# Tensors

Tensors behave almost exactly the same way in PyTorch as they do in Torch.

Create a tensor of size (5 x 7) with uninitialized memory:

```
import torch
a = torch.empty(5, 7, dtype=torch.float)
```

Initialize a double tensor randomized with a normal distribution with mean=0, var=1:

```
a = torch.randn(5, 7, dtype=torch.double)
print(a)
print(a.size())
```

Out:

```
tensor([[ 0.2095, -1.5431, -0.1285, -0.3743, -0.4776, -0.4502,  0.6279],
        [ 0.9514, -1.7334, -0.6382, -0.3209, -0.2505, -1.1160, -1.0959],
        [-0.3199,  1.7155,  1.3164, -0.3394, -0.2693, -1.9001,  0.6337],
        [-0.2756, -1.3569, -0.2064,  2.4318, -1.1204, -0.8584,  0.1493],
        [-0.5671,  0.9550,  0.5539,  0.0953,  0.6315, -1.2639,  0.3836]], dtype=torch.float64)
torch.Size([5, 7])
```

**Note**

`torch.Size` is in fact a tuple, so it supports the same operations

## Inplace / Out-of-place

The first difference is that ALL operations on the tensor that operate in-place on it will have an `_` postfix. For example, `add` is the out-of-place version, and `add_` is the in-place version.

```
a.fill_(3.5)
# a has now been filled with the value 3.5

b = a.add(4.0)
# a is still filled with 3.5
# new tensor b is returned with values 3.5 + 4.0 = 7.5

print(a, b)
```

Out:

```
tensor([[ 3.5000,  3.5000,  3.5000,  3.5000,  3.5000,  3.5000,  3.5000],
        [ 3.5000,  3.5000,  3.5000,  3.5000,  3.5000,  3.5000,  3.5000],
        [ 3.5000,  3.5000,  3.5000,  3.5000,  3.5000,  3.5000,  3.5000],
        [ 3.5000,  3.5000,  3.5000,  3.5000,  3.5000,  3.5000,  3.5000],
        [ 3.5000,  3.5000,  3.5000,  3.5000,  3.5000,  3.5000,  3.5000]], dtype=torch.float64) tensor([[ 7.5000,  7.5000,  7.5000,  7.5000,  7.5000,  7.5000,  7.5000],
        [ 7.5000,  7.5000,  7.5000,  7.5000,  7.5000,  7.5000,  7.5000],
        [ 7.5000,  7.5000,  7.5000,  7.5000,  7.5000,  7.5000,  7.5000],
        [ 7.5000,  7.5000,  7.5000,  7.5000,  7.5000,  7.5000,  7.5000],
        [ 7.5000,  7.5000,  7.5000,  7.5000,  7.5000,  7.5000,  7.5000]], dtype=torch.float64)
```

Some operations like `narrow` do not have in-place versions, and hence, `.narrow_` does not exist. Similarly, some operations like `fill_` do not have an out-of-place version, so `.fill` does not exist.

## Zero Indexing

Another difference is that Tensors are zero-indexed. (In lua, tensors are one-indexed)

```
b = a[0, 3]  # select 1st row, 4th column from a
```

Tensors can be also indexed with Python's slicing

```
b = a[:, 3:5]  # selects all rows, 4th column and  5th column from a
```

## No camel casing

The next small difference is that all functions are now NOT camelCase anymore. For example `indexAdd` is now called `index_add_`

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

Out:

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

```
z = torch.empty(5, 2)
z[:, 0] = 10
z[:, 1] = 100
print(z)
```

Out:

```
tensor([[  10.,  100.],
        [  10.,  100.],
        [  10.,  100.],
        [  10.,  100.],
        [  10.,  100.]])
```

```
x.index_add_(1, torch.tensor([4, 0], dtype=torch.long), z)
print(x)
```

Out:

```
tensor([[ 101.,    1.,    1.,    1.,   11.],
        [ 101.,    1.,    1.,    1.,   11.],
        [ 101.,    1.,    1.,    1.,   11.],
        [ 101.,    1.,    1.,    1.,   11.],
        [ 101.,    1.,    1.,    1.,   11.]])
```

## Numpy Bridge

Converting a torch Tensor to a numpy array and vice versa is a breeze. The torch Tensor and numpy array will share their underlying memory locations, and changing one will change the other.

### Converting torch Tensor to numpy Array

```
a = torch.ones(5)
print(a)
```

Out:

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

```
b = a.numpy()
print(b)
```

Out:

```
[1. 1. 1. 1. 1.]
```

```
a.add_(1)
print(a)
print(b)   # see how the numpy array changed in value
```

Out:

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

### Converting numpy Array to torch Tensor

```
import numpy as np
a = np.ones(5)
b = torch.from_numpy(a)
np.add(a, 1, out=a)
print(a)
print(b)  # see how changing the np array changed the torch Tensor automatically
```

Out:

```
[2. 2. 2. 2. 2.]
tensor([ 2.,  2.,  2.,  2.,  2.], dtype=torch.float64)
```

All the Tensors on the CPU except a CharTensor support converting to NumPy and back.

## CUDA Tensors

CUDA Tensors are nice and easy in pytorch, and transfering a CUDA tensor from the CPU to GPU will retain its underlying type.

```
# let us run this cell only if CUDA is available
if torch.cuda.is_available():

    # creates a LongTensor and transfers it
    # to GPU as torch.cuda.LongTensor
    a = torch.full((10,), 3, device=torch.device("cuda"))
    print(type(a))
    b = a.to(torch.device("cpu"))
    # transfers it to CPU, back to
    # being a torch.LongTensor
```

Out:

```
<class 'torch.Tensor'>
```

**Total running time of the script:** ( 0 minutes 0.004 seconds)

Download Python source code: tensor_tutorial.py

Download Jupyter notebook: tensor_tutorial.ipynb