一、前言

本分析报告的作者是中国科学院大学 2019 级本科生夏瑞阳，本报告是作者在《面向对象程序设计》课程中选取 PyTorch 的 nn 模块进行分析所完成的作业，因此本报告将不会把篇幅过多放在 PyTorch 具体源码实现的解析与算法设计的讨论，而将更注重于主要功能流程设计、类的设计与类间关系、面向对象的设计原则和设计模式分析等方面。

二、导言

导言中将简要介绍 PyTorch 与 Torch.nn 的主要功能、流程与模块。

1、什么是 PyTorch

PyTorch 是 Python 的一个深度学习库，是一个能够在 CPU 或 GPU 上运行并解决各类深度学习问题的深度学习框架。相较于其他深度学习框架，它具有以下几个优势：

①简洁：

PyTorch 的原身是 Lua 语言上的 Torch，因此 PyTorch 继承了 Torch 的设计原则，追求最少的封装，尽量避免重复造轮子。与另一个热门的深度学习框架 TensorFlow 相比，PyTorch 中没有 session、graph、tensor、layer、name_scope、operation 等抽象层次，而是只有 Tensor->Variable->Module 的由低到高的抽象层次。因此这使得 PyTorch 的源代码非常易读。

②速度：

PyTorch 能提供强大的 GPU 加速和更高效的算法来支持网络中的计算，在许多评测中，PyTorch 的速度表现胜过 TensorFlow 和 Keras 等深度学习框架。

③易用：

PyTorch 能基于磁带式的自动微分系统来构建一个深度神经网络。即设计者只需要自己重载 Module 类的 __init__() 和 forward() 方法，而网络反向传播的部分由 torch.autograd 模块自动完成，无需设计者再手动实现。此外，它也能与 NumPy、SciPy、Cython 等 Python 库很好地兼容，从而使得设计者能够复用他们以前的代码。

Pytorch 中除了 nn、autograd 等模块外，还有许多常用模块，现介绍如下：

名称	描述
torch	一个有着强大 GPU 支持的张量库，是 Pytorch 的基本组件
torch.autograd	一个磁带式的自动微分系统，能支持 torch 内所有张量可微操作
torch.jit	一个通过 Pytorch 源码生成可序列化和可优化模型的编译栈
torch.nn	一个与自动微分系统深度集成的神经网络库，十分灵活便捷
torch.multiprocessing	一个支持多进程的库，并且有着强大的张量间内存共享功能
torch.utils	一个工具库，其中有 DataLoader 等便利性的工具

2、PyTorch.nn 中所包含的主要类

torch.nn 中定义了许多个类，其中主要有以下几个：Module、Parameter、Functional、grad、init、utils 等。在之后的报告中会基于这些类对 pytorch.nn 进行详细分析。

三、Pytorch.nn 的工作流程

要分析清楚 pytorch.nn 的内容和代码的设计意图，就必须先明白一个传统的神经网络在设计、搭建、训练、更新参数、导入导出参数、结果检验等方面的工作流程，以及 pytorch 是如何完成这些工作的，每个部分是通过哪些组件配合来完成的。

神经网络的建立从小到大的层次遵循以下的顺序：算子(Operator)->层(Layer)->网络(Network)。一些常用的算子包括四则运算、开方乘方、指数运算、张量操作等，对于某些常用、固定的算子组成的序列，可以将这些算子封装为一个运算层以使用。随后，将这些运算层连接起来，就完成了神经网络结构的搭建。

