
▼ Lab2A - Introduction to PyTorch

Tensor (`torch.tensor`) is the data structure used in PyTorch to build a deep learning system. Tensors are similar to NumPy's `ndarrays`, with the addition being that Tensors can also be used on a GPU to accelerate computing.

Objectives:

In this lab, you learn how to

- Create tensors in PyTorch
- Perform mathematical operation on tensors
- Convert between PyTorch tensor and Numpy array
- Reshape a PyTorch tensor
- Transfer tensor to and from GPU

Table of Content:

1. [Creating tensors](#)
2. [Tensor operations](#)
3. [Indexing](#)
4. [Reshaping tensors](#)
5. [CUDA Tensors](#)
6. [Exercise](#)

Reference:

- [PyTorch Official Tutorial: What is PyTorch](#)
-

```
1 import torch
```

▼ 1. Creating tensors

Create with some predefined value and the data type is `torch.float32`.

```

1 x = torch.tensor(((1., 2., 3.),
2                   (4., 5., 6.)))
3 print(x)
4 print(x.dtype)

```

```

↳ tensor([[1., 2., 3.],
          [4., 5., 6.]])
   torch.float32

```

Create with some predefined value and the data type is `torch.int64`.

```

1 x = torch.tensor(((1, 2, 3),
2                   (4, 5, 6)))
3 print(x)
4 print(x.dtype)

```

```

tensor([[1, 2, 3],
        [4, 5, 6]])
torch.int64

```

Construct a matrix filled with zeros and explicitly specify the data type as `int32`

```

1 x = torch.zeros(5, 3, dtype=torch.int32)
2 print(x)

```

```

tensor([[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]], dtype=torch.int32)

```

Construct a matrix filled with ones and of dtype `float64`

```

1 x = torch.ones(5,3, dtype=torch.float64)
2 print(x)

```

```

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

```

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

Construct a tensor filled with random numbers from a uniform distribution on the interval $[0, 1)$.

```
1 x = torch.rand(5, 3)  
2 print(x)  
  
tensor([[0.2434, 0.8556, 0.6097],  
        [0.5317, 0.9519, 0.4816],  
        [0.1091, 0.9662, 0.9061],  
        [0.2698, 0.3311, 0.2392],  
        [0.4871, 0.6694, 0.2201]])
```

Construct a tensor filled with random numbers from a normal distribution with mean 0 and variance 1.

```
1 x = torch.randn(5, 3)  
2 print(x)  
  
tensor([[ -1.8372,  0.3281,  1.2349],  
        [ 0.8242,  0.1678,  0.7187],  
        [ 0.1902,  0.4981, -0.5814],  
        [ 0.9376,  1.3845, -1.4442],  
        [-1.1858,  0.2200,  2.3943]])
```

Construct a tensor filled with integers ranging from 0 to 9 with a shape of (5, 3). The 1st argument is the starting index (inclusive), the 2nd argument is the ending index (exclusive), and the third argument is a tuple specifying the targeted shape.

```
1 x = torch.randint(0, 10, (5, 3))  
2 print(x)  
  
tensor([[4, 6, 8],  
        [4, 5, 3],  
        [2, 7, 9],  
        [8, 5, 8],  
        [3, 0, 4]])
```

▼ 2. Tensor Operations

Size of tensors

```
1 x = torch.rand(5, 3)
2 print('x:\n', x)
3 print('Shape of x:', x.size())
4 print('Shape of x:', x.shape)

x:
  tensor([[0.0440, 0.0743, 0.1276],
          [0.1466, 0.6794, 0.8985],
          [0.3569, 0.3011, 0.4798],
          [0.4015, 0.9260, 0.7114],
          [0.4184, 0.5118, 0.0248]])
Shape of x: torch.Size([5, 3])
Shape of x: torch.Size([5, 3])
```

▼ Element-wise operations

In PyTorch, many operations, e.g., `*`, `+`, `/`, `torch.exp`, `torch.log`, etc., are basically element-wise operations of two arrays with the **same** shape. E.g., element-wise multiplication multiplies the items at the corresponding location.

