텐서라고 한다.
# 행렬에는 행과 열 2가지의 축이 있다.
import torch
import numpy as np

matrix = np.array([[1, 2, 3],
                   [4, 5, 6]])
print(matrix)
print(matrix.ndim)
print(matrix.shape) # 2행 3열
```

```
[[1 2 3]
 [4 5 6]]
2
(2, 3)
```

In [4]:
```
# 고차원 텐서
# 행렬들을 하나의 새로운 배열을 합치면 숫자로 채워진 직육면체가 되는데 이는 3D Tensor이다
# 딥러닝에서는 보통 5차원 텐서까지 다루게 된다.
import torch
import numpy as np

tensor_3D = np.arange(24).reshape(2, 3, 4)

print(tensor_3D) # 3D텐서는 2개의 3*4 행렬로 구성
print(tensor_3D.ndim) # 텐서의 차원수는 3
print(tensor_3D.shape) # 텐서의 형태는 2개의 3*4 크기 행렬을 포함하고 있음
```

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

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
3
(2, 3, 4)
```

Tensor의 Shape

넘파이의 배열에서 `shape`를 이용하여 텐서의 크기를 알아볼 경우 return값은 다음과 같다

1차원 : (열,)
2차원 : (행, 열)
3차원 : (깊이, 행, 열)

즉, 열의 기준이 되어 차원이 늘어날수록 늘어난 차원의 크기가 열의 앞에 추가된다고 생각하면 된다.
머신러닝에서 사용하는 train_data의 shape은 다음과 같다.

1. 벡터 데이터: (samples, features) 2D tensor 집값 예측 문제라고 생각하고 주어진 데이터가 100개의 연식, 동네, 역세권의 유무에 따른 데이터라고 하면 (100, 3)크기의 텐서에 저장될 수 있다.

2. 이미지: (samples, height, width, channels) 4D tensor 채널 우선방식 과 채널 마지막 방식으로 나뉘지만 보통의 경우 100 장의 28x28의 컬러 이미지라면 (100, 28, 28, 3)크기의 텐서에 저장될 수 있다.

3. 동영상: (samples, frames, height, channels) 5D tenesor

In [5]:
```
import numpy as np
from keras.datasets import mnist

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
print(train_images.shape)

# 28x 28배열의 사진이 6만장이 들어있는 것을 뜻한다.
```

```
(60000, 28, 28)
```

In [6]:
```
use_images = train_images[:100, :, :]
print(use_images.shape)
```

```
# 일반적으로 딥러닝에서 사용하는 모든 데이터 텐서의 첫번째 축(axis0)은 샘플 축(sample ax
```

```
(100, 28, 28)
```

위와 같은 샘플 축의 슬라이싱은 모델 수행 시 배치 데이터를 나눌 때 사용된다.

배치 데이터를 다룰 때는 첫번째 축(axis0)를 배치축(batch axis) 또는 배치차원(batch dimension)이라고 부른다.

In [7]:
```
# 6만장의 데이터 중에서 위쪽 절반만 사용하고 싶을 경우
use_images = train_images[: , :14 , :]
print(use_images.shape)
```

```
(60000, 14, 28)
```

※ Tensor과 Ndarray

- 텐서 (Tensor)는 NumPy 어레이와 비슷하지만, 텐서는 GPU, TPU와 같은 가속기에서 사용할 수 있고, 텐서는 값을 변경할 수 없습니다.

먼저 Pytorch에서 주로 다루는 데이터 자료형인 Tensor과 Numpy에서 주로 다루는 데이터 자료형인 ndarray는 다차원 배열과 배열 생성 시 다양한 데이터 타입 적용이 가능하지만 주요한 차이점이 존재합니다.

우선 가장 중요한 차이점은 GPU를 활용한 연산이 가능한가? 이며, 그 외로 중요한 기능들에서 차이점이 있습니다.

특징	tensor (PyTorch)	ndarray (NumPy)
주요 라이브러리	PyTorch	NumPy
GPU 지원	가능	불가능 (CuPy 필요)
자동 미분	지원 (requires_grad)	미지원
주요 사용 목적	딥러닝, 신경망 훈련 및 추론	수치 계산, 데이터 분석
변환 가능 여부	.numpy()로 변환 가능	torch.from_numpy()로 변환 가능

2.2 텐서 생성하기

- 텐서는 파이토치에서 스칼라, 벡터, 행렬 같은 개념으로 연산에 사용하는 기본적인 자료구조이며 다차원 배열로 표현할 수 있습니다.
- 파이썬 numpy의 ndarray와 유사합니다.(n차원의 배열 객체)
- 텐서는 n차원으로 구성되어 있는데 0차원은 스칼라, 1차원은 벡터, 2차원은 행렬, 3차원부터 텐서입니다.
- 1차원 텐서, 2차원 텐서, 3차원 텐서라고도 합니다.
- torch 패키지를 임포트하고 tensor 함수를 이용하여 텐서를 생성할 수 있습니다.

In [8]:
```
import torch # 파이토치 라이브러리 호출
# 파이토치는 머신러닝과 딥러닝을 위한 강력한 도구로, 텐서라는 특별한 데이터 구조를 사용합

# 1차원 텐서 생성하기

a = torch.tensor([1, 2, 3, 4, 5]) #torch.tensor(대괄호 안에 원소 입력)
# 1차원 텐서 생성, 텐서는 숫자들의 배열이라고 생각하면 됩니다.
# 이 경우, [1, 2, 3, 4, 5]라는 다섯 개의 숫자로 이루어진 리스트를 텐서로 변환

print(a)

print("텐서 크기:", a.size()) #텐서의 크기 확인 size()
# 텐서 a의 크기를 출력, size() 함수는 텐서의 차원과 각 차원의 크기를 알려줍니다.
```

```
tensor([1, 2, 3, 4, 5])
텐서 크기: torch.Size([5])
```

Tip!

이 예제는 파이토치에서 가장 기본적인 데이터 구조인 텐서를 만들고 그 정보를 확인하는 방법을 보여줍니다.

텐서는 숫자 데이터를 다루는 데 매우 유용하며, 머신러닝과 딥러닝에서 중요한 역할을 합니다.

이러한 기초적인 개념을 이해하는 것이 더 복잡한 머신러닝 모델을 다루는 데 도움이 될 것입니다.

In [1]:
```
import torch

# 2차원 텐서 생성하기
# 1차원 텐서들을 바깥 대괄호로 감싸기

x1 = torch.tensor([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])
#  2차원 텐서는 행렬과 같은 구조를 가집니다.
# 이 경우: 첫 번째 행: [1, 2, 3]
#          두 번째 행: [4, 5, 6]
#          세 번째 행: [7, 8, 9]
# 이렇게 3개의 행과 3개의 열로 이루어진 3x3 행렬 형태의 텐서를 만듭니다.

print(x1)

print("size:", x1.size())  # 텐서 크기 [행, 열]
```

```
print("rank(차원):", x1.dim())
# 텐서의 차원 수(rank)를 출력합니다. dim() 함수는 텐서의 차원 수를 반환합니다. 이 경우 2
```

```
tensor([[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]])
size: torch.Size([3, 3])
rank(차원): 2
```

In [10]:
```
import torch

# 3행 2열 리스트로 2차원 텐서 생성하기

data = [[1, 2],[3, 4],[5, 6]]
# 차원 리스트를 만듭니다. 이 리스트는 3개의 내부 리스트로 구성되어 있으며,
# 각 내부 리스트는 2개의 숫자를 포함하고 있습니다.

# 텐서로 변환
x2 = torch.tensor(data)
#  data 리스트를 파이토치 텐서로 변환합니다.
# torch.tensor() 함수는 파이썬 리스트를 받아 텐서로 변환해줍니다.

print(x2)

print(x2.shape)
# 텐서 x2의 형태(shape)를 출력합니다. shape 속성은 텐서의 차원과 각 차원의 크기를 알려줍
# 이는 이 텐서가 3행 2열의 구조를 가지고 있다는 의미입니다.
```

```
tensor([[1, 2],
        [3, 4],
        [5, 6]])
torch.Size([3, 2])
```

