Run in Microsoft Learn | Run in Google Cola | Downloa Note ook | View on GitHu

Learn the Basics || Quickstart || **Tensors** || Datasets & DataLoa ers || Transforms || Buil Mo el || Autogra || Optimization || Save & Loa Mo el

Tensors

Create On: Fe 10, 2021 | Last p ate : Jan 24, 2025 | Last Verifie : Nov 05, 2024

Tensors are a specialize ata structure that are very similar to arrays an matrices. In PyTorch, we use tensors to enco e the inputs an outputs of a mo el, as well as the mo el's parameters.

Tensors are similar to NumPy's n arrays, except that tensors can run on GP s or other har ware accelerators. In fact, tensors an NumPy arrays can often share the same un erlying memory, eliminating the nee to copy ata (see Bri ge with NumPy). Tensors are also optimize for automatic ifferentiation (we'll see more a out that later in the Autogra section). If you're familiar with n arrays, you'll e right at home with the Tensor API. If not, follow along!

```
import torch
import numpy as np
```

Initializing a Tensor

Tensors can e initialize in various ways. Take a look at the following examples:

Directly from ata

Tensors can e create irectly from ata. The ata type is automatically inferre .

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

From a NumPy array

Tensors can e create from NumPy arrays (an vice versa - see Bri ge with NumPy).

```
np_array = np.array(data)
x_np = torch.from_numpy(np_array)
```

From another tensor:

The new tensor retains the properties (shape, atatype) of the argument tensor, unless explicitly overri en.

```
x_ones = torch.ones_like(x_data) # retains the properties of x_data
print(f"Ones Tensor: \n {x_ones} \n")

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

Out:

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

Random Tensor:
 tensor([[0.8823, 0.9150],
        [0.3829, 0.9593]])
```

With ran om or constant values:

`shape` is a tuple of tensor imensions. In the functions elow, it etermines the imensionality of the output tensor.

```
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}")
```

Out:

```
Random Tensor:
 tensor([[0.3904, 0.6009, 0.2566],
        [0.7936, 0.9408, 0.1332]])

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

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

Attri utes of a Tensor

Tensor attri utes escri e their shape, atatype, an the evice on which they are store .

```
print(f"Shape of tensor: {tensor.shape}")

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

Out:

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

O erations on Tensors

Over 1200 tensor operations, inclu ing arithmetic, linear alge ra, matrix manipulation (transposing, in exing, slicing), sampling an more are comprehensively escri e here.

Each of these operations can e run on the CP an Accelerator such as C DA, MPS, MTIA, or XP . If you're using Cola , allocate an accelerator y going to Runtime > Change runtime type > GP .

By efault, tensors are create on the CP . We nee to explicitly move tensors to the accelerator using `.to` metho (after checking for accelerator availa ility). Keep in min that copying large tensors across evices can e expensive in terms of time an memory!

```
# We move our tensor to the current accelerator if available
if torch.accelerator.is_available():
    tensor = tensor.to(torch.accelerator.current_accelerator())
```

Try out some of the operations from the list. If you're familiar with the NumPy API, you'll fin the Tensor API a reeze to use.

Stan ar num y-like in exing an slicing:

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

Out:

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

Joining tensors You can use `torch.cat` to concatenate a sequence of tensors along a given imension. See also torch.stack, another tensor oining operator that is su tly ifferent from `torch.cat`.

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

Out:

```
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.]])
```

Arithmetic o erations

```
# This computes the matrix multiplication between two tensors. y1, y2, y3 will have the same value
# ``tensor.T`` returns the transpose of a tensor
y1 = tensor @ tensor.T
y2 = tensor.matmul(tensor.T)

y3 = torch.rand_like(y1)
torch.matmul(tensor, tensor.T, out=y3)


# This computes the element-wise product. z1, z2, z3 will have the same value
z1 = tensor * tensor
z2 = tensor.mul(tensor)

z3 = torch.rand_like(tensor)
torch.mul(tensor, tensor, out=z3)
```

Out:

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

Single-element tensors If you have a one-element tensor, for example y aggregating all values of a tensor into one value, you can convert it to a Python numerical value using `item()`:

```
agg = tensor.sum()
agg_item = agg.item()
print(agg_item, type(agg_item))
```

Out:

```
12.0 <class 'float'>
```

```
print(f"{tensor} \n")
tensor.add_(5)
print(tensor)
```

Out:

```
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.]])
```

• NOTE

In-place operations save some memory, ut can e pro lematic when computing erivatives ecause of an imme iate loss of history. Hence, their use is iscourage .

Bri ge with NumPy

Tensors on the CP an NumPy arrays can share their un erlying memory locations, an changing one will change the other.

Tensor to NumPy array

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

Out:

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

A change in the tensor reflects in the NumPy array.

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

Out:

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

NumPy array to Tensor

```
n = np.ones(5)
t = torch.from_numpy(n)
```

Changes in the NumPy array reflects in the tensor.

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

Out:

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

Total running time of the scri t: (0 minutes 0.022 secon s)

‹ Previous

Next ›

Rate this Tutorial ☆ ☆ ☆ ☆ ☆