Element-wise multiplication with the `*` operator

```
1 a = torch.randint(0, 10, (5,))
2 print('a:\n', a)
3
4 b = torch.randint(0, 10, (5,))
5 print('b:\n', b)
6
7 c = a * b
8 print('cb:\n', c)

a:
  tensor([6, 6, 1, 7, 1])
b:
  tensor([4, 6, 0, 4, 9])
cb:
  tensor([24, 36, 0, 28, 9])
```

Element-wise multiplication with `torch.multiply` function

```

1 a = torch.randint(0, 10, (5,))
2 print('a:\n', a)
3
4 b = torch.randint(0, 10, (5,))
5 print('b:\n', b)
6
7 c = torch.multiply(a, b)
8 print('c:\n', c)

a:
tensor([2, 8, 3, 2, 8])
b:
tensor([7, 5, 8, 8, 1])
c:
tensor([14, 40, 24, 16, 8])

```

Element-wise multiplication with `<tensor>.multiply` method

```

1 a = torch.randint(0, 10, (5,))
2 print('a:', a)
3
4 b = torch.randint(0, 10, (5,))
5 print('b:', b)
6
7 c = a.multiply(b)
8 print('c:', c)

a: tensor([1, 7, 3, 0, 6])
b: tensor([4, 5, 8, 9, 4])
c: tensor([ 4, 35, 24,  0, 24])

```

Inplace element-wise multiplication with `<tensor>.multiply_` method

```

1 a = torch.randint(0, 10, (5,))
2 print('a:', a)
3
4 b = torch.randint(0, 10, (5,))
5 print('b:', b)
6
7 b.multiply_(a)
8 print('\nInplace multiplication:')

```

```

9 print('a:', a)
10 print('b:', b)

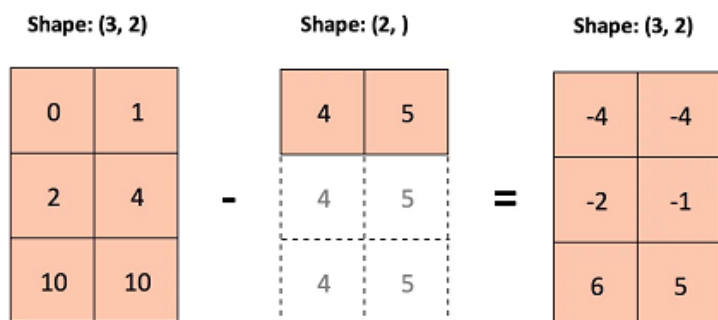
a: tensor([1, 7, 9, 3, 2])
b: tensor([0, 7, 7, 0, 3])

Inplace multiplication:
a: tensor([1, 7, 9, 3, 2])
b: tensor([ 0, 49, 63,  0,  6])

```

▼ Broadcasting

Broadcasting allows element-wise operations on tensors that are not of the same size. Pytorch automatically broadcast the *smaller* tensor to the size of the *larger* tensor, if certain constraints are met.



```

1 a = torch.randint(0, 10, (3, 2))
2 print('a:\n', a)
3
4 b = torch.randint(0, 10, (2,))
5 print('b:\n', b)
6
7 c = a - b
8 print('\na - b:\n', c)

```

```

a:
tensor([[3, 6],
        [1, 2],
        [2, 0]])
b:
tensor([7, 9])

```

```
a - b:
```

```
tensor([[ -4,  -3],
        [ -6,  -7],
        [ -5,  -9]])
```

```
1 a = torch.randint(0, 10, (3, 2))
2 print('a:\n', a)
3
4 b = torch.randint(0, 10, (3, 1))
5 print('b:\n', b)
6
7 c = a - b
8 print('\na - b:\n', c)
```

```
a:
tensor([[3, 7],
        [4, 6],
        [0, 8]])
```

```
b:
tensor([[8],
        [6],
        [4]])
```

```
a - b:
tensor([[ -5,  -1],
        [ -2,   0],
        [ -4,   4]])
```

▼ Matrix operations

Matrix operations are different from *element-wise operation* where the former is based on the rules of linear algebra.

