6 Builders' Guide

Alongside giant datasets and powerful hardware, great software tools have played an indispensable role in the rapid progress of deep learning. Starting with the pathbreaking Theano library released in 2007, flexible open-source tools have enabled researchers to rapidly prototype models, avoiding repetitive work when recycling standard components while still maintaining the ability to make low-level modifications. Over time, deep learning's libraries have evolved to offer increasingly coarse abstractions. Just as semiconductor designers went from specifying transistors to logical circuits to writing code, neural networks researchers have moved from thinking about the behavior of individual artificial neurons to conceiving of networks in terms of whole layers, and now often design architectures with far coarser *blocks* in mind.

So far, we have introduced some basic machine learning concepts, ramping up to fully-functional deep learning models. In the last chapter, we implemented each component of an MLP from scratch and even showed how to leverage high-level APIs to roll out the same models effortlessly. To get you that far that fast, we *called upon* the libraries, but skipped over more advanced details about *how they work*. In this chapter, we will peel back the curtain, digging deeper into the key components of deep learning computation, namely model construction, parameter access and initialization, designing custom layers and blocks, reading and writing models to disk, and leveraging GPUs to achieve dramatic speedups. These insights will move you from *end user* to *power user*, giving you the tools needed to reap the benefits of a mature deep learning library while retaining the flexibility to implement more complex models, including those you invent yourself! While this chapter does not introduce any new models or datasets, the advanced modeling chapters that follow rely heavily on these techniques.

6.1 Layers and Modules

When we first introduced neural networks, we focused on linear models with a single output. Here, the entire model consists of just a single neuron. Note that a single neuron (i) takes some set of inputs; (ii) generates a corresponding scalar output; and (iii) has a set of associated parameters that can be updated to optimize some objective function of interest. Then, once we started thinking about networks with multiple outputs, we leveraged vectorized arithmetic to characterize an entire layer of neurons. Just like individual neurons,

layers (i) take a set of inputs, (ii) generate corresponding outputs, and (iii) are described by a set of tunable parameters. When we worked through softmax regression, a single layer was itself the model. However, even when we subsequently introduced MLPs, we could still think of the model as retaining this same basic structure.

Interestingly, for MLPs, both the entire model and its constituent layers share this structure. The entire model takes in raw inputs (the features), generates outputs (the predictions), and possesses parameters (the combined parameters from all constituent layers). Likewise, each individual layer ingests inputs (supplied by the previous layer) generates outputs (the inputs to the subsequent layer), and possesses a set of tunable parameters that are updated according to the signal that flows backwards from the subsequent layer.

While you might think that neurons, layers, and models give us enough abstractions to go about our business, it turns out that we often find it convenient to speak about components that are larger than an individual layer but smaller than the entire model. For example, the ResNet-152 architecture, which is wildly popular in computer vision, possesses hundreds of layers. These layers consist of repeating patterns of *groups of layers*. Implementing such a network one layer at a time can grow tedious. This concern is not just hypothetical—such design patterns are common in practice. The ResNet architecture mentioned above won the 2015 ImageNet and COCO computer vision competitions for both recognition and detection (He *et al.*, 2016) and remains a go-to architecture for many vision tasks. Similar architectures in which layers are arranged in various repeating patterns are now ubiquitous in other domains, including natural language processing and speech.

To implement these complex networks, we introduce the concept of a neural network *module*. A module could describe a single layer, a component consisting of multiple layers, or the entire model itself! One benefit of working with the module abstraction is that they can be combined into larger artifacts, often recursively. This is illustrated in Fig. 6.1.1. By defining code to generate modules of arbitrary complexity on demand, we can write surprisingly compact code and still implement complex neural networks.

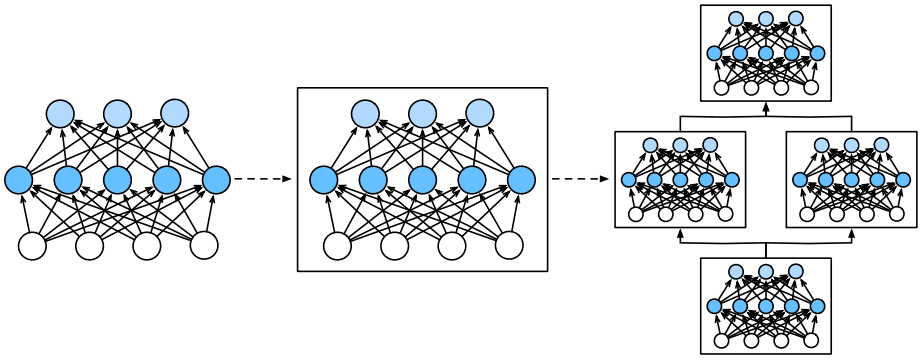

Fig. 6.1.1 Multiple layers are combined into modules, forming repeating patterns of larger models.

From a programming standpoint, a module is represented by a *class*. Any subclass of it must define a forward propagation method that transforms its input into output and must store any necessary parameters. Note that some modules do not require any parameters at

all. Finally a module must possess a backpropagation method, for purposes of calculating gradients. Fortunately, due to some behind-the-scenes magic supplied by the auto differentiation (introduced in Section 2.5) when defining our own module, we only need to worry about parameters and the forward propagation method.

