深度学习计算

> 在本章中，我们将深入探索深度学习计算的关键组件，即模型构建、参数访问与初始化、设计自定义层和块、将模型读写到磁盘，以及利用GPU实现显著的加速。这些知识将使读者从深度学习“基础用户”变为“高级用户”。

1.层和块

为了实现、描述复杂的网络，我们引入 神经网络块 的概念。
块（block）可以描述单个层、由多个层组成的组件或整个模型本身。

使用块的好处：递归实现，可以用简洁的代码实现复杂的神经网络结构。

- 块的任何子类都必须定义一个将其输入转换为输出的前向传播函数，并且必须存储任何必需的参数。
  同时，为了计算梯度，每一个块都需要定义**反向传播函数。**
- 接下来我们要在之前**mlp**的基础上自定义块来实现。对于之前mlp整个结构的概述在P192页下方。

1.1自定义块

- 我们**从零开始编写一个块**。它包含一个多层感知机，其具有256个隐藏单元的隐藏层和一个10维输出层。注意，下面的MLP类继承了表示块的类。我们的实现只需要提供我们自己的构造函数（Python中的__init__函数）和前向传播函数。

```
import torch
import torch
from torch import nn
from torch.nn import functional as F
print('ok')

class MLP(nn.Module):
      # 用模型参数声明层。这里，我们声明两个全连接的层
      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) #隐藏层次
            self.out = nn.Linear(256,10) #输出层

      def forward(self,X):
            #X就是输入的参数，self.hidden(X) 这个整体就是经过第一个线性层之后的参数，F.relu(...)的结果就是第一个线性层的结果再加上激活函数。
            #最后 self.out(...)就是把经过激活函数的结果再进行输出层处理之后得到的结果。
            return self.out(F.relu(self.hidden(X)))
```

- 代码里面forward前向传播函数中将整个过程描述了一遍。
- 我们定义的class继承了nn.Module类，我们定制的__init__函数通过super().__init__() 调用父类的__init__函数，省去了重复编写模版代码。

对于**反向传播的说明:**除非我们实现一个新的运算符，否则我们不必担心反向传播函数或参数初始化，系统将自动生成这些。

运行代码：

```
net = MLP()
X = torch.rand(2, 20)
net(X)
```

1.2实现顺序块

上面的class MLP函数里面中在forward函数里面，直接将整个网络的所有层次都调用上进行计算，如果一直用这个策略，一个100多层的网络结构就需要自己手动添加100次的操作和初始化。**所以我们实现顺序块来简化这个过程：**
1.实现可以把每一部分添加到网络结构的函数;
2.按照顺序一部分一部分进行**前向传播。**

> 这里介绍两种实现方式：

- 第一种(`动手学习深度学习里面的方式`)：
  声明一个类，继承`nn.Module`类，把所有的神经网络结构存储在变量`_modules`中，`_modules`的类型是一个`OrderedDict`有顺序的字典，因为有顺序所有也方便我们可以在forward中直接用for循环**按照顺序遍历。**

```
class Mysequential(nn.Module):
      def __init__(self,*args):
            super().__init__()
            for idx, module in enumerate(args):
                  # 这里，module是Module子类的一个实例。我们把它保存在'Module'类的成员
                  # 变量_modules中。_module的类型是OrderedDict
                  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
```

  - 上面就是整个声明，运行的时候：

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

    第一行就初始化、声明了整个神经网络的结构，net(X)会调用Module里面的`__call__`函数，之后会调用forward函数进行整个网络结构的传播。

- 第二种：（**链接**）

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

class MyNet(torch.nn.Module):
      def __init__(self):
            super(MyNet, self).__init__()    # 第一句话，调用父类的构造函数
            self.conv1 = torch.nn.Conv2d(3, 32, 3, 1, 1)
            self.conv2 = torch.nn.Conv2d(3, 32, 3, 1, 1)

```

```
            self.dense1 = torch.nn.Linear(32 * 3 * 3, 128)
            self.dense2 = torch.nn.Linear(128, 10)

    def forward(self, x):
            x = self.conv1(x)
            x = F.relu(x)
            x = F.max_pool2d(x)
            x = self.conv2(x)
            x = F.relu(x)
            x = F.max_pool2d(x)
            x = self.dense1(x)
            x = self.dense2(x)
            return x

model = MyNet()
print(model)
'''运行结果为:
MyNet(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (dense1): Linear(in_features=288, out_features=128, bias=True)
  (dense2): Linear(in_features=128, out_features=10, bias=True)
)
'''
```

在整个net里面声明结构。forward里面每一步都按照顺序来。

- 1.**relu函数，这种参数不会变化的函数可以在net里面声明，也可以不声明，使用的时候直接调用也可以。**
  2.声明方式还有很多种，这里只是放了一个最 `简单` ，容易理解的方式， **链接** 里面有很多其他的实现方式。

1.3在前向传播函数里面执行代码

并不是所有的架构都是简单的顺序架构。当需要更强的灵活性时，我们需要定义自己的块。例如，我们可能希望在前向传播函数中执行Python的控制流。

- 在这种情况下，就可以自己重写 `net` 里面的forward函数。

2.参数管理

> 可以用print(net)直接获取到 **整个net的结构** 。

2.1参数的访问：打印网络权重

对于一个

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

的结构，对于第i层，可以使 `net[i].state_dict()` 函数，会返回这一层次 `所有参数` 组成的一个字典。
比如： `print(net[2].state_dict())` 之后，得到：

