5 | Deep Learning Computation

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 a multilayer perceptron from scratch and even showed how to leverage MXNet's Gluon library 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 combine the 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.

5.1 Layers and Blocks

When we first started talking about neural networks, we introduced linear models with a single output. Here, the entire model consists of just a single neuron. By itself, a single neuron takes some set of inputs, generates a corresponding (*scalar*) output, and 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, we showed how we could use linear algebra to efficiently express an entire *layer* of neurons. Layers too expect some inputs, generate corresponding outputs, and are described by a set of tunable parameters.

When we worked through softmax regression, a single *layer* was itself *the model*. However, when we subsequently introduced multilayer perceptrons, we developed models consisting of multiple layers. One interesting property of multilayer neural networks is that the *entire model* and its *constituent layers* share the same basic structure. The model takes the true inputs (as stated in the problem formulation), outputs predictions of the true outputs, and possesses parameters (the combined set of all parameters from all layers) Likewise any individual constituent layer in a mul-

tilayer perceptron ingests inputs (supplied by the previous layer) generates outputs (which form the inputs to the subsequent layer), and possesses a set of tunable parameters tht are updated with respect to the ultimate objective (using the signal that flows backwards through 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 will often want to express our model in terms of a components that are large than an indivudal layer. For example, when designing models, like ResNet-152, which possess hundreds (152, thus the name) of layers, implementing the network one layer at a time can grow tedious. Moreover, this concern is not just hypothetical—such deep networks dominate numerous application areas, especially when training data is abundant. For example the ResNet architecture mentioned above won the 2015 ImageNet and COCO computer vision competitions for both recognition and detection (He et al., 2016a). Deep networks with many layers arranged into components with various repeating patterns are now ubiquitous in other domains including natural language processing and speech.

To facilitate the implementation of networks consisting of components of arbitrary complexity, we introduce a new flexible concept: a neural network *block*. A block could describe a single neuron, a high-dimensional layer, or an arbitrarily-complex component consisting of multiple layers. From a software development, a `Block` is a class. Any subclass of `Block` must define a method called `forward` that transforms its input into output, and must store any necessary parameters. Note that some Blocks do not require any parameters at all! Finally a `Block` must possess a `backward` method, for purposes of calculating gradients. Fortunately, due to some behind-the-scenes magic supplied by the autograd `autograd` package (introduced in Chapter 2) when defining our own `Block` typically requires only that we worry about parameters and the `forward` function.

One benefit of working with the `Block` abstraction is that they can be combined into larger artifacts, often recursively, e.g., as illustrated in Fig. 5.1.1.

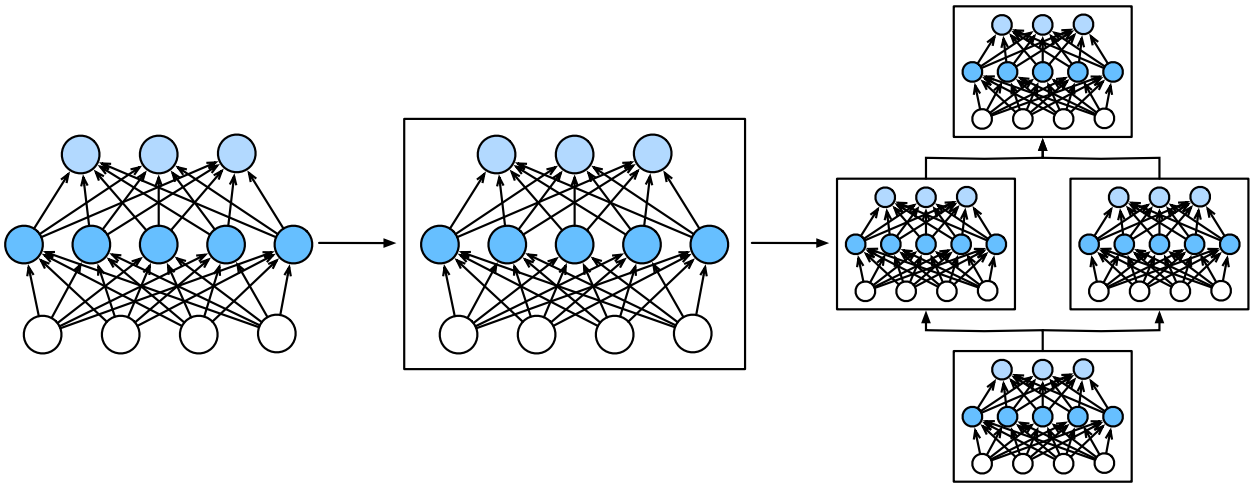

Fig. 5.1.1: Multiple layers are combined into blocks

By defining code to generate Blocks of arbitrary complexity on demand, we can write surprisingly compact code and still implement complex neural networks.

To begin, we revisit the Blocks that played a role in our implementation of the multilayer perceptron (Section 4.3). The following code generates a network with one fully-connected hidden layer containing 256 units followed by a ReLU activation, and then another fully-connected layer consisting of 10 units (with no activation function). Because there are no more layers, this last 10-unit

layer is regarded as the *output layer* and its outputs are also the model's output.

```
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()

x = np.random.uniform(size=(2, 20))

net = nn.Sequential()
net.add(nn.Dense(256, activation='relu'))
net.add(nn.Dense(10))
net.initialize()
net(x)
```

```
array([[ 0.06240272, -0.03268593,  0.02582653,  0.02254182, -0.03728798,
        -0.04253786,  0.00540613, -0.01364186, -0.09915452, -0.02272738],
       [ 0.02816677, -0.03341204,  0.03565666,  0.02506382, -0.04136416,
        -0.04941845,  0.01738528,  0.01081961, -0.09932579, -0.01176298]])
```

In this example, as in previous chapters, our model consists of an object returned by the `nn.Sequential` constructor. After instantiating a `nn.Sequential` and storing the `net` variable, we repeatedly called its `add` method, appending layers in the order that they should be executed. We suspect that you might have already understood *more or less* what was going on here the first time you saw this code. You may even have understood it well enough to modify the code and design your own networks. However, the details regarding what exactly happens inside `nn.Sequential` have remained mysterious so far.

In short, `nn.Sequential` just defines a special kind of Block. Specifically, an `nn.Sequential` maintains a list of constituent `Blocks`, stored in a particular order. You might think of `nnSequential` as your first meta-Block. The `add` method simply facilitates the addition of each successive `Block` to the list. Note that each our layers are instances of the `Dense` class which is itself a subclass of `Block`. The `forward` function is also remarkably simple: it chains each Block in the list together, passing the output of each as the 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.forward(X)`, a slick Python trick achieved via the Block class's `__call__` function.

Before we dive in to implementing our own custom `Block`, we briefly summarize the basic functionality that each `Block` must perform the following duties:

1. Ingest input data as arguments to its `forward` function.
2. Generate an output via the value returned by its `forward` function. Note that the output may have a different shape from the input. For example, the first Dense 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 `backward` method. Typically this happens automatically.
4. Store and provide access to those parameters necessary to execute the `forward` computation.
5. Initialize these parameters as needed.

