

Deep Learning Programming

Lecture 2.1: Pytorch Introduction

Sangryul Jeon

School of Computer Science and Engineering

srjeonn@pusan.ac.kr

PART 1: Manipulation of Tensors

Slides borrowed from Naver Connect Foundation

1. Tensor의 인덱싱과 슬라이싱

1.1 Tensor의 indexing & slicing

1.2 학습정리

2. Tensor의 모양변경

2.1 view() 메서드를 활용한 Tensor의 모양변경

2.2 flatten() 함수를 활용한 Tensor의 평탄화

2.3 reshape() 메서드를 활용한 Tensor의 모양변경

2.4 transpose() 메서드를 활용한 Tensor의 모양변경

2.5 squeeze() 함수를 활용한 Tensor의 차원 축소

2.6 unsqueeze() 함수를 활용한 Tensor의 차원 확장

2.7 stack() 함수를 활용한 Tensor들 간의 결합

2.8 학습정리

1.

Tensor의 인덱싱과 슬라이싱

이번 챕터에서는 Tensor의 인덱싱과 슬라이싱에 대해 살펴봅니다.

1.1 Tensor의 indexing & slicing

1. Tensor의 인덱싱과 슬라이싱

indexing? slicing?

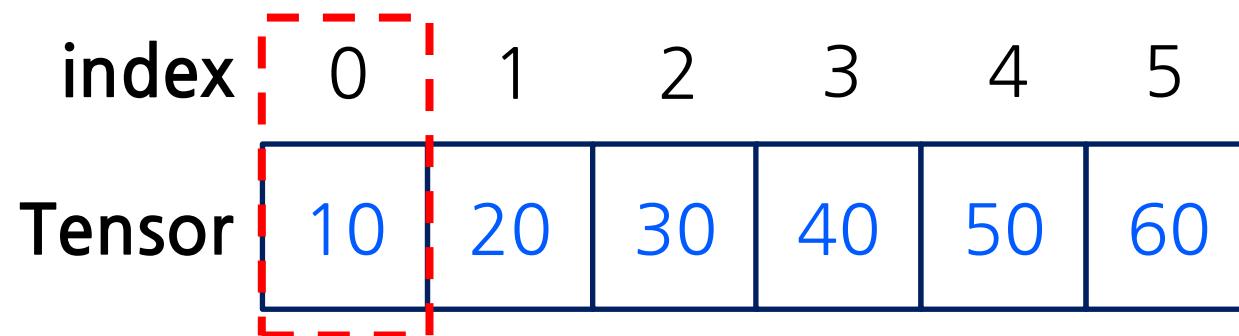
- PyTorch의 Tensor에 대해서 indexing과 slicing은 NumPy에서의 사용법과 유사함
 - indexing이란 Tensor의 특정 위치의 요소에 접근하는 것을 의미함
 - slicing이란 부분집합을 선택하여 새로운 Sub Tensor 생성하는 과정을 의미함
- indexing과 slicing 기법은 모두 머신러닝 모델을 구현하는 데 필수적인 요소로서, Tensor 데이터에 대한 높은 수준의 조작을 가능하게 함

1.1 Tensor의 indexing & slicing

1. Tensor의 인덱싱과 슬라이싱

1-D Tensor의 indexing 표현

- PyTorch에서 index는 생성된 Tensor의 각각의 요소 값을 참조하기 위해서 사용함
- `a = torch.tensor([10, 20, 30, 40, 50, 60])`의 1-D Tensor를 생성했을 때,
 - 0번째 index에 접근하기 위한 코드 표현은 `a[0]`,
 - 1번째 index에 접근하기 위한 코드 표현은 `a[1]...`

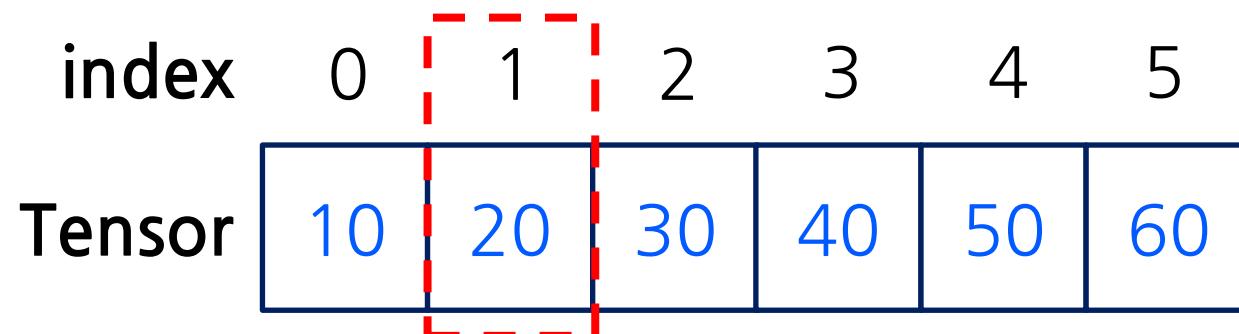


1.1 Tensor의 indexing & slicing

1. Tensor의 인덱싱과 슬라이싱

1-D Tensor의 indexing 표현

- PyTorch에서 index는 생성된 Tensor의 각각의 요소 값을 참조하기 위해서 사용함
- `a = torch.tensor([10, 20, 30, 40, 50, 60])`의 1-D Tensor를 생성했을 때,
 - 0번째 index에 접근하기 위한 코드 표현은 `a[0]`,
 - 1번째 index에 접근하기 위한 코드 표현은 `a[1]...`

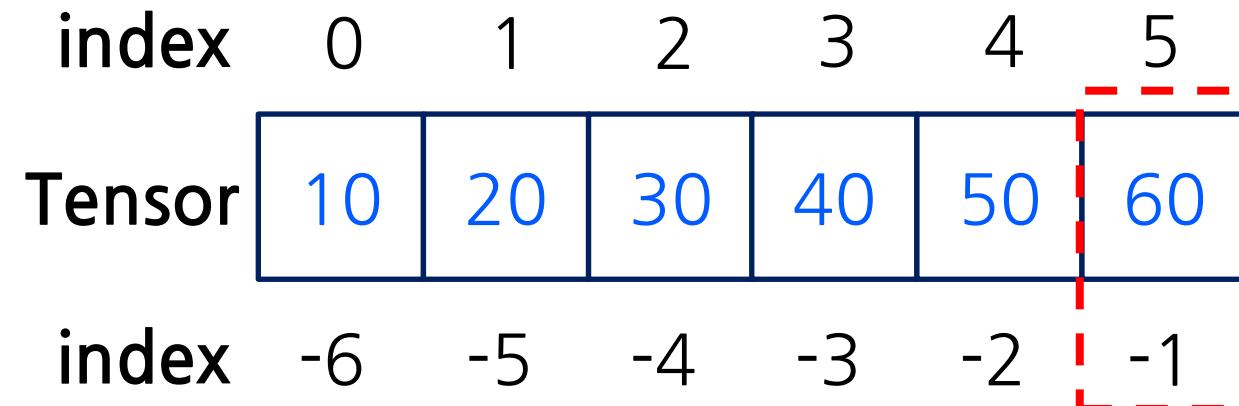


1.1 Tensor의 indexing & slicing

1. Tensor의 인덱싱과 슬라이싱

1-D Tensor의 indexing 표현

- PyTorch에서는 음수 index 사용을 지원함
- $a = \text{torch.tensor}([10, 20, 30, 40, 50, 60])$ 의 1-D Tensor를 생성했을 때,
 - 6번째 index에 접근하기 위한 코드 표현은 $a[-1]$,
 - 5번째 index에 접근하기 위한 코드 표현은 $a[-2]...$

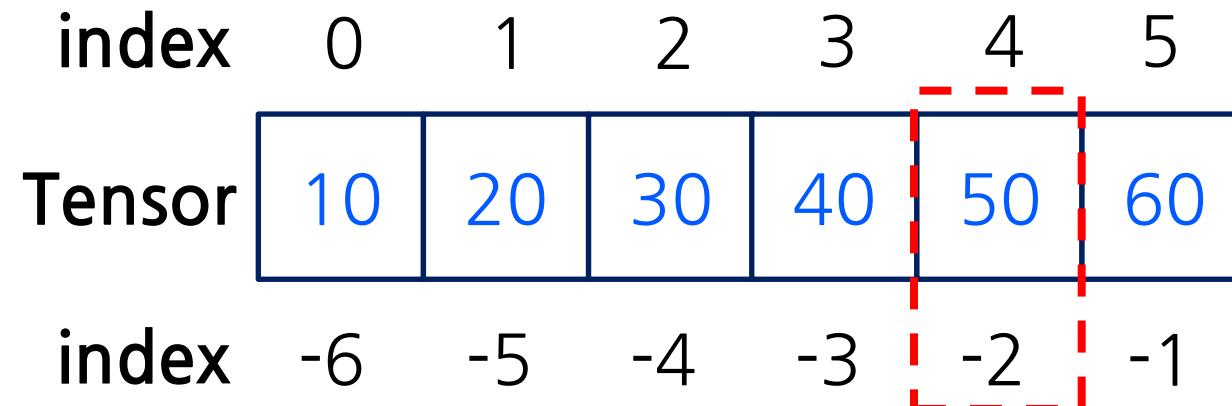


1.1 Tensor의 indexing & slicing

1. Tensor의 인덱싱과 슬라이싱

1-D Tensor의 indexing 표현

- PyTorch에서는 음수 index 사용을 지원함
- $a = \text{torch.tensor}([10, 20, 30, 40, 50, 60])$ 의 1-D Tensor를 생성했을 때,
 - 6번째 index에 접근하기 위한 코드 표현은 $a[-1]$,
 - 5번째 index에 접근하기 위한 코드 표현은 $a[-2]...$

