

# Linear Algebra

## From vector to Tensor (벡터에서 텐서로 - 텐서의 깊은 이해)

소프트웨어 끈대 강의

노기섭 교수

(kafa46@cju.ac.kr)

# 수학적 Tensor

Definition of Tensor (자료 출처: <https://en.Wikipedia.org/wiki/Tensor>)

선형대수학에서 다중선형사상(multilinear map) 또는 텐서(tensor)는 선형 관계를 나타내는 다중선형대수학의 대상이다.  
19세기에 가우스가 곡면에 대한 미분 기하학을 만들면서 도입하였다.  
기본적인 예는 내적과 선형 변환이 있으며 미분 기하학에서 자주 등장한다.  
Tensor는 기저를 선택하여 다차원 배열로 나타낼 수 있으며, ~~~



우리에겐 테러 수준  $\pi$

벡터 공간  $V$ 와 그 쌍대 공간  $V^*$ 에 대하여 음이 아닌 정수  $m, n$ 마다  $(m, n)$ 형의 텐서는 벡터 공간

$$T_n^m(V) = \underbrace{V \otimes \cdots \otimes V}_m \otimes \underbrace{V^* \otimes \cdots \otimes V^*}_n$$

그냥 스킵 할게요 ~~



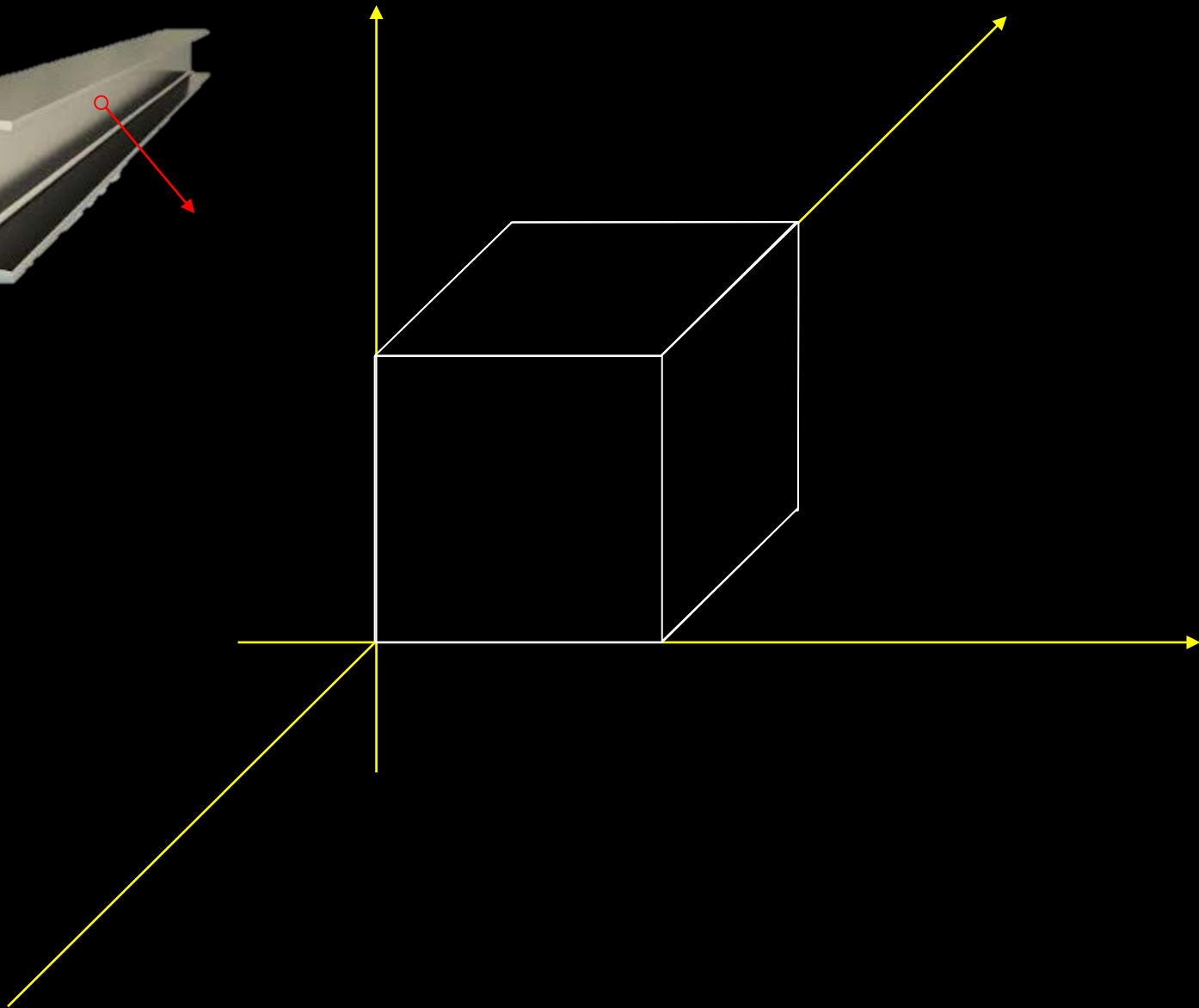
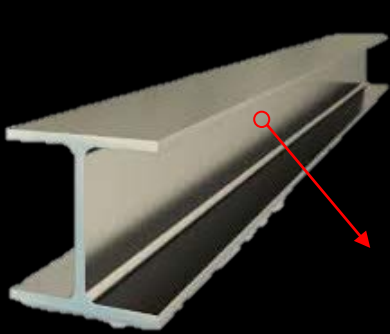
의 원소로 정의된다. 여기에서 **텐서곱**  $\otimes$ 은 **외적**의 일반화로 생각하여 대략

$$\begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix} \otimes \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix} = \begin{bmatrix} a_{1,1} \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix} & a_{1,2} \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix} \\ a_{2,1} \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix} & a_{2,2} \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} a_{1,1}b_{1,1} & a_{1,1}b_{1,2} & a_{1,2}b_{1,1} & a_{1,2}b_{1,2} \\ a_{1,1}b_{2,1} & a_{1,1}b_{2,2} & a_{1,2}b_{2,1} & a_{1,2}b_{2,2} \\ a_{2,1}b_{1,1} & a_{2,1}b_{1,2} & a_{2,2}b_{1,1} & a_{2,2}b_{1,2} \\ a_{2,1}b_{2,1} & a_{2,1}b_{2,2} & a_{2,2}b_{2,1} & a_{2,2}b_{2,2} \end{bmatrix}$$

