PyTorch 101

Deep Learning PhD Course
2017/2018

Marco Ciccone

Dipartimento di Informatica Elettronica e Bioingegneria
Politecnico di Milano

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

```
import torch
x = torch.Tensor(5, 3)
print(x)
```

Multiple syntaxes

Syntax 1

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

Syntax 2

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

Addition: providing an output tensor as argument

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

In-place

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

NOTE: all in-place operations have suffix _

NumPy Bridge

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

PyTorch => Numpy

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

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

Numpy => PyTorch

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

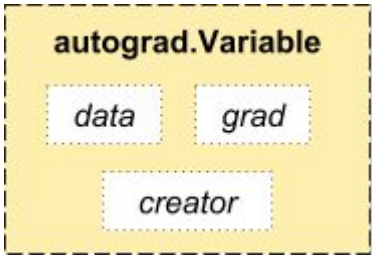
NOTE: The Torch Tensor and NumPy array will share their underlying memory locations, and changing one will change the other.

CUDA Tensors

```
# let us run this cell only if CUDA is available
if torch.cuda.is_available():
    x = x.cuda()
    y = y.cuda()
    x + y
```

Autograd (Automatic Differentiation)

The autograd package provides automatic differentiation for all operations on Tensors. It is a define-by-run framework, which means that your backprop is defined by how your code is run, and that every single iteration can be different.



autograd.Variable is the central class of the package.

It wraps a Tensor, and supports nearly all of operations defined on it.

Once you finish your computation you can call .backward() and have all the gradients computed automatically.

You can access the raw tensor through the .data attribute, while the gradient w.r.t. this variable is accumulated into .grad.

PyTorch Variables have the same API as PyTorch tensors: (almost) any operation you can do on a Tensor you can also do on a Variable; the difference is that autograd allows you to automatically compute gradients.

Autograd Example

```
import torch
from torch.autograd import Variable

x = Variable(torch.ones(2, 2), requires_grad=True)
print(x)

y = x + 2
print(y)
print(y.grad_fn)

z = y * y * 3
out = z.mean()
print(z, out)
out.backward()
print(x.grad)
```

Try it on jupyter!

$$o = \frac{1}{4}\sum_i z_i$$

$$z_i = 3(x_i + 2)^2$$

$$z_i\big|_{x_i=1} = 27$$

$$\frac{\partial o}{\partial x_i} = \frac{3}{2}(x_i + 2)$$

$$\frac{\partial o}{\partial x_i}\Big|_{x_i=1} = \frac{9}{2} = 4.5$$

Static vs Dynamic graph

Again we define a computational graph, and use automatic differentiation to compute gradients.

- **TF: Static graph**
  - The computational graph is defined once and then executed over and over again, possibly feeding different input data to the graph.
  - Graph is optimized upfront, before the execution.
  - Loops requires specific operations (tf.scan)
- **PyTorch: Dynamic graph**
  - Each forward pass defines a new computational graph.
  - Easy control flow (Imperative mode makes loops easier).
  - Easy to perform different operations for different data points.

torch.nn package

Neural network module.

Convenient way of encapsulating parameters, with helpers for moving them to GPU, exporting, loading, etc…

```
>>> Container example

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out),
)
```

Custom module

```
import torch
from torch.autograd import Variable
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel,
        # 6 output channels,
        # 5x5 square convolution kernel
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
```

```
def forward(self, x):
    # Max pooling over a (2, 2) window
    x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
    # If the size is a square
     you can only specify a single number
    x = F.max_pool2d(F.relu(self.conv2(x)), 2)
    x = x.view(-1, self.num_flat_features(x))
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    x = self.fc3(x)
    return x


def num_flat_features(self, x):
    # all dimensions except the batch dimension
    size = x.size()[1:]
    num_features = 1
    for s in size:
        num_features *= s
    return num_features
```

```
net = Net()
print(net)

>>>>>

Net(
  (conv1): Conv2d (1, 6, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d (6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=400, out_features=120)
  (fc2): Linear(in_features=120, out_features=84)
  (fc3): Linear(in_features=84, out_features=10)
)
```

The learnable parameters of a model are returned by **`net.parameters()`**

```
params = list(net.parameters())
print(len(params))
print(params[0].size())  # conv1's .weight
```

Mini-batches in torch.nn

`torch.nn` only supports mini-batches

The entire `torch.nn` package only supports inputs that are a mini-batch of samples, and not a single sample.

For example, `nn.Conv2d` will take in a 4D Tensor of `nSamples x nChannels x Height x Width`.

If you have a single sample, just use `input.unsqueeze(0)` to add a fake batch dimension.

Loss function

```
output = net(input)
target = Variable(torch.arange(1, 11))  # a dummy target, for example
criterion = nn.MSELoss()

loss = criterion(output, target)
print(loss)
```

Now, if you follow `loss` in the backward direction, using it's `.grad_fn` attribute, you will see a graph of computations that looks like this:

```
input -> conv2d -> relu -> maxpool2d -> conv2d -> relu -> maxpool2d
      -> view -> linear -> relu -> linear -> relu -> linear
      -> MSELoss
      -> loss
```

So, when we call `loss.backward()`, the whole graph is differentiated w.r.t. the loss, and all Variables in the graph will have their `.grad` Variable accumulated with the gradient.

BackProp

To backpropagate the error all we have to do is to `loss.backward()`.

You need to clear the existing gradients, **otherwise gradients will be accumulated to existing gradients**

Now we shall call `loss.backward()`, and have a look at **conv1's bias gradients** before and after the backward.

```
net.zero_grad()     # zeroes the gradient buffers of all parameters

print('conv1.bias.grad before backward')
print(net.conv1.bias.grad)

loss.backward()

print('conv1.bias.grad after backward')
print(net.conv1.bias.grad)
```

Gradients after backward

```
conv1.bias.grad before backward
Variable containing:
 0
 0
 0
 0
 0
 0
[torch.FloatTensor of size 6]

conv1.bias.grad after backward
Variable containing:
1.00000e-02 *
  7.4571
 -0.4714
 -5.5774
 -6.2058
  6.6810
  3.1632
[torch.FloatTensor of size 6]
```

Update the weights

The simplest update rule used in practice is the Stochastic Gradient Descent (SGD):

```
weight = weight - learning_rate * gradient
```

It can be implements this using simple python code:

```
learning_rate = 0.01
for f in net.parameters():
    f.data.sub_(f.grad.data * learning_rate)
```

Optimizers

However, as you use neural networks, you want to use various different update rules such as **SGD, Nesterov-SGD, Adam, RMSProp,** etc. To enable this, we built a small package: `torch.optim` that implements all these methods. Using it is very simple:

```
import torch.optim as optim

# create your optimizer
optimizer = optim.SGD(net.parameters(), lr=0.01)

# in your training loop:
optimizer.zero_grad()   # zero the gradient buffers
output = net(input)
loss = criterion(output, target)
loss.backward()
optimizer.step()    # Does the update
```

That was easier!
Let’s open Jupyter again!

Acknowledgements

Slides based on http://pytorch.org/tutorials/