The Torch.nn Module for Creating Neural and Convolutional Networks

Lecture Notes on Deep Learning

Avi Kak and Charles Bouman

Purdue University

Tuesday 4^{th} February, 2020 11:00

Preamble

The `torch.nn` module in PyTorch automates away for us several aspects of PyTorch programming.

As you already know from my Week 4 presentation, Autograd for automatic differentiation plays a central role in what PyTorch does. Ordinarily, in order to take advantage of Autograd, you must tell the system as to which tensors must be subject to the calculation of the partial derivatives by setting their `requires_grad` attribute to `True`. With the `torch.nn` module, you can move up a notch on the level of automation used. The container classes in this module can figure out on their own as to which tensors should be subject to automatic differentiation.

The `torch.nn` module is best appreciated through actual demonstrations of the code examples that use this module. So my plan is give those demos in class. The slides to follow show some of the code snippets from those demos.

Outline

Outline

Possible Levels of Automation in DL Programming

You could say that there exist three levels of automation in DL programming:

- At the lowest level, you manually construct each layer and also manually declare the interfaces between the successive layers.
- You only declare the different components of your network architecture. Subsequently, through explicit declarations you specify the order in which the data is supposed to flow through the different components.
- You eliminate the need for a separate declarations of the components and the order in which the data is supposed to flow through them by using a special container that figures out the order just on the basis of the sequence in which you placed the components in the container.

Obviously, only the 2nd and the 3rd steps listed above could be considered to be automated approaches to network construction. And `torch.nn`

To Elaborate on the Higest Level of Automation

In the context of DL programming, at its highest level of automation, a container class is something in which you drop each layer of your network, without explicitly declaring the layer-to-layer interconnections. The container then makes two assumption:

- The information will flow through the network in the order that is determined by the sequence of the layers you placed in the container.
- And, that you did not make any errors in the sizes input/output parameters associated with the different layers.
- This represents the highest level automation — in the sense that you are saved from have to explicitly declare the learnable parameters that would correspond the interconnections between the layers.
- With `torch.nn`, you achieve this level of automation with the `Sequential` container.

The Module Class in torch.nn

- As you can see in the documentation page at

  https://pytorch.org/docs/stable/nn.html

  practically all of the structures in `torch.nn` are derived from the class `torch.nn.Module`

- Here is a typical example (from the doc page) of how you create a network using `torch.nn`:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

- As the example shows, you declare the individual layers of your network in the constructor initialization code of your own class. Subsequently, it is your declarations in the `forward()` method of this

Outline

The DLStudio Module

- Since there is never a unique DL solution to a problem, that raises the question of how to experiment with different possibilities without getting lost amongst all the alternatives available.

- `DLStudio` is an attempt by me to address the above problem. What I have released so far is still an early version, which will hopefully grow into a more useful tool down the road.

- The main idea in `DLStudio` is to place all of the common code that you'd need to experiment with the different alternatives in the main definition of the class itself. Subsequently, any alternative solutions to a problem that one would want to experiment with would be placed in the user-defined inner classes of `DLStudio`.

DLStudio Even Allows for a String Based Description for a Network

- For networks that are not too complex, with DLStudio you could define them with a string like

```
convo_layers_config = "2x[128,7,7,1]-MaxPool(2) 1x[16,3,3,1]-MaxPool(2)"
fc_layers_config = [-1,1024,10]
```

- In the configuration string shown above, the basic component of a convolutional network is expressed as

```
nx[a,b,c,d]-MaxPool(k)
```

  where

```
n   =  num of this type of convo layer
a   =  number of out_channels                     [in_channels determined by prev layer]
b,c =  kernel for this layer is of size (b,c)     [b along height, c along width]
d   =  stride for convolutions
k   =  maxpooling over kxk patches with stride of k
```

Parsing the Configuration String and Building the Network

- Given a config string based description of a network, DLStudio call on the following method:

  `parse_config_string_for_convo_layers()`

  to parse the string.

- The output of the parser is supplied to the following method

  `build_convo_layers()`

  to actually build the network using the facilities provided by `torch.nn`

The Network Class for String Based Configs

- The call to `build_convo_layers()` only specifies the individual components of the network you have specified with your config string and the data-flow order for the components.
- The network itself is created by the piece of code shown below:

```
class Net(nn.Module):
    def __init__(self, convo_layers, fc_layers):
        super(DLStudio.Net, self).__init__()
        self.my_modules_convo = convo_layers
        self.my_modules_fc = fc_layers
    def forward(self, x):
        for m in self.my_modules_convo:
            x = m(x)
        x = x.view(x.size(0), -1)
        for m in self.my_modules_fc:
            x = m(x)
        return x
```

Outline

The Container Class `torch.nn.Sequential`

Here is the documentation page for this class:

```
class torch.nn.Sequential(*args)

    A sequential container. Modules will be added to it in the order they are passed
    in the constructor. Alternatively, an ordered dict of modules can also be passed in.

    To make it easier to understand, here is a small example:

    # Example of using Sequential
    model = nn.Sequential(
              nn.Conv2d(1,20,5),
              nn.ReLU(),
              nn.Conv2d(20,64,5),
              nn.ReLU()
            )

    # Example of using Sequential with OrderedDict
    model = nn.Sequential(OrderedDict([
              ('conv1', nn.Conv2d(1,20,5)),
              ('relu1', nn.ReLU()),
              ('conv2', nn.Conv2d(20,64,5)),
              ('relu2', nn.ReLU())
            ]))
```

A torch.nn.Sequential Based Demo in DLStudio

To see if the DLStudio class would work with any network that a user may want to experiment with, I copy-and-pasted the the network shown below from the following page by Zhenye at GitHub:

https://zhenye-na.github.io/2018/09/28/pytorch-cnn-cifar10.html

Here is the related code in DLStudio:

```
class Net(nn.Module):
    def __init__(self):
        super(DLStudio.ExperimentsWithSequential.Net, self).__init__()
        self.conv_seqn = nn.Sequential(
            # Conv Layer block 1:
            nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            # Conv Layer block 2:
            nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels=128, out_channels=128, kernel_size=3, padding=1),
```

Continued on the next slide ....

.... continued from the previous slide

```
        nn.ReLU(inplace=True),
        nn.MaxPool2d(kernel_size=2, stride=2),
        nn.Dropout2d(p=0.05),
        # Conv Layer block 3:
        nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, padding=1),
        nn.BatchNorm2d(256),
        nn.ReLU(inplace=True),
        nn.Conv2d(in_channels=256, out_channels=256, kernel_size=3, padding=1),
        nn.ReLU(inplace=True),
        nn.MaxPool2d(kernel_size=2, stride=2),
    )
    self.fc_seqn = nn.Sequential(
        nn.Dropout(p=0.1),
        nn.Linear(4096, 1024),
        nn.ReLU(inplace=True),
        nn.Linear(1024, 512),
        nn.ReLU(inplace=True),
        nn.Dropout(p=0.1),
        nn.Linear(512, 10)
    )
def forward(self, x):
    x = self.conv_seqn(x)
    # flatten
    x = x.view(x.size(0), -1)
    x = self.fc_seqn(x)
    return x
```