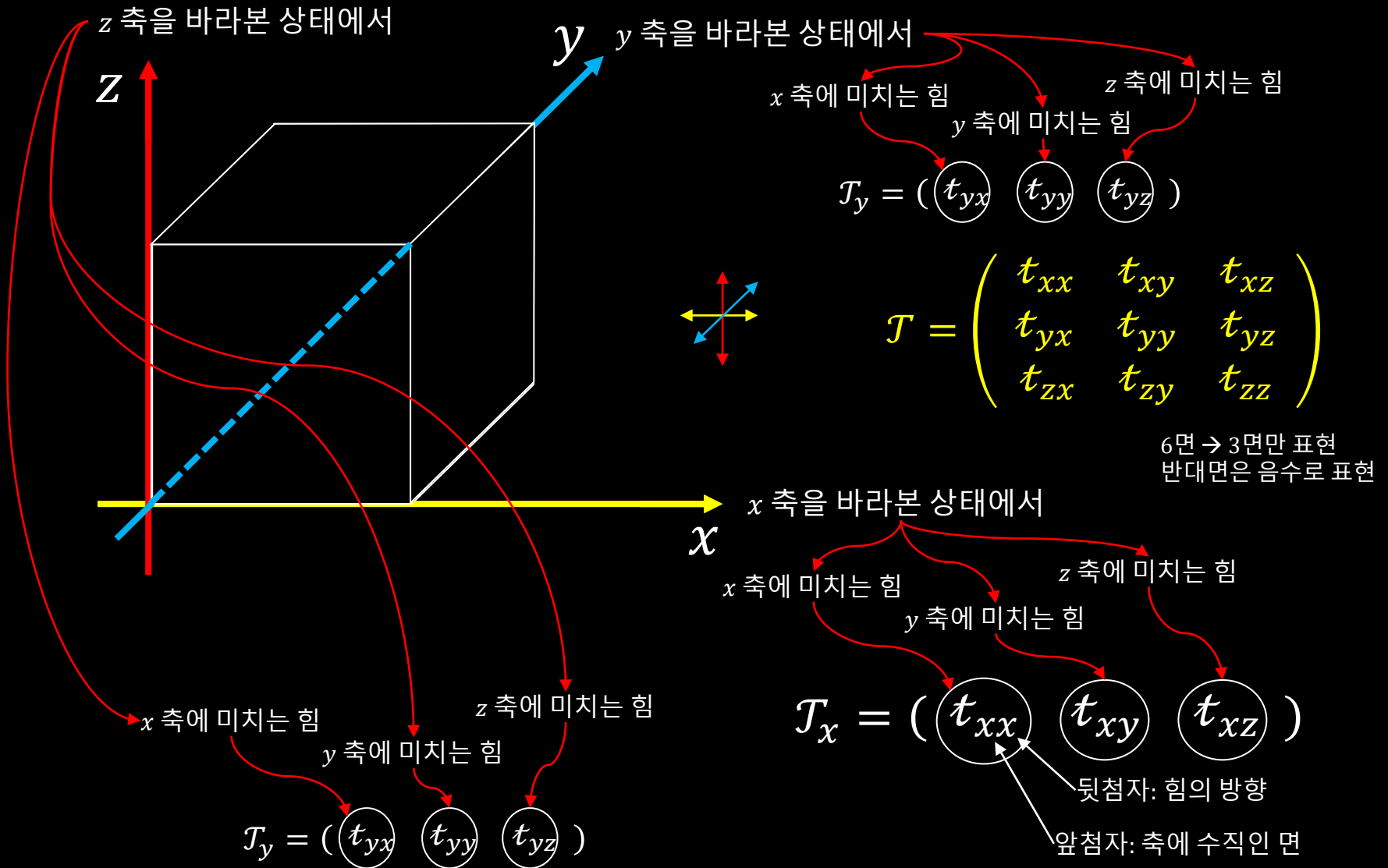
$$A_{ij} \otimes B_{kl} = T_{ijkl}$$

와 같은 연산이다.

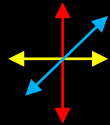
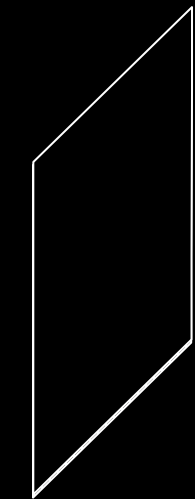
# 물리적 Tensor



# 물리적 Tensor



# 물리적 Tensor, 조금 더 확장하기



Rank:  $\mathcal{T}$ 의 원소들이 갖는 Basis 수

Rank 2

의미는 다르지만,  
어쨌든 Matrix와 같은 모양 ^^

$$\mathcal{T} = \begin{pmatrix} t_{xx} & t_{xy} & t_{xz} \\ t_{yx} & t_{yy} & t_{yz} \\ t_{zx} & t_{zy} & t_{zz} \end{pmatrix}$$

$\mathcal{T}_{ijk}$

$i$ : row index

$$\begin{pmatrix} t_{xxx} & t_{xyx} & t_{xzx} \\ t_{yxx} & t_{yyx} & t_{yzx} \\ t_{zxx} & t_{zyx} & t_{zzx} \end{pmatrix} \begin{pmatrix} t_{xxz} & t_{xyz} & t_{xzz} \\ t_{yyz} & t_{yzz} & t_{zzz} \\ t_{zyz} & t_{zzy} & t_{zzz} \end{pmatrix}$$

