参数初始化

参数初始化对模型具有较大的影响，不同的初始化方式可能会导致截然不同的结果，所幸的是很多深度学习的先驱们已经帮我们探索了各种各样的初始化方式，所以我们只需要学会如何对模型的参数进行初始化的赋值即可。

PyTorch 的初始化方式并没有那么显然，如果你使用最原始的方式创建模型，那么你需要定义模型中的所有参数，当然这样你可以非常方便地定义每个变量的初始化方式，但是对于复杂的模型，这并不容易，而且我们推崇使用 Sequential 和 Module 来定义模型，所以这个时候我们就需要知道如何来自定义初始化方式

使用 NumPy 来初始化

因为 PyTorch 是一个非常灵活的框架，理论上能够对所有的 Tensor 进行操作，所以我们能够通过定义新的 Tensor 来初始化，直接看下面的例子

```
import numpy as np
import torch
from torch import nn
```

```
# 定义一个 Sequential 模型
net1 = nn.Sequential(
    nn.Linear(30, 40),
    nn.ReLU(),
    nn.Linear(40, 50),
    nn.ReLU(),
    nn.Linear(50, 10)
)
```

```
# 访问第一层的参数
w1 = net1[0].weight
b1 = net1[0].bias
```

```
print(w1)
```

```
Parameter containing:
 0.1236 -0.1731 -0.0479  ...   0.0031  0.0784  0.1239
 0.0713  0.1615  0.0500  ...  -0.1757 -0.1274 -0.1625
 0.0638 -0.1543 -0.0362  ...   0.0316 -0.1774 -0.1242
          ...             ⋱              ...
 0.1551  0.1772  0.1537  ...   0.0730  0.0950  0.0627
 0.0495  0.0896  0.0243  ...  -0.1302 -0.0256 -0.0326
-0.1193 -0.0989 -0.1795  ...   0.0939  0.0774 -0.0751
[torch.FloatTensor of size 40x30]
```

注意，这是一个 Parameter，也就是一个特殊的 Variable，我们可以访问其 `.data` 属性得到其中的数据，然后直接定义一个新的 Tensor 对其进行替换，我们可以使用 PyTorch 中的一些随机数据生成的方式，比如 `torch.randn`，如果要使用更多 PyTorch 中没有的随机化方式，可以使用 numpy

```
# 定义一个 Tensor 直接对其进行替换
net1[0].weight.data = torch.from_numpy(np.random.uniform(3, 5, size=(40, 30)))
```

```
print(net1[0].weight)
```

```
Parameter containing:
 4.5768  3.6175  3.3098  ...   4.7374  4.0164  3.3037
 4.1809  3.5624  3.1452  ...   3.0305  4.4444  4.1058
 3.5277  4.3712  3.7859  ...   3.5760  4.8559  4.3252
          ...             ⋱              ...
 4.8983  3.9855  3.2842  ...   4.7683  4.7590  3.3498
 4.9168  4.5723  3.5870  ...   3.2032  3.9842  3.2484
 4.2532  4.6352  4.4857  ...   3.7543  3.9885  4.4211
[torch.DoubleTensor of size 40x30]
```

可以看到这个参数的值已经被改变了，也就是说已经被定义成了我们需要的初始化方式，如果模型中某一层需要我们手动去修改，那么我们可以直接用这种方式去访问，但是更多的时候是模型中相同类型的层都需要初始化成相同的方式，这个时候一种更高效的方式是使用循环去访问，比如

```
for layer in net1:
    if isinstance(layer, nn.Linear): # 判断是否是线性层
        param_shape = layer.weight.shape
        layer.weight.data = torch.from_numpy(np.random.normal(0, 0.5,
size=param_shape))
        # 定义为均值为 0，方差为 0.5 的正态分布
```

小练习：一种非常流行的初始化方式叫 Xavier，方法来源于 2010 年的一篇论文 [Understanding the difficulty of training deep feedforward neural networks]，其通过数学的推到，证明了这种初始化方式可以使得每一层的输出方差是尽可能相等的，有兴趣的同学可以去看看论文

我们给出这种初始化的公式

$$w \sim Uniform[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}] \tag{1}$$