To perform matrix operation, different from numpy, the `dot` command is used only to perform vector-vector multiplication (`dot`). Other specific commands are used for matrix-vector multiplication (`mv`) and matrix-matrix multiplication (`mm`). If you wish to perform matrix multiplication on matrices of different shape, you can use the command `matmul` or the `@` operator.

dot

```
1 a = torch.Tensor([4, 2])
2 b = torch.Tensor([3, 1])
3 r = torch.dot(a, b)
4
5 print(r)
```

```
tensor(14.)
```

mv

```
1 mat = torch.randn(2, 4)
2 vec = torch.randn(4)
3 r = torch.mv(mat, vec)
4 print(r)
```

```
tensor([0.4885, 2.9056])
```

mm

```
1 mat1 = torch.randn(2, 3)
2 mat2 = torch.randn(3, 4)
3 r = torch.mm(mat1, mat2)
4
5 print(r)
```

```
tensor([[ -0.4559, -2.4688, -1.7561,  2.0766],
        [ -0.3222, -1.2150, -7.6438,  3.0513]])
```

matmul

```
1 r1 = torch.matmul(a, b)
2 r2 = torch.matmul(mat, vec)
3 r3 = torch.matmul(mat1, mat2)
4
5 print(r1)
6 print(r2)
7 print(r3)
```

```
tensor(14.)
tensor([0.4885, 2.9056])
tensor([[ -0.4559, -2.4688, -1.7561,  2.0766],
        [ -0.3222, -1.2150, -7.6438,  3.0513]])
```

The @ operator


```

1 r1 = a @ b
2 r2 = mat @ vec
3 r3 = mat1 @ mat2
4
5 print(r1)
6 print(r2)
7 print(r3)

tensor(14.)
tensor([0.4885, 2.9056])
tensor([[ -0.4559, -2.4688, -1.7561,  2.0766],
        [ -0.3222, -1.2150, -7.6438,  3.0513]])

```

▼ 3. Indexing

You can use standard Numpy-like indexing with Torch

```

1 x = torch.randint(0, 100, (5,10))
2 print(x)

tensor([[27, 68, 98, 80, 91,  3, 35, 18, 34,  8],
        [21,  2, 32, 34, 78, 95, 51, 33, 72, 75],
        [46, 57, 26, 11, 47,  3, 81, 28, 92, 41],
        [53, 49, 64, 47, 69, 48, 74, 33, 57, 23],
        [81, 73, 14,  6, 57, 13,  8, 79, 59, 58]])

```

```

1 # accessing column 1
2 print(x[:,1])

tensor([68,  2, 57, 49, 73])

```

```

1 # accessing columns 2 and 3
2 print(x[:, 2:4])

tensor([[98, 80],
        [32, 34],
        [26, 11],
        [64, 47],
        [14,  6]])

```

```

1 # accessing row 1
2 print(x[1,:])

    tensor([21,  2, 32, 34, 78, 95, 51, 33, 72, 75])

1 # accessing rows 2 and 3
2 print(x[2:4,:])

    tensor([[46, 57, 26, 11, 47,  3, 81, 28, 92, 41],
          [53, 49, 64, 47, 69, 48, 74, 33, 57, 23]])

```

▼ 4. Reshaping Tensors

Tensor.reshape

Returns a tensor with the same data and number of elements as self but with the specified shape.

```

1 x = torch.randint(0, 100, (2,4))
2 print('x:\n', x)

```

```

x:
    tensor([[50, 46, 80, 73],
          [70, 12, 47, 47]])

```

```

1 # Reshape from (2, 4) to (8, 1)
2 y = x.reshape(8, -1)
3 print('y:\n', y)

```

```

y:
    tensor([[50],
          [46],
          [80],
          [73],
          [70],
          [12],
          [47],
          [47]])

```

```

1 # Reshape from (2, 4) to (4, 2)
2 z = x.reshape(4, 2)

