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:

Reference:

- PyTorch Official Tutorial: What is PyTorch

```
import torch
```

1. Creating tensors

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

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

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

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

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

```
x = torch.zeros(5, 3, dtype=torch.int32)
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

```
x = torch.ones(5,3, dtype=torch.float64)
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)`.

```
x = torch.rand(5, 3)
print(x)
```

```
tensor([[0.4108, 0.2439, 0.5374],
        [0.2762, 0.4976, 0.7811],
        [0.1253, 0.9014, 0.9606],
        [0.4385, 0.7086, 0.2921],
        [0.6970, 0.0086, 0.1462]])
```

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

```
x = torch.randn(5, 3)
print(x)
```

```
tensor([[-0.3306,  0.0656,  0.5600],
        [ 0.0285,  1.5690,  1.2773],
        [ 2.6518, -1.0800, -1.9852],
        [ 0.0206, -1.0917,  0.1177],
        [-0.9974, -1.2353,  1.0200]])
```

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.

```
x = torch.randint(0, 10, (5, 3))
print(x)
```

```
tensor([[3, 0, 0],
        [2, 2, 8],
        [3, 1, 3],
        [2, 1, 6],
        [4, 2, 0]])
```

---

2. Tensor Operations

Size of tensors

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

```
x:
 tensor([[7.5662e-04, 7.8733e-01, 7.8518e-01],
        [5.7968e-01, 2.6202e-01, 1.1754e-01],
        [7.8490e-01, 9.6953e-01, 6.4651e-02],
        [9.6084e-01, 4.0217e-01, 4.9407e-01],
        [5.7089e-01, 2.9056e-01, 1.9563e-01]])
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

```
a = torch.randint(0, 10, (5,))
print('a:\n', a)

b = torch.randint(0, 10, (5,))
print('b:\n', b)

```

```
c = a * b
print('cb:\n', c)
```

```
a:
 tensor([9, 8, 0, 8, 2])
b:
 tensor([8, 9, 3, 5, 5])
cb:
 tensor([72, 72,  0, 40, 10])
```

Element-wise multiplication with `torch.multiply` function

```
a = torch.randint(0, 10, (5,))
print('a:\n', a)

b = torch.randint(0, 10, (5,))
print('b:\n', b)

c = torch.multiply(a, b)
print('c:\n', c)
```

```
a:
 tensor([4, 1, 9, 6, 8])
b:
 tensor([3, 0, 5, 8, 1])
c:
 tensor([12,  0, 45, 48,  8])
```

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

```
a = torch.randint(0, 10, (5,))
print('a:', a)

b = torch.randint(0, 10, (5,))
print('b:', b)

c = a.multiply(b)
print('c:', c)
```

```
a: tensor([4, 2, 5, 3, 5])
b: tensor([2, 4, 1, 7, 9])
c: tensor([ 8,  8,  5, 21, 45])
```

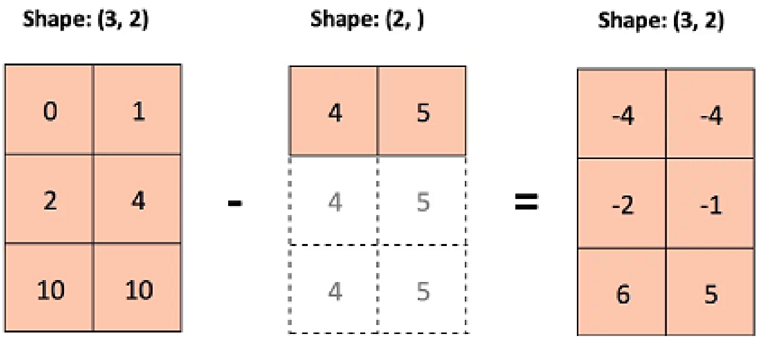
Inplace element-wise multiplication with `<tensor>.multiply_` method

```
a = torch.randint(0, 10, (5,))
print('a:', a)

b = torch.randint(0, 10, (5,))
print('b:', b)

b.multiply_(a)
print('\nInplace multiplication:')
print('a:', a)
print('b:', b)
```

```
a: tensor([8, 3, 9, 2, 0])
b: tensor([4, 0, 8, 6, 3])

Inplace multiplication:
a: tensor([8, 3, 9, 2, 0])
b: tensor([32,  0, 72, 12,  0])
```

Broadcasting

Boadcasting 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.



```python
a = torch.randint(0, 10, (3, 2))
print('a:\n', a)

b = torch.randint(0, 10, (2,))
print('b:\n', b)

c = a - b
print('\na - b:\n', c)
```

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

a - b:
 tensor([[7, 8],
        [1, 8],
        [7, 1]])
```

```python
a = torch.randint(0, 10, (3, 2))
print('a:\n', a)

b = torch.randint(0, 10, (3, 1))
print('b:\n', b)

c = a - b
print('\na - b:\n', c)
```

```
a:
 tensor([[5, 0],
        [8, 1],
        [0, 6]])
b:
 tensor([[4],
        [3],
        [4]])

a - b:
 tensor([[ 1, -4],
```

```
        [ 5, -2],
        [-4,  2]])
```