In [11]:
```
import torch

# 3차원 텐서 생성하기
# 2차원 텐서를 대괄호로 감싸기

x3 = torch.tensor([[[1, 2, 3],[4, 5, 6],[7, 8, 9]]])
# 여기서는 3차원 텐서를 생성합니다.
# 3차원 텐서는 '큐브' 또는 '박스'와 같은 구조를 가집니다.
# 이 경우: 가장 바깥쪽 대괄호는 첫 번째 차원을 나타냅니다 (깊이).
#          중간 대괄호는 두 번째 차원을 나타냅니다 (행).
#          가장 안쪽 대괄호는 세 번째 차원을 나타냅니다 (열).
# 이 텐서는 1개의 2차원 행렬(3x3)을 포함하고 있습니다.

print(x3)

print("텐서 크기:", x3.size())
# 텐서 x3의 크기를 출력합니다. size() 함수는 텐서의 각 차원의 크기를 반환합니다. 결과는
# 이는 다음을 의미합니다: 첫 번째 차원(깊이)의 크기: 1
#                        두 번째 차원(행)의 크기: 3
#                        세 번째 차원(열)의 크기: 3
```

```
tensor([[[1, 2, 3],
         [4, 5, 6],
         [7, 8, 9]]])
텐서 크기: torch.Size([1, 3, 3])
```

In [12]:
```
# NumPy 배열과 PyTorch 텐서 간의 변환을 보여주는 좋은 예시
```

```python
import torch

# Numpy 배열로 텐서 생성하기
import numpy as np
# 2차원 리스트 x3_1을 NumPy 배열로 변환합니다.
# data_np는 3x3 크기의 2차원 NumPy 배열이 됩니다.

x3_1 = [[1, 2, 3],[4, 5, 6],[7, 8, 9]]

data_np = np.array(x3_1)
# torch.from_numpy() 함수를 사용하여 NumPy 배열을 PyTorch 텐서로 변환합니다.
# 이 과정에서 데이터 타입(dtype) 정보도 함께 유지됩니다.

print(data_np)

x3_np = torch.from_numpy(data_np)  # dtype 정보도 포함

print(x3_np)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
tensor([[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]], dtype=torch.int32)
```

2.2.1 특정 값으로 텐서 생성하기

In [13]:
```python
import torch

# 0으로 채워진 텐서
tensor_zeros = torch.zeros(2, 3)
# 2행 3열의 2차원 텐서를 생성하고 모든 원소를 0으로 채웁니다.
print(tensor_zeros)

# 1으로 채워진 텐서
tensor_ones = torch.ones(2, 3)
# 2행 3열의 2차원 텐서를 생성하고 모든 원소를 1로 채웁니다.
print(tensor_ones)

# 특정값으로 채워진 텐서
tensor_full = torch.full((2, 3), 7)
# 2행 3열의 2차원 텐서를 생성하고 모든 원소를 7로 채웁니다.
print(tensor_full)
```

```
tensor([[0., 0., 0.],
        [0., 0., 0.]])
tensor([[1., 1., 1.],
        [1., 1., 1.]])
tensor([[7, 7, 7],
        [7, 7, 7]])
```

`Tip!`

- 이러한 함수들은 특정 크기와 초기값을 가진 텐서를 빠르게 생성할 때 유용합니다.
- 예를 들어, 신경망의 가중치를 초기화하거나 마스크를 만들 때 자주 사용됩니다. 각 함수의 특징을 정리하면 다음과 같습니다:
  - torch.zeros(): 모든 원소가 0인 텐서 생성

    - torch.ones(): 모든 원소가 1인 텐서 생성
    - torch.full(): 모든 원소가 지정한 값으로 채워진 텐서 생성
- 이러한 함수들은 텐서의 크기를 인자로 받으며, torch.full()의 경우 추가로 채울 값을 인자로 받습니다.

2.2.2 random한 텐서 생성

In [2]:
```
# PyTorch를 사용하여 정규분포(표준 정규분포)를 따르는 랜덤 텐서를 생성하는 예제

import torch

# 정규분포 따른 랜덤 텐서 생성
random_tensor = torch.randn((2, 3))
#  torch.randn() 함수를 사용하여 2행 3열의 랜덤 텐서를 생성합니다.
#  이 함수는 평균이 0이고 표준편차가 1인 표준 정규분포에서 무작위로 숫자를 추출합니다.
# (2, 3)은 생성될 텐서의 형태를 지정합니다. 2행 3열의 행렬이 만들어집니다.

print("\n랜덤 값으로 채워진 텐서:")
print(random_tensor)
# 이 결과에서 각 숫자는 표준 정규분포에서 무작위로 추출된 값입니다. 대부분의 값들은 -2와
```

```
랜덤 값으로 채워진 텐서:
tensor([[ 0.6606,  0.4591,  0.0148],
        [-2.0016, -0.4723, -0.0329]])
```

In [3]:
```
import torch

# 위에 생성된 random_tensor와 같은 크기인 2행 3열 랜덤 텐서 생성
tensor_like = torch.randn_like(random_tensor)

print(tensor_like)
# torch.randn_like() 함수는 주어진 텐서와 동일한 크기와 데이터 타입을 가진 새로운 텐서를

# 각 원소는 -1과 1 사이의 랜덤한 값을 가집니다. 이 값들은 표준 정규 분포에서 추출되었으
```

```
tensor([[-1.8724, -0.3398, -0.0133],
        [-0.2630, -0.4519, -0.4844]])
```

Tip!

주요 특징

- 크기 유지: tensor_like는 random_tensor와 정확히 같은 크기(2x3)를 가집니다.
- 독립적인 랜덤 값: 생성된 값들은 random_tensor의 값들과 완전히 독립적입니다.
- 데이터 타입 일치: tensor_like는 random_tensor와 동일한 데이터 타입을 가집니다.
- GPU 호환성: 만약 random_tensor가 GPU에 있다면, tensor_like도 자동으로 GPU에 생성됩니다.

※ 이러한 기능은 딥러닝 모델에서 가중치 초기화나 노이즈 생성 등 다양한 상황에서 유용하게 사용될 수 있습니다.

In [16]:
```
import torch

# rand(행, 열)은 0~1 사이의 실수값으로 채워진 텐서
tensor_random = torch.rand(2, 3)
```

```
# torch.rand() 함수를 사용하여 2행 3열의 텐서를 생성합니다.
# 각 원소는 0과 1 사이의 균일 분포에서 무작위로 추출된 실수값입니다.
# 출력 결과는 매번 다르지만 2행 3열 생성때 마다 유사한 형태일 것입니다:
print(tensor_random)

# 시작값부터 1씩증가하여 이전값까지 범위내 채워진 텐서
tensor_arange = torch.arange(1, 7).reshape(2, 3)
# 이 부분은 두 단계로 이루어집니다:
# torch.arange(1, 7)은 1부터 6까지의 정수를 포함하는 1차원 텐서를 생성합니다.
# .reshape(2, 3)은 이 1차원 텐서를 2행 3열의 2차원 텐서로 재구성합니다.
# 출력 결과는 항상 동일합니다.

print(tensor_arange)
```

```
tensor([[0.6795, 0.9429, 0.8354],
        [0.5226, 0.3432, 0.7782]])
tensor([[1, 2, 3],
        [4, 5, 6]])
```

`Tip!`

- rand()는 무작위 값으로 텐서를 채우는 데 유용하며, arange()와 reshape()의 조합은 특정 패턴이나 순서를 가진 텐서를 만드는 데 사용됩니다.

2.3 텐서 속성

- 텐서의 차원을 확인하는 dim() 메서드가 있습니다. 이는 텐서의 rank라고도 불립니다.
- 텐서의 형상(shape)을 확인하는 shape 속성이 있습니다. 이는 각 차원의 크기를 튜플 형태로 반환합니다.
- 텐서의 크기를 확인하는 size() 메서드도 있습니다. 이는 shape와 동일한 정보를 반환합니다.
- 데이터타입 자료형을 확인하는 dtype 속성은 datatype() 속성이며, 이를 통해 텐서의 데이터 타입(예: 정수, 실수, 불리언 등)을 알 수 있습니다.
- 텐서가 저장된 장치(CPU 또는 GPU)를 확인하는 device 속성이 있습니다.
- 텐서의 총 원소 개수를 반환하는 numel() 메서드도 유용합니다.