5.1.1 A Custom Block

Perhaps the easiest way to develop intuition about how `nn.Block` works is to just dive right in and implement one ourselves. In the following snippet, instead of relying on `nn.Sequential`, we just code up a Block from scratch that implements a multilayer perceptron with one hidden layer, 256 hidden nodes, and 10 outputs.

Our `MLP` class below inherits the `Block` class. While we rely on some predefined methods in the parent class, we need to supply our own `__init__` and `forward` functions to uniquely define the behavior of our model.

```
from mxnet.gluon import nn

class MLP(nn.Block):
    # Declare a layer with model parameters. Here, we declare two fully
    # connected layers
    def __init__(self, **kwargs):
        # Call the constructor of the MLP parent class Block to perform the
        # necessary initialization. In this way, other function parameters can
        # also be specified when constructing an instance, such as the model
        # parameter, params, described in the following sections
        super(MLP, self).__init__(**kwargs)
        self.hidden = nn.Dense(256, activation='relu')  # Hidden layer
        self.output = nn.Dense(10)  # Output layer

    # Define the forward computation of the model, that is, how to return the
    # required model output based on the input x
    def forward(self, x):
        return self.output(self.hidden(x))
```

This code may be easiest to understand by working backwards from `forward`. Note that the `forward` method takes as input `x`. The forward method first evaluates `self.hidden(x)` to produce the hidden representation, passing this output as the input to the output layer `self.output( ... )`.

The constituent layers of each `MLP` must be instance-level variables. After all, if we instantiated two such models `net1` and `net2` and trained them on different data, we would expect them to them to represent two different learned models.

The `__init__` method is the most natural place to instantiate the layers that we subsequently invoke on each call to the `forward` method. Note that before getting on with the interesting parts, our customized `__init__` method must invoke the parent class's init method: `super(MLP, self).__init__(**kwargs)` to save us from reimplementing boilerplate code applicable to most Blocks. Then, all that is left is to instantiate our two `Dense` layers, assigning them to `self.hidden` and `self.output`, respectively. Again note that when dealing with standard functionality like this, we do not have to worry about backpropagation, since the `backward` method is generated for us automatically. The same goes for the `initialize` method. Let's try this out:

```
net = MLP()
net.initialize()
net(x)
```

```
array([[-0.03989594, -0.1041471 ,  0.06799038,  0.05245074,  0.02526059,
        -0.00640342,  0.04182098, -0.01665319, -0.02067346, -0.07863817],
```

(continues on next page)

(continued from previous page)

```
       [-0.03612847, -0.07210436,  0.09159479,  0.07890771,  0.02494172,
        -0.01028665,  0.01732428, -0.02843242,  0.03772651, -0.06671704]])
```

As we argued earlier, the primary virtue of the `Block` abstraction is its versatility. We can subclass `Block` to create layers (such as the `Dense` class provided by Gluon), entire models (such as the `MLP` class implemented above), or various components of intermediate complexity, a pattern that we will lean on heavily throughout the next chapters on convolutinoal neural networks.

5.1.2 The Sequential Block

As we described earlier, the `Sequential` class itself is also just a subclass of `Block`, designed specifically for daisy-chaining other Blocks together. All we need to do to implement our own `MySequential` block is to define a few convenience functions: 1. An `add` method for appending Blocks one by one to a list. 2. A `forward` method to pass inputs through the chain of Blocks (in the order of addition).

The following `MySequential` class delivers the same functionality as Gluon's default Sequential class:

```
class MySequential(nn.Block):
    def __init__(self, **kwargs):
        super(MySequential, self).__init__(**kwargs)

    def add(self, block):
        # Here, block is an instance of a Block subclass, and we assume it has
        # a unique name. We save it in the member variable _children of the
        # Block class, and its type is OrderedDict. When the MySequential
        # instance calls the initialize function, the system automatically
        # initializes all members of _children
        self._children[block.name] = block

    def forward(self, x):
        # OrderedDict guarantees that members will be traversed in the order
        # they were added
        for block in self._children.values():
            x = block(x)
        return x
```

At its core is the `add` method. It adds any block to the ordered dictionary of children. These are then executed in sequence when forward propagation is invoked. Let's see what the MLP looks like now.

```
net = MySequential()
net.add(nn.Dense(256, activation='relu'))
net.add(nn.Dense(10))
net.initialize()
net(x)
```

```
array([[-0.07645682, -0.01130233,  0.04952145, -0.04651389, -0.04131573,
        -0.05884133, -0.0621381 ,  0.01311472, -0.01379425, -0.02514282],
```

(continues on next page)

(continued from previous page)

```
       [-0.05124625,  0.00711231, -0.00155935, -0.07555379, -0.06675334,
        -0.01762914,  0.00589084,  0.01447191, -0.04330775,  0.03317726]])
```

Indeed, it can be observed that the use of the `MySequential` class is no different from the use of the Sequential class described in Section 4.3.

5.1.3 Blocks with Code

Although the Sequential class can make model construction easier, and you do not need to define the `forward` method, directly inheriting the Block class can greatly expand the flexibility of model construction. In particular, we will use Python's control flow within the forward method. While we are at it, we need to introduce another concept, that of the *constant* parameter. These are parameters that are not used when invoking backprop. This sounds very abstract but here's what is really going on. Assume that we have some function

$$f(\mathbf{x}, \mathbf{w}) = 3 \cdot \mathbf{w}^\top \mathbf{x}. \tag{5.1.1}$$

In this case 3 is a constant parameter. We could change 3 to something else, say c via

$$f(\mathbf{x}, \mathbf{w}) = c \cdot \mathbf{w}^\top \mathbf{x}. \tag{5.1.2}$$

Nothing has really changed, except that we can adjust the value of c. It is still a constant as far as $\mathbf{w}$ and $\mathbf{x}$ are concerned. However, since Gluon does not know about this beforehand, it is worth while to give it a hand (this makes the code go faster, too, since we are not sending the Gluon engine on a wild goose chase after a parameter that does not change). `get_constant` is the method that can be used to accomplish this. Let's see what this looks like in practice.

```
class FancyMLP(nn.Block):
    def __init__(self, **kwargs):
        super(FancyMLP, self).__init__(**kwargs)
        # Random weight parameters created with the get_constant are not
        # iterated during training (i.e., constant parameters)
        self.rand_weight = self.params.get_constant(
            'rand_weight', np.random.uniform(size=(20, 20)))
        self.dense = nn.Dense(20, activation='relu')

    def forward(self, x):
        x = self.dense(x)
        # Use the constant parameters created, as well as the relu
        # and dot functions
        x = npx.relu(np.dot(x, self.rand_weight.data()) + 1)
        # Reuse the fully connected layer. This is equivalent to sharing
        # parameters with two fully connected layers
        x = self.dense(x)
        # Here in Control flow, we need to call asscalar to return the scalar
        # for comparison
        while np.abs(x).sum() > 1:
            x /= 2
        if np.abs(x).sum() < 0.8:
            x *= 10
        return x.sum()
```

