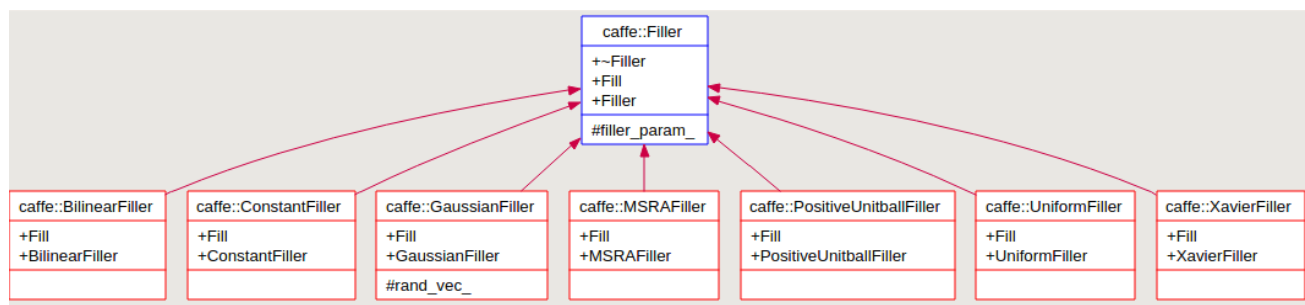


caffe中的数据初始化

如果我们要对一个神经网络从头开始进行训练的话，那么我们需要对网络的权重和偏执等参数进行初始化操作。在caffe的框架中提供了7种初始化的方法，我们来看看它们是如何实现的以及具体的使用方法吧。

Filler类及其派生类的继承关系



Filler是纯虚基类，BilinearFiller、ConstantFiller类等派生类都是要继承Filler这个基类的，并且派生类都必须要去实现Filler类中的Fill这个纯虚函数。该函数的目的就是将传入进来的Blob对象中的数据初始化。

(include/caffe/filler.hpp)

```
18 template <typename Dtype>
19 class Filler {
20 public:
21     explicit Filler(const FillerParameter& param) : filler_param_(param) {}
22     virtual ~Filler() {}
23     virtual void Fill(Blob<Dtype>* blob) = 0;           //留给外界调用的接口
24 protected:
25     FillerParameter filler_param_;
26 }; // class Filler
```

Filler类工厂函数

下面一段代码是生成各个Filler类派生类对象的代码，它根据type类型来生成相应的派生类对象。其实我们可以利用工厂设计模式来进行Filler派生类的生成，其代码和Solver类以及Layer类的工厂代码差不多。

(include/caffe/filler.hpp)

```
276 template <typename Dtype>
277 Filler<Dtype>* GetFiller(const FillerParameter& param) {
278     const std::string& type = param.type();
279     if (type == "constant") {
280         return new ConstantFiller<Dtype>(param);
281     } else if (type == "gaussian") {
282         return new GaussianFiller<Dtype>(param);
283     } else if (type == "positive_unitball") {
284         return new PositiveUnitballFiller<Dtype>(param);
285     } else if (type == "uniform") {
286         return new UniformFiller<Dtype>(param);
287     } else if (type == "xavier") {
```

```

288     return new XavierFiller<Dtype>(param);
289 } else if (type == "msra") {
290     return new MSRAFiller<Dtype>(param);
291 } else if (type == "bilinear") {
292     return new BilinearFiller<Dtype>(param);
293 } else {
294     CHECK(false) << "Unknown filler name: " << param.type();
295 }
296 return (Filler<Dtype>*)(NULL);
297 }

```

那我们自己试着用工厂模式来改写一下这一段代码吧。具体的代码不罗列出来了，filler的工厂类代码在(include/caffe/contrib/filler_factory.hpp)中，filler注册动作代码则是在(src/caffe/contrib/filler.cpp)。注意一个问题，在我们的prototxt文件中所有初始化的type都是小写字母，而我们要求的则是第一个字母为大写，如下面这一段：

(examples/mnist/lenet_train_test.prototxt)

```

layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: 20
    kernel_size: 5
    stride: 1
    weight_filler {
      type: "xavier"           //我们想要的是第一个字母是大写
    }
    bias_filler {
      type: "constant"        //我们想要的是第一个字母是大写
    }
  }
}
}

```

由于这样的一个特性我们需要又两处比较特别的改动。

第一处：我使用了一个try catch机制来解决这个问题，如果采用回调的方法不能生成对象的化那我们则采用原来的方法，这样我们就可以尝试将prototxt中的**xavier**改为**Xavier**，然后测试通过。

(include/caffe/filler.hpp)

```

template <typename Dtype>
Filler<Dtype>* GetFiller(const FillerParameter& param) {
#ifdef FISH_TEST

```

```

try{
return FillerRegistry<Dtype>::CreateFiller(param);    //采用回调机制来进行实例化
}
catch(...){    //如果这么做不成功的话，那么我们调用原来的实例化创建代码
}
const std::string& type = param.type();
if (type == "constant") {
return new ConstantFiller<Dtype>(param);
} else if (type == "gaussian") {
return new GaussianFiller<Dtype>(param);
} else if (type == "positive_unitball") {
return new PositiveUnitballFiller<Dtype>(param);
} else if (type == "uniform") {
return new UniformFiller<Dtype>(param);
} else if (type == "xavier") {
return new XavierFiller<Dtype>(param);
} else if (type == "msra") {
return new MSRAFiller<Dtype>(param);
} else if (type == "bilinear") {
return new BilinearFiller<Dtype>(param);
} else {
CHECK(false) << "Unknown filler name: " << param.type();
}
return (Filler<Dtype>*)(NULL);
#endif
}

