# Tensors (with PyTorch)

Sept. 2023

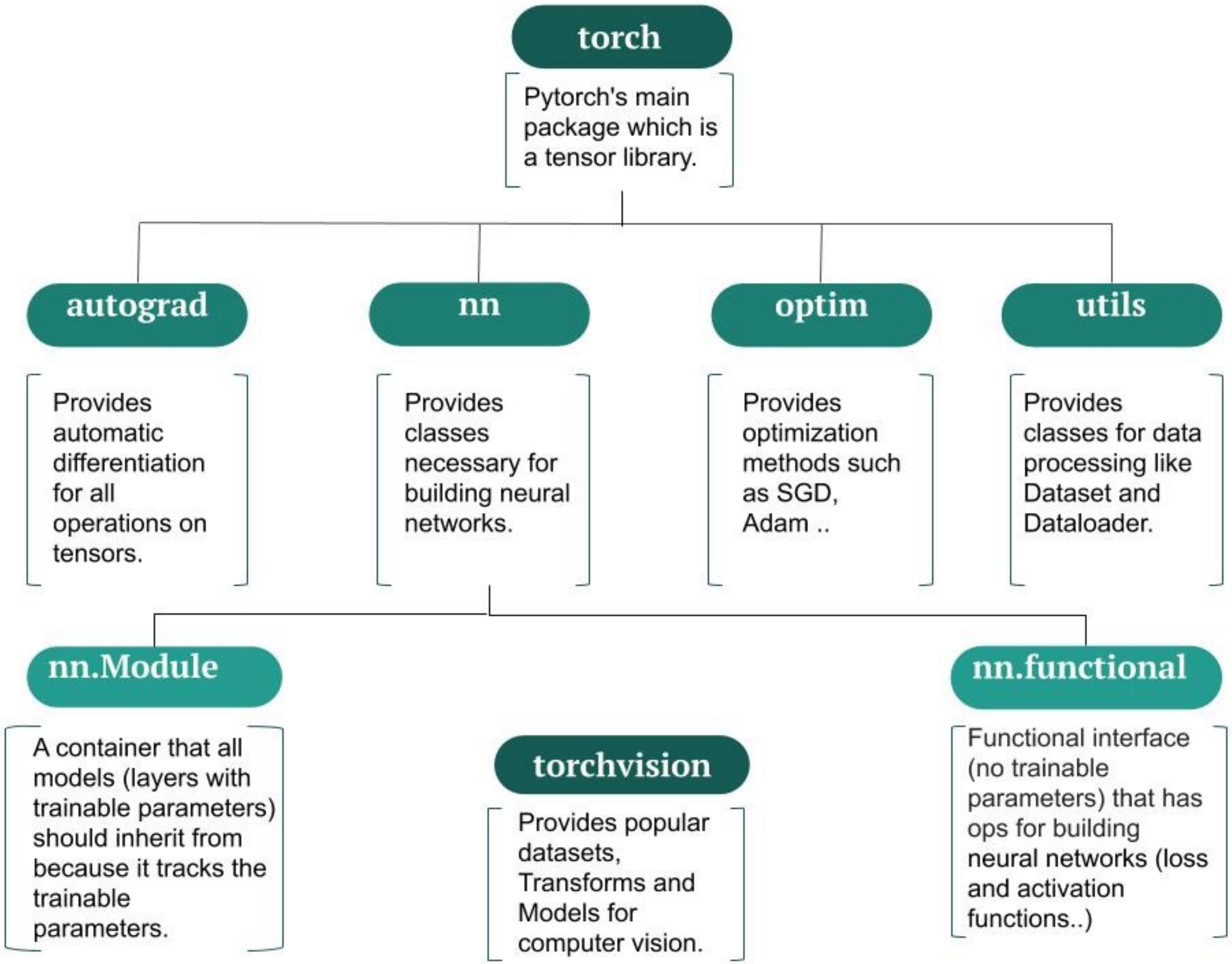
http://link.koreatech.ac.kr

# PyTorch and Tensors 101

# PyTorch Packages

## ◈PyTorch Packages

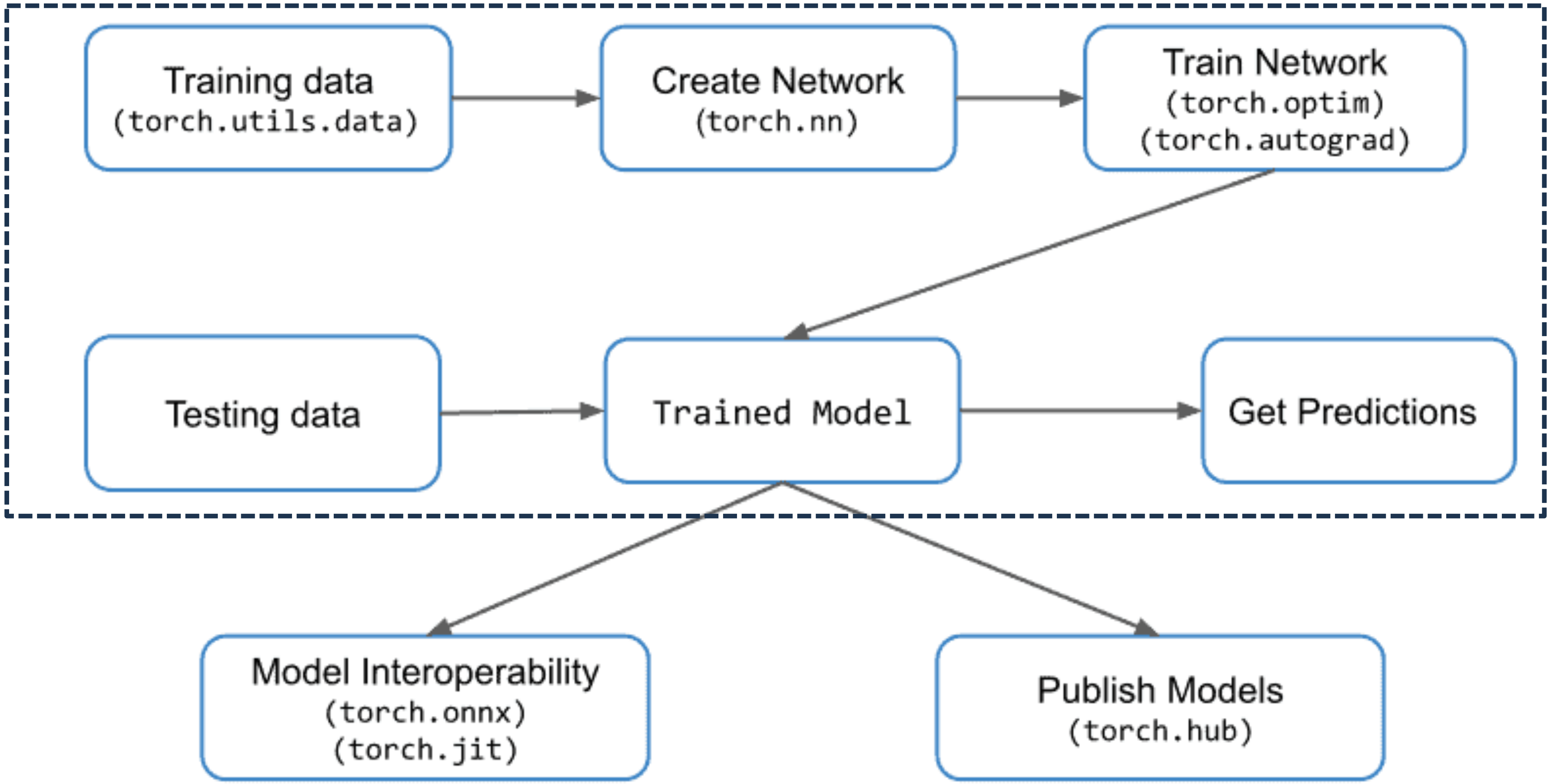
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
```




[Source: https://manalelaidouni.github.io/Pytorch%20guide%20101.html]

# PyTorch Packages

## ◈Basic PyTorch Workflow



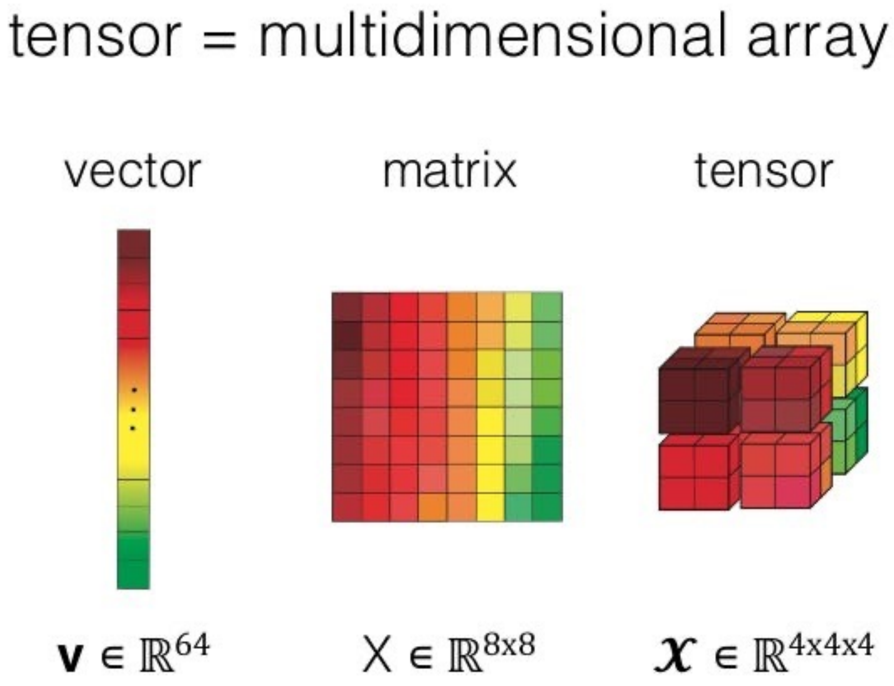

The scope of our class

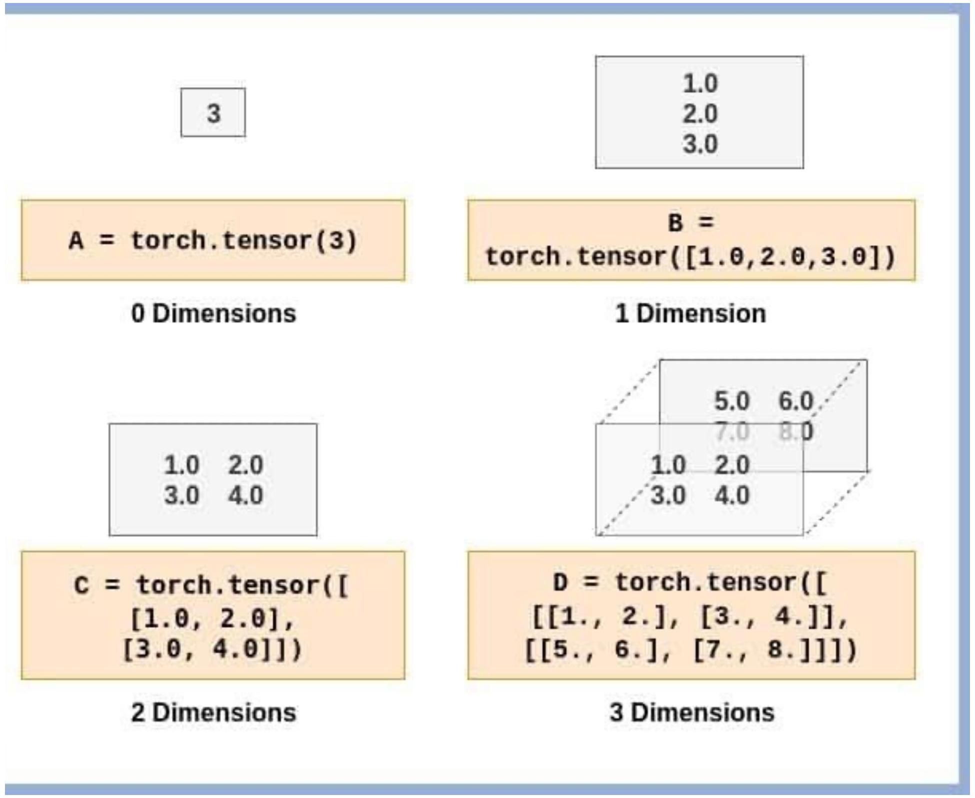
[Source: https://learnopencv.com/pytorch-for-beginners-basics/]

# Tensor

## ◈Tensors

– Multi-dimensional array containing elements of a single data type
  - PyTorch tensors are very similar to NumPy ndarrays
  - PyTorch tensors are used on a GPU as well (this is not the case with NumPy arrays)



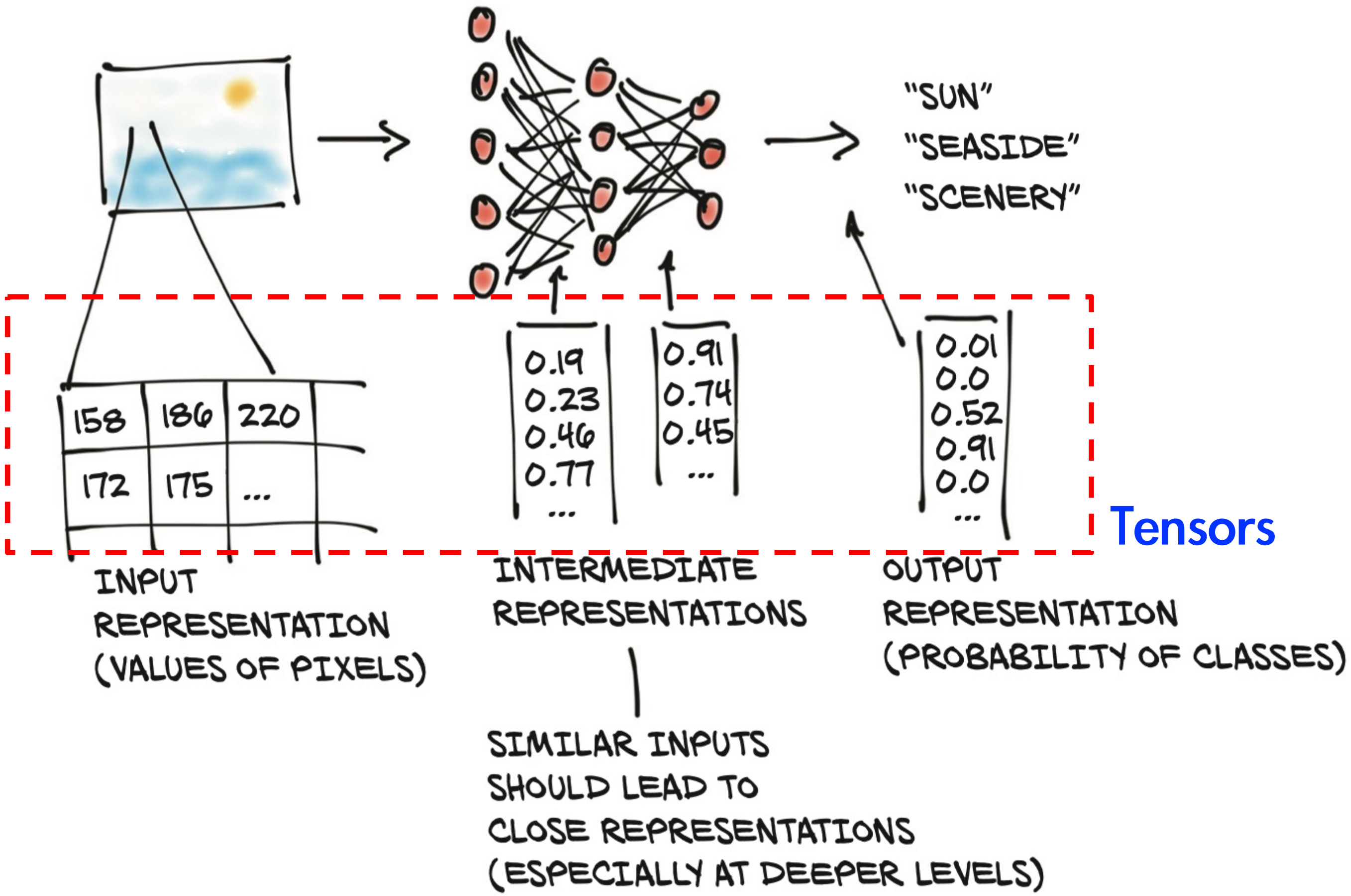




[Source: https://learnopencv.com/pytorch-for-beginners-basics/]

# Tensor

## ◈Tensors

– A deep neural network learns how to transform **an input representation (or input tensor)** to **an output representation (or output tensor)**



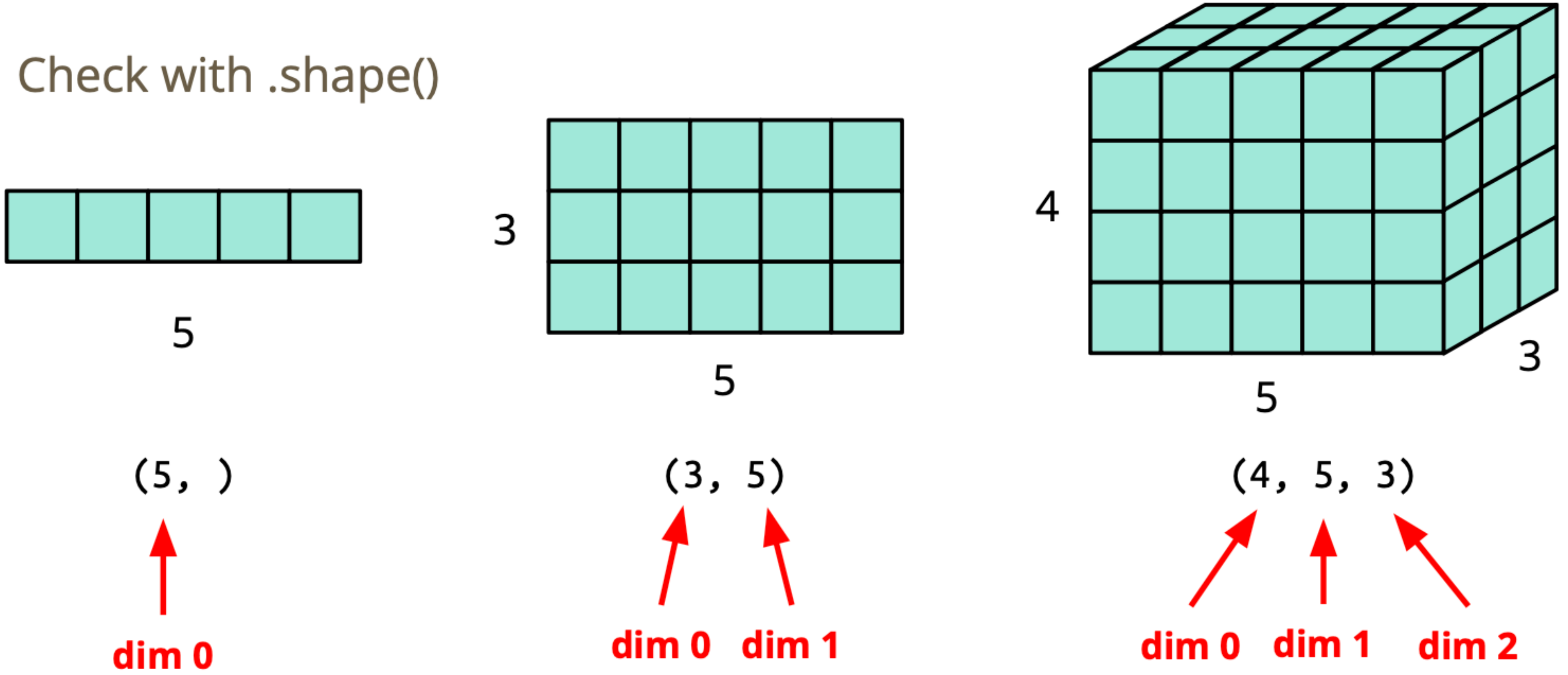

[Source: https://www.manning.com/books/deep-learning-with-python]

# Tensor

## ◈ Shape of Tensors

– check with t.shape or t.size()

- Check with .shape()

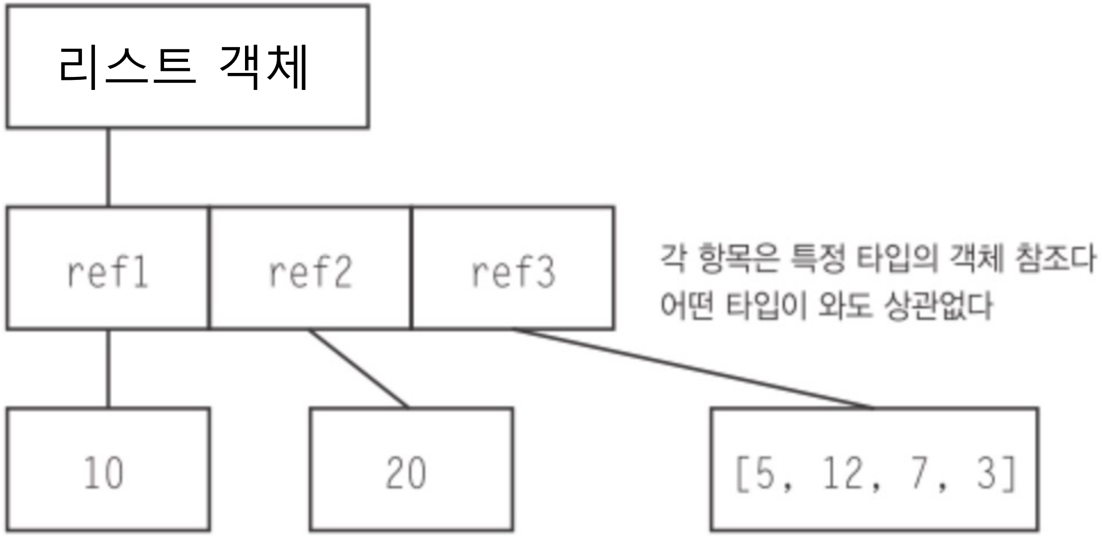



Note: `dim` in PyTorch == `axis` in NumPy

