```
%matplotlib inline
```

What is PyTorch?

It's a Python-based scientific computing package targeted at two sets of audiences:

- A replacement for NumPy to use the power of GPUs
- a deep learning research platform that provides maximum flexibility and speed

Getting Started

Tensors are similar to NumPy's ndarrays, with the addition being that Tensors can also be used on a GPU to accelerate computing.

```
from __future__ import print_function
import torch
```

Note

An uninitialized matrix is declared, but does not contain definite known values before it is used. When an uninitialized matrix is created, whatever values were in the allocated memory at the time will appear as the initial values.

Construct a 5x3 matrix, uninitialized:

```
x = torch.empty(5, 3)
print(x)
```

```
tensor([[2.6446e-35, 0.0000e+00, 3.3631e-44],
        [0.0000e+00,        nan, 0.0000e+00],
        [1.1578e+27, 1.1362e+30, 7.1547e+22],
        [4.5828e+30, 1.2121e+04, 7.1846e+22],
        [9.2198e-39, 7.0374e+22, 9.7127e-36]])
```

Construct a randomly initialized matrix:

```
x = torch.rand(5, 3)
print(x)
```

```
tensor([[0.1827, 0.0768, 0.6706],
        [0.9754, 0.4748, 0.8971],
        [0.2254, 0.1489, 0.4967],
        [0.6707, 0.0099, 0.9771],
        [0.5360, 0.7776, 0.6815]])
```

Construct a matrix filled zeros and of dtype long:

```
x = torch.zeros(5, 3, dtype=torch.long)
print(x)
```

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

Construct a tensor directly from data:

```
x = torch.tensor([5.5, 3])
print(x)
```

```
tensor([5.5000, 3.0000])
```

or create a tensor based on an existing tensor. These methods will reuse properties of the input tensor, e.g. dtype, unless new values are provided by user

```
x = x.new_ones(5, 3, dtype=torch.double)      # new_* methods take in sizes
print(x)

x = torch.randn_like(x, dtype=torch.float)    # override dtype!
print(x)                                      # result has the same size
```

```
tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]], dtype=torch.float64)
tensor([[-0.7807,  0.4437,  1.0370],
        [ 0.4881,  0.0827, -1.7680],
        [ 2.1649,  1.1087, -0.5213],
        [ 0.5470, -1.6048,  0.9358],
        [ 1.9665, -0.2842,  2.1127]])
```

Get its size:

```
print(x.size())
```

```
torch.Size([5, 3])
```

Note

``torch.Size`` is in fact a tuple, so it supports all tuple operations.

Operations

There are multiple syntaxes for operations. In the following example, we will take a look at the addition operation.

Addition: syntax 1

```
y = torch.rand(5, 3)
print(x + y)
```

```
tensor([[-0.3843,  0.6752,  1.8237],
        [ 1.3242,  0.6569, -1.0896],
        [ 2.7236,  1.8565,  0.1990],
        [ 1.2512, -0.6661,  1.1746],
        [ 2.0730,  0.0769,  2.4763]])
```

Addition: syntax 2

```
print(torch.add(x, y))
```

```
tensor([[-0.3843,  0.6752,  1.8237],
        [ 1.3242,  0.6569, -1.0896],
        [ 2.7236,  1.8565,  0.1990],
        [ 1.2512, -0.6661,  1.1746],
        [ 2.0730,  0.0769,  2.4763]])
```

Addition: providing an output tensor as argument

```
result = torch.empty(5, 3)
torch.add(x, y, out=result)
print(result)
```

```
tensor([[-0.3843,  0.6752,  1.8237],
        [ 1.3242,  0.6569, -1.0896],
        [ 2.7236,  1.8565,  0.1990],
        [ 1.2512, -0.6661,  1.1746],
        [ 2.0730,  0.0769,  2.4763]])
```

Addition: in-place

```
# adds x to y
y.add_(x)
print(y)
```

```
tensor([[-0.3843,  0.6752,  1.8237],
        [ 1.3242,  0.6569, -1.0896],
        [ 2.7236,  1.8565,  0.1990],
        [ 1.2512, -0.6661,  1.1746],
        [ 2.0730,  0.0769,  2.4763]])
```

Note

Any operation that mutates a tensor in-place is post-fixed with an ``_``. For example: ``x.copy_(y)``, ``x.t_()``, will change ``x``.

You can use standard NumPy-like indexing with all bells and whistles!

```
print(x[:,1])
```

```
print(x[:, 1])
```

```
tensor([ 0.4437,  0.0827,  1.1087, -1.6048, -0.2842])
```

Resizing: If you want to resize/reshape tensor, you can use `torch.view`:

```
x = torch.randn(4, 4)
y = x.view(16)
z = x.view(-1, 8)  # the size -1 is inferred from other dimensions
print(x.size(), y.size(), z.size())
```

```
torch.Size([4, 4]) torch.Size([16]) torch.Size([2, 8])
```

If you have a one element tensor, use `.item()` to get the value as a Python number

```
x = torch.randn(1)
print(x)
print(x.item())
```

```
tensor([1.5290])
1.5289783477783203
```

