BraveApple 's Blog (http://blog.leanote.com/braveapple)

coding will change world!

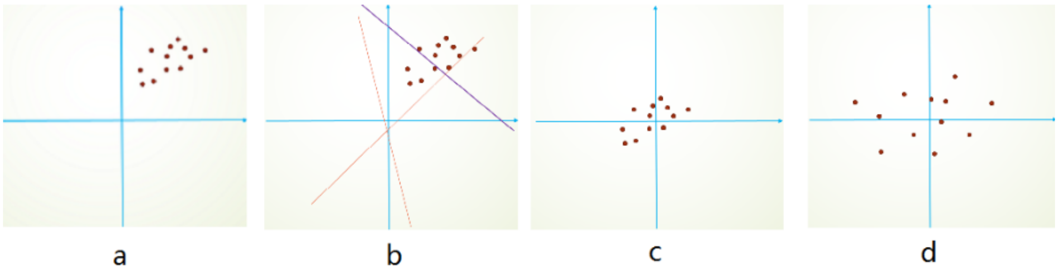
解析 Caffe 之 Batch Normalization Layer

文档导航

1、原理分析

在没有 Batch Normalization 之前，我们的调参工作十分困难，而且网络收敛速度不快。在训练过程中，上一层的参数会不断地变化，导致输出的分布也在不断变化，这样下一层不仅要学习数据自生的分布，而且还要花多余的精力去学习上一层输出的分布，这种现象称之为 internal covariate shift。因此，我们要选择小的学习率和小心翼翼地初始化网络，大大地降低了网络的学习速度。

我们通常在训练网络，要对输入减去均值，甚至还会做数据白化，目的是为了加快训练。为什么减均值和数据白化，会加快训练？我们这里给出四副图片，做简单地解释：



首先，图像数据具有高度相关性，我们简化图像数据，如图 (a)。由于网络初始化时，参数的均值一般为 0，所以刚开始网络拟合函数 $y = \boldsymbol{wx} + b$ 会在原点附近，如图 (b) 中的红色直线。因此，网络经过多次的训练后才会达到最优的紫色直线。如果我们在训练网络之前减去均值，如图 (c)。因此，我们只需要更少的训练次数就会达到最优，加快训练进度。如果我们再对数据进行去相关化操作，增加数据的区分度，进一步加快训练进度。

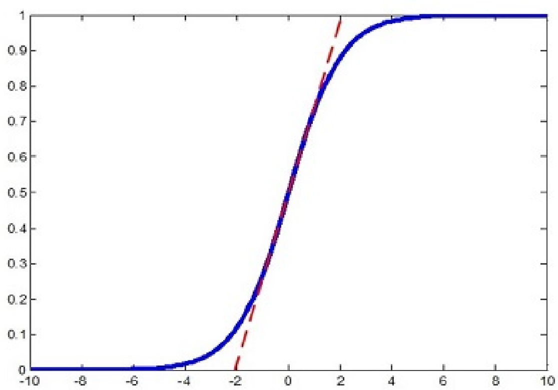
在论文 Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift (https://arxiv.org/pdf/1502.03167.pdf) 中，作者首先考虑对每一层输出数据进行白化操作，但是分析这种方法是不可行的。因为数据白化需要计算协方差矩阵、求逆等操作，计算量很大，而且在反向传播过程中，白化操作不一定可导。

作者最后提出一种基于 mini-batch 统计的数据归一化方法，这样大量地减少了计算量，同时保证了模型收敛速度。对于某一层，输入 d 维数据 $\boldsymbol{x} = (x^{(1)}, x^{(2)}, \ldots, x^{(d)})$，对数据做简单的归一化操作，使得数据均值为 0，方差为 1：

$$\hat{y}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{Var[x^{(k)}]}} \tag{1}$$

其中，$E[x^{(k)}]$ 和 $Var[x^{(k)}]$ 都是基于 mini-batch 的数据求平均和方差，用一个 batch 的统计均值和方差代替整个数据的均值和方差。

但是，作者又提出：如果直接将归一化的数据输入 sigmoid 激活函数，会把数据限制在 $x^{(k)} \in [-1, +1]$ 的激活函数的线性部分，降低网络的非线性表达能力。