$j$ : column index

$k$ : object index

$$\begin{pmatrix} t_{xxz} & t_{xyz} & t_{xzz} \\ t_{yyz} & t_{yzz} & t_{zzz} \\ t_{zyz} & t_{zzy} & t_{zzz} \end{pmatrix}$$

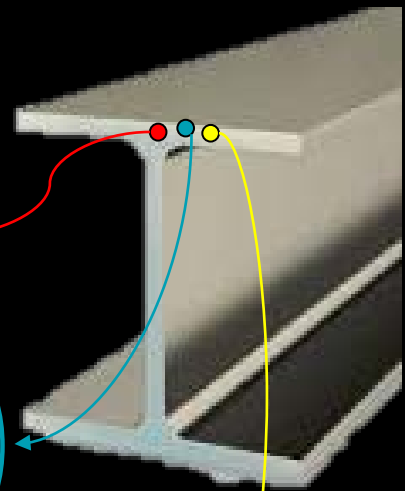
$$\begin{pmatrix} t_{xxy} & t_{xyy} & t_{xzy} \\ t_{yxy} & t_{yyy} & t_{yzy} \\ t_{zxy} & t_{zyy} & t_{zzy} \end{pmatrix}$$

$$\begin{pmatrix} t_{xxx} & t_{xyx} & t_{xzx} \\ t_{yxx} & t_{yyx} & t_{yzx} \\ t_{zxx} & t_{zyx} & t_{zzx} \end{pmatrix}$$

의미는 다르지만, 어쨌든! Matrix를  
여러 개 표현할 수 있는 모양 ^^

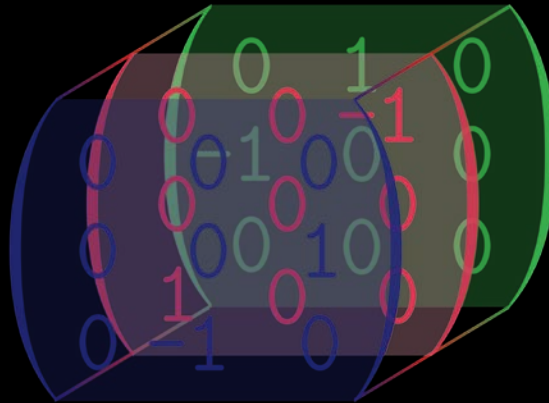


우리가 찾던 것!!!



아하!! 이러면 되겠구나!!

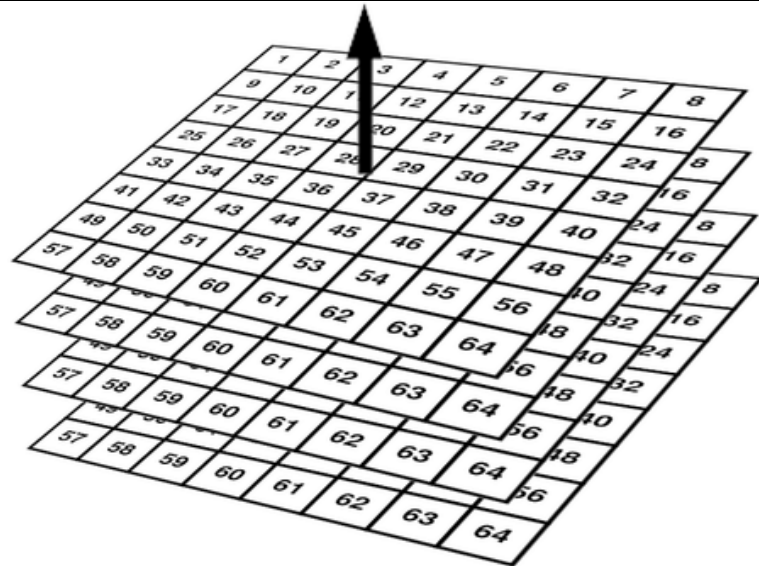
$$T_{ijk} =$$



아하!

이제는 matrix 보다 더 편리한  
자료구조로 Tensor를 쓰면 되겠구나!!

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64



# Deep learning Tensor

사실 Tensor를 정확히 이해하는 것은 매우 어렵습니다.

간단하게 생각하면 선형대수 공부하면서 다루었던  
Scalar, Vector, Matrix를 포함하는 포괄적인 데이터  
표현이라고 생각할 수 있습니다.

우리는 딥러닝 학습/추론을 위한 다차원 데이터 자료구조가 필요할 뿐입니다.




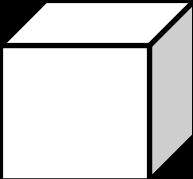
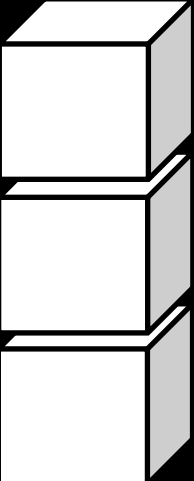
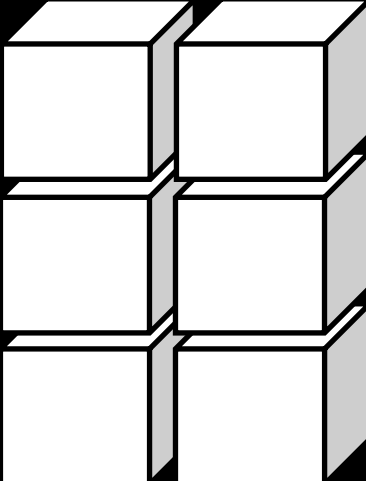
우리는 그냥 행렬을 여러 개 담아 놓은 배열이라고 생각하겠습니다.

물리적 특성은 일단 무시합니다.

Python에서 제공하는 List, Tuple, Set, Dictionary와 같은  
자료구조 중 하나라고 생각합니다.