In this `FancyMLP` model, we used constant weight `Rand_weight` (note that it is not a model parameter), performed a matrix multiplication operation (`np.dot<`), and reused the *same* `Dense` layer. Note that this is very different from using two dense layers with different sets of parameters. Instead, we used the same network twice. Quite often in deep networks one also says that the parameters are *tied* when one wants to express that multiple parts of a network share the same parameters. Let's see what happens if we construct it and feed data through it.

```
net = FancyMLP()
net.initialize()
net(x)
```

```
array(5.2637568)
```

There is no reason why we couldn't mix and match these ways of build a network. Obviously the example below resembles more a chimera, or less charitably, a Rube Goldberg Machine[86]. That said, it combines examples for building a block from individual blocks, which in turn, may be blocks themselves. Furthermore, we can even combine multiple strategies inside the same forward function. To demonstrate this, here's the network.

```
class NestMLP(nn.Block):
    def __init__(self, **kwargs):
        super(NestMLP, self).__init__(**kwargs)
        self.net = nn.Sequential()
        self.net.add(nn.Dense(64, activation='relu'),
                     nn.Dense(32, activation='relu'))
        self.dense = nn.Dense(16, activation='relu')

    def forward(self, x):
        return self.dense(self.net(x))

chimera = nn.Sequential()
chimera.add(NestMLP(), nn.Dense(20), FancyMLP())

chimera.initialize()
chimera(x)
```

```
array(0.97720534)
```

5.1.4 Compilation

The avid reader is probably starting to worry about the efficiency of this. After all, we have lots of dictionary lookups, code execution, and lots of other Pythonic things going on in what is supposed to be a high performance deep learning library. The problems of Python's Global Interpreter Lock[87] are well known. In the context of deep learning it means that we have a super fast GPU (or multiple of them) which might have to wait until a puny single CPU core running Python gets a chance to tell it what to do next. This is clearly awful and there are many ways around it. The best way to speed up Python is by avoiding it altogether.

Gluon does this by allowing for Hybridization (Section 12.1). In it, the Python interpreter executes

[86] https://en.wikipedia.org/wiki/Rube_Goldberg_machine
[87] https://wiki.python.org/moin/GlobalInterpreterLock

the block the first time it is invoked. The Gluon runtime records what is happening and the next time around it short circuits any calls to Python. This can accelerate things considerably in some cases but care needs to be taken with control flow. We suggest that the interested reader skip forward to the section covering hybridization and compilation after finishing the current chapter.

Summary

- Layers are blocks
- Many layers can be a block
- Many blocks can be a block
- Code can be a block
- Blocks take are of a lot of housekeeping, such as parameter initialization, backprop and related issues.
- Sequential concatenations of layers and blocks are handled by the eponymous `Sequential` block.

Exercises

1. What kind of error message will you get when calling an `__init__` method whose parent class not in the `__init__` function of the parent class?
2. What kinds of problems will occur if you remove the `asscalar` function in the `FancyMLP` class?
3. What kinds of problems will occur if you change `self.net` defined by the Sequential instance in the `NestMLP` class to `self.net = [nn.Dense(64, activation='relu'), nn. Dense(32, activation='relu')]`?
4. Implement a block that takes two blocks as an argument, say `net1` and `net2` and returns the concatenated output of both networks in the forward pass (this is also called a parallel block).
5. Assume that you want to concatenate multiple instances of the same network. Implement a factory function that generates multiple instances of the same block and build a larger network from it.

5.2 Parameter Management

The ultimate goal of training deep networks is to find good parameter values for a given architecture. When everything is standard, the `nn.Sequential` class is a perfectly good tool for it. However, very few models are entirely standard and most scientists want to build things that are novel. This section shows how to manipulate parameters. In particular we will cover the following aspects:

- Accessing parameters for debugging, diagnostics, to visualize them or to save them is the first step to understanding how to work with custom models.

- Second, we want to set them in specific ways, e.g., for initialization purposes. We discuss the structure of parameter initializers.
- Last, we show how this knowledge can be put to good use by building networks that share some parameters.

As always, we start from our trusty Multilayer Perceptron with a hidden layer. This will serve as our choice for demonstrating the various features.

```
from mxnet import init, np, npx
from mxnet.gluon import nn
npx.set_np()

net = nn.Sequential()
net.add(nn.Dense(256, activation='relu'))
net.add(nn.Dense(10))
net.initialize()  # Use the default initialization method

x = np.random.uniform(size=(2, 20))
net(x)  # Forward computation
```

```
array([[ 0.06240272, -0.03268593,  0.02582653,  0.02254182, -0.03728798,
        -0.04253786,  0.00540613, -0.01364186, -0.09915452, -0.02272738],
       [ 0.02816677, -0.03341204,  0.03565666,  0.02506382, -0.04136416,
        -0.04941845,  0.01738528,  0.01081961, -0.09932579, -0.01176298]])
```

5.2.1 Parameter Access

In the case of a Sequential class we can access the parameters with ease, simply by indexing each of the layers in the network. The `params` variable then contains the required data. Let's try this out in practice by inspecting the parameters of the first layer.

```
print(net[0].params)
print(net[1].params)
```

```
dense0_ (
  Parameter dense0_weight (shape=(256, 20), dtype=float32)
  Parameter dense0_bias (shape=(256,), dtype=float32)
)
dense1_ (
  Parameter dense1_weight (shape=(10, 256), dtype=float32)
  Parameter dense1_bias (shape=(10,), dtype=float32)
)
```

The output tells us a number of things. First, the layer consists of two sets of parameters: `dense0_weight` and `dense0_bias`, as we would expect. They are both single precision and they have the necessary shapes that we would expect from the first layer, given that the input dimension is 20 and the output dimension 256. In particular the names of the parameters are very useful since they allow us to identify parameters *uniquely* even in a network of hundreds of layers and with nontrivial structure. The second layer is structured accordingly.

Targeted Parameters

In order to do something useful with the parameters we need to access them, though. There are several ways to do this, ranging from simple to general. Let's look at some of them.

```
print(net[1].bias)
print(net[1].bias.data())
```

```
Parameter dense1_bias (shape=(10,), dtype=float32)
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

The first returns the bias of the second layer. Since this is an object containing data, gradients, and additional information, we need to request the data explicitly. Note that the bias is all 0 since we initialized the bias to contain all zeros. Note that we can also access the parameters by name, such as `dense0_weight`. This is possible since each layer comes with its own parameter dictionary that can be accessed directly. Both methods are entirely equivalent but the first method leads to much more readable code.

```
print(net[0].params['dense0_weight'])
print(net[0].params['dense0_weight'].data())
```

```
Parameter dense0_weight (shape=(256, 20), dtype=float32)
[[ 0.06700657 -0.00369488  0.0418822  ... -0.05517294 -0.01194733
  -0.00369594]
 [-0.03296221 -0.04391347  0.03839272 ...  0.05636378  0.02545484
  -0.007007  ]
 [-0.0196689   0.01582889 -0.00881553 ...  0.01509629 -0.01908049
  -0.02449339]
 ...
 [-0.02055008 -0.02618652  0.06762936 ... -0.02315108 -0.06794678
  -0.04618235]
 [ 0.02802853  0.06672969  0.05018687 ... -0.02206502 -0.01315478
  -0.03791244]
 [-0.00638592  0.00914261  0.06667828 ... -0.00800052  0.03406764
  -0.03954004]]
```

Note that the weights are nonzero. This is by design since they were randomly initialized when we constructed the network. `data` is not the only function that we can invoke. For instance, we can compute the gradient with respect to the parameters. It has the same shape as the weight. However, since we did not invoke backpropagation yet, the values are all 0.

