参数管理

在选择了架构并设置了超参数后，我们就进入了训练阶段。 此时，我们的目标是找到使损失函数最小化的模型参数值。 经过训练后，我们将需要使用这些参数来做出未来的预测。 此外，有时我们希望提取参数，以便在其他环境中复用它们， 将模型保存下来，以便它可以在其他软件中执行， 或者为了获得科学的理解而进行检查。

之前的介绍中，我们只依靠深度学习框架来完成训练的工作， 而忽略了操作参数的具体细节。 本节，我们将介绍以下内容：

- 访问参数，用于调试、诊断和可视化；
- 参数初始化；
- 在不同模型组件间共享参数。

(我们首先看一下具有单隐藏层的多层感知机。)

In [3]:
```
import torch
from torch import nn

# Linear(4, 8)，期待输入4个特征，输出一个特征
net = nn.Sequential(nn.Linear(4, 8), nn.ReLU(), nn.Linear(8, 1))

# 输入X有2个样本，每个样本4个特征
X = torch.rand(size=(2, 4))
net(X)
```

Out[3]:
```
tensor([[-0.3875],
        [-0.3005]], grad_fn=<AddmmBackward0>)
```

[参数访问]

我们从已有模型中访问参数。 当通过 Sequential 类定义模型时， 我们可以通过索引来访问模型的任意层。 这就像模型是一个列表一样，每层的参数都在其属性中。 如下所示，我们可以检查第二个全连接层的参数。

state就是w和b

In [5]:
```
print(net[2].state_dict())
```

```
OrderedDict([('weight', tensor([[-0.0820, -0.2908, -0.3486,  0.2764, -0.1803,  0.3136, -0.241
6, -0.3422]])), ('bias', tensor([-0.0444]))])
```

输出的结果告诉我们一些重要的事情： 首先，这个全连接层包含两个参数，分别是该层的权重和偏置。 两者都存储为单精度浮点数（float32）。 注意，参数名称允许唯一标识每个参数，即使在包含数百个层的网络中也是如此。

[目标参数]

注意，每个参数都表示为参数类的一个实例。 要对参数执行任何操作，首先我们需要访问底层的数值。 有几种方法可以做到这一点。有些比较简单，而另一些则比较通用。 下面的代码从第二个全连接层（即第三个神经网络层）提取偏置， 提取后返回的是一个参数类实例，并进一步访问该参数的值。

bias有两个属性，data获取值和grad获取梯度

In [6]:
```
print(type(net[2].bias))
print(net[2].bias)
print(net[2].bias.data)
```

```
<class 'torch.nn.parameter.Parameter'>
Parameter containing:
tensor([-0.0444], requires_grad=True)
tensor([-0.0444])
```

参数是复合的对象，包含值、梯度和额外信息。 这就是我们需要显式参数值的原因。 除了值之外，我们还可以访问每个参数的梯度。 在上面这个网络中，由于我们还没有调用反向传播，所以参数的梯度处于初始状态。

In [7]:
```
net[2].weight.grad == None
```

Out[7]:
```
True
```

[一次性访问所有参数]

当我们需要对所有参数执行操作时，逐个访问它们可能会很麻烦。 当我们处理更复杂的块（例如，嵌套块）时，情况可能会变得特别复杂， 因为我们需要递归整个树来提取每个子块的参数。 下面，我们将通过演示来比较访问第一个全连接层的参数和访问所有层。

通过获取名字来获取data

In [8]:
```
print(*[(name, param.shape) for name, param in net[0].named_parameters()])
print(*[(name, param.shape) for name, param in net.named_parameters()])
```

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

这为我们提供了另一种访问网络参数的方式，如下所示。

In [9]:
```
net.state_dict()['2.bias'].data
```

Out[9]:
```
tensor([-0.0444])
```

[从嵌套块收集参数]

让我们看看，如果我们将多个块相互嵌套，参数命名约定是如何工作的。 我们首先定义一个生成块的函数（可以说是“块工厂”），然后将这些块组合到更大的块中。

In [10]:
```
def block1():
    return nn.Sequential(nn.Linear(4, 8), nn.ReLU(),
                         nn.Linear(8, 4), nn.ReLU())

def block2():
    net = nn.Sequential()
    for i in range(4):
        # 在这里嵌套
        net.add_module(f'block {i}', block1())
    return net

rgnet = nn.Sequential(block2(), nn.Linear(4, 1))
rgnet(X)
```