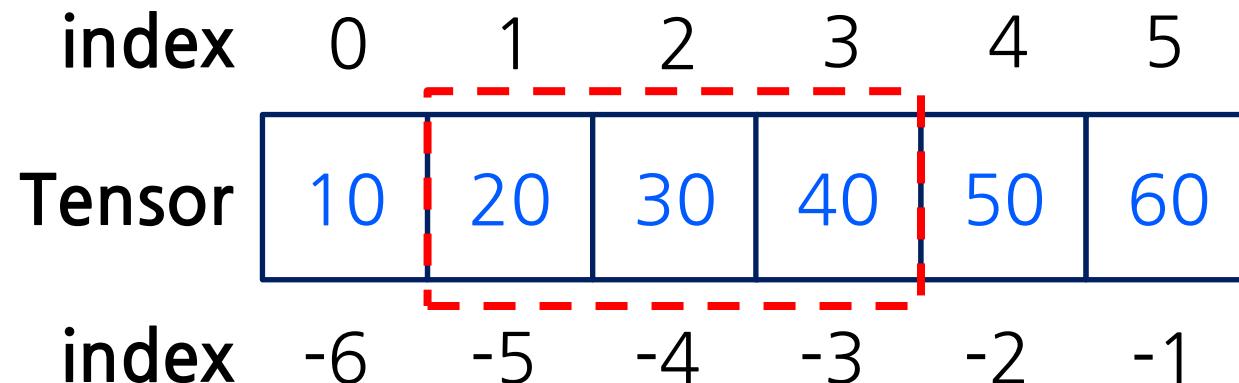


1.1 Tensor의 indexing & slicing

1. Tensor의 인덱싱과 슬라이싱

1-D Tensor의 slicing 표현

- PyTorch에서 slicing은 생성된 Tensor의 여러 개의 요소 값을 가져오기 위해서 사용함
- $a = \text{torch.tensor}([10, 20, 30, 40, 50, 60])$ 의 1-D Tensor를 생성했을 때,
 - 1번째 index의 요소부터 4번째 index의 요소 이전까지의 값으로
Sub Tensor를 생성하는 [코드 표현](#)은 $a[1:4]$ 또는 $a[-5:-2]$

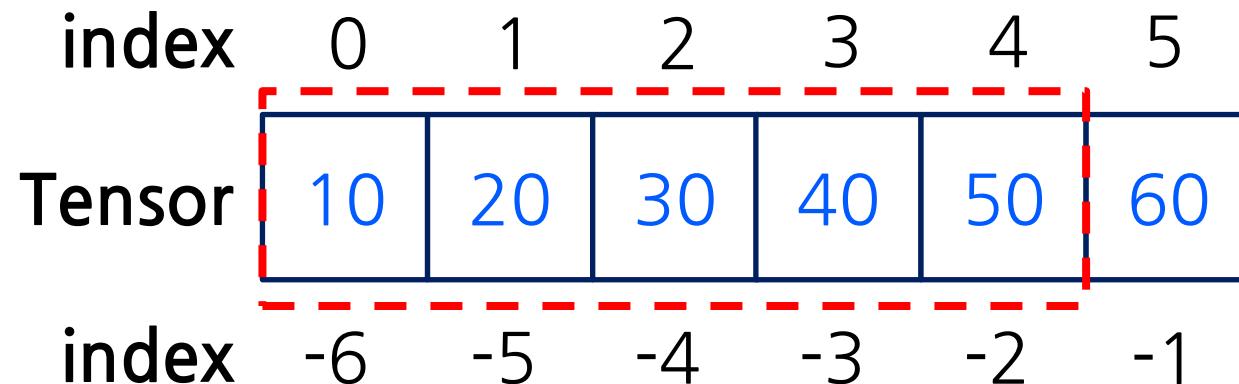


1.1 Tensor의 indexing & slicing

1. Tensor의 인덱싱과 슬라이싱

1-D Tensor의 slicing 표현

- PyTorch에서 slicing은 생성된 Tensor의 여러 개의 요소 값을 가져오기 위해서 사용함
- $a = \text{torch.tensor}([10, 20, 30, 40, 50, 60])$ 의 1-D Tensor를 생성했을 때,
 - 0번째 index의 요소부터 5번째 index의 요소 이전까지의 값으로
Sub Tensor를 생성하는 [코드 표현](#)은 $a[:5]$ 또는 $a[:-1]$

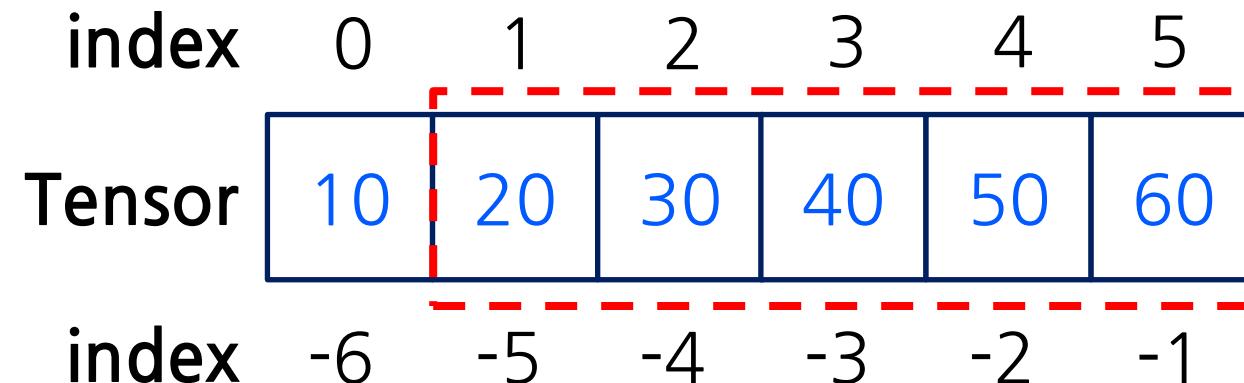


1.1 Tensor의 indexing & slicing

1. Tensor의 인덱싱과 슬라이싱

1-D Tensor의 slicing 표현

- PyTorch에서 slicing은 생성된 Tensor의 여러 개의 요소 값을 가져오기 위해서 사용함
- $a = \text{torch.tensor}([10, 20, 30, 40, 50, 60])$ 의 1-D Tensor를 생성했을 때,
 - 1번째 index의 요소부터 마지막 index의 요소까지의 값으로
Sub Tensor를 생성하는 [코드 표현](#)은 $a[1:]$ 또는 $a[-5:]$

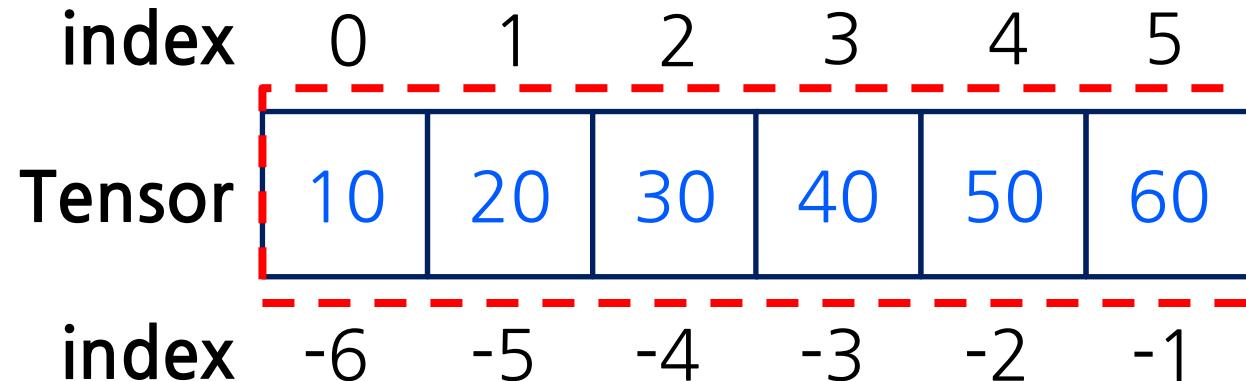


1.1 Tensor의 indexing & slicing

1. Tensor의 인덱싱과 슬라이싱

1-D Tensor의 slicing 표현

- PyTorch에서 slicing은 생성된 Tensor의 여러 개의 요소 값을 가져오기 위해서 사용함
- $a = \text{torch.tensor}([10, 20, 30, 40, 50, 60])$ 의 1-D Tensor를 생성했을 때,
 - 0번째 index의 요소부터 마지막 index의 요소까지의 값으로
Sub Tensor를 생성하는 [코드 표현](#)은 `a[:]`