```
net[0].weight.grad()
```

```
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]])
```

All Parameters at Once

Accessing parameters as described above can be a bit tedious, in particular if we have more complex blocks, or blocks of blocks (or even blocks of blocks of blocks), since we need to walk through the entire tree in reverse order to how the blocks were constructed. To avoid this, blocks come with a method `collect_params` which grabs all parameters of a network in one dictionary such that we can traverse it with ease. It does so by iterating over all constituents of a block and calls `collect_params` on subblocks as needed. To see the difference consider the following:

```
# parameters only for the first layer
print(net[0].collect_params())
# parameters of the entire network
print(net.collect_params())
```

```
dense0_ (
  Parameter dense0_weight (shape=(256, 20), dtype=float32)
  Parameter dense0_bias (shape=(256,), dtype=float32)
)
sequential0_ (
  Parameter dense0_weight (shape=(256, 20), dtype=float32)
  Parameter dense0_bias (shape=(256,), dtype=float32)
  Parameter dense1_weight (shape=(10, 256), dtype=float32)
  Parameter dense1_bias (shape=(10,), dtype=float32)
)
```

This provides us with a third way of accessing the parameters of the network. If we wanted to get the value of the bias term of the second layer we could simply use this:

```
net.collect_params()['dense1_bias'].data()
```

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

Throughout the book we will see how various blocks name their subblocks (Sequential simply numbers them). This makes it very convenient to use regular expressions to filter out the required parameters.

```
print(net.collect_params('.*weight'))
print(net.collect_params('dense0.*'))
```

```
sequential0_ (
  Parameter dense0_weight (shape=(256, 20), dtype=float32)
  Parameter dense1_weight (shape=(10, 256), dtype=float32)
)
sequential0_ (
  Parameter dense0_weight (shape=(256, 20), dtype=float32)
  Parameter dense0_bias (shape=(256,), dtype=float32)
)
```

Rube Goldberg Striking Again

Let's see how the parameter naming conventions work if we nest multiple blocks inside each other. For that we first define a function that produces blocks (a block factory, so to speak) and then we combine these inside yet larger blocks.

```
def block1():
    net = nn.Sequential()
    net.add(nn.Dense(32, activation='relu'))
    net.add(nn.Dense(16, activation='relu'))
    return net

def block2():
    net = nn.Sequential()
    for i in range(4):
        net.add(block1())
    return net

rgnet = nn.Sequential()
rgnet.add(block2())
rgnet.add(nn.Dense(10))
rgnet.initialize()
rgnet(x)
```

```
array([[-4.1923025e-09,  1.9830502e-09,  8.9444063e-10,  6.2912990e-09,
        -3.3241778e-09,  5.4330038e-09,  1.6013515e-09, -3.7408681e-09,
         8.5468477e-09, -6.4805539e-09],
       [-3.7507064e-09,  1.4866974e-09,  6.8314709e-10,  5.6925784e-09,
        -2.6349172e-09,  4.8626667e-09,  1.4280275e-09, -3.4603027e-09,
         7.4127922e-09, -5.7896132e-09]])
```

Now that we are done designing the network, let's see how it is organized. `collect_params` provides us with this information, both in terms of naming and in terms of logical structure.

```
print(rgnet.collect_params)
print(rgnet.collect_params())
```

```
<bound method Block.collect_params of Sequential(
  (0): Sequential(
    (0): Sequential(
      (0): Dense(20 -> 32, Activation(relu))
      (1): Dense(32 -> 16, Activation(relu))
    )
    (1): Sequential(
      (0): Dense(16 -> 32, Activation(relu))
      (1): Dense(32 -> 16, Activation(relu))
    )
    (2): Sequential(
      (0): Dense(16 -> 32, Activation(relu))
      (1): Dense(32 -> 16, Activation(relu))
    )
    (3): Sequential(
      (0): Dense(16 -> 32, Activation(relu))
      (1): Dense(32 -> 16, Activation(relu))
```

(continues on next page)

(continued from previous page)

```
    )
  )
  (1): Dense(16 -> 10, linear)
)>
sequential1_ (
  Parameter dense2_weight (shape=(32, 20), dtype=float32)
  Parameter dense2_bias (shape=(32,), dtype=float32)
  Parameter dense3_weight (shape=(16, 32), dtype=float32)
  Parameter dense3_bias (shape=(16,), dtype=float32)
  Parameter dense4_weight (shape=(32, 16), dtype=float32)
  Parameter dense4_bias (shape=(32,), dtype=float32)
  Parameter dense5_weight (shape=(16, 32), dtype=float32)
  Parameter dense5_bias (shape=(16,), dtype=float32)
  Parameter dense6_weight (shape=(32, 16), dtype=float32)
  Parameter dense6_bias (shape=(32,), dtype=float32)
  Parameter dense7_weight (shape=(16, 32), dtype=float32)
  Parameter dense7_bias (shape=(16,), dtype=float32)
  Parameter dense8_weight (shape=(32, 16), dtype=float32)
  Parameter dense8_bias (shape=(32,), dtype=float32)
  Parameter dense9_weight (shape=(16, 32), dtype=float32)
  Parameter dense9_bias (shape=(16,), dtype=float32)
  Parameter dense10_weight (shape=(10, 16), dtype=float32)
  Parameter dense10_bias (shape=(10,), dtype=float32)
)
```

Since the layers are hierarchically generated, we can also access them accordingly. For instance, to access the first major block, within it the second subblock and then within it, in turn the bias of the first layer, we perform the following.

```
rgnet[0][1][0].bias.data()
```

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

5.2.2 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 initialization in Section 4.8. By default, MXNet initializes the weight matrices uniformly by drawing from $U[-0.07, 0.07]$ and the bias parameters are all set to 0. However, we often need to use other methods to initialize the weights. MXNet's `init` module provides a variety of preset initialization methods, but if we want something out of the ordinary, we need a bit of extra work.

Built-in Initialization

Let's begin with the built-in initializers. The code below initializes all parameters with Gaussian random variables.

```
# force_reinit ensures that the variables are initialized again, regardless of
# whether they were already initialized previously
net.initialize(init=init.Normal(sigma=0.01), force_reinit=True)
net[0].weight.data()[0]
```

```
array([-9.8788980e-03,  5.3957910e-03, -7.0842835e-03, -7.4317548e-03,
       -1.4880489e-02,  6.4959107e-03, -8.2659349e-03,  1.8743129e-02,
        1.6201857e-02,  1.4534278e-03,  2.2331164e-03,  1.5926110e-02,
       -1.2915777e-02, -8.8592555e-05, -1.7293986e-03, -7.2338698e-03,
        8.7698260e-03, -4.9947016e-03, -9.6906107e-03,  2.0079101e-03])
```

If we wanted to initialize all parameters to 1, we could do this simply by changing the initializer to `Constant(1)`.

