tensors

February 27, 2021

1 Tensors

In PyTorch, we use tensors to encode inputs and outputs of a model, as well as the model's parameters.

Tensors are similar to Numpy's ndarrays, except that tensors can run on GPUs or other hardware to accelerate computing.

[1]:
```
import torch
import numpy as np
```

1.1 Tensor Initialization

1.1.1 Directly from data

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

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

1.1.2 From a Numpy array

[3]:
```
np_array = np.array(data)
x_np = torch.from_numpy(np_array)
print(np_array, x_np)
```

```
[[1 2]
 [3 4]] tensor([[1, 2],
        [3, 4]], dtype=torch.int32)
```

1.1.3 From another tensor

The new tensor retains the properties (shape, datatype, etc.) of the argument tensor, unless explicitly overridden.

[4]:
```
x_ones = torch.ones_like(x_data)
print(f"Ones Tensor: \n {x_ones} \n")
```

```
x_rand = torch.rand_like(x_data, dtype=torch.float)
print(f"Random Tensor: \n {x_rand} \n")
```

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

Random Tensor:
 tensor([[0.2093, 0.5540],
        [0.4554, 0.2880]])
```

1.1.4 With random or constant values

Shape is a tuple of tensor dimensions.

[8]:
```
shape = (2, 3, 2, 5)
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} \n")
```

```
Random Tensor:
 tensor([[[[0.8613, 0.6913, 0.5987, 0.9573, 0.8078],
          [0.0586, 0.7480, 0.0604, 0.1908, 0.5255]],

         [[0.8492, 0.0957, 0.2768, 0.3856, 0.1886],
          [0.9923, 0.9237, 0.7892, 0.4579, 0.8463]],

         [[0.1012, 0.3221, 0.8305, 0.9297, 0.8396],
          [0.1169, 0.3381, 0.5853, 0.8617, 0.4281]]],


        [[[0.4221, 0.1436, 0.9533, 0.7004, 0.0499],
          [0.1768, 0.8002, 0.9216, 0.8986, 0.0479]],

         [[0.5647, 0.1131, 0.6868, 0.1815, 0.4398],
          [0.6489, 0.4839, 0.6928, 0.9092, 0.2308]],

         [[0.0053, 0.0978, 0.7919, 0.6107, 0.5375],
          [0.5823, 0.7241, 0.6325, 0.7930, 0.5152]]]])

Ones Tensor:
 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.],
          [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.]],

         [[1., 1., 1., 1., 1.],
          [1., 1., 1., 1., 1.]]]])

Zeros Tensor:
 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., 0.],
          [0., 0., 0., 0., 0.]]],


        [[[0., 0., 0., 0., 0.],
          [0., 0., 0., 0., 0.]],

         [[0., 0., 0., 0., 0.],
          [0., 0., 0., 0., 0.]],

         [[0., 0., 0., 0., 0.],
          [0., 0., 0., 0., 0.]]]])
```

1.2 Tensor Attributes

Tensors' shape, datatype, and the device on which they are stored.

[9]:
```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
```

1.3 Tensor Operations

Over 100: * Transposing * Indexing * Slicing * Mathematical Operations * Linear algebra * Random sampling

Each of them can be run on the GPU (at typically higher speeds than on a CPU).
We move our tensor to the GPU if available:

[11]:
```
if torch.cuda.is_available():
    tensor = tensor.to('cuda')

torch.cuda.is_available()
```

[11]:
```
False
```

1.3.1 Standard numpy-like indexing and slicing

[13]:
```
tensor = torch.ones(4, 4)
print(tensor)
tensor[:, 1] = 0
print(tensor)
```

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

1.3.2 Joining tensors

Concatenate a sequence of tensors along a given dimension. See also `torch.stack` for another tensor joining.

[14]:
```
t1 = torch.cat([tensor, tensor, tensor], dim=1)
print(t1)
```

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

[16]:
```
t2 = torch.cat([tensor, tensor, tensor], dim=0)
print(t2)
```

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

[17]:
```
t3 = torch.cat([tensor, tensor, tensor], dim=-1)
print(t3)
```

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

[18]:
```
t4 = torch.cat([tensor, tensor, tensor], dim=-2)
print(t4)
```

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

1.3.3 Multiplying tensors

- Element-wise product
- Matrix multiplication
- Matrix * Vector

[22]:
```
# Element-wise
print(f"tensor.mul(tensor)\n {tensor.mul(tensor)} \n")
```

```
# Alternative syntax
print(f"tensor*tensor \n {tensor * tensor} \n")

# Use torch
print(f"torch.mul(x, y) \n {torch.mul(tensor, tensor)}")
```

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

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

torch.mul(x, y)
 tensor([[1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])
```

[23]:
```
# Matrix multiplicaiton
print(f"tensor.matmul(tensor.T) \n {tensor.matmul(tensor.T)} \n")
# Alternative syntax
print(f"tensor @ tensor.T \n {tensor @ tensor.T} \n")

# Use torch
print(f"torch.mm(tensor, tensor) \n {torch.mm(tensor, tensor)}") # Needs the␣
 ↪size of two tensors fit
```

```
tensor.matmul(tensor.T)
 tensor([[3., 3., 3., 3.],
        [3., 3., 3., 3.],
        [3., 3., 3., 3.],
        [3., 3., 3., 3.]])

tensor @ tensor.T
 tensor([[3., 3., 3., 3.],
        [3., 3., 3., 3.],
        [3., 3., 3., 3.],
        [3., 3., 3., 3.]])

torch.mm(tensor, tensor)
 tensor([[3., 0., 3., 3.],
        [3., 0., 3., 3.],
```

```
        [3., 0., 3., 3.],
        [3., 0., 3., 3.]])
```

[27]:
```
# Use torch
print(f"torch.mv(tensor, vector) \n {torch.mv(tensor, vector)}")
```

```
torch.mv(tensor, vector)
 tensor([-1.6657, -1.6657, -1.6657, -1.6657])
```

1.3.4 In-place operations

Have a _ suffix. E.g., `x.copy_(y)`, `x.t_()`, will change `x`.

[28]:
```
print(tensor, "\n")
tensor.add_(5)
print(tensor)
```

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

tensor([[6., 5., 6., 6.],
        [6., 5., 6., 6.],
        [6., 5., 6., 6.],
        [6., 5., 6., 6.]])
```

1.4 Bridge with Numpy

Tensors on the CPU and Numpy arrays share their underlying memory locations.
Changing one will change the other.

1.4.1 Tensor to Numpy array

[29]:
```
t = torch.ones(5)
print(f"t: {t}")
n = t.numpy()
print(f"n: {n}")
```

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

A change in the tensor will also change the Numpy array.

[30]:
```
t.add_(1)
print(f"t: {t}")
print(f"n: {n}")
```

```
t: tensor([2., 2., 2., 2., 2.])
n: [2. 2. 2. 2. 2.]
```

1.4.2 Numpy array to Tensor

[31]:
```
n = np.ones(6)
t = torch.from_numpy(n)
print(f"n: {n}")
print(f"t: {t}")
```

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

Changes in the Numpy array also changes the tensor.

[32]:
```
np.add(n, 1, out=n)
print(f"n: {n}")
print(f"t: {t}")
```

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