PyTorch Models and CNN Theory Part 2

PYTORCH CUSTOM MODELS

How are you feeling? Ready for some more Python? Last time around we read through the basic theory behind how convolutional neural networks work. We saw that CNNs extract key features from our visual data (images). Now we're ready to get going and see how everything falls into place as we apply the theory in real models.
In workshop 2 of this series, we introduced PyTorch, and showed you how to create datasets, data loaders, and bind it all together into a training and validation loop. In that article we were using pre-trained models which we fine-tuned with our own dataset to try and get better results. This time around we are delving into depth about how we can build a basic CNN network ourselves!

PyTorch Module Class

All PyTorch models are created by extending the nn.Module class. When we make our own models, the only method we need to define is the forward method. We define our layers in the __init__, and then define how an input is passed through these layers in the forward(). A basic template for a custom model is:

```
class MyModel(nn.Module):
 def __init__(self):
     super().__init__()
     # define layers here

 def forward(self, x):
   # define how our model passes
   # inputs through here
```

It's that easy. PyTorch takes care of everything else.

Basic Model

We now want to put some layers in our model. If we want a model with three linear layers, our code might end up looking something like this:

```
class MyModel(nn.Module):
 def __init__(self, input_size, output_size):
     super(MyModel, self).__init__()
     self.input_layer = nn.Linear(input_size, 64)
     self.hidden_layer = nn.Linear(64, 64)
     self.output_layer = nn.Linear(64, output_size)

 def forward(self, x):
     x = self.input_layer(x)
     x = nn.functional.relu(x)
     x = self.hidden_layer(x)
     x = nn.functional.relu(x)
     x = self.output_layer(x)

     return x
```

Let's break this down:

First, in our __init__, we define three layers:

- output layer

All three of these layers are fully connected layers, called Linear layers in Pytorch. Note that we have to add from torch import nn at the top of our Python file (or notebook).

In our forward method, we:

1. Pass the input through the input layer
2. Then through a ReLU function
3. Repeat for the hidden layer
4. Pass through the output layer
5. Return result

Now that we've created a model, we can use it:

```
inputs = 100
outputs = 3

sample_input = torch.rand(100)

# make our model
model = MyModel(inputs, outputs)

# feed some data through
output = model(sample_input)

print(output) # [0.3118, 0.3585, 0.3297]
```

In this example script, we:

1. Define the input and output size
2. Get some random input, of the same size
3. Initialise the model
4. Feed the random input through the model
5. Print the output

Although this is only a very basic toy example, we can see that our model is now able to pass inputs through.

CNN

Now that we've seen a basic PyTorch model, we can now try something a bit more advanced. We're going to code a CNN that should be able to perform well on image classification tasks, like CIFAR-10 or MNIST (two standard datasets for image classification).

As before, we start with our basic shell:

```
class MyCNN(nn.Module):
 def __init__(self, input_size, output_size):
     pass

 def forward(self, x):
   pass
```

We can then start adding in the layers required for a CNN.

Our model will need:

1. Convolutional Layer 1

4. Fully Connected Layer 1
5. Fully Connected Layer 2

Then, in your forward pass, the input should flow like so:

1. Convolutional Layer 1
2. ReLU
3. Max Pool
4. Convolutional Layer 2
5. ReLU
6. Max Pool (use the same max pool layer)
7. Flatten the input, so it can be passed through the fully connected layers
8. Fully Connected Layer 1
9. ReLU
10. Fully Connected Layer 2

This will all end up looking something like this:

```
class MyCNN(nn.Module):
 def __init__(self, num_classes):
     super().__init__()

     # define all layers

     # first convolutional layer
     self.conv1 = nn.Conv2d(in_channels=3, out_channels=7, kernel_size=5)

     # max pooling layer (has no weights, so can be reused)
     self.pool = nn.MaxPool2d(2, 2)

     # second convolutional layer
     self.conv2 = nn.Conv2d(in_channels=7, out_channels=14, kernel_size=5)

     # first fully connected layer. has input size equal to the size of the
     # flattened output of the convolutional blocks. Using a convolutional
     # size calculator, (like this fantastic one https://madebyollin.github.io/convn
     # the input size to the fully connected layer is 14 * 5 * 5
     self.fc1 = nn.Linear(14 * 5 * 5, 64)

     # second fully connected layer (called output for readability)
     self.output = nn.Linear(64, num_classes)
  def forward(self, x):
     # forward() defines how inputs are passed through the model

     # first convolution block
     x = self.conv1(x)
     x = nn.functional.relu(x)
     x = self.pool(x)

     # second convolution block
     x = self.conv2(x)
     x = nn.functional.relu(x)
     x = self.pool(x)

     # we get a set of feature maps coming out of the convolutional blocks
     # we need to flatten these feature maps into a single vector to be able
     # to pass through the fully connected layers
     # flatten(1) means we flatten out all dimensions except the first
```

```
    # first fully connected layer
    x = self.fcl1(x)
    x = nn.functional.relu(x)

    # second fully connected layer
    x = self.output(x)

    return x
```

Although this is much more code than in our previous example, we've now got the basis for a powerful model. With the addition of a few simple for loops, we can start to create very large CNN's, which (with a few modifications) are more or less state of the art when it comes to image processing.

To use the CNN we've just created, we can use the training loops, dataset, and data loaders from the previous article. We've assembled all of this into a notebook which you can run yourself here.

FINALLY

You've made it to the end of the fourth Introduction to deep learning article. We've introduced the background theory behind CNN's by talking about the layers involved in their use, and then gone through how to implement a CNN in PyTorch.

There was a lot of content in this article, so don't be discouraged if you have to read this article through again, and spend some extra time going through the code in the notebook. As always, you'll get the best possible understanding through practice, so start by modifying the notebook - break things, figure out why they broke, fix them, and you'll start gaining a better understanding of what's going on. If you're struggling with the core concepts, we have a Resources page with some of our recommended content which goes into a bit more detail about some of the theory.

Written by William Maclean (Deep Learning Lead)

Edited by Kamron Bhavnagri (Deep Learning Training Manager)

William Maclean (left) and Kamrom Bhavnagri (right)

Preview POST COMMENT...

NEXT
PyTorch Models and CNN Theory Part 1

ATTRIBUTIONS

Main banner gif: http://maximschoemaker.com/