Introduction to PyTorch

Introduction to Torch's tensor library

All of deep learning is computations on tensors, which are generalizations of a matrix that can be indexed in more than 2 dimensions. We will see exactly what this means in-depth later. First, lets look what we can do with tensors.

```
# Author: Robert Guthrie

import torch
import torch.autograd as autograd
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

torch.manual_seed(1)
```

Creating Tensors

Tensors can be created from Python lists with the torch.Tensor() function.

```
# torch.tensor(data) creates a torch.Tensor object with the given data.
V_data = [1., 2., 3.]
V = torch.tensor(V_data)
print(V)

# Creates a matrix
M_data = [[1., 2., 3.], [4., 5., 6]]
M = torch.tensor(M_data)
print(M)

# Create a 3D tensor of size 2x2x2.
T_data = [[[1., 2.], [3., 4.]],
          [[5., 6.], [7., 8.]]]
T = torch.tensor(T_data)
print(T)
```

Out:

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

        [[ 5.,  6.],
         [ 7.,  8.]]])
```

What is a 3D tensor anyway? Think about it like this. If you have a vector, indexing into the vector gives you a scalar. If you have a matrix, indexing into the matrix gives you a vector. If you have a 3D tensor, then indexing into the tensor gives you a matrix!

A note on terminology: when I say "tensor" in this tutorial, it refers to any torch.Tensor object. Matrices and vectors are special cases of torch.Tensors, where their dimension is 1 and 2 respectively. When I am talking about 3D tensors, I will explicitly use the term "3D tensor".

```
# Index into V and get a scalar (0 dimensional tensor)
print(V[0])
# Get a Python number from it
print(V[0].item())

# Index into M and get a vector
print(M[0])

# Index into T and get a matrix
print(T[0])
```

Out:

```
tensor(1.)
1.0
tensor([ 1.,  2.,  3.])
tensor([[ 1.,  2.],
        [ 3.,  4.]])
```

You can also create tensors of other datatypes. The default, as you can see, is Float. To create a tensor of integer types, try torch.LongTensor(). Check the documentation for more data types, but Float and Long will be the most common.

You can create a tensor with random data and the supplied dimensionality with torch.randn()

```
x = torch.randn((3, 4, 5))
print(x)
```

Out:

```
tensor([[[-1.5256, -0.7502, -0.6540, -1.6095, -0.1002],
         [-0.6092, -0.9798, -1.6091, -0.7121,  0.3037],
         [-0.7773, -0.2515, -0.2223,  1.6871,  0.2284],
         [ 0.4676, -0.6970, -1.1608,  0.6995,  0.1991]],

        [[ 0.8657,  0.2444, -0.6629,  0.8073,  1.1017],
         [-0.1759, -2.2456, -1.4465,  0.0612, -0.6177],
         [-0.7981, -0.1316,  1.8793, -0.0721,  0.1578],
         [-0.7735,  0.1991,  0.0457,  0.1530, -0.4757]],

        [[-0.1110,  0.2927, -0.1578, -0.0288,  0.4533],
         [ 1.1422,  0.2486, -1.7754, -0.0255, -1.0233],
         [-0.5962, -1.0055,  0.4285,  1.4761, -1.7869],
         [ 1.6103, -0.7040, -0.1853, -0.9962, -0.8313]]])
```

Operations with Tensors

You can operate on tensors in the ways you would expect.

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

Out:

```
tensor([ 5.,  7.,  9.])
```

See the documentation for a complete list of the massive number of operations available to you. They expand beyond just mathematical operations.

One helpful operation that we will make use of later is concatenation.

```
# By default, it concatenates along the first axis (concatenates rows)
x_1 = torch.randn(2, 5)
y_1 = torch.randn(3, 5)
z_1 = torch.cat([x_1, y_1])
print(z_1)

# Concatenate columns:
x_2 = torch.randn(2, 3)
y_2 = torch.randn(2, 5)
# second arg specifies which axis to concat along
z_2 = torch.cat([x_2, y_2], 1)
print(z_2)

# If your tensors are not compatible, torch will complain.  Uncomment to see the error
# torch.cat([x_1, x_2])
```

Out:

```
tensor([[-0.8029,  0.2366,  0.2857,  0.6898, -0.6331],
        [ 0.8795, -0.6842,  0.4533,  0.2912, -0.8317],
        [-0.5525,  0.6355, -0.3968, -0.6571, -1.6428],
        [ 0.9803, -0.0421, -0.8206,  0.3133, -1.1352],
        [ 0.3773, -0.2824, -2.5667, -1.4303,  0.5009]])
tensor([[ 0.5438, -0.4057,  1.1341, -0.1473,  0.6272,  1.0935,  0.0939,
          1.2381],
        [-1.1115,  0.3501, -0.7703, -1.3459,  0.5119, -0.6933, -0.1668,
         -0.9999]])
```

Reshaping Tensors

Use the .view() method to reshape a tensor. This method receives heavy use, because many neural network components expect their inputs to have a certain shape. Often you will need to reshape before passing your data to the component.

