4.1 模型构造

让我们回顾一下在3.10节（多层感知机的简洁实现）中含单隐藏层的多层感知机的实现方法。我们首先构造 `Sequential` 实例，然后依次添加两个全连接层。其中第一层的输出大小为256，即隐藏层单元个数是256；第二层的输出大小为10，即输出层单元个数是10。我们在上一章的其他节中也使用了 `Sequential` 类构造模型。这里我们介绍另外一种基于 `Module` 类的模型构造方法：它让模型构造更加灵活。

> **注：其实前面我们陆陆续续已经使用了这些方法了，本节系统介绍一下。**

4.1.1 继承 `Module` 类来构造模型

`Module` 类是 `nn` 模块里提供的一个模型构造类，是所有神经网络模块的基类，我们可以继承它来定义我们想要的模型。下面继承 `Module` 类构造本节开头提到的多层感知机。这里定义的 `MLP` 类重载了 `Module` 类的 `__init__` 函数和 `forward` 函数。它们分别用于创建模型参数和定义前向计算。前向计算也即正向传播。

```
import torch
from torch import nn

class MLP(nn.Module):
    # 声明带有模型参数的层，这里声明了两个全连接层
    def __init__(self, **kwargs):
        # 调用MLP父类Module的构造函数来进行必要的初始化。这样在构造实例时还可以指定其他函数
        # 参数，如“模型参数的访问、初始化和共享”一节将介绍的模型参数params
        super(MLP, self).__init__(**kwargs)
        self.hidden = nn.Linear(784, 256) # 隐藏层
        self.act = nn.ReLU()
        self.output = nn.Linear(256, 10)  # 输出层


    # 定义模型的前向计算，即如何根据输入x计算返回所需要的模型输出
    def forward(self, x):
        a = self.act(self.hidden(x))
        return self.output(a)
```

以上的 `MLP` 类中无须定义反向传播函数。系统将通过自动求梯度而自动生成反向传播所需的 `backward` 函数。

我们可以实例化 `MLP` 类得到模型变量 `net` 。下面的代码初始化 `net` 并传入输入数据 `X` 做一次前向计算。其中， `net(X)` 会调用 `MLP` 继承自 `Module` 类的 `__call__` 函数，这个函数将调用 `MLP` 类定义的 `forward` 函数来完成前向计算。

```
X = torch.rand(2, 784)
net = MLP()
print(net)
net(X)
```

输出：

```
MLP(
  (hidden): Linear(in_features=784, out_features=256, bias=True)
  (act): ReLU()
  (output): Linear(in_features=256, out_features=10, bias=True)
)
tensor([[-0.1798, -0.2253,  0.0206, -0.1067, -0.0889,  0.1818, -0.1474,  0.1845,
         -0.1870,  0.1970],
        [-0.1843, -0.1562, -0.0090,  0.0351, -0.1538,  0.0992, -0.0883,  0.0911,
         -0.2293,  0.2360]], grad_fn=<ThAddmmBackward>)
```

注意，这里并没有将 `Module` 类命名为 `Layer` （层）或者 `Model` （模型）之类的名字，这是因为该类是一个可供自由组建的部件。它的子类既可以是一个层（如PyTorch提供的 `Linear` 类），又可以是一个模型（如这里定义的 `MLP` 类），或者是模型的一个部分。我们下面通过两个例子来展示它的灵活性。

4.1.2 `Module` 的子类

我们刚刚提到， `Module` 类是一个通用的部件。事实上，PyTorch还实现了继承自 `Module` 的可以方便构建模型的类: 如 `Sequential` 、 `ModuleList` 和 `ModuleDict` 等等。

4.1.2.1 `Sequential` 类

当模型的前向计算为简单串联各个层的计算时， `Sequential` 类可以通过更加简单的方式定义模型。这正是 `Sequential` 类的目的：它可以接收一个子模块的有序字典（OrderedDict）或者一系列子模块作为参数来逐一添加 `Module` 的实例，而模型的前向计算就是将这些实例按添加的顺序逐一计算。