所以，最后做了 Scale 和 Shift 操作，通过增加 $\gamma^{(k)}$ 和 $\beta^{(k)}$ 两个参数，保证了网络的非线性表达能力。

$$\hat{y}^{(k)} = \gamma^{(k)} x^{(k)} + \beta^{(k)} \tag{2}$$

Batch Normalization 的算法流程如下：

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

$$\frac{\partial \ell}{\partial \widehat{x}_i} = \frac{\partial \ell}{\partial y_i} \cdot \gamma$$

$$\frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial \widehat{x}_i} \cdot (x_i - \mu_{\mathcal{B}}) \cdot \frac{-1}{2}(\sigma_{\mathcal{B}}^2 + \epsilon)^{-3/2}$$

$$\frac{\partial \ell}{\partial \mu_{\mathcal{B}}} = \left(\sum_{i=1}^{m} \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}\right) + \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{\sum_{i=1}^{m} -2(x_i - \mu_{\mathcal{B}})}{m}$$

$$\frac{\partial \ell}{\partial x_i} = \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} + \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{2(x_i - \mu_{\mathcal{B}})}{m} + \frac{\partial \ell}{\partial \mu_{\mathcal{B}}} \cdot \frac{1}{m}$$

$$\frac{\partial \ell}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i} \cdot \widehat{x}_i$$

$$\frac{\partial \ell}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i}$$

2、源码分析

2.1、BatchNormParameter

```
message BatchNormParameter {
  // If false, accumulate global mean/variance values via a moving average. If
  // true, use those accumulated values instead of computing mean/variance
  // across the batch.
  // 如果为假，采用滑动平均对全局的均值和方差进行累加。如果为真，则使用全局的均值
  // 和方差，而不是使用每个 batch 的均值和方差
  optional bool use_global_stats = 1;
  // How much does the moving average decay each iteration?
  // 滑动平均的衰减系数，默认为 0.999
  optional float moving_average_fraction = 2 [default = .999];
  // Small value to add to the variance estimate so that we don't divide by
  // zero.
  // 分母的附加值，防止除 0 的情况，默认值为 1e-5
  optional float eps = 3 [default = 1e-5];
}
```

2.2、成员变量

建议一般不要主动去设置 `use_global_stats_` 成员变量，默认训练阶段为 `false`，测试阶段为 `true`。如果在训练阶段设置 `use_global_stats_ = true`，会使网络出现梯度爆炸的情况。

```
// 均值、方差、中间值、归一化值
  Blob<Dtype> mean_, variance_, temp_, x_norm_;
  // 如果为假，采用滑动平均对全局的均值和方差进行累加。如果为真，则使用全局的均值
  // 和方差，而不是使用每个 batch 的均值和方差
  bool use_global_stats_;
  // 滑动平均的衰减系数，默认为 0.999
  Dtype moving_average_fraction_;
  int channels_;
  // 分母的附加值，防止除 0 的情况，默认值为 1e-5
  Dtype eps_;

  // extra temporarary variables is used to carry out sums/broadcasting
  // using BLAS
  // 维度是 (batch_size, )
  Blob<Dtype> batch_sum_multiplier_;
  // 维度是 (batch_size, channels)
  Blob<Dtype> num_by_chans_;
  // 维度是 (height, width)
  Blob<Dtype> spatial_sum_multiplier_;
```

2.3、LayerSetUp()

注意 `this->blobs_[0]`、`this->blobs_[1]`、`this->blobs_[2]` 分别是global均值、global方差和滑动平均衰减系数；而 `this->mean_`、`this->variance_` 分别是均值和方差。

```
// 层设置函数
template <typename Dtype>
void BatchNormLayer<Dtype>::LayerSetUp(const vector<Blob<Dtype>*>& bottom,
      const vector<Blob<Dtype>*>& top) {
  // 读取层的参数
  BatchNormParameter param = this->layer_param_.batch_norm_param();
  // 获取滑动平均衰减系数
  moving_average_fraction_ = param.moving_average_fraction();
  // 对测试阶段 use_global_stats_ 默认为真
  use_global_stats_ = this->phase_ == TEST;
  if (param.has_use_global_stats())
    // 如果有 use_global_states 参数，则将修改 use_global_stats_
    use_global_stats_ = param.use_global_stats();
  if (bottom[0]->num_axes() == 1)
    // 如果 bottom blob 的轴数为 1，那么设置 channels_ 为 1
    channels_ = 1;
  else
    // 否则就正常读取 channels_ 值
    channels_ = bottom[0]->shape(1);
  // 读取分母附加项
  eps_ = param.eps();
  if (this->blobs_.size() > 0) {
    LOG(INFO) << "Skipping parameter initialization";
  } else {
    // 设置内部参数 blobs_ 的尺寸大小
    this->blobs_.resize(3);
    // 针对每个 channel 分别存储均值、方差和滑动平均衰减系数
    vector<int> sz;
    sz.push_back(channels_);
    // 设置均值的尺寸 (channels, )
    this->blobs_[0].reset(new Blob<Dtype>(sz));
    // 设置方差的尺寸 (channels, )
    this->blobs_[1].reset(new Blob<Dtype>(sz));
    sz[0] = 1;
    // 设置滑动平均衰减系数的尺寸 (1, )
    this->blobs_[2].reset(new Blob<Dtype>(sz));
    // 将该层的内部参数全部设置为 0
    for (int i = 0; i < 3; ++i) {
      caffe_set(this->blobs_[i]->count(), Dtype(0),
                this->blobs_[i]->mutable_cpu_data());
    }
  }
  // Mask statistics from optimization by setting local learning rates
  // for mean, variance, and the bias correction to zero.
  // 设置优化均值、方差和偏置的本地学习率
  for (int i = 0; i < this->blobs_.size(); ++i) {
    if (this->layer_param_.param_size() == i) {
```

```
      // 如果该层的 ParamSpec 参数数目不够，则添加参数，并且设置学习动量为 0
      ParamSpec* fixed_param_spec = this->layer_param_.add_param();
      fixed_param_spec->set_lr_mult(0.f);
    } else {
      CHECK_EQ(this->layer_param_.param(i).lr_mult(), 0.f)
          << "Cannot configure batch normalization statistics as layer "
          << "parameters.";
    }
  }
}
```

