



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

coding will change world!

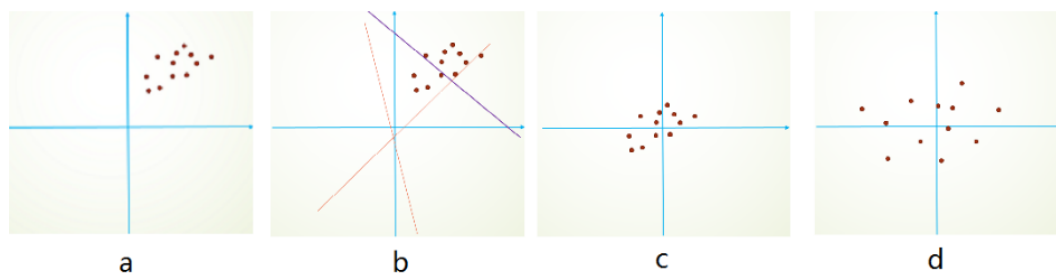
解析 Caffe 之 Batch Normalization Layer ≡ 文档导航

© 2017-04-10 20:56:54 1678 0 0

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 $y = wx + b$ 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。

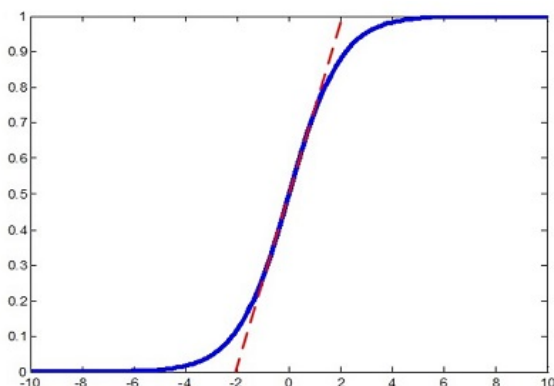
在论文 **Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift** (<https://arxiv.org/pdf/1502.03167.pdf>) 中，作者首先考虑对每一层输出数据进行白化操作，但是分析这种方法是不可行的。因为数据白化需要计算协方差矩阵、求逆等操作，计算量很大，而且在反向传播过程中，白化操作不一定可导。

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 $x = (x^{(1)}, x^{(2)}, \dots, x^{(d)})$ ，对数据做简单的归一化操作，使得数据均值为 0，方差为 1：

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

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。

文档导航

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

Batch Normalization 的算法流程如下：

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots x_m\}$;	
Parameters to be learned: γ, β	
Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$	
$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i$	// mini-batch mean
$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2$	// mini-batch variance
$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$	// normalize
$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i)$	// scale and shift

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

$$\begin{aligned} \frac{\partial \ell}{\partial \hat{x}_i} &= \frac{\partial \ell}{\partial y_i} \cdot \gamma \\ \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} &= \sum_{i=1}^m \frac{\partial \ell}{\partial \hat{x}_i} \cdot (x_i - \mu_{\mathcal{B}}) \cdot \frac{-1}{2} (\sigma_{\mathcal{B}}^2 + \epsilon)^{-3/2} \\ \frac{\partial \ell}{\partial \mu_{\mathcal{B}}} &= \left(\sum_{i=1}^m \frac{\partial \ell}{\partial \hat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \right) + \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{\sum_{i=1}^m -2(x_i - \mu_{\mathcal{B}})}{m} \\ \frac{\partial \ell}{\partial x_i} &= \frac{\partial \ell}{\partial \hat{x}_i} \cdot \frac{1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} + \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{2(x_i - \mu_{\mathcal{B}})}{m} + \frac{\partial \ell}{\partial \mu_{\mathcal{B}}} \cdot \frac{1}{m} \\ \frac{\partial \ell}{\partial \gamma} &= \sum_{i=1}^m \frac{\partial \ell}{\partial y_i} \cdot \hat{x}_i \\ \frac{\partial \ell}{\partial \beta} &= \sum_{i=1}^m \frac{\partial \ell}{\partial y_i} \end{aligned}$$

