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

---

1. Creating tensors

```
import torch
```

Construct a 5x3 matrix, uninitialized

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

```
tensor([[1.5308e+21, 3.0823e-41, 3.3631e-44],
        [0.0000e+00,        nan, 3.0823e-41],
        [1.1578e+27, 1.1362e+30, 7.1547e+22],
        [4.5828e+30, 1.2121e+04, 7.1846e+22],
        [9.2198e-39, 7.0374e+22, 2.8865e+20]])
```

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

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

```
tensor([[0.7521, 0.2607, 0.5136],
        [0.0039, 0.1750, 0.5553],
        [0.7461, 0.3617, 0.8170],
        [0.6385, 0.8919, 0.1916],
        [0.4943, 0.4376, 0.7084]])
```

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([[-0.6189,  0.4508,  0.8886],
        [-1.0557,  1.2414, -0.3561],
        [ 0.9360, -1.5024,  0.9378],
        [ 0.2679, -0.2584, -0.0786],
```

```
            [-1.6949, -0.3799,  0.4828]])
```

Construct a matrix filled with zeros and of dtype long

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

```
tensor([[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]])
```

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

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

---

2. Tensor Operations

Size of tensors

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


print(x.size())   # torch.Size is actually a tuple
print(x.shape)
```

```
tensor([[0.5463, 0.4063, 0.2726],
```

```
tensor([[0.5463, 0.4063, 0.2726],
        [0.6868, 0.6290, 0.0590],
        [0.1826, 0.4690, 0.7819],
        [0.8968, 0.0451, 0.6952],
        [0.8669, 0.0513, 0.2640]])
torch.Size([5, 3])
torch.Size([5, 3])
```

Addition

There are multiple syntaxes for operations.

Addition: syntax 1

```
x = torch.rand(3, 2)
print('x:\n', x)
y = torch.rand(3, 2)
print('y:\n', y)

z = x + y
print('x+y:\n', z)
```

```
x:
 tensor([[0.3215, 0.2230],
        [0.2855, 0.1580],
        [0.3406, 0.8168]])
y:
 tensor([[0.8634, 0.2672],
        [0.2046, 0.7790],
        [0.7036, 0.6863]])
x+y:
 tensor([[1.1849, 0.4902],
        [0.4901, 0.9369],
        [1.0442, 1.5032]])
```

Addition: syntax 2

```
z = torch.add(x, y)
print('x+y:\n', z)
```

```
x+y:
 tensor([[1.1849, 0.4902],
        [0.4901, 0.9369],
        [1.0442, 1.5032]])
```

Addition: syntax 3 (in-place)

- Any operation that mutates a tensor in-place is post-fixed with an `_`. For example: `x.copy_(y)`, `x.t_()`, will change `x`.

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

```
x
 tensor([[0.3215, 0.2230],
        [0.2855, 0.1580],
        [0.3406, 0.8168]])
y
 tensor([[0.8634, 0.2672],
        [0.2046, 0.7790],
        [0.7036, 0.6863]])
```

```
y.add_(x)
print(y)
```

```
tensor([[1.1849, 0.4902],
        [0.4901, 0.9369],
        [1.0442, 1.5032]])
```

Multiplication

Different from numpy which uses mainly `dot` to perform different types of matrix multiplication, PyTorch uses different commands for vector-vector multiplication (`dot`), matrix-vector multiplication (`mv`) and matrix-matrix multiplication (`mm`)

dot

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

print(r)
```

```
tensor(14.)
```

mv

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

```
tensor([ 2.8342, -1.1781])
```

mm

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

print(r)
```

```
tensor([[ 0.6578, -1.6688, -0.2140,  0.7555],
        [-0.1618, -0.1385, -0.0776,  1.5601]])
```

3. Indexing

You can use standard Numpy-like indexing with Torch

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

```
tensor([[61, 31, 57, 48, 65, 98,  7, 13, 58, 14],
        [63,  7, 57, 32, 77, 44, 44, 71, 77, 32],
        [77, 60, 21, 68,  2, 13, 64, 74, 55, 33],
        [99, 74, 96, 71, 99, 25,  8, 77, 60, 70],
        [18, 31, 15,  6, 90, 12, 48, 81, 75, 62]])
```