In [17]:
```
# 이 코드는 3x4x5 크기의 랜덤 텐서를 생성하고 그 속성들을 출력합니다.
# 이 코드는 텐서의 기본적인 속성들을 모두 보여주므로, PyTorch 텐서의 구조와 특성을 이해

import torch

x = torch.randn(3, 4, 5)
#  3x4x5 크기의 3차원 텐서를 생성합니다. torch.randn() 함수는 표준 정규 분포(평균 0, 표

print(f"Dimensions: {x.dim()}")
# dim() 메서드는 텐서의 차원 수를 반환합니다. 이 경우 3을 출력할 것입니다.

print(f"Shape: {x.shape}")
# shape 속성은 텐서의 각 차원의 크기를 튜플 형태로 반환합니다. 출력은 torch.Size([3, 4,

print(f"Size: {x.size()}")
# shape 속성은 텐서의 각 차원의 크기를 튜플 형태로 반환합니다. 출력은 torch.Size([3, 4,

print(f"Data type: {x.dtype}")
```

```
# dtype 속성은 텐서의 데이터 타입을 반환합니다. torch.randn()으로 생성된 텐서의 기본 데
print(f"Device: {x.device}")
# device 속성은 텐서가 저장된 장치(CPU 또는 GPU)를 나타냅니다. 기본적으로 'cpu'가 출력돌
print(f"Number of elements: {x.numel()}")
# numel() 메서드는 텐서의 총 원소 개수를 반환합니다. 이 경우 3 * 4 * 5 = 60을 출력할 것
```

```
Dimensions: 3
Shape: torch.Size([3, 4, 5])
Size: torch.Size([3, 4, 5])
Data type: torch.float32
Device: cpu
Number of elements: 60
```

In [18]:
```
# PyTorch를 사용하여 1차원 텐서를 생성하고 그 속성들을 출력하는 예제
import torch

# 1차원 텐서
# 텐서의 크기 확인 size(), shape
# 차원 확인 dim(), ndimension()
# 데이터타입 dtype()
a = torch.tensor([1, 2, 3, 4, 5])
#  1차원 텐서 a를 생성합니다. 이 텐서는 5개의 정수 원소를 포함합니다.

print("size:", a.size())
#  1차원 텐서 a를 생성합니다. 이 텐서는 5개의 정수 원소를 포함합니다.

print("shape:", a.shape) # 5개 원소
# shape 속성도 텐서의 크기를 반환합니다. size()와 동일한 정보를 제공합니다.

print("dimension:", a.dim())
# shape 속성도 텐서의 크기를 반환합니다. size()와 동일한 정보를 제공합니다.

print("rank(차원):", a.ndimension())
# shape 속성도 텐서의 크기를 반환합니다. size()와 동일한 정보를 제공합니다.

print(a.dtype)
# dtype 속성은 텐서의 데이터 타입을 반환합니다. 이 경우 torch.int64를 출력할 것입니다. (
```

```
size: torch.Size([5])
shape: torch.Size([5])
dimension: 1
rank(차원): 1
torch.int64
```

In [19]:
```
#  PyTorch를 사용하여 2차원 실수형 텐서를 생성하고 그 속성들을 출력하는 예제
import torch

# 실수형 FloatTensor
x4 = torch.FloatTensor([[1, 2],[3, 4]])
# 2x2 크기의 실수형 텐서 x4를 생성
# FloatTensor는 32비트 부동소수점 데이터 타입을 사용

print("size:", x4.size())
# size() 메서드는 텐서의 크기를 반환
# 출력은 torch.Size([2, 2])

print("shape:", x4.shape)
# size() 메서드는 텐서의 크기를 반환
# 출력은 torch.Size([2, 2])
```

```
print("dimension:", x4.dim())
# size() 메서드는 텐서의 크기를 반환

print("rank(차원):", x4.ndimension())
# size() 메서드는 텐서의 크기를 반환

print(x4.dtype)
# dtype 속성은 텐서의 데이터 타입을 반환
# torch.float32를 출력
```

```
size: torch.Size([2, 2])
shape: torch.Size([2, 2])
dimension: 2
rank(차원): 2
torch.float32
```

In [20]:
```
# PyTorch를 사용하여 3차원 텐서를 생성하고 그 속성들을 출력하는 예제
import torch

# 3차원 텐서
x5 = torch.tensor([[[1, 2, 3],[4, 5, 6],[7, 8, 9]]])
# 3차원 텐서 x5를 생성합니다. 이 텐서는 1x3x3 크기를 가집니다.

print("shape:", x5.shape)
# shape 속성은 텐서의 각 차원의 크기를 튜플 형태로 반환
# 출력은 torch.Size([1, 3, 3]), 이는 1개의 3x3 행렬을 포함하는 3차원 텐서를 의미

print("dimension:", x5.dim())
# dim() 메서드는 텐서의 차원 수를 반환
# 이 경우 3을 출력

print(f"rank(차원): {x5.ndimension()}")
# ndimension() 메서드는 dim()과 동일한 기능을 합니다.
# 역시 3을 출력

print(x5.dtype)
# dtype 속성은 텐서의 데이터 타입을 반환
# torch.int64를 출력, 이는 64비트 정수형
```

```
shape: torch.Size([1, 3, 3])
dimension: 3
rank(차원): 3
torch.int64
```

텐서생성 실습

- 이 예제는 행렬 곱셈의 기본 원리를 보여줍니다.
- 첫 번째 행렬의 열 수(3)와 두 번째 행렬의 행 수(3)가 일치해야 행렬 곱셈이 가능합니다.
- 결과 텐서의 크기는 첫 번째 행렬의 행 수(2)와 두 번째 행렬의 열 수(2)로 결정됩니다.

In [21]:
```
import torch

# [2, 3]과 [3, 2] 텐서 생성하기
a = torch.tensor([[1, 2, 3], [4, 5, 6]])
b = torch.tensor([[7, 8], [9, 10], [11, 12]])
```

```
c = torch.matmul(a, b)
# c = torch.matmul(a, b)는 a와 b의 행렬 곱셈을 수행합니다.
# 이 함수는 내부적으로 최적화된 알고리즘을 사용하여 빠른 계산을 수행합니다.

print(c)
print(c.shape)
```

```
tensor([[ 58,  64],
        [139, 154]])
torch.Size([2, 2])
```

- 이 코드는 두 텐서 a와 b의 행렬 곱셈을 수행합니다. 행렬 곱셈 과정을 자세히 설명하겠습니다:
- 텐서 a는 2x3 크기이고, 텐서 b는 3x2 크기입니다.
- 행렬 곱셈 규칙에 따라, 첫 번째 행렬의 열 수(3)와 두 번째 행렬의 행 수(3)가 일치해야 합니다.
- 결과 텐서 c의 크기는 첫 번째 행렬의 행 수(2)와 두 번째 행렬의 열 수(2)가 됩니다. 따라서 c는 2x2 크기가 됩니다.
- 행렬 곱셈 과정:
  - c[0,0] = a[0,0]b[0,0] + a[0,1]b[1,0] + a[0,2]b[2,0] = 17 + 29 + 311 = 7 + 18 + 33 = 58
  - c[0,1] = a[0,0]b[0,1] + a[0,1]b[1,1] + a[0,2]b[2,1] = 18 + 210 + 312 = 8 + 20 + 36 = 64
  - c[1,0] = a[1,0]b[0,0] + a[1,1]b[1,0] + a[1,2]b[2,0] = 47 + 59 + 611 = 28 + 45 + 66 = 139
  - c[1,1] = a[1,0]b[0,1] + a[1,1]b[1,1] + a[1,2]b[2,1] = 48 + 510 + 612 = 32 + 50 + 72 = 154