```
net.initialize(init=init.Constant(1), force_reinit=True)
net[0].weight.data()[0]
```

```
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       1., 1., 1.])
```

If we want to initialize only a specific parameter in a different manner, we can simply set the initializer only for the appropriate subblock (or parameter) for that matter. For instance, below we initialize the second layer to a constant value of 42 and we use the `Xavier` initializer for the weights of the first layer.

```
net[1].initialize(init=init.Constant(42), force_reinit=True)
net[0].weight.initialize(init=init.Xavier(), force_reinit=True)
print(net[1].weight.data()[0, 0])
print(net[0].weight.data()[0])
```

```
42.0
[-0.06319056 -0.10960881  0.11757872 -0.07595599 -0.0849717   0.0851637
  0.08330765  0.04028694 -0.0305525   0.02012795 -0.03856885  0.1375024
  0.10155623 -0.05016676 -0.02575382 -0.14205234  0.14225402  0.02719662
 -0.0888046  -0.00962897]
```

Custom Initialization

Sometimes, the initialization methods we need are not provided in the `init` module. At this point, we can implement a subclass of the `Initializer` class so that we can use it like any other initialization method. Usually, we only need to implement the `_init_weight` function and modify the incoming `ndarray` according to the initial result. In the example below, we pick a decidedly bizarre and nontrivial distribution, just to prove the point. We draw the coefficients from the following

distribution:

$$w \sim \begin{cases} U[5, 10] & \text{with probability } \frac{1}{4} \\ 0 & \text{with probability } \frac{1}{2} \\ U[-10, -5] & \text{with probability } \frac{1}{4} \end{cases} \tag{5.2.1}$$

```
class MyInit(init.Initializer):
    def _init_weight(self, name, data):
        print('Init', name, data.shape)
        data[:] = np.random.uniform(-10, 10, data.shape)
        data *= np.abs(data) >= 5

net.initialize(MyInit(), force_reinit=True)
net[0].weight.data()[0]
```

```
Init dense0_weight (256, 20)
Init dense1_weight (10, 256)
```

```
array([-5.172625 , -7.0209026,  5.1446533, -9.844563 ,  8.545956 ,
       -0.       ,  0.       , -0.       ,  5.107664 ,  9.658335 ,
        5.8564453,  7.4483128,  0.       ,  0.       , -0.       ,
        7.9034443,  0.       ,  5.4223766,  8.5655575,  5.1224785])
```

If even this functionality is insufficient, we can set parameters directly. Since `data()` returns an `ndarray` we can access it just like any other matrix. A note for advanced users: if you want to adjust parameters within an `autograd` scope you need to use `set_data` to avoid confusing the automatic differentiation mechanics.

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

```
array([42.       , -6.0209026,  6.1446533, -8.844563 ,  9.545956 ,
        1.       ,  1.       ,  1.       ,  6.107664 , 10.658335 ,
        6.8564453,  8.448313 ,  1.       ,  1.       ,  1.       ,
        8.903444 ,  1.       ,  6.4223766,  9.5655575,  6.1224785])
```

5.2.3 Tied Parameters

In some cases, we want to share model parameters across multiple layers. For instance when we want to find good word embeddings we may decide to use the same parameters both for encoding and decoding of words. We discussed one such case when we introduced Section 5.1. Let's see how to do this a bit more elegantly. In the following we allocate a dense layer and then use its parameters specifically to set those of another layer.

```
net = nn.Sequential()
# We need to give the shared layer a name such that we can reference its
# parameters
shared = nn.Dense(8, activation='relu')
```

(continues on next page)

(continued from previous page)

```
net.add(nn.Dense(8, activation='relu'),
        shared,
        nn.Dense(8, activation='relu', params=shared.params),
        nn.Dense(10))
net.initialize()

x = np.random.uniform(size=(2, 20))
net(x)

# Check whether the parameters are the same
print(net[1].weight.data()[0] == net[2].weight.data()[0])
net[1].weight.data()[0, 0] = 100
# Make sure that they are actually the same object rather than just having the
# same value
print(net[1].weight.data()[0] == net[2].weight.data()[0])
```

```
[ True  True  True  True  True  True  True  True]
[ True  True  True  True  True  True  True  True]
```

The above example shows that the parameters of the second and third layer are tied. They are identical rather than just being equal. That is, by changing one of the parameters the other one changes, too. What happens to the gradients is quite ingenious. Since the model parameters contain gradients, the gradients of the second hidden layer and the third hidden layer are accumulated in the `shared.params.grad( )` during backpropagation.

Summary

- We have several ways to access, initialize, and tie model parameters.
- We can use custom initialization.
- Gluon has a sophisticated mechanism for accessing parameters in a unique and hierarchical manner.

Exercises

1. Use the FancyMLP defined in Section 5.1 and access the parameters of the various layers.
2. Look at the MXNet documentation[89] and explore different initializers.
3. Try accessing the model parameters after `net.initialize()` and before `net(x)` to observe the shape of the model parameters. What changes? Why?
4. Construct a multilayer perceptron containing a shared parameter layer and train it. During the training process, observe the model parameters and gradients of each layer.
5. Why is sharing parameters a good idea?

[89] http://beta.mxnet.io/api/gluon-related/mxnet.initializer.html

5.3 Deferred Initialization

In the previous examples we played fast and loose with setting up our networks. In particular we did the following things that *shouldn't* work:

- We defined the network architecture with no regard to the input dimensionality.
- We added layers without regard to the output dimension of the previous layer.
- We even "initialized" these parameters without knowing how many parameters were to initialize.

All of those things sound impossible and indeed, they are. After all, there is no way MXNet (or any other framework for that matter) could predict what the input dimensionality of a network would be. Later on, when working with convolutional networks and images this problem will become even more pertinent, since the input dimensionality (i.e., the resolution of an image) will affect the dimensionality of subsequent layers at a long range. Hence, the ability to set parameters without the need to know at the time of writing the code what the dimensionality is can greatly simplify statistical modeling. In what follows, we will discuss how this works using initialization as an example. After all, we cannot initialize variables that we do not know exist.

5.3.1 Instantiating a Network

Let's see what happens when we instantiate a network. We start with our trusty MLP as before.

```
from mxnet import init, np, npx
from mxnet.gluon import nn
npx.set_np()

def getnet():
    net = nn.Sequential()
    net.add(nn.Dense(256, activation='relu'))
    net.add(nn.Dense(10))
    return net

net = getnet()
```

At this point the network does not really know yet what the dimensionalities of the various parameters should be. All one could tell at this point is that each layer needs weights and bias, albeit of unspecified dimensionality. If we try accessing the parameters, that is exactly what happens.

```
print(net.collect_params)
print(net.collect_params())
```

```
<bound method Block.collect_params of Sequential(
  (0): Dense(-1 -> 256, Activation(relu))
  (1): Dense(-1 -> 10, linear)
)>
sequential0_ (
  Parameter dense0_weight (shape=(256, -1), dtype=float32)
  Parameter dense0_bias (shape=(256,), dtype=float32)
  Parameter dense1_weight (shape=(10, -1), dtype=float32)
  Parameter dense1_bias (shape=(10,), dtype=float32)
)
```

In particular, trying to access `net[0].weight.data()` at this point would trigger a runtime error stating that the network needs initializing before it can do anything. Let's see whether anything changes after we initialize the parameters:

```
net.initialize()
net.collect_params()
```

```
sequential0_ (
  Parameter dense0_weight (shape=(256, -1), dtype=float32)
  Parameter dense0_bias (shape=(256,), dtype=float32)
  Parameter dense1_weight (shape=(10, -1), dtype=float32)
  Parameter dense1_bias (shape=(10,), dtype=float32)
)
```

As we can see, nothing really changed. Only once we provide the network with some data do we see a difference. Let's try it out.