```
x = torch.randn(2, 3, 4)
print(x)
print(x.view(2, 12))  # Reshape to 2 rows, 12 columns
# Same as above.  If one of the dimensions is -1, its size can be inferred
print(x.view(2, -1))
```

Out:

```
tensor([[[ 0.4175, -0.2127, -0.8400, -0.4200],
         [-0.6240, -0.9773,  0.8748,  0.9873],
         [-0.0594, -2.4919,  0.2423,  0.2883]],

        [[-0.1095,  0.3126,  1.5038,  0.5038],
         [ 0.6223, -0.4481, -0.2856,  0.3880],
         [-1.1435, -0.6512, -0.1032,  0.6937]]])
tensor([[ 0.4175, -0.2127, -0.8400, -0.4200, -0.6240, -0.9773,  0.8748,
          0.9873, -0.0594, -2.4919,  0.2423,  0.2883],
        [-0.1095,  0.3126,  1.5038,  0.5038,  0.6223, -0.4481, -0.2856,
          0.3880, -1.1435, -0.6512, -0.1032,  0.6937]])
tensor([[ 0.4175, -0.2127, -0.8400, -0.4200, -0.6240, -0.9773,  0.8748,
          0.9873, -0.0594, -2.4919,  0.2423,  0.2883],
        [-0.1095,  0.3126,  1.5038,  0.5038,  0.6223, -0.4481, -0.2856,
          0.3880, -1.1435, -0.6512, -0.1032,  0.6937]])
```

Computation Graphs and Automatic Differentiation

The concept of a computation graph is essential to efficient deep learning programming, because it allows you to not have to write the back propagation gradients yourself. A computation graph is simply a specification of how your data is combined to give you the output. Since the graph totally specifies what parameters were involved with which operations, it contains enough information to compute derivatives. This probably sounds vague, so let's see what is going on using the fundamental flag `requires_grad`.

First, think from a programmers perspective. What is stored in the torch.Tensor objects we were creating above? Obviously the data and the shape, and maybe a few other things. But when we added two tensors together, we got an output tensor. All this output tensor knows is its data and shape. It has no idea that it was the sum of two other tensors (it could have been read in from a file, it could be the result of some other operation, etc.)

If `requires_grad=True`, the Tensor object keeps track of how it was created. Lets see it in action.

```
# Tensor factory methods have a ``requires_grad`` flag
x = torch.tensor([1., 2., 3], requires_grad=True)

# With requires_grad=True, you can still do all the operations you previously
# could
y = torch.tensor([4., 5., 6], requires_grad=True)
z = x + y
print(z)

# BUT z knows something extra.
print(z.grad_fn)
```

Out:

```
tensor([ 5.,  7.,  9.])
<AddBackward1 object at 0x7fb9f74d8ba8>
```

So Tensors know what created them. z knows that it wasn't read in from a file, it wasn't the result of a multiplication or exponential or whatever. And if you keep following z.grad_fn, you will find yourself at x and y.

But how does that help us compute a gradient?

```
# Lets sum up all the entries in z
s = z.sum()
print(s)
print(s.grad_fn)
```

Out:

```
tensor(21.)
<SumBackward0 object at 0x7fb9f74f9320>
```

So now, what is the derivative of this sum with respect to the first component of x? In math, we want

$$\frac{\partial s}{\partial x_0}$$

Well, s knows that it was created as a sum of the tensor z. z knows that it was the sum x + y. So

$$s = \overbrace{x_0 + y_0}^{z_0} + \overbrace{x_1 + y_1}^{z_1} + \overbrace{x_2 + y_2}^{z_2}$$

And so s contains enough information to determine that the derivative we want is 1!

Of course this glosses over the challenge of how to actually compute that derivative. The point here is that s is carrying along enough information that it is possible to compute it. In reality, the developers of Pytorch program the sum() and + operations to know how to compute their gradients, and run the back propagation algorithm. An in-depth discussion of that algorithm is beyond the scope of this tutorial.

Lets have Pytorch compute the gradient, and see that we were right: (note if you run this block multiple times, the gradient will increment. That is because Pytorch *accumulates* the gradient into the .grad property, since for many models this is very convenient.)

```
# calling .backward() on any variable will run backprop, starting from it.
s.backward()
print(x.grad)
```

Out:

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

Understanding what is going on in the block below is crucial for being a successful programmer in deep learning.

```
x = torch.randn(2, 2)
y = torch.randn(2, 2)
# By default, user created Tensors have ``requires_grad=False``
print(x.requires_grad, y.requires_grad)
z = x + y
# So you can't backprop through z
print(z.grad_fn)

# ``.requires_grad_( ... )`` changes an existing Tensor's ``requires_grad``
# flag in-place. The input flag defaults to ``True`` if not given.
x = x.requires_grad_()
y = y.requires_grad_()
# z contains enough information to compute gradients, as we saw above
z = x + y
print(z.grad_fn)
# If any input to an operation has ``requires_grad=True``, so will the output
print(z.requires_grad)

# Now z has the computation history that relates itself to x and y
# Can we just take its values, and **detach** it from its history?
new_z = z.detach()

# ... does new_z have information to backprop to x and y?
# NO!
print(new_z.grad_fn)
# And how could it? ``z.detach()`` returns a tensor that shares the same storage
# as ``z``, but with the computation history forgotten. It doesn't know anything
# about how it was computed.
# In essence, we have broken the Tensor away from its past history
```

Out:

```
False False
None
<AddBackward1 object at 0x7fb9f74d8550>
True
None
```

You can also stops autograd from tracking history on Tensors with requires_grad=True by wrapping the code block in `with torch.no_grad():`

```
print(x.requires_grad)
print((x ** 2).requires_grad)

with torch.no_grad():
    print((x ** 2).requires_grad)
```

Out:

```
True
True
False
```

Total running time of the script: (0 minutes 0.004 seconds)

Download Python source code: pytorch_tutorial.py

Download Jupyter notebook: pytorch_tutorial.ipynb

Gallery generated by Sphinx-Gallery