**[Caffe学习笔记1-安装以及代码结构](https://yufeigan.github.io/2014/12/09/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B01-%E5%AE%89%E8%A3%85%E4%BB%A5%E5%8F%8A%E4%BB%A3%E7%A0%81%E7%BB%93%E6%9E%84/" \o "Caffe学习笔记1-安装以及代码结构)**

By [YuFeiGan](http://yoursite.com/)

 2014-12-09 更新日期:2014-12-09

## 安装

按照[官网教程](http://caffe.berkeleyvision.org/installation.html)安装，我在 OS X 10.9 和 Ubuntu 14.04 上面都安装成功了。主要麻烦在于 glog gflags gtest 这几个依赖项是google上面的需要翻墙。由于我用Mac没有CUDA，所以安装时需要设置 CPU\_ONLY := 1。

*如果不是干净的系统，安装还是有点麻烦的比如我在OS X 10.9上面，简直不是一般的麻烦，OS X 10.9 默认的编译器是clang，所以还要修改编译器和重行编译一大堆依赖库。这方面其实网上教程很多，涵盖了各种你可能遇到的问题，多Google下问题还是可以解决的。*

## 目录结构

caffe文件夹下主要文件： 这表示文件夹

* data 用于存放下载的训练数据
* docs 帮助文档
* example 一些代码样例
* matlab MATLAB接口文件
* python Python接口文件
* model 一些配置好的模型参数
* scripts 一些文档和数据用到的脚本

下面是核心代码文件夹：

* tools 保存的源码是用于生成二进制处理程序的，caffe在训练时实际是直接调用这些二进制文件。
* include Caffe的实现代码的头文件
* src 实现Caffe的源文件

**后面的学习主要围绕后面两个文件目录（include和src）下的代码展开**

## 源码结构

由于include和src两个目录在层次上基本一一对应因此主要分析src即可了解文件结构。

*这里顺便提到一个有意思的东西，我是在Sublime上面利用SublimeClang插件分析代码的（顺便推荐下这插件，值得花点时间装）。在配置的时候发现会有错误提示找不到”caffe/proto/caffe.pb.h”，去看了下果然没有，但编译的时候没有报错，说明是生成过后又删除了，查看Makefile文件后发现这里用了proto编译的，所以在”src/caffe/proto”下面用CMakeLists文件就可以编译出来了。*

* src
  + gtest google test一个用于测试的库你make runtest时看见的很多绿色RUN OK就是它，这个与caffe的学习无关，不过是个有用的库
  + caffe 关键的代码都在这里了
    - test 用gtest测试caffe的代码
    - util 数据转换时用的一些代码。caffe速度快，很大程度得益于内存设计上的优化（blob数据结构采用proto）和对卷积的优化（部分与im2col相关）[1]。
    - proto 即所谓的“Protobuf”[2]，全称“Google Protocol Buffer”，是一种数据存储格式，帮助caffe提速。
    - layers 深度神经网络中的基本结构就是一层层互不相同的网络了，这个文件夹下的源文件以及目前位置“src/caffe”中包含的我还没有提到的所有.cpp文件就是caffe的核心目录下的核心代码了。

## 源码主要关系

如上所言我们现在可以知道，caffe核心中的核心是下面的文档和文件:（这部分目前不清楚的地方先参照别人的观点）

* blob[.cpp .h] 基本的数据结构Blob类[3]。
* common[.cpp .h] 定义Caffe类
* internal\_thread[.cpp .h] 使用boost::thread线程库
* net[.cpp .h] 网络结构类Net
* solver[.cpp .h] 优化方法类Solver
* data\_transformer[.cpp .h] 输入数据的基本操作类DataTransformer
* syncedmem[.cpp .h] 分配内存和释放内存类CaffeMallocHost，用于同步GPU，CPU数据
* layer\_factory.cpp layer.h 层类Layer
* layers 此文件夹下面的代码全部至少继承了类Layer

## Caffe的官方说明

根据Caffe[官方文档](http://caffe.berkeleyvision.org/tutorial/net_layer_blob.html)介绍，caffe大致可以分为三层结构blob，layer，net。数据的保存，交换以及操作都是以blob的形式进行的，layer是模型和计算的基础，net整和并连接layer。solver则是模型的优化求解。

[1]: [linger: 我所写的CNN框架 VS caffe](http://blog.csdn.net/lingerlanlan/article/details/38121443)  
[2]: [Google Protocol Buffer 的使用和原理](http://www.ibm.com/developerworks/cn/linux/l-cn-gpb/)  
[3]: [caffe源码简单解析——Blob（1）](http://www.shwley.com/index.php/archives/64/)

# [Caffe学习笔记2-Caffe的三级结构(Blobs,Layers,Nets)](https://yufeigan.github.io/2014/12/09/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B02-Caffe%E7%9A%84%E4%B8%89%E7%BA%A7%E7%BB%93%E6%9E%84-Blobs-Layers-Nets/)

By [YuFeiGan](http://yoursite.com/)

 2014-12-09 更新日期:2014-12-10

根据Caffe[官方文档](http://caffe.berkeleyvision.org/tutorial/net_layer_blob.html)介绍，caffe大致可以分为三层结构blob，layer，net。数据的保存，交换以及操作都是以blob的形式进行的，layer是模型和计算的基础，net整和并连接layer。

## Blobs

Blob是Caffe的基本数据结构，具有CPU和GPU之间同步的能力,它是4维的数组(Num, Channels, Height, Width)。  
设Blob数据维度为 number N x channel K x height H x width W，Blob是row-major保存的，因此在(n, k, h, w)位置的值物理位置为((n \* K + k) \* H + h) \* W + w，其中Number/N是batch size。  
Blob同时保存了data和diff(梯度)，访问data或diff有两种方法:

|  |  |
| --- | --- |
| 1 2 | const Dtype\* cpu\_data() const; //不修改值 Dtype\* mutable\_cpu\_data(); //修改值 |

