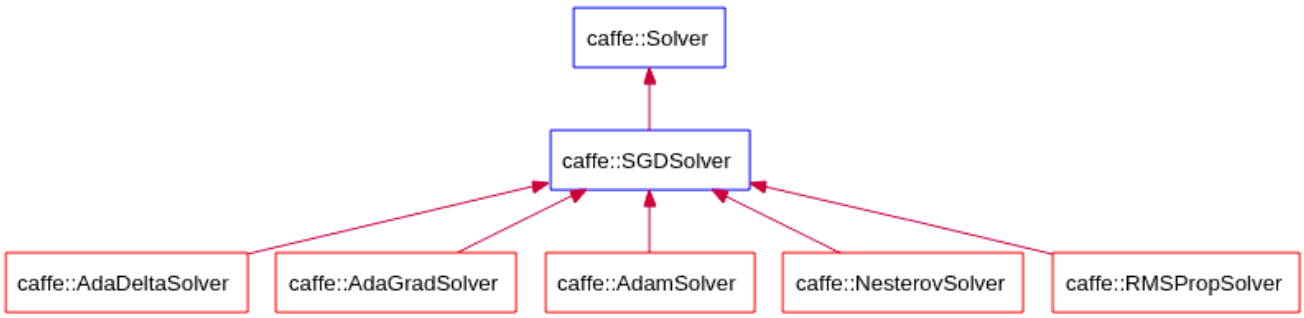
求解器

Solver类的层次关系

我们将caffe中的Solver类及其派生类的继承关系给画出来了，Solver类应该说是caffe层次最高的类，因为我们是通过Solver类一步步去创建Net类以及对应的Layer类，那今天可以具体看一下Solver类及其派生类的相关内容。



总体调用关系

我们在看Solver这部分代码的时候最主要还是要清楚整个求解器相关函数的调用顺序。这里我给出最关键的几个函数的调用顺序，当然这里是从Step函数开始的。关于Step函数的解析在这里并不多介绍，我们重点看的是求解器更新参数的过程，下面是最核心的一个调用关系。

Step函数——> ApllyUpdate函数——>GetLearningRate函数——>ClipGradients函数——>Normalize函数——>Regularize函数——>ComputeUpdateValue函数-——>最后调用net_的Update()函数。

首先我们看一下这几个函数的作用

ApllyUpdate函数

这个函数在父类SGDSolver中定义好，然后其他求解器子类都继承它，它调用了后面的GetLearningRate等等函数。

```
110 template <typename Dtype>
111 void SGDSolver<Dtype>::ApplyUpdate() {
112     Dtype rate = GetLearningRate();
113     if (this->param_.display() && this->iter_ % this->param_.display() == 0) {
114         LOG_IF(INFO, Caffe::root_solver()) << "Iteration " << this->iter_
115             << ", lr = " << rate;
116     }
117     ClipGradients();
118     for (int param_id = 0; param_id < this->net_->learnable_params().size();
119             ++param_id) {
120         Normalize(param_id);
121         Regularize(param_id);
122         ComputeUpdateValue(param_id, rate);
123     }
124     this->net_->Update();
125
126     // Increment the internal iter_ counter -- its value should always indicate
127     // the number of times the weights have been updated.
128     ++this->iter_;
```

```
129 }
```

当然这个ApplyUpdate函数实在Step函数中被调用的，那下面我们看看具体的子函数都干了些什么事情吧。

GetLearningRate函数

在深度学习训练中，我们的学习率并不是一直都是一个定值，它会随着迭代步数有一定的变化，那这边罗列了一下caffe中支持的学习率变化函数。

固定学习率：在训练过程中学习率永远都是base_lr，即

$$rate = base\_lr \tag{1}$$

step学习率:

