《Batch Normalization Accelerating Deep Network Training by Reducing Internal Covariate Shift》阅读笔记与实现

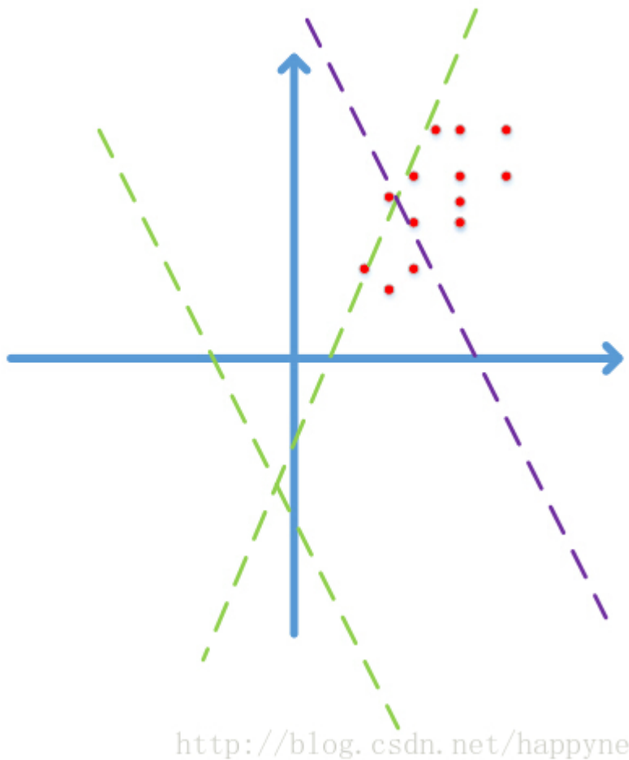
2015年03月13日 12:42:51 43004

在过年之前，MSRA和Google相继在ImagenNet图像识别数据集上报告他们的效果超越了人类水平，下i 分两期介绍两者的算法细节。

这次先讲Google的这篇《Batch Normalization Accelerating Deep Network Training by Reducing Inter Covariate Shift》，主要是因为这里面的思想比较有普适性，而且一直答应群里的人写一个有关预处理的科普，但一直没抽出时间来写。

一、神经网络中的权重初始化与预处理方法的关系

如果做过dnn的实验，大家可能会发现在对数据进行预处理，例如白化或者zscore，甚至是简单的减均值操作都是可以加速收敛的，例如下图所示的一个简单的例子：



图中红点代表2维的数据点，由于图像数据的每一维一般都是0-255之间的数字，因此数据点只会落在第一象限，而且图像数据具有很强的相关性，比如第一个灰度值为30，比较黑，那它旁边的一个像素值一般不会超过100，否则给人的感觉就像噪声一样。由于强相关性，数据点仅会落在第一象限的很小的区域中，形成类似上图所示的狭长分布。

而神经网络模型在初始化的时候，权重W是随机采样生成的，一个常见的神经元表示为：ReLU(Wx+b) = max(Wx+b,0)，即在Wx+b=0的两侧，对数据采用不同的操作方法。具体到ReLU就是一侧收缩，一侧保持不变。

随机的Wx+b=0表现为上图中的随机虚线，注意到，两条绿色虚线实际上并没有什么意义，在使用梯度下降时，可能需要很多次迭代才会使这些虚线对数据点进行有效的分割，就像紫色虚线那样，这势必会带来求解速率变慢的问题。更何况，我们这只是个二维的演示，数据占据四个象限中的一个，如果是几百、几千、上万维呢？而且数据在第一象限中也只是占了很小的一部分区域而已，可想而知不对数据进行预处理带来了多少运算资源的浪费，而且大量的数据外分割面在迭代时很可能会在刚进入数据中时就遇到了一个局部最优，导致overfit的问题。

这时，如果我们将数据减去其均值，数据点就不再只分布在第一象限，这时一个随机分界面落入数据分布的概率增加了多少呢？2^n倍！如果我们使用去除相关性的算法，例如PCA和ZCA白化，数据不再是一个狭长的分布，随机分界面有效的概率就又大大增加了。

不过计算协方差矩阵的特征值太耗时也太耗空间，我们一般最多只用到z-score处理，即每一维度减去自身均值，再除以自身标准差，这样能使数据点在每维上具有相似的宽度，可以起到一定的增大数据分布范围，进而使更多随机分界面有意义的作用。

二、Batch Normalization

上一节我们讲到对输入数据进行预处理，减均值->zscore->白化可以逐级提升随机初始化的权重对数据分割的有效性，还可以降低overfit的可能性。我们都知道，现在的神经网络的层数都是很深的，如果我们对每一层的数据都进行处理，训练时间和overfit程度是否可以降低呢？Google的这篇论文给出了答案。

