源码阅读报告（三）：高级设计意图分析

经过上面两篇的报告，我们在这篇报告中将更加多专注于optimizer部分的设计。在课堂的pre中我没有具体引用工厂模式在pytorch中的具体应用，因为我觉得不太容易讲解清楚，幸好还有一次报告的机会，就把这个分析作为这次报告的开场。

在optimizer中的工厂模式

工厂模式的简单回顾

> *简单来说工厂模式就是造“工厂”，用你自己的“工厂”来生产新的类，供coder使用。可以分为：简单工厂，工厂模式，抽象工厂三种。*

工厂模式的应用

在 PyTorch 中，优化器的实现**没有严格遵循传统的工厂模式**，但其调用方式体现了一种工厂模式的思想：通过统一的接口创建各种优化器实例，而隐藏具体实现细节。用户通过调用 torch.optim 模块中的优化器函数（如 SGD、Adam）即可创建对应的优化器，而不需要直接关注其实现细节。
源代码：`torch/optim/__init__.py`。这里为了方便讲述对源代码做了删减。

```
from .sgd import SGD
from .adam import Adam
from .rmsprop import RMSprop
from .adamw import AdamW
# 其他优化器的导入...

__all__ = ['Optimizer', 'SGD', 'Adam', 'RMSprop', 'AdamW', ...]

# 注册优化器
```

用户通过直接调用 `torch.optim.Adam` 或 `torch.optim.SGD` 来创建优化器实例。这样我们就直接得到了工厂的“产品”

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

那么工厂究竟在哪里？？
源代码：`torch/optim/optimizer.py`

```
class Optimizer:
```

```
    r"""Base class for all optimizers.

    .. warning::
        Parameters need to be specified as collections that have a
deterministic
        ordering that is consistent between runs. Examples of objects
that don't
        satisfy those properties are sets and iterators over values of
dictionaries.

    Args:
        params (iterable): an iterable of :class:`torch.Tensor` s or
            :class:`dict` s. Specifies what Tensors should be optimized.
        defaults: (dict): a dict containing default values of
optimization
            options (used when a parameter group doesn't specify them).
    """


    def __init__(self, params: ParamsT, defaults: Dict[str, Any]) ->
None:  # noqa: D107
      #具体代码省略
```

在这里我想直接引用Base class for all optimizers这句话。这句话点出了'optimizer'这个类是抽象工厂类，其他的类型想要实现其他的功能只需要继承这个类，然后分别实现别的功能。
源代码：torch/optim/adam.py

```
from .optimizer import Optimizer

class Adam(Optimizer):
    def __init__(self, params, lr=1e-3, betas=(0.9, 0.999), eps=1e-8,
weight_decay=0, amsgrad=False):
        defaults = dict(lr=lr, betas=betas, eps=eps,
weight_decay=weight_decay, amsgrad=amsgrad)
        super(Adam, self).__init__(params, defaults)

    def step(self, closure=None):
        # 具体优化逻辑
        pass
```

可以看到这里采用了抽象工厂模式，定义了抽象工厂之后，在具体工厂中实现了Adam优化器的具体功能。
因为面对多种优化器的需求，pytorch设计者没有选择一股脑地全部实现，因为需求是无穷无尽的，面对层出不穷的优化需求，各种各样的优化方法，不可能也没有必要全部实现。只实现

一个基础的抽象工厂类，将最基本的基础功能实现，再在具体的工厂类中的实现这些具体的功能，比如惯性的引进，学习率的自适应等等功能。

optimizer的设计鉴赏

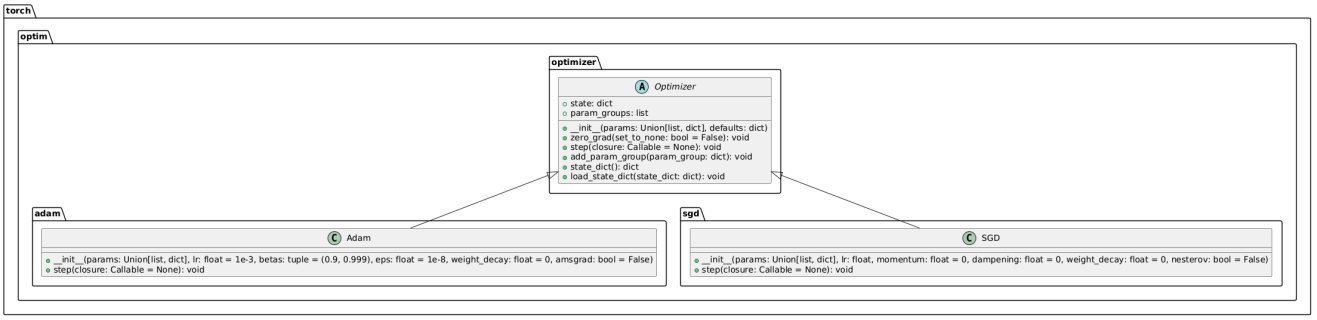
optimizer是干什么的?

