# Caffe Source Code Analysis

## Caffe简介

[Caffe](http://caffe.berkeleyvision.org/" \t "https://buptldy.github.io/2016/10/09/2016-10-09-Caffe_Code/_blank)作为一个优秀的深度学习框架网上已经有很多内容介绍了，这里就不在多说。作为一个C++新手，断断续续看Caffe源码一个月以来发现越看不懂的东西越多，因此在博客里记录和分享一下学习的过程。其中我把自己看源码的一些注释结合了网上一些同学的注释以及在学习源码过程中查到到的一些资源(包括怎么使用IDE单步调试以及一些Caffe中使用的第三方库的介绍)放在github上：[Caffe\_Code\_Analysis](https://github.com/BUPTLdy/Caffe_Code_Analysis" \t "https://buptldy.github.io/2016/10/09/2016-10-09-Caffe_Code/_blank)，感兴趣的同学可以看一看，希望能对你有帮助。

一般在介绍Caffe代码结构的时候，大家都会说Caffe主要由Blob Layer Net 和 Solver这几个部分组成。

* Blob 主要用来表示网络中的数据，包括训练数据，网络各层自身的参数(包括权值、偏置以及它们的梯度)，网络之间传递的数据都是通过 Blob 来实现的，同时 Blob 数据也支持在 CPU 与 GPU 上存储，能够在两者之间做同步。
* Layer 是对神经网络中各种层的一个抽象，包括我们熟知的卷积层和下采样层，还有全连接层和各种激活函数层等等。同时每种 Layer 都实现了前向传播和反向传播，并通过 Blob 来传递数据。
* Net 是对整个网络的表示，由各种 Layer 前后连接组合而成，也是我们所构建的网络模型。
* Solver 定义了针对 Net 网络模型的求解方法，记录网络的训练过程，保存网络模型参数，中断并恢复网络的训练过程。自定义 Solver 能够实现不同的网络求解方式。

不过在刚开始准备阅读Caffe代码的时候，就算知道了代码是由上面四部分组成还是感觉会无从下手，下面我们准备通过一个[Caffe训练LeNet](http://caffe.berkeleyvision.org/gathered/examples/mnist.html" \t "https://buptldy.github.io/2016/10/09/2016-10-09-Caffe_Code/_blank)的实例并结合代码来解释Caffe是如何初始化网络，然后正向传播、反向传播开始训练，最终得到训练好的模型这一过程。

## 训练LeNet

在Caffe提供的例子里，训练LeNet网络的命令为：

|  |
| --- |
|  |

|  |
| --- |
| cd $CAFFE\_ROOT ./build/tools/caffe train --solver=examples/mnist/lenet\_solver.prototxt |

其中第一个参数build/tools/caffe是Caffe框架的主要框架，由[tools/caffe.cpp](https://github.com/BVLC/caffe/blob/master/tools/caffe.cpp" \t "https://buptldy.github.io/2016/10/09/2016-10-09-Caffe_Code/_blank)文件编译而来，第二个参数train表示是要训练网络，第三个参数是 [solver的protobuf描述文件](https://github.com/BVLC/caffe/blob/master/examples/mnist/lenet_solver.prototxt" \t "https://buptldy.github.io/2016/10/09/2016-10-09-Caffe_Code/_blank)。在Caffe中，网络模型的描述及其求解都是通过 [protobuf](https://developers.google.com/protocol-buffers/" \t "https://buptldy.github.io/2016/10/09/2016-10-09-Caffe_Code/_blank) 定义的，并不需要通过敲代码来实现。同时，模型的参数也是通过 protobuf 实现加载和存储，包括 CPU 与 GPU 之间的无缝切换，都是通过配置来实现的，不需要通过硬编码的方式实现，有关  
protobuf的具体内容可参考这篇博文：[http://alanse7en.github.io/caffedai-ma-jie-xi-2/](http://alanse7en.github.io/caffedai-ma-jie-xi-2/" \t "https://buptldy.github.io/2016/10/09/2016-10-09-Caffe_Code/_blank)。

## 网络初始化

下面我们从caffe.cpp的main函数入口开始观察Caffe是怎么一步一步训练网络的。在caffe.cpp中main函数之外通过RegisterBrewFunction这个宏在每一个实现主要功能的函数之后将这个函数的名字和其对应的函数指针添加到了g\_brew\_map中,具体分别为train()，test()，device\_query()，time()这四个函数。

