源码阅读报告（二）：核心流程设计分析

上一次的报告我们将pytorch的主要功能和结构进行了简单的分析，知晓其设计API的简要架构情况。但是这没有解释我心中的一个疑问为什么pytorch能够脱颖而出，我**询问**了chatGPT,现在将**ChatGPT回答**摘录如下：

1. **动态图计算**
   - 支持动态计算图，提供灵活性，特别适合循环神经网络（RNN）和生成对抗网络（GAN）等研究和开发工作。
2. **强大的自动微分机制**
   - Autograd 模块支持自动微分，简化了反向传播计算，适合新手和研究人员。
3. **社区支持和开放源代码**
   - 由 Facebook（Meta）开发并开源，社区活跃，持续更新和改进，促进生态系统发展。
4. **强大的生态系统**
   - 包含 PyTorch Lightning、TorchServe、PyG 和 TorchX 等工具，生态系统丰富，适合 NLP 和 CV 领域应用。
5. **与研究界的紧密结合**
   - 深受学术界支持，大量顶会论文基于 PyTorch 实现，保持前沿地位。
6. **硬件加速与跨平台支持**
   - 优化 GPU、TPU 加速，兼容多种硬件和平台，应用场景广泛。
7. **Pythonic 设计**
   - 符合 Python 语言习惯，API 简洁直观，使得开发者易于上手。

ChatGPT给出了两个很重要的方向 动态图计算 和 强大的自动微分机制 ，现在我将在本次报告中对其一探究竟。

动态图计算

对于动态图，我有三个问题：

- 什么是动态图?
- 为什么要用动态图?
- 怎么用的动态图?

解答好了这三个问题，才能理解pytorch中的动态图计算。

什么是动态图与为什么要用动态图

先看这样一段代码：

```
import tensorflow as tf  # version 1.15.0

a = tf.ones(5) #定义了两个张量 a 和 b
               #每个张量包含 5 个元素，且所有元素都为 1。
b = tf.ones(5)
c = a + b

sess = tf.Session() #创建一个 TensorFlow 会话 sess，用于管理和运行计算图。
print(sess.run(c))  # 输出 [2. 2. 2. 2. 2.]
```

这段代码通过先定义计算图，再创建会话运行图的方式来进行计算，这就是典型的“静态图”执行模式。

```
import torch

# 定义两个张量
a = torch.ones(5)       # 创建一个包含 5 个 1 的张量
b = torch.ones(5)       # 创建另一个包含 5 个 1 的张量

# 执行加法操作
c = a + b               # 立即执行加法并得到结果

print(c)                # 输出结果：[2. 2. 2. 2. 2.]
```

细心的读者会发现少了些什么。很显然就是**不用创建会话**。在 PyTorch 的动态图机制下，每次执行操作时，计算图都是动态构建的，不需要像 TensorFlow 1.x 那样创建会话。这种即时执行的方式非常适合调试和动态结构的模型构建。在PyTorch中，我们可以在计算的任意步骤直接输出结果。PyTorch每一条语句是同步执行的，每一条语句都是一个（或多个）算子，被调用时实时执行。这大致就是动态图的工作方式。

其实在pytorch的官方网页中给出了回答:

```
Deep learning compilers commonly only work for static shapes, that is to say, they produced comp

Some dimensions, such as batch size or sequence length, may vary. For example, an inference serv

Some models exhibit data-dependent output shapes, that is to say, the size of their outputs and

One particularly important case of data-dependent shapes occurs when dealing with sparse represe
```

这段回答其实总结下来就是一个词：**实时**，支持动态图将带来更大的灵活性，有助于应对输入维度变化，避免频繁的重新编译。

怎么用的动态图，又是怎么体现了其中的设计思想。

PyTorch每次执行模型时都会构建新的计算图，而计算图的节点对应的是不同的操作（如加法、矩阵乘法等）。
Pytprch具体计算：每一层的前向和反向传播，其实都是在 ATen 里实现的，但回去看 ATen 的API实现，并没有发现其中有任何建立计算图的代码。我们查看tools/autograd这个目录。这个目录里有derivatives.yaml和用于生成代码的脚本，前者记录了所有需要自动微分的ATen API，后者为它们生成一层wrapper代码，这些代码主要干两件事：

