A First Introduction to Torch.nn for Designing Deep Networks and to DLStudio for Experimenting with Them

Lecture Notes on Deep Learning

Avi Kak and Charles Bouman

Purdue University

Tuesday 23rd February, 2021 22:10

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 container classes of the `torch.nn` module, you can move up a notch on the level of automation used. The container classes 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.

Preamble (contd.)

A second objective of this lecture is to introduce you to my Python module DLStudio.

The long-term goal for creating DLStudio is for it to serve as a framework that would make it easier to experiment with different network layouts, different training and testing protocols, and different ways for a user to interact with a network.

In its current state, DLStudio contains several inner classes, each devoted to a specific applications. These include classes for object detection and localization, semantic segmentation, text classification, etc.

More recently, DLStudio also acquired a co-class for experimenting with adversarial learning. The focus of the adversarial learning part of DLStudio at this time is solely on probabilistic data modeling.

Outline

Outline

Possible Levels of Automation in DL Programming

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

1. At the lowest level, you manually construct each layer and also manually declare the interfaces between the successive layers. [The one-neuron and multi-neuron networks implemented in the ComputationalGraphPrimer that you used for Homework 3 are examples of manually constructed networks.]

2. You declare the different components of your network architecture in the constructor of a network class and, subsequently, in a method of the class, through explicit declarations you specify the order in which the data is supposed to flow through the different components. [The `torch.nn` based network you created for your Homework 2 solution would be an example such a network.]

3. 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.

To Elaborate on the Highest 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 of input/output parameter tensors associated with the different layers.
- This represents the highest level automation — in the sense that you are saved from having to explicitly declare the learnable parameters that would correspond to the interconnections between the layers.
- With `torch.nn`, you achieve this level of automation with the `Sequential` container.

About the Possible Levels of Automation

- Of the three levels of automation listed on Slide 6, obviously, only the 2nd and the 3rd automation levels mentioned could be considered to be automated approaches to network construction. And `torch.nn` allows for both possibilities.
- Despite the fact that torch.nn makes it easy to write your code at the highest level of automation, **that does not make obsolete the practice of creating networks manually or through the second option on the previous slide.**
- Many aspects of research and development with neural networks require manually created networks or those that are created with the second approach.

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 class that tell the system how to route the data through the network.

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
```

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` calls 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 Inner Classes of DLStudio

To make it easy to experiment with different applications of deep learning, DLStudio contains a set of application-specific inner classes that are listed below:

For Experimenting with Skip Connections : Deep networks suffer from the problem of vanishing gradients that degrades their performance. The `SkipConnections` inner class allows you to experiment with different mitigation strategies for addressing this problem.

For Experimenting with Object Detection and Localization: You can experiment with networks for object detection and localization using the inner class `DetectAndLocalize`. In addition to classification, such networks must also localize them. The localization part requires regression for the coordinates of the bounding box that localize the object.

The Inner Classes of DLStudio (contd.)

For Experimenting with Semantic Segmentation: DLStudio comes with the inner class `SemanticSegmentation` for experimenting with semantic segmentation. Given a scene with multiple objects in it, the purpose of semantic segmentation is to assign correct labels to the pixels in the image, while, at the same time, grouping the pixels together that belong to the same object.

For Experimenting with Text Classification :

You can use the inner class `TextClassification` to experiment with recurrent neural networks (RNN) for analyzing user-feedback text for sentiment analysis.

Outline

A Co-Class of DLStudio on Adversarial Learning

- By a co-class I mean a class that resides at the same level of abstraction as the main DLStudio class. The co-class that is the subject of this section comes bundled with the DLStudio module.
- The co-class is meant for experimenting with adversarial learning and, as you might expect, its name is `AdversarialNetworks`. **Probabilistic Data Modeling is the main focus of this class.**
- `AdversarialNetworks` contains three different networks for such learning, two based on the Discriminator-Generator idea and one based on the Critic-Generator idea. The job of a Critc in adversarial learning is to estimate the distance between the probability distribution that describes the training dataset and the probability distribution that the Generator has learned up to the current moment.

Outline

DLStudio's Image Datasets for Deep Learning

- DLStudio comes with multiple small-sized training and testing image datasets for experimenting with its inner classes and with the co-class.
- The images for object detection and localization work are 32×32 and the images for semantic segmentation 64×64.
- As to the reason for small-sized images: I want the students to be able to experiment with the basic idioms of the programming needed for deep learning using their personal laptops that may only come with a rudimentary GPU if any at all.
- All of the datasets that you can use with the image-related inner classes are packaged in the following compressed tar archive:

```
datasets_for_DLStudio.tar.gz
```

DLStudio's Image Datasets for Deep Learning

- When you uncompress the archive mentioned at the bottom of the previous slide, you will gain access to the following PurdueShapes5 datasets:

```
PurdueShapes5-10000-train.gz                  ##  meant for object detection and localization
PurdueShapes5-1000-test.gz

PurdueShapes5-10000-train-noise-20.gz         ##  meant for detection and localization under noisy
PurdueShapes5-1000-test-noise-20.gz           ##    conditions

[there are two additional versions of the above dataset
 for different levels of noise, 50% and 80%]

PurdueShapes5MultiObject-10000-train.gz       ##  meant for semantic segmentation
PurdueShapes5MultiObject-1000-test.gz
```

- The large integer you see in the name of each archive is the number of images it contains. So each of the training datasets mentioned above contains 10,000 training images and the corresponding testing dataset 1000 images.

DLStudio's Dataset for Adversarial Learning

- DLStudio provides the following dataset archive for adversarial learning:

```
datasets_for_AdversarialNetworks.tar.gz
```

- Unpacking the main adversarial learning archive mentioned above gives you access to the following more specific archive:

```
PurdueShapes5GAN-20000.tar.gz
```

- The above mentioned archive consists of 20,000 images of size 64×64 for experimenting with the logic of adversarial learning.

DLStudio's Text Datasets for RNN-Based Learning

- For experimenting with text processing for, say, sentiment analysis, you would need to download the following archive:

```
text_datasets_for_DLStudio.tar.gz
```

- Unpacking this archive will give you the following datasets:

```
sentiment_dataset_train_40.tar.gz                vocab_size = 17,001
sentiment_dataset_test_40.tar.gz

sentiment_dataset_train_200.tar.gz               vocab_size = 43,285
sentiment_dataset_test_200.tar.gz

sentiment_dataset_train_400.tar.gz               vocab_size = 64,350
sentiment_dataset_test_400.tar.gz
```

- I extracted these datasets from the publicly available Amazon user-feedback archive for the year 2007.
- The integer number in the name of each dataset is the number of the positive and the number of the negative reviews I extracted from the Amazon archive. In general, the larger the number of reviews, the larger the vocabulary you have to contend with in your solution.