In [22]:
```
import torch

# 다양한 방법으로 텐서 생성
tensor1 = torch.tensor([1, 2, 3])
# 주어진 데이터로 1차원 텐서를 생성,결과는 [1, 2, 3] 형태의 1차원 텐서
tensor2 = torch.zeros(3, 3)
# 3x3 크기의 모든 원소가 0인 2차원 텐서 생성
tensor3 = torch.ones(2, 2)
# 2x2 크기의 모든 원소가 1인 2차원 텐서 생성
tensor4 = torch.rand(3, 3)
# 3x3 크기의 0과 1 사이의 균일 분포에서 무작위로 값을 추출하여 2차원 텐서를 생성

# 텐서 조작
reshaped = tensor4.reshape(1, 9)
# 3x3 크기의 0과 1 사이의 균일 분포에서 무작위로 값을 추출하여 2차원 텐서를 생성
transposed = tensor4.t()
# tensor4의 전치(transpose)를 수행합니다. 행과 열을 바꾸어 새로운 텐서를 생성

print("Original:", tensor4)
print("Reshaped:", reshaped)
print("Transposed:", transposed)
```

```
Original: tensor([[0.0753, 0.2070, 0.4107],
        [0.4217, 0.2071, 0.5161],
        [0.5075, 0.8243, 0.9350]])
Reshaped: tensor([[0.0753, 0.2070, 0.4107, 0.4217, 0.2071, 0.5161, 0.5075, 0.8243, 0.93
50]])
Transposed: tensor([[0.0753, 0.4217, 0.5075],
        [0.2070, 0.2071, 0.8243],
        [0.4107, 0.5161, 0.9350]])
```

- 출력 결과
  - 출력 결과는 다음과 같은 형태일 것입니다:
  - "Original:" - 원본 tensor4를 출력합니다. 3x3 크기의 랜덤 값을 가진 텐서입니다.
  - "Reshaped:" - tensor4를 1x9 형태로 재구성한 텐서를 출력합니다.
  - "Transposed:" - tensor4의 전치 결과를 출력합니다. 원본 텐서의 행과 열이 바뀐 형태입니다.
- 이 예제는 PyTorch에서 텐서를 생성하고 조작하는 기본적인 방법들을 보여줍니다.
- 이러한 연산들은 딥러닝 모델을 구축하고 데이터를 처리할 때 자주 사용됩니다1

.

데이터 타입을 지정 텐서 생성

- 정수형 텐서나 실수형 텐서를 생성할 수 있습니다.

In [23]:
```python
# PyTorch를 사용하여 실수형과 정수형 텐서를 생성하는 예제
# 이 코드는 PyTorch에서 다양한 데이터 타입의 텐서를 생성하는 방법을 보여줍니다. 데이터 타

import torch

# 실수형 텐서 생성 (float32)
float_tensor = torch.tensor([1.5, 2.5, 3.5], dtype=torch.float32)
print("\n실수형 텐서:")
print(float_tensor)
# 이 부분은 32비트 부동소수점(float32) 데이터 타입의 1차원 텐서를 생성합니다.
# [1.5, 2.5, 3.5]: 텐서의 초기 값들입니다.
# dtype=torch.float32: 텐서의 데이터 타입을 32비트 부동소수점으로 지정합니다.


# 정수형 텐서 생성 (int64)
int_tensor = torch.tensor([1, 2, 3], dtype=torch.int64)
print("\n정수형 텐서:")
print(int_tensor)
# 이 부분은 32비트 부동소수점(float32) 데이터 타입의 1차원 텐서를 생성합니다.
# [1.5, 2.5, 3.5]: 텐서의 초기 값들입니다.
# dtype=torch.float32: 텐서의 데이터 타입을 32비트 부동소수점으로 지정합니다.
```

```
실수형 텐서:
tensor([1.5000, 2.5000, 3.5000])

정수형 텐서:
tensor([1, 2, 3])
```

2.4 텐서 연산

- 파이토치는 앞에서 언급했듯이 머신러닝, 딥러닝 오픈소스 라이브러리입니다.
- 딥러닝을 이해하기 위해서 텐서의 기본적인 연산인 텐서 덧셈, 곱셈, 행렬곱셈 방법을 알아보겠습니다.
- 행렬곱은 왼쪽 행렬의 열의 수와 오른쪽 행렬의 행의 수가 일치해야 한다.

기본 연산

- **덧셈과 뺄셈**: 같은 크기의 텐서 간에 요소별로 수행됩니다.
- **요소별 곱셈**: 두 텐서의 대응하는 요소끼리 곱합니다.
- **스칼라 곱**: 텐서의 모든 요소에 스칼라 값을 곱합니다.

행렬 곱셈

- 행렬 곱셈은 `torch.matmul()` 또는 `@` 연산자를 사용합니다.
- 왼쪽 행렬의 열 수와 오른쪽 행렬의 행 수가 일치해야 합니다.
- 결과 행렬의 크기는 (왼쪽 행렬의 행 수) x (오른쪽 행렬의 열 수)가 됩니다.

고급 연산

- **전치(Transpose)**: 행과 열을 바꾸는 연산으로, `t()` 메서드를 사용합니다.
- **차원 축소**: `sum()`, `mean()` 등의 함수로 특정 차원을 따라 값을 집계합니다.
- **브로드캐스팅**: 크기가 다른 텐서 간의 연산을 자동으로 조정합니다.

비선형 활성화 함수

- ReLU, Sigmoid, Tanh 등의 함수를 텐서에 적용하여 비선형성을 도입합니다.

이러한 연산들은 신경망의 순전파와 역전파 과정에서 핵심적인 역할을 합니다.

효율적인 텐서 연산은 딥러닝 모델의 성능과 학습 속도에 직접적인 영향을 미칩니다.

In [24]:
```python
import torch
import numpy as np

# 1차 텐서 덧셈
a = torch.tensor([1, 2, 3])
b = torch.tensor([4, 5, 6])

c = a + b
print(c)
print(c.shape)
```

```
tensor([5, 7, 9])
torch.Size([3])
```

- 위에서 텐서 a와 b의 덧셈 풀이는 다음과 같습니다.
- [1+4, 2+5, 3+6] = [5, 7, 9]

In [25]:
```python
import torch
import numpy as np

# 1차 텐서 뺄셈
a = torch.tensor([1, 2, 3])
```

```
b = torch.tensor([4, 5, 6])

c = a - b
print(c)
print(c.shape)
```

```
tensor([-3, -3, -3])
torch.Size([3])
```

- 위에서 텐서 a와 b의 뺄셈 풀이는 다음과 같습니다.
- [1-4, 2-5, 3-6] = [-3, -3, -3]

In [26]:
```
import torch

a = torch.tensor([1, 2, 3])
b = torch.tensor([4, 5, 6])

# 기본 연산
addition = a + b
multiplication = a * b
dot_product = torch.dot(a, b)

print("Addition:", addition)
print("Multiplication:", multiplication)
print("Dot Product:", dot_product)
```

```
Addition: tensor([5, 7, 9])
Multiplication: tensor([ 4, 10, 18])
Dot Product: tensor(32)
```

In [27]:
```
import torch

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
tensor = torch.rand(3, 3).to(device)
print(f"Tensor is on {device}")
```

```
Tensor is on cuda
```

이 코드는 PyTorch를 사용하여 텐서를 생성하고 GPU가 사용 가능한 경우 GPU로 텐서를 이동시키는 예제입니다. 각 줄을 자세히 설명하겠습니다:

1. `device = torch.device("cuda" if torch.cuda.is_available() else "cpu")`
   이 줄은 GPU 사용 가능 여부를 확인하고 적절한 장치를 선택합니다:
   - `torch.cuda.is_available()`은 CUDA(GPU 연산을 위한 NVIDIA의 병렬 컴퓨팅 플랫폼)가 사용 가능한지 확인합니다.
   - GPU가 사용 가능하면 `device`는 "cuda"로 설정되고, 그렇지 않으면 "cpu"로 설정됩니다.
2. `tensor = torch.rand(3, 3).to(device)`
   이 줄은 3x3 크기의 랜덤 텐서를 생성하고 선택된 장치로 이동시킵니다:
   - `torch.rand(3, 3)`는 0과 1 사이의 균일 분포에서 무작위로 값을 추출하여 3x3 텐서를 생성합니다.
   - `.to(device)`는 생성된 텐서를 선택된 장치(GPU 또는 CPU)로 이동시킵니다.
3. `print(f"Tensor is on {device}")`
   이 줄은 텐서가 어떤 장치에 있는지 출력합니다.

이 코드는 GPU가 사용 가능한 경우 자동으로 GPU를 활용하여 연산을 수행할 수 있게 해줍니다. GPU를 사용하면 대규모 텐서 연산과 딥러닝 모델 학습 속도를 크게 향상시킬 수 있습니다. GPU가 없는 시스템에서는 자동으로 CPU를 사용하므로, 코드의 호환성도 유지됩니다.

