Tensor

Ramin Zarebidoky (LiterallyTheOne)

12 Aug 2025

Tensor

What is Tensor

`Tensor` is the fundamental of `PyTorch`. Input, output, and the parameters of the model are all in `Tensors`. `Tensor` is like an array (`Numpy array`) but with more power.

- It can be run on `GPU`
- It supports automatic gradients

`Tensor` operations in `Pytorch` are pretty similar to `Numpy array`. So, if you have worked with `Numpy array` before, you are a step ahead.

In our `Hello world` example, we have created random data using `torch.rand((3, 8))` also we got the index of the maximum probability using `logits.argmax(1)`. In this tutorial, we are going to explain more about the main operations in `Tensor` and learn how to use them.

Create a Tensor

There are so many ways that we can create a `Tensor`. One of the simplest ways to create a tensor is as below:

```
data = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9],
]
t1 = torch.tensor(data)
print(t1)

"""
--------
output:
```

```
tensor([[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]])
"""
```

As you can see, we had a 2-dimensional matrix, and we gave it to `torch.tensor` as an argument and stored the result in a variable called `t1`. When we print `t1`, the output would be a `Tensor` of that matrix.

We can also create a `Tensor` by knowing its shape. For example, in our `Hello World` example, we created a random dataset using `torch.rand` function. We also have other functions that we can give the shape of `Tensor` to them and get a `Tensor`. You can see the examples in the code below:

```
s1 = torch.rand((3, 8))
print(s1)
print(s1.shape)

"""
--------
output:

tensor([[0.6667, 0.7057, 0.7670, 0.7719, 0.7298, 0.5729, 0.8281,
 ↪  0.5963],
        [0.1056, 0.5377, 0.3380, 0.4923, 0.0246, 0.8192, 0.3945,
 ↪  0.1150],
        [0.3885, 0.4211, 0.2655, 0.6766, 0.5082, 0.6465, 0.9499,
 ↪  0.2008]])
torch.Size([3, 8])
"""
```

```
s2 = torch.zeros((3, 8))
print(s2)
print(s2.shape)

"""
--------
output:

tensor([[0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0.]])
torch.Size([3, 8])
"""
```

```
s3 = torch.ones((3, 8))
```

```
print(s3)
print(s3.shape)

"""
--------
output:

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.]])
torch.Size([3, 8])
"""
```

In the above examples, we have used 3 functions:

- `torch.rand`: Creates random data
- `torch.ones`: Fills with one
- `torch.zeros`: Fills with zero

As you can see, the shape of all of them is `[3, 8]`, like the way that we gave them. (You can access the shape of a tensor by `.shape` variable)

We can also create a `Tensor` from other `Tensors`.

```
l1 = torch.zeros_like(t1)
print(l1)
print(l1.shape)

"""
--------
output:
tensor([[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]])
torch.Size([3, 3])
"""
```

The first `Tensor` that we created was called `t1` and its shape was `[3, 3]`. In the example above, we created a `Tensor` like `t1`, which is filled with zeros.

Attributes of a Tensor

`Tensor` has different attributes that define how it is stored. We mentioned one of them, which was `shape`. Now we learn two more of them, `dtype` and `device`.

- `shape`: shape of the tensor
- `dtype`: data type of the tensor
- `device`: device of the tensor, like `cpu` or `cuda` (for `gpu`)

```
print(f"shape: {t1.shape}")
print(f"dtype: {t1.dtype}")
print(f"device: {t1.device}")

"""
--------
output:

shape: torch.Size([3, 3])
dtype: torch.int64
device: cpu
"""
```

Control the device

To find if our system has any available accelerators, we can use the code below:

```
if torch.accelerator.is_available():
    device = torch.accelerator.current_accelerator()
else:
    device = "cpu"

print(device)

"""
--------
output:

mps
"""
```

The code above first checks if there are any accelerators like `cuda` or `mps` (for MacBook). Then puts the current accelerator in a variable called `device`. If there wasn't any available, the value of `device` would be `cpu`. In my case, the output is `mps`. If you run this code on `Google Colab` with the `GPU` on, you would get `cuda`.