# Tensor

## ◈Tensor Storage

**파이썬 list 객체 저장 방식**




**파이썬 Numpy & PyTorch Tensor 객체 저장 방식**

# Tensor Initialization

# Tensor Initialization

## ◇Tensor Data Types

- Almost always torch.float32 or torch.int64 are used
- torch.Tensor is an alias for the default tensor type torch.FloatTensor

| Data type | dtype | CPU tensor | GPU tensor |
|---|---|---|---|
| 32-bit floating point | `torch.float32` or `torch.float` | `torch.FloatTensor` | `torch.cuda.FloatTensor` |
| 64-bit floating point | `torch.float64` or `torch.double` | `torch.DoubleTensor` | `torch.cuda.DoubleTensor` |
| 16-bit floating point | `torch.float16` or `torch.half` | `torch.HalfTensor` | `torch.cuda.HalfTensor` |
| 8-bit integer (unsigned) | `torch.uint8` | `torch.ByteTensor` | `torch.cuda.ByteTensor` |
| 8-bit integer (signed) | `torch.int8` | `torch.CharTensor` | `torch.cuda.CharTensor` |
| 16-bit integer (signed) | `torch.int16` or `torch.short` | `torch.ShortTensor` | `torch.cuda.ShortTensor` |
| 32-bit integer (signed) | `torch.int32` or `torch.int` | `torch.IntTensor` | `torch.cuda.IntTensor` |
| 64-bit integer (signed) | `torch.int64` or `torch.long` | `torch.LongTensor` | `torch.cuda.LongTensor` |
| Boolean | `torch.bool` | `torch.BoolTensor` | `torch.cuda.BoolTensor` |

link_dl/_01_code/_02_tensors/a_tensor_initialization.py

# Tensor Initialization

## ◈Tensor initialization directly from data

### – torch.Tensor class & torch.tensor function

- Every tensor has device, dtype, shape, and required_grad attributes
- torch.Tensor is an alias for the default tensor type torch.FloatTensor

