Caffe代码阅读——层次结构

本文收录在无痛的机器学习第一季。

Caffe是一款优秀的深度神经网络的开源软件，下面我们来聊聊它的源代码以及它的实现。Caffe的代码整体上可读性很好，架构比较清晰，阅读代码并不算是一件很困难的事情。不过在阅读代码之前还是要回答两个问题：

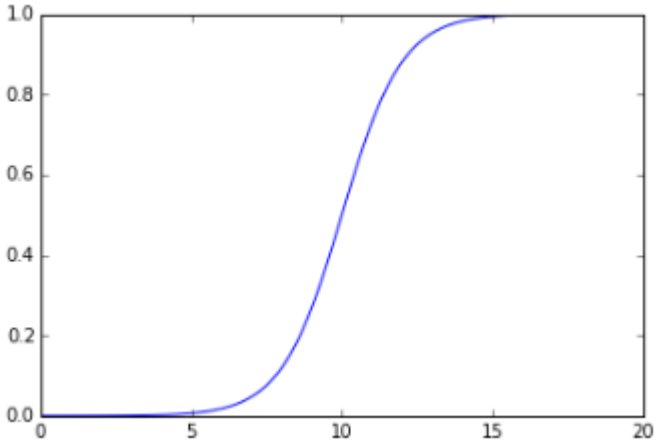
1. 阅读代码是为了什么？
2. 阅读到什么程度？（这个问题实际上和前面的问题相关）

阅读代码大体上来说有下面几个目的：

1. 搞清楚代码所实现的算法或者功能。对算法本身不是很了解，希望通过阅读代码了解算法。
2. 搞清楚代码在实现算法过程中的细节。这种情况下，一般对算法已经有大概的了解，读代码是为了了解代码中对算法细节的考量。当然，如果想使用代码，了解代码细节是很有帮助的。
3. 扩展代码。在开源代码的基础上，利用已有的框架，增加或者修改功能，来实现自己想要的功能。这个就需要对代码的架构细节有更加深入的了解。

我们的目标是扩展代码。Caffe中主要的扩展点就是Layer和Solver，当然其他的部分也可以扩展，只不过要改动的代码会多一些。

当确定了上面第一个问题，下面就是第二个问题了。读代码要读到什么程度？一般来说，我觉得阅读代码这件事情可以用一个Logistic型的函数来表示：



这个图上，横轴是阅读代码花费的时间，纵轴是阅读代码带来的效果。对于代码量比较大的项目，一开始阅读肯定是蒙的，需要花一定的时间梳理清楚各个文件，各个模块之间的关系。随着结构关系逐渐清晰，读者开始领会代码中所表达的含义，阅读代码的效果直线上升。然而当我们把代码主线和重要支线弄懂后，再读一些小支线的收益就不会太大。所以根据阅读代码的性价比和Caffe代码自身的特点，我们只会将主线和一些重要支线阅读完，估计也就是整体代码量的一半。

Caffe代码的主线结构抽象

不同于其他的一些框架，Caffe没有采用符号计算的模式进行编写，整体上的架构以系统级的抽象为主。所谓的抽象，就是逐层地封装一些细节问题，让上层的代码变得更加清晰。那么就让我们来顺着Caffe的抽象层级看看Caffe的主线结构：

SyncedMem：这个类的主要功能是封装CPU和GPU的数据交互操作。一般来说，数据的流动形式都是：硬盘→CPU内存→GPU内存→CPU内存→（硬盘），所以在写代码的过程中经常会写CPU/GPU之间数据传输的代码，同时还要维护CPU和GPU两个处理端的内存指针。这些事情处理起来不会很难，但是会很繁琐。因此SyncedMem的出现就是把CPU/GPU的数据传输操作封装起来，只需要调用简单的接口就可以获得两个处理端同步后的数据。

Blob：这个类做了两个封装：一个是操作数据的封装。在这里使用Blob，我们可以操纵高维的数据，可以快速访问其中的数据，变换数据的维度等等；另一个是对原始数据和更新量的封装。每一个Blob中都有

data和diff两个数据指针，data用于存储原始数据，diff用于存储反。向传播的梯度更新值。Blob使用了SyncedMem，这样也得到了不同处理端访问的便利。这样Blob就基本实现了整个Caffe数据部分结构的封装，在Net类中可以看到所有的前后向数据和参数都用Blob来表示就足够了。

数据的抽象到这个就可以了，接下来是层级的抽象。前面我们也分析过，神经网络的前后向计算可以做到层与层之间完全独立，那么每个层只要依照一定的接口规则实现，就可以确保整个网络的正确性。

Layer: Caffe实现了一个基础的层级类**Layer**，对于一些特殊种类还会有自己的抽象类（比如base_conv_layer），这些类主要采用了模板的设计模式（Template）,也就是说一些必须的代码在基类写好，一些具体的内容在子类中实现。比方说在Layer的Setup中，函数中包括Setup的几个步骤，其中的一些步骤由基类完成，一些步骤由子类完成。还有十分重要的Forward和Backward，基类实现了其中需要的一些逻辑，但是真正的运算部分则交给了子类。这样当我们需要实现一个新的层时，我们不需要管理琐碎的事物，只要关系好层的初始化和前后向即可。

Net: Net将数据和层组合起来做进一步的封装，对外暴露了初始化和前后向的接口，使得整体看上去和一个层的功能类似，但内部的组合可以是多种多样。同时值得一提的是，每一层的输入输出数据统一保存在Net中，同时每个层内的参数指针也保存在Net中，不同的层可以通过WeightShare共享相同的参数，所以我们可以通过配置实现多个神经网络层之间共享参数的功能，这也增强了我们对网络结构的想象力。

