02_Tensors_in_PyTorch

June 26, 2025

[3]:
```python
import torch
print(torch.__version__)
print(torch.cuda.is_available())
```

```
2.6.0+cu124
False
```

[5]:
```python
if torch.cuda.is_available():
    device = torch.device("GPU")
else:
    device = torch.device("cpu")
print(device)
```

```
cpu
```

1 Creating Tensor

[9]:
```python
# using empty
a = torch.empty(2,2)
a
```

[9]:
```
tensor([[4.8144e+17, 4.5399e-41],
        [4.8144e+17, 4.5399e-41]])
```

[11]:
```python
# check type
type(a)
```

[11]:
```
torch.Tensor
```

[12]:
```python
# using zeros
torch.zeros(2,3)
```

[12]:
```
tensor([[0., 0., 0.],
        [0., 0., 0.]])
```

[13]:
```python
# using ones
torch.ones(2,4)
```

[13]:
```
tensor([[1., 1., 1., 1.],
        [1., 1., 1., 1.]])
```

[16]:
```python
# using rand
torch.rand(2,3)
```

[16]:
```
tensor([[0.0886, 0.5391, 0.1133],
        [0.0390, 0.8530, 0.0826]])
```

[21]:
```python
# use of seed
torch.manual_seed(100)
torch.rand(2,3)
```

[21]:
```
tensor([[0.1117, 0.8158, 0.2626],
        [0.4839, 0.6765, 0.7539]])
```

[23]:
```python
# using tensor
torch.tensor([[1,2,3],[4,5,6]])
```

[23]:
```
tensor([[1, 2, 3],
        [4, 5, 6]])
```

[40]:
```python
# other ways
# arange
print(f'using arange ->{torch.arange(0,10,2)}')
print('--------------------------------------------')
# linspace
print(f'using linspace ->{torch.linspace(1,10,10)}')
print('--------------------------------------------')
# eye
print(f'using eye ->{torch.eye(7)}')
print('--------------------------------------------')
# diag
print(f'using diag ->{torch.diag(torch.rand(3,4))}')
print('--------------------------------------------')
# full
print(f'using full ->{torch.full((2,3),6)}')
```

```
using arange ->tensor([0, 2, 4, 6, 8])
--------------------------------------------
using linspace ->tensor([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])
--------------------------------------------
using eye ->tensor([[1., 0., 0., 0., 0., 0., 0.],
        [0., 1., 0., 0., 0., 0., 0.],
        [0., 0., 1., 0., 0., 0., 0.],
        [0., 0., 0., 1., 0., 0., 0.],
        [0., 0., 0., 0., 1., 0., 0.],
        [0., 0., 0., 0., 0., 1., 0.],
```

```
        [0., 0., 0., 0., 0., 0., 1.]])
-------------------------------------------
using diag ->tensor([0.3039, 0.7590, 0.9454])
-------------------------------------------
using full ->tensor([[6, 6, 6],
        [6, 6, 6]])
```

2 Tensor Shapes

[54]:
```
x = torch.tensor([[1,3,5],[2,4,6]])
x
```

[54]:
```
tensor([[1, 3, 5],
        [2, 4, 6]])
```

[45]:
```
x.shape
```

[45]:
```
torch.Size([2, 3])
```

[49]:
```
torch.empty_like(x)
```

[49]:
```
tensor([[    139149907971472,     139149907971472,                   0],
        [                  0,                   0, 7310593858020254331]])
```

[50]:
```
torch.zeros_like(x)
```

[50]:
```
tensor([[0, 0, 0],
        [0, 0, 0]])
```

[51]:
```
torch.ones_like(x)
```

[51]:
```
tensor([[1, 1, 1],
        [1, 1, 1]])
```

[66]:
```
torch.rand_like(x, dtype=torch.float32)
```

[66]:
```
tensor([[0.1893, 0.9186, 0.2131],
        [0.3957, 0.6017, 0.4234]])
```