Out[10]:
```
tensor([[-0.4881],
        [-0.4881]], grad_fn=<AddmmBackward0>)
```

[设计了网络后，我们看看它是如何工作的。]

In [11]:
```
print(rgnet)
```

```
Sequential(
  (0): Sequential(
    (block 0): Sequential(
      (0): Linear(in_features=4, out_features=8, bias=True)
      (1): ReLU()
      (2): Linear(in_features=8, out_features=4, bias=True)
      (3): ReLU()
    )
    (block 1): Sequential(
      (0): Linear(in_features=4, out_features=8, bias=True)
      (1): ReLU()
      (2): Linear(in_features=8, out_features=4, bias=True)
      (3): ReLU()
    )
    (block 2): Sequential(
      (0): Linear(in_features=4, out_features=8, bias=True)
      (1): ReLU()
      (2): Linear(in_features=8, out_features=4, bias=True)
      (3): ReLU()
    )
    (block 3): Sequential(
      (0): Linear(in_features=4, out_features=8, bias=True)
      (1): ReLU()
      (2): Linear(in_features=8, out_features=4, bias=True)
      (3): ReLU()
    )
  )
  (1): Linear(in_features=4, out_features=1, bias=True)
)
```

因为层是分层嵌套的，所以我们也可以像通过嵌套列表索引一样访问它们。 下面，我们访问第一个主要的块中、第二个子块的第一层的偏置项。

In [12]:
```
rgnet[0][1][0].bias.data
```

Out[12]:
```
tensor([ 0.0241,  0.0280,  0.4908,  0.1212, -0.3271, -0.4729,  0.4545, -0.1399])
```

参数初始化

知道了如何访问参数后，现在我们看看如何正确地初始化参数。 我们在 :numref: `sec_numerical_stability` 中讨论了良好初始化的必要性。 深度学习框架提供默认随机初始化， 也允许我们创建自定义初始化方法， 满足我们通过其他规则实现初始化权重。

默认情况下，PyTorch会根据一个范围均匀地初始化权重和偏置矩阵， 这个范围是根据输入和输出维度计算出的。 PyTorch的 `nn.init` 模块提供了多种预置初始化方法。

[内置初始化]

让我们首先调用内置的初始化器。 下面的代码将所有权重参数初始化为标准差为0.01的高斯随机变量， 且将偏置参数设置为0。

In [21]:
```
def init_normal(m):
    if type(m) == nn.Linear:
        nn.init.normal_(m.weight, mean=0, std=0.01)
        nn.init.zeros_(m.bias)
net.apply(init_normal)

# net = nn.Sequential(nn.Linear(4, 8), nn.ReLU(), nn.Linear(8, 1))
# 获取net的第一个输出特征（Linear(4, 8)的输出）的权重和偏置，8个输出特征分别有4个输入特征的权重
net[0].weight.data[0], net[0].bias.data[0]
```

Out[21]:
```
(tensor([-0.0003, -0.0101, -0.0043, -0.0096]), tensor(0.))
```

In [22]:
```
net[0].weight.data, net[0].bias.data
```

Out[22]:
```
(tensor([[-0.0003, -0.0101, -0.0043, -0.0096],
         [-0.0155, -0.0039, -0.0104, -0.0018],
         [ 0.0095,  0.0065, -0.0027, -0.0038],
         [-0.0009, -0.0090,  0.0038, -0.0097],
         [ 0.0034, -0.0137, -0.0195,  0.0160],
         [-0.0060,  0.0048,  0.0102, -0.0104],
         [ 0.0052, -0.0052,  0.0001,  0.0051],
         [ 0.0174, -0.0008,  0.0030,  0.0193]]),
 tensor([0., 0., 0., 0., 0., 0., 0., 0.]))
```

我们还可以将所有参数初始化为给定的常数，比如初始化为1。

In [23]:
```
def init_constant(m):
    if type(m) == nn.Linear:
        nn.init.constant_(m.weight, 1)
        nn.init.zeros_(m.bias)
net.apply(init_constant)
net[0].weight.data[0], net[0].bias.data[0]
```