Blob会使用SyncedMem自动决定什么时候去copy data以提高运行效率，通常情况是仅当gnu或cpu修改后有copy操作，文档里面给了一个例子：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 | ￼// Assuming that data are on the CPU initially, and we have a blob. const Dtype\* foo; Dtype\* bar; foo = blob.gpu\_data(); // data copied cpu->gpu. foo = blob.cpu\_data(); // no data copied since both have up-to-date contents. bar = blob.mutable\_gpu\_data(); // no data copied. // ... some operations ... bar = blob.mutable\_gpu\_data(); // no data copied when we are still on GPU. foo = blob.cpu\_data(); // data copied gpu->cpu, since the gpu side has modified the data foo = blob.gpu\_data(); // no data copied since both have up-to-date contents bar = blob.mutable\_cpu\_data(); // still no data copied. bar = blob.mutable\_gpu\_data(); // data copied cpu->gpu. bar = blob.mutable\_cpu\_data(); // data copied gpu->cpu. |

(顺便查了下一直有疑问[*foo*](http://baike.baidu.com/link?url=coXh-sAgljTqCl-srJQhwHVkq5i24izowBphWUtQUioR8YOVC_b3tf4-ojGZ5VCCbS9ShH4_XE2_31bw5Ne3KK)是什么意思。。)

## Layer

所有的Pooling，Convolve，apply nonlinearities等操作都在这里实现。在Layer中input data用bottom表示output data用top表示。每一层定义了三种操作setup（Layer初始化）, forward（正向传导，根据input计算output）, backward（反向传导计算，根据output计算input的梯度）。forward和backward有GPU和CPU两个版本的实现。

## Net

Net由一系列的Layer组成(无回路有向图DAG)，Layer之间的连接由一个文本文件描述。模型初始化Net::Init()会产生blob和layer并调用Layer::SetUp。在此过程中Net会报告初始化进程。这里的初始化与设备无关，在初始化之后通过Caffe::set\_mode()设置Caffe::mode()来选择运行平台CPU或GPU，结果是相同的。

## 模型的格式

模型定义在.prototxt文件中，训练好的模型在model目录下.binaryproto格式的文件中。模型的格式由caffe.proto定义。采用Google Protocol Buffer可以节省空间还有它对C++和Pyhton的支持也很好。

# [Caffe学习笔记3-Layer的相关学习](https://yufeigan.github.io/2014/12/09/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B03-Layer%E7%9A%84%E7%9B%B8%E5%85%B3%E5%AD%A6%E4%B9%A0/)

By [YuFeiGan](http://yoursite.com/)

 2014-12-09 更新日期:2014-12-10

**文章目錄**

1. [1. Layer](https://yufeigan.github.io/2014/12/09/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B03-Layer%E7%9A%84%E7%9B%B8%E5%85%B3%E5%AD%A6%E4%B9%A0/#Layer)
   1. [1.1. data\_layer](https://yufeigan.github.io/2014/12/09/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B03-Layer%E7%9A%84%E7%9B%B8%E5%85%B3%E5%AD%A6%E4%B9%A0/#data_layer)
      1. [1.1.1. DATA](https://yufeigan.github.io/2014/12/09/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B03-Layer%E7%9A%84%E7%9B%B8%E5%85%B3%E5%AD%A6%E4%B9%A0/#DATA)
      2. [1.1.2. MEMORY\_DATA](https://yufeigan.github.io/2014/12/09/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B03-Layer%E7%9A%84%E7%9B%B8%E5%85%B3%E5%AD%A6%E4%B9%A0/#MEMORY_DATA)
      3. [1.1.3. HDF5\_DATA](https://yufeigan.github.io/2014/12/09/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B03-Layer%E7%9A%84%E7%9B%B8%E5%85%B3%E5%AD%A6%E4%B9%A0/#HDF5_DATA)
      4. [1.1.4. HDF5\_OUTPUT](https://yufeigan.github.io/2014/12/09/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B03-Layer%E7%9A%84%E7%9B%B8%E5%85%B3%E5%AD%A6%E4%B9%A0/#HDF5_OUTPUT)
      5. [1.1.5. IMAGE\_DATA](https://yufeigan.github.io/2014/12/09/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B03-Layer%E7%9A%84%E7%9B%B8%E5%85%B3%E5%AD%A6%E4%B9%A0/#IMAGE_DATA)
   2. [1.2. neuron\_layer](https://yufeigan.github.io/2014/12/09/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B03-Layer%E7%9A%84%E7%9B%B8%E5%85%B3%E5%AD%A6%E4%B9%A0/#neuron_layer)
   3. [1.3. loss\_layer](https://yufeigan.github.io/2014/12/09/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B03-Layer%E7%9A%84%E7%9B%B8%E5%85%B3%E5%AD%A6%E4%B9%A0/#loss_layer)
   4. [1.4. common\_layer](https://yufeigan.github.io/2014/12/09/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B03-Layer%E7%9A%84%E7%9B%B8%E5%85%B3%E5%AD%A6%E4%B9%A0/#common_layer)
      1. [1.4.1. InnerProductLayer](https://yufeigan.github.io/2014/12/09/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B03-Layer%E7%9A%84%E7%9B%B8%E5%85%B3%E5%AD%A6%E4%B9%A0/#InnerProductLayer)
      2. [1.4.2. SplitLayer](https://yufeigan.github.io/2014/12/09/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B03-Layer%E7%9A%84%E7%9B%B8%E5%85%B3%E5%AD%A6%E4%B9%A0/#SplitLayer)
      3. [1.4.3. FlattenLayer](https://yufeigan.github.io/2014/12/09/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B03-Layer%E7%9A%84%E7%9B%B8%E5%85%B3%E5%AD%A6%E4%B9%A0/#FlattenLayer)
      4. [1.4.4. ConcatLayer](https://yufeigan.github.io/2014/12/09/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B03-Layer%E7%9A%84%E7%9B%B8%E5%85%B3%E5%AD%A6%E4%B9%A0/#ConcatLayer)
      5. [1.4.5. SilenceLayer](https://yufeigan.github.io/2014/12/09/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B03-Layer%E7%9A%84%E7%9B%B8%E5%85%B3%E5%AD%A6%E4%B9%A0/#SilenceLayer)
      6. [1.4.6. (Elementwise Operations)](https://yufeigan.github.io/2014/12/09/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B03-Layer%E7%9A%84%E7%9B%B8%E5%85%B3%E5%AD%A6%E4%B9%A0/#(Elementwise_Operations))
   5. [1.5. vision\_layer](https://yufeigan.github.io/2014/12/09/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B03-Layer%E7%9A%84%E7%9B%B8%E5%85%B3%E5%AD%A6%E4%B9%A0/#vision_layer)
      1. [1.5.1. ConvolutionLayer](https://yufeigan.github.io/2014/12/09/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B03-Layer%E7%9A%84%E7%9B%B8%E5%85%B3%E5%AD%A6%E4%B9%A0/#ConvolutionLayer)
      2. [1.5.2. Im2colLayer](https://yufeigan.github.io/2014/12/09/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B03-Layer%E7%9A%84%E7%9B%B8%E5%85%B3%E5%AD%A6%E4%B9%A0/#Im2colLayer)
      3. [1.5.3. LRNLayer](https://yufeigan.github.io/2014/12/09/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B03-Layer%E7%9A%84%E7%9B%B8%E5%85%B3%E5%AD%A6%E4%B9%A0/#LRNLayer)
      4. [1.5.4. PoolingLayer](https://yufeigan.github.io/2014/12/09/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B03-Layer%E7%9A%84%E7%9B%B8%E5%85%B3%E5%AD%A6%E4%B9%A0/#PoolingLayer)