# Tensor 간단히 이해하기 → 컴싸, 컴공 버전!!

원소(entry)를 표현하기 위해 필요한 기저(basis)의 수

Rank (차원)	0	1	2	3	4	5	...
Name (명칭)	scalar	vector	matrix	3D tensor	4D tensor	5D tensor	...
Visualization (시각화)							...

우리가 배운 내용

더 필요한 자료구조



# Tensor Operation in Deeplearning

사실 Tensor는 다차원 배열이기 때문에 수많은 연산이 가능합니다.

어느 차원, 어느 축을 기준으로 하느냐에 따라 달라집니다 ^^.

결국 Matrix, Vector, Scalar 연산이 가능하도록 쪼개야 합니다.

Tensor 쪼개는 방법에 대해 알아보겠습니다 ^^.

다음 슬라이드로 이동~~

# Tensor 쪼개기

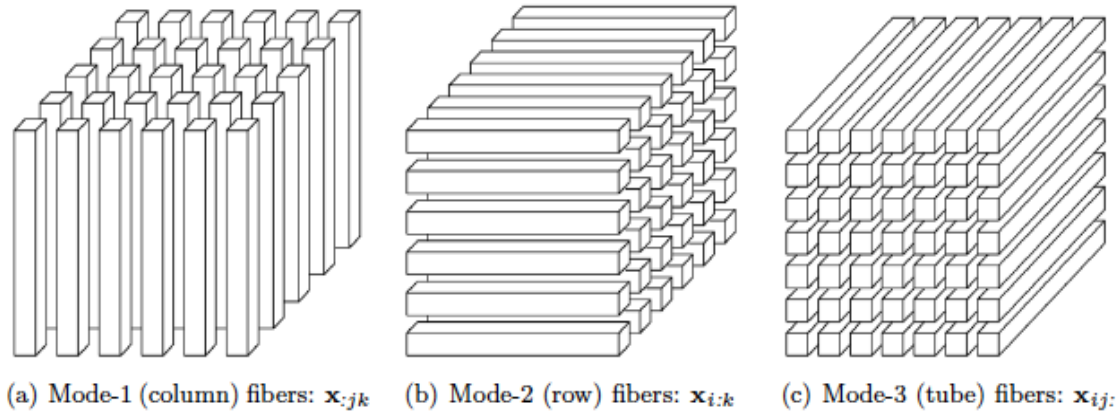


Fig. 2.1 Fibers of a 3rd-order tensor.