2、源码分析

2.1、BatchNormParameter

```

1. message BatchNormParameter {
2.     // If false, accumulate global mean/variance values via a moving
    average. If
3.     // true, use those accumulated values instead of computing mean/
    variance
4.     // across the batch.
5.     // 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
6.     // 和方差，而不是使用每个 batch 的均值和方差
7.     optional bool use_global_stats = 1;
8.     // How much does the moving average decay each iteration?
9.     // 滑动平均的衰减系数，默认为 0.999
10.    optional float moving_average_fraction = 2 [default = .999];
11.    // Small value to add to the variance estimate so that we don't
    divide by
12.    // zero.
13.    // 分母的附加值，防止除 0 的情况，默认值为 1e-5
14.    optional float eps = 3 [default = 1e-5];
15. }

```

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。

```

1. // 均值、方差、中间值、归一化值
2.    Blob<Dtype> mean_, variance_, temp_, x_norm_;
3.    // 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
4.    // 和方差，而不是使用每个 batch 的均值和方差
5.    bool use_global_stats_;
6.    // 滑动平均的衰减系数，默认为 0.999
7.    Dtype moving_average_fraction_;
8.    int channels_;
9.    // 分母的附加值，防止除 0 的情况，默认值为 1e-5
10.    Dtype eps_;
11.
12.    // extra temporary variables is used to carry out sums/broadca
    sting
13.    // using BLAS
14.    // 维度是 (batch_size, )
15.    Blob<Dtype> batch_sum_multiplier_;
16.    // 维度是 (batch_size, channels)
17.    Blob<Dtype> num_by_chans_;
18.    // 维度是 (height, width)
19.    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。

```

1. // 层设置函数
2. template <typename Dtype>
3. void BatchNormLayer<Dtype>::LayerSetUp(const vector<Blob<Dtype>*>&
    bottom,
4.     const vector<Blob<Dtype>*>& top) {
5.     // 读取层的参数
6.     BatchNormParameter param = this->layer_param_.batch_norm_param
    ();
7.     // 获取滑动平均衰减系数
8.     moving_average_fraction_ = param.moving_average_fraction();
9.     // 对测试阶段 use_global_stats_ 默认为真
10.    use_global_stats_ = this->phase_ == TEST;
11.    if (param.has_use_global_stats())
12.        // 如果有 use_global_stats 参数, 则将修改 use_global_stats_
13.        use_global_stats_ = param.use_global_stats();
14.    if (bottom[0]->num_axes() == 1)
15.        // 如果 bottom blob 的轴数为 1, 那么设置 channels_ 为 1
16.        channels_ = 1;
17.    else
18.        // 否则就正常读取 channels_ 值
19.        channels_ = bottom[0]->shape(1);
20.    // 读取分母附加项
21.    eps_ = param.eps();
22.    if (this->blobs_.size() > 0) {
23.        LOG(INFO) << "Skipping parameter initialization";
24.    } else {
25.        // 设置内部参数 blobs_ 的尺寸大小
26.        this->blobs_.resize(3);
27.        // 针对每个 channel 分别存储均值、方差和滑动平均衰减系数
28.        vector<int> sz;
29.        sz.push_back(channels_);
30.        // 设置均值的尺寸 (channels, )
31.        this->blobs_[0].reset(new Blob<Dtype>(sz));
32.        // 设置方差的尺寸 (channels, )
33.        this->blobs_[1].reset(new Blob<Dtype>(sz));
34.        sz[0] = 1;
35.        // 设置滑动平均衰减系数的尺寸 (1, )
36.        this->blobs_[2].reset(new Blob<Dtype>(sz));
37.        // 将该层的内部参数全部设置为 0
38.        for (int i = 0; i < 3; ++i) {
39.            caffe_set(this->blobs_[i]->count(), Dtype(0),
40.                this->blobs_[i]->mutable_cpu_data());
41.        }
42.    }
43.    // Mask statistics from optimization by setting local learning r
    ates
44.    // for mean, variance, and the bias correction to zero.
45.    // 设置优化均值、方差和偏置的本地学习率
46.    for (int i = 0; i < this->blobs_.size(); ++i) {
47.        if (this->layer_param_.param_size() == i) {

```

```

48.          // 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
49.          ParamSpec* fixed_param_spec = this->layer_param_.add_param
           ();
50.          fixed_param_spec->set_lr_mult(0.f);
51.      } else {
52.          CHECK_EQ(this->layer_param_.param(i).lr_mult(), 0.f)
53.              << "Cannot configure batch normalization statistics as l
           ayer "
54.              << "parameters.";
55.      }
56.  }
57. }

```

2.4、Forward_cpu()

前向传播函数主要完成数据的归一化，即：

$$y = \frac{x - E[x]}{\sqrt{Var[x]}} \quad (3)$$

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；