## Layer

Layer是所有层的基类，在Layer的基础上衍生出来的有5种Layers：

* data\_layer
* neuron\_layer
* loss\_layer
* common\_layer
* vision\_layer

它们都有对应的[.hpp .cpp]文件声明和实现了各个类的接口。下面一个一个地讲这5个Layer。

## data\_layer

先看data\_layer.hpp中头文件调用情况：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 | #include "boost/scoped\_ptr.hpp" #include "hdf5.h" #include "leveldb/db.h" #include "lmdb.h" //前4个都是数据格式有关的文件 #include "caffe/blob.hpp" #include "caffe/common.hpp" #include "caffe/data\_transformer.hpp" #include "caffe/filler.hpp" #include "caffe/internal\_thread.hpp" #include "caffe/layer.hpp" #include "caffe/proto/caffe.pb.h" |

不难看出data\_layer主要包含与数据有关的文件。在官方文档中指出data是caffe数据的入口是网络的最低层，并且支持多种格式，在这之中又有5种LayerType：

* DATA
* MEMORY\_DATA
* HDF5\_DATA
* HDF5\_OUTPUT
* ￼￼￼￼￼IMAGE\_DATA

其实还有两种￼￼WINDOW\_DATA, DUMMY\_DATA用于测试和预留的接口，这里暂时不管。

### DATA

|  |  |
| --- | --- |
| 1 2 3 4 5 6 | template <typename Dtype> class BaseDataLayer : public Layer<Dtype> template <typename Dtype> class BasePrefetchingDataLayer : public BaseDataLayer<Dtype>, public InternalThread template <typename Dtype> class DataLayer : public BasePrefetchingDataLayer<Dtype> |

用于LevelDB或LMDB数据格式的输入的类型，输入参数有source, batch\_size, (rand\_skip), (backend)。后两个是可选。

### MEMORY\_DATA

|  |  |
| --- | --- |
| 1 2 | template <typename Dtype> class MemoryDataLayer : public BaseDataLayer<Dtype> |

这种类型可以直接从内存读取数据使用时需要调用MemoryDataLayer::Reset，输入参数有batch\_size, channels, height, width。

### HDF5\_DATA

|  |  |
| --- | --- |
| 1 2 | template <typename Dtype> class HDF5DataLayer : public Layer<Dtype> |

HDF5数据格式输入的类型，输入参数有source, batch\_size。

### HDF5\_OUTPUT

|  |  |
| --- | --- |
| 1 2 | template <typename Dtype> class HDF5OutputLayer : public Layer<Dtype> |

HDF5数据格式输出的类型，输入参数有file\_name。

### IMAGE\_DATA

|  |  |
| --- | --- |
| 1 2 | template <typename Dtype> class ImageDataLayer : public BasePrefetchingDataLayer<Dtype> |

图像格式数据输入的类型，输入参数有source, batch\_size, (rand\_skip), (shuffle), (new\_height), (new\_width)。

## neuron\_layer

先看neuron\_layer.hpp中头文件调用情况

|  |  |
| --- | --- |
| 1 2 3 4 | #include "caffe/blob.hpp" #include "caffe/common.hpp" #include "caffe/layer.hpp" #include "caffe/proto/caffe.pb.h" |

同样是数据的操作层，neuron\_layer实现里大量激活函数，主要是元素级别的操作，具有相同的bottom,topsize。  
Caffe中实现了大量激活函数GPU和CPU的都有很多。它们的父类都是NeuronLayer

|  |  |
| --- | --- |
| 1 2 | template <typename Dtype> class NeuronLayer : public Layer<Dtype> |

这部分目前没什么需要深究的地方值得注意的是一般的参数设置格式如下（以ReLU为例）：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 | ￼layers {  name: "relu1"  ￼type: RELU ￼ bottom: "conv1" ￼ top: "conv1" } |