$$rate = base\_lr * \gamma^{\lfloor iter/stepsize \rfloor} \tag{2}$$

exp学习率：

$$rate = base\_lr * \gamma^{iter} \tag{3}$$

inv学习率：

$$rate = base\_lr * (1 + \gamma * iter)^{-power} \tag{4}$$

multistep学习率：

$$rate = base\_lr * \gamma^{current\_step\_} \tag{5}$$

poly学习率：

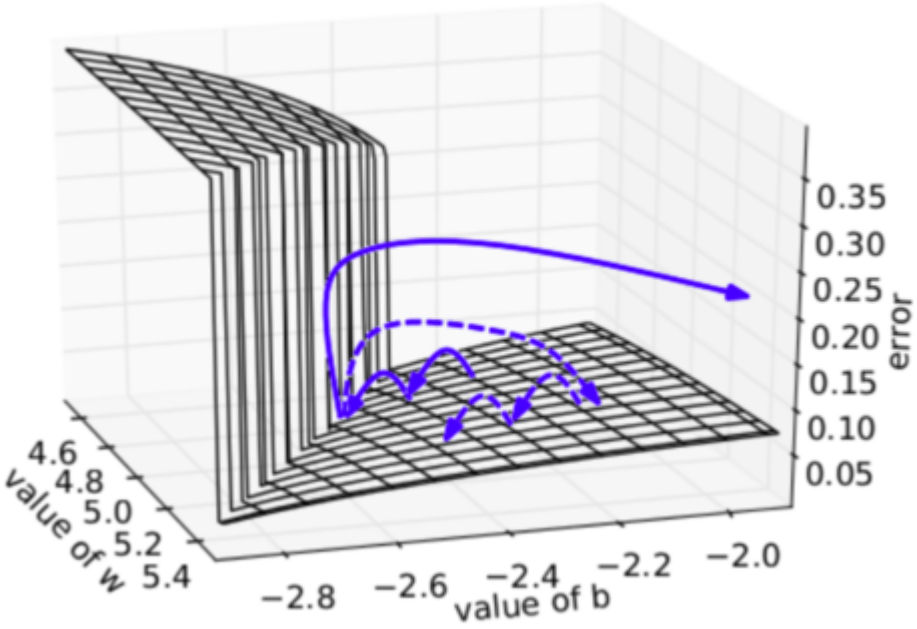
$$rate = base\_lr * (1 - iter/max\_iter)^{power} \tag{6}$$

sigmoid学习率：

$$rate = base\_lr * (1/(1 + exp(-\gamma * (iter - stepsize)))) \tag{7}$$

ClipGradients函数

在具体更新之前我们还得看看有没有梯度爆炸的情况，如果有我们需要做相应的裁剪动作。



一般梯度爆炸问题会在RNN中出现，我们可以看上面的图，整个误差函数出现了一个断截面，而且如果我们一不小心一脚踩到断截面处，那么我们求出来的梯度将会非常大。这样子我们以进行梯度更新就会直接坐✈飞走了，后面就可能不收敛。所以遇到这种梯度爆炸的问题，我们应该要限制它的梯度，具体可以看代码：

```
90 void SGDSolver<Dtype>::ClipGradients() {
91     const Dtype clip_gradients = this->param_.clip_gradients();
92     if (clip_gradients < 0) { return; }
93     const vector<Blob<Dtype>*>& net_params = this->net_->learnable_params();
94     Dtype sumsq_diff = 0;
95     for (int i = 0; i < net_params.size(); ++i) {
96         sumsq_diff += net_params[i]->sumsq_diff();
97     }
98     const Dtype l2norm_diff = std::sqrt(sumsq_diff);
99     if (l2norm_diff > clip_gradients) {
100        Dtype scale_factor = clip_gradients / l2norm_diff;
101        LOG(INFO) << "Gradient clipping: scaling down gradients (L2 norm "
102            << l2norm_diff << " > " << clip_gradients << ") "
103            << "by scale factor " << scale_factor;
104        for (int i = 0; i < net_params.size(); ++i) {
105            net_params[i]->scale_diff(scale_factor);
106        }
107    }
108 }
```

上述代码总的可以讲归纳成三步：

第一步：在Solver中设置一个clip_gradient的值。

第二步：反向传播之后求得相应的梯度，我们想去计算梯度的平方和sumsq_diff，如果这个sumsq_diff大于clip_greadient。那么求得缩放因子为scale_factor = clip_gradient / sumsq_diff。如果权重梯度的平方和越大，则缩放因子越小。

第三步：将所有的权重梯度乘以这个缩放因子，这时才得到最终的梯度信息。

上述这一顿操作可以保证我们的梯度大小可以在合理的范围内，即梯度的平方和不会超过clip_gradient。但是这个clip_gradient需要自己设定。

Normalize函数

这一步同样考虑了一些单一Batch不足以完成训练的问题，通过限制每个Batch的更新量来控制更新总量，代码比较简单。

Regularize函数

这边正则化我们根据caffe源代码的实现中我们只对L1正则化和L2正则化进行介绍，其所对应的数学解释可以参考《深度学习》一书的第7章。

L1正则化：我们将在传统的损失函数后面添加L1正则化，那么更新过的损失函数就变为：

$$\tilde{J}(w; X, y) = \alpha \|w\|_1 + J(w; X, y) \tag{8}$$

那该损失函数的梯度为：

$$\nabla_w \tilde{J}(w; X, y) = \alpha \operatorname{sign}(w) + \nabla_w J(w; X, y) \tag{9}$$

