PyTorch Tutorial 1_Tensor Fundamentals

January 31, 2026

1 Scope of this notebook

This notebook introduces core PyTorch **tensor fundamentals**, including tensor creation from Python and NumPy objects, tensor shapes and dimensionality, data types (`dtype`), device placement, arithmetic operations, and in-place operations.

The focus is on building **correct intuition** for how tensors behave in PyTorch, with particular attention to common pitfalls such as silent type conversions, in-place mutation, and probabilistic operations.

This notebook is intended as an **introductory reference** for readers who are new to PyTorch tensors or who want to solidify their understanding before moving on to more advanced topics.

It does **not** cover automatic differentiation (autograd), neural network modules, or optimisation routines, which will be addressed in subsequent notebooks in this tutorial series.

Author: Angze Li

Last updated: 2026-01-31

Version: v1.0

[188]:
```
pip install torch
```

```
Requirement already satisfied: torch in /opt/anaconda3/lib/python3.11/site-
packages (2.10.0)
Requirement already satisfied: filelock in /opt/anaconda3/lib/python3.11/site-
packages (from torch) (3.13.1)
Requirement already satisfied: typing-extensions>=4.10.0 in
/opt/anaconda3/lib/python3.11/site-packages (from torch) (4.15.0)
Requirement already satisfied: sympy>=1.13.3 in
/opt/anaconda3/lib/python3.11/site-packages (from torch) (1.14.0)
Requirement already satisfied: networkx>=2.5.1 in
/opt/anaconda3/lib/python3.11/site-packages (from torch) (3.1)
Requirement already satisfied: jinja2 in /opt/anaconda3/lib/python3.11/site-
packages (from torch) (3.1.3)
Requirement already satisfied: fsspec>=0.8.5 in
/opt/anaconda3/lib/python3.11/site-packages (from torch) (2023.10.0)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in
/opt/anaconda3/lib/python3.11/site-packages (from sympy>=1.13.3->torch) (1.3.0)
```

```
Requirement already satisfied: MarkupSafe>=2.0 in
/opt/anaconda3/lib/python3.11/site-packages (from jinja2->torch) (2.1.3)
Note: you may need to restart the kernel to use updated packages.
```

[57]:
```python
import torch
import numpy as np
```

2 Initialising a tensor

2.1 Directly from data

[19]:
```python
data = [[1, 2],[3, 4]]
x_data = torch.tensor(data)
print(type(data), type(x_data))
print(data)
print(x_data)
```

```
<class 'list'> <class 'torch.Tensor'>
[[1, 2], [3, 4]]
tensor([[1, 2],
        [3, 4]])
```

2.2 From a NumPy array

[24]:
```python
x_ones = torch.ones_like(x_data) # retains the properties of x_data, but all␣
↪entries has scalar value of 1.
print(f"Ones Tensor: \n {x_ones} \n")

x_rand = torch.rand_like(x_data, dtype=torch.float) # retains the properties of␣
↪x_data, but all entries are filled in with random numbers.
print(f"Random Tensor: \n {x_rand} \n")
```

```
Ones Tensor:
 tensor([[1, 1],
        [1, 1]])

Random Tensor:
 tensor([[0.7313, 0.7006],
        [0.2860, 0.9104]])
```

[18]:
```python
np_array = np.array(data)
x_np = torch.from_numpy(np_array)
print(type(np_array), type(x_np))
print(np_array)
print(x_np)
```

```
<class 'numpy.ndarray'> <class 'torch.Tensor'>
[[1 2]
```

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

2.3 From another tensor

2.4 With random or constant values

[38]:
```python
shape = (2,3,)
rand_tensor = torch.rand(shape)
ones_tensor = torch.ones(shape)
zeros_tensor = torch.zeros(shape)

print(f"Random Tensor: \n {rand_tensor} \n")
print(f"Ones Tensor: \n {ones_tensor} \n")
print(f"Zeros Tensor: \n {zeros_tensor}")
```

```
Random Tensor:
 tensor([[0.6172, 0.8733, 0.0051],
        [0.6939, 0.4970, 0.8543]])

Ones Tensor:
 tensor([[1., 1., 1.],
        [1., 1., 1.]])

Zeros Tensor:
 tensor([[0., 0., 0.],
        [0., 0., 0.]])
```

- `shape = (2,3)` and `shape = (2,3,)` produce exactly the same tensor. The trailing comma is optional for tuples with 2 elements.