在运行的时候,根据传入的参数在main函数中，通过GetBrewFunction得到了我们需要调用的那个函数的函数指针，并完成了调用。

|  |
| --- |
|  |

|  |
| --- |
| // caffe.cpp return GetBrewFunction(caffe::string(argv[1])) (); |

在我们上面所说的训练LeNet的例子中，传入的第二个参数为train，所以调用的函数为caffe.cpp中的int train()函数，接下来主要看这个函数的内容。在train函数中有下面两行代码，下面的代码定义了一个指向Solver的shared\_ptr。其中主要是通过调用SolverRegistry这个类的静态成员函数CreateSolver得到一个指向Solver的指针来构造shared\_ptr类型的solver。而且由于C++多态的特性，尽管solver是一个指向基类Solver类型的指针，通过solver这个智能指针来调用各个成员函数会调用到各个子类(SGDSolver等)的函数。

|  |
| --- |
|  |

|  |
| --- |
| // caffe.cpp // 其中输入参数solver\_param就是上面所说的第三个参数：网络的模型及求解文件 shared\_ptr<caffe::Solver<float> >  solver(caffe::SolverRegistry<float>::CreateSolver(solver\_param); |

因为在caffe.proto文件中默认的优化type为SGD,所以上面的代码会实例化一个SGDSolver的对象，’SGDSolver’类继承于Solver类，在新建SGDSolver对象时会调用其构造函数如下所示：

|  |
| --- |
|  |

|  |
| --- |
| //sgd\_solvers.hpp explicit SGDSolver(const SolverParameter& param)  : Solver<Dtype>(param) { PreSolve(); } |

从上面代码可以看出，会先调用父类Solver的构造函数，如下所示。Solver类的构造函数通过Init(param)函数来初始化网络。

|  |
| --- |
|  |

|  |
| --- |
| //solver.cpp template <typename Dtype> Solver<Dtype>::Solver(const SolverParameter& param, const Solver\* root\_solver)  : net\_(), callbacks\_(), root\_solver\_(root\_solver),requested\_early\_exit\_(false) {  Init(param); } |

而在Init(paran)函数中，又主要是通过InitTrainNet()和InitTestNets()函数分别来搭建训练网络结构和测试网络结构。

训练网络只能有一个,在InitTrainNet()函数中首先会设置一些基本参数，包括设置网络的状态为TRAIN，确定训练网络只有一个等，然会会通过下面这条语句新建了一个Net对象。InitTestNets()函数和InitTrainNet()函数基本类似，不再赘述。

|  |
| --- |
|  |

|  |
| --- |
| //solver.cpp net\_.reset(new Net<Dtype>(net\_param)); |

上面语句新建了Net对象之后会调用Net类的构造函数，如下所示。可以看出构造函数是通过Init(param)函数来初始化网络结构的。

|  |
| --- |
|  |

|  |
| --- |
| //net.cpp template <typename Dtype> Net<Dtype>::Net(const NetParameter& param, const Net\* root\_net)  : root\_net\_(root\_net) {  Init(param); } |

下面是net.cpp文件里Init()函数的主要内容(忽略具体细节)，其中LayerRegistry<Dtype>::CreateLayer(layer\_param)主要是通过调用LayerRegistry这个类的静态成员函数CreateLayer得到一个指向Layer类的shared\_ptr类型指针。并把每一层的指针存放在vector<shared\_ptr<Layer<Dtype> > > layers\_这个指针容器里。这里相当于根据每层的参数layer\_param实例化了对应的各个子类层，比如conv\_layer(卷积层)和pooling\_layer(池化层)。实例化了各层就会调用每个层的构造函数，但每层的构造函数都没有做什么大的设置。

接下来在Init()函数中主要由四部分组成：

* AppendBottom：设置每一层的输入数据
* AppendTop：设置每一层的输出数据
* layers\_[layer\_id]->SetUp：对上面设置的输入输出数据计算分配空间，并设置每层的可学习参数(权值和偏置),下面会详细降到这个函数
* AppendParam：对上面申请的可学习参数进行设置，主要包括学习率和正则率等。

|  |
| --- |
|  |

|  |
| --- |
| //net.cpp Init() for (int layer\_id = 0; layer\_id < param.layer\_size(); ++layer\_id) {//param是网络参数，layer\_size()返回网络拥有的层数  const LayerParameter& layer\_param = param.layer(layer\_id);//获取当前layer的参数  layers\_.push\_back(LayerRegistry<Dtype>::CreateLayer(layer\_param));//根据参数实例化layer   //下面的两个for循环将此layer的bottom blob的指针和top blob的指针放入bottom\_vecs\_和top\_vecs\_,bottom blob和top blob的实例全都存放在blobs\_中。相邻的两层，前一层的top blob是后一层的bottom blob，所以blobs\_的同一个blob既可能是bottom blob，也可能使top blob。  for (int bottom\_id = 0; bottom\_id < layer\_param.bottom\_size();++bottom\_id) {  const int blob\_id=AppendBottom(param,layer\_id,bottom\_id,&available\_blobs,&blob\_name\_to\_idx);  }   for (int top\_id = 0; top\_id < num\_top; ++top\_id) {  AppendTop(param, layer\_id, top\_id, &available\_blobs, &blob\_name\_to\_idx);  }  // 调用layer类的Setup函数进行初始化，输入参数：每个layer的输入blobs以及输出blobs,为每个blob设置大小 layers\_[layer\_id]->SetUp(bottom\_vecs\_[layer\_id], top\_vecs\_[layer\_id]);  //接下来的工作是将每层的parameter的指针塞进params\_，尤其是learnable\_params\_。  const int num\_param\_blobs = layers\_[layer\_id]->blobs().size();  for (int param\_id = 0; param\_id < num\_param\_blobs; ++param\_id) {  AppendParam(param, layer\_id, param\_id);  //AppendParam负责具体的dirtywork  }    } |