```python
# torch.Tensor class
t1 = torch.Tensor([1, 2, 3], device='cpu')
print(t1.dtype)  # >>> torch.float32
print(t1.device) # >>> cpu
print(t1.requires_grad) # >>> False
print(t1.size()) # torch.Size([3])
print(t1.shape)  # torch.Size([3])

# if you have gpu device
t1_cuda = t1.to(torch.device('cuda'))
# or you can use shorthand
t1_cuda = t1.cuda()

t1_cpu = t1.cpu()
```

```python
# torch.tensor function
t2 = torch.tensor([1, 2, 3], device='cpu')
print(t2.dtype)  # >>> torch.int64
print(t2.device) # >>> cpu
print(t2.requires_grad) # >>> False
print(t2.size()) # torch.Size([3])
print(t2.shape)  # torch.Size([3])

# if you have gpu device
t2_cuda = t2.to(torch.device('cuda'))
# or you can use shorthand
t2_cuda = t2.cuda()

t2_cpu = t2.cpu()
```

# Tensor Initialization

## ◈Tensor initialization directly from data

– torch.Tensor & torch.tensor always copies the given data

```
import torch

l1 = [1, 2, 3]
t1 = torch.Tensor(l1)

l2 = [1, 2, 3]
t2 = torch.tensor(l2)

l3 = [1, 2, 3]
t3 = torch.as_tensor(l3)

l1[0] = 100
l2[0] = 100
l3[0] = 100

print(t1) # >>> tensor([1., 2., 3.])
print(t2) # >>> tensor([1, 2, 3])
print(t3) # >>> tensor([1, 2, 3])
```

```
import torch
import numpy as np
l4 = np.array([1, 2, 3])
t4 = torch.Tensor(l4)

l5 = np.array([1, 2, 3])
t5 = torch.tensor(l5)

l6 = np.array([1, 2, 3])
t6 = torch.as_tensor(l6)

l4[0] = 100
l5[0] = 100
l6[0] = 100

print(t4) # >>> tensor([1., 2., 3.])
print(t5) # >>> tensor([1, 2, 3])
print(t6) # >>> tensor([100, 2, 3])
```

If you have a numpy array and want to avoid a copy, use torch.as_tensor().

# Tensor Initialization

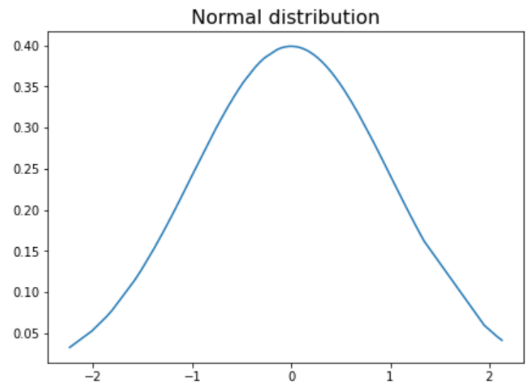
- Tensor initialization with constant values
  - torch.zeros(*size)
  - torch.zeros_like(input_tensor)
  - torch.ones(*size)
  - torch.ones_like(input_tensor)
  - torch.empty(*size)
    - filled with uninitialized data
  - torch.eye(n)
    - return identity matrix with size n x n

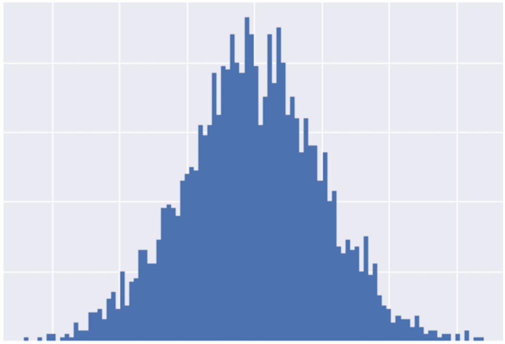
```
import torch

t1 = torch.ones(size=(5,))   # or torch.ones(5)
t1_like = torch.ones_like(input=t1)
print(t1)        # >>> tensor([1., 1., 1., 1., 1.])
print(t1_like)   # >>> tensor([1., 1., 1., 1., 1.])

t2 = torch.zeros(size=(6,))  # or torch.zeros(6)
t2_like = torch.zeros_like(input=t2)
print(t2)        # >>> tensor([0., 0., 0., 0., 0., 0.])
print(t2_like)   # >>> tensor([0., 0., 0., 0., 0., 0.])

t3 = torch.empty(size=(4,))  # or torch.zeros(4)
t3_like = torch.empty_like(input=t3)
print(t3)        # >>> tensor([0., 0., 0., 0.])
print(t3_like)   # >>> tensor([0., 0., 0., 0.])

t4 = torch.eye(n=3)
print(t4)         # >>> tensor([[1., 0., 0.],
                  #             [0., 1., 0.],
                  #             [0., 0., 1.]])
```

# Tensor Initialization

## ◈Tensor initialization with random values (1/2)

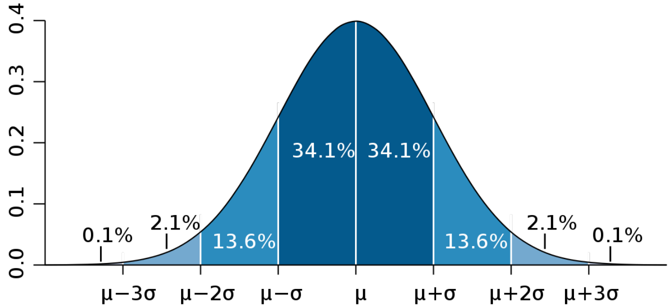
- torch.randint(low=0, high, size, ...)
  - filled with integer values that generate uniformly between low (inclusive) and high (exclusive)
- torch.rand(*size, ...)
  - filled with float values ranging between 0 and 1 from a uniform distribution
- torch.randn(*size, ...)
  - filled with a random float number from a standard normal distribution with a mean 0 and a variance of 1



```
import torch

t1 = torch.randint(low=10, high=20, size=(1, 2))
t2 = torch.rand(size=(1, 3))  # or torch.rand(1, 3)
t3 = torch.randn(size=(1, 3))  # or torch.randn(1, 3)
print(t1)  # >>> tensor([[11, 15]])
print(t2)  # >>> tensor([[0.3704, 0.3847, 0.2096]])
print(t3)  # >>> tensor([[-1.1459, -0.4099, -0.6727]])
```

# Tensor Initialization




## ◈Tensor initialization with random values (2/2)

– torch.normal(mean, std, size, ...)
  - filled with a random float number from a normal distribution whose mean and standard deviation are given

torch.linspace(3, 4, steps=7)


– torch.linspace(start, end, steps, ...)
  - returns a one-dimensional tensor between start and end and it is equally spaced based on steps

– torch.arange(start=0, end, steps=1,...)
  - returns a one-dimensional tensor with values from the interval [start, end) (start: inclusive, end: exclusive) taken with common difference step beginning from start

```
import torch

t4 = torch.normal(mean=10.0, std=1.0, size=(3, 2))
t5 = torch.linspace(start=0.0, end=5.0, steps=3)
t6 = torch.arange(5)
print(t4)  # >>> tensor([[11.4634, 12.1609],
           #             [ 9.8654,  9.6206],
           #             [11.4928, 10.3766]])
print(t5)  # >>> tensor([0.0000, 2.5000, 5.0000])
print(t6)  # >>> tensor([0, 1, 2, 3, 4])
```

# Tensor Initialization

## ◈Random seed

– torch.manual_seed(seed)

- it will set the seed of the random number generator to a fixed value, so that when you call for example torch.rand(2), the results will be reproducible

```
import torch


torch.manual_seed(1729)
random1 = torch.rand(2, 3)
print(random1)
# >>> tensor([[0.3126, 0.3791, 0.3087],
#              [0.0736, 0.4216, 0.0691]])

random2 = torch.rand(2, 3)
print(random2)
# >>> tensor([[0.2332, 0.4047, 0.2162],
#              [0.9927, 0.4128, 0.5938]])
```

```
import torch

torch.manual_seed(1729)
random3 = torch.rand(2, 3)
print(random3)
# >>> tensor([[0.3126, 0.3791, 0.3087],
#              [0.0736, 0.4216, 0.0691]])

random4 = torch.rand(2, 3)
print(random4)
# >>> tensor([[0.2332, 0.4047, 0.2162],
#              [0.9927, 0.4128, 0.5938]])
```

# Tensor Operations

# Tensor Type Conversion

## ◈Tensor type conversion

```
import torch
a = torch.ones((2, 3))
print(a.dtype) # >>> torch.float32

b = torch.ones((2, 3), dtype=torch.int16)
print(b)
# >>> tensor([[1, 1, 1],
#             [1, 1, 1]], dtype=torch.int16)

c = torch.rand((2, 3), dtype=torch.float64) * 20.
print(c)
# >>> tensor([[16.0387, 16.0498, 11.2471],
#             [18.4285, 12.5039,  0.7902]],
#             dtype=torch.float64)

d = c.to(torch.int32)
print(d)
# >>> tensor([[16, 16, 11],
#             [18, 12,  0]], dtype=torch.int32)
```

broadcasting

```
double_d = torch.ones(10, 2, dtype=torch.double)
short_e = torch.tensor([[1, 2]], dtype=torch.short)

double_d = torch.zeros(10, 2).double()
short_e = torch.ones(10, 2).short()

double_d = torch.zeros(10, 2).to(torch.double)
short_e = torch.ones(10, 2).to(dtype=torch.short)

print(double_d.dtype) # >>> torch.float64
print(short_e.dtype)  # >>> torch.int16

double_f = torch.rand(5, dtype=torch.double)
short_g = double_f.to(torch.short)
print((double_f * short_g).dtype)
# >>> torch.float64
```

# Tensor Operations

- Tensor operations
  - [NOTE] Element-wise operations and Broadcasting
  - Torch.add(other) or torch.add(input, other)
  - Torch.sub(other) or torch.sub(input, other)
  - Torch.mul(other) or torch.mul(input, other)
  - Torch.div(other) or torch.div(input, other)
  - Torch.dot(other) or torch.dot(input, other)
  - Torch.mm(other) or torch.mm(input, other)
  - Torch.bmm(other) or torch.bmm(input, other)
  - Torch.matmul(other) or torch.matmul(Tensor, Tensor)
  - Torch.pow(exponent) or torch.pow(input, exponent)

link_dl/_01_code/_02_tensors/f_tensor_operations.py

# Tensor Operations

$$\begin{pmatrix} V_1 \\ V_2 \\ V_3 \\ V_4 \end{pmatrix} \odot \begin{pmatrix} W_1 \\ W_2 \\ W_3 \\ W_4 \end{pmatrix} = \begin{pmatrix} V_1 * W_1 \\ V_2 * W_2 \\ V_3 * W_3 \\ V_4 * W_4 \end{pmatrix}$$

## ◈ Element-wise operations

- torch.add(input, other) ⟺ input + other
- torch.sub(input, other) ⟺ input - other

$$\begin{pmatrix} X_{1,1} & X_{1,2} \\ X_{2,1} & X_{2,2} \end{pmatrix} \odot \begin{pmatrix} Y_{1,1} & Y_{1,2} \\ Y_{2,1} & Y_{2,2} \end{pmatrix} = \begin{pmatrix} X_{1,1} * Y_{1,1} & X_{1,2} * Y_{1,2} \\ X_{2,1} * Y_{2,1} & X_{2,2} * Y_{2,2} \end{pmatrix}$$

```python
import torch

t1 = torch.ones(size=(2, 3))
t2 = torch.ones(size=(2, 3))

t3 = torch.add(t1, t2)
t4 = t1 + t2
print(t3) # >>> tensor([[2., 2., 2.],
          #             [2., 2., 2.]])
print(t4) # >>> tensor([[2., 2., 2.],
          #             [2., 2., 2.]])
```