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

```
tensor([31,  7, 60, 74, 31])
```

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

```
tensor([[57, 48],
        [57, 32],
        [21, 68],
        [96, 71],
        [15,  6]])
```

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

```
tensor([63,  7, 57, 32, 77, 44, 44, 71, 77, 32])
```

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

```
tensor([[77, 60, 21, 68,  2, 13, 64, 74, 55, 33],
```

```
tensor([[77, 60, 21, 68,  2, 13, 64, 74, 55, 55],
        [99, 74, 96, 71, 99, 25,  8, 77, 60, 70]])
```

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([[15, 97, 63, 37],
        [45, 41,  4, 97]])
```

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

```
y:
 tensor([[15],
        [97],
        [63],
        [37],
        [45],
        [41],
        [ 4],
        [97]])
```

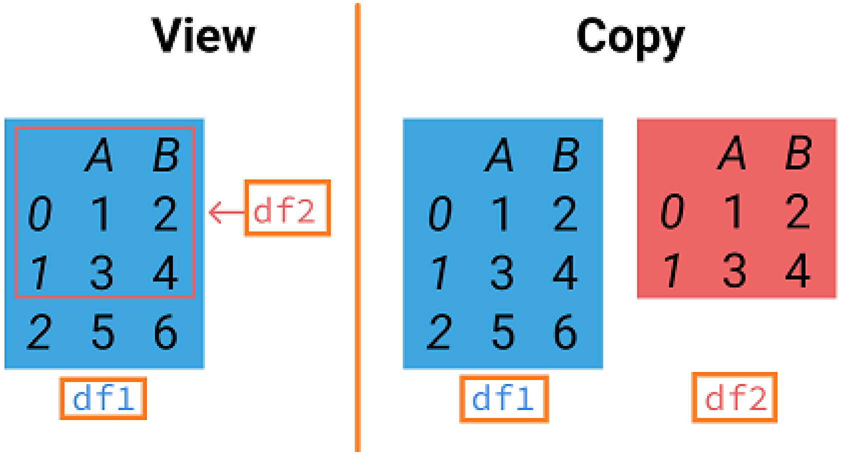
```
# Reshape from (2, 4) to (4, 2)

z = x.reshape(4, 2)
print('z:\n', z)
```

```
z:
```

```
z:
 tensor([[15, 97],
        [63, 37],
        [45, 41],
        [ 4, 97]])
```

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, 97, 63, 37],
        [45, 41,  4, 97]])
y:
 tensor([[-3],
```

```
        [97],
        [63],
        [37],
        [45],
        [41],
        [ 4],
        [97]])
z:
 tensor([[-3, 97],
        [63, 37],
        [45, 41],
        [ 4, 97]])
```

`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([[ 6, 94,  0, 65],
        [24, 59, 71, 69]])
```

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

```

```
y:
 tensor([[ 6],
        [94],
        [ 0],

        [65],
        [24],
        [59],
        [71],
```

```
        [69]])
```

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

```
z:
 tensor([[ 6, 94],
        [ 0, 65],
        [24, 59],
        [71, 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([[0.6650, 0.4313],
        [0.0991, 0.8726],
        [0.5617, 0.5138]])
tensor([[0.6650, 0.4313],
        [0.0991, 0.8726],
        [0.5617, 0.5138]], 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([[0.0312, 0.4889],
        [0.3286, 0.7417],
        [0.9349, 0.8032]])
tensor([[0.0312, 0.4889],
        [0.3286, 0.7417],
        [0.9349, 0.8032]], 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([[0.9459, 0.8531],
        [0.0874, 0.3044],
        [0.0451, 0.5759]], device='cuda:0')
tensor([[0.9459, 0.8531],
        [0.0874, 0.3044],
        [0.0451, 0.5759]])
```

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([[0.6613, 0.8984],
        [0.9873, 0.5311],
        [0.3895, 0.8701]], device='cuda:0')
tensor([[0.6613, 0.8984],

        [0.9873, 0.5311],
        [0.3895, 0.8701]])
```

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)
```

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.]])
```

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

2.5 Compute the mean of all rows. You may need to convert the tensor to a double first. Note that in the computational graph, when you create

```
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(...)
```

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

--- END OF LAB02A ---