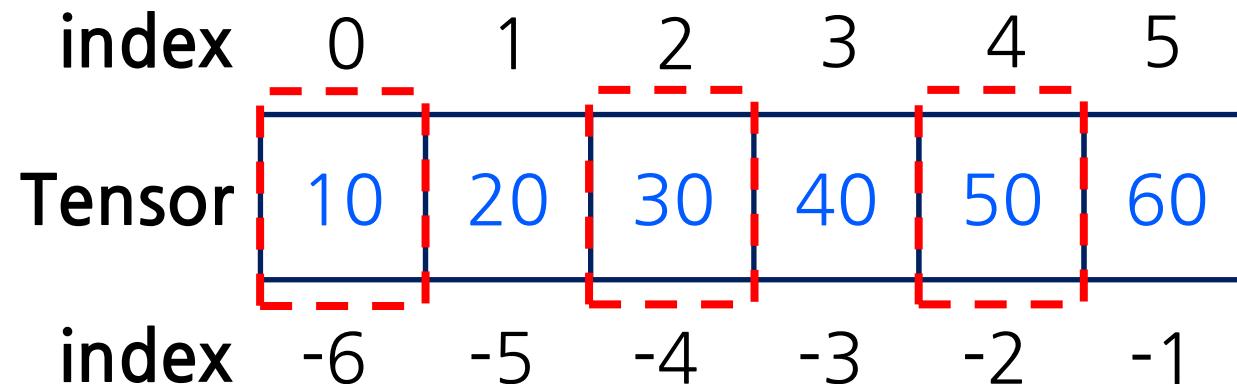


1.1 Tensor의 indexing & slicing

1. Tensor의 인덱싱과 슬라이싱

1-D Tensor의 slicing 표현

- PyTorch에서 slicing은 생성된 Tensor의 여러 개의 요소 값을 가져오기 위해서 사용함
- $a = \text{torch.tensor}([10, 20, 30, 40, 50, 60])$ 의 1-D Tensor를 생성했을 때,
 - 0번째 index의 요소부터 5번째 index의 요소 이전까지의 값을 2단계씩 건너뛰며 Sub Tensor를 생성하는 [코드 표현](#)은 $a[0:5:2]$ 또는 $a[-6:-1:2]$ 또는 $a[:5:2]$ 등

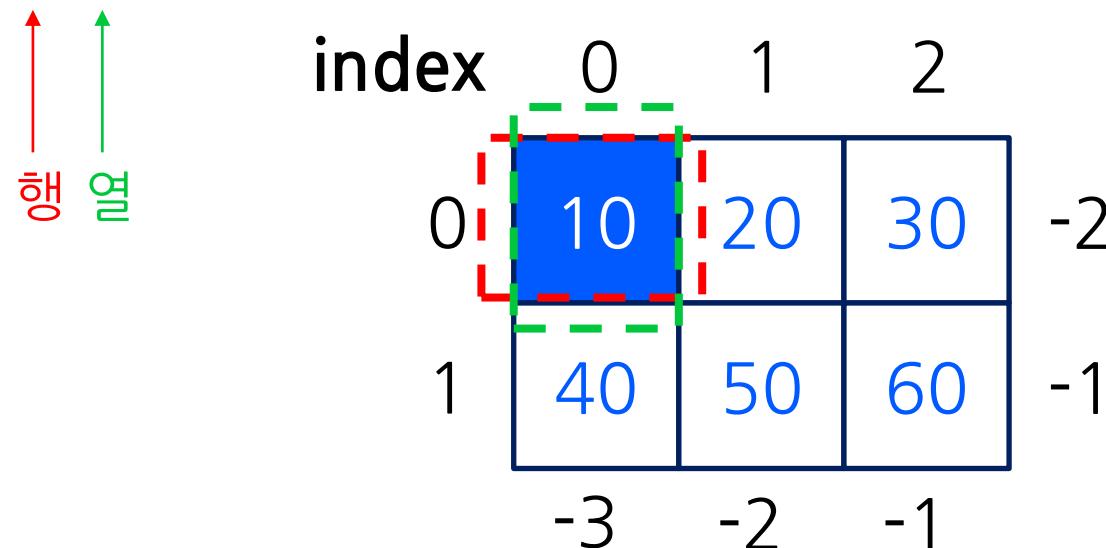


1.1 Tensor의 indexing & slicing

1. Tensor의 인덱싱과 슬라이싱

2-D Tensor의 indexing 표현

- `b = torch.tensor([[10, 20, 30], [40, 50, 60]])`의 2-D Tensor를 생성했을 때,
 - 0번째 index행, 0번째 index열에 있는 요소에 접근하기 위한
코드 표현은 `b[0, 0]`

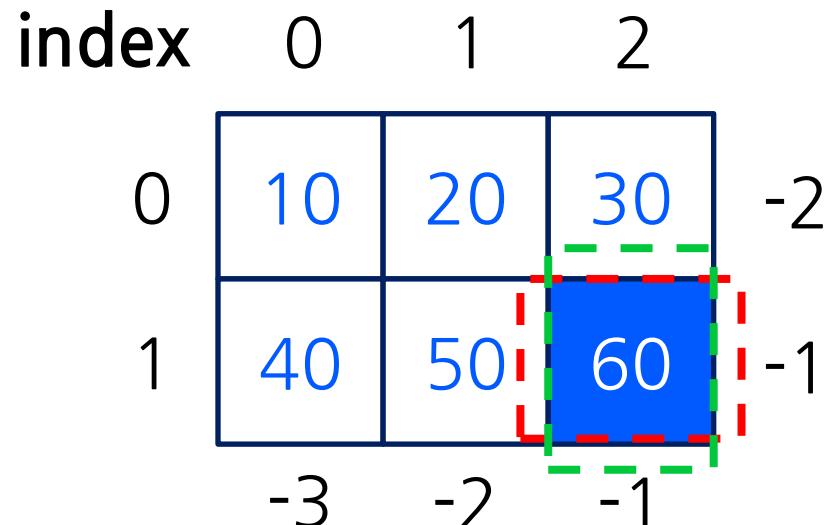


1.1 Tensor의 indexing & slicing

1. Tensor의 인덱싱과 슬라이싱

2-D Tensor의 indexing 표현

- $b = \text{torch.tensor}([[10, 20, 30], [40, 50, 60]])$ 의 2-D Tensor를 생성했을 때,
 - 1번째 index행, 2번째 index열에 있는 요소에 접근하기 위한 코드 표현은 $b[1, 2]$ 또는 $b[-1, -1]$

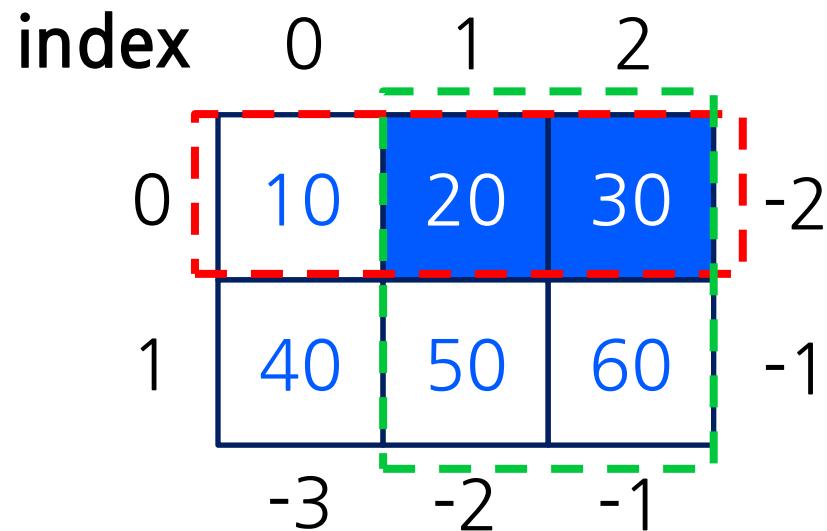


1.1 Tensor의 indexing & slicing

1. Tensor의 인덱싱과 슬라이싱

2-D Tensor의 slicing 표현

- $b = \text{torch.tensor}([[10, 20, 30], [40, 50, 60]])$ 의 2-D Tensor를 생성했을 때,
 - 0번째 index 행의 요소들과 1번째 index 열 이후의 요소들과의 교집합으로 Sub Tensor를 생성하는 코드 표현은 $b[0, 1:]$ 또는 $b[0, -2:]$