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

To begin, we revisit the code that we used to implement MLPs (Section 5.1). The following code generates a network with one fully connected hidden layer with 256 units and ReLU activation, followed by a fully connected output layer with ten units (no activation function).

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

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

```
torch.Size([2, 10])
```

In this example, we constructed our model by instantiating an `nn.Sequential`, with layers in the order that they should be executed passed as arguments. In short, `nn.Sequential` defines a special kind of `Module`, the class that presents a module in PyTorch. It maintains an ordered list of constituent `Modules`. Note that each of the two fully connected layers is an instance of the `Linear` class which is itself a subclass of `Module`. The forward propagation (`forward`) method is also remarkably simple: it chains each module in the list together, passing the output of each as input to the next. Note that until now, we have been invoking our models via the construction `net(X)` to obtain their outputs. This is actually just shorthand for `net.__call__(X)`.

6.1.1 A Custom Module

Perhaps the easiest way to develop intuition about how a module works is to implement one ourselves. Before we do that, we briefly summarize the basic functionality that each module must provide:

1. Ingest input data as arguments to its forward propagation method.
2. Generate an output by having the forward propagation method return a value. Note that the output may have a different shape from the input. For example, the first fully connected layer in our model above ingests an input of arbitrary dimension but returns an output of dimension 256.
3. Calculate the gradient of its output with respect to its input, which can be accessed via its backpropagation method. Typically this happens automatically.

4. Store and provide access to those parameters necessary for executing the forward propagation computation.

5. Initialize model parameters as needed.

In the following snippet, we code up a module from scratch corresponding to an MLP with one hidden layer with 256 hidden units, and a 10-dimensional output layer. Note that the `MLP` class below inherits the class that represents a module. We will heavily rely on the parent class's methods, supplying only our own constructor (the `__init__` method in Python) and the forward propagation method.

```
class MLP(nn.Module):
    def __init__(self):
        # Call the constructor of the parent class nn.Module to perform
        # the necessary initialization
        super().__init__()
        self.hidden = nn.LazyLinear(256)
        self.out = nn.LazyLinear(10)

    # 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):
        return self.out(F.relu(self.hidden(X)))
```

Let's first focus on the forward propagation method. Note that it takes `X` as input, calculates the hidden representation with the activation function applied, and outputs its logits. In this `MLP` implementation, both layers are instance variables. To see why this is reasonable, imagine instantiating two MLPs, `net1` and `net2`, and training them on different data. Naturally, we would expect them to represent two different learned models.

We instantiate the MLP's layers in the constructor and subsequently invoke these layers on each call to the forward propagation method. Note a few key details. First, our customized `__init__` method invokes the parent class's `__init__` method via `super().__init__()` sparing us the pain of restating boilerplate code applicable to most modules. We then instantiate our two fully connected layers, assigning them to `self.hidden` and `self.out`. Note that unless we implement a new layer, we need not worry about the backpropagation method or parameter initialization. The system will generate these methods automatically. Let's try this out.

```
net = MLP()
net(X).shape
```

```
torch.Size([2, 10])
```

A key virtue of the module abstraction is its versatility. We can subclass a module to create layers (such as the fully connected layer class), entire models (such as the `MLP` class above), or various components of intermediate complexity. We exploit this versatility throughout the coming chapters, such as when addressing convolutional neural networks.

6.1.2 The Sequential Module

We can now take a closer look at how the `Sequential` class works. Recall that `Sequential` was designed to daisy-chain other modules together. To build our own simplified `MySequential`, we just need to define two key methods:

1. A method for appending modules one by one to a list.
2. A forward propagation method for passing an input through the chain of modules, in the same order as they were appended.

The following `MySequential` class delivers the same functionality of the default `Sequential` class.

```
class MySequential(nn.Module):
    def __init__(self, *args):
        super().__init__()
        for idx, module in enumerate(args):
            self.add_module(str(idx), module)

    def forward(self, X):
        for module in self.children():
            X = module(X)
        return X
```

In the `__init__` method, we add every module by calling the `add_modules` method. These modules can be accessed by the `children` method at a later date. In this way the system knows the added modules, and it will properly initialize each module's parameters.

When our `MySequential`'s forward propagation method is invoked, each added module is executed in the order in which they were added. We can now reimplement an MLP using our `MySequential` class.

```
net = MySequential(nn.LazyLinear(256), nn.ReLU(), nn.LazyLinear(10))
net(X).shape
```

```
torch.Size([2, 10])
```

Note that this use of `MySequential` is identical to the code we previously wrote for the `Sequential` class (as described in Section 5.1).

6.1.3 Executing Code in the Forward Propagation Method

The `Sequential` class makes model construction easy, allowing us to assemble new architectures without having to define our own class. However, not all architectures are simple daisy chains. When greater flexibility is required, we will want to define our own blocks. For example, we might want to execute Python's control flow within the forward propagation method. Moreover, we might want to perform arbitrary mathematical operations, not simply relying on predefined neural network layers.