텐서 곱셈

a = torch.tensor([[1, 2], [3, 4]]) b = torch.tensor([[5, 6], [7, 8]])

c = torch.mul(a, b) # 요소별 곱셈 c1 = torch.matmul(a, b) # 행렬곱 print(c) print(c1)

- 행렬곱 matmul 주의할 점은 행렬 a의 열의 수와 b의 행의 수가 같아야 합니다.

$$\begin{bmatrix}1&2\\3&4\end{bmatrix}\begin{bmatrix}5&6\\7&8\end{bmatrix}=\begin{bmatrix}1*5+2*7&1*6+2*8\\3*5+4*7&3*6+4*8\end{bmatrix}=\begin{bmatrix}19&22\\43&50\end{bmatrix}$$

그림 2.1 텐서 행렬곱

In [28]:
```
import torch
import numpy as np

a = torch.tensor([10, 20, 30])
b = torch.tensor([2, 4, 5])

result = a / b
# 또는 torch.div(a, b) 사용 가능

# a를 b로 요소별로 나눕니다. 이는 다음과 같이 계산됩니다
# 10 / 2 = 5
# 20 / 4 = 5
# 30 / 5 = 6
# 참고: torch.div(a, b)를 사용해도 동일한 결과를 얻을 수 있습니다.

print(result)  # tensor([5., 5., 6.]) 결과가 부동소수점 형식
```

```
tensor([5., 5., 6.])
```

In [29]:
```
import torch
import numpy as np

a = torch.tensor([2, 3, 4])

# 텐서 요소 각각의 거듭제곱
result = a ** 2
# 또는 torch.pow(a, 2)
# 2^2 = 4, 3^2 = 9, 4^2 = 16

print(result)  # tensor([ 4,  9, 16])
```

```
tensor([ 4,  9, 16])
```

In [30]:
```
import torch
import numpy as np

tensor = torch.tensor([[1, 2, 3], [4, 5, 6]])
```

```
# 전체 합계
sum_value = torch.sum(tensor)
print(sum_value)  # tensor(21) 1 + 2 + 3 + 4 + 5 + 6 = 21

# 특정 차원의 합계
sum_value_dim0 = torch.sum(tensor, dim=0)  # 행을 기준으로 열의 합
#  dim=0 매개변수를 사용하여 행을 기준으로 열의 합을 계산합니다.
# 결과는 [5, 7, 9]가 됩니다. 계산 과정은 다음과 같습니다
# 첫 번째 열: 1 + 4 = 5
# 두 번째 열: 2 + 5 = 7
# 세 번째 열: 3 + 6 = 9
# 이러한 합계 연산은 딥러닝에서 자주 사용됩니다.
# 예를 들어, 손실 함수의 계산이나 특성 집계 등에 활용될 수 있습니다.

print(sum_value_dim0)  # tensor([5, 7, 9])
```

```
tensor(21)
tensor([5, 7, 9])
```

In [31]:
```
import torch
import numpy as np

tensor = torch.tensor([1, 2, 3, 4, 5])

# 최댓값
max_value = torch.max(tensor)
print(max_value)  # tensor(5)

# 최솟값
min_value = torch.min(tensor)
print(min_value)  # tensor(1)
```

```
tensor(5)
tensor(1)
```

In [32]:
```
# 비교 연산은 데이터 필터링, 조건부 연산, 마스킹 등 다양한 상황에서 유용하게 사용됩니다.
# 예를 들어, 이 결과를 사용하여 원본 텐서에서 3보다 큰 값만 선택하거나, 특정 조건을 만족

import torch
import numpy as np

tensor = torch.tensor([1, 2, 3, 4, 5])

# 3보다 큰 값들만 True
result = tensor > 3
# 첫 세 요소(1, 2, 3)는 3보다 크지 않으므로 False입니다.
# 마지막 두 요소(4, 5)는 3보다 크므로 True입니다.

print(result)  # tensor([False, False, False,  True,  True])
```

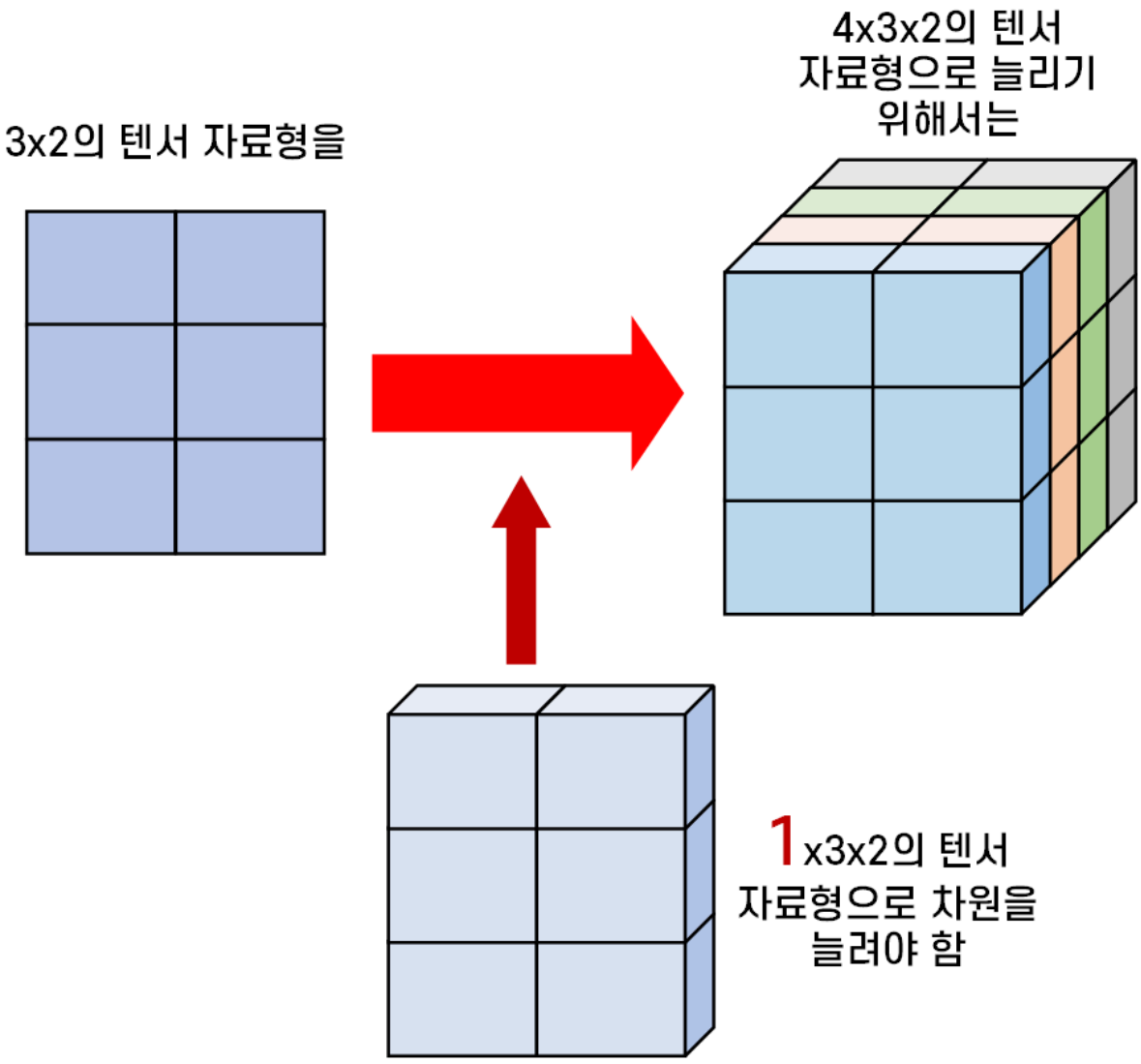
```
tensor([False, False, False,  True,  True])
```

2.5 텐서 연산을 위한 텐서 변환

- 텐서 연산은 필요에 따라 차원(dimensions)을 변경할 수 있습니다.
- view(), unsqueeze(), squeeze() 함수를 이용하여 텐서 변환을 할 수 있습니다.
- 중요한건 차원크기가 무조건 1로 늘어나거나 1인 것만 줄일 수 있습니다.

