


# Caffe Source Code Analysis

 2016-10-09

The word "Caffe" is displayed in a large, red, serif font, centered within a white rectangular box.

Caffe简介

Caffe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的一些资源(包括如何使用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](#)，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：

```
1 cd $CAFFE_ROOT
```

```
2 ./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` 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。在Caffe中，网络模型描述及其求解都是通过 `protobuf` 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/>。

## 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。

```
1 // caffe.cpp
2 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。

```

1 // caffe.cpp
2 // 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
3 shared_ptr<caffe::Solver<float>>
4 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：

```

1 //sgd_solvers.hpp
2 explicit SGDSolver(const SolverParameter& param)
3 : 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。

```

1 //solver.cpp
2 template <typename Dtype>
3 Solver<Dtype>::Solver(const SolverParameter& param, const Solver* root_solver)
4 : net_(), callbacks_(), root_solver_(root_solver),requested_early_exit_(false)
5 {
6   Init(param);
7 }

```

而在 Init(param) 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，然后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。

```
1 //solver.cpp
2 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。

```
1 //net.cpp
2 template <typename Dtype>
3 Net<Dtype>::Net(const NetParameter& param, const Net* root_net)
4     : root_net_(root_net) {
5     Init(param);
6 }
```

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。

```

1  //net.cpp Init()
2  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
3      const LayerParameter& layer_param = param.layer(layer_id); //获取当前layer的参数
4      layers_.push_back(LayerRegistry<Dtype>::CreateLayer(layer_param)); //根据参数实例化layer
5
6
7  //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_
8      for (int bottom_id = 0; bottom_id < layer_param.bottom_size(); ++bottom_id) {
9          const int blob_id = AppendBottom(param, layer_id, bottom_id, &available_blobs, &blob_name_to_idx);
10     }
11
12     for (int top_id = 0; top_id < num_top; ++top_id) {
13         AppendTop(param, layer_id, top_id, &available_blobs, &blob_name_to_idx);
14     }
15
16     // 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
17     layers_[layer_id]->Setup(bottom_vecs_[layer_id], top_vecs_[layer_id]);
18
19     //接下来的工作是将每层的parameter的指针塞进params_，尤其是learnable_params_。
20     const int num_param_blobs = layers_[layer_id]->blobs().size();
21     for (int param_id = 0; param_id < num_param_blobs; ++param_id) {
22         AppendParam(param, layer_id, param_id);
23         //AppendParam负责具体的dirtywork
24     }
25
26
27     }

```

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进行随机初始化。

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。

```
1 //layer.hpp
2 // layer 初始化设置
3 void SetUp(const vector<Blob<Dtype>*>& bottom,
4     const vector<Blob<Dtype>*>& top) {
5     InitMutex();
6     CheckBlobCounts(bottom, top);
7     LayerSetUp(bottom, top);
8     Reshape(bottom, top);
9     SetLossWeights(top);
10 }
```

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。

```
1 //caffe.cpp
2 shared_ptr<caffe::Solver<float>> > //初始化
3 solver(caffe::SolverRegistry<float>::CreateSolver(solver_param));
```

完成初始化之后，就可以开始对网络进行训练了，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。

```
1 // 开始优化
2 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。

```
1 //solver.cpp
2 template <typename Dtype>
3 void Solver<Dtype>::Solve(const char* resume_file) {
4     ...
5     int start_iter = iter_;
6     ...
7     // 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
8     Step(param_.max_iter() - iter_);
9     ...
10    LOG(INFO) << "Optimization Done.";
11 }
```

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。

```
1 //solver.cpp
2 template <typename Dtype>
```



```
3 void Solver<Dtype>::Step(int iters) {
4     ...
5     //迭代
6     while (iter_ < stop_iter) {
7         ...
8         // 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，
9         // 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` i
10        // accumulate gradients over `iter_size` x `batch_size` instances
11        for (int i = 0; i < param_.iter_size(); ++i) {
12            /*
13             * 调用了Net中的代码，主要完成了前向后向的计算，
14             * 前向用于计算模型的最终输出和Loss，后向用于
15             * 计算每一层网络和参数的梯度。
16             */
17            loss += net_->ForwardBackward();
18        }
19        ...
20        ...
21        ...
22        /*
23         * 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
24         * 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
25         * 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。
26         */
27        UpdateSmoothedLoss(loss, start_iter, average_loss);
28        ...
29        ...
30        ...
31        // 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
32        //，后面具体分析`SGDSolver`的实现
33        ApplyUpdate();
34        ...
35        // 迭代次数加1
36        ++iter_;
37        ...
```

```
38
39     }
40 }
```

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。

```
1 // net.hpp
2 // 进行一次正向传播，一次反向传播
3 Dtype ForwardBackward() {
4     Dtype loss;
5     Forward(&loss);
6     Backward();
7     return loss;
8 }
```

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。

```
1 //net.cpp
2 for (int i = start; i <= end; ++i) {
3     // 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
4     Dtype layer_loss = layers_[i]->Forward(bottom_vecs_[i], top_vecs_[i]);
5     loss += layer_loss;
```

```
6     if (debug_info_) { ForwardDebugInfo(i); }
7 }
```

### 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，在必要时将Loss和历史过程中更新的Loss求平均可以减少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；最后最后把计算得到的最终梯度对权值进行更新。

```
1  template <typename Dtype>
2  void SGDSolver<Dtype>::ApplyUpdate() {
3      CHECK(Caffe::root_solver());
4
5      // GetLearningRate根据设置的lr_policy来计算当前迭代的learning rate的值
6      Dtype rate = GetLearningRate();
7
8      // 判断是否需要输出当前的learning rate
9      if (this->param_.display() && this->iter_ % this->param_.display() == 0) {
10         LOG(INFO) << "Iteration " << this->iter_ << ", lr = " << rate;
11     }
12
13     // 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
14     ClipGradients();
15 }
```

```
16 // 对所有可更新的网络参数进行操作
17 for (int param_id = 0; param_id < this->net_->learnable_params().size();
18     ++param_id) {
19     // 将第param_id个参数的梯度除以iter_size ,
20     // 这一步的作用是保证实际的batch_size=iter_size*设置的batch_size
21     Normalize(param_id);
22
23     // 将正则化部分的梯度降入到每个参数的梯度中
24     Regularize(param_id);
25
26     // 计算SGD算法的梯度(momentum等)
27     ComputeUpdateValue(param_id, rate);
28 }
29 // 调用`Net::Update`更新所有的参数
30 this->net_->Update();
31 }
```

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 衍生类的注册及各个具体层前向反向的实现、 Solver 衍生类的注册、网络结构的读取、模型的保存等等大量内容。

# Deep Learning    # Caffe    # C++

---

◀ Implementing convolution as a matrix  
multiplication

Transposed Convolution, Fractionally Strided ▶  
Convolution or Deconvolution

【版本更新】来必力：添加“GIF搜索引擎”功能。

撰写评论

发布

账号（邮件地址）

还没有评论，快来抢沙发吧！

© LiveRe.

© 2017 ♥ Ldy

Powered by [Hexo](#) | Theme - [NexT.Mist](#)