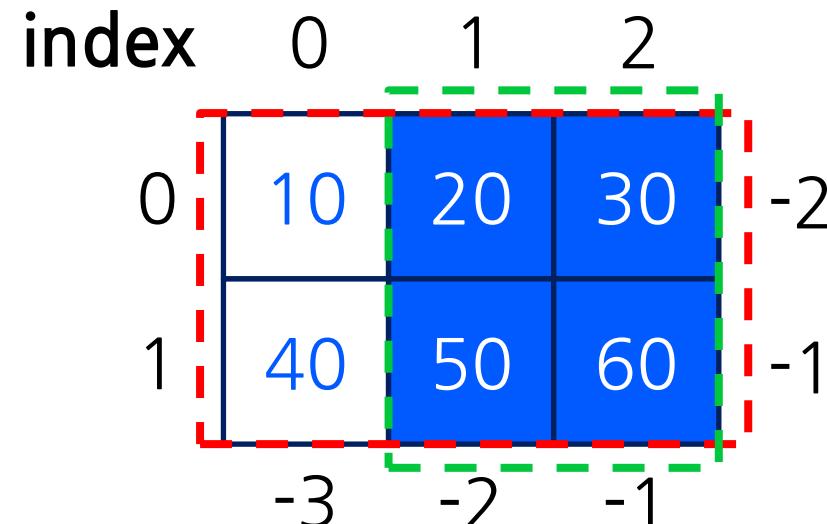


1.1 Tensor의 indexing & slicing

1. Tensor의 인덱싱과 슬라이싱

2-D Tensor의 slicing 표현

- $b = \text{torch.tensor}([[10, 20, 30], [40, 50, 60]])$ 의 2-D Tensor를 생성했을 때,
 - 모든 index 행의 요소들과 1번째 index 열 이후의 요소들과의 교집합으로 Sub Tensor를 생성하는 코드 표현은 `b[:, 1:]` 또는 `b[:, -2:]`

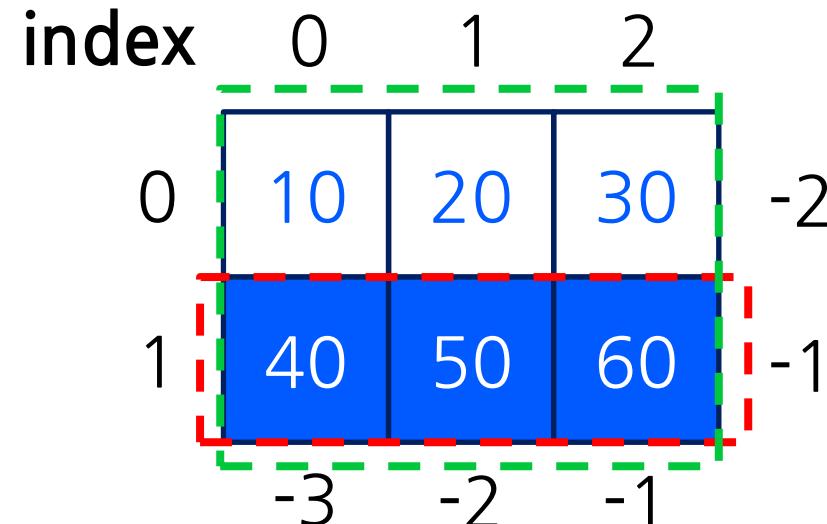


1.1 Tensor의 indexing & slicing

1. Tensor의 인덱싱과 슬라이싱

2-D Tensor의 slicing 표현

- $b = \text{torch.tensor}([[10, 20, 30], [40, 50, 60]])$ 의 2-D Tensor를 생성했을 때,
 - 1번째 index 행의 요소들과 모든 index 열의 요소들과의 교집합으로 Sub Tensor를 생성하는 코드 표현은 $b[1, \dots]$ 또는 $b[1, :]$ 또는 $b[-1, \dots]$ 또는 $b[-1, :]$



Tensor의 인덱싱과 슬라이싱

- indexing이란 Tensor의 특정 위치의 요소에 접근하는 것을 의미한다.
- slicing란 부분집합을 선택하여 새로운 Sub Tensor 생성하는 과정을 의미한다.

2.

Tensor의 모양변경 1

이번 챕터에서는 Tensor의 모양을 변경하는 함수와 메서드에 대해 살펴봅니다.

2.1 view() 메서드를 활용한 Tensor의 모양변경

2. Tensor의 모양변경1

view() 메서드를 활용한 Tensor의 모양변경

- Tensor의 모양을 변경하는 방법 중 하나는 view() 메서드를 활용하는 것임
- 단, view() 메서드는 Tensor의 메모리가 연속적으로 할당된 경우에 사용이 가능함

Tensor의 메모리가 연속적으로 할당된 경우?

2.1 view() 메서드를 활용한 Tensor의 모양변경

2. Tensor의 모양변경1

연속적 메모리 할당의 예시

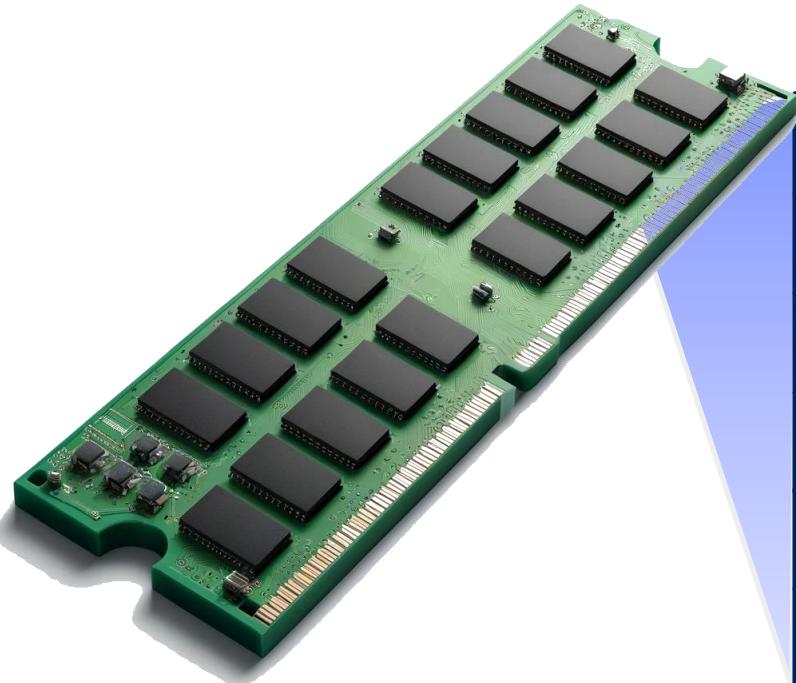
- 예를 들어, $c = \text{torch.tensor}([[0, 1, 2], [3, 4, 5]])$ 의 2-D Tensor를 생성했을 때,
- PyTorch는 해당 Tensor의 데이터 타입과 차원 정보에 기반하여, 컴퓨터 메모리에서 충분한 공간을 할당 받아 데이터를 저장함

$$c = \begin{array}{|c|c|c|} \hline 0 & 1 & 2 \\ \hline 3 & 4 & 5 \\ \hline \end{array}$$

2.1 view() 메서드를 활용한 Tensor의 모양변경

2. Tensor의 모양변경1

연속적 메모리 할당의 시각적 표현



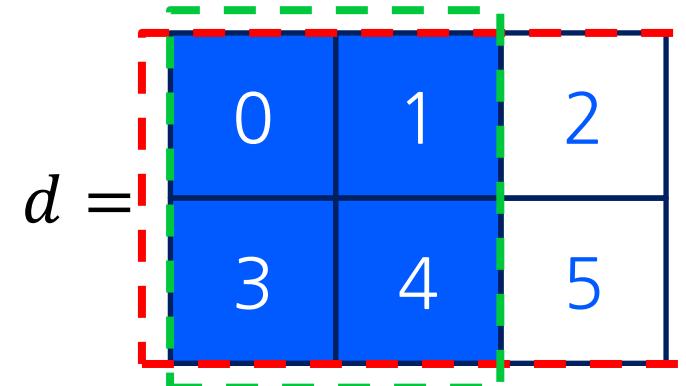
Memory	Address	Element
c[1, 2]	0x14	5
c[1, 1]	0x10	4
c[1, 0]	0x0C	3
c[0, 2]	0x08	2
c[0, 1]	0x04	1
c[0, 0]	0x00	0