```

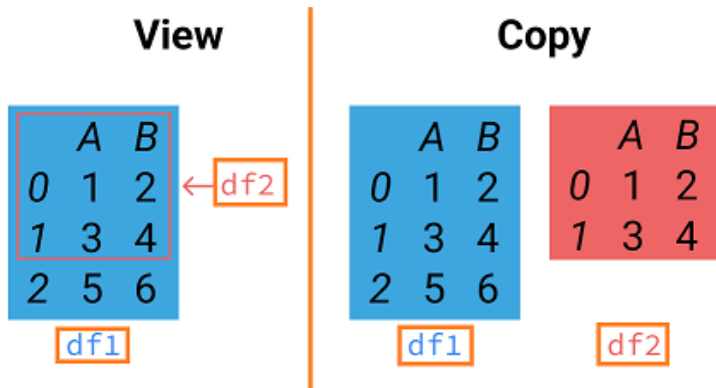
```

3 print('z:\n', z)

z:
  tensor([[50, 46],
          [80, 73],
          [70, 12],
          [47, 47]])

```

This method returns a **view** if shape is compatible with the current shape. Else, it may return a **copy**. This allows it to work with both [contiguous](#) and [non-contiguous](#) data.



In the examples above, we create a view since the shapes of reshaped tensors *y* and *z* are compatible with the original tensor *x*. Note that after a change is performed on *x*, then the changes will occur to both *y* and *z*.

The following code confirms that *y* and *z* are indeed **views** of *x*. Any changes to *x* will be observed in *y* and *z* as well.

```

1 x[0,0] = -3
2
3 print('x:\n', x)
4 print('y:\n', y)
5 print('z:\n', z)

x:
  tensor([[ -3, 46, 80, 73],
          [70, 12, 47, 47]])
y:
  tensor([[-3],
          [46],
          [80],
          [73],
          [70],

```

```

        [12],
        [47],
        [47]])
z:
  tensor([[ -3, 46],
         [80, 73],
         [70, 12],
         [47, 47]])

```

To check if two tensors have the same base content, use the command `data_ptr()`

```

1 y.data_ptr() == x.data_ptr()

True

```

Example of use of reshape that results in a copy

```

1 p = x.T.reshape(-1)
2
3 p.data_ptr() == x.data_ptr()

False

```

Tensor.view

`Tensor.view` always returns a **view** of the original tensor with the new shape, i.e., it will share the underlying data with the original tensor.

```

1 x = torch.randint(0, 100, (2, 4))
2 print('x:\n', x)

```

```

x:
  tensor([[37, 39, 67, 95],
         [46, 47, 98, 69]])

```

```

1 # Convert from (2, 4) to (8, 1)
2 y = x.view(8, -1)
3 print('y:\n', y)
4

```

```

y:
  tensor([[37],
         [39],
         [67],
         [95],
         [46],
         [47],
         [98],
         [69]])

```

```

1 # Convert from (2, 4) to (4, 2)
2 z = x.view(4, 2)
3 print('z:\n', z)

```

```

z:
  tensor([[37, 39],
         [67, 95],
         [46, 47],
         [98, 69]])

```

Similar to the numpy's `reshape` function, pytorch's `view` returns a reference of the original matrix albeit in a different shape

▼ 5. CUDA Tensors

Creating tensor in the GPU

```

1 if torch.cuda.is_available():
2     gpu = torch.device("cuda") # define a cuda device
3     x = torch.ones((2, 4), device = gpu) # Create the tensor in the GPU
4     print(x)

  tensor([[1., 1., 1., 1.],
         [1., 1., 1., 1.]], device='cuda:0')

```

▼ Creating tensor in the cpu explicitly (default)

```

1 cpu = torch.device("cpu")
2 x = torch.ones((2, 4), device = cpu)    # create a tensor in the CPU
3 print(x)

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

```

▼ Transferring tensor from cpu to gpu

Transfer using the `.cuda()` command.

```

1 x = torch.rand(3, 2, device = "cpu") # create tensor in cpu. The device argument also accepts a string besides a device object
2 print(x)
3
4 x = x.cuda() # move to GPU
5 print(x)

tensor([[0.1697, 0.0962],
        [0.7331, 0.6139],
        [0.5735, 0.4806]])
tensor([[0.1697, 0.0962],
        [0.7331, 0.6139],
        [0.5735, 0.4806]], device='cuda:0')