## loss\_layer

Loss层计算网络误差，loss\_layer.hpp头文件调用情况：

|  |  |
| --- | --- |
| 1 2 3 4 5 | #include "caffe/blob.hpp" #include "caffe/common.hpp" #include "caffe/layer.hpp" #include "caffe/neuron\_layers.hpp" #include "caffe/proto/caffe.pb.h" |

可以看见调用了neuron\_layers.hpp，估计是需要调用里面的函数计算Loss，一般来说Loss放在最后一层。caffe实现了大量loss function，它们的父类都是LossLayer。

|  |  |
| --- | --- |
| 1 2 | template <typename Dtype> class LossLayer : public Layer<Dtype> |

## common\_layer

先看common\_layer.hpp头文件调用：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 | #include "caffe/blob.hpp" #include "caffe/common.hpp" #include "caffe/data\_layers.hpp" #include "caffe/layer.hpp" #include "caffe/loss\_layers.hpp" #include "caffe/neuron\_layers.hpp" #include "caffe/proto/caffe.pb.h" |

用到了前面提到的data\_layers.hpp, loss\_layers.hpp, neuron\_layers.hpp说明这一层肯定开始有复杂的操作了。  
这一层主要进行的是vision\_layer的连接  
声明了9个类型的common\_layer，部分有GPU实现：

* InnerProductLayer
* SplitLayer
* FlattenLayer
* ConcatLayer
* SilenceLayer
* (Elementwise Operations) 这里面是我们常说的激活函数层Activation Layers。
  + EltwiseLayer
  + SoftmaxLayer
  + ArgMaxLayer
  + MVNLayer

### InnerProductLayer

常常用来作为全连接层，设置格式为：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 | ￼layers {  name: "fc8"  ￼￼type: INNER\_PRODUCT ￼ blobs\_lr: 1 # learning rate multiplier for the filters ￼ blobs\_lr: 2 # learning rate multiplier for the biases ￼ weight\_decay: 1 # weight decay mu  weight\_decay: 0 # weight decay multiplier for the biases  inner\_product\_param {  ￼num\_output: 1000 ￼ weight\_filler { ￼ type: "gaussian" ￼ std: 0.01 ￼ } ￼ bias\_filler { ￼ type: "constant" ￼ value: 0 ￼ }  }  ￼bottom: "fc7" ￼ top: "fc8 } |

### SplitLayer

用于一输入对多输出的场合（对blob）

### FlattenLayer

将n \* c \* h \* w变成向量的格式n \* ( c \* h \* w ) \* 1 \* 1

### ConcatLayer

用于多输入一输出的场合。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 | ￼layers {  ￼name: "concat"  ￼bottom: "in1"  ￼bottom: "in2"  ￼top: "out"  ￼type: CONCAT  ￼concat\_param {  ￼ concat\_dim: 1  } } |

### SilenceLayer

用于一输入对多输出的场合（对layer）

### (Elementwise Operations)

EltwiseLayer, SoftmaxLayer, ArgMaxLayer, MVNLayer

## vision\_layer

头文件包含前面所有文件，也就是说包含了最复杂的操作。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 | #include "caffe/blob.hpp" #include "caffe/common.hpp" #include "caffe/common\_layers.hpp" #include "caffe/data\_layers.hpp" #include "caffe/layer.hpp" #include "caffe/loss\_layers.hpp" #include "caffe/neuron\_layers.hpp" #include "caffe/proto/caffe.pb.h" |

它主要是实现Convolution和Pooling操作。主要有以下几个类。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 | template <typename Dtype> class ConvolutionLayer : public Layer<Dtype> template <typename Dtype> class Im2colLayer : public Layer<Dtype> template <typename Dtype> class LRNLayer : public Layer<Dtype> template <typename Dtype> class PoolingLayer : public Layer<Dtype> |

### ConvolutionLayer

最常用的卷积操作，设置格式如下

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 | ￼layers {  name: "conv1" ￼ type: CONVOLUTION ￼ bottom: "data" ￼ top: "conv1" ￼ blobs\_lr: 1 # learning rate multiplier for the filters ￼ blobs\_lr: 2 # learning rate multiplier for the biases ￼ weight\_decay: 1 # weight decay multiplier for the filters ￼￼ weight\_decay: 0 # weight decay multiplier for the biases  convolution\_param { ￼ num\_output: 96 # learn 96 filters ￼ kernel\_size: 11 # each filter is 11x11 ￼ stride: 4 # step 4 pixels between each filter application  ￼weight\_filler { ￼￼￼ type: "gaussian" # initialize the filters from a Gaussian  std: 0.01 # distribution with stdev 0.01 (default mean: 0)  } ￼ bias\_filler { ￼￼ type: "constant" # initialize the biases to zero (0) ￼ value: 0  }  } ￼} |

### Im2colLayer

与MATLAB里面的im2col类似，即image-to-column transformation，转换后方便卷积计算

### LRNLayer

全称local response normalization layer，在Hinton论文中有详细介绍[ImageNet Classification with Deep Convolutional Neural Networks](http://www.cs.toronto.edu/~fritz/absps/imagenet.pdf)。

### PoolingLayer

即Pooling操作，格式：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 | layers {  name: "pool1"  type: POOLING  bottom: "conv1"  top: "pool1"  pooling\_param {  pool: MAX  kernel\_size: 3 # pool over a 3x3 region  stride: 2 # step two pixels (in the bottom blob) between pooling regions  } } |

# [Caffe学习笔记4-caffe安装需要注意的libraries](https://yufeigan.github.io/2014/12/14/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B04-caffe%E5%AE%89%E8%A3%85%E9%9C%80%E8%A6%81%E6%B3%A8%E6%84%8F%E7%9A%84libraries/)

By [YuFeiGan](http://yoursite.com/)