Slicing

```
x = torch.rand((4,5))

print("x = {}".format(x))
print("x[1] = {}".format(x[1]))
print("x[2,3] = {}".format(x[2,3]))
print("x[0, 1:4] = {}".format(x[0,1:4]))
```

```
x = tensor([[0.5842, 0.3401, 0.9660, 0.8443, 0.2799],
        [0.0044, 0.2676, 0.9896, 0.6935, 0.4798],
        [0.6876, 0.2655, 0.9937, 0.3069, 0.8988],
        [0.0436, 0.7233, 0.5062, 0.2757, 0.3717]])
x[1] = tensor([0.0044, 0.2676, 0.9896, 0.6935, 0.4798])
x[2,3] = 0.30691617727279663
x[0, 1:4] = tensor([0.3401, 0.9660, 0.8443])
```

Multiplication

1) Elementwise multiplication

```
x = torch.tensor((1,2,3))
y = torch.tensor((4,5,6))

z = torch.mul(x,y) # same as x * y

print(z)
```

```
tensor([ 4, 10, 18])
```

2) Dot product

```
x = torch.tensor([1,2,3])
y = torch.tensor([4,5,6])

z = torch.dot(x,y)
print(z)
```

```
tensor(32)
```

3) Matrix multiplication

```
x = torch.tensor([[1,1],[2,2],[3,3]])
y = torch.tensor([[6,5],[4,3]])

z = torch.matmul(x,y)
print("x.size() = {}".format(x.shape))
print("y.size() = {}".format(y.shape))
print("z.size() = {}".format(z.shape))

print("z = {}" .format(z))
```

```
x.size() = torch.Size([3, 2])
y.size() = torch.Size([2, 2])
z.size() = torch.Size([3, 2])
z = tensor([[10,  8],
        [20, 16],
        [30, 24]])
```

Transpose

'torch.transpose' returns a tensor that is a transposed version of input.

The given dimension dim0 and dim1 are swapped.

```
x = torch.rand((4,2))
print("x.size() = {}".format(x.shape))

y = x.T
print("y.size() = {}".format(y.shape))

a = torch.rand((1,2,3,4))
print("a.size() = {}".format(a.shape))

b = torch.transpose(a, dim0=1, dim1=2)
print("b.size() = {}".format(b.shape))
```

```
x.size() = torch.Size([4, 2])
y.size() = torch.Size([2, 4])
a.size() = torch.Size([1, 2, 3, 4])
b.size() = torch.Size([1, 3, 2, 4])
```

Squeeze, Unsqueenze

'torch.squeeze' returns a tensor with all the dimensions of input of size 1 removed.

'torch.unsqueeze' returns a tensor with a dimension of size one inserted at the specified position.

```
x = torch.rand(1,2,3,1,4)
y = torch.unsqueeze(x, dim=1)
print("x.size() = {}".format(x.shape))
print("y.size() = {}".format(y.shape))

z = torch.squeeze(x)
print("z.size() = {}".format(z.shape))
```

```
x.size() = torch.Size([1, 2, 3, 1, 4])
y.size() = torch.Size([1, 1, 2, 3, 1, 4])
z.size() = torch.Size([2, 3, 4])
```

Cat

Concatenates the given sequence of seq tensors in the given dimensions.

```
x = torch.rand(2,3)
print(x)
print("x.size() = {}".format(x.shape))

y = torch.cat((x,x), dim=0)
print(y)
print("y.size() = {}".format(y.shape))

z = torch.cat((x,x), dim=1)
print(z)
print("z.size() = {}".format(z.shape))
```

```
tensor([[0.4262, 0.0682, 0.4141],
        [0.3715, 0.0162, 0.0012]])
x.size() = torch.Size([2, 3])
tensor([[0.4262, 0.0682, 0.4141],
        [0.3715, 0.0162, 0.0012],
        [0.4262, 0.0682, 0.4141],
        [0.3715, 0.0162, 0.0012]])
y.size() = torch.Size([4, 3])
tensor([[0.4262, 0.0682, 0.4141, 0.4262, 0.0682, 0.4141],
        [0.3715, 0.0162, 0.0012, 0.3715, 0.0162, 0.0012]])
z.size() = torch.Size([2, 6])
```

Read later:

100+ Tensor operations, including transposing, indexing, slicing, mathematical operations, linear algebra, random numbers, etc., are described `here <https://pytorch.org/docs/torch>`_.

NumPy Bridge

Converting a Torch Tensor to a NumPy array and vice versa is a breeze.

The Torch Tensor and NumPy array will share their underlying memory locations (if the Torch Tensor is on CPU), and changing one will change the other.

Converting a Torch Tensor to a NumPy Array ^^^

```
a = torch.ones(5)
print(a)
```

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

```
b = a.numpy()
print(b)
```

```
[1. 1. 1. 1. 1.]
```

See how the numpy array changed in value.

```
a.add_(1)
print(a)
print(b)
```

```
tensor([2., 2., 2., 2., 2.])
[2. 2. 2. 2. 2.]
```

Converting NumPy Array to Torch Tensor ^^^ See how changing the np array changed the Torch Tensor automatically

```
import numpy as np
a = np.ones(5)
b = torch.from_numpy(a)
np.add(a, 1, out=a)
print(a)
print(b)
```

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

All the Tensors on the CPU except a CharTensor support converting to NumPy and back.

CUDA Tensors

Tensors can be moved onto any device using the `.to` method.

```
# # let us run this cell only if CUDA is available
# # We will use ``torch.device`` objects to move tensors in and out of GPU
# if torch.cuda.is_available():
#     device = torch.device("cuda")          # a CUDA device object
#     y = torch.ones_like(x, device=device)  # directly create a tensor on GPU
#     x = x.to(device)                       # or just use strings ``.to("cuda")``
#     z = x + y
#     print(z)
#     print(z.to("cpu", torch.double))       # ``.to`` can also change dtype together!
```