把ATen的反向传播API转换成Function
在ATen的正向传播API中加入建图过程

这是这些的代码截图：

```
- name: abs(Tensor self) -> Tensor
  self: grad * self.sgn()
  result: handle_r_to_c(result.scalar_type(), self_t.conj() * self_p.sgn())

- name: acos(Tensor self) -> Tensor
  self: grad * -((-self * self + 1).rsqrt()).conj()
  result: auto_element_wise

- name: add.Tensor(Tensor self, Tensor other, *, Scalar alpha=1) -> Tensor
  self: handle_r_to_c(self.scalar_type(), grad)
  other: handle_r_to_c(other.scalar_type(), maybe_multiply(grad, alpha.conj()))
  result: self_t + maybe_multiply(other_t, alpha)

- name: add.Scalar(Tensor self, Scalar other, Scalar alpha=1) -> Tensor
  self: handle_r_to_c(self.scalar_type(), grad)
  result: self_t.clone()

- name: addbmm(Tensor self, Tensor batch1, Tensor batch2, *, Scalar beta=1, Scalar alpha=1) -> Tensor
  self: maybe_multiply(grad, beta.conj())
  batch1: maybe_multiply(grad.unsqueeze(0).expand_symint({ batch1.sym_size(0), batch1.sym_size(1), batch2.sym_size(2) }).bmm(batch2.transpose(1, 2).conj(
  batch2: maybe_multiply(batch1.transpose(1, 2).conj().bmm(grad.unsqueeze(0).expand_symint({ batch1.sym_size(0), batch1.sym_size(1), batch2.sym_size(2) }
  result: maybe_multiply(self_t, beta) + maybe_multiply(batch1_t.bmm(batch2_p).sum(0), alpha) + maybe_multiply(batch1_p.bmm(batch2_t).sum(0), alpha)

- name: addcdiv(Tensor self, Tensor tensor1, Tensor tensor2, *, Scalar value=1) -> Tensor
  self: handle_r_to_c(self.scalar_type(), grad)
  tensor1: handle_r_to_c(tensor1.scalar_type(), grad * (value / tensor2).conj())
  tensor2: handle_r_to_c(tensor2.scalar_type(), -grad * (value * tensor1 / (tensor2 * tensor2)).conj())
  result: self_t + maybe_multiply(tensor1_t / tensor2_p, value) - maybe_multiply(tensor2_t * (tensor1_p / tensor2_p) / tensor2_p, value)
```

这些API也体现我们设计中的**高内聚，低耦合**。

自动求导机制

什么是自动求导机制

自动求导（Automatic Differentiation，简称Autograd） 是一种计算梯度的技术，在深度学习中非常重要，它能够自动地计算复杂函数的导数，尤其是在神经网络的训练过程中用于梯度更新。自动求导的核心目的是通过计算图（computation graph）自动获取函数在某一点的梯度值，从而可以进行反向传播（backpropagation）来优化模型参数。

自动求导的基本原理

自动求导依赖于计算图的构建，通过跟踪前向传播中每个操作来生成计算图，然后通过反向传播来计算每个操作的梯度。计算图是一个有向图，其中每个节点表示一个操作或变量，边表示操作之间的依赖关系：

- 前向传播：根据输入数据，逐步执行操作并生成计算图。每执行一个操作，就在图中添加一个节点，表示此操作。
- 反向传播：从最终的输出开始，按计算图的反向顺序逐层计算梯度。梯度通过链式法则（chain rule）传递，从而得到每个输入对最终输出的导数。