You may have noticed that until now, all of the operations in our networks have acted upon our network's activations and its parameters. Sometimes, however, we might want to incorporate terms that are neither the result of previous layers nor updatable parameters. We call these *constant parameters*. Say for example that we want a layer that calculates the function $f(\mathbf{x}, \mathbf{w}) = c \cdot \mathbf{w}^\top \mathbf{x}$, where $\mathbf{x}$ is the input, $\mathbf{w}$ is our parameter, and c is some specified constant that is not updated during optimization. So we implement a `FixedHiddenMLP` class as follows.

```
class FixedHiddenMLP(nn.Module):
    def __init__(self):
        super().__init__()
        # Random weight parameters that will not compute gradients and
        # therefore keep constant during training
        self.rand_weight = torch.rand((20, 20))
        self.linear = nn.LazyLinear(20)

    def forward(self, X):
        X = self.linear(X)
        X = F.relu(X @ self.rand_weight + 1)
        # Reuse the fully connected layer. This is equivalent to sharing
        # parameters with two fully connected layers
        X = self.linear(X)
        # Control flow
        while X.abs().sum() > 1:
            X /= 2
        return X.sum()
```

In this model, we implement a hidden layer whose weights (`self.rand_weight`) are initialized randomly at instantiation and are thereafter constant. This weight is not a model parameter and thus it is never updated by backpropagation. The network then passes the output of this "fixed" layer through a fully connected layer.

Note that before returning the output, our model did something unusual. We ran a while-loop, testing on the condition its ℓ_1 norm is larger than 1, and dividing our output vector by 2 until it satisfied the condition. Finally, we returned the sum of the entries in `X`. To our knowledge, no standard neural network performs this operation. Note that this particular operation may not be useful in any real-world task. Our point is only to show you how to integrate arbitrary code into the flow of your neural network computations.

```
net = FixedHiddenMLP()
net(X)
```

```
tensor(-0.3836, grad_fn=<SumBackward0>)
```

We can mix and match various ways of assembling modules together. In the following example, we nest modules in some creative ways.

```
class NestMLP(nn.Module):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(nn.LazyLinear(64), nn.ReLU(),
                                 nn.LazyLinear(32), nn.ReLU())
        self.linear = nn.LazyLinear(16)

    def forward(self, X):
        return self.linear(self.net(X))

chimera = nn.Sequential(NestMLP(), nn.LazyLinear(20), FixedHiddenMLP())
chimera(X)
```

```
tensor(0.0679, grad_fn=<SumBackward0>)
```

6.1.4 Summary

Individual layers can be modules. Many layers can comprise a module. Many modules can comprise a module.

A module can contain code. Modules take care of lots of housekeeping, including parameter initialization and backpropagation. Sequential concatenations of layers and modules are handled by the `Sequential` module.

6.1.5 Exercises

1. What kinds of problems will occur if you change `MySequential` to store modules in a Python list?
2. Implement a module that takes two modules as an argument, say `net1` and `net2` and returns the concatenated output of both networks in the forward propagation. This is also called a *parallel module*.
3. Assume that you want to concatenate multiple instances of the same network. Implement a factory function that generates multiple instances of the same module and build a larger network from it.

Discussions[111].

6.2 Parameter Management

Once we have chosen an architecture and set our hyperparameters, we proceed to the training loop, where our goal is to find parameter values that minimize our loss function. After training, we will need these parameters in order to make future predictions. Additionally, we will sometimes wish to extract the parameters perhaps to reuse them in some other

context, to save our model to disk so that it may be executed in other software, or for examination in the hope of gaining scientific understanding.

Most of the time, we will be able to ignore the nitty-gritty details of how parameters are declared and manipulated, relying on deep learning frameworks to do the heavy lifting. However, when we move away from stacked architectures with standard layers, we will sometimes need to get into the weeds of declaring and manipulating parameters. In this section, we cover the following:

- Accessing parameters for debugging, diagnostics, and visualizations.
- Sharing parameters across different model components.

```
import torch
from torch import nn
```

We start by focusing on an MLP with one hidden layer.

```
net = nn.Sequential(nn.LazyLinear(8),
                    nn.ReLU(),
                    nn.LazyLinear(1))

X = torch.rand(size=(2, 4))
net(X).shape
```

```
torch.Size([2, 1])
```

6.2.1 Parameter Access

Let's start with how to access parameters from the models that you already know.

When a model is defined via the `Sequential` class, we can first access any layer by indexing into the model as though it were a list. Each layer's parameters are conveniently located in its attribute.

We can inspect the parameters of the second fully connected layer as follows.

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

```
OrderedDict([('weight',
              tensor([[-0.1649,  0.0605,  0.1694, -0.2524,  0.3526, -0.3414, -
↪0.2322,  0.0822]])),
             ('bias', tensor([0.0709]))])
```

We can see that this fully connected layer contains two parameters, corresponding to that layer's weights and biases, respectively.

Targeted Parameters

Note that each parameter is represented as an instance of the parameter class. To do anything useful with the parameters, we first need to access the underlying numerical values. There are several ways to do this. Some are simpler while others are more general. The following code extracts the bias from the second neural network layer, which returns a parameter class instance, and further accesses that parameter's value.