```python
import torch

t1 = torch.ones(size=(2, 3))
t2 = torch.ones(size=(2, 3))

t5 = torch.sub(t1, t2)
t6 = t1 - t2
print(t3) # >>> tensor([[0., 0., 0.],
          #             [0., 0., 0.]])
print(t4) # >>> tensor([[0., 0., 0.],
          #             [0., 0., 0.]])
```

link_dl/_01_code/_02_tensors/f_tensor_operations.py

# Tensor Operations

## ◈ Element-wise operations

– torch.mul(input, other) ⟺ input * other

– torch.div(input, other) ⟺ input / other

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} * \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix} = \begin{bmatrix} a_1b_1 & a_2b_2 & a_3b_3 \\ a_4b_4 & a_5b_5 & a_6b_6 \\ a_7b_7 & a_8b_8 & a_9b_9 \end{bmatrix}$$

```
import torch

t1 = torch.ones(size=(2, 3))
t2 = torch.ones(size=(2, 3))

t7 = torch.mul(t1, t2)
t8 = t1 * t2
print(t7) # >>> tensor([[1., 1., 1.],
          #             [1., 1., 1.]])
print(t8) # >>> tensor([[1., 1., 1.],
          #             [1., 1., 1.]])
```

```
import torch

t1 = torch.ones(size=(2, 3))
t2 = torch.ones(size=(2, 3))

t9 = torch.div(t1, t2)
t10 = t1 / t2
print(t9)  # >>> tensor([[1., 1., 1.],
           #             [1., 1., 1.]])
print(t10) # >>> tensor([[1., 1., 1.],
           #             [1., 1., 1.]])
```

# Tensor Operations

## ◇Multiple matrix multiplications (1/2)

- torch.mul(input, other)
  - performs a element-wise multiplication with broadcasting
  - (Tensor) by (Tensor or Number)
- torch.dot(input, other)
  - Computes the dot product of two 1D tensors
  - (1D tensor) by (1D tensor)
- torch.mm(input, other)
  - performs a matrix multiplication without broadcasting
  - (2D tensor, n×m) by (2D tensor, m×p) → (n×p)
- torch.bmm(input, other)
  - performs a batch matrix multiplication without broadcasting
  - (b×n×m) by (b×m×p) → (b×n×p)

```
import torch
t1 = torch.dot(
    torch.tensor([2, 3]),
    torch.tensor([2, 1])
)
print(t1, t1.size())
# >>> tensor(7) torch.Size([])

t2 = torch.randn(2, 3)
t3 = torch.randn(3, 2)
t4 = torch.mm(t2, t3)
print(t4, t4.size())
# >>> tensor([[2.8075, 2.3547],
#             [2.4447, 2.0428]]),
#             torch.Size([2, 2])

t5 = torch.randn(10, 3, 4)
t6 = torch.randn(10, 4, 5)
t7 = torch.bmm(t5, t6)
print(t7.size())
# >>> tensor.Size([10, 3, 5])
```

# Tensor Operations

## ◈ Multiple matrix multiplications (2/2)

- torch.matmul(Tensor, Tensor) ⟺ Tensor @ Tensor
  - Various tensor product with broadcasting
  - it uses different modes depending on the input tensor shapes
  - Supported operation
    - ➢ dot product
    - ➢ matrix product
    - ➢ batched matrix products

```python
import torch

# vector x vector: dot
t1 = torch.randn(3)
t2 = torch.randn(3)
print(torch.matmul(t1, t2).size()) # >>> torch.Size([])

# matrix x vector: broadcasted dot
t3 = torch.randn(3, 4)
t4 = torch.randn(4)
print(torch.matmul(t3, t4).size()) # >>> torch.Size([3])

# batched matrix x vector: broadcasted dot
t5 = torch.randn(10, 3, 4)
t6 = torch.randn(4)
print(torch.matmul(t5, t6).size()) # >>> torch.Size([10, 3])

# batched matrix x batched matrix: bmm
t7 = torch.randn(10, 3, 4)
t8 = torch.randn(10, 4, 5)
print(torch.matmul(t7, t8).size()) # >>> torch.Size([10, 3, 5])

# batched matrix x matrix: bmm
t9 = torch.randn(10, 3, 4)
t10 = torch.randn(4, 5)
print(torch.matmul(t9, t10).size()) # >>> torch.Size([10, 3, 5])
```

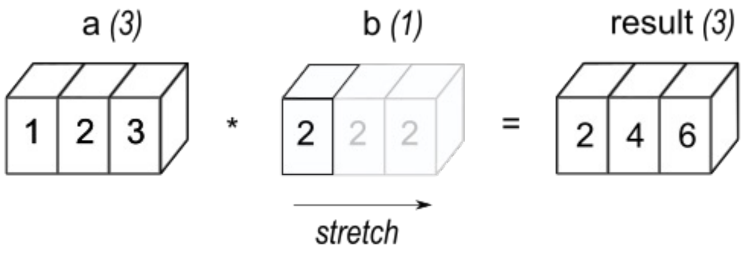
# Tensor Operations

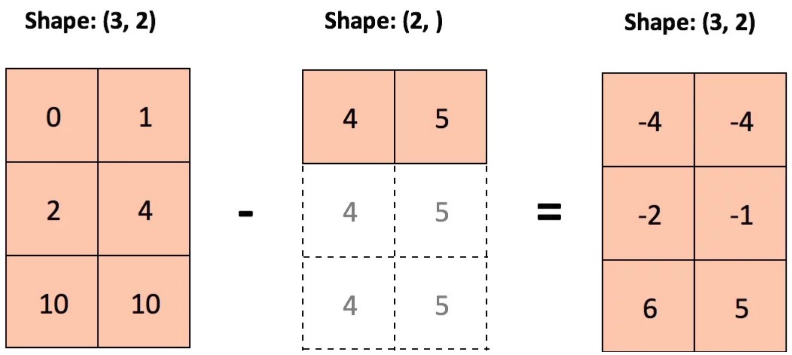
## ◈Broadcasting

– the smaller tensor is "broadcast" across the larger tensor so that they have compatible shapes and element-wise operations can be performed on them

- In broadcasting, the smaller array is found, the new axes are added as per the larger array, and data is added appropriately to the transformed array

```
import torch

t1 = torch.tensor([1.0, 2.0, 3.0])
t2 = 2.0
print(t1 * t2) # >>> tensor([2., 4., 6.])

t3 = torch.tensor([[0, 1], [2, 4], [10, 10]])
t4 = torch.tensor([4, 5])
print(t3 - t4) # >>> tensor([[-4, -4],
               #             [-2, -1],
               #             [ 6,  5]])
```



# Tensor Operations

## ◈Broadcasting Examples

```
import torch
t5 = torch.tensor([[1., 2.], [3., 4.]])

print(t5 + 2.0)  # t5.add(2.0)
# >>> tensor([[3., 4.],
#             [5., 6.]])

print(t5 - 2.0)  # t5.sub(2.0)
# >>> tensor([[-1.,  0.],
#             [ 1.,  2.]])

print(t5 * 2.0)  # t5.mul(2.0)
# >>> tensor([[2., 4.],
#             [6., 8.]])

print(t5 / 2.0)  # t5.div(2.0)
# >>> tensor([[0.5000, 1.0000],
#             [1.5000, 2.0000]])
```

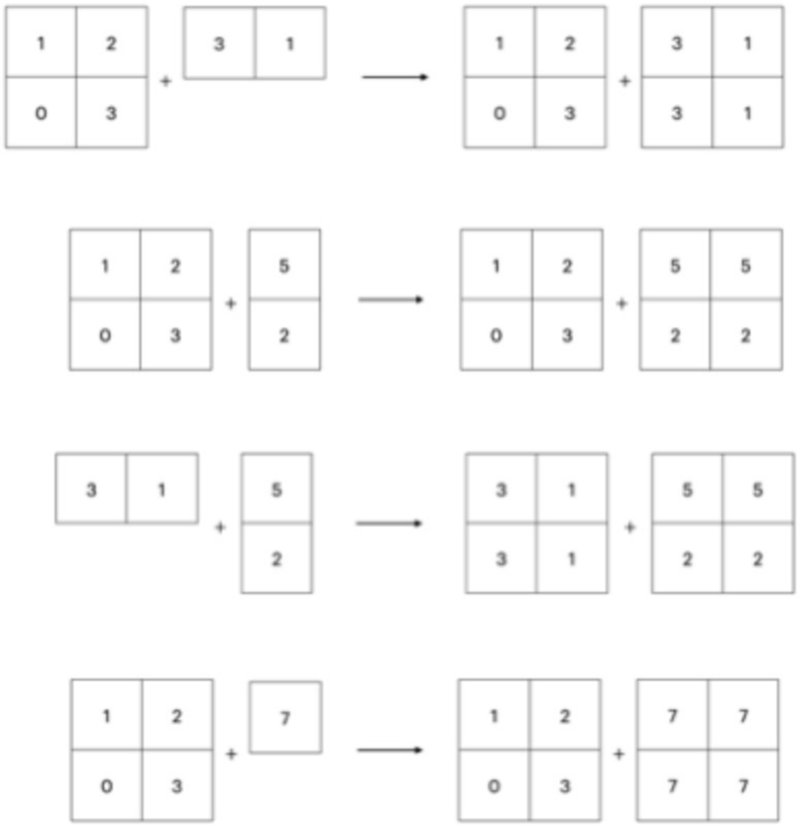
```
import torch


def normalize(x):
    return x / 255

t6 = torch.randn(3, 28, 28)

print(normalize(t6).size())
# >>> torch.Size([3, 28, 28])
```

# Tensor Operations

## ◈Broadcasting Rules

- Each tensor must have at least one dimension - no empty tensors.
- Comparing the dimension sizes of the two tensors, going from last to first:
  - Each dimension must be equal, or
  - One of the dimensions must be of size 1, or
  - The dimension does not exist in one of the tensors

```
import torch

t7 = torch.tensor([[1, 2], [0, 3]])  # torch.Size([2, 2])
t8 = torch.tensor([[3, 1]])          # torch.Size([1, 2])
t9 = torch.tensor([[5], [2]])        # torch.Size([2, 1])
t10 = torch.tensor([7])              # torch.Size([1])

print(t7 + t8)   # >>> tensor([[4, 3], [3, 4]])
print(t7 + t9)   # >>> tensor([[6, 7], [2, 5]])
print(t8 + t9)   # >>> tensor([[8, 6], [5, 3]])
print(t7 + t10)  # >>> tensor([[8, 9], [7, 10]])
```



# Tensor Operations

## ◈Broadcasting Rules

– Comparing the dimension sizes of the two tensors, going from last to first

```
import torch

t11 =       torch.ones(4, 3, 2)
t12 = t11 * torch.rand(   3, 2) # 3rd & 2nd dims identical to t11, dim 0 absent
print(t12.shape) # >>> torch.Size([4, 3, 2])

t13 =       torch.ones(4, 3, 2)
t14 = t13 * torch.rand(   3, 1) # 3rd dim = 1, 2nd dim is identical to t13
print(t14.shape) # >>> torch.Size([4, 3, 2])

t15 =       torch.ones(4, 3, 2)
t16 = t15 * torch.rand(   1, 2) # 3rd dim is identical to t15, 2nd dim is 1
print(t16.shape) # >>> torch.Size([4, 3, 2])

t17 = torch.ones(5, 3, 4, 1)
t18 = torch.rand(   3, 1, 1) # 2nd dim is identical to t17, 3rd and 4th dims are 1
print((t17 + t18).size()) # >>> torch.Size([5, 3, 4, 1])
```

# Tensor Operations

## ◈Broadcasting Rules

– Comparing the dimension sizes of the two tensors, going from last to first

