4.2 模型参数的访问、初始化和共享

在3.3节（线性回归的简洁实现）中，我们通过 `init` 模块来初始化模型的参数。我们也介绍了访问模型参数的简单方法。本节将深入讲解如何访问和初始化模型参数，以及如何在多个层之间共享同一份模型参数。

我们先定义一个与上一节中相同的含单隐藏层的多层感知机。我们依然使用默认方式初始化它的参数，并做一次前向计算。与之前不同的是，在这里我们从 `nn` 中导入了 `init` 模块，它包含了多种模型初始化方法。

```
import torch
from torch import nn
from torch.nn import init

net = nn.Sequential(nn.Linear(4, 3), nn.ReLU(), nn.Linear(3, 1))  # pytorch已进行默认

print(net)
X = torch.rand(2, 4)
Y = net(X).sum()
```

输出：

```
Sequential(
  (0): Linear(in_features=4, out_features=3, bias=True)
  (1): ReLU()
  (2): Linear(in_features=3, out_features=1, bias=True)
)
```

4.2.1 访问模型参数

回忆一下上一节中提到的 `Sequential` 类与 `Module` 类的继承关系。对于 `Sequential` 实例中含模型参数的层，我们可以通过 `Module` 类的 `parameters()` 或者 `named_parameters` 方法来访问所有参数（以迭代器的形式返回），后者除了返回参数 `Tensor` 外还会返回其名字。下面，访问多层感知机 `net` 的所有参数：

```
print(type(net.named_parameters()))
for name, param in net.named_parameters():
    print(name, param.size())
```

输出：

```
<class 'generator'>
0.weight torch.Size([3, 4])
0.bias torch.Size([3])
2.weight torch.Size([1, 3])
2.bias torch.Size([1])
```

可见返回的名字自动加上了层数的索引作为前缀。 我们再来访问 `net` 中单层的参数。对于使用 `Sequential` 类构造的神经网络，我们可以通过方括号 `[]` 来访问网络的任一层。索引0表示隐藏层为 `Sequential` 实例最先添加的层。

```
for name, param in net[0].named_parameters():
    print(name, param.size(), type(param))
```

输出：

```
weight torch.Size([3, 4]) <class 'torch.nn.parameter.Parameter'>
bias torch.Size([3]) <class 'torch.nn.parameter.Parameter'>
```

因为这里是单层的所以没有了层数索引的前缀。另外返回的 `param` 的类型为 `torch.nn.parameter.Parameter` ，其实这是 `Tensor` 的子类，和 `Tensor` 不同的是如果一个 `Tensor` 是 `Parameter` ，那么它会自动被添加到模型的参数列表里，来看下面这个例子。

```
class MyModel(nn.Module):
    def __init__(self, **kwargs):
        super(MyModel, self).__init__(**kwargs)
        self.weight1 = nn.Parameter(torch.rand(20, 20))
        self.weight2 = torch.rand(20, 20)
    def forward(self, x):
```

```
            pass

n = MyModel()
for name, param in n.named_parameters():
    print(name)
```

输出:

```
weight1
```

上面的代码中 `weight1` 在参数列表中但是 `weight2` 却没在参数列表中。

因为 `Parameter` 是 `Tensor` ，即 `Tensor` 拥有的属性它都有，比如可以根据 `data` 来访问参数数值，用 `grad` 来访问参数梯度。

```
weight_0 = list(net[0].parameters())[0]
print(weight_0.data)
print(weight_0.grad) # 反向传播前梯度为None
Y.backward()
print(weight_0.grad)
```

输出:

```
tensor([[ 0.2719, -0.0898, -0.2462,  0.0655],
        [-0.4669, -0.2703,  0.3230,  0.2067],
        [-0.2708,  0.1171, -0.0995,  0.3913]])
None
tensor([[-0.2281, -0.0653, -0.1646, -0.2569],
        [-0.1916, -0.0549, -0.1382, -0.2158],
        [ 0.0000,  0.0000,  0.0000,  0.0000]])
```

4.2.2 初始化模型参数

我们在3.15节（数值稳定性和模型初始化）中提到了PyTorch中 `nn.Module` 的模块参数都采取了较为合理的初始化策略（不同类型的layer具体采样的哪一种初始化方法的可参考**源代码**）。但我们经常需要使用其

