Ch. 5: Deep Learning Computation Methods

Joe Adamo

We’ve covered how machine learning works

Now we’ll cover how to actually implement it

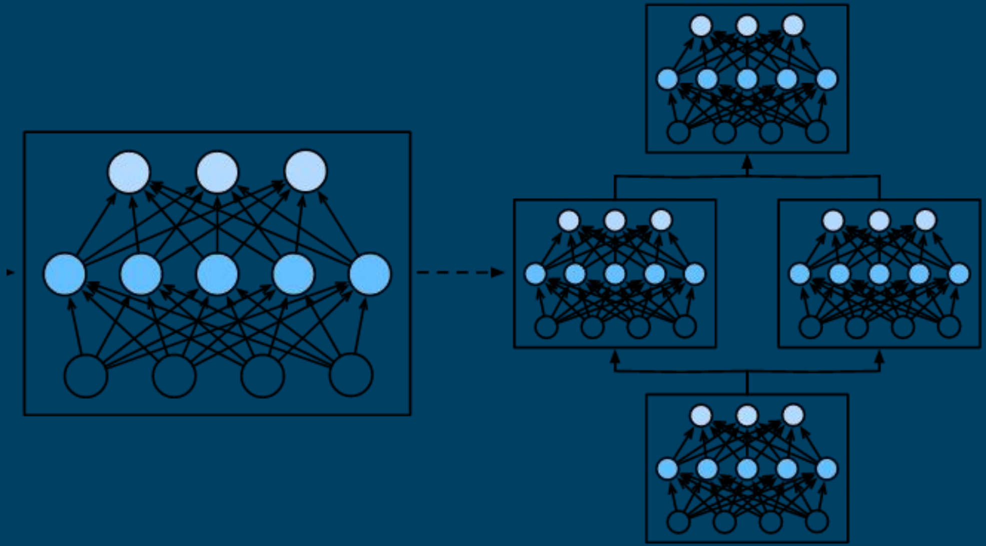
Outline

- Quick primer on PyTorch
- Using Blocks and Layers to build more complex networks
- Building custom layers
- Manipulating network parameters
- Saving and loading networks to file
- How to use GPU(s) in deep learning computation

All code snippets shown are also available on github (Chapter-5-example-code.ipynb)

Defining Blocks of Layers

- Defining each layer individually gets tedious
  - Ex: ResNet:152 has 152 individual layers
- Defining blocks of layers can more easily build complex networks
  - Outputs of one block -> inputs of another block
  - Can combine different types of blocks



Defining Blocks of Layers - Example

Define a class (inheriting from nn.Module) to create a new block

Need to define a constructor

Setup block structure

Need to define a forward propagation function

```
class MLP(nn.Module):
    # Declare a layer with model parameters. Here, we declare two fully
    # connected layers
    def __init__(self):
        # Call the constructor of the `MLP` parent class `Module` to perform
        # the necessary initialization. In this way, other function arguments
        # can also be specified during class instantiation, such as the model
        # parameters, `params` (to be described later)
        super().__init__()
        self.hidden = nn.Linear(20, 256)  # Hidden layer
        self.out = nn.Linear(256, 10)  # Output layer

    # Define the forward propagation of the model, that is, how to return the
    # required model output based on the input `X`
    def forward(self, X):
        # Note here we use the funtional version of ReLU defined in the
        # nn.functional module.
        return self.out(F.relu(self.hidden(X)))
```

Why don’t we need to define a backward-propagation function?

How you initialize your network changes

Without blocks

```
net = nn.Sequential(nn.Linear(20, 256), nn.ReLU(), nn.Linear(256, 10))

X = torch.rand(2, 20)
net(X)
```

```
tensor([[-0.0274,  0.0033,  0.2134, -0.0805, -0.1224,  0.1400,  0.0997, -0.1026,
          0.0693, -0.0095],
        [-0.1082,  0.0152,  0.1658, -0.1724, -0.2302,  0.3031, -0.0847, -0.1288,
          0.0258, -0.0049]], grad_fn=<AddmmBackward>)
```

With blocks

```
net = MLP()
net(X)
```

```
tensor([[-0.0072, -0.1604, -0.2128, -0.0826, -0.0159,  0.1213, -0.2071,  0.1703,
          0.0339, -0.0438],
        [ 0.0079, -0.1232, -0.1938, -0.0976, -0.0303,  0.0733, -0.3295,  0.1253,
         -0.0089, -0.1042]], grad_fn=<AddmmBackward>)
```

Advantage becomes apparent when using more complex networks

Custom Layers

- Using blocks allows us to easily make custom layers
  - Useful for implementing layers specially designed for specific problems

This example has no tunable parameters

```
class CenteredLayer(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, X):
        return X - X.mean()
```

This example explicitly defines a normal layer

```
class MyLinear(nn.Module):
    def __init__(self, in_units, units):
        super().__init__()
        self.weight = nn.Parameter(torch.randn(in_units, units))
        self.bias = nn.Parameter(torch.randn(units,))
    def forward(self, X):
        linear = torch.matmul(X, self.weight.data) + self.bias.data
        return F.relu(linear)
```

Accessing Parameters

- Can be useful for debugging / visualizing
- Several methods based on needs

Access from the whole model
(specify specific entry you want)

```
net.state_dict()['2.bias'].data
```

```
tensor([0.2390])
```

Targeted parameter access

```
print(type(net[2].bias))
print(net[2].bias)
print(net[2].bias.data)
```

```
<class 'torch.nn.parameter.Parameter'>
Parameter containing:
tensor([0.2390], requires_grad=True)
tensor([0.2390])
```