2.4、Forward_cpu()

前向传播函数主要完成数据的归一化，即：

$$y = \frac{x - E[x]}{\sqrt{Var[x]}} \tag{3}$$

根据 `this->use_global_stats_` 的真假，我们会去判断是否使用 global 均值和 global 方差；如果为真，我们会分别将 `this->mean_` 和 `this->variance_` 设置为 global 均值和 global 方差；

```
// 如果 use_global_stats_ 为真，那么我们使用全局的均值和方差
// use the stored mean/variance estimates.
// 如果滑动平均系数为 0，设置 scale_factor 为 0，否则设置 scale_factor 为滑动平均系数的倒数
const Dtype scale_factor = this->blobs_[2]->cpu_data()[0] == 0 ?
    0 : 1 / this->blobs_[2]->cpu_data()[0];
// 设置局部的均值
caffe_cpu_scale(this->variance_.count(), scale_factor,
    this->blobs_[0]->cpu_data(), this->mean_.mutable_cpu_data());
// 设置局部方差
caffe_cpu_scale(this->variance_.count(), scale_factor,
    this->blobs_[1]->cpu_data(), this->variance_.mutable_cpu_data());
```

如果为假，我们会根据 mini-batch 的数据计算均值和方差。

```
// compute mean

/*
** 函数：caffe_cpu_gemv<Dtype>(const CBLAS_TRANSPOSE TransA, const int M,
                        const int N, const Dtype alpha, const Dtype* A,
                        const Dtype* x, const Dtype beta, Dtype* y)
** 功能：y = alpha * A * x + beta * y
** 其中 x 和 y 是向量，A 是矩阵
** M：A 的行数
** N：A 的列数
*/
// 数学表达式：num_by_chans_ = 1. / (num * spatial_dim) * bottom_data * spatial_sum_multiplier_
// bottom_data 是 (channels_ * num, spatial_dim)
// spatial_sum_multiplier_ 是 (spatial_dim, 1)，元素全为 1
// num_by_chans_ 是 (channels_ * num, 1)
caffe_cpu_gemv<Dtype>(CblasNoTrans, this->channels_ * num, spatial_dim,
    1. / (num * spatial_dim), bottom_data,
    this->spatial_sum_multiplier_.cpu_data(), 0.,
    this->num_by_chans_.mutable_cpu_data());
// 数学表达式：mean_ = Trans(num_by_chans) * batch_sum_multiplier
// num_by_chans_ 是 (num, channels_)
// batch_sum_multiplier_ 是 (num, 1)
// mean_ 是 (channels_, 1)
// 最终得到每个 channel 的平均值
caffe_cpu_gemv<Dtype>(CblasTrans, num, this->channels_, 1.,
    this->num_by_chans_.cpu_data(), this->batch_sum_multiplier_.cpu_data(), 0.,
    this->mean_.mutable_cpu_data());
```

与论文计算 global 均值和 global 方差的方式不同之处在于，Caffe 中的 global 均值和 global 方差采用的是滑动平均的更新方式，因此，BN 层的 `this->blobs[0]`、`this->blobs[1]` 和 `this->blobs[2]` 分别会存储均值滑动和、方差滑动和以及滑动系数和。

我们假设滑动衰减系数 `this->moving_average_fraction_` 为 λ，$m = bottom[0]->count()/channels\_$，存储均值滑动和、方差滑动和以及滑动系数和分别为 $\mu_{old}, \sigma^2_{old}, s_{old}$，且当前的 mini-batch 的均值和方差分别为 μ_B, σ^2_B：

$$\begin{aligned} s_{new} &= \lambda s_{old} + 1 \\ \mu_{new} &= \lambda \mu_{old} + \mu_B \end{aligned}$$

对于方差，采用的是无偏估计：

$$\sigma^2_{new} = \begin{cases} \lambda \sigma^2_{old} + \frac{m-1}{m}\sigma^2_B & m > 1 \\ \lambda \sigma^2_{old} & m = 1 \end{cases}$$