1、算法描述

按照第一章的理论，应当在每一层的激活函数之后，例如ReLU=max(Wx+b,0)之后，对数据进行归一化。然而，文章中说这样做在训练初期，分界面还在剧烈变化时，计算出的参数不稳定，所以退而求其次，在Wx+b之后进行归一化。因为初始的W是从标准高斯分布中采样得到的，而W中元素的数量远大于x，Wx+b每维的均值本身就接近0、方差接近1，所以在Wx+b后使用Batch Normalization能得到更稳定的结果。

文中使用了类似z-score的归一化方式：每一维度减去自身均值，再除以自身标准差，由于使用的是随机梯度下降法，这些均值和方差也只能在当前迭代的batch中计算，故作者给这个算法命名为Batch Normalization。这里有一点需要注意，像卷积层这样具有权值共享的层，Wx+b的均值和方差是对整张map求得的，在batch_size * channel * height * width这么大的一层中，对总共batch_size*height*width个像素点统计得到一个均值和一个标准差，共得到channel组参数。

在Normalization完成后，Google的研究员仍对数值稳定性不放心，又加入了两个参数gamma和beta，

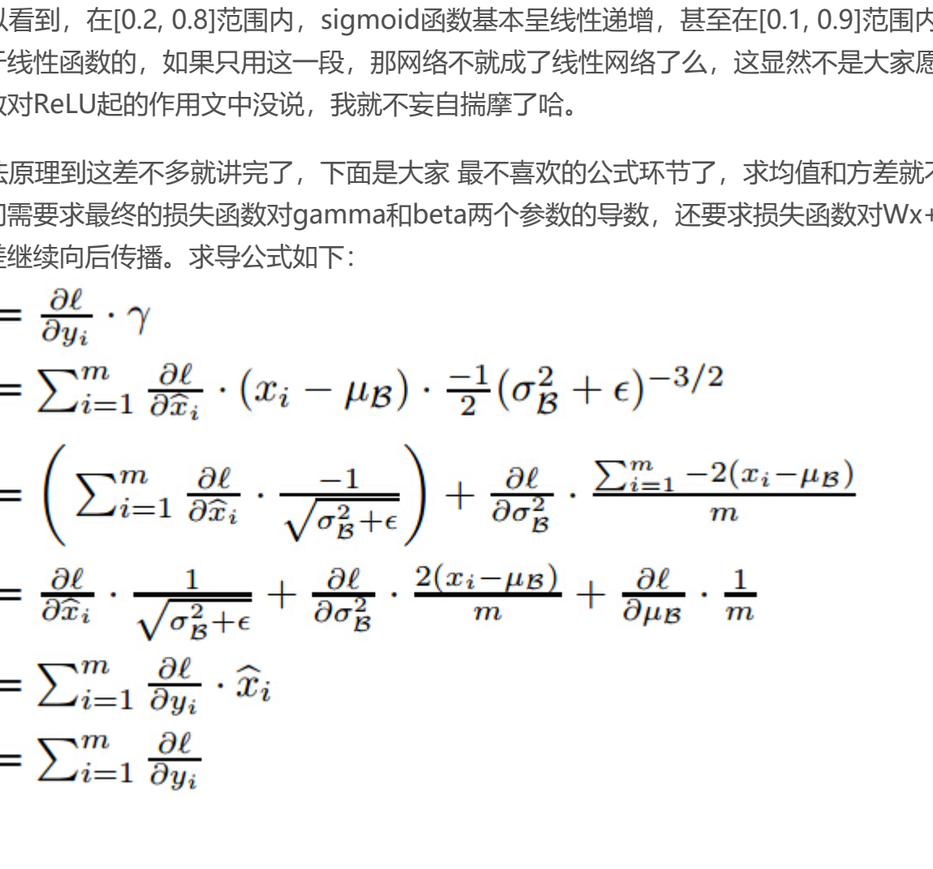
happynear

原创	粉丝	喜欢	评论
11	474	15	430

等级：博客 4 访问量：21万+
积分：1233 排名：4万+

$$y^{(k)} = \gamma^{(k)}\widehat{x}^{(k)} + \beta^{(k)}.$$

注意到，如果我们令gamma等于之前求得的标准差，beta等于之前求得的均值，则这个变换就又将数据还原回去了。在他们的模型中，这两个参数与每层的W和b一样，是需要迭代求解的。文章中举了个例子，在sigmoid激活函数的中间部分，函数近似于一个线性函数（如下图所示），使用BN后会使归一化后的数据仅使用这一段线性的部分（吐槽一下：再乘个2之类的不就行了）。



可以看到，在[0.2, 0.8]范围内，sigmoid函数基本呈线性递增，甚至在[0.1, 0.9]范围内，sigmoid函数都是类似于线性函数的，如果只用这一段，那网络不就成了线性网络了么，这显然不是大家愿意见到的。至于这两个参数对ReLU起的作用文中没说，我就不妄自揣摩了哈。

算法原理到这差不多就讲完了，下面是大家 最不喜欢的公式环节了，求均值和方差就不用说了，在BP的时候，我们需要求最终的损失函数对gamma和beta两个参数的导数，还要求损失函数对Wx+b中的x的导数，以便使误差继续向后传播。求导公式如下：