```

1. // 如果 use_global_stats_ 为真，那么我们使用全局的均值和方差
2. // use the stored mean/variance estimates.
3. // 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
4. const Dtype scale_factor = this->blobs_[2]->cpu_data()[0] == 0 ?
5.     0 : 1 / this->blobs_[2]->cpu_data()[0];
6. // 设置局部的均值
7. caffe_cpu_scale(this->variance_.count(), scale_factor,
8.     this->blobs_[0]->cpu_data(), this->mean_.mutable_cpu_data());
9. // 设置局部方差
10. caffe_cpu_scale(this->variance_.count(), scale_factor,
11.     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。

```

1. // compute mean
2.
3. /*
4. ** 函数: caffe_cpu_gemv<Dtype>(const CBLAS_TRANSPOSE TransA, const
      int M,
5.                                const int N, const Dtype alpha, const Dtype
      e* A,
6.                                const Dtype* x, const Dtype beta, Dtype*
      y)
7. ** 功能:  $y = \alpha * A * x + \beta * y$ 
8. ** 其中  $x$  和  $y$  是向量,  $A$  是矩阵
9. **  $M$ :  $A$  的行数
10. **  $N$ :  $A$  的列数
11. */
12. // 数学表达式:  $\text{num\_by\_chans\_} = 1. / (\text{num} * \text{spatial\_dim}) * \text{bottom\_data} * \text{spatial\_sum\_multiplier\_}$ 
13. //  $\text{bottom\_data}$  是  $(\text{channels\_} * \text{num}, \text{spatial\_dim})$ 
14. //  $\text{spatial\_sum\_multiplier\_}$  是  $(\text{spatial\_dim}, 1)$ , 元素全为 1
15. //  $\text{num\_by\_chans\_}$  是  $(\text{channels\_} * \text{num}, 1)$ 
16. caffe_cpu_gemv<Dtype>(CblasNoTrans, this->channels_ * num, spatial_
    _dim,
17.    1. / (num * spatial_dim), bottom_data,
18.    this->spatial_sum_multiplier_.cpu_data(), 0.,
19.    this->num_by_chans_.mutable_cpu_data());
20. // 数学表达式:  $\text{mean\_} = \text{Trans}(\text{num\_by\_chans\_}) * \text{batch\_sum\_multiplier\_}$ 
21. //  $\text{num\_by\_chans\_}$  是  $(\text{num}, \text{channels\_})$ 
22. //  $\text{batch\_sum\_multiplier\_}$  是  $(\text{num}, 1)$ 
23. //  $\text{mean\_}$  是  $(\text{channels\_}, 1)$ 
24. // 最终得到每个 channel 的平均值
25. caffe_cpu_gemv<Dtype>(CblasTrans, num, this->channels_, 1.,
26.    this->num_by_chans_.cpu_data(), this->batch_sum_multiplier_.cp
    u_data(), 0.,
27.    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 , $m = \text{bottom}[0] \rightarrow \text{count}() / \text{channels_}$, 存储均值滑动和、方差滑动和以及滑动系数和分别为 μ_{old} , σ_{old}^2 , s_{old} , 且当前的 mini-batch 的均值和方差分别为 μ_B , σ_B^2 :

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

对于方差, 采用的是无偏估计:

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

```

1. // compute and save moving average
2. this->blobs_[2]->mutable_cpu_data()[0] *= this->moving_average_fra
   ction_;
3. this->blobs_[2]->mutable_cpu_data()[0] += 1;
4. /*
5. ** caffe_cpu_axpby<Dtype>(const int N, const Dtype alpha, const Dt
   ype* X,
6.                               const Dtype beta, Dtype* Y)
7. ** 功能:  $Y = \alpha * X + \beta * Y$ 
8. ** X 是 (N, 1)
9. ** Y 是 (N, 1)
10. */
11.
12. // 数学表达式:  $blobs\_ [0] = mean\_ + moving\_ average\_ fraction\_ * blobs\_ [0]$ 
13. caffe_cpu_axpby(this->mean_.count(), Dtype(1), this->mean_.cpu_dat
   a(),
14.     this->moving_average_fraction_, this->blobs_[0]->mutable_cpu_d
   ata());
15. int m = bottom[0]->count()/channels_;
16. // 计算无偏差估计的系数  $m/(m - 1)$ 
17. Dtype bias_correction_factor = m > 1 ? Dtype(m)/(m-1) : 1;
18. // 数学表达式:  $blobs\_ [1] = bias\_ correction\_ factor * variance\_ + mov
   ing\_ average\_ fraction\_ * blobs\_ [1]$ 
19. caffe_cpu_axpby(variance_.count(), bias_correction_factor,
20.     variance_.cpu_data(), moving_average_fraction_,
21.     this->blobs_[1]->mutable_cpu_data());

```

源代码注释如下所示:


```

1. template <typename Dtype>
2. void BatchNormLayer<Dtype>::Forward_cpu(const vector<Blob<Dtype>*>
    & bottom,
3.     const vector<Blob<Dtype>*>& top) {
4.     // 获取只读 bottom_data 的指针
5.     const Dtype* bottom_data = bottom[0]->cpu_data();
6.     // 获取读写 top_data 的指针
7.     Dtype* top_data = top[0]->mutable_cpu_data();
8.     // 获取 batch_size 的大小
9.     int num = bottom[0]->shape(0);
10.    int spatial_dim = bottom[0]->count()/(bottom[0]->shape(0) * this
        ->channels_);
11.
12.    // 如果输入 bottom 与输出 bottom 的地址不一致,
13.    // 则需要将 bottom_data 的数据拷贝到 top_data
14.    if (bottom[0] != top[0]) {
15.        caffe_copy(bottom[0]->count(), bottom_data, top_data);
16.    }
17.
18.    if (this->use_global_stats_) {
19.        // 如果 use_global_stats_ 为真, 那么我们使用全局的均值和方差
20.        // use the stored mean/variance estimates.
21.        // 如果滑动平均系数为 0, 设置 scale_factor 为 0, 否则设置 scale_fa
        ctor 为滑动平均系数的倒数
22.        const Dtype scale_factor = this->blobs_[2]->cpu_data()[0] == 0
            ?
23.            0 : 1 / this->blobs_[2]->cpu_data()[0];
24.        // 设置局部的均值
25.        caffe_cpu_scale(this->variance_.count(), scale_factor,
26.            this->blobs_[0]->cpu_data(), this->mean_.mutable_cpu_data
            ());
27.        // 设置局部方差
28.        caffe_cpu_scale(this->variance_.count(), scale_factor,
29.            this->blobs_[1]->cpu_data(), this->variance_.mutable_cpu_d
            ata());
30.    } else {
31.        // compute mean
32.
33.        /*
34.        ** 函数: caffe_cpu_gemv<Dtype>(const CBLAS_TRANSPOSE TransA, co
            nst int M,
35.
            const int N, const Dtype alpha, const
            Dtype* A,
36.
            const Dtype* x, const Dtype beta, Dtyp
            e* y)
37.        ** 功能:  $y = \alpha * A * x + \beta * y$ 
38.        ** 其中  $x$  和  $y$  是向量,  $A$  是矩阵
39.        **  $M$ :  $A$  的行数
40.        **  $N$ :  $A$  的列数
41.        */

```