PyTorch中的自动求导

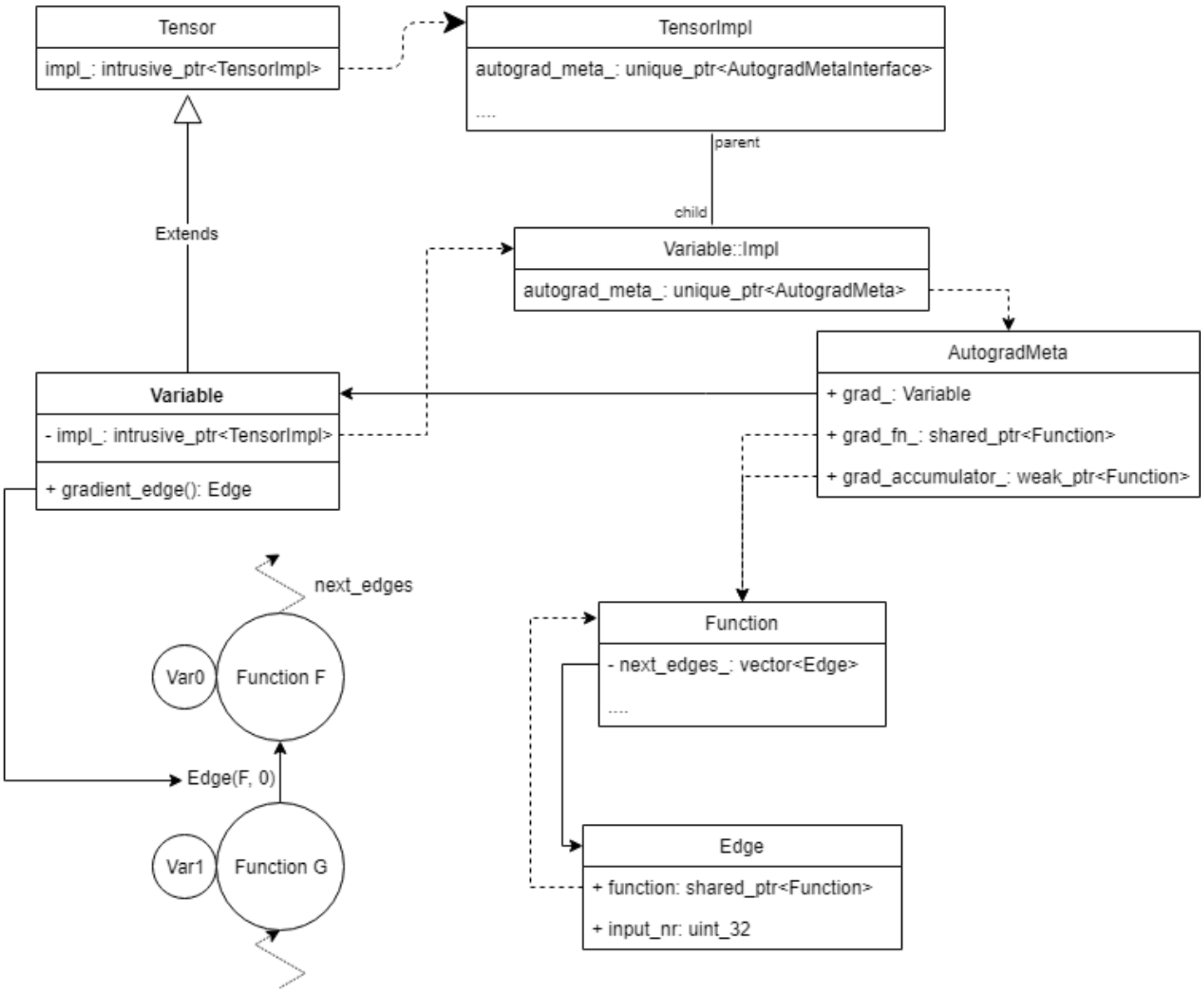
对于深度学习来说，找到最佳的那一组参数，来确保设计的模型能够有一个较小的损失是最终目标，这一切都依赖于对于由这些参数组成的损失函数的求梯度，即自动求导。这是为人们在那些参数组成的高维空间里面指引方向的罗盘，一步一步，就能找到损失的最小值。
在PyTorch中，自动求导是通过 autograd 模块实现的。通过设置 requires_grad=True，PyTorch会自动为你追踪该张量的操作，构建计算图，并在反向传播时计算梯度

```python
import torch

# 创建一个张量，并要求计算梯度
x = torch.randn(2, 2, requires_grad=True)

# 进行一些计算
y = x * 2
z = y.mean()

# 执行反向传播，计算梯度
z.backward()

# 查看x的梯度
print(x.grad)
```