经过上面的过程，Net类的初始化工作基本就完成了，接着我们具体来看看上面所说的layers\_[layer\_id]->SetUp对每一具体的层结构进行设置，我们来看看Layer类的Setup()函数，对每一层的设置主要由下面三个函数组成：  
LayerSetUp(bottom, top)：由Layer类派生出的特定类都需要重写这个函数，主要功能是设置权值参数(包括偏置)的空间以及对权值参数经行随机初始化。  
Reshape(bottom, top)：根据输出blob和权值参数计算输出blob的维数，并申请空间。

|  |
| --- |
|  |

|  |
| --- |
| //layer.hpp // layer 初始化设置 void SetUp(const vector<Blob<Dtype>\*>& bottom,   const vector<Blob<Dtype>\*>& top) {  InitMutex();  CheckBlobCounts(bottom, top);  LayerSetUp(bottom, top);  Reshape(bottom, top);  SetLossWeights(top); } |

经过上述过程基本上就完成了初始化的工作，总体的流程大概就是新建一个Solver对象，然后调用Solver类的构造函数，然后在Solver的构造函数中又会新建Net类实例，在Net类的构造函数中又会新建各个Layer的实例,一直具体到设置每个Blob,大概就介绍完了网络初始化的工作，当然里面还有很多具体的细节，但大概的流程就是这样。

## 训练过程

上面介绍了网络初始化的大概流程，如上面所说的网络的初始化就是从下面一行代码新建一个solver指针开始一步一步的调用Solver，Net,Layer,Blob类的构造函数，完成整个网络的初始化。

|  |
| --- |
|  |

|  |
| --- |
| //caffe.cpp shared\_ptr<caffe::Solver<float> > //初始化  solver(caffe::SolverRegistry<float>::CreateSolver(solver\_param)); |

完成初始化之后，就可以开始对网络经行训练了，开始训练的代码如下所示，指向Solver类的指针solver开始调用Solver类的成员函数Solve()，名称比较绕啊。

|  |
| --- |
|  |

|  |
| --- |
| // 开始优化 solver->Solve(); |

接下来我们来看看Solver类的成员函数Solve(),Solve函数其实主要就是调用了Solver的另一个成员函数Step（）来完成实际的迭代训练过程。

|  |
| --- |
|  |

|  |
| --- |
| //solver.cpp template <typename Dtype> void Solver<Dtype>::Solve(const char\* resume\_file) {  ...  int start\_iter = iter\_;  ...  // 然后调用了'Step'函数，这个函数执行了实际的逐步的迭代过程  Step(param\_.max\_iter() - iter\_);  ...  LOG(INFO) << "Optimization Done."; } |

顺着来看看这个Step()函数的主要代码,首先是一个大循环设置了总的迭代次数，在每次迭代中训练iter\_size x batch\_size个样本，这个设置是为了在GPU的显存不够的时候使用，比如我本来想把batch\_size设置为128，iter\_size是默认为1的，但是会out\_of\_memory，借助这个方法，可以设置batch\_size=32，iter\_size=4，那实际上每次迭代还是处理了128个数据。

|  |
| --- |
|  |

|  |
| --- |
| //solver.cpp template <typename Dtype> void Solver<Dtype>::Step(int iters) {  ...  //迭代  while (iter\_ < stop\_iter) {  ...  // iter\_size也是在solver.prototxt里设置，实际上的batch\_size=iter\_size\*网络定义里的batch\_size，  // 因此每一次迭代的loss是iter\_size次迭代的和，再除以iter\_size，这个loss是通过调用`Net::ForwardBackward`函数得到的  // accumulate gradients over `iter\_size` x `batch\_size` instances  for (int i = 0; i < param\_.iter\_size(); ++i) {  /\*  \* 调用了Net中的代码，主要完成了前向后向的计算，  \* 前向用于计算模型的最终输出和Loss，后向用于  \* 计算每一层网络和参数的梯度。  \*/  loss += net\_->ForwardBackward();  }   ...   /\*  \* 这个函数主要做Loss的平滑。由于Caffe的训练方式是SGD，我们无法把所有的数据同时  \* 放入模型进行训练，那么部分数据产生的Loss就可能会和全样本的平均Loss不同，在必要  \* 时候将Loss和历史过程中更新的Loss求平均就可以减少Loss的震荡问题。  \*/  UpdateSmoothedLoss(loss, start\_iter, average\_loss);    ...  // 执行梯度的更新，这个函数在基类`Solver`中没有实现，会调用每个子类自己的实现  //，后面具体分析`SGDSolver`的实现  ApplyUpdate();   // 迭代次数加1  ++iter\_;  ...   } } |