## Fiber로 쪼개기

1개의 인덱스는 자유롭게  
나머지 인덱스는 모두 고정  
설정에 따라 다양한  
Vector가 생성됨

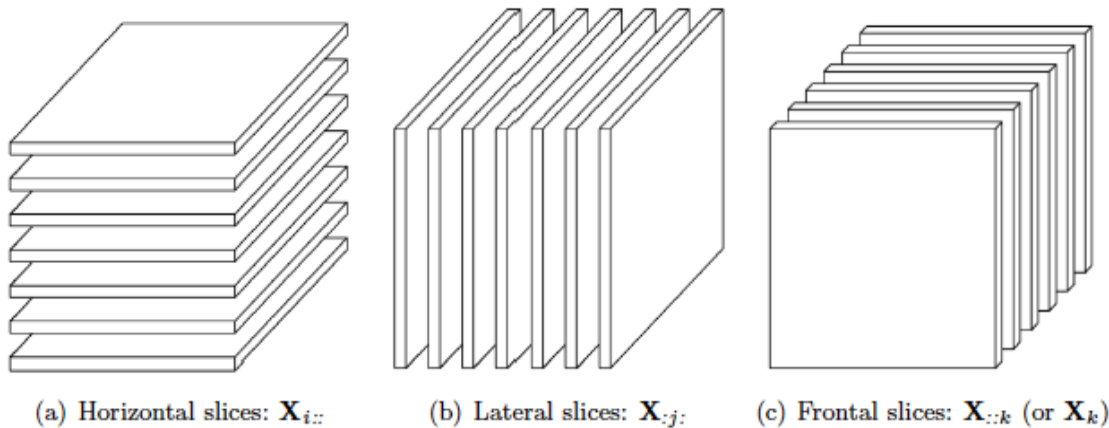


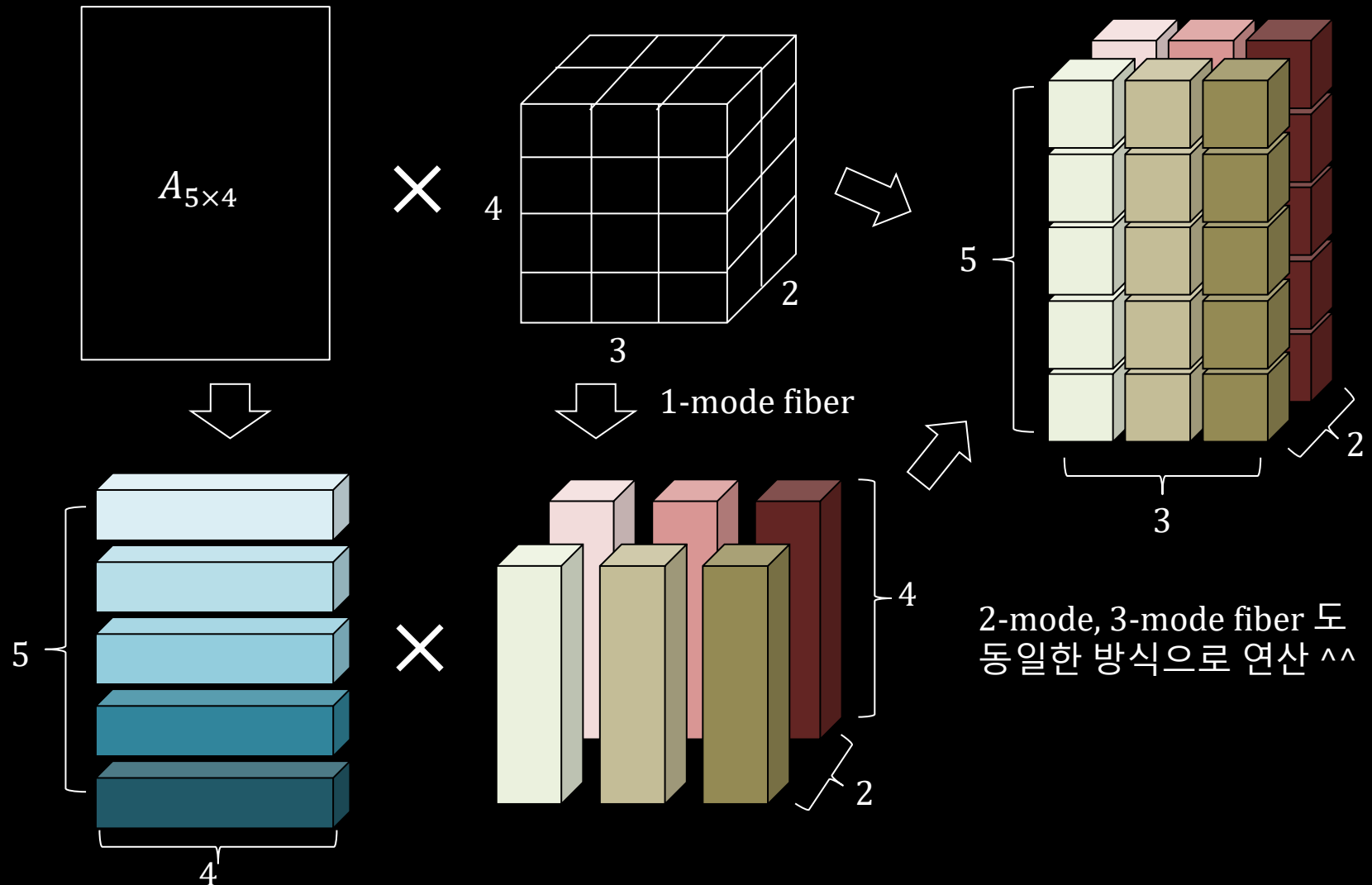
Fig. 2.2 Slices of a 3rd-order tensor.

## Slice로 쪼개기

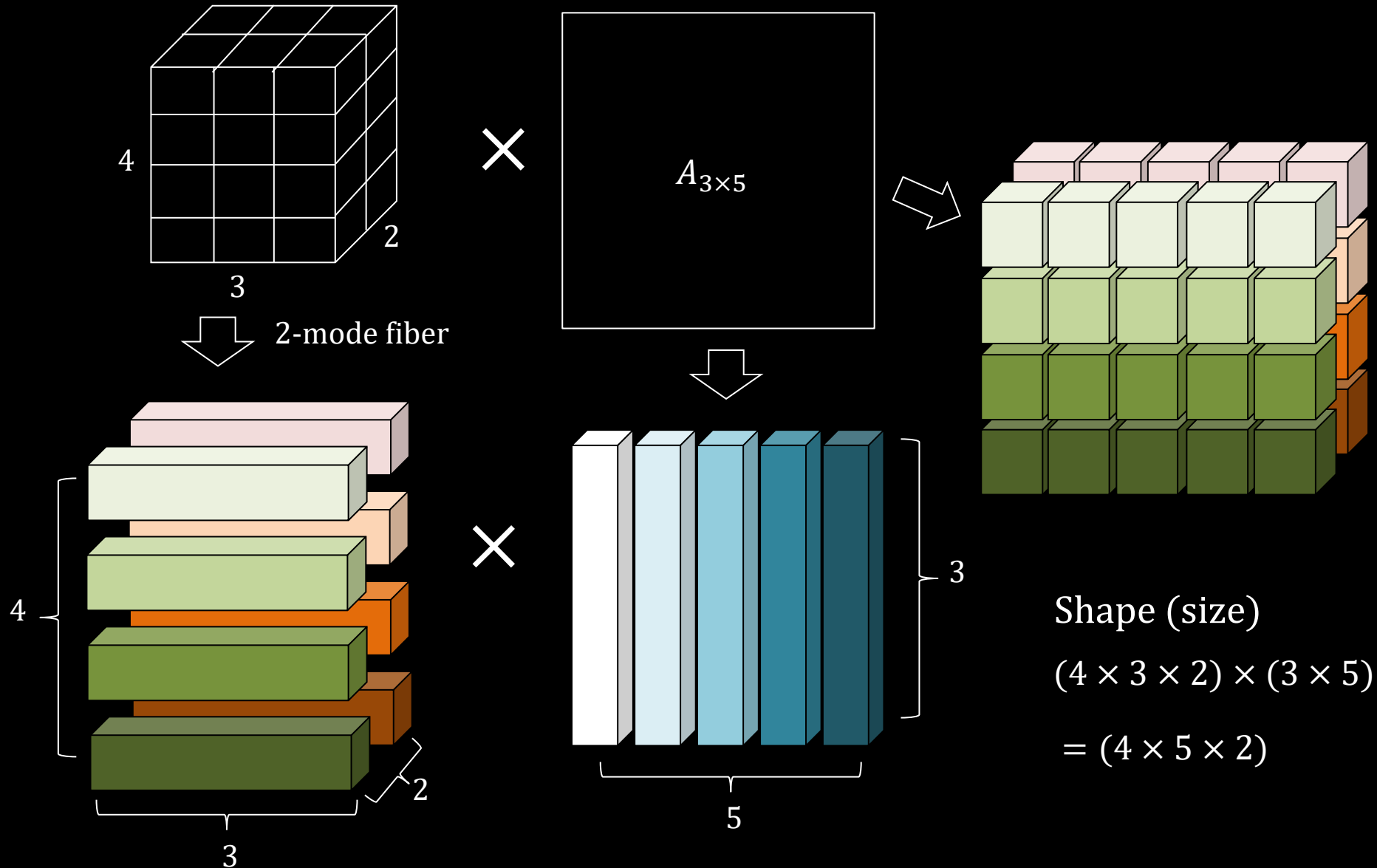
2개의 인덱스는 자유롭게  
나머지 인덱스는 모두 고정  
설정에 따라 다양한  
Matrix가 생성됨