2.1 view() 메서드를 활용한 Tensor의 모양변경

2. Tensor의 모양변경1

메모리 레이아웃의 변화로 인한 비연속적 메모리 할당

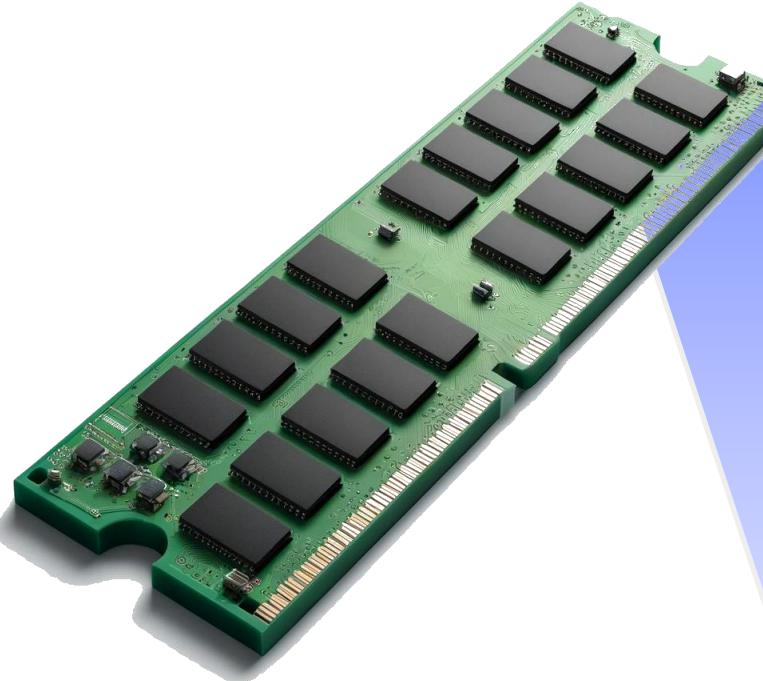
- 하지만, 슬라이싱으로 $d = c[:, :2]$ 의 2-D Tensor를 조작했을 때,
- d 는 원본 Tensor c 의 일부 요소들을 선택하여 Sub-Tensor를 생성하는 과정에서 contiguous 속성이 깨지게 됨



2.1 view() 메서드를 활용한 Tensor의 모양변경

2. Tensor의 모양변경1

메모리 레이아웃의 변화로 인한 비연속적 메모리 할당의 시각적 표현



Memory	Address	Element
c[1, 2]	0x14	5
c[1, 1]	0x10	4
c[1, 0]	0x0C	3
c[0, 2]	0x08	2
c[0, 1]	0x04	1
c[0, 0]	0x00	0

2.1 view() 메서드를 활용한 Tensor의 모양변경

2. Tensor의 모양변경1

Tensor의 메모리가 연속적 또는 비연속적으로 할당되었는지를 확인하는 코드 표현

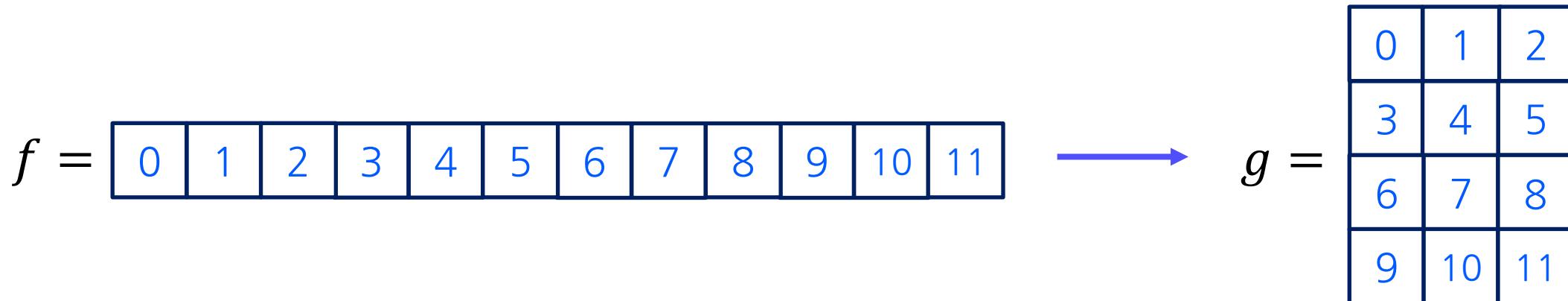
- Tensor의 메모리가 연속적 또는 비연속적으로 할당되었는지를 확인하는 [코드 표현](#)
 - `c.is_contiguous()`
 - `d.is_contiguous()`
- Tensor c 또는 Tensor d의 메모리가 연속적으로 할당되었다면 True를 출력하고 비연속적으로 할당되었다면 False를 출력함

2.1 view() 메서드를 활용한 Tensor의 모양변경

2. Tensor의 모양변경1

view() 메서드를 활용한 Tensor의 모양변경 표현

- $f = \text{torch.arange}(12)$ 의 1-D Tensor를 생성했을 때,
- Tensor f 의 (4×3) view를 생성하여, 2-D Tensor g 를 생성하는 [코드 표현](#)
 - $g = f.\text{view}(4, 3)$ 또는 $g = f.\text{view}(4, -1)$

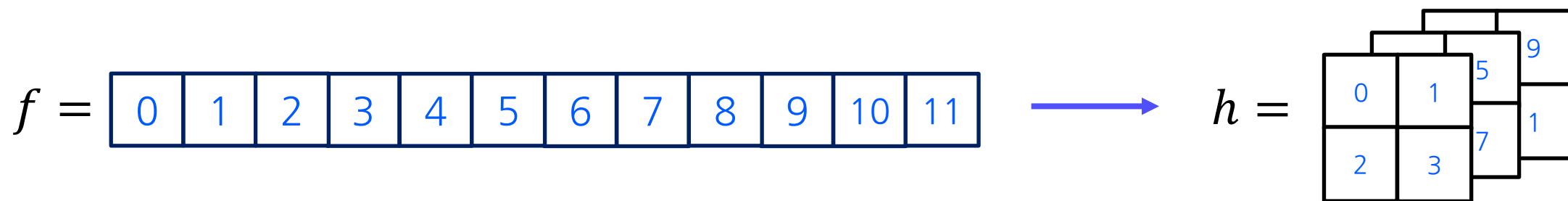


2.1 view() 메서드를 활용한 Tensor의 모양변경

2. Tensor의 모양변경1

view() 메서드를 활용한 Tensor의 모양변경 표현

- $f = \text{torch.arange}(12)$ 의 1-D Tensor를 생성했을 때,
- Tensor f 의 $(3 \times 2 \times 2)$ view를 생성하여, 3-D Tensor h 를 생성하는 코드 표현
 - $h = f.view(3, 2, 2)$ 또는 $h = f.view(3, 2, -1)$



2.2 flatten() 함수를 활용한 Tensor의 평탄화

2. Tensor의 모양변경1

flatten() 함수를 활용한 Tensor의 평탄화

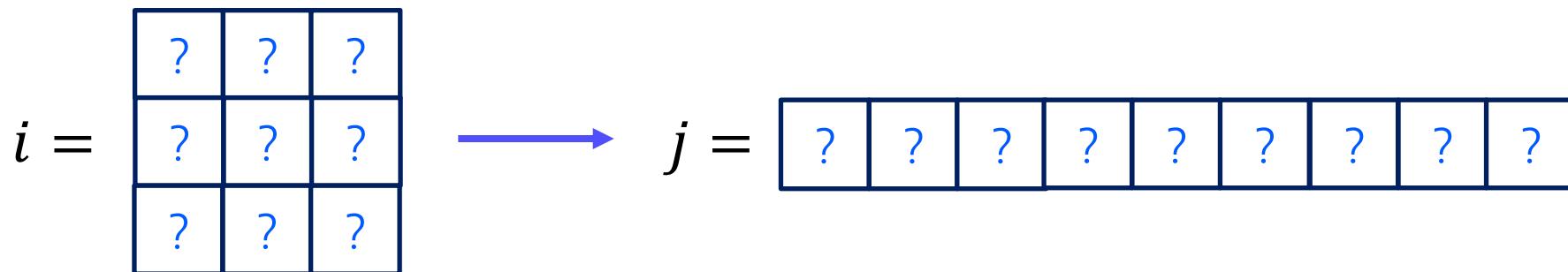
- Tensor를 평탄화하는 모양변경 방법으로 flatten() 함수를 활용할 수 있음
- flatten() 함수는 다차원 데이터를 처리할 때 유용하며, 데이터를 신경망 모델에 적합한 형태로 전처리하기 위해 많이 활용함
 - 전처리란 데이터를 분석하거나 모델에 입력하기 전에 데이터를 적절하게 준비하고 정리하는 과정을 의미함

2.2 flatten() 함수를 활용한 Tensor의 평탄화

2. Tensor의 모양변경1

flatten() 함수를 활용한 1-D Tensor로 평탄화하는 표현

- $i = \text{torch.randn}(3, 3)$ 의 2-D Tensor를 생성했을 때,
- Tensor i 를 1-D Tensor로 평탄화하는 [코드 표현](#)
 - $j = \text{torch.flatten}(i)$ 또는 $j = i.\text{flatten}()$