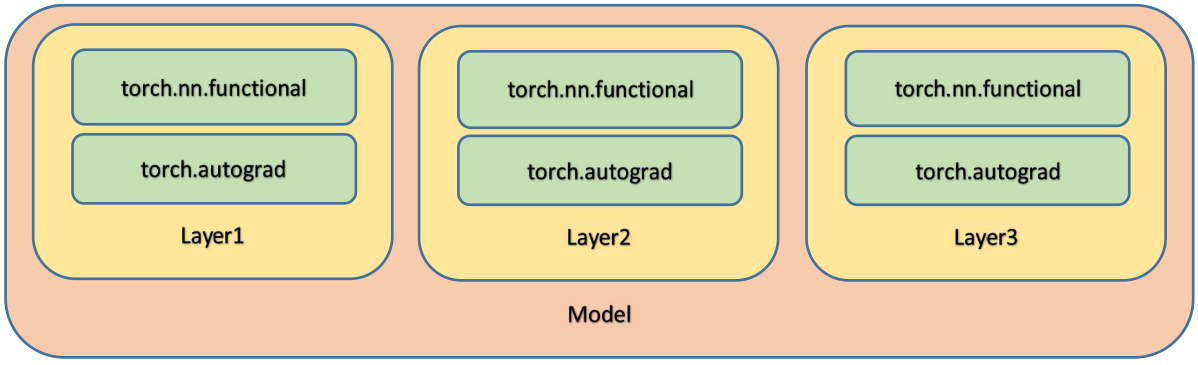

图 1-算子-运算层-神经网络层次结构示意图

pytorch 中有官方自带的常用算子，通常定义在 nn.functional 类中。pytorch 中也有官方自带的运算层，比如 Conv2d、BatchNorm1d、Hardtanh、Linear 等，也可以通过继承基类 Module 的方式自定义运算层。

神经网络搭建完成后，需要参数的初始化。考虑到权重参数的分布会对训练的精度产生一定影响，因此 nn.init 类中能够通过多种方法设置对不同层、不同数据的不同随机方法，以使参数能分布在期望的空间上。

网络中的参数分为可训练的和不可训练的两类，对于官方给出的运算层中的参数，torch 会自动将需要训练的参数加入迭代器 Parameter()中，并在之后的反向传播过程中进行更新，而对于那些自定义运算层中的参数，则可以通过 nn.Parameter 类来进行标记，将其加入到 Module 的迭代器 Parameter()中。

对于训练集和验证集的数据文件，可以通过 torch.utils 中的 dataloader 类完成这一工作，这个类中包含许多常用的数据集的下载方法，如 MNIST、ImageNet 等，在本地未找到数据集文件时会自动进行下载，并且类内部的多进程设置能够加快下载、加载速度。并且 dataloader 类内维护了一个生成器，会随着训练进行不断分批将数据加载进内存之中，从而减少数据集文件对内存的占用。最后 dataloader 类内还有许多常用的数据预处理方法，方便使用者直接调用。

运算层 Layer 和模型 Model 都是继承了 nn.Module 的子类。在继承 nn.Module 时，子类需要重写父类的 forward 方法，即主动设置运算层或者网络模型的前向传播运算顺序和连接次序，否则会调用基类原本的 forward 方法从而造成一个错误调用。前向传播运算完成后会根据比较结果进行反向传播并更新模型中的参数，但值得注意的是，pytorch 中并不需要手动为每个层/算子实现其 backward()的反向传播部分，这部分工作由 torch.autograd 模块完成，autograd 会维护一个磁带式的记录系统，反向传播部分需要进行梯度下降时，会将记录的磁带记录“倒带”，从而完成各个运算层参数的更新。由于 torch.autograd 和 torch.nn 分属两个不同的模块，因此本源码分析中并不会对自动微分的部分进行过多的涉及，只需要了解神经网络的模型在搭建过程中无需使用者手动完成反向传播函数的部分，而是由 pytorch 自动完成即可。

四、nn.module 及其子类的分析

nn.module 模块内部主要定义了 nn.module 类，并且 pytorch 官方还基于 nn.module 类生成了许多常用的子类，如_ConvNd、_MaxPoolNd 等

1、module.py 的类定义分析

(1) **__init__()函数：**是 module 类的初始化函数，会在类的对象生成时被调用以初始化一系列重要的成员变量。比如在 forward、backward、权重加载时会被用到的 hooks 需要提前初始化加入全局的 OrderedDict，从而使所有继承自 nn.Module 的子类实例在运行时都会触发这些 hook；又或者是上文提及的迭代器 Parameter 等，会在之后的训练过程中被使用到，因此子类的__init__()函数中都需要 super().__init__()来进行子类的初始化。