文档导航

```
// compute and save moving average
this->blobs_[2]->mutable_cpu_data()[0] *= this->moving_average_fraction_;
this->blobs_[2]->mutable_cpu_data()[0] += 1;
/*
** caffe_cpu_axpby<Dtype>(const int N, const Dtype alpha, const Dtype* X,
                              const Dtype beta, Dtype* Y)
** 功能：Y= alpha * X + beta * Y
** X 是 (N, 1)
** Y 是 (N, 1)
*/

// 数学表达式：blobs_[0] = mean_ + moving_average_fraction_ * blobs_[0]
caffe_cpu_axpby(this->mean_.count(), Dtype(1), this->mean_.cpu_data(),
    this->moving_average_fraction_, this->blobs_[0]->mutable_cpu_data());
int m = bottom[0]->count()/channels_;
// 计算无偏差估计的系数 m/(m - 1)
Dtype bias_correction_factor = m > 1 ? Dtype(m)/(m-1) : 1;
// 数学表达式：blobs_[1] = bias_correction_factor * variance_ + moving_average_fraction_ * blobs_[1]
caffe_cpu_axpby(variance_.count(), bias_correction_factor,
    variance_.cpu_data(), moving_average_fraction_,
    this->blobs_[1]->mutable_cpu_data());
```