3 Tensor Data Types

[57]:
```
# find data types
x.dtype
```

[57]:
```
torch.int64
```

[59]:
```python
# assign data type
torch.tensor([1,2,3],dtype=torch.float64)
```

[59]:
```
tensor([1., 2., 3.], dtype=torch.float64)
```

[60]:
```python
torch.tensor([1.4,4.2,5.3],dtype=torch.int64)
```

[60]:
```
tensor([1, 4, 5])
```

[61]:
```python
# using to()
x.to(torch.float64)
```

[61]:
```
tensor([[1., 3., 5.],
        [2., 4., 6.]], dtype=torch.float64)
```

Data Type	Dtype	Description
32-bit Floating Point	`torch.float32`	Standard floating-point type used for most deep learning tasks. Provides a balance between precision and memory usage.
64-bit Floating Point	`torch.float64`	Double-precision floating point. Useful for high-precision numerical tasks but uses more memory.
16-bit Floating Point	`torch.float16`	Half-precision floating point. Commonly used in mixed-precision training to reduce memory and computational overhead on modern GPUs.
BFloat16	`torch.bfloat16`	Brain floating-point format with reduced precision compared to `float16`. Used in mixed-precision training, especially on TPUs.
8-bit Floating Point	`torch.float8`	Ultra-low-precision floating point. Used for experimental applications and extreme memory-constrained environments (less common).
8-bit Integer	`torch.int8`	8-bit signed integer. Used for quantized models to save memory and computation in inference.
16-bit Integer	`torch.int16`	16-bit signed integer. Useful for special numerical tasks requiring intermediate precision.
32-bit Integer	`torch.int32`	Standard signed integer type. Commonly used for indexing and general-purpose numerical tasks.
64-bit Integer	`torch.int64`	Long integer type. Often used for large indexing arrays or for tasks involving large numbers.
8-bit Unsigned Integer	`torch.uint8`	8-bit unsigned integer. Commonly used for image data (e.g., pixel values between 0 and 255).
Boolean	`torch.bool`	Boolean type, stores `True` or `False` values. Often used for masks in logical operations.
Complex 64	`torch.complex64`	Complex number type with 32-bit real and 32-bit imaginary parts. Used for scientific and signal processing tasks.

Data Type	Dtype	Description
Complex 128	`torch.complex128`	Complex number type with 64-bit real and 64-bit imaginary parts. Offers higher precision but uses more memory.
Quantized Integer	`torch.qint8`	Quantized signed 8-bit integer. Used in quantized models for efficient inference.
Quantized Unsigned Integer	`torch.quint8`	Quantized unsigned 8-bit integer. Often used for quantized tensors in image-related tasks.

4 Mathematical operations

4.1 1. Scalar operation

[69]:
```
x = torch.rand(2,2)
x
```

[69]:
```
tensor([[0.3079, 0.6269],
        [0.8277, 0.6594]])
```

[74]:
```
# addition
x + 2

# substraction
x - 2

# multiplication
x * 3

# division
(x * 100)//3

# mod
((x*100)//3)%2
```

[74]:
```
tensor([[0., 0.],
        [1., 1.]])
```

[75]:
```
# power
x**2
```

[75]:
```
tensor([[0.0948, 0.3931],
        [0.6850, 0.4347]])
```

4.2 2. Element wise operations

[78]:
```
a = torch.rand(2,3)
b = torch.rand(2,3)

print(a)
print(b)
```

```
tensor([[0.4847, 0.9436, 0.3904],
        [0.2499, 0.3206, 0.9753]])
tensor([[0.7582, 0.6688, 0.2651],
        [0.2336, 0.5057, 0.5688]])
```

[79]:
```
# add
a + b
# sub
a - b
# multi
a * b
# div
a / b
# power
a ** b
# mod
a % b
```

[79]:
```
tensor([[0.4847, 0.2748, 0.1253],
        [0.0163, 0.3206, 0.4064]])
```

