Deep Learning Computation

5.1 Layers and Blocks

```
import torch
from torch import nn
from torch.nn import functional as F

net = nn.Sequential(nn.Linear(20, 256), nn.ReLU(), nn.Linear(256, 10))

X = torch.rand(2, 20)#2行20列的张量，值为[0,1)内的随机数
#print(net(X))

#自定义块
class MLP(nn.Module):
    # 用模型参数声明层，这里声明两个全连接层
    def __init__(self):
        # 调用MLP的父类Module的构造函数来执行必要的初始化。
        # 这样，在类实例化时也可以指定其他函数参数，例如模型参数params（稍后将介绍）
        super().__init__()
        self.hidden = nn.Linear(20, 256)  # 隐藏层
        self.out = nn.Linear(256, 10)  # 输出层

    # 定义模型的前向传播，即如何根据输入X返回所需的模型输出
    def forward(self, X):
        # 这里使用ReLU的函数版本，其在nn.functional模块中定义。
        return self.out(F.relu(self.hidden(X)))

net = MLP()
print(net(X))

#自定义顺序块
class MySequential(nn.Module):
    def __init__(self, *args):
        super().__init__()
        for idx, module in enumerate(args):
        #module是Module子类的一个实例,保存在'Module'类的成员变量_modules中。_module的
            self._modules[str(idx)] = module
```

```
    def forward(self, X):
        # OrderedDict保证了按照成员添加的顺序遍历它们
        for block in self._modules.values():
            X = block(X)
        return X

net = MySequential(nn.Linear(20, 256), nn.ReLU(), nn.Linear(256, 10))
print(net(X))

#自定义权重为常数的隐藏层
class FixedHiddenMLP(nn.Module):
    def __init__(self):
        super().__init__()
        # 不计算梯度的随机权重参数。因此其在训练期间保持不变
        self.rand_weight = torch.rand((20, 20), requires_grad=False)
        self.linear = nn.Linear(20, 20)

    def forward(self, X):
        X = self.linear(X)
        # 使用创建的常量参数以及relu和mm函数
        X = F.relu(torch.mm(X, self.rand_weight) + 1)
        # 复用全连接层。这相当于两个全连接层共享参数
        X = self.linear(X)
        # 下面代码演示如何把代码集成到网络计算流程中
        while X.abs().sum() > 1:
            X /= 2
        return X.sum()

net = FixedHiddenMLP()
print(net(X))

#嵌套块
class NestMLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(nn.Linear(20, 64), nn.ReLU(),
                                 nn.Linear(64, 32), nn.ReLU())
        self.linear = nn.Linear(32, 16)

    def forward(self, X):
        return self.linear(self.net(X))
```

```
chimera = nn.Sequential(NestMLP(), nn.Linear(16, 20), FixedHiddenMLP())
print(chimera(X))
```

5.2 Parameter Management

```
import torch
from torch import nn

net = nn.Sequential(nn.Linear(4, 8), nn.ReLU(), nn.Linear(8, 1))
X = torch.rand(size=(2, 4))
#print(net(X))

#查看第二个全连接层的参数
print(net[2].state_dict())
print(type(net[2].bias))
print(net[2].bias)
print(net[2].bias.data)
print(net.state_dict()['2.bias'].data)#此行作用和上行相同

#访问第一个全连接层的参数和访问所有层
print(*[(name, param.shape) for name, param in net[0].named_parameters()])
print(*[(name, param.shape) for name, param in net.named_parameters()])
#由于没有在nn.Sequential中明确指定ReLU层的权重和偏置，因此它们在输出中没有被显示

#从嵌套块收集参数
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))
print(rgnet(X))
print(rgnet)
print(rgnet[0][1][0].bias.data)

#用内置函数进行参数初始化
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)
print(net[0].weight.data[0], net[0].bias.data[0])

def init_constant(m):
    if type(m) == nn.Linear:
        nn.init.constant_(m.weight, 1)#初始化参数为常数1
        nn.init.zeros_(m.bias)
net.apply(init_constant)
print(net[0].weight.data[0], net[0].bias.data[0])

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[0].apply(init_xavier)
net[2].apply(init_42)
print(net[0].weight.data[0])
print(net[2].weight.data)##只有一层
```

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：

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

```
    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)
    print(net[0].weight[:2])
    net[0].weight.data[:] += 1
    net[0].weight.data[0, 0] = 42
    print(net[0].weight.data[0])
```

```
#参数绑定
#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])
```

注：在PyTorch中，模型的权重通常在实例化时就进行初始化，但有时候我们希望将权重的初始化推迟到模型第一次被调用的时候(比如有些模型的输入尺寸只有在实际输入数据时才能确定),这时候框架会自动使用*延后初始化*（deferred initialization）来解决这个问题。