```
OrderedDict([('weight', tensor([[ 0.0056, -0.3223,  0.2429,  0.1277, -0.0788,  0.2913,  0
.0408, -0.0469]])), ('bias', tensor([0.0869]))])
```

- **一次性访问所有的参数：**
  使用`print(*[(name, param.shape) for name, param in net.named_parameters()])`
  其中，`net.named_parameters()`：这个方法返回模型`net`中所有参数的迭代器，返回的每个元素是一个元组`(name, parameter)`。
  `name`：参数的名称（例如：`'weight'`、`'bias'`等），它是一个字符串。
  `parameter`：对应的`torch.Tensor`对象，包含该参数的值。

2.2 参数初始化

- 说明一下apply函数的作用：（基本初始化一定会用到这个函数）
  在 PyTorch 中，`apply()`是`torch.nn.Module`类的一个方法，它用于将一个函数应用到模型中的所有子模块（layers）。**也就是整个模型的所有模块，就会进行对应的apply函数处理。**
  具体例子见下面使用内置初始化函数初始化权重。
- 以下初始化都是在net上进行的。
  net的结构：

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

- **内置初始化**：
  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。

```
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)
```

  apply函数会使得整个net的所有模块都处理一次`init_normal()`函数。

- 也可以将不同块进行不同的初始化方式：

```
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)
```

  对于第一个层次和第三个层次的初始化，采用的是不同的方式。

2.3参数绑定、共享层次

实现一个net，其中有两个层次都是线性层，size都是(8，8)。实现这两个层次完全一样。（**两个实际上就是一个对象，而不仅仅是值一样**）。

```
# 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)
```

不过还没有研究，之后如果进行梯度下降更新参数，这里会怎么更新。

3.延后初始化

在实际的运行过程中，经常出现第一层次的输入维度并不会在 定义网络结构 的时候就直接给出。这种情况下，并不会在一开始就进行网络参数的初始化，而是有数据输入之后，才会进行网络参数的初始化。
即直到数据第一次通过模型传递时，框架才会动态地推断出每个层的大小。

4.构建自定义层

深度学习成功背后的一个因素是神经网络的灵活性：我们可以用创造性的方式组合不同的层，从而设计出适用于各种任务的架构。
必须要学会如何自定义层，因为一定会遇到很多目前架构中尚未存在的层。

> 接下来说明如何构建一个带参数的自定义层次。

- 实现自定义版本的全连接层。回想一下，该层需要两个参数，一个用于表示权重，另一个用于表示偏置项。在此实现中，我们使用修正线性单元作为激活函数。该层需要输入参数：in_units和units，分别表示输入数和输出数。
- 模型定义：

```
class MyLinear(nn.Module):
      def __init__(self, in_units, units):
            super().__init__()
            self.weight = nn.Parameter(torch.randn(in_units, units))
            self.bias = nn.Parameter(torch.randn(units,))
      def forward(self, X):
            linear = torch.matmul(X, self.weight.data) + self.bias.data
            return F.relu(linear)
```

  - 实例化一个网络结构的实体：
    随机构造出数据，进行forward:

```
linear = MyLinear(5, 3) #linear就是一个我们定义的网络结构的实体。
X = torch.rand(2,5)
linear(X) #调用前向传播
```

- 上面已经实现了自己定义一个层次，可以借助Sequential来和其他层次连接在一起：

```
net = nn.Sequential(MyLinear(64, 8), MyLinear(8, 1))
net(torch.rand(2, 64))
```

5.读写文件

有时我们**希望保存训练的模型**，以备将来在各种环境中使用（比如在部署中进行预测）。此外，当运行一个耗时较长的训练过程时，最佳的做法是定期保存中间结果，以确保在服务器电源被不小心断掉时，我们不会损失几天的计算结果。

- 所以这一节课我们学习一下如何：**加载和存储权重向量和整个模型**了。

5.1学习使用load和save来保存、加载张量/张量列表/字典

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

x = torch.arange(4)#生成一个维度为4的张量
torch.save(x,'x-file')#调用save函数，把这个张量保存到x-file文件里面。
#同时，会在当前目录下创建一个x-file的文件

#读取：
x2 = torch.load('x-file') #使用load函数，就可以做到从对应的位置加载数据。
print(x2)
```

```
tensor([0, 1, 2, 3])
```