> *它是用于更新模型参数的工具，其目的是通过迭代的方式最小化损失函数，从而训练模型使其在给定任务上表现更好。简单来说就是，一盏指路明灯，为人类训练模型指路*

可以看到这关系到核心需求**训练**。面对多种多样的需求，optimizer的设计就必须兼顾考虑这些需求，同时还得易于使用，让人们只需要简单的设置就可以快速开始训练，不必在实现optimizer上花费时间。

optimizer的类图

这个类图只选用了最常用的两个优化器来展示和optimizer类的关系，但是实际上，优化器有很多，感兴趣的话，可以参考这个连接



上面类图简单的说明如下:
Optimizer:

- 定义为抽象基类，包含优化器的通用接口。
- 包括属性 state 和 param_groups 及方法如

Adam 和 SGD:

- 分别继承自 Optimizer。
- 重写了 step() 方法，实现各自的参数更新逻辑。

依赖关系:

- Optimizer 是 Adam 和 SGD 的父类

optimizer中的开闭原则

我在关于工厂模式的pre中曾提到了optimizer的设计中**开闭原则**，因为优化器显然是一个需要进行大量修改的模块，来实现各种各样的功能。

> *开闭原则的核心含义：*

- 对扩展开放：当需求发生变化或需要增加新功能时，软件可以通过添加新代码来实现，而不需要修改现有代码。
- 对修改封闭：现有的代码应保持不变，尽量减少修改原有代码的风险和成本。

优化器基类（Optimizer）： PyTorch 提供了一个基类 Optimizer，它定义了所有优化器共有的行为和方法，如 step() 和 zero_grad()。这些方法提供了优化器的通用操作流程，但具体的梯度更新逻辑留给了子类实现
具体优化器（SGD, Adam）： 具体的优化算法如 SGD 和 Adam 继承自 Optimizer，并实现了自己特定的 step() 方法。在 Adam 中，step() 方法会根据具体的优化策略来更新参数，而 SGD 则执行简单的随机梯度下降。

总结来说，在 PyTorch 中，优化器的设计实现了开闭原则，因为它允许开发者通过扩展 Optimizer 基类来添加新的优化器类型，而不需要修改现有的 Optimizer 类或具体的优化器实现。