[81]:
```
c = torch.tensor([1, -2, 3, -4])
c
```

[81]:
```
tensor([ 1, -2,  3, -4])
```

[82]:
```
# absulote
torch.abs(c)
```

[82]:
```
tensor([1, 2, 3, 4])
```

[84]:
```
# neg
torch.neg(c)
```

[84]:
```
tensor([-1,  2, -3,  4])
```

[86]:
```
d = torch.tensor([1.9,3.5,9.3,2.6,8.4])
d
```

[86]:
```
tensor([1.9000, 3.5000, 9.3000, 2.6000, 8.4000])
```

[87]:
```
# round
torch.round(d)
```

[87]:
```
tensor([2., 4., 9., 3., 8.])
```

[90]:
```
# ceil
torch.ceil(d)
```

[90]:
```
tensor([ 2.,  4., 10.,  3.,  9.])
```

[91]:
```
# floor
torch.floor(d)
```

[91]:
```
tensor([1., 3., 9., 2., 8.])
```

[92]:
```
# clamp
torch.clamp(d, min=3, max=8)
```

[92]:
```
tensor([3.0000, 3.5000, 8.0000, 3.0000, 8.0000])
```

5 Reduction operation

[95]:
```
e = torch.randint(size=(2,3), low=0, high=10)
e
```

[95]:
```
tensor([[2, 2, 3],
        [7, 1, 5]])
```

[97]:
```
# sum
torch.sum(e)
```

[97]:
```
tensor(20)
```

[98]:
```
# sum along columns
torch.sum(e, dim=0)
```

[98]:
```
tensor([9, 3, 8])
```

[99]:
```
# sum along rows
torch.sum(e, dim=1)
```

[99]:
```
tensor([ 7, 13])
```

[100]:
```
# mean
torch.mean(e.float(), dim=1)
```

[100]:
```
tensor([2.3333, 4.3333])
```

[101]:
```
# mean along columns
torch.mean(e.float(), dim=0)
```

[101]:
```
tensor([4.5000, 1.5000, 4.0000])
```

[102]:
```
# median
torch.median(e.float())
```

[102]:
```
tensor(2.)
```

[103]:
```
# max and min
torch.max(e)
torch.min(e)
```

[103]:
```
tensor(1)
```

[104]:
```
# product
torch.prod(e)
```

[104]:
```
tensor(420)
```

[106]:
```
# Standard deviaiton
torch.std(e.float())
```

[106]:
```
tensor(2.2509)
```

[107]:
```
# variance
torch.var(e.float())
```

[107]:
```
tensor(5.0667)
```

[108]:
```
# argmax
torch.argmax(e)
```

[108]:
```
tensor(3)
```

[109]:
```
# argmin
torch.argmin(e)
```

[109]:
```
tensor(4)
```

6 Matrix Operations

[112]:
```
f = torch.randint(size=(2,3), low=0, high=10)
g = torch.randint(size=(3,2), low=0, high=10)

print(f)
print(g)
```

```
tensor([[3, 6, 7],
        [8, 1, 2]])
tensor([[3, 2],
        [7, 9],
        [3, 9]])
```

[115]:
```
# matrix multiplication
torch.matmul(f,g)
```

[115]:
```
tensor([[ 72, 123],
        [ 37,  43]])
```

[116]:
```
vector1 = torch.tensor([1,2,3])
vector2 = torch.tensor([4,5,6])

print(vector1)
print(vector2)
```

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

[117]:
```
# dot product
torch.dot(vector1, vector2)
```

[117]:
```
tensor(32)
```

[119]:
```
# transpose
print(f)
torch.t(f)
```

```
tensor([[3, 6, 7],
        [8, 1, 2]])
```

[119]:
```
tensor([[3, 8],
        [6, 1],
        [7, 2]])
```

[127]:
```
h = torch.randint(size=(3,3), low=0, high=10, dtype=torch.float32)
h
```

[127]:
```
tensor([[5., 0., 6.],
        [7., 7., 8.],
        [8., 9., 6.]])
```

[128]:
```python
# determinant
torch.det(h)
```

[128]:
```
tensor(-108.0000)
```

[129]:
```python
# inverse
torch.inverse(h)
```

[129]:
```
tensor([[ 0.2778, -0.5000,  0.3889],
        [-0.2037,  0.1667, -0.0185],
        [-0.0648,  0.4167, -0.3241]])
```

6.1 Comparison operations

[130]:
```python
i = torch.randint(size=(2,3), low=0, high=10)
j = torch.randint(size=(2,3), low=0, high=10)

print(i)
print(j)
```

```
tensor([[2, 9, 4],
        [5, 5, 3]])
tensor([[5, 0, 7],
        [4, 7, 1]])
```

[131]:
```python
# greater than
i > j
```

[131]:
```
tensor([[False,  True, False],
        [ True, False,  True]])
```

[132]:
```python
# less than
i < j
```

[132]:
```
tensor([[ True, False,  True],
        [False,  True, False]])
```

[133]:
```python
# equal to
i == j
```

[133]:
```
tensor([[False, False, False],
        [False, False, False]])
```

[134]:
```python
# not equal to
i != j
```

[134]:
```
tensor([[True, True, True],
        [True, True, True]])
```

[137]:
```
# greater than equal to
i >= j
```

[137]:
```
tensor([[False,  True, False],
        [ True, False,  True]])
```

[138]:
```
# less than equal to
i <= j
```

[138]:
```
tensor([[ True, False,  True],
        [False,  True, False]])
```

6.2 6. Special functions

[146]:
```
k = torch.randint(size=(2,3), low=0, high=10, dtype=torch.float32)
k
```