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

```
(torch.nn.parameter.Parameter, tensor([0.0709]))
```

Parameters are complex objects, containing values, gradients, and additional information. That is why we need to request the value explicitly.

In addition to the value, each parameter also allows us to access the gradient. Because we have not invoked backpropagation for this network yet, it is in its initial state.

```
net[2].weight.grad == None
```

```
True
```

All Parameters at Once

When we need to perform operations on all parameters, accessing them one-by-one can grow tedious. The situation can grow especially unwieldy when we work with more complex, e.g., nested, modules, since we would need to recurse through the entire tree to extract each sub-module's parameters. Below we demonstrate accessing the parameters of all layers.

```
[(name, param.shape) for name, param in net.named_parameters()]
```

```
[('0.weight', torch.Size([8, 4])),
 ('0.bias', torch.Size([8])),
 ('2.weight', torch.Size([1, 8])),
 ('2.bias', torch.Size([1]))]
```

6.2.2 Tied Parameters

Often, we want to share parameters across multiple layers. Let's see how to do this elegantly. In the following we allocate a fully connected layer and then use its parameters specifically to set those of another layer. Here we need to run the forward propagation `net(X)` before accessing the parameters.

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

net(X)
# Check whether the parameters are the same
print(net[2].weight.data[0] == net[4].weight.data[0])
net[2].weight.data[0, 0] = 100
# Make sure that they are actually the same object rather than just having the
# same value
print(net[2].weight.data[0] == net[4].weight.data[0])
```

```
tensor([True, True, True, True, True, True, True, True])
tensor([True, True, True, True, True, True, True, True])
```

This example shows that the parameters of the second and third layer are tied. They are not just equal, they are represented by the same exact tensor. Thus, if we change one of the parameters, the other one changes, too.

You might wonder, when parameters are tied what happens to the gradients? Since the model parameters contain gradients, the gradients of the second hidden layer and the third hidden layer are added together during backpropagation.

6.2.3 Summary

We have several ways of accessing and tying model parameters.

6.2.4 Exercises

1. Use the `NestMLP` model defined in Section 6.1 and access the parameters of the various layers.
2. Construct an MLP containing a shared parameter layer and train it. During the training process, observe the model parameters and gradients of each layer.
3. Why is sharing parameters a good idea?

Discussions[112].

112

6.3 Parameter Initialization

Now that we know how to access the parameters, let's look at how to initialize them properly. We discussed the need for proper initialization in Section 5.4. The deep learning

framework provides default random initializations to its layers. However, we often want to initialize our weights according to various other protocols. The framework provides most commonly used protocols, and also allows to create a custom initializer.

```
import torch
from torch import nn
```

By default, PyTorch initializes weight and bias matrices uniformly by drawing from a range that is computed according to the input and output dimension. PyTorch's `nn.init` module provides a variety of preset initialization methods.

```
net = nn.Sequential(nn.LazyLinear(8), nn.ReLU(), nn.LazyLinear(1))
X = torch.rand(size=(2, 4))
net(X).shape
```

```
torch.Size([2, 1])
```

6.3.1 Built-in Initialization

Let's begin by calling on built-in initializers. The code below initializes all weight parameters as Gaussian random variables with standard deviation 0.01, while bias parameters are cleared to zero.

```
def init_normal(module):
    if type(module) == nn.Linear:
        nn.init.normal_(module.weight, mean=0, std=0.01)
        nn.init.zeros_(module.bias)

net.apply(init_normal)
net[0].weight.data[0], net[0].bias.data[0]
```

```
(tensor([-0.0129, -0.0007, -0.0033,  0.0276]), tensor(0.))
```

We can also initialize all the parameters to a given constant value (say, 1).

```
def init_constant(module):
    if type(module) == nn.Linear:
        nn.init.constant_(module.weight, 1)
        nn.init.zeros_(module.bias)

net.apply(init_constant)
net[0].weight.data[0], net[0].bias.data[0]
```

```
(tensor([1., 1., 1., 1.]), tensor(0.))
```

We can also apply different initializers for certain blocks. For example, below we initialize

the first layer with the Xavier initializer and initialize the second layer to a constant value of 42.

```
def init_xavier(module):
    if type(module) == nn.Linear:
        nn.init.xavier_uniform_(module.weight)

def init_42(module):
    if type(module) == nn.Linear:
        nn.init.constant_(module.weight, 42)

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

```
tensor([-0.0974,  0.1707,  0.5840, -0.5032])
tensor([[42., 42., 42., 42., 42., 42., 42., 42.]])
```

Custom Initialization

Sometimes, the initialization methods we need are not provided by the deep learning framework. In the example below, we define an initializer for any weight parameter w using the following strange distribution:

$$w \sim \begin{cases} U(5, 10) & \text{with probability } \frac{1}{4} \\ 0 & \text{with probability } \frac{1}{2} \\ U(-10, -5) & \text{with probability } \frac{1}{4} \end{cases} \tag{6.3.1}$$