```
import torch


t19 = torch.empty(5, 1, 4, 1)
t20 = torch.empty(   3, 1, 1)
print((t19 + t20).size())   # torch.Size([5, 3, 4, 1])

t21 = torch.empty(1)
t22 = torch.empty(3, 1, 7)
print((t21 + t22).size())   # torch.Size([3, 1, 7])

t23 = torch.ones(3, 3, 3)
t24 = torch.ones(3, 1, 3)
print((t23 + t24).size())   # torch.Size([3, 3, 3])

t25 = torch.empty(5, 2, 4, 1)
t26 = torch.empty(3, 1, 1)
print((t25 + t26).size())   # RuntimeError: The size of tensor a (2) must match
                            # the size of tensor b (3) at non-singleton dimension
```

# Tensor Operations

## ◈Pow() operation and its broadcasting

– Torch.pow(exponent) or torch.pow(input, exponent)

```
import torch


t27 = torch.ones(4) * 5
print(t27) # >>> tensor([ 5, 5, 5, 5])

t28 = torch.pow(t27, 2)
print(t28) # >>> tensor([ 25, 25, 25, 25])

exp = torch.arange(1., 5.) # tensor([ 1.,  2.,  3.,  4.])
a = torch.arange(1., 5.)   # tensor([ 1.,  2.,  3.,  4.])

t29 = torch.pow(a, exp)
print(t29) # >>> tensor([   1.,    4.,   27.,  256.])
```

# Tensor Indexing & Slicing

# Tensor Indexing & Slicing

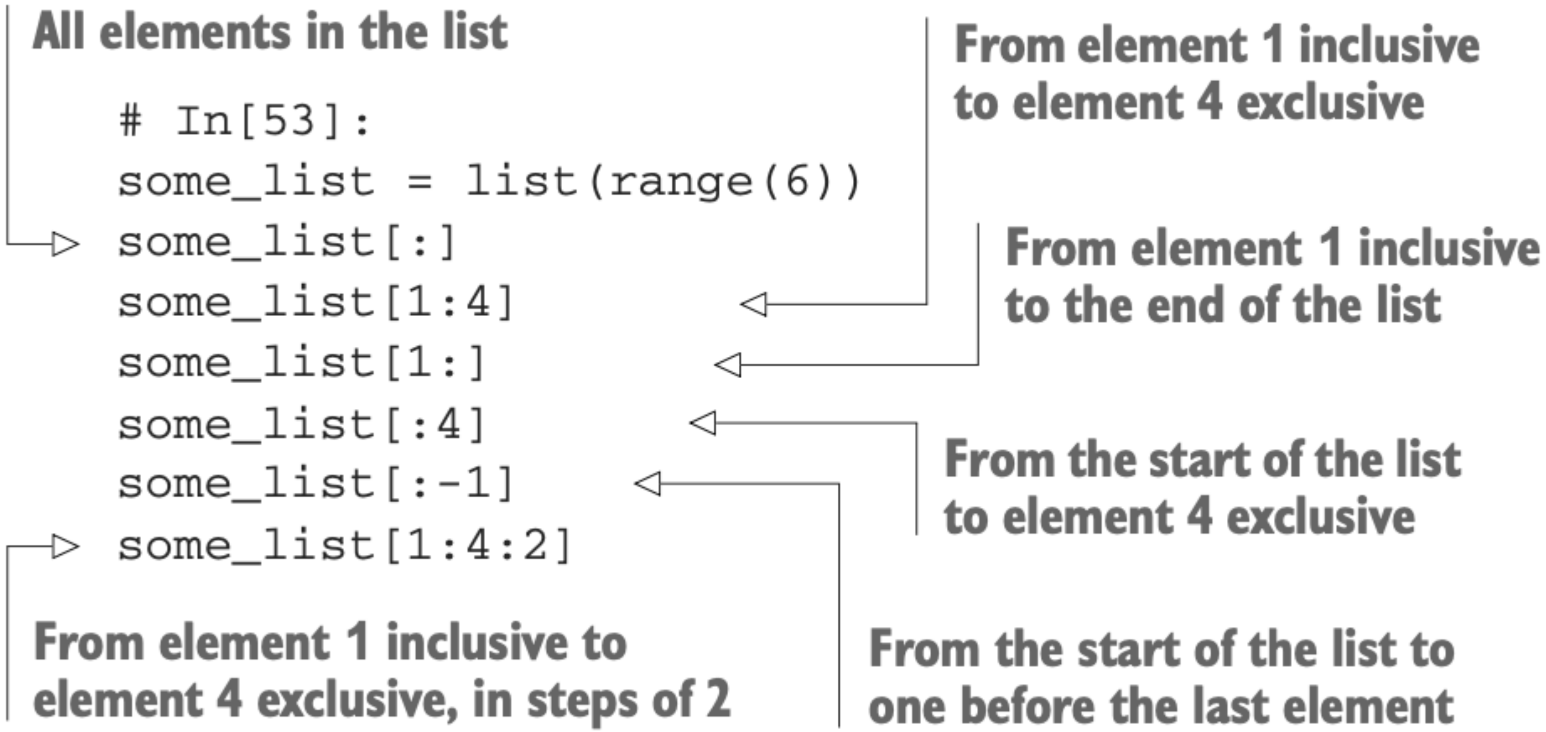
## ◈Tensor Indexing & Slicing

- accessing or retrieving specific elements or slices from a tensor
- common methods
  - Basic indexing: accessing individual elements of a tensor by specifying the indices along each dimension
  - Slicing: Extracting a specific sub-tensor from a larger tensor by specifying ranges along each dimension
  - Advanced indexing: Some libraries support more advanced indexing methods, such as using boolean masks or arrays of indices to select elements from a tensor based on certain conditions

# Tensor Indexing & Slicing

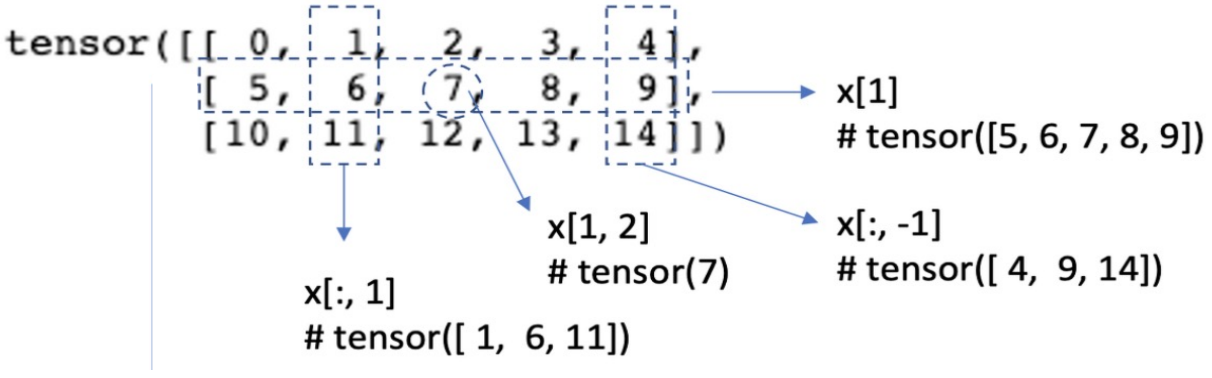
## ◈Tensor Indexing & Slicing

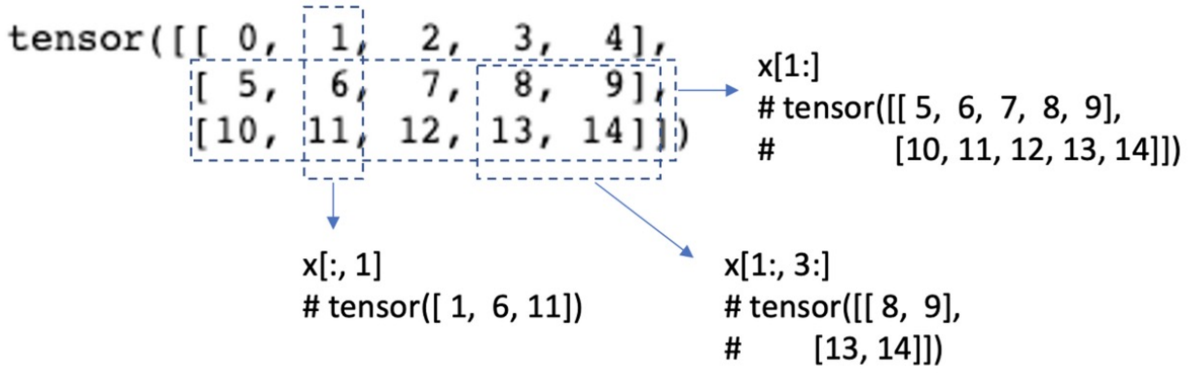
– They are almost same as the classical python indexing & slicing



# Tensor Indexing & Slicing

## ◈Tensor Indexing & Slicing




```python
import torch

x = torch.tensor(
    [[ 0,  1,  2,  3,  4],
     [ 5,  6,  7,  8,  9],
     [10, 11, 12, 13, 14]]
)

print(x[1])  # >>> tensor([5, 6, 7, 8, 9])
print(x[:, 1])   # >>> tensor([1, 6, 11])
print(x[1, 2])   # >>> tensor(7)
print(x[:, -1])  # >>> tensor([4, 9, 14)

print(x[1:])  # >>> tensor([[ 5,  6,  7,  8,  9],
              #              [10, 11, 12, 13, 14]])
print(x[1:, 3:])  # >>> tensor([[ 8,  9],
                  #              [13, 14]])
```

[Source: https://rfriend.tistroy.com

# Tensor Indexing & Slicing

## ◈Indexing & Slicing - Examples

```
import torch

y = torch.zeros((6, 6))
y[1:4, 2] = 1
print(y)
# >>> tensor([[0., 0., 0., 0., 0., 0.],
#              [0., 0., 1., 0., 0., 0.],
#              [0., 0., 1., 0., 0., 0.],
#              [0., 0., 1., 0., 0., 0.],
#              [0., 0., 0., 0., 0., 0.],
#              [0., 0., 0., 0., 0., 0.]])

print(y[1:4, 1:4])
# >>> tensor([[0., 1., 0.],
#              [0., 1., 0.],
#              [0., 1., 0.]])
```

```
import torch

z = torch.tensor(
    [[1, 2, 3, 4],
     [2, 3, 4, 5],
     [5, 6, 7, 8]]
)
print(z[:2])  # >>> tensor([[1, 2, 3, 4],
              #             [2, 3, 4, 5]])

print(z[1:, 1:3])  # >>> tensor([[3, 4],
                   #             [6, 7]])

print(z[:, 1:])  # >>> tensor([[2, 3, 4],
                 #             [3, 4, 5],
                 #             [6, 7, 8]])
z[1:, 1:3] = 0
print(z)  # >>> tensor([[1, 2, 3, 4],
          #             [2, 0, 0, 5],
          #             [5, 0, 0, 8]])
```

# Tensor Reshaping

# Tensor Reshaping

## ◈Tensor Reshaping Methods

- Tensor.view(*shape)
- Tensor.reshape(*shape) or torch.reshape(input, shape)
- Tensor.unsqueeze(dim) or torch.unsqueeze(input, dim)
- Tensor.squeeze(dim) or torch.squeeze(input, dim)
- Tensor.flatten(start_dim=0, end_dim=-1) or
  torch.flatten(input, start_dim=0, end_dim=-1)
- Tensor.permute(*dims) or torch.permute(input, dims)
- Tensor.transpose(dim0, dim1) or torch.transpose(input, dim0, dim1)
- Tensor.t() or torch.t()

# Tensor Reshaping

- torch.view(input, *shape) & torch.reshape(input, shape)
  - change the shape of a tensor without modifying its data
  - The returned tensor will share the underling data with the original tensor
  - But, torch.reshape() may return a copy for the non-contiguous tensor

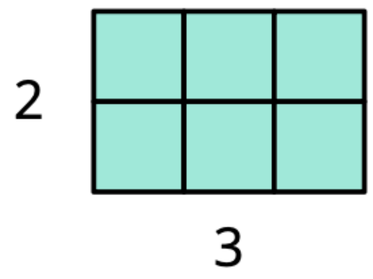
```
import torch
import torch

t1 = torch.tensor([[1, 2, 3], [4, 5, 6]])
t2 = t1.view(3, 2)     # Shape becomes (3, 2)
t3 = t1.reshape(1, 6)  # Shape becomes (1, 6)
print(t2)
# >>> tensor([[1, 2],
#             [3, 4],
#             [5, 6]])
print(t3)
# >>> tensor([[1, 2, 3, 4, 5, 6]])

t4 = torch.arange(8).view(2, 4)
t5 = torch.arange(6).view(2, 3)
print(t4)
# >>> tensor([[0, 1, 2, 3],
#             [4, 5, 6, 7]])
print(t5)
# >>> tensor([[0, 1, 2],
#             [3, 4, 5]])
```

# Tensor Reshaping

- **torch.unsqueeze()**
  - torch.unsqueeze() adds a new dimension to the tensor at the specified position

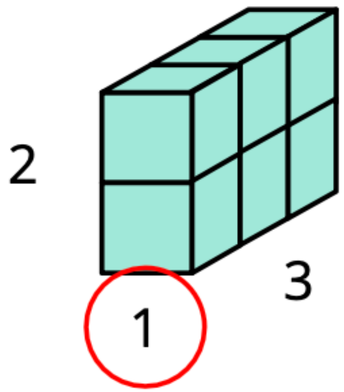
**Unsqueeze**: expand a new dimension