传统的损失函数梯度还是一样，这里多了一项L1正则化梯度$\alpha \operatorname{sign}(w)$，另外需要注意这里的$\alpha$是**权重衰减参数**（weight decay）。

那么其简单的梯度更新函数为：

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \epsilon \left(\alpha \operatorname{sign}(w) + \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})\right) \tag{10}$$

L2正则化：在一些其他的方面，L2正则化可能也会被成为岭回归或者Tikhonov正则，它同样有权重衰减参数，那添加了L2正则化的损失函数就变为：

$$\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \frac{\alpha}{2} \boldsymbol{w}^\top \boldsymbol{w} + J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) \tag{11}$$

损失函数的梯度为：

$$\nabla_w \tilde{J}(w; X, y) = \alpha w + \nabla_w J(w; X, y) \tag{12}$$

那么其简单的梯度更新函数为：

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \epsilon \left(\alpha \boldsymbol{w} + \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})\right) \tag{13}$$

那我们可以简单的看一下代码是如何实现这一正则化操作的：

```
if (local_decay) {
    if (regularization_type == "L2") {
        caffe_axpy(net_params[param_id]->count(),        //调用数学库函数将梯度更新到cpu_diff中去
                local_decay,
                net_params[param_id]->cpu_data(),
                net_params[param_id]->mutable_cpu_diff());
    } else if (regularization_type == "L1") {
        caffe_cpu_sign(net_params[param_id]->count(),    //L1正则化第一步则是获取其符号位
                net_params[param_id]->cpu_data(),
                temp_[param_id]->mutable_cpu_data());
```

```
        caffe_axpy(net_params[param_id]->count(),       //更新梯度
                local_decay,
                temp_[param_id]->cpu_data(),
                net_params[param_id]->mutable_cpu_diff());
    } else {
        LOG(FATAL) << "Unknown regularization type: " << regularization_type;
    }
}
break;
```

这里要讲一个问题，关于L1正则化和L2正则化做增加的损失项的反向传播过程并不是在各个Layer层中体现的，而是在Solver这边进行统一处理的。其实L1正则化和L2正则化只会直接影响到虽对应的权重，其实并没有一个反向传播的过程，所以也没有必要在Layer中进行体现。

ComputeUpdateValue函数

这个是一个虚函数，SGDSolver类定义了自己的更新方法，每个Solver派生类需要自己重新定义该方法，将各自的更新参数的方法写到这里面，通过ComputeUpdateValue函数计算，梯度的信息都会保存在参数blob的diff内存块中。

net->Update函数

最后调用net->Update函数来进行更新梯度。那我们看一下是如何进行的吧，我们可以看到它最终会调用参数blob的Update函数，我们在来看一下Blob中的Update函数。

```
1000 template <typename Dtype>
1001 void Net<Dtype>::Update() {
1002     for (int i = 0; i < learnable_params_.size(); ++i) {
1003         learnable_params_[i]->Update();
1004     }
1005 }
```

那就到权重更新的最后一个步骤了，我们来看一下最终是如何更新的，代码如下，看到这里一切都明了了，在blob的Update函数中调用一个向量减法来进行权重更新。

```
 226             caffe_axpy<Dtype>(count_, Dtype(-1),
 227                     static_cast<const Dtype*>(diff_->cpu_data()),
 228                     static_cast<Dtype*>(data_->mutable_cpu_data()));
```

所以上述整个过程就是权重更细的过程，但是具体是如何更新的则在ComputeUpdateValue函数中定义，这个可以根据自身需求来读这一部分的源码，也可以对其进行修改，后面对常用的权重更新算法稍微总结一下，同时对几个比较简单算法的源码看了一下。

SGDSolver

随机梯度下降的算法描述

Require：学习率为ϵ。

Require：初始化参数θ。

while停止准则未满足do

从训练集中采m个样本$\{\boldsymbol{x}^{(1)},\ldots,\boldsymbol{x}^{(m)}\}$的小批量，对应的目标为$y^{(i)}$。