In [33]:
```python
import torch

exam = torch.rand(3,1,4,1)
print(f"exam텐서의 차원 : {exam.shape}")

#3, 1, 4, 1차원의 idx는 0, 1, 2, 3
exam = exam.squeeze(dim=1)
print(f"exam텐서의 차원 : {exam.shape}")

# idx는 리스트 인덱싱처럼 뒤에서도 순번을 지정
exam = exam.squeeze(dim=-1)
print(f"exam텐서의 차원 : {exam.shape}")

# 차원을 늘리는것은 인자값에 해당하는 순번에 `1`로 빈 차원이 삽입
exam = exam.unsqueeze(dim=0)
print(f"exam텐서의 차원 : {exam.shape}")
```

```
exam텐서의 차원 : torch.Size([3, 1, 4, 1])
exam텐서의 차원 : torch.Size([3, 4, 1])
exam텐서의 차원 : torch.Size([3, 4])
exam텐서의 차원 : torch.Size([1, 3, 4])
```



`Tensor` 자료형의 '확장'은 무조건 차원 확장 후 → 차원값 확장 으로 진행되기 때문이다

> `torch.repeat()`, `torch.expand()`

위에서 차원 확장&축소와 관련된 메서드 `squeeze()` & `unsqueeze()`를 설명했으니 이제 확장된 차원크기를 조정하는 메서드를 알아보자

In [34]:
```
# 알아보기 쉽게 임의 데이터 하나를 난수(정수 데이터타입)
origin_data = torch.randint(0, 4, (3,2))

# 차원 확장 [3, 2] -> [1, 3, 2]
temp_data = origin_data.unsqueeze(0)

# 확장된 차원을 반복하여 늘림 [1, 3, 2] -> [4, 3, 2]
exp_res_data = temp_data.expand(4, 3, 2)

# 차원 사이즈가 어떻게 늘어났는지 확인
print(exp_res_data.shape)

# 원소가 어떻게 늘어났는지 확인하기
print(exp_res_data)
```

```
torch.Size([4, 3, 2])
tensor([[[0, 0],
         [1, 0],
         [2, 1]],

        [[0, 0],
         [1, 0],
         [2, 1]],

        [[0, 0],
         [1, 0],
         [2, 1]],

        [[0, 0],
         [1, 0],
         [2, 1]]])
```

In [35]:
```
# 확장된 차원을 repeat메서드로 늘림
rep_res_data = temp_data.repeat(4, 1, 1)

# 차원 사이즈가 어떻게 늘어났는지 확인
print(rep_res_data.shape)

# 원소가 어떻게 늘어났는지 확인하기
print(rep_res_data)
```

```
torch.Size([4, 3, 2])
tensor([[[0, 0],
         [1, 0],
         [2, 1]],

        [[0, 0],
         [1, 0],
         [2, 1]],

        [[0, 0],
         [1, 0],
         [2, 1]],

        [[0, 0],
         [1, 0],
         [2, 1]]])
```

`expand`는 차원 사이즈가 늘어나도 메모리는 공유되고 있기에
하나의 원소값을 변조하면 공유 메모리로 묶여있는 다른 원소도 같이 영향받는다.

따라서 `expand`로 차원을 늘리는 경우는 브로드 캐스팅 방식으로 읽기전용 데이터를 만들 때 최적화 측면에서 사용하는 메서드라 보면 된다.

반대로 `repeat` 메서드는 차원사이즈가 늘어나면 복제된 데이터는 완전히 독립적으로 운용되어 원소 변조가 발생하는 쓰기데이터일 때 위 메서드를 사용한다.

주요 텐서 변환 함수

view() 함수

- 텐서의 크기를 변경하는 데 사용됩니다.
- 변경 후에도 텐서 원소의 총 개수는 동일해야 합니다.
- 예: `x1_tensor = x_tensor.view(2, 3)`는 3x2 텐서를 2x3 텐서로 변환합니다.

unsqueeze() 함수

- 지정된 위치에 새로운 차원을 추가합니다.
- `dim` 매개변수로 차원을 추가할 위치를 지정합니다.
- 예: `uns_tensor = x_tensor.unsqueeze(0)`는 첫 번째 위치에 새 차원을 추가합니다.

squeeze() 함수

- 크기가 1인 차원을 제거합니다.
- 특정 연산 후 불필요한 차원을 제거할 때 유용합니다.
- 예: `s_tensor = uns_tensor.squeeze()`는 크기가 1인 모든 차원을 제거합니다.

추가적인 텐서 변환 기능

chunk() 함수

- 텐서를 지정된 수의 덩어리로 나눕니다.
- 예: `chunks = torch.chunk(tensor, 3)`는 텐서를 3개의 덩어리로 나눕니다.

split() 함수

- 텐서를 지정된 크기의 덩어리로 나눕니다.
- 예: `splits = torch.split(tensor, 2)`는 텐서를 2개의 원소를 가진 덩어리들로 나눕니다.

이러한 텐서 변환 함수들은 딥러닝 모델 구축 및 데이터 전처리 과정에서 매우 유용하게 사용됩니다.

특히 복잡한 신경망 구조에서 텐서의 형태를 조정하거나, 배치 처리를 위해 데이터의 차원을 조정할 때 자주 활용됩니다.

In [36]:
```
# view() 함수는 텐서의 크기를 변경하는 데 사용됩니다.
# 변경 후에도 텐서 원소의 개수는 같아야 합니다.
import torch
```

```
import numpy as np

# randn() 함수를 사용해 정규분포 텐서 생성하기
# 3x2 텐서를 2x3 텐서로 변환하기
x_tensor = torch.randn(3, 2)
# 이 함수는 평균이 0이고 표준편차가 1인 정규분포에서 무작위로 값을 추출하여 텐서를 채웁니

# 텐서 크기 변경
x1_tensor = x_tensor.view(2,3)
# view() 함수는 텐서의 형태를 변경합니다. 여기서는 3x2 텐서를 2x3 텐서로 변환합니다. 이

# view() 함수는 텐서의 데이터를 실제로 복사하지 않고 같은 데이터를 다른 형태로 보는 방식

print(x_tensor)
print(x1_tensor)
print(x1_tensor.shape)
```

```
tensor([[2.1400, 0.9733],
        [1.7000, 0.0979],
        [0.3879, 1.5896]])
tensor([[2.1400, 0.9733, 1.7000],
        [0.0979, 0.3879, 1.5896]])
torch.Size([2, 3])
```

In [37]:
```
# 이 예제는 텐서의 형태를 변경하는 방법을 보여주며, 데이터 처리나 신경망 구조 설계 시 자

import torch
import numpy as np

# 2차원 텐서
x_tensor = torch.randn(3, 2)
# torch.randn(3, 2)는 3행 2열의 2차원 텐서를 생성합니다. 이 함수는 표준 정규 분포(평균 (

# 1차원으로 텐서 차원 변경
x1_tensor = x_tensor.view(6)
# torch.randn(3, 2)는 3행 2열의 2차원 텐서를 생성합니다. 이 함수는 표준 정규 분포(평균 (
# x_tensor는 원본 2차원 텐서를 보여주고, x1_tensor는 같은 데이터를 1차원으로 펼친 형태를

print(x_tensor)
print(x1_tensor)
print(x1_tensor.shape)
```

```
tensor([[ 0.4026,  0.0643],
        [ 0.5243,  0.3758],
        [ 0.0673, -0.8276]])
tensor([ 0.4026,  0.0643,  0.5243,  0.3758,  0.0673, -0.8276])
torch.Size([6])
```

In [38]:
```
#  PyTorch를 사용하여 1차원 텐서를 여러 개의 작은 텐서로 나누는 예제

import torch
import numpy as np

# 1x6 텐서를 3개의 텐서로 나누기
tensor = torch.tensor([1, 2, 3, 4, 5, 6])
# 1부터 6까지의 정수를 포함하는 1차원 텐서를 생성

# 텐서 분할
chunks = torch.chunk(tensor, 3)  # 3개의 텐서로 나눔
# torch.chunk() 함수는 텐서를 지정된 수의 덩어리로 나눕니다.
```

```
for chunk in chunks:
    print(chunk)
# tensor([1, 2])
# tensor([3, 4])
# tensor([5, 6])
```

```
tensor([1, 2])
tensor([3, 4])
tensor([5, 6])
```

`torch.chunk()` 함수의 특징:

- 가능한 한 균등하게 텐서를 나누려고 시도합니다.
- 텐서의 크기가 chunk 수로 나누어 떨어지지 않을 경우, 마지막 chunk가 더 작을 수 있습니다.
- 각 chunk는 원본 텐서의 뷰(view)이므로, 메모리를 추가로 사용하지 않습니다.