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

```
a = torch.randn(2,)
print('a\n', a)

b = torch.randn(2,)
print('b\n', b)

r = torch.dot(a, b)
print('r\n', r)
```

```
a
 tensor([-1.1423,  0.9024])
b
 tensor([-0.9470, -1.4128])
r
 tensor(-0.1932)
```

mv

```
mat = torch.randn(2, 4)
print('mat\n', mat)

vec = torch.randn(4)
print('vec\n', vec)

r = torch.mv(mat, vec)
print('r\n', r)
```

```
mat
 tensor([[ 0.6821,  0.1801, -0.8310, -0.4715],
        [-1.5674, -0.5946,  0.6564,  2.5081]])
vec
 tensor([-2.8798,  1.3170, -0.5992,  1.5482])
r
 tensor([-1.9591,  7.2202])
```

mm

```
mat1 = torch.randn(2, 3)
print('mat1\n', mat1)

mat2 = torch.randn(3, 4)
print('mat2\n', mat2)

r = torch.mm(mat1, mat2)
print('r\n', r)
```

```
mat1
 tensor([[-0.0256, -0.4530, -1.2581],
        [-0.2969, -0.5649,  1.0206]])
mat2
 tensor([[-0.3827,  1.5844,  0.5748,  1.4324],
        [-0.6114, -0.4763,  1.5103,  1.7844],
        [-0.0804, -0.1819,  0.9593, -0.7520]])
r
 tensor([[ 0.3880,  0.4041, -1.9059,  0.1011],
        [ 0.3770, -0.3870, -0.0448, -2.2008]])
```

matmul

```
r1 = torch.matmul(a, b)
print('r1\n', r1)

r2 = torch.matmul(mat, vec)
print('r2\n', r2)

r3 = torch.matmul(mat1, mat2)
print('r3\n', r3)
```

```
r1
 tensor(-0.1932)
r2
 tensor([-1.9591,  7.2202])
r3
 tensor([[ 0.3880,  0.4041, -1.9059,  0.1011],
        [ 0.3770, -0.3870, -0.0448, -2.2008]])
```

The `@` operator

```
r1 = a @ b
print('r1\n', r1)

r2 = mat @ vec
print('r2\n', r2)

r3 = mat1 @ mat2
print('r3\n', r3)
```

```
r1
 tensor(-0.1932)
r2
 tensor([-1.9591,  7.2202])
r3
 tensor([[ 0.3880,  0.4041, -1.9059,  0.1011],
        [ 0.3770, -0.3870, -0.0448, -2.2008]])
```

3. Indexing

You can use standard Numpy-like indexing with Torch

```
x = torch.randint(0, 100, (5,10))
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]])
```

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

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

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

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

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

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

```
# accessing rows 2 and 3
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.

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

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

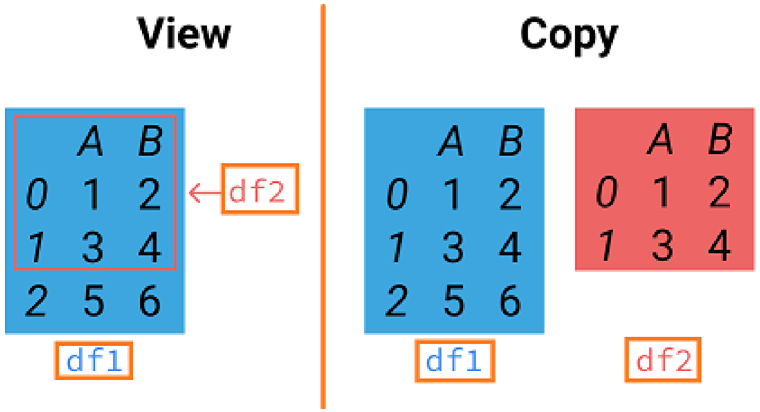
```
# Reshape from (2, 4) to (8, 1)
y = x.reshape(8, -1)
print('y:\n', y)
```

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

```
# Reshape from (2, 4) to (4, 2)
z = x.reshape(4, 2)
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-contigous 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.

```
x[0,0] = -3

print('x:\n', x)
print('y:\n', y)
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()`

```
y.data_ptr() == x.data_ptr()
```

```
True
```

Example of use of reshape that results in a copy

```
p = x.T.reshape(-1)

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.

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

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

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

```

```
y:
 tensor([[37],
        [39],
        [67],
        [95],
        [46],
        [47],
```

```
        [98],
        [69]])
```

```
# Convert from (2, 4) to (4, 2)
z = x.view(4, 2)
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

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

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

Creating tensor in the cpu explicitly (default)

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

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

Transfering tensor from cpu to gpu

Transfer using the `.cuda()` command.

```
x = torch.rand(3, 2, device = "cpu")  # create tensor in cpu. The device argument also accepts a string besides a device object
print(x)

x = x.cuda()  # move to GPU
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

```
x = torch.rand(3, 2) # create tensor in the CPU (default)
print(x)

gpu = torch.device('cuda')  # move to GPU
x = x.to(gpu)
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')
```

Transfering tensor from gpu to cpu

Transfer using the `.cpu()` command.

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

x = x.cpu()  # move to CPU
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.

```
x = torch.rand(3, 2, device = 'cuda')  # create tensor in the GPU
print(x)

device = torch.device('cpu')
x = x.to(device)  # move to CPU
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])`

```
...
```

```
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])
```

```
...
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
```

```
if ...gpu is available...
   ... load x to gpu ...

if ...x is successfully loaded into GPU:
    print('X is loaded to GPU:', ... get the GPU ID...)
else:
    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]]
```

```
...
print(A)
```

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

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

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

```
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.]])
```

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

2.5 Compute the mean of the rows.

Expected ans:

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

```
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)
```

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

--- END OF LAB02A ---