Out[23]:
```
(tensor([1., 1., 1., 1.]), tensor(0.))
```

In [ ]:
```
```

我们还可以[**对某些块应用不同的初始化方法**]。 例如，下面我们使用Xavier初始化方法初始化第一个神经网络层， 然后将第三个神经网络层初始化为常量值42。

In [24]:
```
def init_xavier(m):
    if type(m) == nn.Linear:
        nn.init.xavier_uniform_(m.weight)
def init_42(m):
    if type(m) == nn.Linear:
        nn.init.constant_(m.weight, 42)

# net = nn.Sequential(nn.Linear(4, 8), nn.ReLU(), nn.Linear(8, 1))

net[0].apply(init_xavier)
net[2].apply(init_42)

#第一个层的输出的，第一层特征（共8层），的权重（4个输入特征）
print(net[0].weight.data[0])

#第三层的输出的，所有权重（8个输入特征）
print(net[2].weight.data)
```

```
tensor([-0.1569, -0.5829,  0.4882,  0.3071])
tensor([[42., 42., 42., 42., 42., 42., 42., 42.]])
```

In [ ]:
```

```

[自定义初始化]

有时，深度学习框架没有提供我们需要的初始化方法。 在下面的例子中，我们使用以下的分布为任意权重参数w定义初始化方法：

$$w \sim \begin{cases} U(5, 10) & \text{可能性 } \frac{1}{4} \\ 0 & \text{可能性 } \frac{1}{2} \\ U(-10, -5) & \text{可能性 } \frac{1}{4} \end{cases}$$

同样，我们实现了一个 `my_init` 函数来应用到 `net` 。

In [25]:
```
def my_init(m):
    if type(m) == nn.Linear:
        print("Init", *[(name, param.shape)
                        for name, param in m.named_parameters()][0])
        nn.init.uniform_(m.weight, -10, 10)
        m.weight.data *= m.weight.data.abs() >= 5

net.apply(my_init)

#第一个层的输出的，前2层特征的权重（4个输入特征）
net[0].weight[:2]
```

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

Out[25]:
```
tensor([[ 6.2512,  0.0000, -7.5172, -0.0000],
        [-0.0000,  6.5203,  0.0000, -7.2301]], grad_fn=<SliceBackward0>)
```

注意，我们始终可以直接设置参数。

In [26]:
```
net[0].weight.data[:] += 1
net[0].weight.data[0, 0] = 42
net[0].weight.data[0]
```

Out[26]:
```
tensor([42.0000,  1.0000, -6.5172,  1.0000])
```

[参数绑定]

有时我们希望在多个层间共享参数： 我们可以定义一个稠密层，然后使用它的参数来设置另一个层的参数。

In [ ]:
```
# 我们需要给共享层一个名称，以便可以引用它的参数
shared = nn.Linear(8, 8)
net = nn.Sequential(nn.Linear(4, 8), nn.ReLU(),
                    shared, nn.ReLU(),
                    shared, nn.ReLU(),
                    nn.Linear(8, 1))
net(X)
# 检查参数是否相同
print(net[2].weight.data[0] == net[4].weight.data[0])
net[2].weight.data[0, 0] = 100
# 确保它们实际上是同一个对象，而不只是有相同的值
print(net[2].weight.data[0] == net[4].weight.data[0])
```

这个例子表明第三个和第五个神经网络层的参数是绑定的。 它们不仅值相等，而且由相同的张量表示。 因此，如果我们改变其中一个参数，另一个参数也会改变。 这里有一个问题：当参数绑定时，梯度会发生什么情况？ 答案是由于模型参数包含梯度，因此在反向传播期间第二个隐藏层 （即第三个神经网络层）和第三个隐藏层（即第五个神经网络层）的梯度会加在一起。

小结

- 我们有几种方法可以访问、初始化和绑定模型参数。
- 我们可以使用自定义初始化方法。

练习

1. 使用 :numref: `sec_model_construction` 中定义的 `FancyMLP` 模型，访问各个层的参数。
2. 查看初始化模块文档以了解不同的初始化方法。
3. 构建包含共享参数层的多层感知机并对其进行训练。在训练过程中，观察模型各层的参数和梯度。
4. 为什么共享参数是个好主意？

Discussions (https://discuss.d2l.ai/t/1829)