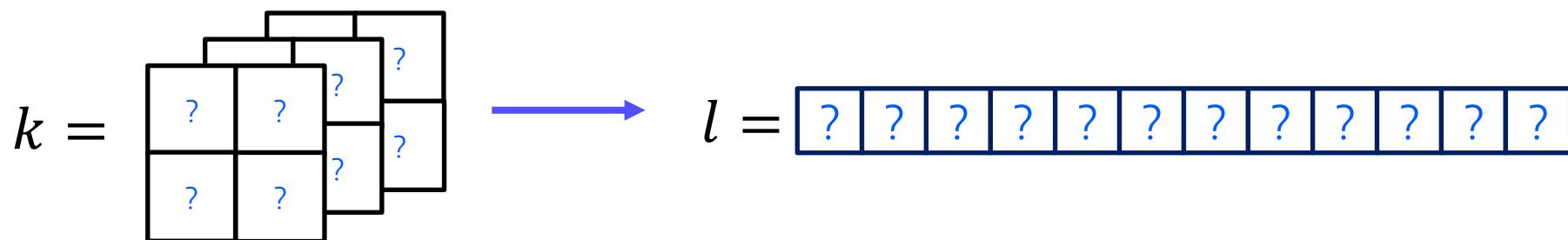


2.2 flatten() 함수를 활용한 Tensor의 평탄화

2. Tensor의 모양변경1

flatten() 함수를 활용한 특정 차원 범위를 평탄화하는 표현

- $k = \text{torch.randn}(3, 2, 2)$ 의 3-D Tensor를 생성했을 때,
- 0번째 차원부터 마지막 차원까지 평탄화를 수행하는 [코드 표현](#)
 - $l = \text{torch.flatten}(k, 0)$
 - 즉, $(3, 2, 2)$ 를 $(12,)$ 크기의 1-D Tensor로 변환시킴

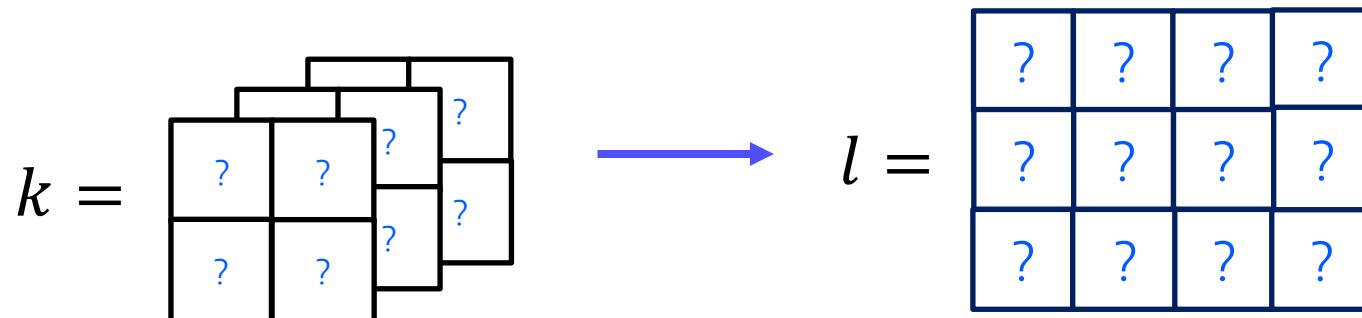


2.2 flatten() 함수를 활용한 Tensor의 평탄화

2. Tensor의 모양변경1

flatten() 함수를 활용한 특정 차원 범위를 평탄화하는 표현

- $k = \text{torch.randn}(3, 2, 2)$ 의 3-D Tensor를 생성했을 때,
- 1번째 차원부터 마지막 차원까지 평탄화를 수행하는 [코드 표현](#)
 - $l = \text{torch.flatten}(k, 1)$
 - 즉, $(3, 2, 2)$ 를 $(3, 4)$ 크기의 2-D Tensor로 변환시킴

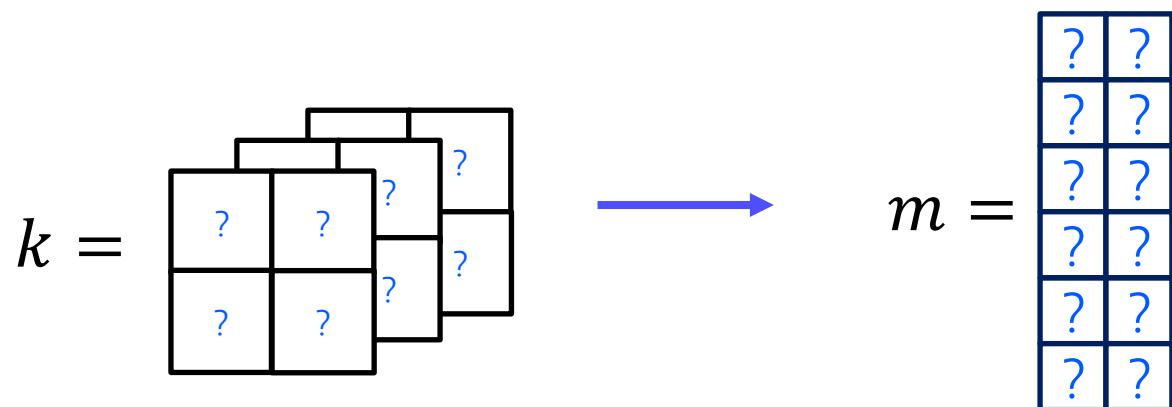


2.2 flatten() 함수를 활용한 Tensor의 평탄화

2. Tensor의 모양변경1

flatten() 함수를 활용한 특정 차원 범위를 평탄화하는 표현

- $k = \text{torch.randn}(3, 2, 2)$ 의 3-D Tensor를 생성했을 때,
- 0번째 차원부터 1번째 차원까지 평탄화를 수행하는 [코드 표현](#)
 - $m = \text{torch.flatten}(k, 0, 1)$
 - 즉, $(3, 2, 2)$ 를 $(6, 2)$ 크기의 2-D Tensor로 변환시킴



2.3 reshape() 메서드를 활용한 Tensor의 모양변경

2. Tensor의 모양변경1

reshape() 메서드를 활용한 Tensor의 모양변경

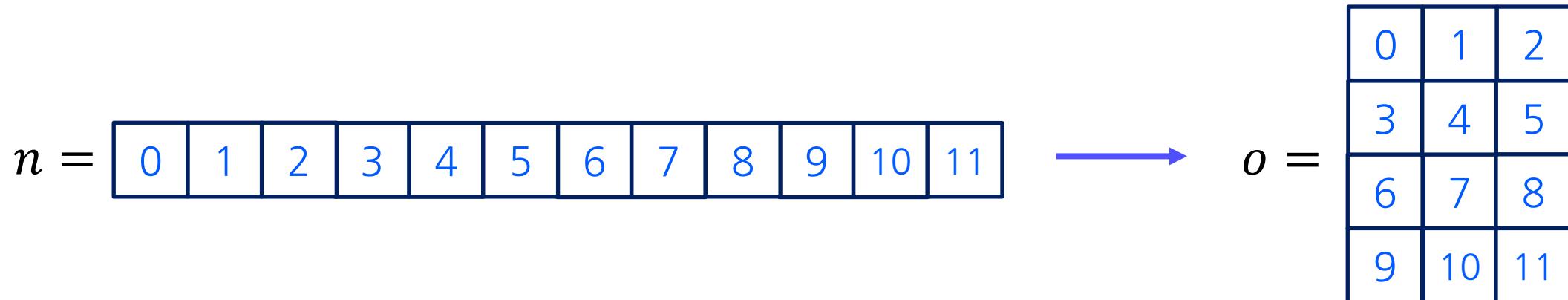
- Tensor의 모양을 변경하는 또 다른 방법은 `reshape()` 메서드를 활용하는 것임
- `reshape()` 메서드는 `view()` 메서드와는 달리 메모리가 연속적이지 않아도 사용 가능함
 - `reshape()` 메서드는 안전하고 유연성이 좋다는 장점이 있으나 성능 저하의 단점 있음
 - 메모리의 연속성이 확실하고 성능이 중요한 경우 `view()` 메서드를 사용하는 것이 좋음

2.3 reshape() 메서드를 활용한 Tensor의 모양변경

2. Tensor의 모양변경1

reshape() 메서드를 활용한 Tensor의 모양변경 표현

- $n = \text{torch.arange}(12)$ 의 1-D Tensor를 생성했을 때,
- Tensor n 을 (4, 3) 모양으로 변경하여, 2-D Tensor o 를 생성하는 코드 표현
 - $o = n.reshape(4, 3)$ 또는 $o = n.reshape(4, -1)$

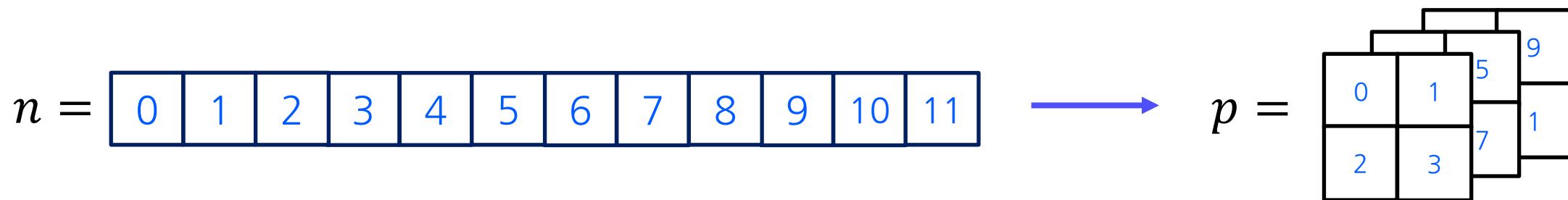


2.3 reshape() 메서드를 활용한 Tensor의 모양변경

2. Tensor의 모양변경1

reshape() 메서드를 활용한 Tensor의 모양변경 표현

- $n = \text{torch.arange}(12)$ 의 1-D Tensor를 생성했을 때,
- Tensor n 을 $(3, 2, 2)$ 모양으로 변경하여, 3-D Tensor p 를 생성하는 [코드 표현](#)
 - $p = n.reshape(3, 2, 2)$ 또는 $p = f.reshape(3, 2, -1)$