他方法来初始化权重。PyTorch的 `init` 模块里提供了多种预设的初始化方法。在下面的例子中，我们将权重参数初始化成均值为0、标准差为0.01的正态分布随机数，并依然将偏差参数清零。

```
for name, param in net.named_parameters():
    if 'weight' in name:
        init.normal_(param, mean=0, std=0.01)
        print(name, param.data)
```

输出：

```
0.weight tensor([[ 0.0030,  0.0094,  0.0070, -0.0010],
        [ 0.0001,  0.0039,  0.0105, -0.0126],
        [ 0.0105, -0.0135, -0.0047, -0.0006]])
2.weight tensor([[-0.0074,  0.0051,  0.0066]])
```

下面使用常数来初始化权重参数。

```
for name, param in net.named_parameters():
    if 'bias' in name:
        init.constant_(param, val=0)
        print(name, param.data)
```

输出：

```
0.bias tensor([0., 0., 0.])
2.bias tensor([0.])
```

4.2.3 自定义初始化方法

有时候我们需要的初始化方法并没有在 `init` 模块中提供。这时，可以实现一个初始化方法，从而能够像使用其他初始化方法那样使用它。在这之前我们先来看看PyTorch是怎么实现这些初始化方法的，例如 `torch.nn.init.normal_` ：

```
def normal_(tensor, mean=0, std=1):
    with torch.no_grad():
        return tensor.normal_(mean, std)
```

可以看到这就是一个inplace改变 `Tensor` 值的函数，而且这个过程是不记录梯度的。 类似的我们来实现一个自定义的初始化方法。在下面的例子里，我们令权重有一半概率初始化为0，有另一半概率初始化为$[-10,-5]$和$[5,10]$两个区间里均匀分布的随机数。

```
def init_weight_(tensor):
    with torch.no_grad():
        tensor.uniform_(-10, 10)
        tensor *= (tensor.abs() >= 5).float()

for name, param in net.named_parameters():
    if 'weight' in name:
        init_weight_(param)
        print(name, param.data)
```

输出：

```
0.weight tensor([[ 7.0403,  0.0000, -9.4569,  7.0111],
        [-0.0000, -0.0000,  0.0000,  0.0000],
        [ 9.8063, -0.0000,  0.0000, -9.7993]])
2.weight tensor([[-5.8198,  7.7558, -5.0293]])
```

此外，参考2.3.2节，我们还可以通过改变这些参数的 `data` 来改写模型参数值同时不会影响梯度:

```
for name, param in net.named_parameters():
    if 'bias' in name:
        param.data += 1
        print(name, param.data)
```

输出：

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

4.2.4 共享模型参数

在有些情况下，我们希望在多个层之间共享模型参数。4.1.3节提到了如何共享模型参数: `Module` 类的 `forward` 函数里多次调用同一个层。此外，如果我们传入 `Sequential` 的模块是同一个 `Module` 实例的话参数也是共享的，下面来看一个例子:

```
linear = nn.Linear(1, 1, bias=False)
net = nn.Sequential(linear, linear)
print(net)
for name, param in net.named_parameters():
    init.constant_(param, val=3)
    print(name, param.data)
```

输出:

```
Sequential(
  (0): Linear(in_features=1, out_features=1, bias=False)
  (1): Linear(in_features=1, out_features=1, bias=False)
)
0.weight tensor([[3.]])
```

在内存中，这两个线性层其实一个对象:

```
print(id(net[0]) == id(net[1]))
print(id(net[0].weight) == id(net[1].weight))
```

输出:

```
True
True
```

因为模型参数里包含了梯度，所以在反向传播计算时，这些共享的参数的梯度是累加的:

```
x = torch.ones(1, 1)
y = net(x).sum()
print(y)
y.backward()
print(net[0].weight.grad) # 单次梯度是3，两次所以就是6
```

输出:

```
tensor(9., grad_fn=<SumBackward0>)
tensor([[6.]])
```

小结

- 有多种方法来访问、初始化和共享模型参数。
- 可以自定义初始化方法。