- 对于张量列表的save 和 load:

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

```
(tensor([0, 1, 2, 3]), tensor([0., 0., 0., 0.]))
```

- 对于字典mydict的save和load:

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

```
{'x': tensor([0, 1, 2, 3]), 'y': tensor([0., 0., 0., 0.])}
```

5.2保存和加载整个模型

> 上面已经学习过如何save单个变量，如果网络结构也使用单个变量进行保存的形式，会很麻烦。
>
> **深度学习框架提供了内置函数来保存和加载整个网络。**
> 注意：这里只是保存模型的参数，而不是保存完整的模型

下面通过实际的例子，save一个mlp网络结构，然后加载。**保存、加载模型**

- 自定义一个MLP结构：

```python
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() #实例化一个对象
```

- 将mlp的参数保存到文件里：

```python
torch.save(net.state_dict(), 'mlp.params')
```

- 再实例化一个MLP的对象，然后从mlp.params里面load模型的参数：

```python
clone = MLP()
clone.load_state_dict(torch.load('mlp.params'))
```

- 此时 clone 和 net 在结构和参数上都是完全一致的。
  如果我们随机生成数据跑两个模型，结果也会是一摸一样。

```python
X = torch.randn(size = (2,20))
Y = net(X)
YY = clone(X)
print(Y == YY)
```

```
tensor([[True, True, True, True, True, True, True, True, True, True],
        [True, True, True, True, True, True, True, True, True, True]])
```

6.GPU

- 可以使用`nvidia-smi`命令查询当前电脑的gpu情况。

```
(chatglm3) D:\Users\gxy\OneDrive\directory\graduate_study\DEEP_learning\Code_of_book\pytorch\
Sun Nov 10 15:35:52 2024
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 560.94                 Driver Version: 560.94         CUDA Version: 12.6     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                  Driver-Model | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce RTX 4060 Ti   WDDM  |   00000000:01:00.0  On |                  N/A |
|  0%   37C    P8             7W /  165W |     698MiB /  16380MiB |      2%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+
```

我的电脑上只有一块。
后面有很多两个gpu之间相互传输的操作，都做不了。

6.1计算设备的概念

- 在PyTorch中，每个数组都有一个设备（device），我们通常将其称为环境（context）。默认情况下，所有变量和相关的计算都分配给CPU。有时环境可能是GPU。当我们跨多个服务器部署作业时，事情会变得更加棘手。
- 查看当前设备的cpu gpu情况：
  CPU和GPU可以用torch.device('cpu') 和torch.device('cuda')表示。
  如果有多个GPU，我们使用torch.device(f'cuda:{i}') 来表示第 i 块GPU（i 从0开始）。另外，cuda:0和cuda是等价的。
  - `torch.device.count()` 函数可以返回当前计算机内部可用的gpu的数目

```
>>> torch.cuda.device_count()
1
```

  - 以下两个函数，可以帮助判断机子是否有对应的运行环境：

```
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')]

try_gpu(), try_gpu(10), try_all_gpus()
```

    如果当前可以，就会返回对应的设备。

6.2张量与GPU：

- 需要注意的是，无论何时我们要对多个项进行操作，它们都必须在同一个设备上。例如，如果我们对两个张量求和，我们需要确保两个张量都位于同一个设备上，否则框架将不知道在哪里存储结果，甚至不知道在哪里执行计算。
- 用实际的例子演示一个过程：（假如我们的机子上有>=2数量的gpu，是可以运行成功的）
  - 在第一个gpu上存储张量X：

```
X = torch.ones(2, 3, device=try_gpu())
```

```
tensor([[1., 1., 1.],
        [1., 1., 1.]], device='cuda:0')
```

  - 在第二个gpu上面存储张量Y

```
Y = torch.rand(2, 3, device=try_gpu(1))
```

```
tensor([[0.4860, 0.1285, 0.0440],
        [0.9743, 0.4159, 0.9979]], device='cuda:1')
```

- 此时，如果要计算X+Y，会失败。
  因为两个不在同一个设备上。
  必须**将一个张量复制到另外一个gpu里面，才可以进行计算操作。**

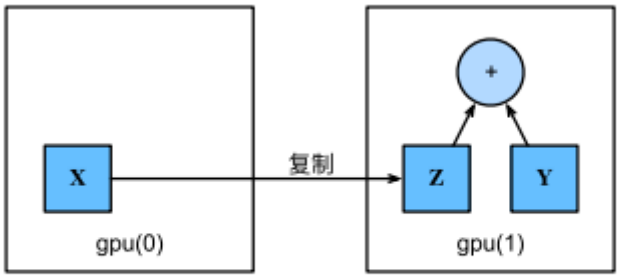

图5.6.1: 复制数据以在同一设备上执行操作

```
Z = X.cuda(1) #实现在第二张gpu上面复制一份X
#此时 运行 Y+Z 就可以运行成功
```