下面我们实现一个与 `Sequential` 类有相同功能的 `MySequential` 类。这或许可以帮助读者更加清晰地理解 `Sequential` 类的工作机制。

```
class MySequential(nn.Module):
    from collections import OrderedDict
    def __init__(self, *args):
        super(MySequential, self).__init__()
        if len(args) == 1 and isinstance(args[0], OrderedDict): # 如果传入的是一个Orde
            for key, module in args[0].items():
                self.add_module(key, module)  # add_module方法会将module添加进self._mc
        else:  # 传入的是一些Module
            for idx, module in enumerate(args):
                self.add_module(str(idx), module)
    def forward(self, input):
        # self._modules返回一个 OrderedDict，保证会按照成员添加时的顺序遍历成员
        for module in self._modules.values():
            input = module(input)
        return input
```

我们用 `MySequential` 类来实现前面描述的 `MLP` 类，并使用随机初始化的模型做一次前向计算。

```
net = MySequential(
        nn.Linear(784, 256),
        nn.ReLU(),
        nn.Linear(256, 10),
        )
print(net)
net(X)
```

输出：

```
MySequential(
  (0): Linear(in_features=784, out_features=256, bias=True)
  (1): ReLU()
  (2): Linear(in_features=256, out_features=10, bias=True)
)
tensor([[-0.0100, -0.2516,  0.0392, -0.1684, -0.0937,  0.2191, -0.1448,  0.0930,
```

```
        0.1228, -0.2540],
      [-0.1086, -0.1858,  0.0203, -0.2051, -0.1404,  0.2738, -0.0607,  0.0622,
        0.0817, -0.2574]], grad_fn=<ThAddmmBackward>)
```

可以观察到这里 `MySequential` 类的使用跟3.10节（多层感知机的简洁实现）中 `Sequential` 类的使用没什么区别。

4.1.2.2 `ModuleList` 类

`ModuleList` 接收一个子模块的列表作为输入，然后也可以类似List那样进行append和extend操作:

```
net = nn.ModuleList([nn.Linear(784, 256), nn.ReLU()])
net.append(nn.Linear(256, 10)) # # 类似List的append操作
print(net[-1])  # 类似List的索引访问
print(net)
# net(torch.zeros(1, 784)) # 会报NotImplementedError
```

输出:

```
Linear(in_features=256, out_features=10, bias=True)
ModuleList(
  (0): Linear(in_features=784, out_features=256, bias=True)
  (1): ReLU()
  (2): Linear(in_features=256, out_features=10, bias=True)
)
```

既然 `Sequential` 和 `ModuleList` 都可以进行列表化构造网络，那二者区别是什么呢。 `ModuleList` 仅仅是一个储存各种模块的列表，这些模块之间没有联系也没有顺序（所以不用保证相邻层的输入输出维度匹配），而且没有实现 `forward` 功能需要自己实现，所以上面执行 `net(torch.zeros(1, 784))` 会报 `NotImplementedError` ；而 `Sequential` 内的模块需要按照顺序排列，要保证相邻层的输入输出大小相匹配，内部 `forward` 功能已经实现。

`ModuleList` 的出现只是让网络定义前向传播时更加灵活，见下面官网的例子。

```
class MyModule(nn.Module):
    def __init__(self):
```

```
            super(MyModule, self).__init__()
            self.linears = nn.ModuleList([nn.Linear(10, 10) for i in range(10)])

        def forward(self, x):
            # ModuleList can act as an iterable, or be indexed using ints
            for i, l in enumerate(self.linears):
                x = self.linears[i // 2](x) + l(x)
            return x
```

另外， `ModuleList` 不同于一般的Python的 `list` ，加入到 `ModuleList` 里面的所有模块的参数会被自动添加到整个网络中，下面看一个例子对比一下。

```
class Module_ModuleList(nn.Module):
    def __init__(self):
        super(Module_ModuleList, self).__init__()
        self.linears = nn.ModuleList([nn.Linear(10, 10)])

class Module_List(nn.Module):
    def __init__(self):
        super(Module_List, self).__init__()
        self.linears = [nn.Linear(10, 10)]

net1 = Module_ModuleList()
net2 = Module_List()

print("net1:")
for p in net1.parameters():
    print(p.size())

print("net2:")
for p in net2.parameters():
    print(p)
```