```

42.    // 数学表达式: num_by_chans_ = 1. / (num * spatial_dim) * botto
    m_data * spatial_sum_multiplier_
43.    // bottom_data 是 (channels_ * num, spatial_dim)
44.    // spatial_sum_multiplier_ 是 (spatial_dim, 1), 元素全为 1
45.    // num_by_chans_ 是 (channels_ * num, 1)
46.    caffe_cpu_gemv<Dtype>(CblasNoTrans, this->channels_ * num, spa
    tial_dim,
47.        1. / (num * spatial_dim), bottom_data,
48.        this->spatial_sum_multiplier_.cpu_data(), 0.,
49.        this->num_by_chans_.mutable_cpu_data());
50.    // 数学表达式: mean_ = Trans(num_by_chans) * batch_sum_multipli
    er
51.    // num_by_chans_ 是 (num, channels_)
52.    // batch_sum_multiplier_ 是 (num, 1)
53.    // mean_ 是 (channels_, 1)
54.    // 最终得到每个 channel 的平均值
55.    caffe_cpu_gemv<Dtype>(CblasTrans, num, this->channels_, 1.,
56.        this->num_by_chans_.cpu_data(), this->batch_sum_multiplier
    _.cpu_data(), 0.,
57.        this->mean_.mutable_cpu_data());
58.    }
59.
60.    // subtract mean
61.
62.    /*
63.    ** 函数: caffe_cpu_gemm<Dtype>(const CBLAS_TRANSPOSE TransA, co
    nst CBLAS_TRANSPOSE TransB,
64.        const int M, const int N, const int K,
    const Dtype alpha,
65.        const Dtype* A, const Dtype* B, const
    Dtype beta, Dtype* C)
66.    ** 功能:  $C = \alpha * A * B + \beta * C$ 
67.    ** 其中 A 是 (M, K); B 是 (K, N); C 是 (M, N)
68.    */
69.    // 数学表达式: num_by_chans_ = batch_sum_multiplier_ * mean_
70.    // batch_sum_multiplier_ 是 (num, 1)
71.    // mean_ 是 (1, channels_)
72.    // num_by_chans_ 是 (num, channels_)
73.    caffe_cpu_gemm<Dtype>(CblasNoTrans, CblasNoTrans, num, this->cha
    nnels_, 1, 1,
74.        this->batch_sum_multiplier_.cpu_data(), this->mean_.cpu_data
    (), 0.,
75.        this->num_by_chans_.mutable_cpu_data());
76.    // 数学表达式: top_data = -1 * num_by_chans_ * spatial_sum_multip
    lier_ + top_data
77.    // num_by_chans_ 是 (channels_ * num, 1)
78.    // spatial_sum_multiplier_ 是 (1, spatial_dim)
79.    // top_data 是 (channels_ * num, spatial_dim)
80.    // 为每一个像素点减去均值

```

```

81.     caffe_cpu_gemm<Dtype>(CblasNoTrans, CblasNoTrans, this->channels_
    _ * num,
82.         spatial_dim, 1, -1, this->num_by_chans_.cpu_data(),
83.         this->spatial_sum_multiplier_.cpu_data(), 1., top_data);
84.
85.     // 如果 use_global_stats_ 为真
86.     if (!this->use_global_stats_) {
87.         // compute variance using  $\text{var}(X) = E((X-EX)^2)$ 
88.         // caffe_powx 是 element-wise 操作, 这里实现对每个元素平方
89.         caffe_powx(top[0]->count(), top_data, Dtype(2),
90.             this->temp_.mutable_cpu_data()); //  $(X-EX)^2$ 
91.
92.         // 数学表达式:  $\text{num\_by\_chans\_} = 1. / (\text{num} * \text{spatial\_dim}) * \text{temp\_}$ 
        * spatial_sum_multiplier_
93.         // temp_ 是 (channels_ * num, spatial_dim)
94.         // spatial_sum_multiplier_ 是 (spatial_dim, 1), 元素全为 1
95.         // num_by_chans_ 是 (channels_ * num, 1)
96.         caffe_cpu_gemv<Dtype>(CblasNoTrans, this->channels_ * num, spa
        tial_dim,
97.             1. / (num * spatial_dim), this->temp_.cpu_data(),
98.             this->spatial_sum_multiplier_.cpu_data(), 0.,
99.             this->num_by_chans_.mutable_cpu_data());
100.
101.        // 数学表达式:  $\text{variance\_} = \text{Trans}(\text{num\_by\_chans\_}) * \text{batch\_sum\_mul}$ 
        tiplier_
102.        // num_by_chans 是 (num, channels_)
103.        // batch_sum_multiplier_ 是 (num, 1), 元素全为 1
104.        // variance_ 是 (channels_, 1)
105.        // 计算出方差
106.        caffe_cpu_gemv<Dtype>(CblasTrans, num, this->channels_, 1.,
107.            this->num_by_chans_.cpu_data(), this->batch_sum_multiplier
            _.cpu_data(), 0.,
108.            this->variance_.mutable_cpu_data()); //  $E((X-EX)^2)$ 
109.
110.        // compute and save moving average
111.        this->blobs_[2]->mutable_cpu_data()[0] *= this->moving_average
            _fraction_;
112.        this->blobs_[2]->mutable_cpu_data()[0] += 1;
113.        /*
114.        ** caffe_cpu_axpby<Dtype>(const int N, const Dtype alpha, cons
        t Dtype* X,
115.                                const Dtype beta, Dtype* Y)
116.        ** 功能:  $Y = \alpha * X + \beta * Y$ 
117.        ** X 是 (N, 1)
118.        ** Y 是 (N, 1)
119.        */
120.
121.        // 数学表达式:  $\text{blobs\_}[0] = \text{mean\_} + \text{moving\_average\_fraction\_} * b$ 
        lobs_[0]

```