이밖에 다양한 방법도 가능

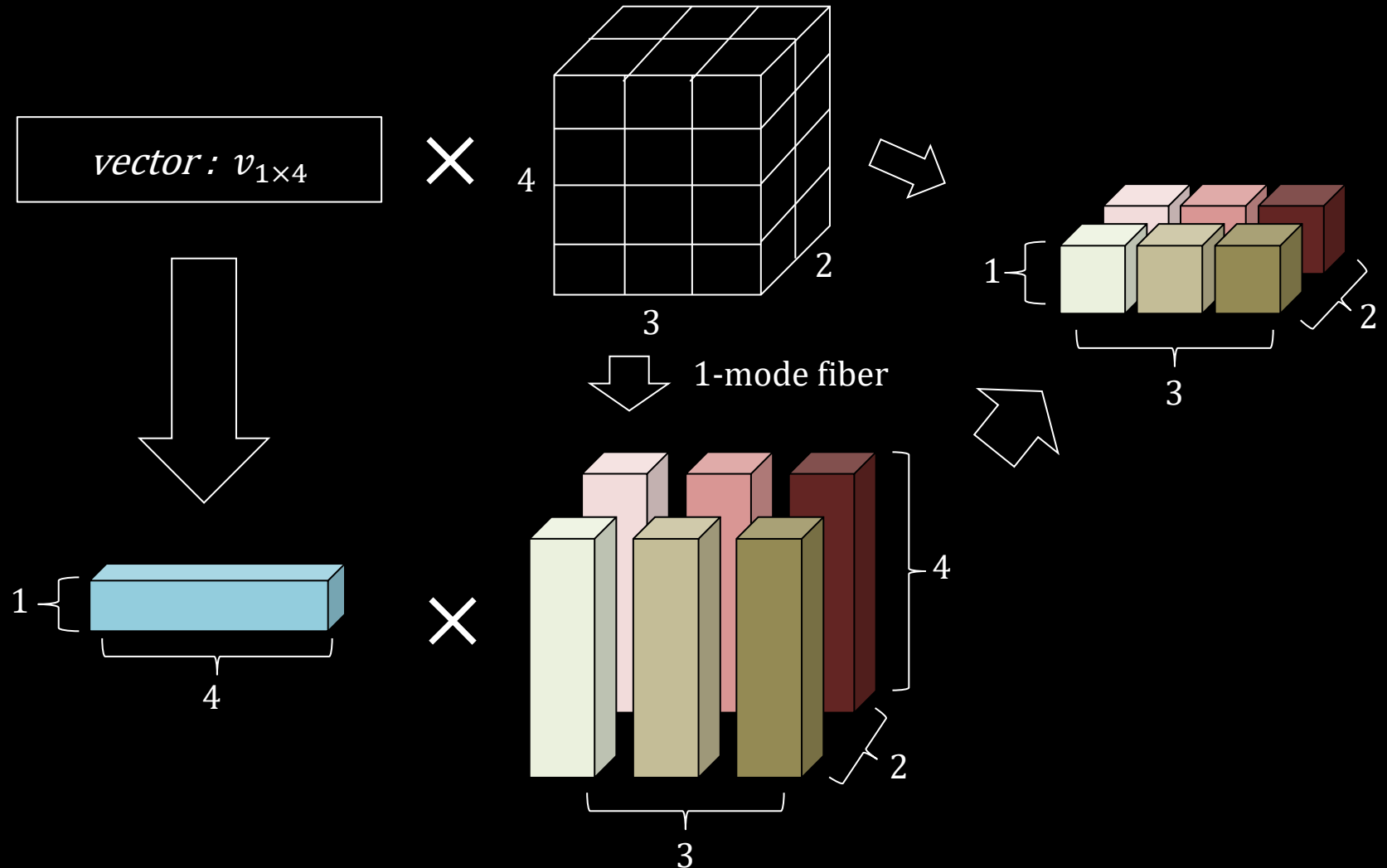
# Operation in Deeplearning: $Matrix \times Tensor$ (1-mode fiber)