[40]:
```python
shape1 = (2, 3)
shape2 = (2, 3,)

torch.rand(shape1).shape == torch.rand(shape2).shape
```

[40]: `True`

2.5 3D tensor

We can start from **1D** tensor, which has the same shape as a vector:

[41]:
```python
x = torch.tensor([1, 2, 3])
x.shape == (3,)
```

[41]: `True`

2D tensor has the same shape as a table or matrix, with first number indicating the number of **rows** and second for number of **columns**:

[42]:
```python
x = torch.tensor([
    [1, 2, 3],
    [4, 5, 6]
])
x.shape == (2, 3)
```

[42]:
```
True
```

3D tensor is a stack of matrices. - The first number indicates the number of matrices - The second and third number indicate the row number and column number of each matrix

[43]:
```python
x = torch.rand((2, 3, 4))
print(x)
```

```
tensor([[[0.5022, 0.3114, 0.7381, 0.2250],
         [0.6941, 0.9313, 0.1761, 0.4454],
         [0.2353, 0.1207, 0.7735, 0.2856]],

        [[0.6181, 0.9139, 0.7393, 0.8570],
         [0.7942, 0.7370, 0.0470, 0.7061],
         [0.6873, 0.8154, 0.5302, 0.4817]]])
```

Indexing makes it obvious. Each index *peels off* one dimension.

[52]:
```python
print(x.shape)
print(x[0].shape)
print(x[0,0].shape)
print(x[0,0,0])     # scalar at [0,0,0]
```

```
torch.Size([2, 3, 4])
torch.Size([3, 4])
torch.Size([4])
tensor(0.5022)
```

3 Attributes of a Tensor

Tensor attributes describe their shape, datatype, and the device on which they are stored.

[53]:
```python
tensor = torch.rand(3,4)

print(f"Shape of tensor: {tensor.shape}")
print(f"Datatype of tensor: {tensor.dtype}")
print(f"Device tensor is stored on: {tensor.device}")
```

```
Shape of tensor: torch.Size([3, 4])
Datatype of tensor: torch.float32
Device tensor is stored on: cpu
```

3.1 float32

`float32` can represent: - each number in the tensor is stored as a 32-bit floating-point number, aka **single precision** - ~7 decimal digits of precision - numbers up to ~10^3

float32 is the **default** dtype for PyTorch tensors, since float32 is fast on CPUs and GPUs, and accurate enough for most ML tasks.

3.2 Device type

By default, PyTorch creates tensors on **cpu** because: - CPUs are always available - GPUs are optional - moving data to GPU has overhead

We need to manually specify if we wish to create the tensor on GPU. For **MacBook**, PyTorch uses **MPS (Metal Performance Shaders)**. We can check if MPS works by

[54]:
```
torch.backends.mps.is_available()
```

[54]:
```
True
```

And if it is `True`, we can transfer the tensor to GPU. But we almost always do not need for computation unless datasets get large or models are complex.

[55]:
```
device = torch.device("mps")
x = x.to(device)
```

It is also for this reason that we should have this setup at the start of the program: "`python import torch  torch.set_default_dtype(torch.double) device = torch.device("cpu")`

For Nvidia GPUs, we can use `tensor = torch.rand(3, 4).to("cuda")` or `tensor = torch.rand(3, 4, device="cuda")`, which only if works when `torch.cuda.is_available() == True`.

4 Operations on Tensors

4.1 Standard NumPy-like indexing and slicing

- Either `:` or `...` can be used for slicing.
- `tensor[:,1] = 0` can set the **whole column** with index 1 (second column) to be 0.

[85]:
```
tensor = torch.ones(4, 4)
print(f"First row: {tensor[0]}")
print(f"First column: {tensor[:, 0]}")
print(f"Last column: {tensor[..., -1]}")
print(f"First row, first column: {tensor[0,0]}")

tensor[:,1] = 0
print(tensor)
```

```
First row: tensor([1., 1., 1., 1.])
First column: tensor([1., 1., 1., 1.])
Last column: tensor([1., 1., 1., 1.])
```

```
First row, first column: 1.0
tensor([[1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])
```

4.2 Joining tensors

`torch.cat` can be used to concatenate a sequence of tensors along a given dimension. - Setting `dim=0` concatenates rows of the tensor. - Setting `dim=1` concatenates columns of the tensor. - If the tensor is 3D, setting `dim=2` concatenates "features" of the tensor. - `dim=-1` refers to the last dimension of the tensor, and so on for `dim=-2`, `dim=-3`, etc.