Again, we implement a `my_init` function to apply to `net`.

```
def my_init(module):
    if type(module) == nn.Linear:
        print("Init", *[(name, param.shape)
                        for name, param in module.named_parameters()][0])
        nn.init.uniform_(module.weight, -10, 10)
        module.weight.data *= module.weight.data.abs() >= 5

net.apply(my_init)
net[0].weight[:2]
```

```
Init weight torch.Size([8, 4])
Init weight torch.Size([1, 8])
```

```
tensor([[ 0.0000, -7.6364, -0.0000, -6.1206],
        [ 9.3516, -0.0000,  5.1208, -8.4003]], grad_fn=<SliceBackward0>)
```

Note that we always have the option of setting parameters directly.

```
net[0].weight.data[:] += 1
net[0].weight.data[0, 0] = 42
net[0].weight.data[0]
```

```
tensor([42.0000, -6.6364,  1.0000, -5.1206])
```

6.3.2 Summary

We can initialize parameters using built-in and custom initializers.

6.3.3 Exercises

Look up the online documentation for more built-in initializers.

Discussions[113].

113

6.4 Lazy Initialization

So far, it might seem that we got away with being sloppy in setting up our networks. Specifically, we did the following unintuitive things, which might not seem like they should work:

- We defined the network architectures without specifying the input dimensionality.
- We added layers without specifying the output dimension of the previous layer.
- We even "initialized" these parameters before providing enough information to determine how many parameters our models should contain.

You might be surprised that our code runs at all. After all, there is no way the deep learning framework could tell what the input dimensionality of a network would be. The trick here is that the framework *defers initialization*, waiting until the first time we pass data through the model, to infer the sizes of each layer on the fly.

Later on, when working with convolutional neural networks, this technique will become even more convenient since the input dimensionality (e.g., the resolution of an image) will affect the dimensionality of each subsequent layer. Hence the ability to set parameters without the need to know, at the time of writing the code, the value of the dimension can greatly simplify the task of specifying and subsequently modifying our models. Next, we go deeper into the mechanics of initialization.

```
import torch
from torch import nn
from d2l import torch as d2l
```

To begin, let's instantiate an MLP.

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

At this point, the network cannot possibly know the dimensions of the input layer's weights because the input dimension remains unknown.

Consequently the framework has not yet initialized any parameters. We confirm by attempting to access the parameters below.

```
net[0].weight
```

```
<UninitializedParameter>
```

Next let's pass data through the network to make the framework finally initialize parameters.

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

net[0].weight.shape
```

```
torch.Size([256, 20])
```

As soon as we know the input dimensionality, 20, the framework can identify the shape of the first layer's weight matrix by plugging in the value of 20. Having recognized the first layer's shape, the framework proceeds to the second layer, and so on through the computational graph until all shapes are known. Note that in this case, only the first layer requires lazy initialization, but the framework initializes sequentially. Once all parameter shapes are known, the framework can finally initialize the parameters.

The following method passes in dummy inputs through the network for a dry run to infer all parameter shapes and subsequently initializes the parameters. It will be used later when default random initializations are not desired.

```
@d2l.add_to_class(d2l.Module)  #@save
def apply_init(self, inputs, init=None):
    self.forward(*inputs)
    if init is not None:
        self.net.apply(init)
```

6.4.1 Summary

Lazy initialization can be convenient, allowing the framework to infer parameter shapes automatically, making it easy to modify architectures and eliminating one common source of errors. We can pass data through the model to make the framework finally initialize parameters.

6.4.2 Exercises

1. What happens if you specify the input dimensions to the first layer but not to subsequent layers? Do you get immediate initialization?
2. What happens if you specify mismatching dimensions?
3. What would you need to do if you have input of varying dimensionality? Hint: look at the parameter tying.

Discussions[114].

6.5 Custom Layers

One factor behind deep learning's success is the availability of a wide range of layers that can be composed in creative ways to design architectures suitable for a wide variety of tasks. For instance, researchers have invented layers specifically for handling images, text, looping over sequential data, and performing dynamic programming. Sooner or later, you will need a layer that does not exist yet in the deep learning framework. In these cases, you must build a custom layer. In this section, we show you how.

```
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l
```

6.5.1 Layers without Parameters

To start, we construct a custom layer that does not have any parameters of its own. This should look familiar if you recall our introduction to modules in Section 6.1. The following `CenteredLayer` class simply subtracts the mean from its input. To build it, we simply need to inherit from the base layer class and implement the forward propagation function.

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

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

Let's verify that our layer works as intended by feeding some data through it.

```
layer = CenteredLayer()
layer(torch.tensor([1.0, 2, 3, 4, 5]))
```

```
tensor([-2., -1.,  0.,  1.,  2.])
```

We can now incorporate our layer as a component in constructing more complex models.

```
net = nn.Sequential(nn.LazyLinear(128), CenteredLayer())
```

As an extra sanity check, we can send random data through the network and check that the mean is in fact 0. Because we are dealing with floating point numbers, we may still see a very small nonzero number due to quantization.