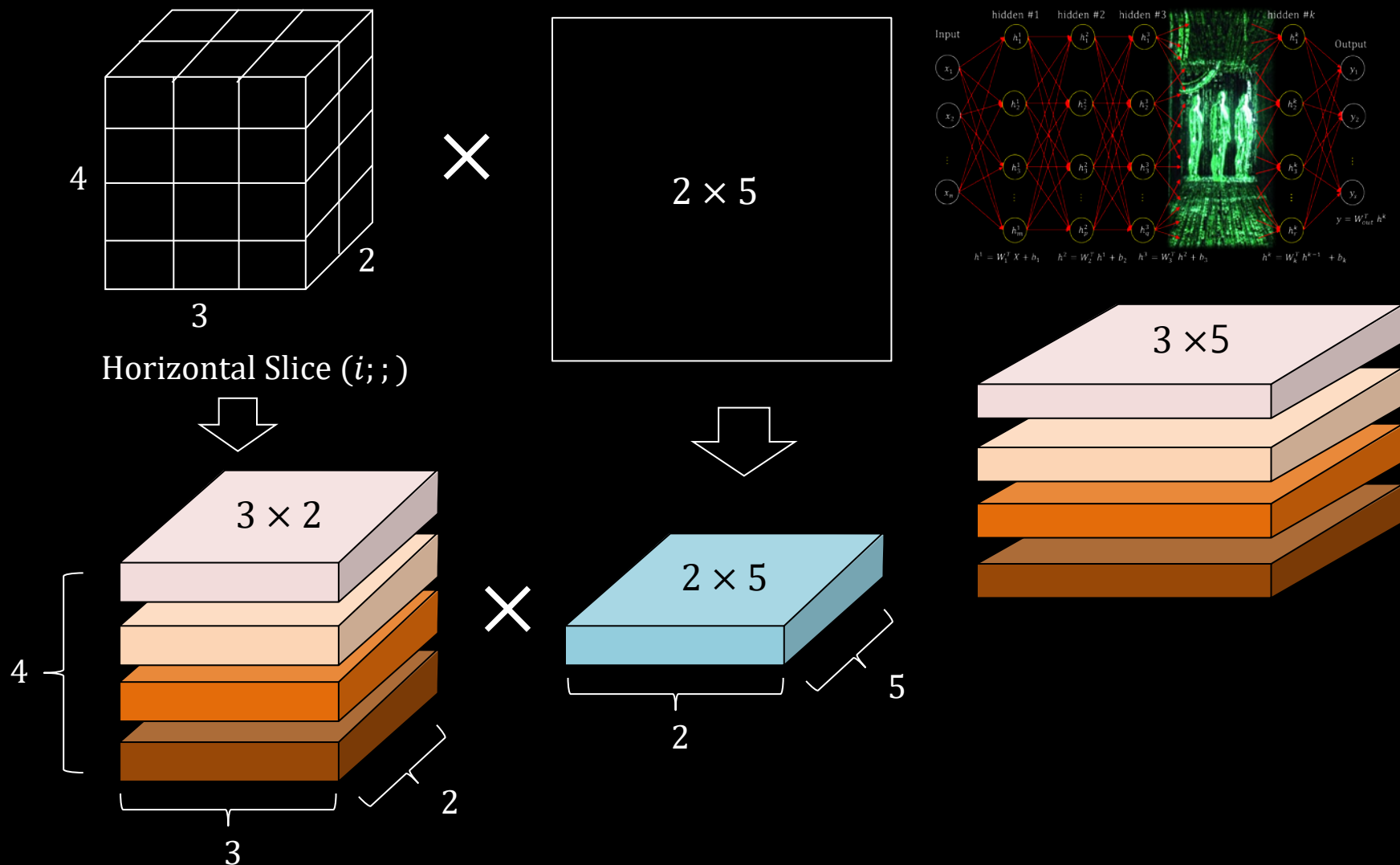
# Operation in Deeplearning: *Tensor* (2-mode fiber) $\times$ *Matrix*



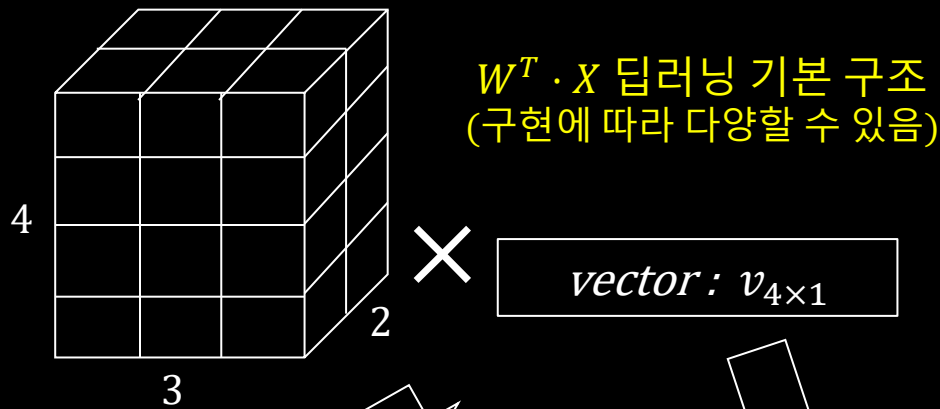
# Operation in Deeplearning: $Vector \times Tensor$ (1-mode fiber)



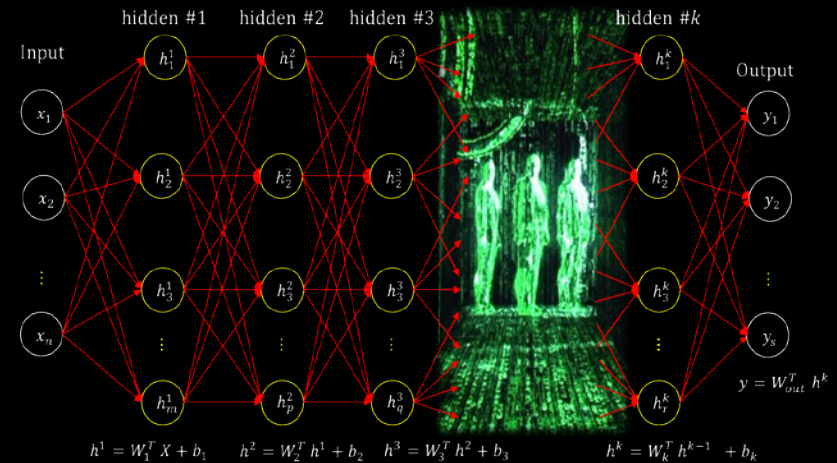
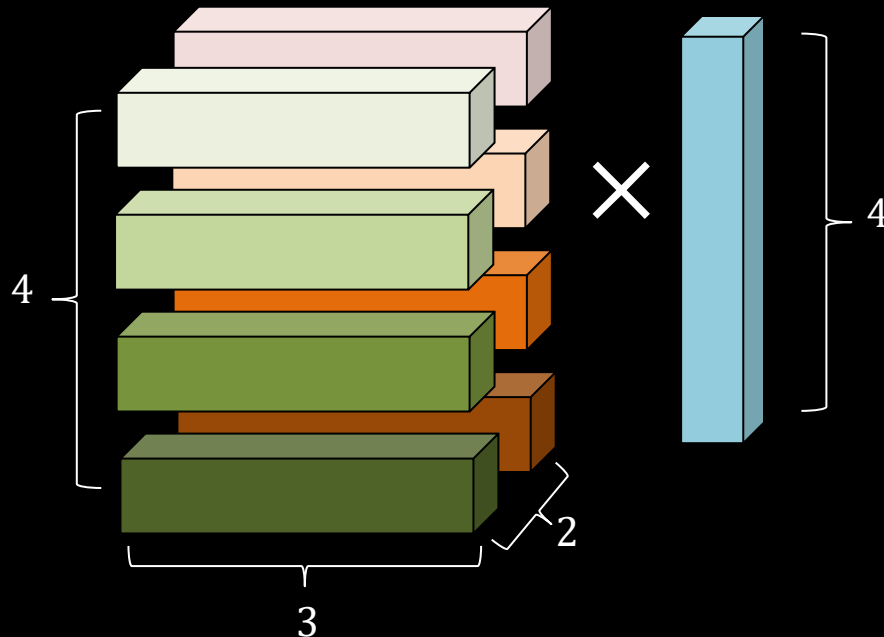
# Operation in Deeplearning: *Tensor (slice) × Matrix*