 2014-12-14 更新日期:2014-12-14

**文章目錄**

1. [1. Google Code](https://yufeigan.github.io/2014/12/14/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B04-caffe%E5%AE%89%E8%A3%85%E9%9C%80%E8%A6%81%E6%B3%A8%E6%84%8F%E7%9A%84libraries/#Google_Code)
   1. [1.1. glog](https://yufeigan.github.io/2014/12/14/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B04-caffe%E5%AE%89%E8%A3%85%E9%9C%80%E8%A6%81%E6%B3%A8%E6%84%8F%E7%9A%84libraries/#glog)
   2. [1.2. gflags](https://yufeigan.github.io/2014/12/14/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B04-caffe%E5%AE%89%E8%A3%85%E9%9C%80%E8%A6%81%E6%B3%A8%E6%84%8F%E7%9A%84libraries/#gflags)
   3. [1.3. gtest](https://yufeigan.github.io/2014/12/14/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B04-caffe%E5%AE%89%E8%A3%85%E9%9C%80%E8%A6%81%E6%B3%A8%E6%84%8F%E7%9A%84libraries/#gtest)
2. [2. 关于CPU加速](https://yufeigan.github.io/2014/12/14/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B04-caffe%E5%AE%89%E8%A3%85%E9%9C%80%E8%A6%81%E6%B3%A8%E6%84%8F%E7%9A%84libraries/#关于CPU加速)
   1. [2.1. 安装ATLAS](https://yufeigan.github.io/2014/12/14/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B04-caffe%E5%AE%89%E8%A3%85%E9%9C%80%E8%A6%81%E6%B3%A8%E6%84%8F%E7%9A%84libraries/#安装ATLAS)
   2. [2.2. 安装OpenBLAS](https://yufeigan.github.io/2014/12/14/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B04-caffe%E5%AE%89%E8%A3%85%E9%9C%80%E8%A6%81%E6%B3%A8%E6%84%8F%E7%9A%84libraries/#安装OpenBLAS)

当调用caffe训练数据的时候实际上调用的文件是在caffe.cpp中实现的train() 和 test()两个函数。搞清楚这两个函数对caffe的调用过程很有帮助。在这之前要提到几个相关的库，虽然与caffe实现关系不大，但完全不懂还是多少有点影响代码的理解。

## Google Code

## glog

google出的一个C++轻量级日志库，网络上的教程文档很多[English docs](http://google-glog.googlecode.com/svn/trunk/doc/glog.html), [Chinese docs](http://www.cnblogs.com/tianyajuanke/archive/2013/02/22/2921850.html), [glog使用与功能修改](http://www.cppfans.org/1566.html)。  
看代码的时候充斥这类似CHECK\_EQ、 CHECK\_NOTNULL、CHECK\_STREQ、CHECK\_DOUBLE\_EQ的函数这就是glog里面的函数，类似ASSERT()的断言。

## gflags

它是一个标记选项的库，看了这个[GFLAGS使用手册](http://download.csdn.net/detail/onlinesoon1/7713855)马上就能明白作用。另外的文档有[Google gflags使用说明](http://blog.csdn.net/lezardfu/article/details/23753741)。

## gtest

Google的开源C++单元测试框架，caffe里面test代码中大量用到，网上教程也是一大堆，中文里面最好的[玩转Google开源C++单元测试框架Google Test系列(gtest)(总)](http://www.cnblogs.com/coderzh/archive/2009/04/06/1426755.html)，英文文档[googletest](https://code.google.com/p/googletest/wiki/Samples)

## 关于CPU加速

Caffe推荐的BLAS（Basic Linear Algebra Subprograms）有三个选择ATLAS，Intel MKL，OpenBLAS。其中ATLAS是caffe是默认选择开源免费，如果没有安装CUDA的不太推荐使用，因为CPU多线程的支持不太好；Intel MKL是商业库要收费，我没有试过但caffe的作者安装的是这个库，估计效果应该是最好的；OpenBLAS开源免费，支持CPU多线程，我安装的就是这个。顺便贴上安装代码，其实很简单：

## 安装ATLAS

|  |  |
| --- | --- |
| 1 | sudo apt-get install libatlas-base-dev |

## 安装OpenBLAS

|  |  |
| --- | --- |
| 1 | sudo apt-get install libopenblas-base |

参考链接：  
[Caffe issues #16: Remove Intel MKL dependency](https://github.com/BVLC/caffe/issues/16)  
[Caffe issues #79: Support multithreading in the CPU mode of Solver::Solve](https://github.com/BVLC/caffe/issues/79)

# [Caffe学习笔记5-BLAS与boost::thread加速](https://yufeigan.github.io/2015/01/02/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B05-BLAS%E4%B8%8Eboost-thread%E5%8A%A0%E9%80%9F/)

By [YuFeiGan](http://yoursite.com/)

 2015-01-02 更新日期:2015-01-02

**文章目錄**

1. [1. BLAS是什么？](https://yufeigan.github.io/2015/01/02/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B05-BLAS%E4%B8%8Eboost-thread%E5%8A%A0%E9%80%9F/#BLAS是什么？)
2. [2. OpenBLAS](https://yufeigan.github.io/2015/01/02/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B05-BLAS%E4%B8%8Eboost-thread%E5%8A%A0%E9%80%9F/#OpenBLAS)
3. [3. Boost是什么？](https://yufeigan.github.io/2015/01/02/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B05-BLAS%E4%B8%8Eboost-thread%E5%8A%A0%E9%80%9F/#Boost是什么？)
   1. [3.1. ublas](https://yufeigan.github.io/2015/01/02/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B05-BLAS%E4%B8%8Eboost-thread%E5%8A%A0%E9%80%9F/#ublas)
   2. [3.2. date\_time](https://yufeigan.github.io/2015/01/02/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B05-BLAS%E4%B8%8Eboost-thread%E5%8A%A0%E9%80%9F/#date_time)
   3. [3.3. thread](https://yufeigan.github.io/2015/01/02/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B05-BLAS%E4%B8%8Eboost-thread%E5%8A%A0%E9%80%9F/#thread)
      1. [3.3.1. 线程的并行化](https://yufeigan.github.io/2015/01/02/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B05-BLAS%E4%B8%8Eboost-thread%E5%8A%A0%E9%80%9F/#线程的并行化)
      2. [3.3.2. 成员函数没有实例但又要传参的方法](https://yufeigan.github.io/2015/01/02/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B05-BLAS%E4%B8%8Eboost-thread%E5%8A%A0%E9%80%9F/#成员函数没有实例但又要传参的方法)
      3. [3.3.3. 线程同步互斥锁boost::mutex](https://yufeigan.github.io/2015/01/02/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B05-BLAS%E4%B8%8Eboost-thread%E5%8A%A0%E9%80%9F/#线程同步互斥锁boost::mutex)