```
x = np.random.uniform(size=(2, 20))
net(x)  # Forward computation

net.collect_params()
```

```
sequential0_ (
  Parameter dense0_weight (shape=(256, 20), dtype=float32)
  Parameter dense0_bias (shape=(256,), dtype=float32)
  Parameter dense1_weight (shape=(10, 256), dtype=float32)
  Parameter dense1_bias (shape=(10,), dtype=float32)
)
```

The main difference to before is that as soon as we knew the input dimensionality, $\mathbf{x} \in \mathbb{R}^{20}$ it was possible to define the weight matrix for the first layer, i.e., $\mathbf{W}_1 \in \mathbb{R}^{256 \times 20}$. With that out of the way, we can progress to the second layer, define its dimensionality to be 10×256 and so on through the computational graph and bind all the dimensions as they become available. Once this is known, we can proceed by initializing parameters. This is the solution to the three problems outlined above.

5.3.2 Deferred Initialization in Practice

Now that we know how it works in theory, let's see when the initialization is actually triggered. In order to do so, we mock up an initializer which does nothing but report a debug message stating when it was invoked and with which parameters.

```
class MyInit(init.Initializer):
    def _init_weight(self, name, data):
        print('Init', name, data.shape)
        # The actual initialization logic is omitted here

net = getnet()
net.initialize(init=MyInit())
```

Note that, although `MyInit` will print information about the model parameters when it is called, the above `initialize` function does not print any information after it has been executed. Therefore there is no real initialization parameter when calling the `initialize` function. Next, we define the input and perform a forward calculation.

```
x = np.random.uniform(size=(2, 20))
y = net(x)
```

```
Init dense2_weight (256, 20)
Init dense3_weight (10, 256)
```

At this time, information on the model parameters is printed. When performing a forward calculation based on the input `x`, the system can automatically infer the shape of the weight parameters of all layers based on the shape of the input. Once the system has created these parameters, it calls the `MyInit` instance to initialize them before proceeding to the forward calculation.

Of course, this initialization will only be called when completing the initial forward calculation. After that, we will not re-initialize when we run the forward calculation `net(x)`, so the output of the `MyInit` instance will not be generated again.

```
y = net(x)
```

As mentioned at the beginning of this section, deferred initialization can also cause confusion. Before the first forward calculation, we were unable to directly manipulate the model parameters, for example, we could not use the `data` and `set_data` functions to get and modify the parameters. Therefore, we often force initialization by sending a sample observation through the network.

5.3.3 Forced Initialization

Deferred initialization does not occur if the system knows the shape of all parameters when calling the `initialize` function. This can occur in two cases:

- We have already seen some data and we just want to reset the parameters.
- We specified all input and output dimensions of the network when defining it.

The first case works just fine, as illustrated below.

```
net.initialize(init=MyInit(), force_reinit=True)
```

```
Init dense2_weight (256, 20)
Init dense3_weight (10, 256)
```

The second case requires us to specify the remaining set of parameters when creating the layer. For instance, for dense layers we also need to specify the `in_units` so that initialization can occur immediately once `initialize` is called.

```
net = nn.Sequential()
net.add(nn.Dense(256, in_units=20, activation='relu'))
net.add(nn.Dense(10, in_units=256))

net.initialize(init=MyInit())
```

```
Init dense4_weight (256, 20)
Init dense5_weight (10, 256)
```

Summary

- Deferred initialization is a good thing. It allows Gluon to set many things automatically and it removes a great source of errors from defining novel network architectures.
- We can override this by specifying all implicitly defined variables.
- Initialization can be repeated (or forced) by setting the `force_reinit=True` flag.

Exercises

1. What happens if you specify only parts of the input dimensions. Do you still 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 parameter tying.

5.4 Custom Layers

One of the reasons for the success of deep learning can be found in the wide range of layers that can be used in a deep network. This allows for a tremendous degree of customization and adaptation. For instance, scientists have invented layers for images, text, pooling, loops, dynamic programming, even for computer programs. Sooner or later you will encounter a layer that does not exist yet in Gluon, or even better, you will eventually invent a new layer that works well for your problem at hand. This is when it is time to build a custom layer. This section shows you how.

5.4.1 Layers without Parameters

Since this is slightly intricate, we start with a custom layer (also known as Block) that does not have any inherent parameters. Our first step is very similar to when we introduced blocks in Section 5.1. The following `CenteredLayer` class constructs a layer that subtracts the mean from the input. We build it by inheriting from the Block class and implementing the `forward` method.

```
from mxnet import gluon, np, npx
from mxnet.gluon import nn
npx.set_np()

class CenteredLayer(nn.Block):
    def __init__(self, **kwargs):
        super(CenteredLayer, self).__init__(**kwargs)

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

To see how it works let's feed some data into the layer.

```
layer = CenteredLayer()
layer(np.array([1, 2, 3, 4, 5]))
```

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

We can also use it to construct more complex models.

```
net = nn.Sequential()
net.add(nn.Dense(128), CenteredLayer())
net.initialize()
```

Let's see whether the centering layer did its job. For that we send random data through the network and check whether the mean vanishes. Note that since we are dealing with floating point numbers, we are going to see a very small albeit typically nonzero number.

```
y = net(np.random.uniform(size=(4, 8)))
y.mean()
```

```
array(3.783498e-10)
```

5.4.2 Layers with Parameters

Now that we know how to define layers in principle, let's define layers with parameters. These can be adjusted through training. In order to simplify things for an avid deep learning researcher the `Parameter` class and the `ParameterDict` dictionary provide some basic housekeeping functionality. In particular, they govern access, initialization, sharing, saving and loading model parameters. For instance, this way we do not need to write custom serialization routines for each new custom layer.