(2) **forward()函数：**是该模型/运算层前向传播的函数，需要继承 nn.module 的子类进行重载。

(3) **state_dict()和 load_state_dict()函数：**能够返回、加载一个字典，从而用于读出、写入模型的所有状态，通常用于加载模型参数。

(4) **_apply()和 apply()函数：**_apply()函数可以将 module 类传进给定的函数 func，将这个模块和它的所有子模块的参数通过 func 函数进行处理。_apply()首先对 self.children()进行递归调用，随后对 self._parameters 和 self._buffers 逐个通过 func 来进行处理。通常可以用于模型参数初始化等需要对模型中所有参数进行批量预处理的场合。apply()与_apply()功能类似，只不过_apply()只是针对 module 的成员 parameters 和 buffers 进行处理，类似私有函数，而 apply()可以自定义对象，可以认为是 module 类留给外部进行数据处理的公有接口。

2、nn.module 的子类

module 类是 pytorch 体系下所有神经网络模块的基类，有许多子类由其继承而来，例如下图中所显示的那样，在 nn.module 中，官方为 module 类构造了 Linear、_ConvNd、Pooling、Padding、Normalization 等子类，这些子类对应神经网络中常用的卷积层、线性层、池化层、批归一化层、激活层等。而根据不同的输入数据场景，我们又能根据这些子类构造新的子类，比如以_ConvNd 作为基类，有 Conv1d、Conv2d、Conv3d、ConvTransposeNd 等。