源代码注释如下所示：

```
template <typename Dtype>
void BatchNormLayer<Dtype>::Forward_cpu(const vector<Blob<Dtype>*>& bottom,
    const vector<Blob<Dtype>*>& top) {
  // 获取只读 bottom_data 的指针
  const Dtype* bottom_data = bottom[0]->cpu_data();
  // 获取读写 top_data 的指针
  Dtype* top_data = top[0]->mutable_cpu_data();
  // 获取 batch_size 的大小
  int num = bottom[0]->shape(0);
  int spatial_dim = bottom[0]->count()/(bottom[0]->shape(0) * this->channels_);

  // 如果输入 bottom 与输出 bottom 的地址不一致，
  // 则我们需要将 bottom_data 的数据拷贝到 top_data
  if (bottom[0] != top[0]) {
    caffe_copy(bottom[0]->count(), bottom_data, top_data);
  }

  if (this->use_global_stats_) {
    // 如果 use_global_stats_ 为真，那么我们使用全局的均值和方差
    // use the stored mean/variance estimates.
    // 如果滑动平均系数为 0，设置 scale_factor 为 0，否则设置 scale_factor 为滑动平均系数的倒数
    const Dtype scale_factor = this->blobs_[2]->cpu_data()[0] == 0 ?
        0 : 1 / this->blobs_[2]->cpu_data()[0];
    // 设置局部的均值
    caffe_cpu_scale(this->variance_.count(), scale_factor,
        this->blobs_[0]->cpu_data(), this->mean_.mutable_cpu_data());
    // 设置局部方差
    caffe_cpu_scale(this->variance_.count(), scale_factor,
        this->blobs_[1]->cpu_data(), this->variance_.mutable_cpu_data());
  } else {
    // compute mean

    /*
    ** 函数：caffe_cpu_gemv<Dtype>(const CBLAS_TRANSPOSE TransA, const int M,
                                   const int N, const Dtype alpha, const Dtype* A,
                                   const Dtype* x, const Dtype beta, Dtype* y)
    ** 功能：y = alpha * A * x + beta * y
    ** 其中 x 和 y 是向量，A 是矩阵
    ** M：A 的行数
    ** N：A 的列数
    */
```

```
    // 数学表达式：num_by_chans_ = 1. / (num * spatial_dim) * bottom_data * spatial_sum_multiplier_
    // bottom_data 是 (channels_ * num, spatial_dim)
    // spatial_sum_multiplier_ 是 (spatial_dim, 1)，元素全为 1
    // num_by_chans_ 是 (channels_ * num, 1)
    caffe_cpu_gemv<Dtype>(CblasNoTrans, this->channels_ * num, spatial_dim,
        1. / (num * spatial_dim), bottom_data,
        this->spatial_sum_multiplier_.cpu_data(), 0.,
        this->num_by_chans_.mutable_cpu_data());
    // 数学表达式：mean_ = Trans(num_by_chans) * batch_sum_multiplier
    // num_by_chans_ 是 (num, channels_)
    // batch_sum_multiplier_ 是 (num, 1)
    // mean_ 是 (channels_, 1)
    // 最终得到每个 channel 的平均值
    caffe_cpu_gemv<Dtype>(CblasTrans, num, this->channels_, 1.,
        this->num_by_chans_.cpu_data(), this->batch_sum_multiplier_.cpu_data(), 0.,
        this->mean_.mutable_cpu_data());
  }

  // subtract mean

  /*
    ** 函数：caffe_cpu_gemm<Dtype>(const CBLAS_TRANSPOSE TransA, const CBLAS_TRANSPOSE TransB,
                                 const int M, const int N, const int K, const Dtype alpha,
                                 const Dtype* A, const Dtype* B, const Dtype beta, Dtype* C)
    ** 功能：C = alpha * A * B + beta * C
    ** 其中 A 是 (M, K)；B 是 (K, N)；C 是 (M, N)
    */
  // 数学表达式：num_by_chans_ = batch_sum_multiplier_ * mean_
  // batch_sum_multiplier_ 是 (num, 1)
  // mean_ 是 (1, channels_)
  // num_by_chans_ 是 (num, channels_)
  caffe_cpu_gemm<Dtype>(CblasNoTrans, CblasNoTrans, num, this->channels_, 1, 1,
      this->batch_sum_multiplier_.cpu_data(), this->mean_.cpu_data(), 0.,
      this->num_by_chans_.mutable_cpu_data());
  // 数学表达式：top_data = -1 * num_by_chans_ * spatial_sum_multiplier_ + top_data
  // num_by_chans_ 是 (channels_ * num, 1)
  // spatial_sum_multiplier_ 是 (1, spatial_dim)
  // top_data 是 (channels_ * num, spatial_dim)
  // 为每一个像素点减去均值
```

```
  caffe_cpu_gemm<Dtype>(CblasNoTrans, CblasNoTrans, this->channels_ * num,
      spatial_dim, 1, -1, this->num_by_chans_.cpu_data(),
      this->spatial_sum_multiplier_.cpu_data(), 1., top_data);

  // 如果 use_global_stats_ 为真
  if (!this->use_global_stats_) {
    // compute variance using var(X) = E((X-EX)^2)
    // caffe_powx 是 element-wise 操作，这里实现对每个元素平方
    caffe_powx(top[0]->count(), top_data, Dtype(2),
        this->temp_.mutable_cpu_data());  // (X-EX)^2

    // 数学表达式：num_by_chans_ = 1. / (num * spatial_dim) * temp_ * spatial_sum_multiplier_
    // temp_ 是 (channels_ * num, spatial_dim)
    // spatial_sum_multiplier_ 是 (spatial_dim, 1)，元素全为 1
    // num_by_chans_ 是 (channels_ * num, 1)
    caffe_cpu_gemv<Dtype>(CblasNoTrans, this->channels_ * num, spatial_dim,
        1. / (num * spatial_dim), this->temp_.cpu_data(),
        this->spatial_sum_multiplier_.cpu_data(), 0.,
        this->num_by_chans_.mutable_cpu_data());

    // 数学表达式：variance_ = Trans(num_by_chans_) * batch_sum_multiplier_
    // num_by_chans 是 (num, channels_)
    // batch_sum_multiplier_ 是 (num, 1)，元素全为 1
    // varaince_ 是 (channels_, 1)
    // 计算出方差
    caffe_cpu_gemv<Dtype>(CblasTrans, num, this->channels_, 1.,
        this->num_by_chans_.cpu_data(), this->batch_sum_multiplier_.cpu_data(), 0.,
        this->variance_.mutable_cpu_data());  // E((X_EX)^2)

    // compute and save moving average
    this->blobs_[2]->mutable_cpu_data()[0] *= this->moving_average_fraction_;
    this->blobs_[2]->mutable_cpu_data()[0] += 1;
    /*
    ** caffe_cpu_axpby<Dtype>(const int N, const Dtype alpha, const Dtype* X,
                                   const Dtype beta, Dtype* Y)
    ** 功能：Y= alpha * X + beta * Y
    ** X 是 (N, 1)
    ** Y 是 (N, 1)
    */

    // 数学表达式：blobs_[0] = mean_ + moving_average_fraction_ * blobs_[0]
```

```
    caffe_cpu_axpby(this->mean_.count(), Dtype(1), this->mean_.cpu_data(),
        this->moving_average_fraction_, this->blobs_[0]->mutable_cpu_data());
    int m = bottom[0]->count()/channels_;
    // 计算无偏差估计的系数 m/(m - 1)
    Dtype bias_correction_factor = m > 1 ? Dtype(m)/(m-1) : 1;
    // 数学表达式：blobs_[1] = bias_correction_factor * variance_ + moving_average_fraction_ * blobs_[1]
    caffe_cpu_axpby(variance_.count(), bias_correction_factor,
        variance_.cpu_data(), moving_average_fraction_,
        this->blobs_[1]->mutable_cpu_data());
  }

  // normalize variance
  /*
  ** caffe_add_scalar(const int N, const float alpha, float* Y)
  ** 功能：给 Y 的每个元素加上 alpha
  */
  // 为方差加上一个附加值，防止除数为 0
  caffe_add_scalar(this->variance_.count(), this->eps_, this->variance_.mutable_cpu_data());
  // 为每个元素求开平方
  caffe_powx(this->variance_.count(), this->variance_.cpu_data(), Dtype(0.5),
             this->variance_.mutable_cpu_data());

  // replicate variance to input size
  // 数学表达式：num_by_chans_ = batch_sum_multiplier_ * variance_
  // batch_sum_multiplier_ 是 (num, 1)，元素全为 1
  // variance_ 是 (1, channels)
  // num_by_chans 是 (num, channels)
  caffe_cpu_gemm<Dtype>(CblasNoTrans, CblasNoTrans, num, this->channels_, 1, 1,
      this->batch_sum_multiplier_.cpu_data(), this->variance_.cpu_data(), 0.,
      this->num_by_chans_.mutable_cpu_data());

  // 数学表达式：temp_ = num_by_chans_ * spatial_sum_multiplier
  // num_by_chans_ 是 (channels * num, 1)
  // spatial_sum_multiplier_ 是 (1, spatial_dim)，元素全为 1
  // temp_ 是 (channels * num, spatial_dim)
  caffe_cpu_gemm<Dtype>(CblasNoTrans, CblasNoTrans, this->channels_ * num,
      spatial_dim, 1, 1., this->num_by_chans_.cpu_data(),
      this->spatial_sum_multiplier_.cpu_data(), 0., this->temp_.mutable_cpu_data());
  // 实现 element-wise 相除
  caffe_div(this->temp_.count(), top_data, this->temp_.cpu_data(), top_data);
```

```
  // TODO(cdoersch): The caching is only needed because later in-place layers
  //                 might clobber the data.  Can we skip this if they won't?
  caffe_copy(this->x_norm_.count(), top_data, this->x_norm_.mutable_cpu_data());
}
```