```
>>> x = torch.zeros([2, 3])
>>> x.shape
torch.Size([2, 3])
```




```
>>> x = x.unsqueeze(1)
>>> x.shape
torch.Size([2, 1, 3])
```

(dim = 1)






[Source: Yuan Tseng]

# Tensor Reshaping

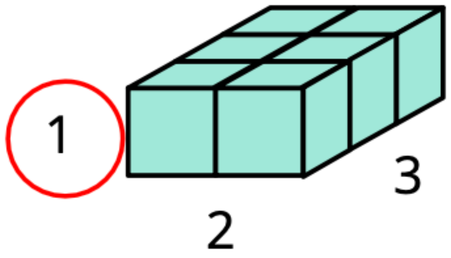
## ◇ torch.squeeze()

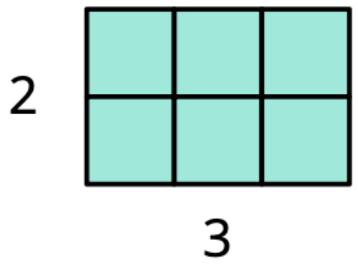
- torch.squeeze() method removes all dimensions of size 1 or the specified dimension if its size is 1

**Squeeze**: remove the specified dimension with length = 1

```
>>> x = torch.zeros([1, 2, 3])
>>> x.shape
torch.Size([1, 2, 3])
>>> x = x.squeeze(0)
>>> x.shape
torch.Size([2, 3])
```

(dim = 0)

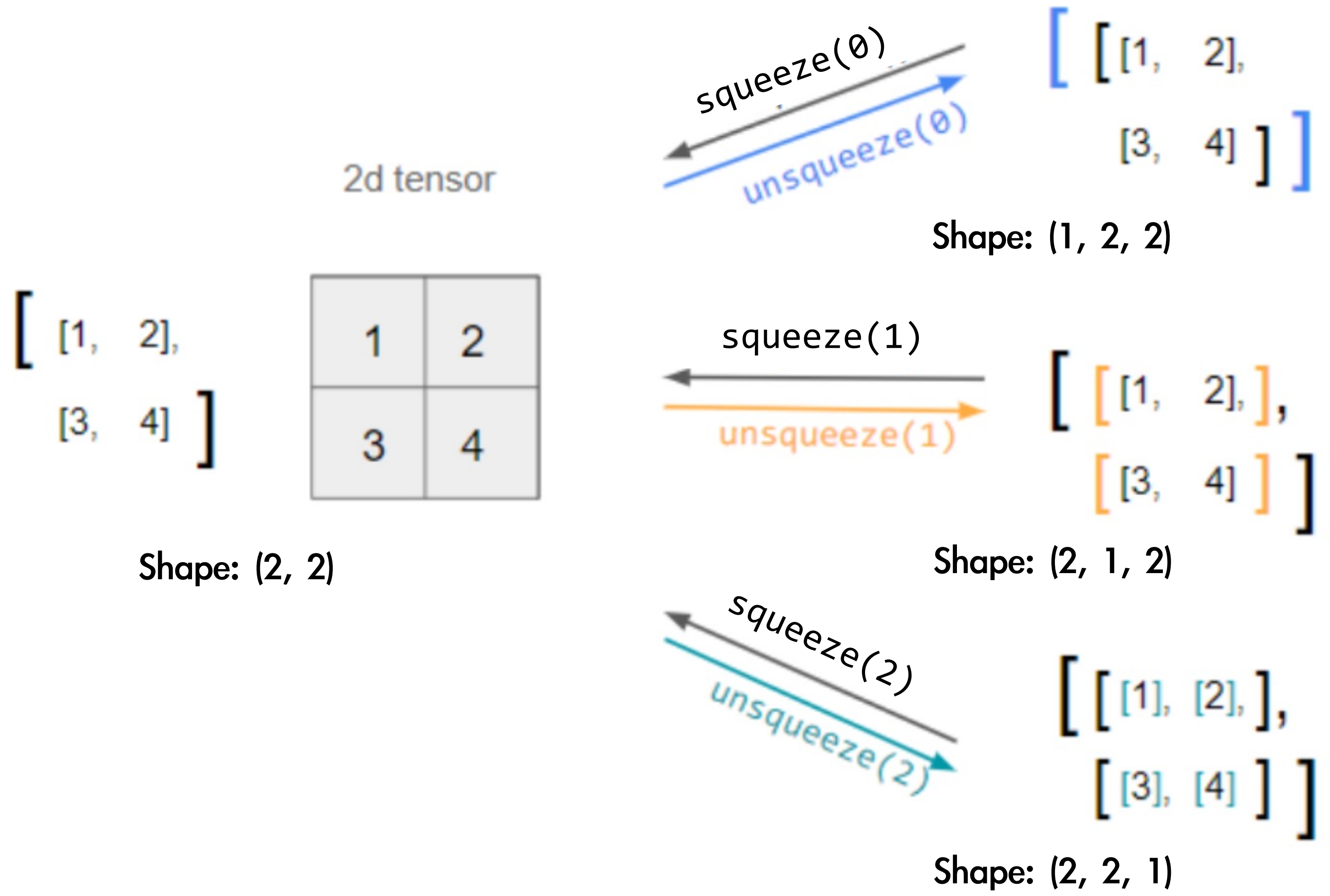



[Source: Yuan Tseng]

# Tensor Reshaping

## ◈ torch.unsqueeze() & torch.squeeze()




[Source: https://amber-chaeeunk.tistory.com/84]

# Tensor Reshaping

## ◈ torch.unsqueeze() & torch.squeeze()

```python
import torch

# Original tensor with shape (1, 3, 1)
t6 = torch.tensor([[[1], [2], [3]]])

# Remove all dimensions of size 1
t7 = t6.squeeze()   # Shape becomes (3,)

# Remove dimension at position 0
t8 = t6.squeeze(0)  # Shape becomes (3, 1)

print(t7)
# tensor([1, 2, 3])

print(t8)
# tensor([[1],
#         [2],
#         [3]])
```

```python
import torch

# Original tensor with shape (3,)
t9 = torch.tensor([1, 2, 3])

# Add a new dimension at position 1
t10 = t9.unsqueeze(1)  # Shape becomes (3, 1)
print(t10)
# >>> tensor([[1],
#              [2],
#              [3]])

t11 = torch.tensor(
    [[1, 2, 3],
     [4, 5, 6]]
)
t12 = t11.unsqueeze(1)  # Shape becomes (2, 1, 3)
print(t12)
# >>> tensor([[[1, 2, 3]],
#              [[4, 5, 6]]])
```

# Tensor Reshaping

- torch.flatten(input, start_dim=0, end_dim=-1)
  - Flattens input by reshaping it into a one-dimensional tensor
  - If start_dim or end_dim are passed, only dimensions starting with start_dim and ending with end_dim are flattened

```
import torch

# Original tensor with shape (2, 3)
t13 = torch.tensor([[1, 2, 3], [4, 5, 6]])

# Flatten the tensor
t14 = t13.flatten()  # Shape becomes (6,)

print(t14)
# >>> tensor([1, 2, 3, 4, 5, 6])
```

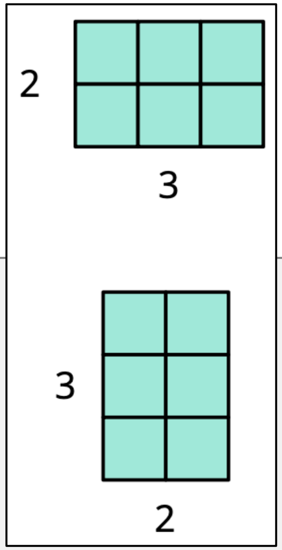
```
import torch
# Original tensor with shape (2, 2, 2)
t15 = torch.tensor([[[1, 2],
                     [3, 4]],
                    [[5, 6],
                     [7, 8]]])
t16 = torch.flatten(t15)
t17 = torch.flatten(t15, start_dim=1)

print(t16)
# >>> tensor([1, 2, 3, 4, 5, 6, 7, 8])

print(t17) # Shape becomes (2, 4)
# >>> tensor([[1, 2, 3, 4],
#             [5, 6, 7, 8]])
```

# Tensor Reshaping

- torch.permute(input, dims) & torch.transpose(input, dim0, dim1)
  - Returns a view of the original tensor input with its dimensions permuted
  - In case of torch.transpose, the given dimensions dim0 and dim1 are swapped

```
import torch


t18 = torch.randn(2, 3, 5)
print(t18.shape)                          # >>> torch.Size([2, 3, 5])
print(torch.permute(t18, (2, 0, 1)).size()) # >>> torch.Size([5, 2, 3])


# Original tensor with shape (2, 3)
t19 = torch.tensor([[1, 2, 3], [4, 5, 6]])
t20 = torch.permute(t19, dims=(0, 1))  # Shape becomes (2, 3) still
t21 = torch.permute(t19, dims=(1, 0))  # Shape becomes (3, 2)
print(t20)  # >>> tensor([[1, 2, 3],
            #             [4, 5, 6]])
print(t21)  # >>> tensor([[1, 4],
            #             [2, 5],
            #             [3, 6]])
t22 = torch.transpose(t19, 0, 1)  # Shape becomes (3, 2)
print(t22)  # >>> tensor([[1, 4],
            #             [2, 5],
            #             [3, 6]])
```

# Tensor Reshaping




## ◇ torch.transpose(input, dim0, dim1) & torch.t()

- In case of torch.transpose(), the given dimensions dim0 and dim1 are swapped
- In case of torch.t(), it expects input to be 2D tensor and transposes dimensions 0 and 1