이 기능은 대규모 텐서를 처리할 때나 병렬 처리를 위해 데이터를 나눌 때 유용하게 사용될 수 있습니다.

In [39]:
```
# PyTorch를 사용하여 1차원 텐서를 여러 개의 작은 텐서로 나누는 예제
# torch.split() 함수를 사용하여 텐서를 분할

import torch
import numpy as np

# 1x6 텐서를 2개의 원소를 가지는 텐서로 나누기
splits = torch.split(tensor, 2)  # 각 텐서가 2개의 원소를 가짐
# torch.split() 함수는 텐서를 지정된 크기의 덩어리로 나눕니다.

for split in splits:
    print(split)
# tensor([1, 2])
# tensor([3, 4])
# tensor([5, 6])
```

```
tensor([1, 2])
tensor([3, 4])
tensor([5, 6])
```

`torch.split()` 함수의 특징:

- 지정된 크기(여기서는 2)만큼 텐서를 균등하게 나눕니다.
- 마지막 덩어리가 지정된 크기보다 작을 수 있습니다(이 예제에서는 모든 덩어리가 동일한 크기입니다).
- 각 분할된 텐서는 원본 텐서의 뷰(view)이므로, 메모리를 추가로 사용하지 않습니다.

이 기능은 대규모 텐서를 처리할 때나 배치 처리를 위해 데이터를 나눌 때 유용하게 사용될 수 있습니다.

차원 추가

- unsqueeze() 함수는 차원을 추가하는 기능을 합니다. unsqueeze(x_tensor, dim)
- dim을 0이라고 지정하면 텐서 크기에서 인덱스 0 즉, 첫 번째 자리에 차원을 추가하라는 뜻입니다.

In [40]:
```
import torch
import numpy as np

# 예시 텐서 생성
x_tensor = torch.randn(3, 2)  # 3x2 크기의 랜덤 텐서 생성

# 차원 추가 unsqueeze(x_tensor, dim)

# dim = 0은 차원을 추가할 첫 번째 자리 (가장 바깥쪽)
uns_tensor = x_tensor.unsqueeze(0)
# x_tensor의 첫 번째 위치(가장 바깥쪽)에 새로운 차원을 추가

print(f'차원추가: {uns_tensor}')
print(uns_tensor.shape)

# dim = 1은 차원을 추가할 두 번째 자리
uns1_tensor = x_tensor.unsqueeze(1)
# x_tensor의 두 번째 위치에 새로운 차원을 추가

print(f'차원추가: {uns1_tensor}')
print(uns1_tensor.shape)
```

```
차원추가: tensor([[[-0.7151,  0.3879],
         [-0.9935, -1.8468],
         [-0.4689,  0.8934]]])
torch.Size([1, 3, 2])
차원추가: tensor([[[-0.7151,  0.3879]],

        [[-0.9935, -1.8468]],

        [[-0.4689,  0.8934]]])
torch.Size([3, 1, 2])
```

- squeeze() 함수는 텐서에서 크기가 1인 차원을 제거하는 기능을 합니다.
- 특정 연산 후 불필요한 차원을 제거할 때 사용됩니다.

In [41]:
```
import torch
import numpy as np

# 차원 축소
s_tensor = uns_tensor.squeeze()
print(f'차원축소: {s_tensor}')
print(s_tensor.shape)

s1_tensor = uns1_tensor.squeeze()
print(f'차원축소: {s1_tensor}')
print(s1_tensor.shape)
```

```
차원축소: tensor([[-0.7151,  0.3879],
        [-0.9935, -1.8468],
        [-0.4689,  0.8934]])
torch.Size([3, 2])
차원축소: tensor([[-0.7151,  0.3879],
        [-0.9935, -1.8468],
        [-0.4689,  0.8934]])
torch.Size([3, 2])
```

2.6 브로드캐스팅

- 브로드캐스팅(broadcasting)은 두 텐서 간의 연산이 발생할 때, 텐서의 크기가 일치하지 않으면 연산이 가능하도록 자동으로 크기를 맞춰줍니다.
- 차원 크기가 1인 텐서는 그 차원의 크기를 다른 텐서의 크기에 맞춰 확장됩니다.
- 브로드캐스팅은 실제로 텐서를 확장하지 않고 연산만을 처리하기 때문에 메모리 낭비를 줄일 수 있습니다.

두 텐서의 차원 크기가 같거나 1차원

b의 텐서 크기가 a의 텐서 크기에 자동으로 맞춰진다.

a = torch.tensor([[1, 2],[3, 5]]) b = torch.tensor([1, 2])

c = a + b # (2, 2) + (2,) -> (2, 2)로 브로드캐스팅 print(c) print(c.shape)

In [42]:
```
import torch
import numpy as np

a = torch.tensor([[1, 2],[3, 5]]) # 2x2 크기의 2차원 텐서 생성
b = torch.tensor([1, 2]) # 1차원 텐서 생성
c = a * b
print(c)
```

```
tensor([[ 1,  4],
        [ 3, 10]])
```

이 코드는 PyTorch를 사용하여 텐서 간의 브로드캐스팅 곱셈 연산을 수행합니다. 각 부분을 자세히 설명하겠습니다:

1. 텐서 생성 :

- a는 2x2 크기의 2차원 텐서입니다: [[1, 2], [3, 5]]
- b는 1차원 텐서입니다: [1, 2]

2. 브로드캐스팅 곱셈 :

c = a * b는 텐서 a와 b의 요소별 곱셈을 수행합니다. 이 과정에서 브로드캐스팅이 적용됩니다.

3. 브로드캐스팅 과정 :

- b는 자동으로 [[1, 2], [1, 2]]로 확장됩니다.
- 이는 a의 형태와 일치하도록 b가 "브로드캐스트" 되는 것입니다.

4. 실제 연산 :

[[1, 2], * [[1, 2], = [[1_1, 2_2], [3, 5]] [1, 2]] [3_1, 5_2]]

5. 결과 출력 : tensor([[1, 4], [3, 10]])

이 예제는 브로드캐스팅의 강력함을 보여줍니다. 크기가 다른 텐서 간의 연산을 가능하게 하여 코드를 간결하게 만들고 메모리 사용을 최적화합니다. 브로드캐스팅은 딥러닝에서 배치 처리나 다양한 차원의 데이터를 다룰 때 매우 유용합니다.

$$\begin{bmatrix} 1 & 2 \\ 3 & 5 \end{bmatrix} [1\ 2] = \begin{bmatrix} 1 & 2 \\ 3 & 5 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 1 & 2 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 3 & 10 \end{bmatrix}$$

그림 2.2 브로드캐스팅 곱셈

In [43]:
```python
import torch
import numpy as np

# 3차원 텐서와 2차원 텐서
A = torch.randn(3, 4, 5)  # 크기 (3, 4행, 5열)
B = torch.randn(4, 5)     # 크기 (4, 5)

C = A + B  # (3, 4, 5) + (4, 5) -> (3, 4, 5)로 브로드캐스팅
print(A)
print(B)
print(C.shape)
```

```
tensor([[[-0.1017,  0.5508, -0.0263, -1.1689,  0.1823],
         [ 0.6236, -0.6920, -0.5608, -0.7498, -1.4801],
         [ 1.1648, -0.6563,  0.0324, -0.8441, -0.3976],
         [-0.0877,  0.8775,  0.8003,  0.8570, -1.0512]],

        [[-2.0099,  1.0895,  0.0305, -1.1157,  0.2162],
         [-0.5469, -1.3497,  0.2899,  0.0644,  0.9533],
         [-1.2488, -0.0140,  0.2980,  0.8563, -0.4699],
         [ 1.1343, -0.8757,  1.3432,  0.2651, -1.0929]],

        [[ 1.5422, -2.2483,  0.3372, -0.4801, -0.8077],
         [ 0.1153, -0.1846, -1.1562, -3.0225,  0.7471],
         [ 0.0901,  0.3301,  0.2159,  0.8252, -0.4201],
         [-2.1411, -0.6138, -0.6678,  0.6670, -0.7289]]])
tensor([[-1.2313,  1.7790, -1.4732, -0.1007,  0.3938],
        [-0.2928, -1.0828, -0.2046,  0.2485, -1.1299],
        [ 0.8741, -1.0767,  0.0259, -1.1824,  0.0313],
        [-0.2953,  0.6649, -0.7653,  1.2066, -0.3395]])
torch.Size([3, 4, 5])
```

