```
# For tips on running notebooks in Google Colab, see
# https://pytorch.org/tutorials/beginner/colab
%matplotlib inline
```

Tensors

Tensors are a specialized data structure that are very similar to arrays and matrices. In PyTorch, we use tensors to encode the 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 specialized hardware to accelerate computing. If you're familiar with ndarrays, you'll be right at home with the Tensor API. If not, follow along in this quick API walkthrough.

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

Tensor Initialization

Tensors can be initialized in various ways. Take a look at the following examples:

Directly from data

Tensors can be created directly from data. The data type is automatically inferred.

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

From a NumPy array

Tensors can be created from NumPy arrays (and vice versa - see `bridge-to-np-label` {.interpreted-text role="ref"}).

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

From another tensor:

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

```
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")
```

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

Random Tensor:
 tensor([[0.0963, 0.3674],
        [0.4730, 0.3499]])
```

With random or constant values:

`shape` is a tuple of tensor dimensions. In the functions below, it determines the dimensionality 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}")
```

```
Random Tensor:
 tensor([[0.1786, 0.4696, 0.3313],
        [0.8208, 0.4230, 0.1553]])

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

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

Tensor Attributes

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

```
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
```

Tensor Operations

Over 100 tensor operations, including transposing, indexing, slicing, mathematical operations, linear algebra, random sampling, and more are comprehensively described here.

Each of them can be run on the GPU (at typically higher speeds than on a CPU). If you're using Colab, allocate a GPU by going to Edit > Notebook Settings.

```
# We move our tensor to the GPU if available
if torch.cuda.is_available():
  tensor = tensor.to('cuda')
  print(f"Device tensor is stored on: {tensor.device}")
```

```
Device tensor is stored on: cuda:0
```

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

Standard numpy-like indexing and slicing:

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

```
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 dimension. See also [torch.stack](), another tensor joining op that is subtly different from `torch.cat`.

```
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.]])
```

Multiplying tensors

```
# This computes the element-wise product
print(f"tensor.mul(tensor) \n {tensor.mul(tensor)} \n")
# Alternative syntax:
print(f"tensor * tensor \n {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.]])
```

This computes the matrix multiplication between two tensors

```
print(f"tensor.matmul(tensor.T) \n {tensor.matmul(tensor.T)} \n")
# Alternative syntax:
print(f"tensor @ tensor.T \n {tensor @ tensor.T}")
```

```
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.]])
```

In-place operations Operations that have a `_` suffix are in-place. For example: `x.copy_(y)`, `x.t_()`, will change `x`.

```
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.]])
```

NOTE:

In-place operations save some memory, but can be problematic when computing derivatives because of an immediate lossof history. Hence, their use is discouraged.

Bridge with NumPy {#bridge-to-np-label}

Tensors on the CPU and NumPy arrays can share their underlying memory locations, and 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}")
```

```
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}")
```

```
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}")
```

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