```

122.     caffe_cpu_axpby(this->mean_.count(), Dtype(1), this->mean_.cpu
_data(),
123.         this->moving_average_fraction_, this->blobs_[0]->mutable_c
pu_data());
124.     int m = bottom[0]->count()/channels_;
125.     // 计算无偏差估计的系数  $m/(m - 1)$ 
126.     Dtype bias_correction_factor = m > 1 ? Dtype(m)/(m-1) : 1;
127.     // 数学表达式:  $\text{blobs}[1] = \text{bias\_correction\_factor} * \text{variance\_} +$ 
moving_average_fraction_ * blobs_[1]
128.     caffe_cpu_axpby(variance_.count(), bias_correction_factor,
129.         variance_.cpu_data(), moving_average_fraction_,
130.         this->blobs_[1]->mutable_cpu_data());
131. }
132.
133. // normalize variance
134. /*
135. ** caffe_add_scalar(const int N, const float alpha, float* Y)
136. ** 功能: 给 Y 的每个元素加上 alpha
137. */
138. // 为方差加上一个附加值, 防止除数为 0
139. caffe_add_scalar(this->variance_.count(), this->eps_, this->vari
ance_.mutable_cpu_data());
140. // 为每个元素求开平方
141. caffe_powx(this->variance_.count(), this->variance_.cpu_data(),
Dtype(0.5),
142.         this->variance_.mutable_cpu_data());
143.
144. // replicate variance to input size
145. // 数学表达式:  $\text{num\_by\_chans\_} = \text{batch\_sum\_multiplier\_} * \text{variance\_}$ 
146. // batch_sum_multiplier_ 是 (num, 1), 元素全为 1
147. // variance_ 是 (1, channels)
148. // num_by_chans 是 (num, channels)
149. caffe_cpu_gemm<Dtype>(CblasNoTrans, CblasNoTrans, num, this->cha
nnels_, 1, 1,
150.     this->batch_sum_multiplier_.cpu_data(), this->variance_.cpu_
data(), 0.,
151.     this->num_by_chans_.mutable_cpu_data());
152.
153. // 数学表达式:  $\text{temp\_} = \text{num\_by\_chans\_} * \text{spatial\_sum\_multiplier}$ 
154. // num_by_chans_ 是 (channels * num, 1)
155. // spatial_sum_multiplier_ 是 (1, spatial_dim), 元素全为 1
156. // temp_ 是 (channels * num, spatial_dim)
157. caffe_cpu_gemm<Dtype>(CblasNoTrans, CblasNoTrans, this->channels
_ * num,
158.     spatial_dim, 1, 1., this->num_by_chans_.cpu_data(),
159.     this->spatial_sum_multiplier_.cpu_data(), 0., this->temp_.mu
table_cpu_data());
160. // 实现 element-wise 相除
161. caffe_div(this->temp_.count(), top_data, this->temp_.cpu_data(),
top_data);

```

```

162. // TODO(cdoersch): The caching is only needed because later in-p
    lace Layers
163. //                               might clobber the data. Can we skip this if
    they won't?
164. caffe_copy(this->x_norm_.count(), top_data, this->x_norm_.mutabl
    e_cpu_data());
165. }

```

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

$$\mu_B = \frac{1}{m} \sum_{i=1}^m x_i \quad (4)$$

- min-batch 方差 σ_B^2

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

- min-batch 归一化的数据 y_i

$$y_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \quad (6)$$

- mini-batch 损失值 L 关于 x_i 的梯度

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

(1) 当 $i = j$ 时，有：

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

(2) 当 $i \neq j$ 时，有：

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

最终，我们可以求得：

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

源代码注释如下所示：