이 코드는 PyTorch를 사용하여 3차원 텐서와 2차원 텐서 간의 브로드캐스팅 연산을 수행하는 예제입니다. 각 부분을 자세히 설명하겠습니다:

1. 텐서 생성:
   - `A = torch.randn(3, 4, 5)` : 3x4x5 크기의 3차원 텐서를 생성합니다. 이는 3개의 4x5 행렬로 구성됩니다.
   - `B = torch.randn(4, 5)` : 4x5 크기의 2차원 텐서(행렬)를 생성합니다.
2. 브로드캐스팅 연산:
   `C = A + B` 는 텐서 A와 B의 요소별 덧셈을 수행합니다. 이 과정에서 브로드캐스팅이 적용됩니다.
3. 브로드캐스팅 과정:
   - B(4, 5)는 자동으로 (1, 4, 5)로 확장됩니다.
   - 그 다음, B는 A의 첫 번째 차원(3)을 따라 복제되어 (3, 4, 5) 형태로 확장됩니다.

- 이제 A와 확장된 B는 같은 형태(3, 4, 5)를 가지게 되어 요소별 덧셈이 가능해집니다.

4. 결과:

- `C.shape` 는 (3, 4, 5)를 출력합니다. 이는 결과 텐서 C가 A와 같은 형태를 가짐을 나타냅니다.

이 예제는 PyTorch의 브로드캐스팅 기능을 잘 보여줍니다. 브로드캐스팅은 크기가 다른 텐서 간의 연산을 가능하게 하여 코드를 간결하게 만들고 메모리 사용을 최적화합니다. 이는 특히 배치 처리나 다양한 차원의 데이터를 다룰 때 매우 유용한 기능입니다.

In [44]:
```python
import torch

# 예시 텐서 생성 (2x3 텐서)
tensor = torch.tensor([[1, 2, 3], [4, 5, 6]])

# 첫 번째 행만 선택
slice_1 = tensor[0, :]
# 0은 첫 번째 행을, :은 모든 열을 선택
print(slice_1)  # tensor([1, 2, 3])

# 두 번째 열만 선택
slice_2 = tensor[:, 1]
# :은 모든 행을, 1은 두 번째 열(0부터 시작)을 선택
print(slice_2)  # tensor([2, 5])

# 두 번째 행과 세 번째 열의 원소 선택
slice_3 = tensor[1, 2]
print(slice_3)  # tensor(6)

# 첫 번째 행의 첫 번째와 두 번째 열을 선택, 0:2는 인덱스 0부터 1까지를 의미
slice_4 = tensor[0, 0:2]
print(slice_4)  # tensor([1, 2])
```

```
tensor([1, 2, 3])
tensor([2, 5])
tensor(6)
tensor([1, 2])
```

In [45]:
```python
import torch

# 2x2 텐서 두 개 생성
tensor_a = torch.tensor([[1, 2], [3, 4]])
tensor_b = torch.tensor([[5, 6], [7, 8]])

# 행 기준(0번 축)으로 연결 (2x2 + 2x2 = 4x2)
concat_1 = torch.cat((tensor_a, tensor_b), dim=0)
print(concat_1)
# tensor([[1, 2],
#         [3, 4],
#         [5, 6],
#         [7, 8]])

# 열 기준(1번 축)으로 연결 (2x2 + 2x2 = 2x4)
concat_2 = torch.cat((tensor_a, tensor_b), dim=1)
print(concat_2)
# tensor([[1, 2, 5, 6],
#         [3, 4, 7, 8]])
```

```
tensor([[1, 2],
        [3, 4],
        [5, 6],
        [7, 8]])
tensor([[1, 2, 5, 6],
        [3, 4, 7, 8]])
```

- 지금까지 생성된 텐서는 CPU에 저장이 됩니다.
- GPU를 사용하고 싶다면 GPU 사용 여부를 확인 하고 .to 메소드를 사용하면 GPU로 텐서를 저장할 수 있습니다.

In [46]:
```python
# PyTorch에서 GPU를 사용할 수 있는지 확인하고, 사용 가능한 경우 텐서를 GPU로 이동시키는

import torch

# 예시 텐서 생성
C = torch.tensor([1, 2, 3, 4, 5])

if torch.cuda.is_available():
    tensor = C.to("cuda")
    print(f"텐서가 {tensor.device}로 이동되었습니다.")
else:
    print("CUDA를 사용할 수 없어 텐서가 CPU에 남아있습니다.")
```

```
텐서가 cuda:0로 이동되었습니다.
```

1. `torch.cuda.is_available()` :
    - 이 함수는 시스템에 CUDA 지원 GPU가 있고 PyTorch가 이를 사용할 수 있는지 확인합니다.
    - GPU가 사용 가능하면 True를, 그렇지 않으면 False를 반환합니다.
2. `if` 조건문:
    - GPU가 사용 가능한 경우에만 내부 코드를 실행합니다.
3. `tensor = C.to("cuda")` :
    - 'C'라는 기존 텐서를 GPU 메모리로 이동시킵니다.
    - `.to("cuda")` 메서드는 텐서를 CPU에서 GPU로 전송합니다.

주의사항 :

- 이 코드는 'C'라는 텐서가 이미 정의되어 있다고 가정합니다.
- GPU로 텐서를 이동시키면 연산 속도가 크게 향상될 수 있습니다.
- GPU에서의 연산은 대규모 신경망이나 데이터셋 처리에 특히 유용합니다.
- GPU가 없는 시스템에서는 텐서가 CPU에 남아 있게 됩니다.

이 방식은 GPU가 있는 시스템에서는 자동으로 GPU를 활용하고, 없는 시스템에서도 코드가 정상적으로 실행되도록 하는 유연한 접근법입니다.

GPU 가용성에 따라 자동으로 적응하여, 다양한 환경에서 코드의 유연한 실행을 가능하게 합니다.

GPU 사용 시 성능이 크게 향상되며, 특히 대규모 신경망이나 데이터셋 처리에 유용합니다.

In [4]:
```python
import torch

# GPU 장치 사용 여부 확인
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(device)
```

```
cuda
```

- 이 코드는 PyTorch에서 GPU 사용 가능 여부를 확인하고, 사용할 장치를 설정하는 중요한 부분입니다.

1. `torch.cuda.is_available()`:

- torch.cuda.is_available() 함수는 CUDA가 사용 가능한지 확인합니다.
- 이는 시스템에 CUDA 지원 GPU가 있고 PyTorch가 GPU를 사용할 수 있도록 설정되어 있는지 검사합니다.

2. `torch.device()`

- torch.device() 함수는 텐서나 모듈을 할당할 장치를 지정합니다.
- 삼항 연산자를 사용하여, CUDA가 사용 가능하면 "cuda"를, 그렇지 않으면 "cpu"를 선택합니다.
- Pytorch의 Tensor데이터는 GPU ⟺ CPU 으로 연산 위치가 시시각각 변하는 경우가 많기에 선언한 Tensor 변수가 어느 위치에 있어야 하는지 잘 알아야 합니다.

결과적으로 device 변수에는 "cuda" 또는 "cpu" 중 하나가 할당됩니다.

이 코드의 장점은 GPU가 있는 시스템에서는 자동으로 GPU를 사용하고, GPU가 없는 시스템에서는 CPU를 사용하도록 하여 코드의 호환성을 높입니다.

이후 텐서나 모델을 이 device로 이동시켜 연산을 수행할 수 있습니다.