2.5、Backward_cpu()

与论文中的 BN 算法不同的是，Caffe 的 Batch Normalization Layer 只完成了数据的归一化部分，没有实现数据的 Scale 和 Shift 操作。还有 Caffe 使用的反向传播数学表达式与论文中有点不同，所以我们将会在这里简单地推导一下反向传播公式。假设我们只考察像素点 x_i，其中 $i \in [1, n \times h \times w]$，$n, h, w$ 分别表示 batch_size、height 和 width。我们设 $m = n \times h \times w$，有:

- mini-batch 均值 μ_B

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{4}$$

- min-batch 方差 σ_B^2

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2 \tag{5}$$

- min-batch 归一化的数据 y_i

$$y_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}} \tag{6}$$

- mini-batch 损失值 L 关于 x_i 的梯度

$$\frac{\partial L}{\partial x_i} = \sum_{j=1}^{m} \frac{\partial L}{\partial y_j} \times \frac{\partial y_j}{\partial x_i} \tag{7}$$

(1) 当 $i = j$ 时，有:

$$\begin{aligned} \frac{\partial y_i}{\partial x_i} &= \frac{1}{\sqrt{\sigma_B^2+\varepsilon}} - \frac{1}{\sqrt{\sigma_B^2+\varepsilon}} \times \frac{1}{m} - \frac{x_i-\mu_B}{2} \times (\sigma_B^2 + \varepsilon)^{-\frac{3}{2}} \times \frac{2}{m} \times (x_i - \mu_B) \\ &= \frac{1}{\sqrt{\sigma_B^2+\varepsilon}}\left[1 - \frac{1}{m} - \frac{y_i^2}{m}\right] \end{aligned} \tag{8}$$

(2) 当 $i \neq j$ 时，有:

$$\begin{aligned} \frac{\partial y_j}{\partial x_i} &= -\frac{1}{\sqrt{\sigma_B^2+\varepsilon}} \times \frac{1}{m} - \frac{x_i-\mu_B}{2} \times (\sigma_B^2 + \varepsilon)^{-\frac{3}{2}} \times \frac{2}{m} \times (x_j - \mu_B) \\ &= -\frac{1}{\sqrt{\sigma_B^2+\varepsilon}}\left[\frac{1}{m} + \frac{y_i y_j}{m}\right] \end{aligned} \tag{9}$$

最终，我们可以求得:

$$\frac{\partial L}{\partial x_i} = \frac{1}{\sqrt{\sigma_B^2 + \varepsilon}}\left[\frac{\partial L}{\partial y_i} - \frac{1}{m}\sum_{j=1}^{m}\frac{\partial L}{\partial y_j} - \frac{y_i}{m}\sum_{j=1}^{m} y_j \frac{\partial L}{\partial y_j}\right] \tag{10}$$