For instance, we can use the member variable `params` of the `ParameterDict` type that comes with the Block class. It is a dictionary that maps string type parameter names to model parameters in the `Parameter` type. We can create a `Parameter` instance from `ParameterDict` via the `get` function.

```
params = gluon.ParameterDict()
params.get('param2', shape=(2, 3))
params
```

```
(
  Parameter param2 (shape=(2, 3), dtype=<class 'numpy.float32'>)
)
```

Let's use this to implement our own version of the dense layer. It has two parameters: bias and weight. To make it a bit nonstandard, we bake in the ReLU activation as default. Next, we implement a fully connected layer with both weight and bias parameters. It uses ReLU as an activation function, where `in_units` and `units` are the number of inputs and the number of outputs, respectively.

```
class MyDense(nn.Block):
    # units: the number of outputs in this layer; in_units: the number of
    # inputs in this layer
    def __init__(self, units, in_units, **kwargs):
        super(MyDense, self).__init__(**kwargs)
        self.weight = self.params.get('weight', shape=(in_units, units))
        self.bias = self.params.get('bias', shape=(units,))

    def forward(self, x):
        linear = np.dot(x, self.weight.data()) + self.bias.data()
        return npx.relu(linear)
```

Naming the parameters allows us to access them by name through dictionary lookup later. It is a good idea to give them instructive names. Next, we instantiate the `MyDense` class and access its model parameters.

```
dense = MyDense(units=3, in_units=5)
dense.params
```

```
mydense0_ (
  Parameter mydense0_weight (shape=(5, 3), dtype=<class 'numpy.float32'>)
  Parameter mydense0_bias (shape=(3,), dtype=<class 'numpy.float32'>)
)
```

We can directly carry out forward calculations using custom layers.

```
dense.initialize()
dense(np.random.uniform(size=(2, 5)))
```

```
array([[0.        , 0.01633355, 0.        ],
       [0.        , 0.01581812, 0.        ]])
```

We can also construct models using custom layers. Once we have that we can use it just like the built-in dense layer. The only exception is that in our case size inference is not automatic. Please consult the MXNet documentation[92] for details on how to do this.

```
net = nn.Sequential()
net.add(MyDense(8, in_units=64),
        MyDense(1, in_units=8))
net.initialize()
net(np.random.uniform(size=(2, 64)))
```

```
array([[0.06508517],
       [0.0615553 ]])
```

Summary

- We can design custom layers via the Block class. This is more powerful than defining a block factory, since it can be invoked in many contexts.
- Blocks can have local parameters.

Exercises

1. Design a layer that learns an affine transform of the data, i.e., it removes the mean and learns an additive parameter instead.
2. Design a layer that takes an input and computes a tensor reduction, i.e., it returns $y_k = \sum_{i,j} W_{ijk} x_i x_j$.
3. Design a layer that returns the leading half of the Fourier coefficients of the data. Hint: look up the `fft` function in MXNet.

[92] http://www.mxnet.io

5.5 File I/O

So far we discussed how to process data, how to build, train and test deep learning models. However, at some point we are likely happy with what we obtained and we want to save the results for later use and distribution. Likewise, when running a long training process it is best practice to save intermediate results (checkpointing) to ensure that we do not lose several days worth of computation when tripping over the power cord of our server. At the same time, we might want to load a pre-trained model (e.g., we might have word embeddings for English and use it for our fancy spam classifier). For all of these cases we need to load and store both individual weight vectors and entire models. This section addresses both issues.

5.5.1 Loading and Saving `ndarray`s

In its simplest form, we can directly use the `load` and `save` functions to store and read `ndarrays` separately. This works just as expected.

```
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()

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

Then, we read the data from the stored file back into memory.

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

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

We can also store a list of `ndarrays` and read them back into memory.

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

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

We can even write and read a dictionary that maps from a string to an `ndarray`. This is convenient, for instance when we want to read or write all the weights in a model.

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

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

5.5.2 Gluon Model Parameters

Saving individual weight vectors (or other `ndarray` 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. Writing a script that collects all the terms and matches them to an architecture is quite some work. For this reason Gluon provides built-in functionality to load and save entire networks rather than just single weight vectors. An important detail to note is that this saves model *parameters* and not the entire model. I.e. 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 quite so easily (there is a way to do this for compiled models: please refer to the MXNet documentation[94] for the technical details on it). The result is that in order to reinstate a model we need to generate the architecture in code and then load the parameters from disk. The deferred initialization (Section 5.3) is quite advantageous here since we can simply define a model without the need to put actual values in place. Let's start with our favorite MLP.

```
class MLP(nn.Block):
    def __init__(self, **kwargs):
        super(MLP, self).__init__(**kwargs)
        self.hidden = nn.Dense(256, activation='relu')
        self.output = nn.Dense(10)

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

net = MLP()
net.initialize()
x = np.random.uniform(size=(2, 20))
y = net(x)
```

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

```
net.save_parameters('mlp.params')
```

To check whether we are able to recover the model we instantiate a clone of the original MLP model. Unlike the random initialization of model parameters, here we read the parameters stored in the file directly.

```
clone = MLP()
clone.load_parameters('mlp.params')
```

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

```
yclone = clone(x)
yclone == y
```

```
array([[ True,  True,  True,  True,  True,  True,  True,  True,  True,
         True],
       [ True,  True,  True,  True,  True,  True,  True,  True,  True,
         True]])
```

[94] http://www.mxnet.io

Summary

- The `save` and `load` functions can be used to perform File I/O for `ndarray` objects.
- The `load_parameters` and `save_parameters` functions allow us to save entire sets of parameters for a network in Gluon.
- Saving the architecture has to be done in code rather than in parameters.

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 of 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 network architecture and parameters? What restrictions would you impose on the architecture?

5.6 GPUs

In the introduction to this book we discussed 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 opportunity but it also suggests a significant need to provide such performance.

Decade	Dataset	Memory	Floating Point Calculations per Second
1970	100 (Iris)	1 KB	100 KF (Intel 8080)
1980	1 K (House prices in Boston)	100 KB	1 MF (Intel 80186)
1990	10 K (optical character recognition)	10 MB	10 MF (Intel 80486)
2000	10 M (web pages)	100 MB	1 GF (Intel Core)
2010	10 G (advertising)	1 GB	1 TF (NVIDIA C2050)
2020	1 T (social network)	100 GB	1 PF (NVIDIA DGX-2)

In this section we begin to discuss how to harness this compute performance for your research. First by using single GPUs and at a later point, how to use multiple GPUs and multiple servers (with multiple GPUs). You might have noticed that MXNet `ndarray` looks almost identical to NumPy. But there are a few crucial differences. One of the key features that differentiates MXNet from NumPy is its support for diverse hardware devices.

In MXNet, every array has a context. In fact, whenever we displayed an `ndarray` so far, it added a cryptic `@cpu(0)` notice to the output which remained unexplained so far. As we will discover, this

just indicates that the computation is being executed on the CPU. 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.

In short, for complex neural networks and large-scale data, using only CPUs for computation may be inefficient. In this section, 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 CUDA[96] 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.

```
!nvidia-smi
```

```
Sat Dec 14 03:45:01 2019
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 418.67       Driver Version: 418.67       CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla V100-SXM2...  Off  | 00000000:00:1B.0 Off |                    0 |
| N/A   34C    P0    50W / 300W |      0MiB / 16130MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Tesla V100-SXM2...  Off  | 00000000:00:1C.0 Off |                    0 |
| N/A   34C    P0    51W / 300W |      0MiB / 16130MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   2  Tesla V100-SXM2...  Off  | 00000000:00:1D.0 Off |                    0 |
| N/A   36C    P0    54W / 300W |      0MiB / 16130MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   3  Tesla V100-SXM2...  Off  | 00000000:00:1E.0 Off |                    0 |
| N/A   34C    P0    53W / 300W |      0MiB / 16130MiB |      4%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

Next, we need to confirm that the GPU version of MXNet is installed. If a CPU version of MXNet is already installed, we need to uninstall it first. For example, use the `pip uninstall mxnet` command, then install the corresponding MXNet version according to the CUDA version. Assuming you have CUDA 9.0 installed, you can install the MXNet version that supports CUDA 9.0 by `pip install mxnet-cu90`. 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. Instead, this is simply to illustrate how data flows between different devices.

