Deep learning

4.6. Writing a PyTorch module

François Fleuret

https://fleuret.org/dlc/

UNIVERSITÉ DE GENÈVE

We now have all the bricks needed to build our first convolutional network from scratch. The last technical point is the tensor shape between layers.

Both the convolutional and pooling layers take as input batches of samples, each one being itself a 3d tensor $C \times H \times W$.

The output has the same structure, and tensors have to be explicitly reshaped before being forwarded to a fully connected layer.

```
>>> from torchvision.datasets import MNIST
>>> mnist = MNIST('./data/mnist/', train = True, download = True)
>>> d = mnist.train_data
>>> d.size()
torch.Size([60000, 28, 28])
>>> x = d.view(d.size(0), 1, d.size(1), d.size(2))
>>> x.size()
torch.Size([60000, 1, 28, 28])
>>> x = x.view(x.size(0), -1)
>>> x.size()
torch.Size([60000, 784])
```


Notes

Using `-1` for one of the `Tensor.view()`'s arguments automatically computes the proper value, given the original tensor size and the other dimensions.

A classical LeNet-like model could be:

Input sizes / operations	Nb. parameters	Nb. products
$1 \times 28 \times 28$		
`nn.Conv2d(1, 32, kernel_size=5)`	$32 \times (5^2 + 1) = 832$	$32 \times 24^2 \times 5^2 = 460{,}800$
$32 \times 24 \times 24$		
`F.max_pool2d(., kernel_size=3)`	0	0
$32 \times 8 \times 8$		
`F.relu(.)`	0	0
$32 \times 8 \times 8$		
`nn.Conv2d(32, 64, kernel_size=5)`	$64 \times (32 \times 5^2 + 1) = 51{,}264$	$32 \times 64 \times 4^2 \times 5^2 = 819{,}200$
$64 \times 4 \times 4$		
`F.max_pool2d(., kernel_size=2)`	0	0
$64 \times 2 \times 2$		
`F.relu(.)`	0	0
$64 \times 2 \times 2$		
`x.view(-1, 256)`	0	0
256		
`nn.Linear(256, 200)`	$200 \times (256 + 1) = 51{,}400$	$200 \times 256 = 51{,}200$
200		
`F.relu(.)`	0	0
200		
`nn.Linear(200, 10)`	$10 \times (200 + 1) = 2{,}010$	$10 \times 200 = 2{,}000$
10		

Total 105,506 parameters and 1,333,200 products for the forward pass.

Notes

A convolutional layer with a kernel of size 5×5 with a single input channel has, for every output channel 25 parameters for the filter, and 1 bias, so $5^2 + 1$ parameters in total.
After the first two convolutional layers, the resulting tensor for each sample is 64×2, it can hence be reshaped as a row vector of dimension 256.
The overall network goes from a $1 \times 28 \times 28$ image to a vector of 10 elements.
Such a small network inputting small images requires more than 1 million products per sample to be computed: this gives an idea of why we need GPUs and very fast processors.

Creating a module

PyTorch offers a sequential container module `torch.nn.Sequential` to build simple architectures.

For instance a MLP with a 10 dimension input, 2 dimension output, ReLU activation and two hidden layers of dimensions 100 and 50 can be written as:

```
model = nn.Sequential(
    nn.Linear(10, 100), nn.ReLU(),
    nn.Linear(100, 50), nn.ReLU(),
    nn.Linear(50, 2)
)
```

However for any model of reasonable complexity, the best is to write a sub-class of `torch.nn.Module`.

To create a `Module`, one has to inherit from the base class and implement the constructor `__init__(self, ...)` and the forward pass `forward(self, x)`.

```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=5)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=5)
        self.fc1 = nn.Linear(256, 200)
        self.fc2 = nn.Linear(200, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), kernel_size=3, stride=3))
        x = F.relu(F.max_pool2d(self.conv2(x), kernel_size=2, stride=2))
        x = x.view(-1, 256)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

Inheriting from `torch.nn.Module` provides many mechanisms implemented in the superclass.

First, the `(...)` operator is redefined to call the `forward(...)` method and run additional operations. The forward pass should be executed through this operator and not by calling `forward` explicitly.

Using the class `Net` we just defined

```
model = Net()
input = torch.randn(12, 1, 28, 28)
output = model(input)
print(output.size())
```

prints

```
torch.Size([12, 10])
```

Also, the `Parameter`s added as class attributes, or from modules added as class attributes, are seen by `Module.parameters()`.

```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=5)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=5)
        self.fc1 = nn.Linear(256, 200)
        self.fc2 = nn.Linear(200, 10)
/.../

model = Net()

for n, k in model.named_parameters():
    print(n, k.size())
```

prints

```
conv1.weight torch.Size([32, 1, 5, 5])
conv1.bias torch.Size([32])
conv2.weight torch.Size([64, 32, 5, 5])
conv2.bias torch.Size([64])
fc1.weight torch.Size([200, 256])
fc1.bias torch.Size([200])
fc2.weight torch.Size([10, 200])
fc2.bias torch.Size([10])
```

Parameters added in dictionaries or arrays are not seen.

```
class Buggy(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv = nn.Conv2d(1, 32, kernel_size=5)
        self.param = Parameter(torch.zeros(123, 456))
        self.other_stuff = [ nn.Linear(543, 21) ]

model = Buggy()

for k in model.parameters():
    print(k.size())
```

prints

```
param torch.Size([123, 456])
conv.weight torch.Size([32, 1, 5, 5])
conv.bias torch.Size([32])
```

A simple option is to add modules in a `torch.nn.ModuleList`, which is a list of modules properly dealt with by PyTorch's machinery.

```
class NotBuggy(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv = nn.Conv2d(1, 32, kernel_size=5)
        self.param = Parameter(torch.zeros(123, 456))
        self.other_stuff = nn.ModuleList()
        self.other_stuff.append(nn.Linear(543, 21))

model = NotBuggy()

for n, k in model.named_parameters():
    print(n, k.size())
```

prints

```
param torch.Size([123, 456])
conv.weight torch.Size([32, 1, 5, 5])
conv.bias torch.Size([32])
other_stuff.0.weight torch.Size([21, 543])
other_stuff.0.bias torch.Size([21])
```

As long as you use autograd-compliant operations, the backward pass is implemented automatically.

This is crucial to allow the optimization of the `Parameter`s with gradient descent.