4.4 自定义层

深度学习的一个魅力在于神经网络中各式各样的层，例如全连接层和后面章节中将要介绍的卷积层、池化层与循环层。虽然PyTorch提供了大量常用的层，但有时候我们依然希望自定义层。本节将介绍如何使用 `Module` 来自定义层，从而可以被重复调用。

4.4.1 不含模型参数的自定义层

我们先介绍如何定义一个不含模型参数的自定义层。事实上，这和4.1节（模型构造）中介绍的使用 `Module` 类构造模型类似。下面的 `CenteredLayer` 类通过继承 `Module` 类自定义了一个将输入减掉均值后输出的层，并将层的计算定义在了 `forward` 函数里。这个层里不含模型参数。

```
import torch
from torch import nn

class CenteredLayer(nn.Module):
    def __init__(self, **kwargs):
        super(CenteredLayer, self).__init__(**kwargs)
    def forward(self, x):
        return x - x.mean()
```

我们可以实例化这个层，然后做前向计算。

```
layer = CenteredLayer()
layer(torch.tensor([1, 2, 3, 4, 5], dtype=torch.float))
```

输出：

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

我们也可以用它来构造更复杂的模型。

```
net = nn.Sequential(nn.Linear(8, 128), CenteredLayer())
```

下面打印自定义层各个输出的均值。因为均值是浮点数，所以它的值是一个很接近0的数。

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

输出：

```
0.0
```

4.4.2 含模型参数的自定义层

我们还可以自定义含模型参数的自定义层。其中的模型参数可以通过训练学出。

在4.2节（模型参数的访问、初始化和共享）中介绍了 `Parameter` 类其实是 `Tensor` 的子类，如果一个 `Tensor` 是 `Parameter` ，那么它会自动被添加到模型的参数列表里。所以在自定义含模型参数的层时，我们应该将参数定义成 `Parameter` ，除了像4.2.1节那样直接定义成 `Parameter` 类外，还可以使用 `ParameterList` 和 `ParameterDict` 分别定义参数的列表和字典。

`ParameterList` 接收一个 `Parameter` 实例的列表作为输入然后得到一个参数列表，使用的时候可以用索引来访问某个参数，另外也可以使用 `append` 和 `extend` 在列表后面新增参数。

```
class MyDense(nn.Module):
    def __init__(self):
        super(MyDense, self).__init__()
        self.params = nn.ParameterList([nn.Parameter(torch.randn(4, 4)) for i in rang
        self.params.append(nn.Parameter(torch.randn(4, 1)))

    def forward(self, x):
        for i in range(len(self.params)):
            x = torch.mm(x, self.params[i])
        return x
net = MyDense()
print(net)
```

输出：

```
MyDense(
  (params): ParameterList(
      (0): Parameter containing: [torch.FloatTensor of size 4x4]
      (1): Parameter containing: [torch.FloatTensor of size 4x4]
      (2): Parameter containing: [torch.FloatTensor of size 4x4]
      (3): Parameter containing: [torch.FloatTensor of size 4x1]
  )
)
```

而 `ParameterDict` 接收一个 `Parameter` 实例的字典作为输入然后得到一个参数字典，然后可以按照字典的规则使用了。例如使用 `update()` 新增参数，使用 `keys()` 返回所有键值，使用 `items()` 返回所有键值对等等，可参考**官方文档**。

```
class MyDictDense(nn.Module):
    def __init__(self):
        super(MyDictDense, self).__init__()
        self.params = nn.ParameterDict({
                'linear1': nn.Parameter(torch.randn(4, 4)),
                'linear2': nn.Parameter(torch.randn(4, 1))
        })
        self.params.update({'linear3': nn.Parameter(torch.randn(4, 2))}) # 新增

    def forward(self, x, choice='linear1'):
        return torch.mm(x, self.params[choice])

net = MyDictDense()
print(net)
```

输出：

```
MyDictDense(
  (params): ParameterDict(
      (linear1): Parameter containing: [torch.FloatTensor of size 4x4]
      (linear2): Parameter containing: [torch.FloatTensor of size 4x1]
      (linear3): Parameter containing: [torch.FloatTensor of size 4x2]
  )
)
```

这样就可以根据传入的键值来进行不同的前向传播：

```
x = torch.ones(1, 4)
print(net(x, 'linear1'))
print(net(x, 'linear2'))
print(net(x, 'linear3'))
```

输出：

```
tensor([[1.5082, 1.5574, 2.1651, 1.2409]], grad_fn=<MmBackward>)
tensor([[-0.8783]], grad_fn=<MmBackward>)
tensor([[ 2.2193, -1.6539]], grad_fn=<MmBackward>)
```

我们也可以使用自定义层构造模型。它和PyTorch的其他层在使用上很类似。

```
net = nn.Sequential(
    MyDictDense(),
    MyListDense(),
)
print(net)
print(net(x))
```

输出：

Copy to clipboard

```
Sequential(
  (0): MyDictDense(
    (params): ParameterDict(
        (linear1): Parameter containing: [torch.FloatTensor of size 4x4]
        (linear2): Parameter containing: [torch.FloatTensor of size 4x1]
        (linear3): Parameter containing: [torch.FloatTensor of size 4x2]
    )
  )
  (1): MyListDense(
    (params): ParameterList(
        (0): Parameter containing: [torch.FloatTensor of size 4x4]
        (1): Parameter containing: [torch.FloatTensor of size 4x4]
```

```
            (2): Parameter containing: [torch.FloatTensor of size 4x4]
            (3): Parameter containing: [torch.FloatTensor of size 4x1]
        )
      )
    )
    tensor([[-101.2394]], grad_fn=<MmBackward>)
```

小结

- 可以通过 `Module` 类自定义神经网络中的层，从而可以被重复调用。