```

第二处：这里向上抛出一个异常，可以通知上层程序这里出问题了，如果不抛出异常的话那么CHECK_EQ会终止整个程序。但是注意在程序中加异常处理程序一定要注意资源泄露的问题，这里的程序比较简单，没有资源泄露的问题，（其实不建议用异常处理机制，因为写一个异常安全的代码需要考虑到太多的问题了）。

(include/caffe/contrib/filler_factory.hpp)

```

static Filler<Dtype>* CreateFiller(const FillerParameter& param) {
const string& type = param.type();
CreatorRegistry& registry = Registry();
if(registry.count(type) != 1){
throw "error";    //向上抛出一个异常
}
CHECK_EQ(registry.count(type), 1) << "Unknown filler type: " << type
<< " (known types: " << FillerTypeListString() << ")";
return registry[type](param);
}

```

Filler类在哪里被调用

Filler类主要作用是初始化权重参数和偏置参数，而在一般情况下，一个层的权重参数和偏置参数都是每个层类的LayerSetUp函数中调用，因为在LayerSetUp函数中开辟好权重参数的内存空间，然后接着就调用Filler类来进行初权重参数初始化操作。

FillerParameter数据以及派生类解析

常量初始化类ConstantFiller

常量初始化就是将所有的权重值初始化成一个值。代码很简单，就不在这罗列了。

$$x = value \quad (1)$$

均匀分布初始化类UniformFiller

给定上限和下限，然后调用均匀分布发生器获得初始值，代码也很简单。

$$x \sim U(a, b) \quad (2)$$

高斯分布初始化类GaussianFiller

高斯分布分为两种，一种是单纯的高斯分布，另外一种是稀疏化的高斯分布。一般的高斯分布公式如下：

$$X \sim N(\mu, \sigma^2) \quad (3)$$

而稀疏化的高斯分布是将高斯分布的一些值归为0，我们具体来看一下代码吧。

```
64 template <typename Dtype>
65 class GaussianFiller : public Filler<Dtype> {
66 public:
67     explicit GaussianFiller(const FillerParameter& param)
68         : Filler<Dtype>(param) {}
69     virtual void Fill(Blob<Dtype>* blob) {
70         Dtype* data = blob->mutable_cpu_data();
71         CHECK(blob->count());
72         //下面是常规的高斯分布
73         caffe_rng_gaussian<Dtype>(blob->count(), Dtype(this->filler_param_.mean()),
74             Dtype(this->filler_param_.std()), blob->mutable_cpu_data());
75         int sparse = this->filler_param_.sparse();
76         CHECK_GE(sparse, -1);
77         //如果我们使用了稀疏化参数，则将高斯分布获得的矩阵进行稀疏化处理。
78         if (sparse >= 0) {
79             // Sparse initialization is implemented for "weight" blobs; i.e. matrices.
80             // These have num == channels == 1; width is number of inputs; height is
81             // number of outputs. The 'sparse' variable specifies the mean number
82             // of non-zero input weights for a given output.
83             CHECK_GE(blob->num_axes(), 1);
84             const int num_outputs = blob->shape(0);
85             Dtype non_zero_probability = Dtype(sparse) / Dtype(num_outputs);
86             //rand_vec用来记录0-1矩阵
87             rand_vec_.reset(new SyncedMemory(blob->count() * sizeof(int)));
88             int* mask = reinterpret_cast<int*>(rand_vec_->mutable_cpu_data());
89             //调用0-1分布来来获得0还是1的系数
90             caffe_rng_bernoulli(blob->count(), non_zero_probability, mask);
91             for (int i = 0; i < blob->count(); ++i) {
92                 //将原来的高斯分布乘以系数从而获得稀疏化的高斯分布
93                 data[i] *= mask[i];
94             }
95         }
96     }
97 }
```

```

92
93 protected:
94     shared_ptr<SyncedMemory> rand_vec_;
95 };

```

PositiveUnitballFiller初始化类

$$\begin{aligned} x &\in [0, 1] \\ \forall i \sum_j x_{ij} &= 1 \end{aligned} \quad (4)$$

这里可以举两个例子，如果我们要初始化的权重是全连接层的权重矩阵，那么经过该初始化之后这个权重所有值之和为1。如果我们的权重是好多个kernel，那么每个kernel的权重之和都为1。

XavierFiller初始化类

$$x \sim U(-scale, +scale) \quad (5)$$

这边的 $scale$ 和该层输入节点数、输出节点数相关。

$$scale = \sqrt{3/n} \quad (6)$$

n 可以是 fan_in 或者 fan_out 或者 fan_avg 。对于输入的一个blob(N, C, H, W), $fan_in = C \times H \times W$, $fan_out = N \times H \times W$ 。具体的我们是选用 fan_out 、 fan_in 还是 fan_avg ，这个得自己决定。

MSRAFiller初始化类

这个初始化和上面的类似，

$$x \sim N(0, \sigma^2) \quad (7)$$

这里的 σ 和输入输出相关。

$$\sigma = \sqrt{2/n} \quad (8)$$

n 可以是 fan_in 或者 fan_out 或者 fan_avg 。对于输入的一个blob(N, C, H, W), $fan_in = C \times H \times W$, $fan_out = N \times H \times W$ 。具体的我们是选用 fan_out 、 fan_in 还是 fan_avg ，这个得自己决定。

BilinearFiller初始化类

这个初始化之前并没有听过，好像在反卷积的时候用到的比较多，以后遇到了再完善这一块。