```
import torch

# Original tensor with shape (2, 3)
t19 = torch.tensor([[1, 2, 3], [4, 5, 6]])

t22 = torch.transpose(t19, 0, 1)  # Shape becomes (3, 2)
print(t22)  # >>> tensor([[1, 4],
            #             [2, 5],
            #             [3, 6]])


t23 = torch.t(t19)                # Shape becomes (3, 2)
print(t23) # >>> tensor([[1, 4],
           #             [2, 5],
           #             [3, 6]])
```

# Tensor Stacking

# Tensor Stacking

## ◈Tensor Stacking Methods

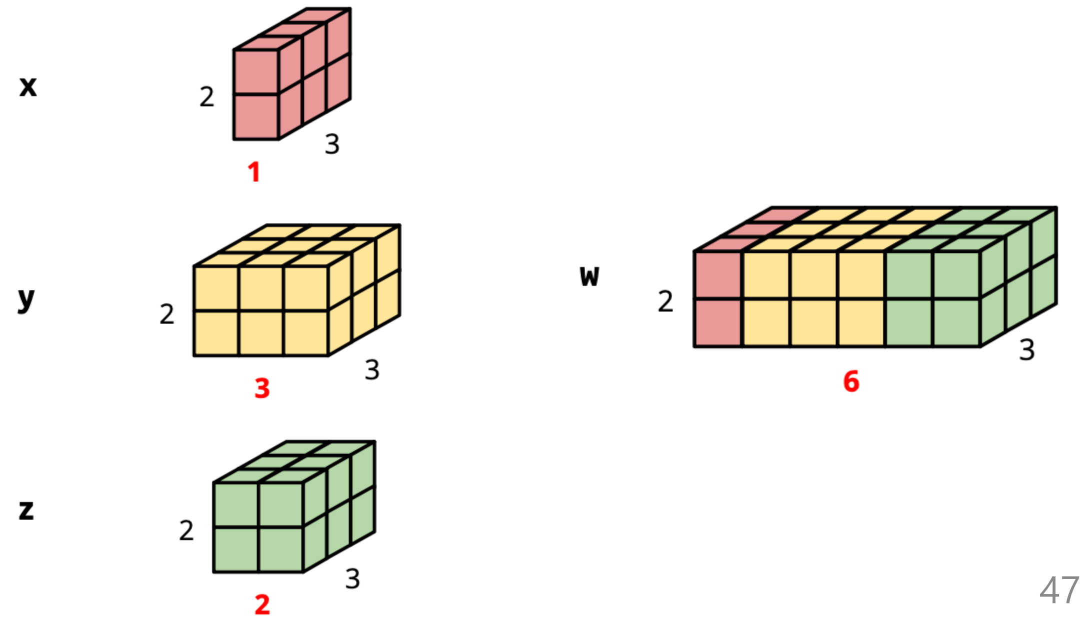
– torch.cat(tensors, dim=0) or torch.concat(tensors, dim=0)

- It takes a sequence of tensors and concatenates them along the specified dimension, resulting in a new tensor
- torch.concat(tensors, dim=0) is alias of torch.cat(tensors, dim=0)

# Tensor Stacking

- torch.cat(tensors, dim=0) or torch.concat(tensors, dim=0)
  - It takes a sequence of tensors and concatenates them along the specified dimension, resulting in a new tensor
  - torch.concat(tensors, dim=0) is alias of torch.cat(tensors, dim=0)

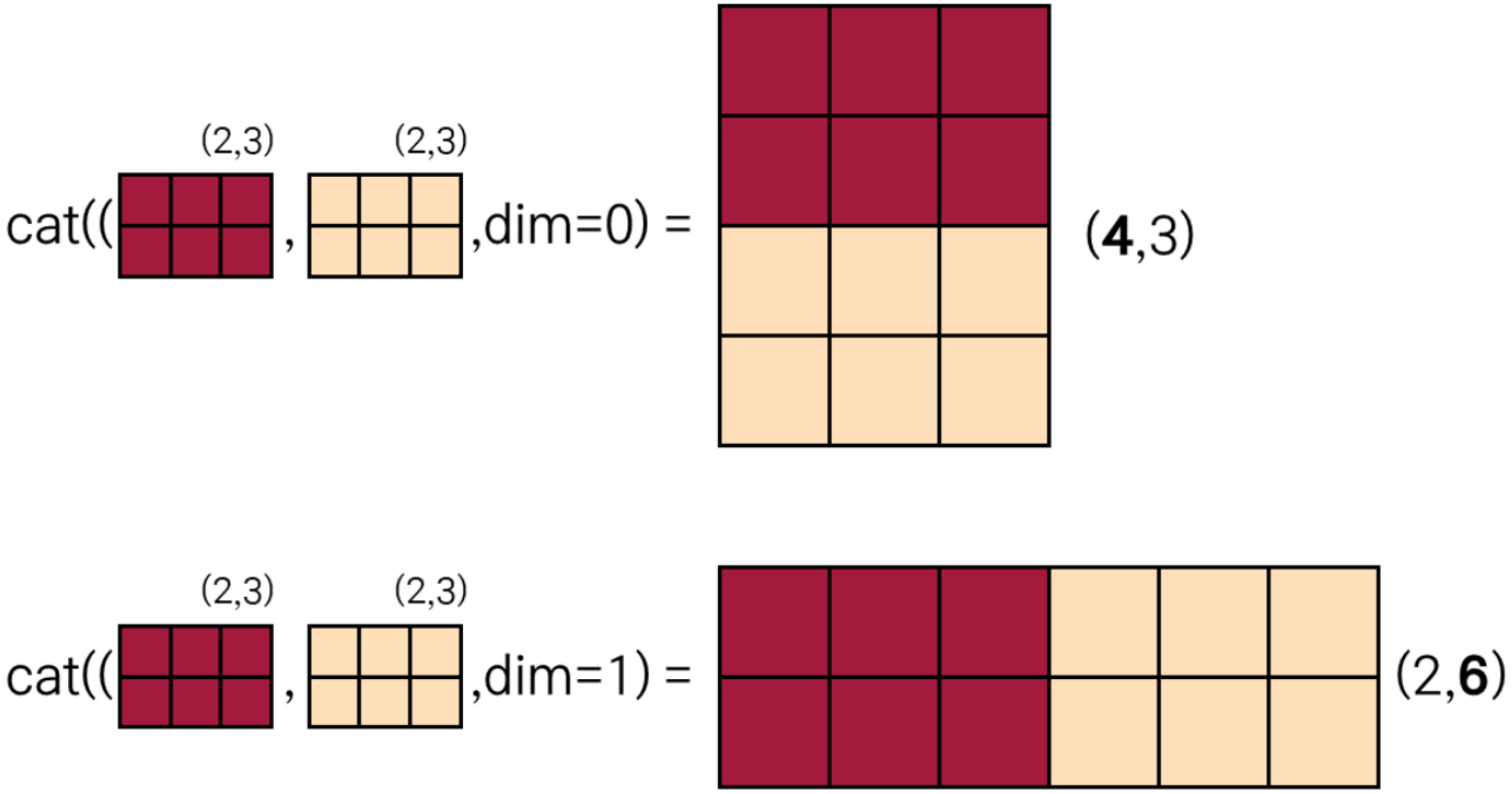
```
>>> x = torch.zeros([2, 1, 3])
>>> y = torch.zeros([2, 3, 3])
>>> z = torch.zeros([2, 2, 3])
>>> w = torch.cat([x, y, z], dim=1)
>>> w.shape
torch.Size([2, 6, 3])
```



# Tensor Stacking

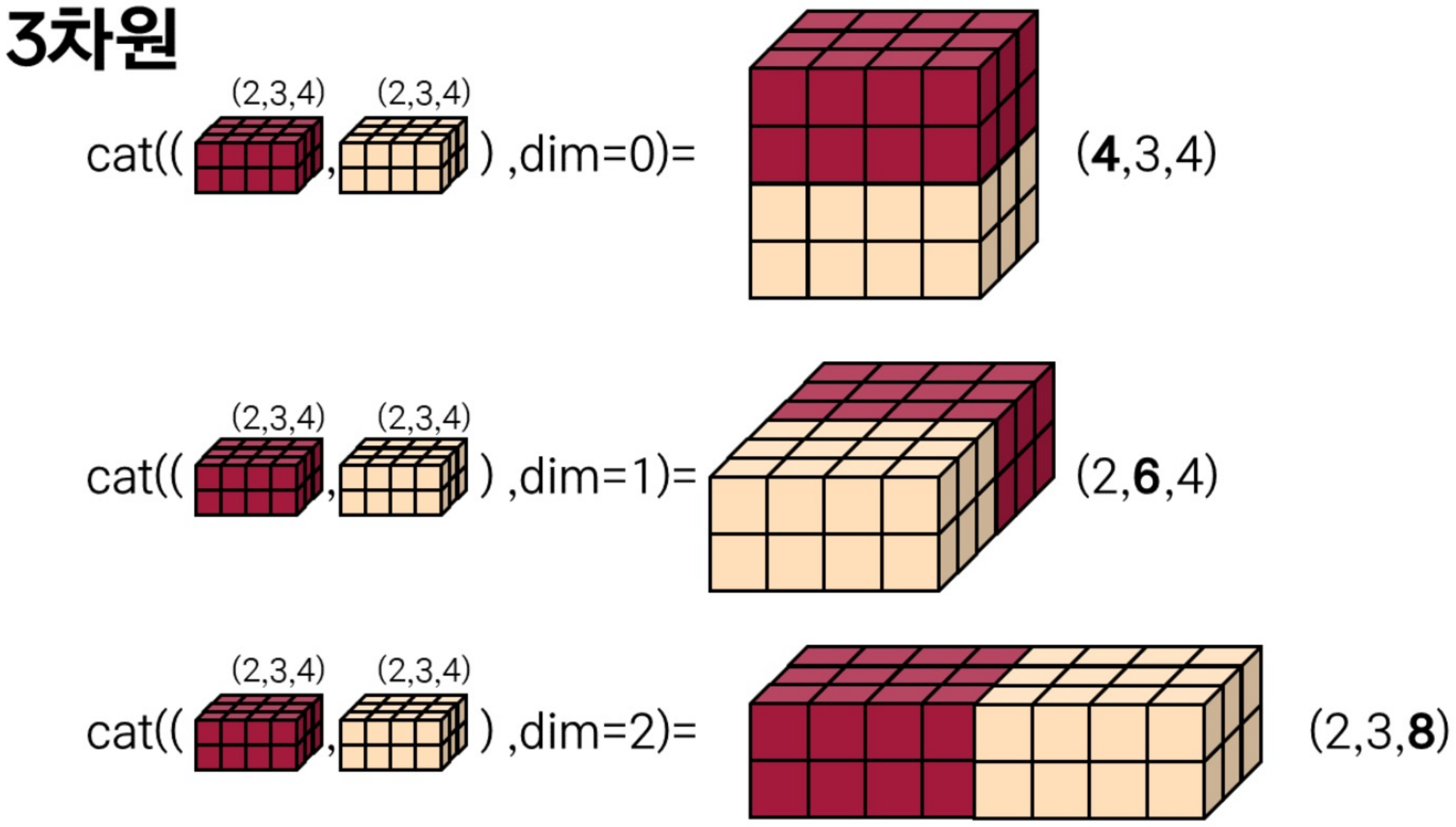
◈ torch.cat(tensors, dim=0) or torch.concat(tensors, dim=0)

**2차원**



# Tensor Stacking

◈ torch.cat(tensors, dim=0) or torch.concat(tensors, dim=0)



# Tensor Stacking

- torch.cat(tensors, dim=0) or torch.concat(tensors, dim=0)

```
import torch

t1 = torch.zeros([2, 1, 3])
t2 = torch.zeros([2, 3, 3])
t3 = torch.zeros([2, 2, 3])

t4 = torch.cat([t1, t2, t3], dim=1)
print(t4.shape)  # >>> torch.Size([2, 6, 3])

t5 = torch.arange(0, 3)  # tensor([0, 1, 2])
t6 = torch.arange(3, 8)  # tensor([3, 4, 5, 6, 7])

t7 = torch.cat((t5, t6), dim=0)
print(t7.shape)  # >>> torch.Size([8])
print(t7)   # >>> tensor([0, 1, 2, 3, 4, 5, 6, 7])
```

```
t8 = torch.arange(0, 6).reshape(2, 3)
t9 = torch.arange(6, 12).reshape(2, 3)

t10 = torch.cat((t8, t9), dim=0)
print(t10.size())  # >>> torch.Size([4, 3])
print(t10)
# >>> tensor([[ 0,  1,  2],
#             [ 3,  4,  5],
#             [ 6,  7,  8],
#             [ 9, 10, 11]])

t11 = torch.cat((t8, t9), dim=1)
print(t11.size())  # >>> torch.Size([2, 6])
print(t11)
# >>> tensor([[ 0,  1,  2,  6,  7,  8],
#             [ 3,  4,  5,  9, 10, 11]])
```

# Tensor Stacking

◇ torch.cat(tensors, dim=0) or torch.concat(tensors, dim=0)