Caffe中运用了大量的优化方法，最近在优化自己代码时候恰好运用了其中的BLAS和Boost::thread。使用过程中遇到了不少问题，这里把我遇到的问题和解决方法整理一下。

# BLAS是什么？

BLAS (Basic Linear Algebra Subprograms)基础线性代数子程序库，是一个应用程序接口（API）标准，说的简单点就是向量、矩阵之间乘加这些运算。BLAS虽然本身就有实现但效率不高，因此有大量的开源或商业项目对BLAS进行优化比如OpenBLAS（开源），Intel MKL（收费），ATLAS（开源）。我用的是OpenBLAS这个库。

# OpenBLAS

OpenBLAS是C语言实现的，这个库安装比较简单，没有什么问题，唯一的一个问题是使用方法。前面介绍BLAS提供了接口，文档在[这里](http://www.netlib.org/blas/blasqr.pdf)。  
这个文档中 Level 1 是vector与vector的操作，Level 2 是vector与matrix的操作，Level 3是matrix与matrix的操作。

每个函数的开头有一个x表示精度比如替换成s表示float类型，d表示double类型。

其实虽然函数很多但其实使用方法大同小异，BLAS之所以分的这么细（区分到对称矩阵，三角矩阵）是为了方便针对不同的情况做优化。所以其实搞清楚最关键的矩阵与矩阵的运算就已经理解了一大半。

以dgemm为例，全称为double-precision generic matrix-matrix muliplication，就是矩阵相乘，在OpenBLAS中的声明是:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 | cblas\_dgemm(const enum CBLAS\_ORDER Order,  const enum CBLAS\_TRANSPOSE TransA,  const enum CBLAS\_TRANSPOSE TransB,  const blasint M,  const blasint N,  const blasint K,  const double alpha,  const double \*A,  const blasint lda,  const double \*B,  const blasint ldb,  const double beta,  double \*C,  const blasint ldc) |

对应的公式如下：  
$$C\leftarrow\alpha op(A)op(B)+\beta C, op(X)=X, X^T, X^H, C-m\times n$$

参数的说明如下：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 | const enum CBLAS\_ORDER Order, \\指定矩阵的存储方式如RowMajor const enum CBLAS\_TRANSPOSE TransA, \\A运算后是否转置 const enum CBLAS\_TRANSPOSE TransB, \\B运算后是否转置 const blasint M, \\A的行数 const blasint N, \\B的列数 const blasint K, \\A的列数 const double alpha, \\公式中的alpha const double \*A, \\A const blasint lda, \\A一行的存储间隔 const double \*B, \\B const blasint ldb, \\B一行的存储间隔 const double beta, \\公式中的beta double \*C, \\C const blasint ldc \\C一行的存储间隔 |

因为这里的A、B、C矩阵都是以一维数组的形式存储所以需要告诉函数他们一行的存储间隔就是lda、ldb、ldc它们。

[OpenBLAS wiki](https://github.com/xianyi/OpenBLAS/wiki/User-Manual)上面的demo

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 | #include <cblas.h> #include <stdio.h> void main() {  int i=0;  double A[6] = {1.0,2.0,1.0,-3.0,4.0,-1.0};   double B[6] = {1.0,2.0,1.0,-3.0,4.0,-1.0};   double C[9] = {.5,.5,.5,.5,.5,.5,.5,.5,.5};   cblas\_dgemm(CblasColMajor, CblasNoTrans, CblasTrans,3,3,2,1,A, 3, B, 3,2,C,3);  for(i=0; i<9; i++)  printf("%lf ", C[i]);  printf("\n"); } |

# Boost是什么？

用[百度百科](http://baike.baidu.com/link?url=W5XnZZwZBBknw49TQlptNQ33fvYF2EfU3wZTN9NSvUVxcgtwMInwJLPaCiVGAjZpfubi9-vRgrbY7qejPpyX7z1wD8LY7U3gczavikzEUCu)的话说，Boost库是一个经过千锤百炼、可移植、提供源代码的C++库,作为标准库的后备,是C++标准化进程的发动机之一。

实际感受就是一个相当强大的C++拓展库，很多C++标准库里面没有的功能得以实现。最近就用到了ublas，thread，date\_time这三个模块。这里做一些简要的介绍。

## ublas

调用位置boost::numeric::ublas，是boost的BLAS实现，虽然速度一般，但用起来非常方便可以直接用 + - 运算符号操作，还可以直接用 << >> 等标准输入输出流。

## date\_time

在我计算多线程运行时间的时候发现标准C++提供的std::clock\_t对于多CPU跑线程的情况会把几个CPU的时间加在一起，所以采用了 boost::posix\_time::ptime 这种类型计数。解决了计时不准确的问题。

## thread

Boost提供的thread虽然功能据说不是非常强大，但是由于使用C++的思想重新设计，使用起来相对比较方便。网上文档非常多，比如：

1. [C++ Boost Thread 编程指南](http://www.cnblogs.com/chengmin/archive/2011/12/29/2306416.html)
2. [Boost::Thread使用示例](http://blog.csdn.net/zhuxiaoyang2000/article/details/6588031)  
   在实际运用中还是发现不少问题比如