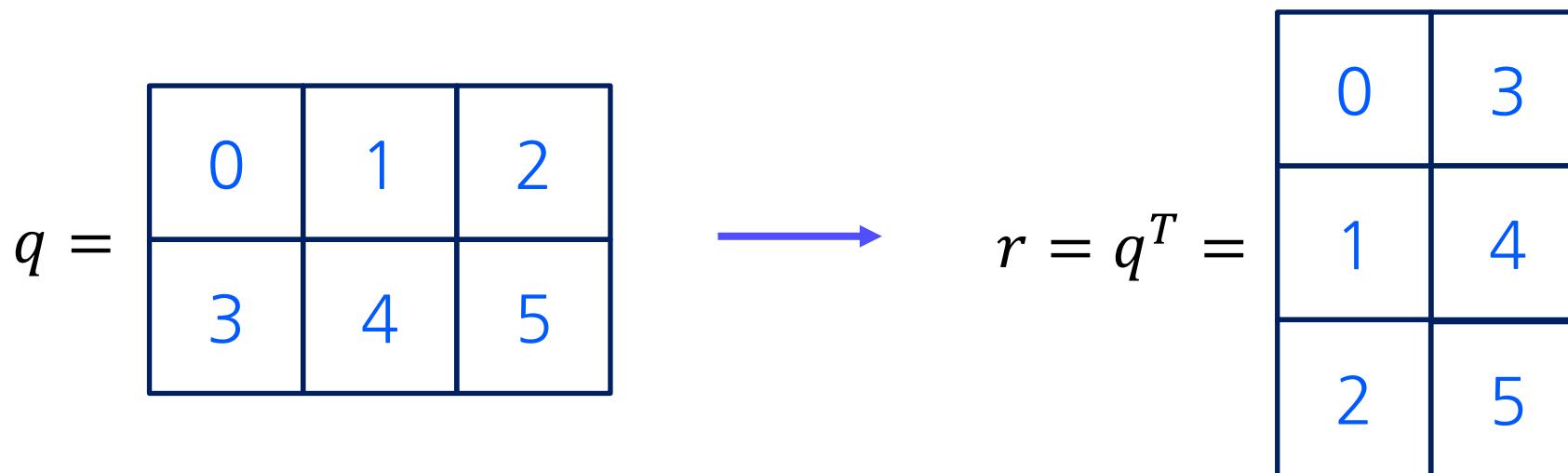


2.4 transpose() 메서드를 활용한 Tensor의 모양변경

2. Tensor의 모양변경1

transpose() 메서드를 활용한 Tensor의 모양변경 표현

- transpose는 Tensor의 특정한 두 차원의 축을 서로 바꾸는 메서드임
- $q = \text{torch.tensor}([[0, 1, 2], [3, 4, 5]])$ 일 때, 0차원과 1차원의 축을 바꾸는 [코드 표현](#)
 - $r = q.\text{transpose}(0, 1)$

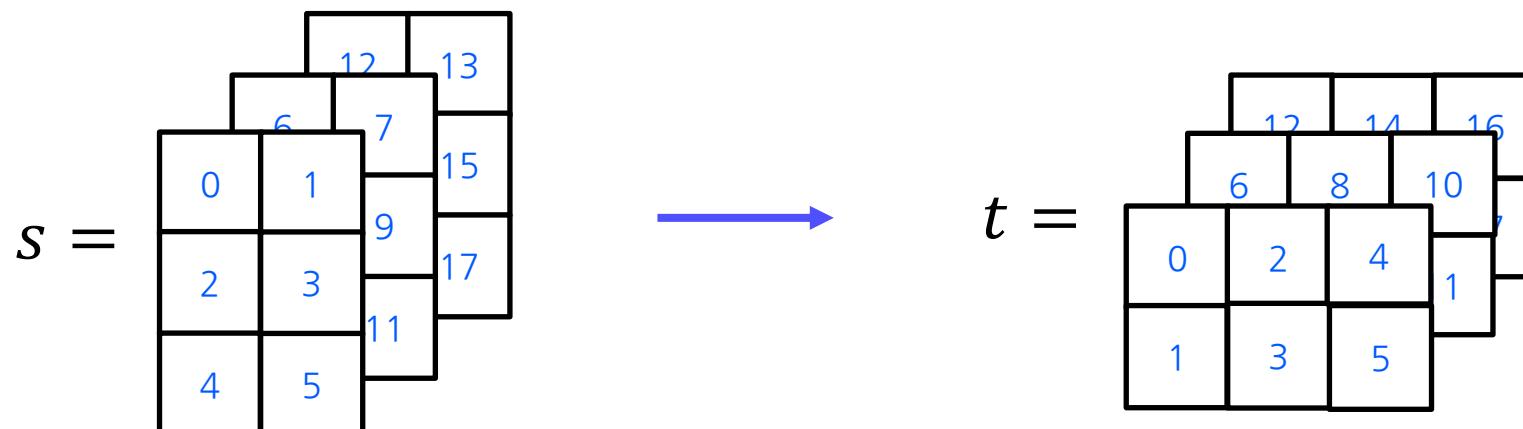


2.4 transpose() 메서드를 활용한 Tensor의 모양변경

2. Tensor의 모양변경1

transpose() 메서드를 활용한 Tensor의 모양변경 표현

- `s = torch.tensor([[[0, 1], [2, 3], [4, 5]], [[6, 7], [8, 9], [10, 11]], [[12, 13], [14, 15], [16, 17]]])`와 같이
3-D Tensor에서 1차원과 2차원의 축을 바꾸는 [코드 표현](#)
 - `t = s.transpose(1, 2)`



2.5 squeeze() 함수를 활용한 Tensor의 차원 축소

2. Tensor의 모양변경1

squeeze() 함수를 활용한 dim이 1인 특정 차원의 축소 코드 표현

- u = torch.randn(1, 3, 4)의 3-D Tensor를 생성했을 때,
Tensor u에서 dim이 1인 특정 차원을 축소하는 [코드 표현](#)
 - `v = torch.squeeze(u)`
- w = torch.randn(1, 1, 4)의 3-D Tensor를 생성했을 때,
Tensor w에서 dim이 1인 특정 차원들을 축소하는 [코드 표현](#)
 - `x = torch.squeeze(w)`
- Tensor w에서 dim이 1인 특정 차원을 축소하는 [코드 표현](#)
 - `x = torch.squeeze(w, dim = 0) or x = torch.squeeze(w, dim = 1)`

2.6 unsqueeze() 함수를 활용한 Tensor의 차원 확장

2. Tensor의 모양변경1

unsqueeze() 함수를 활용한 dim 1인 특정 차원의 확장 코드 표현

- y = torch.randn(3, 4)의 2-D Tensor를 생성했을 때,
Tensor y에서 0차원으로 dim이 1인 특정 차원을 확장하는 [코드 표현](#)
 - `z = torch.unsqueeze(y, dim = 0)`
- Tensor y에서 1차원으로 dim이 1인 특정 차원을 확장하는 [코드 표현](#)
 - `z = torch.unsqueeze(y, dim = 1)`
- Tensor y에서 2차원으로 dim이 1인 특정 차원을 확장하는 [코드 표현](#)
 - `z = torch.unsqueeze(y, dim = 2)`

2.7 stack()함수를 활용한 Tensor들 간의 결합

2. Tensor의 모양변경1

stack() 함수를 활용한 Tensor들 간의 결합에 대한 표현

- red_channel = torch.tensor([[255, 0],
[0, 255]])



255	0
0	255

- green_channel = torch.tensor([[0, 255],
[0, 255]])



0	255
0	255

- blue_channel = torch.tensor([[0, 0],
[255, 0]])