源代码注释如下所示:

```cpp
template <typename Dtype>
void BatchNormLayer<Dtype>::Backward_cpu(const vector<Blob<Dtype>*>& top,
    const vector<bool>& propagate_down,
    const vector<Blob<Dtype>*>& bottom) {
  // 获取只读 top_diff 指针
  const Dtype* top_diff;
  if (bottom[0] != top[0]) { // 如果 bottom[0] 与 top[0] 所指的地址不一致
    top_diff = top[0]->cpu_diff();
  } else {
    caffe_copy(this->x_norm_.count(), top[0]->cpu_diff(), this->x_norm_.mutable_cpu_diff());
    top_diff = this->x_norm_.cpu_diff();
  }
  // 获取读写 bottom_diff 的指针
  Dtype* bottom_diff = bottom[0]->mutable_cpu_diff();
  // 如果 use_global_stats_ 为真
  if (this->use_global_stats_) {
    caffe_div(this->temp_.count(), top_diff, this->temp_.cpu_data(), bottom_diff);
    return;
  }
  // 获取只读 x_norm_ 指针
  const Dtype* top_data = this->x_norm_.cpu_data();
  // 获取 batch_size 的大小
  int num = bottom[0]->shape()[0];
  // 获取 spatial_dim 的大小
  int spatial_dim = bottom[0]->count()/(bottom[0]->shape(0) * this->channels_);
  // if Y = (X-mean(X))/(sqrt(var(X)+eps)), then
  //
  // dE(Y)/dX =
  //   (dE/dY - mean(dE/dY) - mean(dE/dY \cdot Y) \cdot Y)
  //     ./ sqrt(var(X) + eps)
  //
  // where \cdot and ./ are hadamard product and elementwise division,
  // respectively, dE/dY is the top diff, and mean/var/sum are all computed
  // along all dimensions except the channels dimension.  In the above
  // equation, the operations allow for expansion (i.e. broadcast) along all
  // dimensions except the channels dimension where required.

  // sum(dE/dY \cdot Y)

  // 实现 element wise 相乘
  // bottom_diff[i] = top_data[i] * top_diff[i]
```

```
  caffe_mul(this->temp_.count(), top_data, top_diff, bottom_diff);

  // 数学表达式：num_by_chans_ = bottom_diff * spatial_sum_multiplier_
  // bottom_diff 是 (channels_ * num, spatial_dim)
  // spatial_sum_multiplier_ 是 (spatial_dim, 1)
  // num_by_chans 是 (channels_ * num, 1)
  caffe_cpu_gemv<Dtype>(CblasNoTrans, this->channels_ * num, spatial_dim, 1.,
      bottom_diff, this->spatial_sum_multiplier_.cpu_data(), 0.,
      this->num_by_chans_.mutable_cpu_data());

  // 数学表达式：mean_ = Trans(num_by_chans_) * batch_sum_multiplier_
  // num_by_chans_ 是 (num, channels)
  // batch_sum_multiplier_ 是 (num, 1)，元素全是 1
  // mean_ 是 (channels, 1)
  caffe_cpu_gemv<Dtype>(CblasTrans, num, this->channels_, 1.,
      this->num_by_chans_.cpu_data(), this->batch_sum_multiplier_.cpu_data(), 0.,
      this->mean_.mutable_cpu_data());

  // reshape (broadcast) the above
  // 数学表达式：num_by_chans_ = batch_sum_multiplier_ * mean_
  // batch_sum_multiplier_ 是 (num, 1)，元素全是 1
  // mean_ 是 (1, channels_)
  // num_by_chans_ 是 (num, channels_)
  caffe_cpu_gemm<Dtype>(CblasNoTrans, CblasNoTrans, num, this->channels_, 1, 1,
      this->batch_sum_multiplier_.cpu_data(), this->mean_.cpu_data(), 0.,
      this->num_by_chans_.mutable_cpu_data());

  // 数学表达式：bottom_diff = num_by_chans_ * saptial_sum_multiplier_
  // num_by_chans_ 是 (channels * num, 1)
  // spatial_sum_multiplier_ 是 (1, spatial_dim)，所有元素为 1
  // bottom_diff 是 (channels * num, spatial_dim)
  caffe_cpu_gemm<Dtype>(CblasNoTrans, CblasNoTrans, this->channels_ * num,
      spatial_dim, 1, 1., this->num_by_chans_.cpu_data(),
      this->spatial_sum_multiplier_.cpu_data(), 0., bottom_diff);

  // sum(dE/dY \cdot Y) \cdot Y
  // 采用 element-wise 的乘法
  // bottom_diff[i] = top_data[i] * bottom_diff[i]
  caffe_mul(this->temp_.count(), top_data, bottom_diff, bottom_diff);

  // sum(dE/dY)-sum(dE/dY \cdot Y) \cdot Y
```

```
  // 数学表达式：num_by_chans_ = top_diff * spatial_sum_multiplier_
  // top_diff 是 (channels_ * num, spatial_dim)
  // spatial_sum_multiplier_ 是 (spatial_dim, 1)，所有元素为 1
  // num_by_chans_ 是 (channels_ * num, 1)
  caffe_cpu_gemv<Dtype>(CblasNoTrans, this->channels_ * num, spatial_dim, 1.,
      top_diff, this->spatial_sum_multiplier_.cpu_data(), 0.,
      this->num_by_chans_.mutable_cpu_data());

  // 数学表达式：mean_ = Trans(num_by_chans_) * batch_sum_multiplier_
  // num_by_chans 是 (num, channels_)
  // batch_sum_multiplier_ 是 (num, 1)，所有元素为 1
  // mean_ 是 (channels_, 1)
  caffe_cpu_gemv<Dtype>(CblasTrans, num, this->channels_, 1.,
      this->num_by_chans_.cpu_data(), this->batch_sum_multiplier_.cpu_data(), 0.,
      this->mean_.mutable_cpu_data());

  // reshape (broadcast) the above to make
  // sum(dE/dY)-sum(dE/dY \cdot Y) \cdot Y
  // 数学表达式：num_by_chans_ = batch_sum_multiplier_ * mean_
  // batch_sum_multiplier_ 是 (num, 1)，元素全为 1
  // mean_ 是 (1, channels)
  // num_by_chans_ 是 (num, channels)
  caffe_cpu_gemm<Dtype>(CblasNoTrans, CblasNoTrans, num, this->channels_, 1, 1,
      this->batch_sum_multiplier_.cpu_data(), this->mean_.cpu_data(), 0.,
      this->num_by_chans_.mutable_cpu_data());

  // 数学表达式：bottom_diff = num_by_chans_ * spatial_sum_multiplier_
  // num_by_chans_ 是 (num * channels_, 1)
  // spatial_sum_multiplier_ 是 (1, spatial_dim)，元素全为 1
  // bottom_diff 是 (num * channels_, spatial_dim)
  caffe_cpu_gemm<Dtype>(CblasNoTrans, CblasNoTrans, num * channels_,
      spatial_dim, 1, 1., num_by_chans_.cpu_data(),
      spatial_sum_multiplier_.cpu_data(), 1., bottom_diff);

  // dE/dY - mean(dE/dY)-mean(dE/dY \cdot Y) \cdot Y

  /*
    ** caffe_cpu_axpby<Dtype>(const int N, const Dtype alpha, const Dtype* X,
                                    const Dtype beta, Dtype* Y)
    ** 功能：Y= alpha * X + beta * Y
    ** X 是 (N, 1)
    ** Y 是 (N, 1)
```

```
    */
  // 数学表达式: bottom_diff = top_diff - 1 / (num * spatial_dim) * bottom_diff
  caffe_cpu_axpby(temp_.count(), Dtype(1), top_diff,
      Dtype(-1. / (num * spatial_dim)), bottom_diff);

  // note: temp_ still contains sqrt(var(X)+eps), computed during the forward
  // pass.
  // 实现 element-wise 元素相除
  caffe_div(temp_.count(), bottom_diff, temp_.cpu_data(), bottom_diff);
}
```