```

1. template <typename Dtype>
2. void BatchNormLayer<Dtype>::Backward_cpu(const vector<Blob<Dtype>*>
    & top,
3.     const vector<bool>& propagate_down,
4.     const vector<Blob<Dtype>*>& bottom) {
5.     // 获取只读 top_diff 指针
6.     const Dtype* top_diff;
7.     if (bottom[0] != top[0]) { // 如果 bottom[0] 与 top[0] 所指的地址
        不一致
8.         top_diff = top[0]->cpu_diff();
9.     } else {
10.         caffe_copy(this->x_norm_.count(), top[0]->cpu_diff(), this->x_
            norm_.mutable_cpu_diff());
11.         top_diff = this->x_norm_.cpu_diff();
12.     }
13.     // 获取读写 bottom_diff 的指针
14.     Dtype* bottom_diff = bottom[0]->mutable_cpu_diff();
15.     // 如果 use_global_stats_ 为真
16.     if (this->use_global_stats_) {
17.         caffe_div(this->temp_.count(), top_diff, this->temp_.cpu_data
            (), bottom_diff);
18.         return;
19.     }
20.     // 获取只读 x_norm_ 指针
21.     const Dtype* top_data = this->x_norm_.cpu_data();
22.     // 获取 batch_size 的大小
23.     int num = bottom[0]->shape()[0];
24.     // 获取 spatial_dim 的大小
25.     int spatial_dim = bottom[0]->count()/(bottom[0]->shape(0) * this
        ->channels_);
26.     // if  $Y = (X - \text{mean}(X)) / (\text{sqrt}(\text{var}(X) + \text{eps}))$ , then
27.     //
28.     //  $dE/dY/dX =$ 
29.     //  $(dE/dY - \text{mean}(dE/dY) - \text{mean}(dE/dY \cdot Y) \cdot Y)$ 
30.     //  $\cdot \text{sqrt}(\text{var}(X) + \text{eps})$ 
31.     //
32.     // where  $\cdot$  and  $\cdot$  are hadamard product and elementwise divis
        ion,
33.     // respectively,  $dE/dY$  is the top diff, and mean/var/sum are all
        computed
34.     // along all dimensions except the channels dimension. In the a
        bove
35.     // equation, the operations allow for expansion (i.e. broadcast)
        along all
36.     // dimensions except the channels dimension where required.
37.
38.     //  $\text{sum}(dE/dY \cdot Y)$ 
39.
40.     // 实现 element wise 相乘
41.     //  $\text{bottom\_diff}[i] = \text{top\_data}[i] * \text{top\_diff}[i]$ 

```

```

42.     caffe_mul(this->temp_.count(), top_data, top_diff, bottom_diff);
43.
44.     // 数学表达式: num_by_chans_ = bottom_diff * spatial_sum_multipli
        er_
45.     // bottom_diff 是 (channels_ * num, spatial_dim)
46.     // spatial_sum_multiplier_ 是 (spatial_dim, 1)
47.     // num_by_chans 是 (channels_ * num, 1)
48.     caffe_cpu_gemv<Dtype>(CblasNoTrans, this->channels_ * num, spati
        al_dim, 1.,
49.         bottom_diff, this->spatial_sum_multiplier_.cpu_data(), 0.,
50.         this->num_by_chans_.mutable_cpu_data());
51.
52.     // 数学表达式: mean_ = Trans(num_by_chans_) * batch_sum_multiplie
        r_
53.     // num_by_chans_ 是 (num, channels)
54.     // batch_sum_multiplier_ 是 (num, 1), 元素全是 1
55.     // mean_ 是 (channels, 1)
56.     caffe_cpu_gemv<Dtype>(CblasTrans, num, this->channels_, 1.,
57.         this->num_by_chans_.cpu_data(), this->batch_sum_multiplier_.
        cpu_data(), 0.,
58.         this->mean_.mutable_cpu_data());
59.
60.     // reshape (broadcast) the above
61.     // 数学表达式: num_by_chans_ = batch_sum_multiplier_ * mean_
62.     // batch_sum_multiplier_ 是 (num, 1), 元素全是 1
63.     // mean_ 是 (1, channels_)
64.     // num_by_chans_ 是 (num, channels_)
65.     caffe_cpu_gemm<Dtype>(CblasNoTrans, CblasNoTrans, num, this->cha
        nnels_, 1, 1,
66.         this->batch_sum_multiplier_.cpu_data(), this->mean_.cpu_data
        (), 0.,
67.         this->num_by_chans_.mutable_cpu_data());
68.
69.     // 数学表达式: bottom_diff = num_by_chans_ * saptial_sum_multipli
        er_
70.     // num_by_chans_ 是 (channels * num, 1)
71.     // spatial_sum_multiplier_ 是 (1, spatial_dim), 所有元素为 1
72.     // bottom_diff 是 (channels * num, spatial_dim)
73.     caffe_cpu_gemm<Dtype>(CblasNoTrans, CblasNoTrans, this->channels
        _ * num,
74.         spatial_dim, 1, 1, this->num_by_chans_.cpu_data(),
75.         this->spatial_sum_multiplier_.cpu_data(), 0., bottom_diff);
76.
77.     // sum(dE/dY \cdot Y) \cdot Y
78.     // 采用 element-wise 的乘法
79.     // bottom_diff[i] = top_data[i] * bottom_diff[i]
80.     caffe_mul(this->temp_.count(), top_data, bottom_diff, bottom_dif
        f);
81.
82.     // sum(dE/dY)-sum(dE/dY \cdot Y) \cdot Y

```