```
import torch

t12 = torch.arange(0, 6).reshape(2, 3)
t13 = torch.arange(6, 12).reshape(2, 3)
t14 = torch.arange(12, 18).reshape(2, 3)

t15 = torch.cat((t12, t13, t14), dim=0)
print(t15.size())  # >>> torch.Size([6, 3])
print(t15)
# >>> tensor([[ 0,  1,  2],
#             [ 3,  4,  5],
#             [ 6,  7,  8],
#             [ 9, 10, 11],
#             [12, 13, 14],
#             [15, 16, 17]])
t16 = torch.cat((t12, t13, t14), dim=1)
print(t16.size())  # >>> torch.Size([2, 9])
print(t16)
# >>> tensor([[ 0,  1,  2,  6,  7,  8, 12, 13, 14],
#             [ 3,  4,  5,  9, 10, 11, 15, 16, 17]])
```

```
t17 = torch.arange(0, 6).reshape(1, 2, 3)
t18 = torch.arange(6, 12).reshape(1, 2, 3)
t19 = torch.cat((t17, t18), dim=0)
print(t19.size())  # >>> torch.Size([2, 2, 3])
print(t19)
# >>> tensor([[[ 0,  1,  2],
#              [ 3,  4,  5]],
#             [[ 6,  7,  8],
#              [ 9, 10, 11]]])
t20 = torch.cat((t17, t18), dim=1)
print(t20.size())  # >>> torch.Size([1, 4, 3])
print(t20)
# >>> tensor([[[ 0,  1,  2],
#              [ 3,  4,  5],
#              [ 6,  7,  8],
#              [ 9, 10, 11]]])
t21 = torch.cat((t17, t18), dim=2)
print(t21.size())  # >>> torch.Size([1, 2, 6])
print(t21)
# >>> tensor([[[ 0,  1,  2,  6,  7,  8],
#              [ 3,  4,  5,  9, 10, 11]]])
```

# Tensor Stacking

- torch.cat(tensors, dim=0) or torch.concat(tensors, dim=0)

```
import torch

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

print(torch.cat([a, b], dim=0))
# >>> tensor([1, 2, 3, 4, 5, 6])

print(torch.cat([a, b], dim=1))
# >>> IndexError: Dimension out of range
# (expected to be in range of [-1, 0], but got 1)
```

# Tensor Stacking

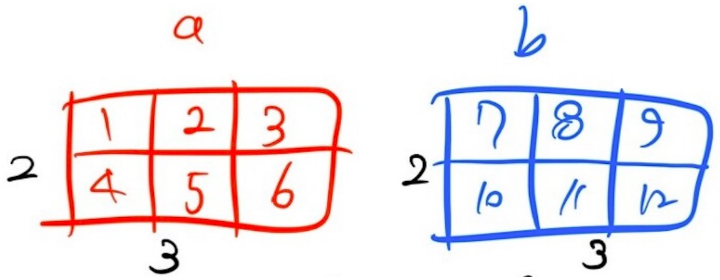
## ◈Tensor Stacking Methods

– torch.stack(tensors, dim=0)

- It takes a sequence of tensors and stacks them along a new dimension, creating a new tensor with one additional dimension
  - ➢새로운 차원으로 확장하여 텐서 시퀀스를 병합
- It expects each tensor to be equal size

$$\begin{aligned}
\text{cat}\left([(a,b),(a,b)],\text{dim}=0\right) &= (2,a,b)\\
\text{cat}\left([(a,b),(a,b)],\text{dim}=1\right) &= (a,2,b)\\
\text{cat}\left([(a,b),(a,b)],\text{dim}=2\right) &= (a,b,2)
\end{aligned}$$

# Tensor Stacking




## ◈Tensor Stacking Methods

– torch.stack([a, b], dim=0)
- a: [2, 3] → [1, 2, 3]
- b: [2, 3] → [1, 2, 3]

➔ [2, 2, 3]

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

```
tensor([[ 7,  8,  9],
        [10, 11, 12]])
```

```
tensor([[[ 1,  2,  3],
         [ 4,  5,  6]],
        [[ 7,  8,  9],
         [10, 11, 12]]])
```

– torch.stack([a, b], dim=1)
- a: [2, 3] → [2, 1, 3]
- b: [2, 3] → [2, 1, 3]

➔ [2, 2, 3]

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

```
tensor([[[ 7,  8,  9]],
        [[10, 11, 12]]])
```

```
tensor([[[ 1,  2,  3],
         [ 7,  8,  9]],
        [[ 4,  5,  6],
         [10, 11, 12]]])
```

– torch.stack([a, b], dim=2)
- a: [2, 3] → [2, 3, 1]
- b: [2, 3] → [2, 3, 1]

➔ [2, 3, 2]

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

```
tensor([[[7],
         [8],
         [9]],
        [[10],
         [11],
         [12]]])
```

```
tensor([[[ 1,  7],
         [ 2,  8],
         [ 3,  9]],
        [[ 4, 10],
         [ 5, 11],
         [ 6, 12]]])
```

# Tensor Stacking

## ◈Tensor Stacking Methods

- torch.stack([a, b], dim=0)
  - a: [2, 3] → [1, 2, 3]
  - b: [2, 3] → [1, 2, 3]
  - ➔ [2, 2, 3]
- torch.stack([a, b], dim=1)
  - a: [2, 3] → [2, 1, 3]
  - b: [2, 3] → [2, 1, 3]
  - ➔ [2, 2, 3]
- torch.stack([a, b], dim=2)
  - a: [2, 3] → [2, 3, 1]
  - b: [2, 3] → [2, 3, 1]
  - ➔ [2, 3, 2]

```
import torch

t1 = torch.tensor([[1, 2, 3], [4, 5, 6]])
t2 = torch.tensor([[7, 8, 9], [10, 11, 12]])

t3 = torch.stack([t1, t2], dim=0)
t4 = torch.cat([t1.unsqueeze(dim=0), t2.unsqueeze(dim=0)], dim=0)
print(t3.shape, t3.equal(t4))
# >>> torch.Size([2, 2, 3]) True

t5 = torch.stack([t1, t2], dim=1)
t6 = torch.cat([t1.unsqueeze(dim=1), t2.unsqueeze(dim=1)], dim=1)
print(t5.shape, t5.equal(t6))
# >>> torch.Size([2, 2, 3]) True

t7 = torch.stack([t1, t2], dim=2)
t8 = torch.cat([t1.unsqueeze(dim=2), t2.unsqueeze(dim=2)], dim=2)
print(t7.shape, t7.equal(t8))
# >>> torch.Size([2, 3, 2]) True
```

# Tensor Stacking

## ◆torch.stack(tensors, dim=0)

```
t9 = torch.arange(0, 3)   # tensor([0, 1, 2])
t10 = torch.arange(3, 6)  # tensor([3, 4, 5])

print(t9.size(), t10.size())
# >>> torch.Size([3]) torch.Size([3])

t11 = torch.stack((t9, t10), dim=0)
print(t11.size())  # >>> torch.Size([2,3])
print(t11)
# >>> tensor([[0, 1, 2],
#             [3, 4, 5]])

t12 = torch.cat(
    (t9.unsqueeze(0), t10.unsqueeze(0)),
    dim=0
)
print(t11.equal(t12))
# >>> True
```

```
t13 = torch.stack((t9, t10), dim=1)
print(t13.size()) # >>> torch.Size([3,2])
print(t13)
# >>> tensor([[0, 3],
#             [1, 4],
#             [2, 5]])

t14 = torch.cat(
    (t9.unsqueeze(1), t10.unsqueeze(1)),
    dim=1
)
print(t13.equal(t14))
# >>> True
```

# Tensor Stacking

## ❖Tensor Stacking Methods

- torch.vstack(tensors)
  - Stack tensors in sequence vertically (row wise)
    - ➢the tensors should have the same number of columns
- torch.hstack(tensors)
  - Stack tensors in sequence horizontally (column wise)
    - ➢the tensors should have the same number of rows

link_dl/_01_code/_02_tensors/n_tensor_vstack_hstack.py

# Tensor Stacking

## ◈ torch.vstack(tensors)

```python
import torch

t1 = torch.tensor([1, 2, 3])
t2 = torch.tensor([4, 5, 6])
t3 = torch.vstack((t1, t2))
print(t3)
# >>> tensor([[1, 2, 3],
#             [4, 5, 6]])

t4 = torch.tensor([[1], [2], [3]])
t5 = torch.tensor([[4], [5], [6]])
t6 = torch.vstack((t4, t5))
# >>> tensor([[1],
#             [2],
#             [3],
#             [4],
#             [5],
#             [6]])
```

```python
t7 = torch.tensor([
    [[1, 2, 3], [4, 5, 6]],
    [[7, 8, 9], [10, 11, 12]]
])
print(t7.shape)
# >>> (2, 2, 3)

t8 = torch.tensor([
    [[13, 14, 15], [16, 17, 18]],
    [[19, 20, 21], [22, 23, 24]]
])
print(t8.shape)
# >>> (2, 2, 3)

t9 = torch.vstack([t7, t8])
print(t9.shape)
# >>> (4, 2, 3)

print(t9)
# >>> tensor([[[ 1,  2,  3],
#              [ 4,  5,  6]],
#             [[ 7,  8,  9],
#              [10, 11, 12]],
#             [[13, 14, 15],
#              [16, 17, 18]],
#             [[19, 20, 21],
#              [22, 23, 24]]])
```

# Tensor Stacking

## ◈ torch.hstack(tensors)

```
import torch

t10 = torch.tensor([1, 2, 3])
t11 = torch.tensor([4, 5, 6])
t12 = torch.hstack((t10, t11))
print(t12)
# >>> tensor([1, 2, 3, 4, 5, 6])

t13 = torch.tensor([[1], [2], [3]])
t14 = torch.tensor([[4], [5], [6]])
t15 = torch.hstack((t13, t14))
print(t15)
# >>> tensor([[1, 4],
#             [2, 5],
#             [3, 6]])
```

```
t16 = torch.tensor([
    [[1, 2, 3], [4, 5, 6]],
    [[7, 8, 9], [10, 11, 12]]
])
print(t16.shape)
# >>> (2, 2, 3)

t17 = torch.tensor([
    [[13, 14, 15], [16, 17, 18]],
    [[19, 20, 21], [22, 23, 24]]
])
print(t17.shape)
# >>> (2, 2, 3)

t18 = torch.hstack([t16, t17])
print(t18.shape)
# >>> (2, 4, 3)

print(t18)
# >>> tensor([[[ 1,  2,  3],
#               [ 4,  5,  6],
#               [13, 14, 15],
#               [16, 17, 18]],
#              [[ 7,  8,  9],
#               [10, 11, 12],
#               [19, 20, 21],
#               [22, 23, 24]]])
```

# Moving to Tensors into GPU

# Moving tensors to the GPU

## ◈Tensors at GPU

– Every PyTorch tensor can be transferred to (one of) the GPU(s) in order to perform massively parallel, fast computations

```
import torch

points = torch.tensor([4.0, 1.0, 5.0, 3.0, 2.0, 1.0])
points_gpu = points.to(device='cuda')
```

– We can create a tensor on the GPU

```
import torch

points_gpu = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]], device='cuda')
# or
points_gpu = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]], device='cuda:0')
```

# Moving tensors to the GPU

## ◈Tensors at GPU

– At this point, any operation performed on the tensor, such as multiplying all elements by a constant, is carried out on the GPU

```
# Multiplication performed on the GPU
points_gpu = 2 * points.to(device='cuda’)
points_gpu = points_gpu + 4
```

– We can move the tensor back to the CPU

```
points_cpu = points_gpu.to(device='cpu')
```

– We can also use the shorthand methods

```
points_gpu = points.cuda() # Defaults to GPU index 0
points_gpu = points.cuda(0)
points_cpu = points_gpu.cpu()
```

# Moving tensors to the GPU

## ◈Tensors at GPU

– Check if your computer has NVIDIA GPU

```
if torch.cuda.is_available():
    points_gpu = points.cuda()
    # or
    points_gpu = points.to('cuda')
```

– Multiple GPUs

- specify 'cuda:0', 'cuda:1', 'cuda:2', ...