计算梯度：$g \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}}\sum_i L\left(f\left(\boldsymbol{x}^{(i)};\boldsymbol{\theta}\right),\boldsymbol{y}^{(i)}\right)$。

应用更新：$\theta \leftarrow \theta - \epsilon g$

end while

该算法是最简单的梯度下降算法，但是caffe实现的是下面这种带有动量版本的梯度下降算法。

使用动量的随机梯度下降算法描述

Require：学习率为ϵ，动量参数为α。

Require：初始化参数θ，初始速度为v。

while没有达到停止准则do 从训练集中采m个样本$\{\boldsymbol{x}^{(1)},\ldots,\boldsymbol{x}^{(m)}\}$的小批量，对应的目标为$y^{(i)}$。

计算梯度：$g \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}}\sum_i L\left(f\left(\boldsymbol{x}^{(i)};\boldsymbol{\theta}\right),\boldsymbol{y}^{(i)}\right)$。

计算速度更新：$v \leftarrow \alpha v - \epsilon g$

应用更新：$\theta \leftarrow \theta + v$。

end while

那我们看一下CPU端的代码是如何实现的吧：

首先是计算速度更新：

history_ = momentum * history_ + local_rate * cpu_diff();

这个history存的就是更新之后的速度。

```
234                             caffe_cpu_axpby(net_params[param_id]->count(), local_rate,
235                                     net_params[param_id]->cpu_diff(), momentum,
236                                     history_[param_id]->mutable_cpu_data());
```

将更新之后的梯度存放到各个学习参数blob的cpu_diff()中。

```
237                             caffe_copy(net_params[param_id]->count(),
238                                     history_[param_id]->cpu_data(),
239                                     net_params[param_id]->mutable_cpu_diff());
240                             break;
```

最后调用各个参数blob的Update参数更新权重值和偏置值。

Nesterov动量算法描述

Require：学习率为ϵ，动量参数为α。

Require：初始化参数θ，初始速度为v。

while没有达到停止准则do 从训练集中采m个样本$\{\boldsymbol{x}^{(1)},\ldots,\boldsymbol{x}^{(m)}\}$的小批量，对应的目标为$y^{(i)}$。

应用零时更新：$\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha\boldsymbol{v}$。

计算梯度（在临时点）：$g \leftarrow \frac{1}{m}\nabla_{\tilde{\theta}} \sum_i L\left(f\left(\boldsymbol{x}^{(i)};\tilde{\boldsymbol{\theta}}\right),\boldsymbol{y}^{(i)}\right)$。

计算速度更新：$v \leftarrow \alpha v - \epsilon g$。

应用更新：$\theta \leftarrow \theta + v$。

end while

这个算法的伪代码我是摘自《深度学习》这本书的，刚开始我对这个算法有一个致命的误解，我刚开始一位这个算法每个迭代步数会有两个前向传播和反向传播（因为这边有一个在零时点的梯度更新），当时还非常愚蠢的想这种算法能快吗，后来看了源码之后才理解了这一切。

AdaGradSolver

算法描述

Require：全局学习率ϵ。

Require：初始化参数θ。

Require：小常数δ，为了数值稳定大约设置在10^{-7}。

初始化梯度累积量$r = 0$

while没有达到停止准则do

从训练集中采m个样本$\left\{\boldsymbol{x}^{(1)},\ldots,\boldsymbol{x}^{(m)}\right\}$的小批量，对应的目标为$y^{(i)}$。

计算梯度：$g \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L\left(f\left(\boldsymbol{x}^{(i)};\boldsymbol{\theta}\right),\boldsymbol{y}^{(i)}\right)$。

累计平方梯度：$r \leftarrow r + g \odot g$。 //这里的$g \odot g$是点积的意思。

计算更新：$\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta+\sqrt{r}} \odot g$。

应用更新：$\theta \leftarrow \theta + \Delta\theta$。

end while

AdamSolver

算法描述

Require：学习率ϵ（建议默认值为：0.001）。

Require：矩估计的指数衰减速率，ρ_1和ρ_2在区间[0, 1)内（建议为：0.9和0.999）。

Require：用于数值稳定的小常数δ（建议默认为：10^{-8}）。

Require：初始参数θ。

初始化一阶和二阶矩变量$s = 0, r = 0$。

初始化时间步长$t = 0$。

while没有达到停止准则do

从训练集中采m个样本$\left\{\boldsymbol{x}^{(1)},\ldots,\boldsymbol{x}^{(m)}\right\}$的小批量，对应的目标为$y^{(i)}$。