```
Y = net(torch.rand(4, 8))
Y.mean()
```

```
tensor(-6.5193e-09, grad_fn=<MeanBackward0>)
```

6.5.2 Layers with Parameters

Now that we know how to define simple layers, let's move on to defining layers with parameters that can be adjusted through training. We can use built-in functions to create parameters, which provide some basic housekeeping functionality. In particular, they govern access, initialization, sharing, saving, and loading model parameters. This way, among other benefits, we will not need to write custom serialization routines for every custom layer.

Now let's implement our own version of the fully connected layer. Recall that this layer requires two parameters, one to represent the weight and the other for the bias. In this implementation, we bake in the ReLU activation as a default. This layer requires two input arguments: `in_units` and `units`, which denote the number of inputs and outputs, respectively.

```
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)
```

Next, we instantiate the `MyLinear` class and access its model parameters.

```
linear = MyLinear(5, 3)
linear.weight
```

```
Parameter containing:
tensor([[ 0.4783,  0.4284, -0.0899],
        [-0.6347,  0.2913, -0.0822],
        [-0.4325, -0.1645, -0.3274],
        [ 1.1898,  0.6482, -1.2384],
        [-0.1479,  0.0264, -0.9597]], requires_grad=True)
```

We can directly carry out forward propagation calculations using custom layers.

```
linear(torch.rand(2, 5))
```

```
tensor([[0.0000, 0.9316, 0.0000],
        [0.1808, 1.4208, 0.0000]])
```

We can also construct models using custom layers. Once we have that we can use it just like the built-in fully connected layer.

```
net = nn.Sequential(MyLinear(64, 8), MyLinear(8, 1))
net(torch.rand(2, 64))
```

```
tensor([[ 0.0000],
        [13.0800]])
```

6.5.3 Summary

We can design custom layers via the basic layer class. This allows us to define flexible new layers that behave differently from any existing layers in the library. Once defined, custom layers can be invoked in arbitrary contexts and architectures. Layers can have local parameters, which can be created through built-in functions.

6.5.4 Exercises

1. Design a layer that takes an input and computes a tensor reduction, i.e., it returns $y_k = \sum_{i,j} W_{ijk} x_i x_j$.
2. Design a layer that returns the leading half of the Fourier coefficients of the data.

Discussions[115].

115

6.6 File I/O

So far we have discussed how to process data and how to build, train, and test deep learning models. However, at some point we will hopefully be happy enough with the learned

models that we will want to save the results for later use in various contexts (perhaps even to make predictions in deployment). Additionally, when running a long training process, the best practice is to periodically save intermediate results (checkpointing) to ensure that we do not lose several days' worth of computation if we trip over the power cord of our server. Thus it is time to learn how to load and store both individual weight vectors and entire models. This section addresses both issues.

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

6.6.1 Loading and Saving Tensors

For individual tensors, we can directly invoke the `load` and `save` functions to read and write them respectively. Both functions require that we supply a name, and `save` requires as input the variable to be saved.

```
x = torch.arange(4)
torch.save(x, 'x-file')
```

We can now read the data from the stored file back into memory.

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

```
tensor([0, 1, 2, 3])
```

We can store a list of tensors and read them back into memory.

```
y = torch.zeros(4)
torch.save([x, y],'x-files')
x2, y2 = torch.load('x-files')
(x2, y2)
```

```
(tensor([0, 1, 2, 3]), tensor([0., 0., 0., 0.]))
```

We can even write and read a dictionary that maps from strings to tensors. This is convenient when we want to read or write all the weights in a model.

```
mydict = {'x': x, 'y': y}
torch.save(mydict, 'mydict')
mydict2 = torch.load('mydict')
mydict2
```

```
{'x': tensor([0, 1, 2, 3]), 'y': tensor([0., 0., 0., 0.])}
```

6.6.2 Loading and Saving Model Parameters

Saving individual weight vectors (or other tensors) is useful, but it gets very tedious if we want to save (and later load) an entire model. After all, we might have hundreds of parameter groups sprinkled throughout. For this reason the deep learning framework provides built-in functionalities to load and save entire networks. An important detail to note is that this saves model *parameters* and not the entire model. For example, if we have a 3-layer MLP, we need to specify the architecture separately. The reason for this is that the models themselves can contain arbitrary code, hence they cannot be serialized as naturally. Thus, in order to reinstate a model, we need to generate the architecture in code and then load the parameters from disk. Let's start with our familiar MLP.

```
class MLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.hidden = nn.LazyLinear(256)
        self.output = nn.LazyLinear(10)

    def forward(self, x):
        return self.output(F.relu(self.hidden(x)))

net = MLP()
X = torch.randn(size=(2, 20))
Y = net(X)
```

Next, we store the parameters of the model as a file with the name "mlp.params".

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

To recover the model, we instantiate a clone of the original MLP model. Instead of randomly initializing the model parameters, we read the parameters stored in the file directly.

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

```
MLP(
  (hidden): LazyLinear(in_features=0, out_features=256, bias=True)
  (output): LazyLinear(in_features=0, out_features=10, bias=True)
)
```

Since both instances have the same model parameters, the computational result of the same input `X` should be the same. Let's verify this.