上一篇: Ubuntu16.04 LTS 如何配置 NFS (http://blog.leanote.com/post/braveapple/Ubuntu-%E5%A6%82%E4%BD%95%E9%85%8D%E7%BD%AE-NFS)

下一篇: 解析 Caffe 之 Hinge Loss (http://blog.leanote.com/post/braveapple/Hinge-Loss-%E7%9A%84%E7%90%86%E8%A7%A3)

导航

主页 (http://blog.leanote.com/braveapple)
About Me (http://blog.leanote.com/single/braveapple/About-Me)
归档 (http://blog.leanote.com/archives/braveapple)
标签 (http://blog.leanote.com/tags/braveapple)

最近发表

【CVPR2018】Unsupervised Cross-dataset Person Re-identification by Transfer Learning of Spatial-Temporal Patterns (http://blog.leanote.com/post/braveapple/Unsupervised-Cross-dataset-Person-Re-identification-by-Transfer-Learning-of-Spatial-Temporal-Patterns)
Linux的解压命令 (http://blog.leanote.com/post/braveapple/Linux%E7%9A%84%E8%A7%A3%E5%8E%8B%E5%91%BD%E4%BB%A4)
Linux 挂载硬盘 (http://blog.leanote.com/post/braveapple/Linux-%E6%8C%82%E8%BD%BD%E7%A1%AC%E7%9B%98)
Linux 安装配置 tmux (http://blog.leanote.com/post/braveapple/Linux-%E5%AE%89%E8%A3%85%E9%85%8D%E7%BD%AE-tmux)
定时任务 crontab 使用技巧 (http://blog.leanote.com/post/braveapple/%E5%AE%9A%E6%97%B6%E4%BB%BB%E5%8A%A1-crontab-%E4%BD%BF%E7%94%A8%E6%8A%80%E5%B7%A7)

友情链接

My Note (https://leanote.com/note)
Leanote Home (https://leanote.com)
Leanote BBS (http://bbs.leanote.com)
Leanote Github (https://github.com/leanote/leanote)