# Operation in Deeplearning: $Tensor(2\text{-mode fiber}) \times Vector$



2-mode fiber



# Framework: Pytorch 구현 맛보기

Official page: <https://pytorch.org/docs/stable/generated/torch.matmul.html>

```
torch.matmul(input, other, *, out=None) → Tensor
```

Matrix product of two tensors.

The behavior depends on the dimensionality of the tensors as follows:

- If both tensors are 1-dimensional, the dot product (scalar) is returned.
- If both arguments are 2-dimensional, the matrix-matrix product is returned.
- If the first argument is 1-dimensional and the second argument is 2-dimensional, a 1 is prepended to its dimension for the purpose of the matrix multiply. After the matrix multiply, the prepended dimension is removed.
- If the first argument is 2-dimensional and the second argument is 1-dimensional, the matrix-vector product is returned.
- If both arguments are at least 1-dimensional and at least one argument is N-dimensional (where  $N > 2$ ), then a batched matrix multiply is returned. If the first argument is 1-dimensional, a 1 is prepended to its dimension for the purpose of the batched matrix multiply and removed after. If the second argument is 1-dimensional, a 1 is appended to its dimension for the purpose of the batched matrix multiply and removed after. The non-matrix (i.e. batch) dimensions are **broadcasted** (and thus must be broadcastable). For example, if `input` is a  $(j \times 1 \times n \times n)$  tensor and `other` is a  $(k \times n \times n)$  tensor, `out` will be a  $(j \times k \times n \times n)$  tensor.

Note that the broadcasting logic only looks at the batch dimensions when determining if the inputs are broadcastable, and not the matrix dimensions. For example, if `input` is a  $(j \times 1 \times n \times m)$  tensor and `other` is a  $(k \times m \times p)$  tensor, these inputs are valid for broadcasting even though the final two dimensions (i.e. the matrix dimensions) are different. `out` will be a  $(j \times k \times n \times p)$  tensor.

This operation has support for arguments with **sparse layouts**. In particular the matrix-matrix (both arguments 2-dimensional) supports sparse arguments with the same restrictions as `torch.mm()`

```
>>> # vector x vector
>>> tensor1 = torch.randn(3)
>>> tensor2 = torch.randn(3)
>>> torch.matmul(tensor1, tensor2).size()
torch.Size([1])
>>> # matrix x vector
>>> tensor1 = torch.randn(3, 4)
>>> tensor2 = torch.randn(4)
>>> torch.matmul(tensor1, tensor2).size()
torch.Size([3])
>>> # batched matrix x broadcasted vector
>>> tensor1 = torch.randn(10, 3, 4)
>>> tensor2 = torch.randn(4)
>>> torch.matmul(tensor1, tensor2).size()
torch.Size([10, 3])
>>> # batched matrix x batched matrix
>>> tensor1 = torch.randn(10, 3, 4)
>>> tensor2 = torch.randn(10, 4, 5)
>>> torch.matmul(tensor1, tensor2).size()
torch.Size([10, 3, 5])
>>> # batched matrix x broadcasted matrix
>>> tensor1 = torch.randn(10, 3, 4)
>>> tensor2 = torch.randn(4, 5)
>>> torch.matmul(tensor1, tensor2).size()
torch.Size([10, 3, 5])
```



# Pytorch: vector $\times$ vector

```
>>> # vector x vector  
>>> tensor1 = torch.randn(3)  
>>> tensor2 = torch.randn(3)  
>>> torch.matmul(tensor1, tensor2).size()  
torch.Size([1])
```

## Pytorch: matrix $\times$ vector

```
>>> # matrix x vector
>>> tensor1 = torch.randn(3, 4)
>>> tensor2 = torch.randn(4)
>>> torch.matmul(tensor1, tensor2).size()
```

## Pytorch: tensor(batched matrix) $\times$ vector

```
>>> # batched matrix x broadcasted vector  
>>> tensor1 = torch.randn(10, 3, 4)  
>>> tensor2 = torch.randn(4)  
>>> torch.matmul(tensor1, tensor2).size()
```

## Pytorch: tensor(batched matrix) $\times$ tensor(batched matrix)

```
>>> # batched matrix x batched matrix
>>> tensor1 = torch.randn(10, 3, 4)
>>> tensor2 = torch.randn(10, 4, 5)
>>> torch.matmul(tensor1, tensor2).size()
torch.Size([10, 3, 5])
```

## Pytorch: tensor(batched matrix) × tensor(broadcasted matrix)

```
>>> # batched matrix x broadcasted matrix  
>>> tensor1 = torch.randn(10, 3, 4)  
>>> tensor2 = torch.randn(4, 5)  
>>> torch.matmul(tensor1, tensor2).size()  
torch.Size([10, 3, 5])
```



수고하셨습니다 ..^^..