```

83. // 数学表达式: num_by_chans_ = top_diff * spatial_sum_multiplier_
84. // top_diff 是 (channels_ * num, spatial_dim)
85. // spatial_sum_multiplier_ 是 (spatial_dim, 1), 所有元素为 1
86. // num_by_chans_ 是 (channels_ * num, 1)
87. caffe_cpu_gemv<Dtype>(CblasNoTrans, this->channels_ * num, spatial_dim, 1.,
88.     top_diff, this->spatial_sum_multiplier_.cpu_data(), 0.,
89.     this->num_by_chans_.mutable_cpu_data());
90.
91. // 数学表达式: mean_ = Trans(num_by_chans_) * batch_sum_multiplier_
92. // num_by_chans 是 (num, channels_)
93. // batch_sum_multiplier_ 是 (num, 1), 所有元素为 1
94. // mean_ 是 (channels_, 1)
95. caffe_cpu_gemv<Dtype>(CblasTrans, num, this->channels_, 1.,
96.     this->num_by_chans_.cpu_data(), this->batch_sum_multiplier_.cpu_data(), 0.,
97.     this->mean_.mutable_cpu_data());
98.
99. // reshape (broadcast) the above to make
100. // sum(dE/dY)-sum(dE/dY \cdot Y) \cdot Y
101. // 数学表达式: num_by_chans_ = batch_sum_multiplier_ * mean_
102. // batch_sum_multiplier_ 是 (num, 1), 元素全为 1
103. // mean_ 是 (1, channels)
104. // num_by_chans_ 是 (num, channels)
105. caffe_cpu_gemm<Dtype>(CblasNoTrans, CblasNoTrans, num, this->channels_, 1, 1,
106.     this->batch_sum_multiplier_.cpu_data(), this->mean_.cpu_data(), 0.,
107.     this->num_by_chans_.mutable_cpu_data());
108.
109. // 数学表达式: bottom_diff = num_by_chans_ * spatial_sum_multiplier_
110. // num_by_chans_ 是 (num * channels_, 1)
111. // spatial_sum_multiplier_ 是 (1, spatial_dim), 元素全为 1
112. // bottom_diff 是 (num * channels_, spatial_dim)
113. caffe_cpu_gemm<Dtype>(CblasNoTrans, CblasNoTrans, num * channels_,
114.     spatial_dim, 1, 1., num_by_chans_.cpu_data(),
115.     spatial_sum_multiplier_.cpu_data(), 1., bottom_diff);
116.
117. // dE/dY - mean(dE/dY)-mean(dE/dY \cdot Y) \cdot Y
118.
119. /*
120. ** caffe_cpu_axpby<Dtype>(const int N, const Dtype alpha, const Dtype* X,
121.                             const Dtype beta, Dtype* Y)
122. ** 功能: Y= alpha * X + beta * Y
123. ** X 是 (N, 1)
124. ** Y 是 (N, 1)

```



```

125.     */
126.     // 数学表达式: bottom_diff = top_diff - 1 / (num * spatial_dim) *
        bottom_diff
127.     caffe_cpu_axpby(temp_.count(), Dtype(1), top_diff,
128.         Dtype(-1. / (num * spatial_dim)), bottom_diff);
129.
130.     // note: temp_ still contains sqrt(var(X)+eps), computed during
        the forward
131.     // pass.
132.     // 实现 element-wise 元素相除
133.     caffe_div(temp_.count(), bottom_diff, temp_.cpu_data(), bottom_d
        iff);
134. }
```

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>)

0 赞

1678 人读过

新浪微博

微信

...

立即登录, 发表评论.
没有帐号? 立即注册

0 条评论

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%E5%9C%A8%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>)

Proudly powered by Leanote (<https://leanote.com>)