其中 n_j 和 n_{j+1} 表示该层的输入和输出数目，所以请尝试实现以下这种初始化方式

对于 Module 的参数初始化，其实也非常简单，如果想对其中的某层进行初始化，可以直接像 Sequential 一样对其 Tensor 进行重新定义，其唯一不同的地方在于，如果要用循环的方式访问，需要介绍两个属性，children 和 modules，下面我们举例来说明

```
class sim_net(nn.Module):
    def __init__(self):
        super(sim_net, self).__init__()
        self.l1 = nn.Sequential(
            nn.Linear(30, 40),
            nn.ReLU()
        )

        self.l1[0].weight.data = torch.randn(40, 30) # 直接对某一层初始化

        self.l2 = nn.Sequential(
            nn.Linear(40, 50),
            nn.ReLU()
        )

        self.l3 = nn.Sequential(
            nn.Linear(50, 10),
            nn.ReLU()
        )

    def forward(self, x):
        x = self.l1(x)
        x =self.l2(x)
        x = self.l3(x)
        return x
```

```
net2 = sim_net()
```

```
# 访问 children
for i in net2.children():
    print(i)
```

```
Sequential(
  (0): Linear(in_features=30, out_features=40)
  (1): ReLU()
)
Sequential(
  (0): Linear(in_features=40, out_features=50)
  (1): ReLU()
)
Sequential(
  (0): Linear(in_features=50, out_features=10)
  (1): ReLU()
)
```

```
# 访问 modules
for i in net2.modules():
    print(i)
```

```
sim_net(
  (l1): Sequential(
    (0): Linear(in_features=30, out_features=40)
    (1): ReLU()
  )
  (l2): Sequential(
    (0): Linear(in_features=40, out_features=50)
    (1): ReLU()
  )
  (l3): Sequential(
    (0): Linear(in_features=50, out_features=10)
    (1): ReLU()
  )
)
Sequential(
  (0): Linear(in_features=30, out_features=40)
  (1): ReLU()
)
Linear(in_features=30, out_features=40)
ReLU()
Sequential(
```

```
  (0): Linear(in_features=40, out_features=50)
  (1): ReLU()
)
Linear(in_features=40, out_features=50)
ReLU()
Sequential(
  (0): Linear(in_features=50, out_features=10)
  (1): ReLU()
)
Linear(in_features=50, out_features=10)
ReLU()
```

通过上面的例子，看到区别了吗?

children 只会访问到模型定义中的第一层，因为上面的模型中定义了三个 Sequential，所以只会访问到三个 Sequential，而 modules 会访问到最后的结构，比如上面的例子，modules 不仅访问到了 Sequential，也访问到了 Sequential 里面，这就对我们做初始化非常方便，比如

```
for layer in net2.modules():
    if isinstance(layer, nn.Linear):
        param_shape = layer.weight.shape
        layer.weight.data = torch.from_numpy(np.random.normal(0, 0.5,
size=param_shape))
```

这上面实现了和 Sequential 相同的初始化，同样非常简便

torch.nn.init

因为 PyTorch 灵活的特性，我们可以直接对 Tensor 进行操作从而初始化，PyTorch 也提供了初始化的函数帮助我们快速初始化，就是 `torch.nn.init`，其操作层面仍然在 Tensor 上，下面我们举例说明

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

```
print(net1[0].weight)
```

```
Parameter containing:
 0.8453  0.2891 -0.5276  ...  -0.1530 -0.4474 -0.5470
-0.1983 -0.4530 -0.1950  ...   0.4107 -0.4889  0.3654
 0.9149 -0.5641 -0.6594  ...   0.0734  0.1354 -0.4152
          ...             ⋱             ...
-0.4718 -0.5125 -0.5572  ...   0.0824 -0.6551  0.0840
-0.2374 -0.0036  0.6497  ...   0.7856 -0.1367 -0.8795
 0.0774  0.2609 -0.2358  ...  -0.8196  0.1696  0.5976
[torch.DoubleTensor of size 40x30]
```

```
init.xavier_uniform(net1[0].weight) # 这就是上面我们讲过的 Xavier 初始化方法，PyTorch 直接内置了其实现
```

```
Parameter containing:
-0.2114  0.2704 -0.2186  ...   0.1727  0.2158  0.0775
-0.0736 -0.0565  0.0844  ...   0.1793  0.2520 -0.0047
 0.1331 -0.1843  0.2426  ...  -0.2199 -0.0689  0.1756
          ...             ⋱             ...
 0.2751 -0.1404  0.1225  ...   0.1926  0.0175 -0.2099
 0.0970 -0.0733 -0.2461  ...   0.0605  0.1915 -0.1220
 0.0199  0.1283 -0.1384  ...  -0.0344 -0.0560  0.2285
[torch.DoubleTensor of size 40x30]
```

```
print(net1[0].weight)
```

```
Parameter containing:
-0.2114  0.2704 -0.2186  ...   0.1727  0.2158  0.0775
-0.0736 -0.0565  0.0844  ...   0.1793  0.2520 -0.0047
 0.1331 -0.1843  0.2426  ...  -0.2199 -0.0689  0.1756
          ...             ⋱             ...
 0.2751 -0.1404  0.1225  ...   0.1926  0.0175 -0.2099
 0.0970 -0.0733 -0.2461  ...   0.0605  0.1915 -0.1220
 0.0199  0.1283 -0.1384  ...  -0.0344 -0.0560  0.2285
[torch.DoubleTensor of size 40x30]
```

可以看到参数已经被修改了

`torch.nn.init` 为我们提供了更多的内置初始化方式，避免了我们重复去实现一些相同的操作

上面讲了两种初始化方式，其实它们的本质都是一样的，就是去修改某一层参数的实际值，而 `torch.nn.init` 提供了更多成熟的深度学习相关的初始化方式，非常方便

下一节课，我们将讲一下目前流行的各种基于梯度的优化算法