```
Y_clone = clone(X)
Y_clone == Y
```

```
tensor([[True, True, True, True, True, True, True, True, True, True],
        [True, True, True, True, True, True, True, True, True, True]])
```

6.6.3 Summary

The save and load functions can be used to perform file I/O for tensor objects. We can save and load the entire sets of parameters for a network via a parameter dictionary. Saving the architecture has to be done in code rather than in parameters.

6.6.4 Exercises

1. Even if there is no need to deploy trained models to a different device, what are the practical benefits of storing model parameters?
2. Assume that we want to reuse only parts of a network to be incorporated into a network having a different architecture. How would you go about using, say the first two layers from a previous network in a new network?
3. How would you go about saving the network architecture and parameters? What restrictions would you impose on the architecture?

Discussions[116].

116

6.7 GPUs

In tab_intro_decade, we illustrated the rapid growth of computation over the past two decades. In a nutshell, GPU performance has increased by a factor of 1000 every decade since 2000. This offers great opportunities but it also suggests that there was significant demand for such performance.

In this section, we begin to discuss how to harness this computational performance for your research. First by using a single GPU and at a later point, how to use multiple GPUs and multiple servers (with multiple GPUs).

Specifically, we will discuss how to use a single NVIDIA GPU for calculations. First, make sure you have at least one NVIDIA GPU installed. Then, download the NVIDIA driver and CUDA[117] and follow the prompts to set the appropriate path. Once these preparations are complete, the nvidia-smi command can be used to view the graphics card information.

117

In PyTorch, every array has a device; we often refer it as a *context*. So far, by default, all

variables and associated computation have been assigned to the CPU. Typically, other contexts might be various GPUs. Things can get even hairier when we deploy jobs across multiple servers. By assigning arrays to contexts intelligently, we can minimize the time spent transferring data between devices. For example, when training neural networks on a server with a GPU, we typically prefer for the model's parameters to live on the GPU.

To run the programs in this section, you need at least two GPUs. Note that this might be extravagant for most desktop computers but it is easily available in the cloud, e.g., by using the AWS EC2 multi-GPU instances. Almost all other sections do *not* require multiple GPUs, but here we simply wish to illustrate data flow between different devices.

```
import torch
from torch import nn
from d2l import torch as d2l
```

6.7.1 Computing Devices

We can specify devices, such as CPUs and GPUs, for storage and calculation. By default, tensors are created in the main memory and then the CPU is used for calculations.

In PyTorch, the CPU and GPU can be indicated by `torch.device('cpu')` and `torch.device('cuda')`. It should be noted that the cpu device means all physical CPUs and memory. This means that PyTorch's calculations will try to use all CPU cores. However, a gpu device only represents one card and the corresponding memory. If there are multiple GPUs, we use `torch.device(f'cuda:{i}')` to represent the i^{th} GPU (i starts at 0). Also, `gpu:0` and gpu are equivalent.

```
def cpu():  #@save
    """Get the CPU device."""
    return torch.device('cpu')

def gpu(i=0):  #@save
    """Get a GPU device."""
    return torch.device(f'cuda:{i}')

cpu(), gpu(), gpu(1)
```

```
(device(type='cpu'),
 device(type='cuda', index=0),
 device(type='cuda', index=1))
```

We can query the number of available GPUs.

```
def num_gpus():  #@save
    """Get the number of available GPUs."""
    return torch.cuda.device_count()

num_gpus()
```

```
2
```

Now we define two convenient functions that allow us to run code even if the requested GPUs do not exist.

```
def try_gpu(i=0):  #@save
    """Return gpu(i) if exists, otherwise return cpu()."""
    if num_gpus() >= i + 1:
        return gpu(i)
    return cpu()

def try_all_gpus():  #@save
    """Return all available GPUs, or [cpu(),] if no GPU exists."""
    return [gpu(i) for i in range(num_gpus())]

try_gpu(), try_gpu(10), try_all_gpus()
```

```
(device(type='cuda', index=0),
 device(type='cpu'),
 [device(type='cuda', index=0), device(type='cuda', index=1)])
```

6.7.2 Tensors and GPUs

By default, tensors are created on the CPU. We can query the device where the tensor is located.

```
x = torch.tensor([1, 2, 3])
x.device
```

```
device(type='cpu')
```

It is important to note that whenever we want to operate on multiple terms, they need to be on the same device. For instance, if we sum two tensors, we need to make sure that both arguments live on the same device—otherwise the framework would not know where to store the result or even how to decide where to perform the computation.

Storage on the GPU

There are several ways to store a tensor on the GPU. For example, we can specify a storage device when creating a tensor. Next, we create the tensor variable `X` on the first gpu. The tensor created on a GPU only consumes the memory of this GPU. We can use the `nvidia-smi` command to view GPU memory usage. In general, we need to make sure that we do not create data that exceeds the GPU memory limit.

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

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

Assuming that you have at least two GPUs, the following code will create a random tensor, `Y`, on the second GPU.