Access from an entire layer with state_dict()

```
print(net[2].state_dict())
```

```
OrderedDict([('weight', tensor([[-0.0172, -0.2535, -0.1108, -0.2766, -0.1250, -0.2304, -0.0741, -0.0548]])), ('bias', tensor([0.2390]))])
```

Initializing Parameters

- Defining a network automatically initializes parameters
- We can manually gives more control to initialization
    - Several in-built functions
    - Can also create custom methods

```
def xavier(m):
    if type(m) == nn.Linear:
        nn.init.xavier_uniform_(m.weight)
def init_42(m):
    if type(m) == nn.Linear:
        nn.init.constant_(m.weight, 42)

net[0].apply(xavier)
net[2].apply(init_42)
print(net[0].weight.data[0])
print(net[2].weight.data)
```

```
tensor([-0.4839, -0.6999, -0.3319,  0.6275])
tensor([[42., 42., 42., 42., 42., 42., 42., 42.]])
```

```
def init_normal(m):
    if type(m) == nn.Linear:
        nn.init.normal_(m.weight, mean=0, std=0.01)
        nn.init.zeros_(m.bias)
net.apply(init_normal)
net[0].weight.data[0], net[0].bias.data[0]
```

```
(tensor([-0.0005, -0.0046, -0.0179,  0.0098]), tensor(0.))
```

Sharing Parameters

- Sometimes you want multiple layers to share same parameters
  - Changing params in one layer changes them in the others
- Pass named layer to nn.Sequential() multiple times
- How does that affect the gradient?
  - Gradient of each layer is added in backpropagation

```
# We need to give the shared layer a name so that we can refer to its
# parameters
shared = nn.Linear(8, 8)
net = nn.Sequential(nn.Linear(4, 8), nn.ReLU(),
                    shared, nn.ReLU(),
                    shared, nn.ReLU(),
                    nn.Linear(8, 1))
```

Basic File I/O

- Useful for checkpointing while training or saving results
- Can save individual parameter lists or full networks
  - Saving network only saves the parameters, not the structure

Tensors: Just like numpy

```
torch.save(x, 'x-file')
```

```
x2 = torch.load('x-file')
```

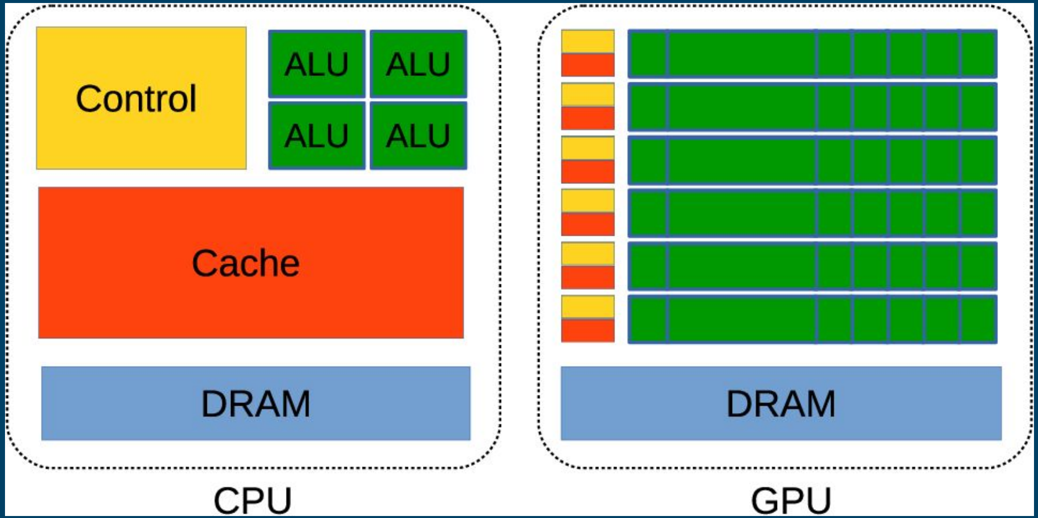
Full Network: Have to also keep track of model structure

```
torch.save(net.state_dict(), 'mlp.params')
```

```
clone = MLP()
clone.load_state_dict(torch.load('mlp.params'))
clone.eval()
```

GPUs vs CPUs

- CPU - Can run wide variety of instructions slowly
- GPU - Can run certain types of instructions very quickly
    - Useful for parallelizable problems (matrix arithmetic)



Using GPUs with PyTorch

- Requires NVIDIA GPU and CUDA installed
- PyTorch does most of the heavy lifting
  - Have to initialize or copy to GPU device - that's it!
  - Everything you operate on has to be on the same device (all GPU or all CPU)
- You have to be careful!
  - Transferring data to/from GPUs is expensive!
  - Be mindful about how much GPU memory you are using

Send network to GPU

```
net = nn.Sequential(nn.Linear(3, 1))
net = net.to(device=try_gpu())
```

Initialize tensor on a GPU

```
X = torch.ones(2, 3, device=try_gpu())
X
```

```
tensor([[1., 1., 1.],
        [1., 1., 1.]], device='cuda:0')
```

```
def try_gpu(i=0):  #@save
    """Return gpu(i) if exists, otherwise return cpu()."""
    if torch.cuda.device_count() >= i + 1:
        return torch.device(f'cuda:{i}')
    return torch.device('cpu')
```

Summary

- We talked about the basics of using PyTorch, and how auto-differentiation works
- We covered how to create blocks of layers for building complex networks, and how to make custom layers
- We learned how to manipulate network parameters and how to save / load your progress
- We introduced running your neural network on GPUs with PyTorch

jadamo@email.arizona.edu

Thank you! Questions?