计算梯度：$g \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L\left(f\left(\boldsymbol{x}^{(i)};\boldsymbol{\theta}\right),\boldsymbol{y}^{(i)}\right)$。

更新时间步长：$t \leftarrow t + 1$

更新有偏一阶矩估计：$s \leftarrow \rho_1 s + (1-\rho_1)\, g$

更新有偏二阶矩估计：$r \leftarrow \rho_2 r + (1-\rho_2)\, g \odot g$

修正一阶矩的偏差：$\hat{\boldsymbol{s}} \leftarrow \frac{s}{1-\rho_1^t}$

修正二阶矩的偏差：$\hat{r} \leftarrow \frac{r}{1-\rho_2^t}$

计算更新：$\Delta\theta = -\epsilon \frac{\hat{s}}{\sqrt{\hat{r}}+\delta}$（逐元素应用操作）

应用更新：$\theta \leftarrow \theta + \Delta\theta$。

NesterovSolver

算法描述

Require：全局学习率ϵ，衰减速率ρ，动量系数α。

Require：初始参数θ，初始速度v。

初始化累计量$r = 0$。

while没有达到停止准则do

从训练集中采m个样本$\left\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\right\}$的小批量，对应的目标为$y^{(i)}$。

计算临时更新：$\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$。

计算梯度：$g \leftarrow \frac{1}{m} \nabla_{\tilde{\theta}} \sum_i L\left(f\left(x^{(i)}; \tilde{\theta}\right), y^{(i)}\right)$。

累计梯度：$\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1-\rho) \boldsymbol{g} \odot \boldsymbol{g}$。

计算速度更新：$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \frac{\epsilon}{\sqrt{r}} \odot \boldsymbol{g}$（$\frac{1}{\sqrt{r}}$逐元素应用）

应用更新：$\theta \leftarrow \theta + v$

end while

RMSPropSolver

算法描述

Require：全局学习率ϵ，衰减速率ρ。

Require：初始化参数θ。

Require：小常数δ，通常设置为10^{-6}，用于被小数除时的数值稳定。

初始化累积变量$r = 0$。

while没有达到停止准则do

从训练集中采m个样本$\left\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\right\}$的小批量，对应的目标为$y^{(i)}$。

计算梯度：$g \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L\left(f\left(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right), \boldsymbol{y}^{(i)}\right)$。

累计平方梯度：$r \leftarrow \rho r + (1-\rho) g \odot g$。

计算参数更新：$\Delta \boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta+r}} \odot g$（$\frac{1}{\sqrt{\delta+r}}$逐元素应用）。

应用更新：$\theta \leftarrow \theta + \Delta\theta$。

end while

第一步：计算梯度的按元素平方$g \odot g$

```
26    caffe_powx(net_params[param_id]->count(),
27        net_params[param_id]->cpu_diff(), Dtype(2),
28        this->update_[param_id]->mutable_cpu_data());
```

第二步：计算累计平方梯度$r \leftarrow \rho r + (1-\rho) g \odot g$

```
31    caffe_cpu_axpby(net_params[param_id] -> count(),
32        Dtype(1-rms_decay), this->update_[param_id]->cpu_data(),
33        rms_decay, this->history_[param_id]-> mutable_cpu_data());
```

第三步：开根号并按元素加为小量

这里和书上的公式稍微有点不同，因为书上的是先加微小量再开根号，不过这个不影响，本来微小量就是为了防止除零错误，放在外面和里面没啥区别。

```
36    caffe_powx(net_params[param_id]->count(),
37        this->history_[param_id]->cpu_data(), Dtype(0.5),
38        this->update_[param_id]->mutable_cpu_data());
39
40    caffe_add_scalar(net_params[param_id]->count(),
41        delta, this->update_[param_id]->mutable_cpu_data());
```

第四步：按元素来除上一步得到的值，并乘以学习率ϵ

```
43    caffe_div(net_params[param_id]->count(),
44        net_params[param_id]->cpu_diff(), this->update_[param_id]->cpu_data(),
45        this->update_[param_id]->mutable_cpu_data());
```

第五步：最后调用net的Update函数来更新梯度