运行结果：

```
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/l
文件夹/学科资料/面向对象的程序设计/code_reading_reports/code_reading_reports/te
tensor([[0.5000, 0.5000],
        [0.5000, 0.5000]])
```

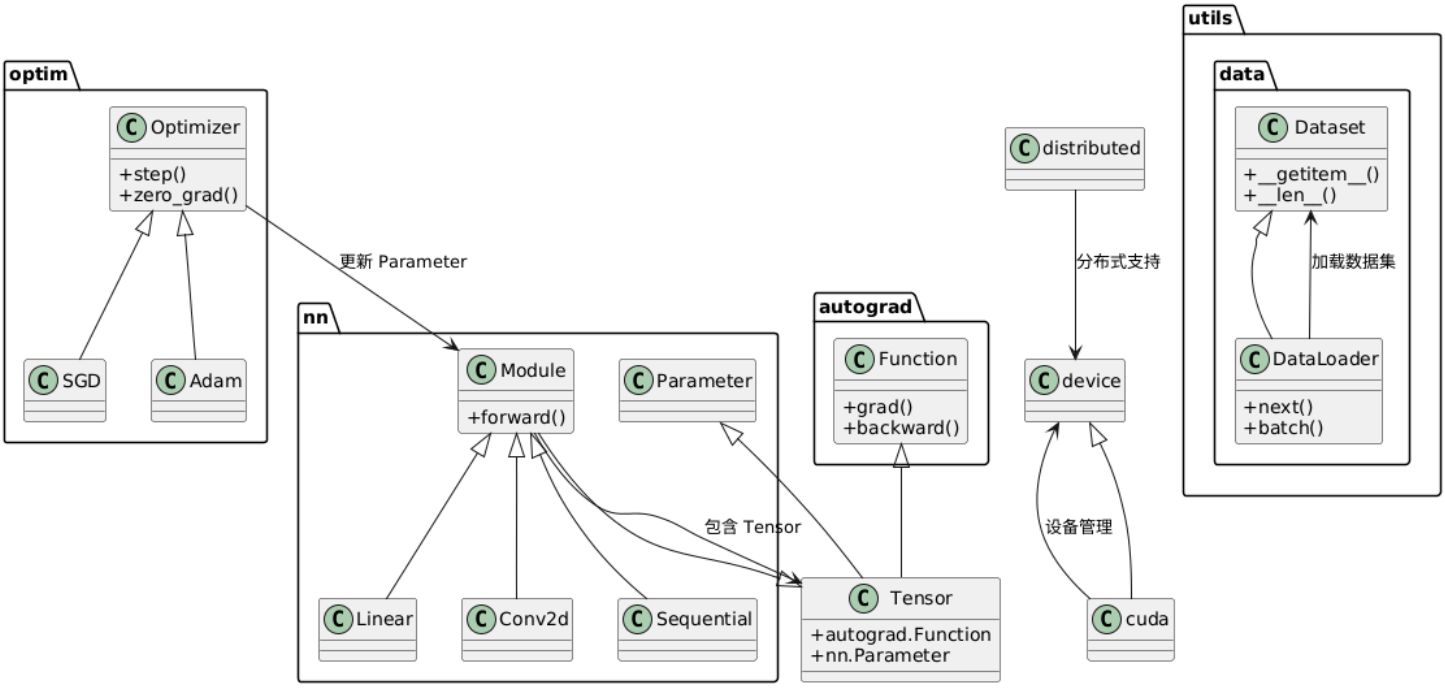
使用这个自动求导很简单，但是这是如何实现的呢?
首先要看几个相关的数据结构，分别是Variable, AutogradMeta, Function和Edge。
这是它们的大致关系：