### 线程的并行化

用一般的方法并不能真正的并行程序，比如[]这里的代码](http://stackoverflow.com/questions/6215511/how-to-run-multiple-threads-created-by-loop-simultaneous-using-boost-thread/6215646#6215646)：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 | std::vector<boost::thread \*> z; for (int i = 0; i < 2; ++i) z.push\_back(new boost::thread(workerFunc)); for (int i = 0; i < 2; ++i){  z[i]->join();  delete z[i]; } |

即使是这样在某些情况下我的线程还是不能并行运行，只是同时开始创建而已。  
解决方案是使用[boost::thread\_group](http://stackoverflow.com/questions/3344028/how-to-make-boostthread-group-execute-a-fixed-number-of-parallel-threads)

|  |  |
| --- | --- |
| 1 2 3 4 | boost::thread\_group group; for (int i = 0; i < 15; ++i)  group.create\_thread(aFunctionToExecute); group.join\_all(); |

当执行join\_all()的时候才是真正的并行了程序。

### 成员函数没有实例但又要传参的方法

编译时候错误：error: reference to non-static member function must be called; did you mean to call it with no arguments?  
查了Google发现是因为我定义的类中的成员函数用group.create\_thread()中调用了没有实例化的成员函数解决方法是[使用std::bind](http://stackoverflow.com/questions/20250860/can-you-initialize-a-thread-in-a-class-which-references-a-function-inside-that-c)  
我直接用的是boost::bind

|  |  |
| --- | --- |
| 1 | group.create\_thread(boost::bind(&myfunction, this)); |

### 线程同步互斥锁boost::mutex

接触过多线程的人一定不会陌生，当我们用操作操作统一数据的读写时或是数据输出输出时必须要用到互斥量。一开始用boost::mutex::scoped\_lock给boost::mutex上锁时只知道用[boost::condition](http://blog.csdn.net/fengge8ylf/article/details/6683527)控制线程的状态，后来发现代码越写越复杂，后来发现只需要改变boost::mutex作用域就可以自动给boost::mutex lock/unlock。比如如果每次只让一个线程输出信息到屏幕：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 | boost::mutex io\_mutex; void foo(){  {  boost::mutex::scoped\_lock lock(io\_mutex);  std::cout << "something output!" << std::endl;  }  // something to do! } |

用这种方法多个函数在对统一个数据操作的时候就不会有冲突了。

# [Caffe学习笔记6-Caffe结构简析（知乎同步）](https://yufeigan.github.io/2015/02/12/Caffe%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B06-Caffe%E7%BB%93%E6%9E%84%E7%AE%80%E6%9E%90%EF%BC%88%E7%9F%A5%E4%B9%8E%E5%90%8C%E6%AD%A5%EF%BC%89/)

By [YuFeiGan](http://yoursite.com/)

 2015-02-12 更新日期:2015-02-12

来自本人的知乎回答，[全文同步](http://www.zhihu.com/question/27982282/answer/39350629)

我是从Torch7转移到Caffe的人，仅供参考，当你阅读前你应该具备一些有关DL的基础知识，本文集中写Caffe代码结构而非介绍DL知识。  
我是去年底开始看Caffe代码的，看代码的时间加在一起也不到一个月，也算半个新手，我的回答是从新手角度作一个入门阶段的经验分享。

本文只涉及Caffe结构的相关问题，不涉及具体实现技巧等细节。

==============================================================

### 初识Caffe

1.1. Caffe相对与其他DL框架的优点和缺点：  
优点：  
速度快。Google Protocol Buffer数据标准为Caffe提升了效率  
更新快。还在不断地完善中，以后功能也会越来越多  
缺点：  
更新快。有人反映，偶尔会出现接口变换的情况，自己很久前写的代码可能过了一段时间就不能和新版本很好地兼容了。  
对于某些研究方向来说的人并不适合。这个需要对Caffe的结构有一定了解，（后面提到）。  
1.2. Caffe代码层次。  
回答里面有人说熟悉Blob，Layer，Net，Solver这样的几大类，我比较赞同。我基本是从这个顺序开始学习的，这四个类复杂性从低到高，贯穿了整个Caffe。把他们分为三个层次介绍。  
Blob：是基础的数据结构，是用来保存学习到的参数以及网络传输过程中产生数据的类。  
Layer：是网络的基本单元，由此派生出了各种层类。修改这部分的人主要是研究特征表达方向的。  
Net：是网络的搭建，将Layer所派生出层类组合成网络。Solver：是Net的求解，修改这部分人主要会是研究DL求解方向的。

==============================================================

### Caffe进阶

2.1. Blob：  
Caffe支持CUDA，在数据级别上也做了一些优化，这部分最重要的是知道它主要是对protocol buffer所定义的数据结构的继承，Caffe也因此可以在尽可能小的内存占用下获得很高的效率。（追求性能的同时Caffe也牺牲了一些代码可读性）  
在更高一级的Layer中Blob用下面的形式表示学习到的参数：

|  |  |
| --- | --- |
| 1 | vector<shared\_ptr<Blob<Dtype> > > blobs\_; |

这里使用的是一个Blob的容器是因为某些Layer包含多组学习参数，比如多个卷积核的卷积层。  
以及Layer所传递的数据形式，后面还会涉及到这里：

|  |  |
| --- | --- |
| 1 2 | vector<Blob<Dtype>\*> &bottom; vector<Blob<Dtype>\*> \*top |