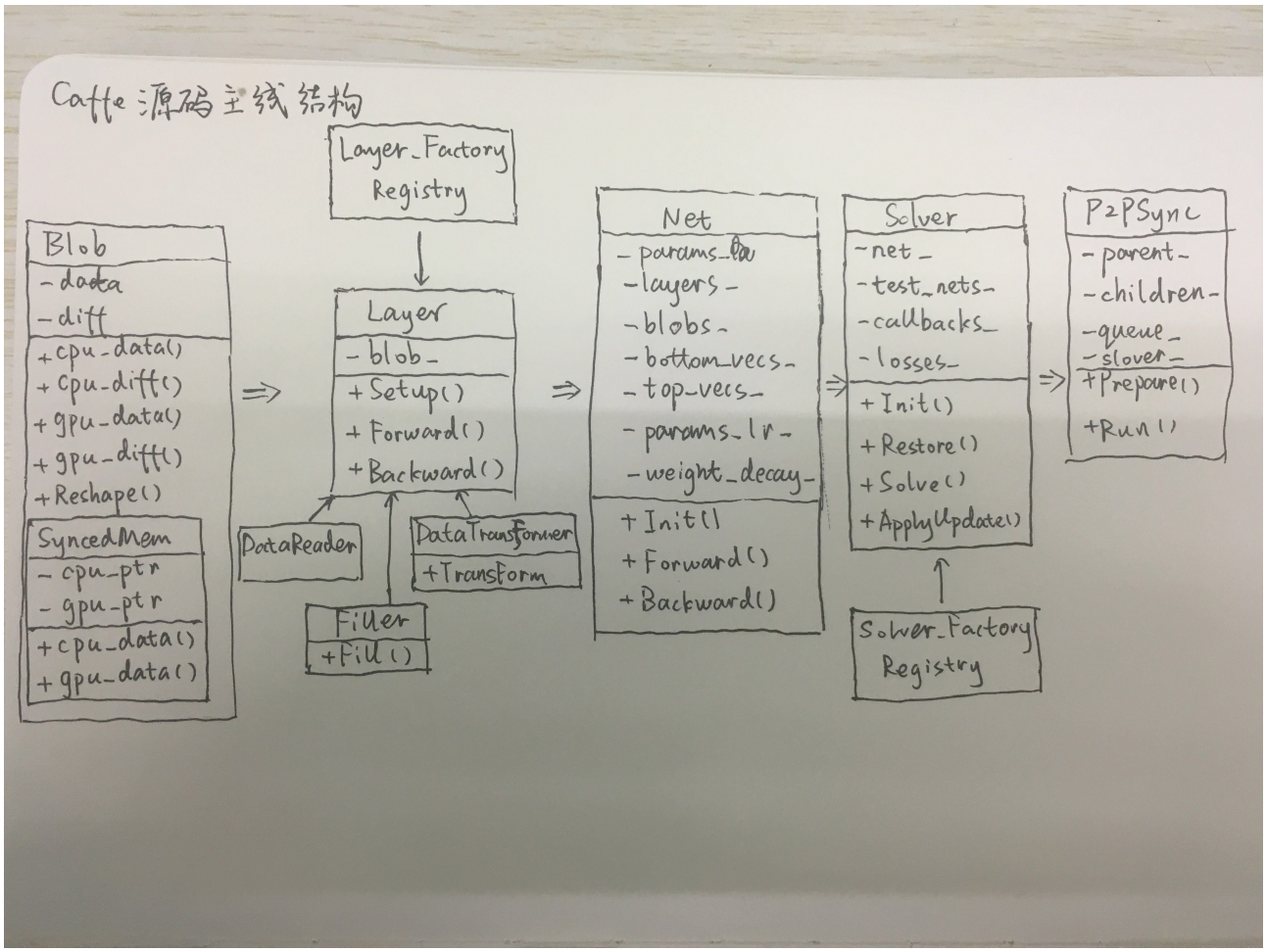
Solver：有了Net我们实际上就可以进行网络的前向后向计算了，但是关于网络的学习训练的功能还有些缺乏，于是在此之上，**Solver**类进一步封装了训练和预测相关的一些功能。与此同时，它还开放了两类接口：一个是更新参数的接口，继承Solver可以实现不同的参数更新方法，如大家喜闻乐见的Momentum，Nesterov，Adagrad等。这样使得不同的优化算法能够应用其中。另外一个是训练过程中每一轮特定状态下的可注入的一些回调函数，在代码中这个回调点的直接使用者就是多卡训练算法。

IO：有了上面的东西就够了？还不够，我们还需要输入数据和参数，正所谓巧妇难为无米之炊，没有数据都是白搭。**DataReader**和

DataTransformer帮助准备输入数据，**Filler**对参数进行初始化。一些**Snapshot**方法帮助模型的持久化，这样模型和数据的IO问题也解决了。

多卡：对于单GPU训练来说，基本的层次关系到这里也就结束了，如果要进行多GPU训练，那么上层还会有**InternalThread**和**P2PSync**两个类，这两个类属于最上层的类了，而他们所调用的也只有Solver和一些参数类。

其实到这里，Caffe的主线也就基本走完了。我们可以画一张图把Caffe的整体层次关系展示出来：



如果对这张图和图中的一些细节比较清楚的话，那么你对Caffe的了解应该已经不错了。后面关于Caffe源码分析的文章就可以不看了。如果没有，那么我们还是可以继续关注一下。当然如果想真正理解这张图中所表达的含义，还是要真正地读一下代码，去理解一些细节。但是有些细节这里就不做详细的分析了，下一回我们会站在Layer的角度去看一个Layer在训练过程的全部经历。