输出：

```
net1:
torch.Size([10, 10])
torch.Size([10])
net2:
```

4.1.2.3 `ModuleDict` 类

`ModuleDict` 接收一个子模块的字典作为输入, 然后也可以类似字典那样进行添加访问操作:

```
net = nn.ModuleDict({
    'linear': nn.Linear(784, 256),
    'act': nn.ReLU(),
})
net['output'] = nn.Linear(256, 10) # 添加
print(net['linear']) # 访问
print(net.output)
print(net)
# net(torch.zeros(1, 784)) # 会报NotImplementedError
```

输出:

```
Linear(in_features=784, out_features=256, bias=True)
Linear(in_features=256, out_features=10, bias=True)
ModuleDict(
  (act): ReLU()
  (linear): Linear(in_features=784, out_features=256, bias=True)
  (output): Linear(in_features=256, out_features=10, bias=True)
)
```

和 `ModuleList` 一样， `ModuleDict` 实例仅仅是存放了一些模块的字典，并没有定义 `forward` 函数需要自己定义。同样， `ModuleDict` 也与Python的 `Dict` 有所不同， `ModuleDict` 里的所有模块的参数会被自动添加到整个网络中。

4.1.3 构造复杂的模型

虽然上面介绍的这些类可以使模型构造更加简单，且不需要定义 `forward` 函数，但直接继承 `Module` 类可以极大地拓展模型构造的灵活性。下面我们构造一个稍微复杂点的网络 `FancyMLP` 。在这个网络中，我们通过 `get_constant` 函数创建训练中不被迭代的参数，即常数参数。在前向计算中，除了使用创建的常数参数外，我们还使用 `Tensor` 的函数和Python的控制流，并多次调用相同的层。

```
class FancyMLP(nn.Module):
    def __init__(self, **kwargs):
        super(FancyMLP, self).__init__(**kwargs)

        self.rand_weight = torch.rand((20, 20), requires_grad=False) # 不可训练参数（常
        self.linear = nn.Linear(20, 20)

    def forward(self, x):
        x = self.linear(x)
        # 使用创建的常数参数，以及nn.functional中的relu函数和mm函数
        x = nn.functional.relu(torch.mm(x, self.rand_weight.data) + 1)

        # 复用全连接层。等价于两个全连接层共享参数
        x = self.linear(x)
        # 控制流，这里我们需要调用item函数来返回标量进行比较
        while x.norm().item() > 1:
            x /= 2
        if x.norm().item() < 0.8:
            x *= 10
        return x.sum()
```

在这个 `FancyMLP` 模型中，我们使用了常数权重 `rand_weight` （注意它不是可训练模型参数）、做了矩阵乘法操作（ `torch.mm` ）并重复使用了相同的 `Linear` 层。下面我们来测试该模型的前向计算。

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

输出：

```
FancyMLP(
  (linear): Linear(in_features=20, out_features=20, bias=True)
)
tensor(0.8432, grad_fn=<SumBackward0>)
```

因为 `FancyMLP` 和 `Sequential` 类都是 `Module` 类的子类，所以我们可以嵌套调用它们。

```
class NestMLP(nn.Module):
    def __init__(self, **kwargs):
        super(NestMLP, self).__init__(**kwargs)
        self.net = nn.Sequential(nn.Linear(40, 30), nn.ReLU())

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

net = nn.Sequential(NestMLP(), nn.Linear(30, 20), FancyMLP())

X = torch.rand(2, 40)
print(net)
net(X)
```

输出：

```
Sequential(
  (0): NestMLP(
    (net): Sequential(
      (0): Linear(in_features=40, out_features=30, bias=True)
      (1): ReLU()
    )
  )
  (1): Linear(in_features=30, out_features=20, bias=True)
  (2): FancyMLP(
    (linear): Linear(in_features=20, out_features=20, bias=True)
  )
)
tensor(14.4908, grad_fn=<SumBackward0>)
```

小结

- 可以通过继承 `Module` 类来构造模型。
- `Sequential` 、 `ModuleList` 、 `ModuleDict` 类都继承自 `Module` 类。

- 与 `Sequential` 不同， `ModuleList` 和 `ModuleDict` 并没有定义一个完整的网络，它们只是将不同的模块存放在一起，需要自己定义 `forward` 函数。
- 虽然 `Sequential` 等类可以使模型构造更加简单，但直接继承 `Module` 类可以极大地拓展模型构造的灵活性。