[91]:
```python
#Note that these tensors are 2D
tensor1 = torch.ones(4, 4)
tensor0 = torch.zeros(4, 4)
```

[92]:
```python
t2 = torch.cat([tensor1, tensor0], dim=0)
print(t2)
```

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

[93]:
```python
t1 = torch.cat([tensor1, tensor0], dim=1)
print(t1)
torch.cat([tensor1, tensor0], dim=1) == torch.cat([tensor1, tensor0], dim=-1)
```

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

[93]:
```
tensor([[True, True, True, True, True, True, True, True],
        [True, True, True, True, True, True, True, True],
        [True, True, True, True, True, True, True, True],
        [True, True, True, True, True, True, True, True]])
```

4.3 Matrix arithmetic computations of 2D tensor

- `tensor.T` returns the **transpose** of a tensor.
- The **matrix multiplication** can be computed by `tensor1 @ tensor2`.
- Another way of matrix multiplication is by `tensor1.matmul(tensor2)`, which returns the same result as the `@` method.

- The third way of matrix multiplication is by `torch.matmul(tensor1, tensor2, out=tensor3)`. In this way, the result can be stored in a specified way, i.e., in `tensor3`.
- Matrix multiplication is only defined when the inner dimensions of `tensor1` and `tensor2` are compatible; otherwise, PyTorch raises a **runtime error**.

[96]:
```
A = torch.tensor([
    [1., 2., 3.],
    [4., 5., 6.]
])   # shape (2, 3)

B = torch.tensor([
    [7.,  8.],
    [9., 10.],
    [11., 12.]
])   # shape (3, 2)
```

[106]:
```
# This computes the matrix multiplication between two tensors. y1, y2, y3 will␣
↪have the same value
y1 = A @ B
y2 = A.matmul(B)
torch.matmul(A, B, out=y3)

print(y1)
print(y2)
print(y3)
```

```
tensor([[ 58.,  64.],
        [139., 154.]])
tensor([[ 58.,  64.],
        [139., 154.]])
tensor([[ 58.,  64.],
        [139., 154.]])
```

4.4 Element-wise arithmetic computations of 2D tensor

- The **element-wise product** multiplies corresponding entries of two tensors with the same shape.
- The element-wise product can be computed using the `*` operator, i.e. `tensor1 * tensor2`.
- Another way to compute the element-wise product is by `tensor1.mul(tensor2)`, which returns the same result as the `*` operator.
- A third way to compute the element-wise product is by `torch.mul(tensor1, tensor2, out=tensor3)`. In this way, the result is stored directly in `tensor3`.
- Element-wise multiplication requires `tensor1` and `tensor2` to have the **same shape** (or be broadcastable); otherwise, PyTorch raises a **runtime error**.

[103]:
```
C = torch.tensor([
    [1., 2., 3.],
    [4., 5., 6.]
```

```
])   # shape (2, 3)

D = torch.tensor([
    [10., 20., 30.],
    [40., 50., 60.]
])   # shape (2, 3)
```

[105]:
```
# This computes the element-wise product. z1, z2, z3 will have the same value
z1 = C * D
z2 = C.mul(D)
torch.mul(C, D, out=z3)

print(z1)
print(z2)
print(z3)
```

```
tensor([[ 10.,  40.,  90.],
        [160., 250., 360.]])
tensor([[ 10.,  40.,  90.],
        [160., 250., 360.]])
tensor([[ 10.,  40.,  90.],
        [160., 250., 360.]])
```

4.5 Aggregation and single-element tensors

- `tensor.sum()` computes the sum, i.e. aggregates of all elements in `tensor`.
- The result of this opeartion is still a **0D tensor** (scalar tensor).

[107]:
```
agg = z1.sum()
print(agg, type(agg))
```

```
tensor(910.) <class 'torch.Tensor'>
```

- agg can be converted to a **numerical value** by `item()`:

Compare the result of the cell below and the cell above.