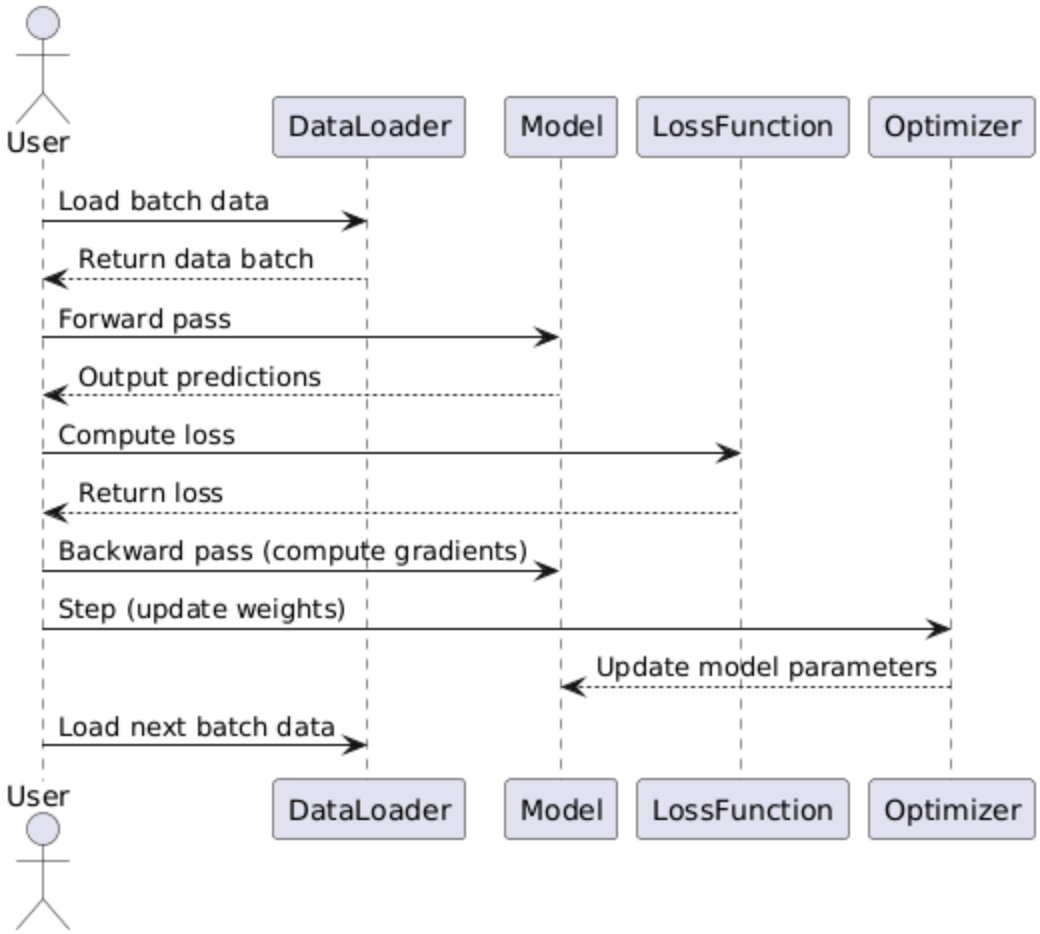
Variable可以表示计算图中的叶子节点，如权重，也可以表示图中的中间变量，虽然在新版本中Tensor与Variable合并了，在前端中可以直接用Tensor代替Variable。
AutogradMeta不但存储了梯度，还存储了该Variable对应的反向传播函数，也就是计算图的节点。
Function本质是一个函数对象，可以当作反向传播的函数来用。
其实前向传播就是生成图；反向传播就是对于图的遍历，计算梯度。这样就简单地理解所谓的自动求导。
这样就简单得理解了pytorch中的自动求导

类间关系图与时序关系图

pytorch的类间关系如下图所示



主要的分为四类功能，在下列的时序图中展现，分别是：dataloader,model,LossFunction,Optimizer



结语

在学习pytorch的源码之后，我感受到深度学习技术的本身并不是一个新技术（本质上是之前的机器学习的换皮）。pytorch为深度学习开发的组件让人们可以很容易地进行训练，比如Optimizer,这里面集成了学界的先进技术，不仅有自适应学习率还加入了惯性；当人们使用它时只需要简单的调用就可以，这极大地方便了对于深度神经网络的训练，让人们只需要专注于设计神经网络就可以。