[146]:
```
tensor([[0., 8., 5.],
        [1., 3., 8.]])
```

[140]:
```
# log
torch.log(k)
```

[140]:
```
tensor([[1.0986, 1.7918, 2.1972],
        [1.9459,   -inf, 0.0000]])
```

[141]:
```
#  exp
torch.exp(k)
```

[141]:
```
tensor([[2.0086e+01, 4.0343e+02, 8.1031e+03],
        [1.0966e+03, 1.0000e+00, 2.7183e+00]])
```

[142]:
```
# sqrt
torch.sqrt(k)
```

[142]:
```
tensor([[1.7321, 2.4495, 3.0000],
        [2.6458, 0.0000, 1.0000]])
```

[143]:
```
# sigmoid
torch.sigmoid(k)
```

[143]:
```
tensor([[0.9526, 0.9975, 0.9999],
        [0.9991, 0.5000, 0.7311]])
```

[147]:
```
# softmax
torch.softmax(k, dim=0)
```

[147]:
```
tensor([[0.2689, 0.9933, 0.0474],
        [0.7311, 0.0067, 0.9526]])
```

[148]:
```
# relu
torch.relu(k)
```

[148]:
```
tensor([[0., 8., 5.],
        [1., 3., 8.]])
```

7 Inplace Operations

[149]:
```
m = torch.rand(2,3)
n = torch.rand(2,3)

print(m)
print(n)
```

```
tensor([[0.9355, 0.1430, 0.3933],
        [0.1124, 0.3087, 0.9973]])
tensor([[0.4257, 0.6890, 0.9657],
        [0.0257, 0.4205, 0.0656]])
```

[150]:
```
m + n
```

[150]:
```
tensor([[1.3612, 0.8320, 1.3590],
        [0.1382, 0.7292, 1.0629]])
```

[151]:
```
m.add_(n)
```

[151]:
```
tensor([[1.3612, 0.8320, 1.3590],
        [0.1382, 0.7292, 1.0629]])
```

[152]:
```
m
```

[152]:
```
tensor([[1.3612, 0.8320, 1.3590],
        [0.1382, 0.7292, 1.0629]])
```

[153]:
```
n
```

[153]:
```
tensor([[0.4257, 0.6890, 0.9657],
        [0.0257, 0.4205, 0.0656]])
```

[154]:
```
torch.relu(m)
```

[154]:
```
tensor([[1.3612, 0.8320, 1.3590],
        [0.1382, 0.7292, 1.0629]])
```

[158]:
```
m.relu_()
```

[158]:
```
tensor([[1.3612, 0.8320, 1.3590],
        [0.1382, 0.7292, 1.0629]])
```

[157]:
```
m
```

[157]:
```
tensor([[1.3612, 0.8320, 1.3590],
        [0.1382, 0.7292, 1.0629]])
```

8 Copying a tensor

[159]:
```
a = torch.rand(2,3)
a
```

[159]:
```
tensor([[0.4508, 0.0553, 0.3140],
        [0.7460, 0.9357, 0.8925]])
```

[160]:
```
b = a
```

[161]:
```
b
```

[161]:
```
tensor([[0.4508, 0.0553, 0.3140],
        [0.7460, 0.9357, 0.8925]])
```

[164]:
```
a[0][0] = 0
a, b
```

[164]:
```
(tensor([[0.0000, 0.0553, 0.3140],
         [0.7460, 0.9357, 0.8925]]),
 tensor([[0.0000, 0.0553, 0.3140],
         [0.7460, 0.9357, 0.8925]]))
```

[166]:
```
id(a) , id(b)
```

[166]:
```
(139145171801808, 139145171801808)
```

[168]:
```
b = a.clone()
a,b
```

[168]:
```
(tensor([[0.0000, 0.0553, 0.3140],
         [0.7460, 0.9357, 0.8925]]),
 tensor([[0.0000, 0.0553, 0.3140],
         [0.7460, 0.9357, 0.8925]]))
```

[169]:
```
id(a), id(b)
```

[169]:
```
(139145171801808, 139145170351120)
```

[171]:
```
a[0][0] = 9
a , b
```

[171]:
```
(tensor([[9.0000, 0.0553, 0.3140],
         [0.7460, 0.9357, 0.8925]]),
 tensor([[0.0000, 0.0553, 0.3140],
         [0.7460, 0.9357, 0.8925]]))
```