2.2. Layer：  
2.2.1. 5大Layer派生类型  
Caffe十分强调网络的层次性，也就是说卷积操作，非线性变换（ReLU等），Pooling，权值连接等全部都由某一种Layer来表示。具体来说分为5大类Layer  
NeuronLayer类 定义于neuron\_layers.hpp中，其派生类主要是元素级别的运算（比如Dropout运算，激活函数ReLu，Sigmoid等），运算均为同址计算（in-place computation，返回值覆盖原值而占用新的内存）。  
LossLayer类 定义于loss\_layers.hpp中，其派生类会产生loss，只有这些层能够产生loss。  
数据层 定义于data\_layer.hpp中，作为网络的最底层，主要实现数据格式的转换。  
特征表达层（我自己分的类）定义于vision\_layers.hpp（为什么叫vision这个名字，我目前还不清楚），实现特征表达功能，更具体地说包含卷积操作，Pooling操作，他们基本都会产生新的内存占用（Pooling相对较小）。  
网络连接层和激活函数（我自己分的类）定义于common\_layers.hpp，Caffe提供了单个层与多个层的连接，并在这个头文件中声明。这里还包括了常用的全连接层InnerProductLayer类。  
2.2.2. Layer的重要成员函数  
在Layer内部，数据主要有两种传递方式，正向传导（Forward）和反向传导（Backward）。Forward和Backward有CPU和GPU（部分有）两种实现。Caffe中所有的Layer都要用这两种方法传递数据。

|  |  |
| --- | --- |
| 1 2 3 4 5 | virtual void Forward(const vector<Blob<Dtype>\*> &bottom,   vector<Blob<Dtype>\*> \*top) = 0; virtual void Backward(const vector<Blob<Dtype>\*> &top,  const vector<bool> &propagate\_down,   vector<Blob<Dtype>\*> \*bottom) = 0; |

Layer类派生出来的层类通过这实现这两个虚函数，产生了各式各样功能的层类。Forward是从根据bottom计算top的过程，Backward则相反（根据top计算bottom）。注意这里为什么用了一个包含Blob的容器（vector），对于大多数Layer来说输入和输出都各连接只有一个Layer，然而对于某些Layer存在一对多的情况，比如LossLayer和某些连接层。在网路结构定义文件（\*.proto）中每一层的参数bottom和top数目就决定了vector中元素数目。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 | layers {  bottom: "decode1neuron" // 该层底下连接的第一个Layer  bottom: "flatdata" // 该层底下连接的第二个Layer  top: "l2\_error" // 该层顶上连接的一个Layer  name: "loss" // 该层的名字  type: EUCLIDEAN\_LOSS // 该层的类型  loss\_weight: 0 } |

2.2.3. Layer的重要成员变量  
loss

|  |  |
| --- | --- |
| 1 | vector<Dtype> loss\_; |

每一层又有一个loss值，只不多大多数Layer都是0，只有LossLayer才可能产生非0的loss。计算loss是会把所有层的loss\_相加。  
learnable parameters

|  |  |
| --- | --- |
| 1 | vector<shared\_ptr<Blob<Dtype> > > blobs\_; |

前面提到过的，Layer学习到的参数。  
2.3. Net：  
Net用容器的形式将多个Layer有序地放在一起，其自身实现的功能主要是对逐层Layer进行初始化，以及提供Update( )的接口（更新网络参数），本身不能对参数进行有效地学习过程。

|  |  |
| --- | --- |
| 1 | vector<shared\_ptr<Layer<Dtype> > > layers\_; |

同样Net也有它自己的

|  |  |
| --- | --- |
| 1 2 3 | vector<Blob<Dtype>\*>& Forward(const vector<Blob<Dtype>\* > & bottom,  Dtype\* loss = NULL); void Net<Dtype>::Backward(); |

他们是对整个网络的前向和方向传导，各调用一次就可以计算出网络的loss了。  
2.4. Solver  
这个类中包含一个Net的指针，主要是实现了训练模型参数所采用的优化算法，它所派生的类就可以对整个网络进行训练了。

|  |  |
| --- | --- |
| 1 | shared\_ptr<Net<Dtype> > net\_; |

不同的模型训练方法通过重载函数ComputeUpdateValue( )实现计算update参数的核心功能

|  |  |
| --- | --- |
| 1 | virtual void ComputeUpdateValue() = 0; |

最后当进行整个网络训练过程（也就是你运行Caffe训练某个模型）的时候，实际上是在运行caffe.cpp中的train( )函数，而这个函数实际上是实例化一个Solver对象，初始化后调用了Solver中的Solve( )方法。而这个Solve( )函数主要就是在迭代运行下面这两个函数，就是刚才介绍的哪几个函数。

|  |  |
| --- | --- |
| 1 2 | ComputeUpdateValue(); net\_->Update(); |

==============================================================

至此，从底层到顶层对Caffe的主要结构都应该有了大致的概念。为了集中重点介绍Caffe的代码结构，文中略去了大量Caffe相关的实现细节和技巧，比如Layer和Net的参数如何初始化，proto文件的定义，基于cblas的卷积等操作的实现（cblas实现卷积这一点我的个人主页GanYuFei中的《Caffe学习笔记5-BLAS与boost::thread加速》有介绍）等等就不一一列举了。

整体来看Layer部分代码最多，也反映出Caffe比较重视丰富网络单元的类型，然而由于Caffe的代码结构高度层次化，使得某些研究以及应用（比如研究类似非逐层连接的神经网络这种复杂的网络连接方式）难以在该平台实现。这也就是一开始说的一个不足。

另外，Caffe基本数据单元都用Blob，使得数据在内存中的存储变得十分高效，紧凑，从而提升了整体训练能力，而同时带来的问题是我们看见的一些可读性上的不便，比如forward的参数也是直接用Blob而不是设计一个新类以增强可读性。所以说性能的提升是以可读性为代价的。  
最后一点也是最重要的一点，我从Caffe学到了很多。第一次看的C++项目就看到这么好的代码，实在是受益匪浅，在这里也感谢作者贾扬清等人的贡献。

甘宇飞更新于2/12/2015