[108]:
```
agg_item = agg.item()
print(agg_item, type(agg_item))
```

```
910.0 <class 'float'>
```

4.6 In-place operations

Operations that store the result into the operand are called **in-place**.

Golden rule: Any PyTorch operation whose name ends with _ modifies the tensor in place.

4.6.1 Arithmetic operations

- Common in-place **arithmetic operations** include `tensor.add_(x)`, `tensor.sub_(x)`, `tensor.mul_(x)`, and `tensor.div_(x)`.
- In-place operations can be **memory-efficient**, as they avoid allocating new tensors.
- In-place operations should be used with caution when **automatic differentiation** is involved, as modifying tensors that participate in the computation graph may raise a **runtime error**.

[143]:
```python
D = torch.tensor([
    [10., 20., 30.],
    [40., 50., 60.]
])

print("Original tensor:\n", D)

E = D.clone()
E.add_(2)
print("After add_(2):\n", E)

E = D.clone()
E.sub_(1)
print("After sub_(1):\n", E)

E = D.clone()
E.mul_(2)
print("After mul_(2):\n", E)

E = D.clone()
E.div_(2)
print("After div_(2):\n", E)
```

```
Original tensor:
 tensor([[10., 20., 30.],
        [40., 50., 60.]])
After add_(2):
 tensor([[12., 22., 32.],
        [42., 52., 62.]])
After sub_(1):
 tensor([[ 9., 19., 29.],
        [39., 49., 59.]])
After mul_(2):
 tensor([[ 20.,  40.,  60.],
        [ 80., 100., 120.]])
After div_(2):
 tensor([[ 5., 10., 15.],
        [20., 25., 30.]])
```

4.6.2 Filling and assignment operations

- Common in-place **tensor filling and assignment operations** include `tensor.copy_(other_tensor)`, `tensor.fill_(constant)`, and `tensor.zero_()`.
- `tensor.copy_(other_tensor)` copies the contents of `other_tensor` into `tensor`, making the two tensors have identical values.
- `tensor.fill_(constant)` sets all entries of `tensor` to the specified constant value.
- `tensor.zero_()` sets all entries of `tensor` to zero.

[147]:
```
C = torch.tensor([
    [1., 2., 3.],
    [4., 5., 6.]
])

print("Original tensor:\n", C)

F = C.clone()
F.copy_(D)
print("After copy_(D):\n", F)

F = C.clone()
F.fill_(10)
print("After fill_(10):\n", F)

F = C.clone()
F.zero_()
print("After zero_():\n", F)
```

```
Original tensor:
 tensor([[1., 2., 3.],
        [4., 5., 6.]])
After copy_(D):
 tensor([[10., 20., 30.],
        [40., 50., 60.]])
After fill_(10):
 tensor([[10., 10., 10.],
        [10., 10., 10.]])
After zero_():
 tensor([[0., 0., 0.],
        [0., 0., 0.]])
```

4.6.3 Element-wise activation and transformation operations

- In-place **element-wise activation and transformation operations** replace the entries of a tensor by applying nonlinear activation functions or value transformations.
- Common in-place **element-wise activation and transformation operations** include `tensor.relu_()`, `tensor.sigmoid_()`, `tensor.clamp_(min, max)`, and `tensor.abs_()`.
- `tensor.relu_()` applies the **Rectified Linear Unit (ReLU) function** to each entry of `tensor` in place, setting all negative values to zero.

- `tensor.sigmoid_()` applies the **sigmoid function** to each entry of `tensor` in place, mapping all values to the interval `(0, 1)`.
- `tensor.clamp_(min, max)` restricts all entries of `tensor` to lie within the specified interval `[min, max]`.
- `tensor.abs_()` replaces each entry of `tensor` with its **absolute value**.