```
Y = torch.rand(2, 3, device=try_gpu(1))
Y
```

```
tensor([[0.0022, 0.5723, 0.2890],
        [0.1456, 0.3537, 0.7359]], device='cuda:1')
```

Copying

If we want to compute `X + Y`, we need to decide where to perform this operation. For instance, as shown in Fig. 6.7.1, we can transfer `X` to the second GPU and perform the operation there. *Do not* simply add `X` and `Y`, since this will result in an exception. The runtime engine would not know what to do: it cannot find data on the same device and it fails. Since `Y` lives on the second GPU, we need to move `X` there before we can add the two.

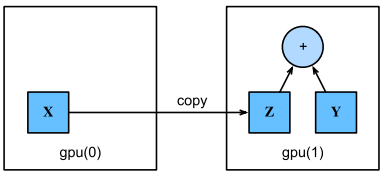

Fig. 6.7.1 Copy data to perform an operation on the same device.

```
Z = X.cuda(1)
print(X)
print(Z)
```

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

Now that the data (both `Z` and `Y`) are on the same GPU), we can add them up.

```
Y + Z
```

```
tensor([[1.0022, 1.5723, 1.2890],
        [1.1456, 1.3537, 1.7359]], device='cuda:1')
```

But what if your variable `Z` already lived on your second GPU? What happens if we still call `Z.cuda(1)`? It will return `Z` instead of making a copy and allocating new memory.

```
Z.cuda(1) is Z
```

```
True
```

Side Notes

People use GPUs to do machine learning because they expect them to be fast. But transferring variables between devices is slow: much slower than computation. So we want you to be 100% certain that you want to do something slow before we let you do it. If the deep learning framework just did the copy automatically without crashing then you might not realize that you had written some slow code.

Transferring data is not only slow, it also makes parallelization a lot more difficult, since we have to wait for data to be sent (or rather to be received) before we can proceed with more operations. This is why copy operations should be taken with great care. As a rule of thumb, many small operations are much worse than one big operation. Moreover, several operations at a time are much better than many single operations interspersed in the code unless you know what you are doing. This is the case since such operations can block if one device has to wait for the other before it can do something else. It is a bit like ordering your coffee in a queue rather than pre-ordering it by phone and finding out that it is ready when you are.

Last, when we print tensors or convert tensors to the NumPy format, if the data is not in the main memory, the framework will copy it to the main memory first, resulting in additional transmission overhead. Even worse, it is now subject to the dreaded global interpreter lock that makes everything wait for Python to complete.

6.7.3 Neural Networks and GPUs

Similarly, a neural network model can specify devices. The following code puts the model parameters on the GPU.

```
net = nn.Sequential(nn.LazyLinear(1))
net = net.to(device=try_gpu())
```

We will see many more examples of how to run models on GPUs in the following chapters, simply because the models will become somewhat more computationally intensive.

For example, when the input is a tensor on the GPU, the model will calculate the result on the same GPU.

```
net(X)
```

```
tensor([[0.7802],
        [0.7802]], device='cuda:0', grad_fn=<AddmmBackward0>)
```

Let's confirm that the model parameters are stored on the same GPU.

```
net[0].weight.data.device
```

```
device(type='cuda', index=0)
```

Let the trainer support GPU.

```
@d2l.add_to_class(d2l.Trainer)  #@save
def __init__(self, max_epochs, num_gpus=0, gradient_clip_val=0):
    self.save_hyperparameters()
    self.gpus = [d2l.gpu(i) for i in range(min(num_gpus, d2l.num_gpus()))]

@d2l.add_to_class(d2l.Trainer)  #@save
def prepare_batch(self, batch):
    if self.gpus:
        batch = [a.to(self.gpus[0]) for a in batch]
    return batch

@d2l.add_to_class(d2l.Trainer)  #@save
def prepare_model(self, model):
    model.trainer = self
    model.board.xlim = [0, self.max_epochs]
    if self.gpus:
        model.to(self.gpus[0])
    self.model = model
```

In short, as long as all data and parameters are on the same device, we can learn models efficiently. In the following chapters we will see several such examples.

6.7.4 Summary

We can specify devices for storage and calculation, such as the CPU or GPU. By default, data is created in the main memory and then uses the CPU for calculations. The deep learning framework requires all input data for calculation to be on the same device, be it CPU or the same GPU. You can lose significant performance by moving data without care. A typical mistake is as follows: computing the loss for every minibatch on the GPU and reporting it back to the user on the command line (or logging it in a NumPy `ndarray`) will trigger a global interpreter lock which stalls all GPUs. It is much better to allocate memory for logging inside the GPU and only move larger logs.

6.7.5 Exercises

1. Try a larger computation task, such as the multiplication of large matrices, and see the difference in speed between the CPU and GPU. What about a task with a small number of calculations?

2. How should we read and write model parameters on the GPU?
3. Measure the time it takes to compute 1000 matrix–matrix multiplications of 100×100 matrices and log the Frobenius norm of the output matrix one result at a time. Compare it with keeping a log on the GPU and transferring only the final result.
4. Measure how much time it takes to perform two matrix–matrix multiplications on two GPUs at the same time. Compare it with computing in in sequence on one GPU. Hint: you should see almost linear scaling.

Discussions[118].

118