[96] https://developer.nvidia.com/cuda-downloads

5.6.1 Computing Devices

MXNet can specify devices, such as CPUs and GPUs, for storage and calculation. By default, MXNet creates data in the main memory and then uses the CPU to calculate it. In MXNet, the CPU and GPU can be indicated by `cpu()` and `gpu()`. It should be noted that `cpu()` (or any integer in the parentheses) means all physical CPUs and memory. This means that MXNet's calculations will try to use all CPU cores. However, `gpu()` only represents one graphic card and the corresponding graphic memory. If there are multiple GPUs, we use `gpu(i)` to represent the i^{th} GPU (i starts from 0). Also, `gpu(0)` and `gpu()` are equivalent.

```
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()

npx.cpu(), npx.gpu(), npx.gpu(1)
```

```
(cpu(0), gpu(0), gpu(1))
```

We can query the number of available GPUs through `num_gpus()`.

```
npx.num_gpus()
```

```
2
```

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

```
# Saved in the d2l package for later use
def try_gpu(i=0):
    """Return gpu(i) if exists, otherwise return cpu()."""
    return npx.gpu(i) if npx.num_gpus() >= i + 1 else npx.cpu()

# Saved in the d2l package for later use
def try_all_gpus():
    """Return all available GPUs, or [cpu(),] if no GPU exists."""
    ctxes = [npx.gpu(i) for i in range(npx.num_gpus())]
    return ctxes if ctxes else [npx.cpu()]

try_gpu(), try_gpu(3), try_all_gpus()
```

```
(gpu(0), cpu(0), [gpu(0), gpu(1)])
```

5.6.2 ndarray and GPUs

By default, ndarray objects are created on the CPU. Therefore, we will see the @cpu(0) identifier each time we print an ndarray.

```
x = np.array([1, 2, 3])
x
```

```
array([1., 2., 3.])
```

We can use the context property of ndarray to view the device where the ndarray is located. It is important to note that whenever we want to operate on multiple terms they need to be in the same context. For instance, if we sum two variables, we need to make sure that both arguments are on the same device—otherwise MXNet would not know where to store the result or even how to decide where to perform the computation.

```
x.context
```

```
cpu(0)
```

Storage on the GPU

There are several ways to store an ndarray on the GPU. For example, we can specify a storage device with the ctx parameter when creating an ndarray. Next, we create the ndarray variable a on gpu(0). Notice that when printing a, the device information becomes @gpu(0). The ndarray 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 we do not create data that exceeds the GPU memory limit.

```
x = np.ones((2, 3), ctx=try_gpu())
x
```

```
array([[1., 1., 1.],
       [1., 1., 1.]], ctx=gpu(0))
```

Assuming you have at least two GPUs, the following code will create a random array on gpu(1).

```
y = np.random.uniform(size=(2, 3), ctx=try_gpu(1))
y
```

```
array([[0.67478997, 0.07540122, 0.9956977 ],
       [0.09488854, 0.415456  , 0.11231736]], ctx=gpu(1))
```

Copying

If we want to compute $\mathbf{x} + \mathbf{y}$ we need to decide where to perform this operation. For instance, as shown in Fig. 5.6.1, we can transfer $\mathbf{x}$ to `gpu(1)` and perform the operation there. *Do not* simply add `x + 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.

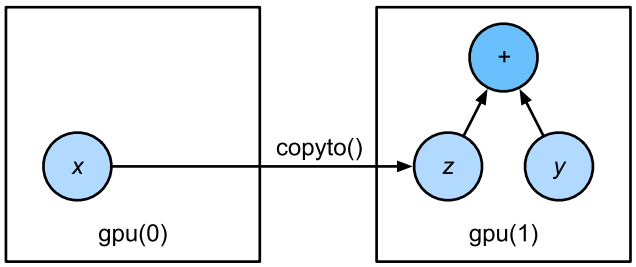

Fig. 5.6.1: Copyto copies arrays to the target device

`copyto` copies the data to another device such that we can add them. Since $\mathbf{y}$ lives on the second GPU we need to move $\mathbf{x}$ there before we can add the two.

```
z = x.copyto(try_gpu(1))
print(x)
print(z)
```

```
[[1. 1. 1.]
 [1. 1. 1.]] @gpu(0)
[[1. 1. 1.]
 [1. 1. 1.]] @gpu(1)
```

Now that the data is on the same GPU (both $\mathbf{z}$ and $\mathbf{y}$ are), we can add them up. In such cases MXNet places the result on the same device as its constituents. In our case that is `@gpu(1)`.

```
y + z
```

```
array([[1.6747899, 1.0754012, 1.9956977],
       [1.0948886, 1.415456 , 1.1123173]], ctx=gpu(1))
```

Imagine that your variable z already lives on your second GPU (gpu(1)). What happens if we call z.copyto(gpu(1))? It will make a copy and allocate new memory, even though that variable already lives on the desired device! There are times where depending on the environment our code is running in, two variables may already live on the same device. So we only want to make a copy if the variables currently lives on different contexts. In these cases, we can call `as_in_context()`. If the variable is already the specified context then this is a no-op. In fact, unless you specifically want to make a copy, `as_in_context()` is the method of choice.

```
z = x.as_in_context(try_gpu(1))
z
```

```
array([[1., 1., 1.],
       [1., 1., 1.]], ctx=gpu(1))
```

It is important to note that, if the context of the source variable and the target variable are consistent, then the as_in_context function causes the target variable and the source variable to share the memory of the source variable.

```
y.as_in_context(try_gpu(1)) is y
```

```
False
```

The copyto function always creates new memory for the target variable.

```
y.copyto(try_gpu(1)) is y
```

```
False
```

Side Notes

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

Also, transferring data between devices (CPU, GPUs, other machines) is something that is *much slower* than computation. 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 ndarrays or convert ndarrays to the NumPy format, if the data is not in main memory, MXNet 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 which makes everything wait for Python to complete.

5.6.3 Gluon and GPUs

Similarly, Gluon's model can specify devices through the ctx parameter during initialization. The following code initializes the model parameters on the GPU (we will see many more examples of how to run models on GPUs in the following, simply since they will become somewhat more compute intensive).

```
net = nn.Sequential()
net.add(nn.Dense(1))
net.initialize(ctx=try_gpu())
```

When the input is an ndarray on the GPU, Gluon will calculate the result on the same GPU.

```
net(x)
```

```
array([[0.04995865],
       [0.04995865]], ctx=gpu(0))
```

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

```
net[0].weight.data()
```

```
array([[0.0068339 , 0.01299825, 0.0301265 ]], ctx=gpu(0))
```

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

Summary

- MXNet can specify devices for storage and calculation, such as CPU or GPU. By default, MXNet creates data in the main memory and then uses the CPU to calculate it.
- MXNet 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 array) 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.

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 amount 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 matrix norm $\mathrm{tr} MM^\top$ one result at a time vs. 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 vs. in sequence on one GPU (hint: you should see almost linear scaling).