[148]:
```python
tensor = torch.tensor([
    [-2.0, -0.5,  0.0],
    [ 0.5,  1.0,  2.0]
])

print("Original tensor:\n", tensor)

E = tensor.clone()
E.relu_()
print("After relu_():\n", E)

E = tensor.clone()
E.sigmoid_()
print("After sigmoid_():\n", E)

E = tensor.clone()
E.clamp_(min=0, max=1)
print("After clamp_():\n", E)

E = tensor.clone()
E.abs_()
print("After abs_():\n", E)
```

```
Original tensor:
 tensor([[-2.0000, -0.5000,  0.0000],
        [ 0.5000,  1.0000,  2.0000]])
After relu_():
 tensor([[0.0000, 0.0000, 0.0000],
        [0.5000, 1.0000, 2.0000]])
After sigmoid_():
 tensor([[0.1192, 0.3775, 0.5000],
        [0.6225, 0.7311, 0.8808]])
After clamp_():
 tensor([[0.0000, 0.0000, 0.0000],
        [0.5000, 1.0000, 1.0000]])
After abs_():
 tensor([[2.0000, 0.5000, 0.0000],
        [0.5000, 1.0000, 2.0000]])
```

4.7 In-place random value generation operations

- In-place **random value generation operations** replace the entries of a tensor with randomly generated values.
- `tensor.normal_()` fills `tensor` with values drawn from a **normal (Gaussian) distribution** with mean 0 and standard deviation 1.
- `tensor.uniform_()` fills `tensor` with values drawn from a **uniform distribution** on the interval `[0, 1)`.
- `tensor.bernoulli_(p)` fills `tensor` with values drawn from a **Bernoulli distribution**, where each entry takes the value 1 with probability `p` and 0 with probability `1 - p`.

[176]:
```
tensor = torch.tensor([
    [-2.0, -0.5,  0.0],
    [ 0.5,  1.0,  2.0]
])

print("Original tensor:\n", tensor)

G = tensor.clone()
G.normal_()
print("After normal_():\n", G)

G = tensor.clone()
G.uniform_()
print("After uniform_():\n", G)

G = tensor.clone()
G.bernoulli_(0.3)
print("After bernoulli_(0.3):\n", G)

G = tensor.clone()
G.bernoulli_(0.7)
print("After bernoulli_(0.7):\n", G)
```

```
Original tensor:
 tensor([[-2.0000, -0.5000,  0.0000],
        [ 0.5000,  1.0000,  2.0000]])
After normal_():
 tensor([[-0.1153, -1.1834, -0.5861],
        [-1.2766,  0.8045, -0.1688]])
After uniform_():
 tensor([[0.0560, 0.1999, 0.4096],
        [0.0254, 0.5228, 0.8127]])
After bernoulli_(0.3):
 tensor([[0., 0., 1.],
        [0., 0., 0.]])
After bernoulli_(0.7):
 tensor([[0., 0., 1.],
        [1., 0., 1.]])
```

5 Bridging PyTorch with NumPy

[ ]:
```

```

5.1 Tensor to NumPy array

Tensors on the CPU and NumPy arrays can share their underlying memory locations, and **changing one will change the other**.

[178]:
```python
t = torch.ones(2,3)
print(f"t: {t}")
n = t.numpy()
print(f"n: {n}")
```

```
t: tensor([[1., 1., 1.],
        [1., 1., 1.]])
n: [[1. 1. 1.]
 [1. 1. 1.]]
```

A change in the tensor reflects in the NumPy array.

[179]:
```python
t.add_(10)
print(f"t: {t}")
print(f"n: {n}")
```

```
t: tensor([[11., 11., 11.],
        [11., 11., 11.]])
n: [[11. 11. 11.]
 [11. 11. 11.]]
```

5.2 NumPy array to Tensor

- Tensors with the **same shape and dimensionality** as the original NumPy array can be created using `torch.from_numpy()`.
- Note that `dtype=torch.float64`. This is beacause NumPy creates float64 arrays by default, and `torch.from_numpy()` **preserves** the NumPy dtype exactly.

[184]:
```python
n = np.ones(5)
print(f"n: {n}")
t = torch.from_numpy(n)
print(f"t: {t}")
```

```
n: [1. 1. 1. 1. 1.]
t: tensor([1., 1., 1., 1., 1.], dtype=torch.float64)
```

[185]:
```python
n = np.array([
    [1.0, 2.0, 3.0],
    [4.0, 5.0, 6.0]
])
print(f"n: {n}")
```

```
t = torch.from_numpy(n)
print(f"t: {t}")
```

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

Changes in the NumPy array also reflects in the tensor.

[187]:
```
n += 10
print(f"n: {n}")
print(f"t: {t}")
```

```
n: [[11. 12. 13.]
 [14. 15. 16.]]
t: tensor([[11., 12., 13.],
        [14., 15., 16.]], dtype=torch.float64)
```