```

Transfer using the `.to()` command

```

1 x = torch.rand(3, 2) # create tensor in the CPU (default)
2 print(x)
3
4 gpu = torch.device('cuda') # move to GPU
5 x = x.to(gpu)
6 print(x)

tensor([[0.6615, 0.6024],
        [0.3098, 0.0767],
        [0.5146, 0.0107]])
tensor([[0.6615, 0.6024],
        [0.3098, 0.0767],
        [0.5146, 0.0107]], device='cuda:0')

```

▼ Transferring tensor from gpu to cpu

Transfer using the `.cpu()` command.

```
1 x = torch.rand(3, 2, device = 'cuda') # create tensor in gpu. The device argument also accepts a string besides a device object
2 print(x)
3
4 x = x.cpu() # move to CPU
5 print(x)

tensor([[0.8319, 0.3893],
        [0.3291, 0.6083],
        [0.1008, 0.6127]], device='cuda:0')
tensor([[0.8319, 0.3893],
        [0.3291, 0.6083],
        [0.1008, 0.6127]])
```

Transfer using the `.to()` command.

```
1 x = torch.rand(3, 2, device = 'cuda') # create tensor in the GPU
2 print(x)
3
4 device = torch.device('cpu')
5 x = x.to(device) # move to CPU
6 print(x)

tensor([[0.4656, 0.7950],
        [0.9062, 0.2038],
        [0.2981, 0.6544]], device='cuda:0')
tensor([[0.4656, 0.7950],
        [0.9062, 0.2038],
        [0.2981, 0.6544]])
```

▼ Exercise

Question 1. The following code is used to preprocess a batch data for Logistic Regression.

1.1 Create a random tensor `x_ori` using the normal distribution of shape `(4, 16, 16, 3)`. The tensor represent $m=4$ color image samples, each having a resolution of `(16, 16)` Expected ans: Shape of `x_ori`: `torch.Size([4, 16, 16, 3])`

```
1 ...
2 print('Shape of X_ori:', X_ori.shape)
```

Double-click (or enter) to edit

1.2 Reshape `X_ori` into a shape of `(4, 16*16*3)`. Then transpose the result to get a tensor of shape `(768, 4)` where each column represents a sample. Save the result as `x`.

Expected ans:

```
Shape of X: torch.Size([768, 4])
```

```
1 ...
2 print('Shape of X:', X.shape)
```

1.3 Check if a GPU is available in the system. If yes, transfer the tensor `x` to the GPU. Then, verify if `X` has really been loaded into the GPU (`X.is_cuda`) and print out the device ID of the GPU (`X.get_device()`).

Expected ans:

```
X is loaded to GPU: 0
```

```
1 if ...gpu is available...
2     ... load x to gpu ...
3
4 if ...x is successfully loaded into GPU:
5     print('X is loaded to GPU:', ... get the GPU ID...)
6 else:
7     print('X is loaded to CPU')
```

Question 2.

2.1 Create the tensor `A`. Ensure that the datatype for `A` is `float32`:


```
A = [[3, 2, 4, 6],  
      [2, 4, 2, 2],  
      [5, 1, 2, 1]]
```

```
1 ...  
2 print(A)
```

2.2 Extract the 2nd row from A. (Expected ans: tensor([2., 4., 2., 2.]))

```
1 print(...)
```

2.3 Extract the 3rd column from A. (Expected ans: tensor([4., 2., 2.]))

```
1 print(...)
```

2.4 Write the code to extract the following sub-block (rows 1 to 2 and columns 1 to 2) from A.

```
tensor([[4., 2.],  
        [1., 2.]])
```

```
1 print(...)
```

2.5 Compute the mean of all columns.

Expected ans:

```
tensor([3.7500, 2.5000, 2.2500], dtype=torch.float64)
```

```
1 print(...)
```

2.6 Repeat question 2.5, but this time retain the original dimensions such that the output has a shape of (3,1)

Expected ans:

```
tensor([[3.7500],  
        [2.5000],  
        [2.2500]], dtype=torch.float64)
```

```
1 print(...)
```

-- END OF LAB02A --

✓ 0s completed at 10:19 AM