上面Step()函数主要分为三部分：

### loss += net\_->ForwardBackward();

这行代码通过Net类的net\_指针调用其成员函数ForwardBackward()，其代码如下所示,分别调用了成员函数Forward(&loss)和成员函数Backward()来进行前向传播和反向传播。

|  |
| --- |
|  |

|  |
| --- |
| // net.hpp // 进行一次正向传播，一次反向传播 Dtype ForwardBackward() {  Dtype loss;  Forward(&loss);  Backward();  return loss; } |

前面的Forward(&loss)函数最终会执行到下面一段代码,Net类的Forward()函数会对网络中的每一层执行Layer类的成员函数Forward()，而具体的每一层Layer的派生类会重写Forward()函数来实现不同层的前向计算功能。上面的Backward()反向求导函数也和Forward()类似，调用不同层的Backward()函数来计算每层的梯度。

|  |
| --- |
|  |

|  |
| --- |
| //net.cpp for (int i = start; i <= end; ++i) { // 对每一层进行前向计算，返回每层的loss，其实只有最后一层loss不为0  Dtype layer\_loss = layers\_[i]->Forward(bottom\_vecs\_[i], top\_vecs\_[i]);  loss += layer\_loss;  if (debug\_info\_) { ForwardDebugInfo(i); } } |

### UpdateSmoothedLoss();

这个函数主要做Loss的平滑。由于Caffe的训练方式是SGD，我们无法把所有的数据同时放入模型进行训练，那么部分数据产生的Loss就可能会和全样本的平均Loss不同，在必要时候将Loss和历史过程中更新的Loss求平均就可以减少Loss的震荡问题

### ApplyUpdate();

这个函数是Solver类的纯虚函数，需要派生类来实现，比如SGDSolver类实现的ApplyUpdate();函数如下，主要内容包括：设置参数的学习率；对梯度进行Normalize；对反向求导得到的梯度添加正则项的梯度；最后根据SGD算法计算最终的梯度；最后的最后把计算得到的最终梯度对权值进行更新。

|  |
| --- |
|  |

|  |
| --- |
| template <typename Dtype> void SGDSolver<Dtype>::ApplyUpdate() {  CHECK(Caffe::root\_solver());   // GetLearningRate根据设置的lr\_policy来计算当前迭代的learning rate的值  Dtype rate = GetLearningRate();   // 判断是否需要输出当前的learning rate  if (this->param\_.display() && this->iter\_ % this->param\_.display() == 0) {  LOG(INFO) << "Iteration " << this->iter\_ << ", lr = " << rate;  }   // 避免梯度爆炸，如果梯度的二范数超过了某个数值则进行scale操作，将梯度减小  ClipGradients();   // 对所有可更新的网络参数进行操作  for (int param\_id = 0; param\_id < this->net\_->learnable\_params().size();  ++param\_id) {  // 将第param\_id个参数的梯度除以iter\_size，  // 这一步的作用是保证实际的batch\_size=iter\_size\*设置的batch\_size  Normalize(param\_id);   // 将正则化部分的梯度降入到每个参数的梯度中  Regularize(param\_id);   // 计算SGD算法的梯度(momentum等)  ComputeUpdateValue(param\_id, rate);  }  // 调用`Net::Update`更新所有的参数  this->net\_->Update(); } |

等进行了所有的循环，网络的训练也算是完成了。上面大概说了下使用Caffe进行网络训练时网络初始化以及前向传播、反向传播、梯度更新的过程，其中省略了大量的细节。上面还有很多东西都没提到，比如说Caffe中Layer派生类的注册及各个具体层前向反向的实现、Solver派生类的注册、网络结构的读取、模型的保存等等大量内容。

[# Deep Learning](https://buptldy.github.io/tags/Deep-Learning/) [# Caffe](https://buptldy.github.io/tags/Caffe/) [# C++](https://buptldy.github.io/tags/C/)

[Implementing convolution as a matrix multiplication](https://buptldy.github.io/2016/10/01/2016-10-01-im2col/" \o "Implementing convolution as a matrix multiplication)

[Transposed Convolution, Fractionally Strided Convolution or Deconvolution](https://buptldy.github.io/2016/10/29/2016-10-29-deconv/" \o "Transposed Convolution, Fractionally Strided Convolution or Deconvolution)