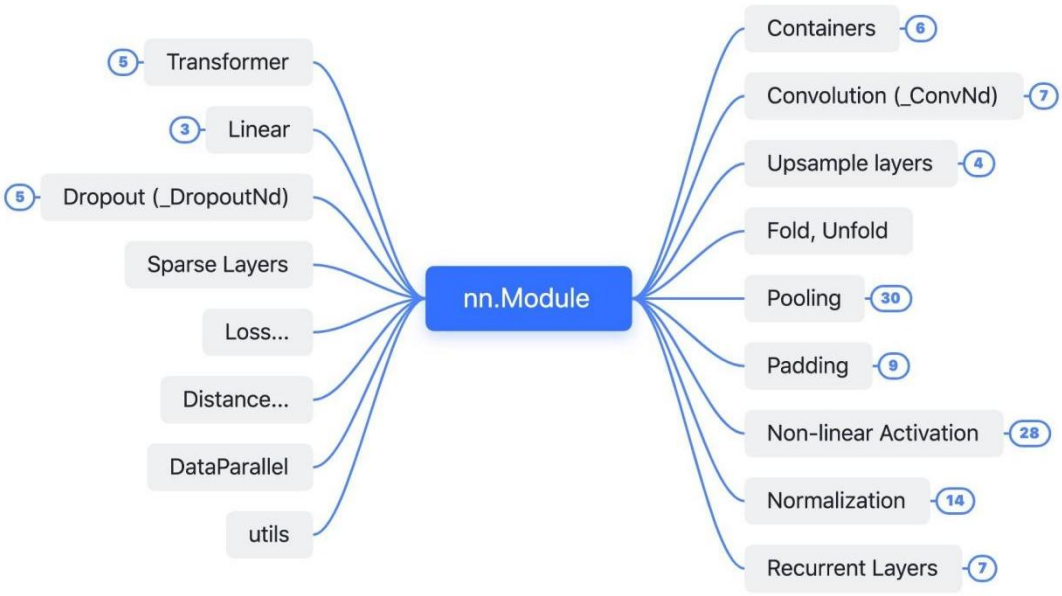

图 2-nn.Module 的子类继承关系示意图

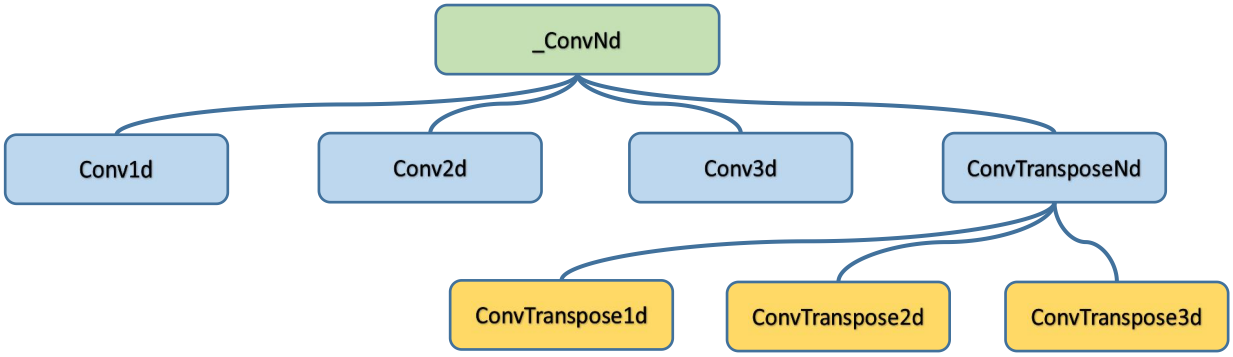

图 3-自_ConvNd 继承的子类关系示意图

五、面向对象程序设计思想的分析

1、抽象和模块化、层次化的设计思想

pytorch 针对神经网络计算的特点，提取并抽象出了算子这一最小单元，并将其封装在 torch.nn.functional 类中。而对于某些常用、固定的算子组成的序列，可以将这些算子封装为一个运算层以使用，并用 torch.nn.module 类来进行封装。模块化的设计思想使得 pytorch 的使用者在设计神经网络时无需关注全局，可以将精力集中到当前的部分；而层次化的设计思想能够让使用者只考虑同一层次的对象之间的相关关系，而不用过多关注层间关系。这样能够有效降低设计难度，从而提高效率。

2、单一职责和接口隔离的设计思想

以 nn.Module 类的成员函数_apply()和 apply()为例，这两个函数都能够对 module 类内的数据进行操作，比如将模型中的所有变量都置成 0 或是置成随机值。这两个函数唯一的不同在于_apply()是类的私有成员函数，仅供类内部使用（下划线作前缀即表示其为私有成员）。在此基础上，nn.Module 类还有以下接口函数 float()、double()、half()等，用于转换模型内参数的类型。注意到这些成员函数之间是基于单一职责的设计原则，彼此独立而分隔开，将功能之间进行解耦，从而能够降低类的复杂度，并且最重要的是降低之后的版本中，代码变更可能引起的风险。同时这样做也蕴含着接口隔离的设计思想，能够使类具有很好的可读性、可扩展性和可维护性，用多个单一的、精简的、细化的接口来实现功能。

3、里氏替换的设计思想

pytorch 非常具有特色的一点就是对 nn.Module 类的继承。由于在构建模型时，大部分工作都会继承自父类 nn.Module 类，使用者只需要完成对__init__()和 forward()函数的重写即可，因此使用者能够方便快捷地完成模型的设计与构建，从而大大减少设计中的机械性劳动，将时间精力投入到更重要的部分中去，也更有利于短时间、轻便式的敏捷开发。

4、定义和实现分离的设计思想

下面是一个简单的神经网络用 pytorch 实现的代码，其中网络结构通过类的成员 self.classifier 定义，并在 forward()函数中进行前向传播的计算。这种定义和实现分离的设计大大增强了代码的可读性和可扩展性，例如 pytorch 版本更新后，我们可能只需要修改很少的代码就能够解决更新带来的兼容性问题，但对于其他的神经网络编程框架例如 Tensorflow，可能就不具有这样的优势。

```
self.classifier = nn.Sequential(
    BinarizeLinear(256 * 2 * 2, 4096),
    nn.BatchNorm1d(4096),
    nn.Hardtanh(inplace=True),

    BinarizeLinear(4096, 4096),
    nn.BatchNorm1d(4096),
    nn.Hardtanh(inplace=True),

    BinarizeLinear(4096, 10),
    nn.BatchNorm1d(10),
    nn.LogSoftmax()
)
```

```
def forward(self, input):

    self.output = self.classfier(input)
```

图 4-定义与实现分离的神经网络代码图