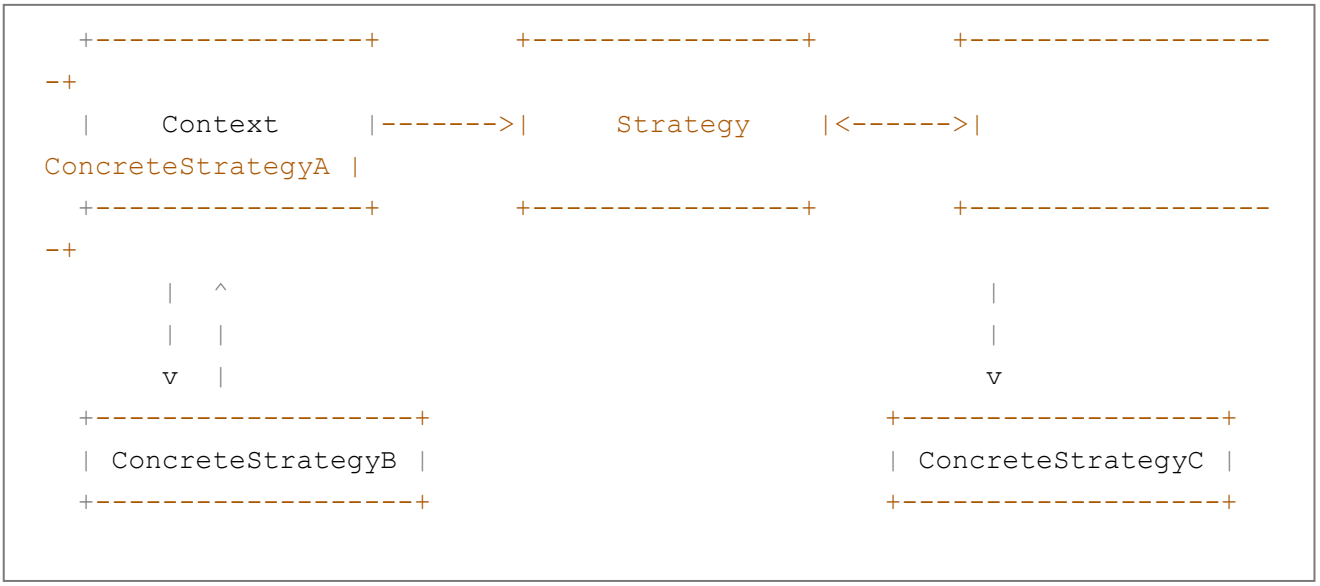
optimizer中的策略模式

在策略模式中一个类的行为或其算法可以在运行时更改。这种类型的设计模式属于行为型模式。

在策略模式定义了一系列算法或策略，并将每个算法封装在独立的类中，使得它们可以互相替换。通过使用策略模式，可以在运行时根据需要选择不同的算法，而不需要修改客户端代码。

在策略模式中，我们创建表示各种策略的对象和一个行为随着策略对象改变而改变的 context 对象。策略对象改变 context 对象的执行算法。

策略模式的类图：



策略接口：
Optimizer 类提供了优化器的通用接口step()、zero_grad()，这些方法在不同的具体优化器类中被实现。所有具体优化器（如 SGD、Adam、RMSprop 等）都遵循这个接口，但实现细节根据不同的优化算法而有所不同。
具体策略类：
pyTorch 中的每个优化器实现了 Optimizer 类，提供了自己的优化步骤方法。每个优化器代表一种不同的优化策略，这些策略类封装了特定的算法。
上下文：
PyTorch 中的上下文相当于训练过程中使用的具体优化器。每次训练时，用户会选择一个优化器（如 SGD、Adam），然后在训练过程中动态地调用该优化器的策略来更新参数。

在训练过程中，用户通过选择不同的优化器来选择不同的策略,optimizer.step() 会根据选择的优化器策略执行相应的优化步骤。如果选择的是 SGD，则执行 SGD.step() 方法；如果选择的是 Adam，则执行 Adam.step() 方法。这就是策略模式的核心：不同的策略类提供了不同的行为，训练过程可以动态选择并执行相应的策略

写道这里基本上就分析完了优化器的设计意图，但是我觉得最好的理解方式便是实际使用一下。所以下面我将进行一个简单的机器学习模型的训练。

简单代码示例

```
import torch
import torch.nn as nn
import torch.optim as optim

class SimpleNet(nn.Module):
    def __init__(self):
        super(SimpleNet, self).__init__()
        self.fc1 = nn.Linear(2, 2)
        self.fc2 = nn.Linear(2, 1)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x

model = SimpleNet()
criterion = nn.MSELoss()

optimizer = optim.SGD([
    {'params': model.fc1.parameters(), 'lr': 0.01},
    {'params': model.fc2.parameters(), 'lr': 0.001}
], momentum=0.9)

inputs = torch.tensor([[1.0, 2.0], [2.0, 3.0]], requires_grad=False)
targets = torch.tensor([[1.0], [2.0]], requires_grad=False)
```

```python
for epoch in range(10):
    optimizer.zero_grad()
    outputs = model(inputs)
    loss = criterion(outputs, targets)
    loss.backward()
    optimizer.step()
    print(f"Epoch [{epoch+1}/10], Loss: {loss.item():.4f}")
```

输出的结果如下：

```
PS C:\Users\lenovo\Desktop\新建文件夹\学科资料\面向对象的程序设计> & C:/Users/lenovo/AppData/Local/
/Users/lenovo/Desktop/新建文件夹/学科资料/面向对象的程序设计/code_reading_reports/code_reading_repo
Epoch [1/10], Loss: 1.3663
Epoch [2/10], Loss: 1.3618
Epoch [3/10], Loss: 1.3534
Epoch [4/10], Loss: 1.3414
Epoch [5/10], Loss: 1.3264
Epoch [6/10], Loss: 1.3086
Epoch [7/10], Loss: 1.2886
Epoch [8/10], Loss: 1.2666
Epoch [9/10], Loss: 1.2430
Epoch [10/10], Loss: 1.2181
PS C:\Users\lenovo\Desktop\新建文件夹\学科资料\面向对象的程序设计>
```

后记

至此，第三个报告完成！这一路看过来源码实属不易。一开始像一个无头苍蝇一样在这么多的代码中不知所措，到渐渐熟悉代码的套路，逐渐适应了代码在讲些什么。最后完成了这三篇报告。学习了优化的几种主要算法，我对于pytorch的使用也渐渐熟悉适应。无论怎么说，报告已经完结，但学习并未结束，还是需要继续走下去。