0	0
255	0

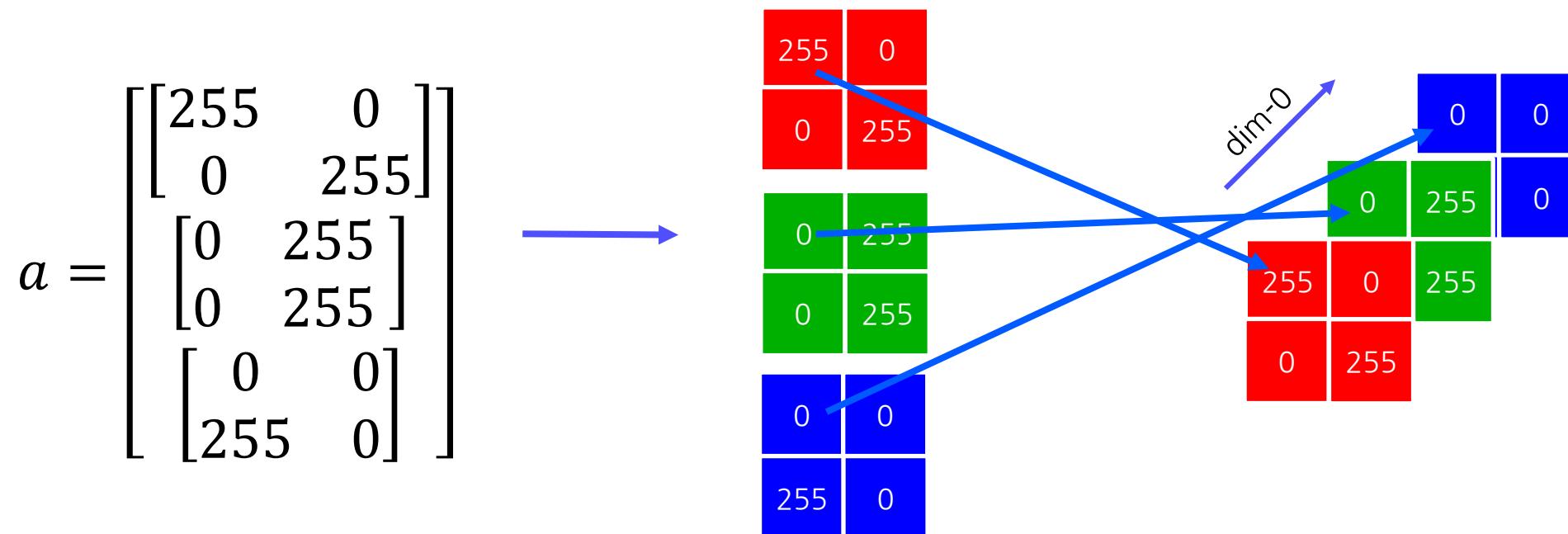
와 같이 결합하고자 하는 3개의 2-D Tensor를 생성함

2.7 stack()함수를 활용한 Tensor들 간의 결합

2. Tensor의 모양변경1

stack() 함수를 활용한 Tensor들 간의 결합에 대한 표현

- dim-0인 축을 생성하여 3개의 2D Tensor를 결합하는 코드 표현
 - `a = torch.stack((red_channel, green_channel, blue_channel))`



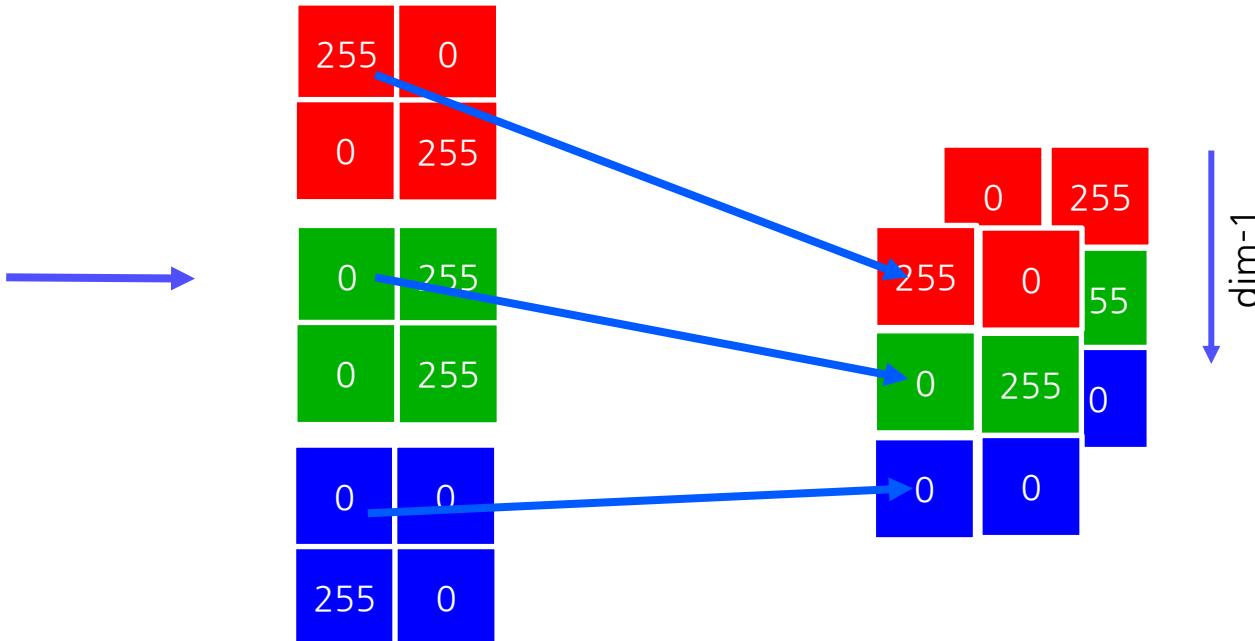
2.7 stack()함수를 활용한 Tensor들 간의 결합

2. Tensor의 모양변경1

stack() 함수를 활용한 Tensor들 간의 결합에 대한 표현

- dim-1인 축을 생성하여 3개의 2D Tensor를 결합하는 코드 표현
 - `a = torch.stack((red_channel, green_channel, blue_channel), dim = 1)`

$$a = \begin{bmatrix} [255 & 0] \\ [0 & 255] \\ [0 & 0] \\ [0 & 255] \\ [0 & 255] \\ [255 & 0] \end{bmatrix}$$

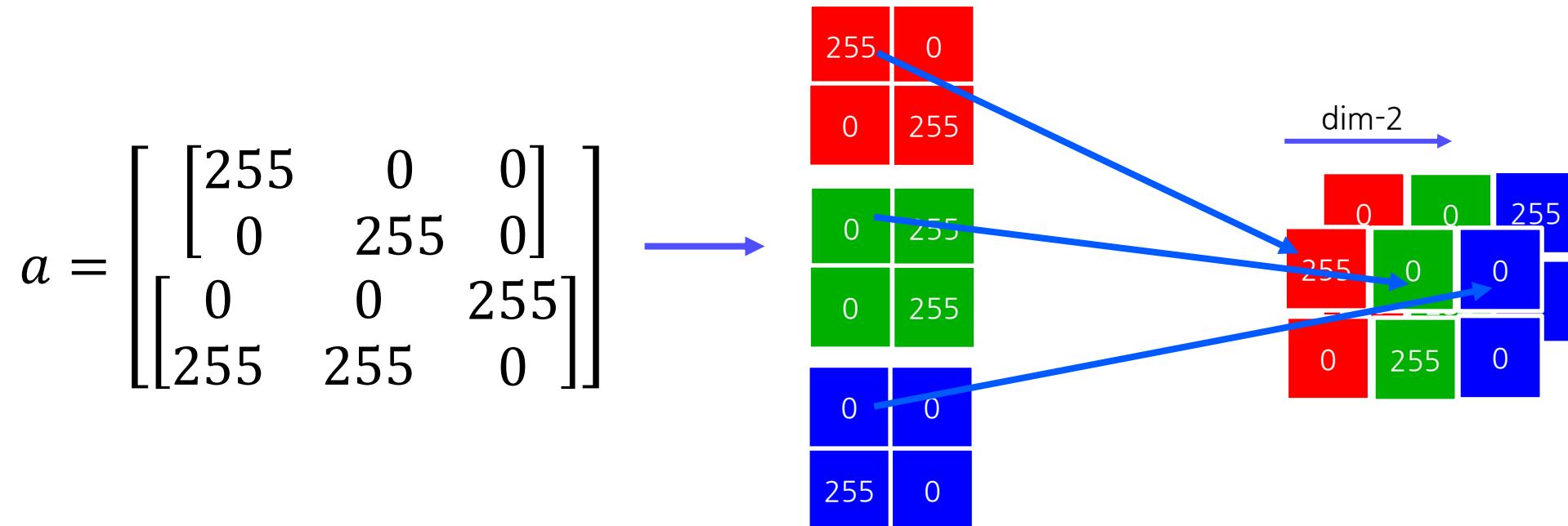


2.7 stack()함수를 활용한 Tensor들 간의 결합

2. Tensor의 모양변경1

stack() 함수를 활용한 Tensor들 간의 결합에 대한 표현

- dim-2인 축을 생성하여 3개의 2D Tensor를 결합하는 [코드 표현](#)
 - `a = torch.stack((red_channel, green_channel, blue_channel), dim = 2)`



Tensor의 모양변경1

- Tensor의 모양을 변경하는 메서드로 view()와 reshape()가 있다.
- Tensor를 평탄화하는 모양변경 함수로 flatten()이 있다.
- Tensor의 특정한 두 차원의 축을 서로 바꾸는 메서드로 transpose()가 있다.
- Tensor의 차원을 축소하는 함수로 squeeze(), 차원을 확장하는 함수로 unsqueeze()가 있다.
- Tensor들을 결합하는 함수로 stack()이 있다.

Thank you

Prof. Jeon, Sangryul

Computer Vision Lab.

Pusan National University, Korea

Tel: +82-51-510-2423

Web: <http://sr-jeon.github.io/>

E-mail: srjeonn@pusan.ac.kr