We can change the device of any `Tensor` by using a function called `.to()`. For example:

```
t1 = t1.to(device)
print(t1.device)

"""
--------
output:
```

```
mps:0
"""
```

In the code above, we changed the device of the `Tensor` called `t1` to the current accelerator, which in my case is `mps`.

Operations on Tensor

The syntax of `Tensor` operations is pretty much like the `Numpy Arrays`. As you recall, we had a `Tensor` called `t1` that we cast it to run on `gpu`. `t1` was a 2D matrix with the shape of `[3, 3]` and the content of it was like below:

```
"""
tensor([[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]], device='mps:0')
"""
```

If we want to only select the first row of it, we can use the code below:

```
t1_first_row = t1[0]
print(t1_first_row)

"""
--------
output:

tensor([1, 2, 3], device='mps:0')
"""
```

If we want to select its first column, we can use the code below:

```
t1_first_column = t1[:, 0]
print(t1_first_column)

"""
--------
output:

tensor([1, 4, 7], device='mps:0')
"""
```

If we want to select a slice of that tensor, for example, the second row till the end, and the second column till the end, the code below would be useful:

```
t1_slice = t1[1:, 1:]
print(t1_slice)
```

```
"""
--------
output:

tensor([[5, 6],
        [8, 9]], device='mps:0')
"""
```

We can join (concatenate) two tensors using `torch.cat`. For example, let's make two 2D tensors and concatenate them.

```
c1 = torch.zeros((5, 4))
c2 = torch.ones((5, 2))

c3 = torch.concat((c1, c2), dim=1)
print(c3)

"""
--------
output:

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

We can transpose a tensor, using `.T`.

```
a1 = torch.tensor([
    [1, 2, 3],
    [4, 5, 6],
])
a1t = a1.T

print(a1)
print(a1t)

"""
--------
output:

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

```
        [3, 6]])

"""
```

We can do arithmetic operations on `Tensors` as well. For example, let's create 2 matrices and multiply them.

```
matrix_1 = torch.Tensor([
    [1.0, 2.0, 3.0],
    [4.0, 5.0, 6.0],
])

matrix_2 = torch.tensor([
    [1.0],
    [2.0],
    [3.0],
])

result = matrix_1 @ matrix_2
print(result)

"""
--------
output:

tensor([[14.],
        [32.]])
"""
```

As you can see, `1x1+2x2+3x3=1+4+9=14` and `4x1+5x2+6x3=4+10+18=32`.

Also, we can calculate the sum of a matrix using `.sum`.

```
sum_matrix_1 = matrix_1.sum()
print(sum_matrix_1)

"""
--------
output:

tensor(21.)
"""
```

In the `hello world` example, we used `argmax`. Now, let's use the `max` function, which calculates the maximum and the index of the maximum as well.

```
b1 = torch.tensor([
    [3, 1, 7, 2],
```

```
    [2, 4, 1, 3],
    [9, 1, 2, 5],
])

max_of_each_row = b1.max(dim=1)

print(max_of_each_row)

"""
--------
output:

torch.return_types.max(
values=tensor([7, 4, 9]),
indices=tensor([2, 1, 0]))
"""
```

As you can see, in the code above, the maximum number in the first row is `7`, and it is in the index of the `2` (we start with `0`) and so on.

Now, let's use the `argmax` function and compare the results.

```
argmax_of_each_row = b1.argmax(dim=1)

print(argmax_of_each_row)

"""
--------
output:

tensor([2, 1, 0])
"""
```

As you can see, the indices of both results are the same.

Conclusion

In this tutorial, we learned more about `Tensor`, which is the core concept of `PyTorch`. We learned how to create them, what their most important attributes are, how to control the device, also how to perform an operation on them. There are so many things that you can do with tensors, and these were only some of them to show the concept of a `Tensor`.