5.3 Custom Layers

```
import torch
from torch import nn
from torch.nn import functional as F

net = nn.Sequential(nn.Linear(20, 256), nn.ReLU(), nn.Linear(256, 10))

X = torch.rand(2, 20)#2行20列的张量，值为[0,1)内的随机数
#print(net(X))

#自定义块
class MLP(nn.Module):
    # 用模型参数声明层，这里声明两个全连接层
    def __init__(self):
        # 调用MLP的父类Module的构造函数来执行必要的初始化。
        # 这样，在类实例化时也可以指定其他函数参数，例如模型参数params（稍后将介绍）
        super().__init__()
        self.hidden = nn.Linear(20, 256)  # 隐藏层
        self.out = nn.Linear(256, 10)  # 输出层

    # 定义模型的前向传播，即如何根据输入X返回所需的模型输出
    def forward(self, X):
        # 这里使用ReLU的函数版本，其在nn.functional模块中定义。
        return self.out(F.relu(self.hidden(X)))

net = MLP()
print(net(X))

#自定义顺序块
class MySequential(nn.Module):
    def __init__(self, *args):
        super().__init__()
        for idx, module in enumerate(args):
        #module是Module子类的一个实例,保存在'Module'类的成员变量_modules中。_module的
            self._modules[str(idx)] = module

    def forward(self, X):
        # OrderedDict保证了按照成员添加的顺序遍历它们
        for block in self._modules.values():
            X = block(X)
        return X
```

```
net = MySequential(nn.Linear(20, 256), nn.ReLU(), nn.Linear(256, 10))
print(net(X))

#自定义权重为常数的隐藏层
class FixedHiddenMLP(nn.Module):
    def __init__(self):
        super().__init__()
        # 不计算梯度的随机权重参数。因此其在训练期间保持不变
        self.rand_weight = torch.rand((20, 20), requires_grad=False)
        self.linear = nn.Linear(20, 20)

    def forward(self, X):
        X = self.linear(X)
        # 使用创建的常量参数以及relu和mm函数
        X = F.relu(torch.mm(X, self.rand_weight) + 1)
        # 复用全连接层。这相当于两个全连接层共享参数
        X = self.linear(X)
        # 下面代码演示如何把代码集成到网络计算流程中
        while X.abs().sum() > 1:
            X /= 2
        return X.sum()

net = FixedHiddenMLP()
print(net(X))

#嵌套块
class NestMLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(nn.Linear(20, 64), nn.ReLU(),
                                 nn.Linear(64, 32), nn.ReLU())
        self.linear = nn.Linear(32, 16)

    def forward(self, X):
        return self.linear(self.net(X))

chimera = nn.Sequential(NestMLP(), nn.Linear(16, 20), FixedHiddenMLP())
print(chimera(X))#由于可能两个维度的计算结果都小于等于0，因此结果可能是tensor([[0.],[0
```

5.4 File I/O

```
import torch
from torch import nn
from torch.nn import functional as F

x = torch.arange(4)
torch.save(x, 'x-file')
x2 = torch.load('x-file')
print(x2)

y = torch.zeros(4)
torch.save([x, y],'x-files')
x2, y2 = torch.load('x-files')
print(x2, y2)

mydict = {'x': x, 'y': y}
torch.save(mydict, 'mydict')
mydict2 = torch.load('mydict')
print(mydict2)

class MLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.hidden = nn.Linear(20, 256)
        self.output = nn.Linear(256, 10)

    def forward(self, x):
        return self.output(F.relu(self.hidden(x)))

net = MLP()
X = torch.randn(size=(2, 20))
Y = net(X)

torch.save(net.state_dict(), 'mlp.params')#保存模型参数
clone = MLP()
clone.load_state_dict(torch.load('mlp.params'))
print(clone.eval())
#clone.eval()的目的是切换到评估模式，以确保在加载完模型参数后，模型的行为与推断时一致。
#在训练模式下，某些层的行为可能会导致不同的输出，因此通过切换到评估模式来避免这种不一致性
Y_clone = clone(X)
```

```
print(Y_clone)
print(Y_clone == Y)
```

5.5 GPU Management

```
import torch
from torch import nn

print(torch.__version__)#查看torch版本
print(torch.version.cuda)#查看torch适配的cuda版本
print(torch.cuda.is_available())#查看cuda是否和torch适配
print(torch.device('cpu'), torch.device('cuda'), torch.device('cuda:1'))
print(torch.cuda.device_count())#查询可用gpu的数量

def try_gpu(i=0):  #@save
    """如果存在，则返回gpu(i)，否则返回cpu()"""
    if torch.cuda.device_count() >= i + 1:
        return torch.device(f'cuda:{i}')
    return torch.device('cpu')

def try_all_gpus():  #@save
    """返回所有可用的GPU，如果没有GPU，则返回[cpu(),]"""
    devices = [torch.device(f'cuda:{i}')
            for i in range(torch.cuda.device_count())]
    return devices if devices else [torch.device('cpu')]

print(try_gpu(), try_gpu(10), try_all_gpus())

x = torch.tensor([1, 2, 3])#张量是默认在CPU上创建的
print(x.device)
X = torch.ones(2, 3, device=try_gpu())
print(X)
Y = torch.rand(2, 3, device=try_gpu(1))
print(Y)

#注意，一般电脑只有一个gpu,因此在下行会报错
Z = X.cuda(1)#在gpu(1)创建X的一个副本Z
print(Z)

net = nn.Sequential(nn.Linear(3, 1))
net = net.to(device=try_gpu())#将模型参数放在GPU上
print(net(X))
print(net[0].weight.data.device)
```