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

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

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

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

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

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

具体的公式推导就不写了，有兴趣的读者可以自己推一下，主要用到了链式法则。

在训练的最后一个epoch时，要对这一epoch所有的训练样本的均值和标准差进行统计，这样在一张测试图片进来时，使用训练样本中的标准差的期望和均值的期望（好绕口）对测试数据进行归一化，注意这里标准差使用的期望是其无偏估计：

$$\mathrm{E}[x] \leftarrow \mathrm{E}_{\mathcal{B}}[\mu_{\mathcal{B}}]$$

$$\mathrm{Var}[x] \leftarrow \frac{m}{m-1}\mathrm{E}_{\mathcal{B}}[\sigma_{\mathcal{B}}^2]$$

2、算法优势

论文中将Batch Normalization的作用说得突破天际，好似一下解决了所有问题，下面就来一一列举一下：

(1) 可以使用更高的学习率。如果每层的scale不一致，实际上每层需要的学习率是不一样的，同一层不同维度的scale往往也需要不同大小的学习率，通常需要使用最小的那个学习率才能保证损失函数有效下降，Batch Normalization将每层、每维的scale保持一致，那么我们就可以直接使用较高的学习率进行优化。

(2) 移除或使用较低的dropout。 dropout是常用的防止overfitting的方法，而导致overfit的位置往往在数据边界处，如果初始化权重就已经落在数据内部，overfit现象就可以得到一定的缓解。论文中最后的模型分别使用10%、5%和0%的dropout训练模型，与之前的40%-50%相比，可以大大提高训练速度。

(3) 降低L2权重衰减系数。 还是一样的问题，边界处的局部最优往往有几维的权重（斜率）较大，使用L2衰减可以缓解这一问题，现在用了Batch Normalization，就可以把这个值降低了，论文中降低为原来的5倍。

(4) 取消Local Response Normalization层。 由于使用了一种Normalization，再使用LRN就显得没那么必要了。而且LRN实际上也没那么work。

(5) 减少图像扭曲的使用。 由于现在训练epoch数降低，所以要对输入数据少做一些扭曲，让神经网络多看看真实的数据。

三、实验

这里我只在matlab上面对算法进行了仿真，修改了DeepLearnToolbox 里面的NN模型，代码如下：

在前向传播时，分两种情况进行讨论：如果是在train过程，就使用当前batch的数据统计均值和标准差，并按照第二章所述公式对Wx+b进行归一化，之后再乘上gamma，加上beta得到Batch Normalization层的输出；如果在进行test过程，则使用记录下的均值和标准差，还有之前训练好的gamma和beta计算得到结果

```
    nn.a_pre{i} = nn.a{i - 1} * nn.W{i - 1}';
    norm_factor = nn.gamma{i-1}./sqrt(nn.mean_sigma2{i-1}+nn.epsilon);
    nn.a_hat{i} = bsxfun(@times, nn.a_pre{i}, norm_factor);
    nn.a_hat{i} = bsxfun(@plus, nn.a_hat{i}, nn.beta{i-1} -  norm_factor .* nn.mean_mu{i-1});
else
    nn.a_pre{i} = nn.a{i - 1} * nn.W{i - 1}';
    nn.mu{i-1} = mean(nn.a_pre{i});
    x_mu = bsxfun(@minus,nn.a_pre{i},nn.mu{i-1});
    nn.sigma2{i-1} = mean(x_mu.^2);
    norm_factor = nn.gamma{i-1}./sqrt(nn.sigma2{i-1}+nn.epsilon);
    nn.a_hat{i} = bsxfun(@times, nn.a_pre{i}, norm_factor);
    nn.a_hat{i} = bsxfun(@plus, nn.a_hat{i}, nn.beta{i-1} -  norm_factor .* nn.mu{i-1});
end;
```


]传播就跟上面那一堆公式一样啦，注意为了运行效率，尽量使用向量化的代码，避免使用for循环：

```
[plain]
d_xhat = bsxfun(@times, d{i}(:,2:end), nn.gamma{i-1});
x_mu = bsxfun(@minus, nn.a_pre{i}, nn.mu{i-1});
inv_sqrt_sigma = 1 ./ sqrt(nn.sigma2{i-1} + nn.epsilon);
d_sigma2 = -0.5 * sum(d_xhat .* x_mu) .* inv_sqrt_sigma.^3;
d_mu = bsxfun(@times, d_xhat, inv_sqrt_sigma);
d_mu = -1 * sum(d_mu) -2 .* d_sigma2 .* mean(x_mu);
d_gamma = mean(d{i}(:,2:end) .* nn.a_hat{i});
d_beta = mean(d{i}(:,2:end));
di1 = bsxfun(@times,d_xhat,inv_sqrt_sigma);
di2 = 2/m * bsxfun(@times, d_sigma2,x_mu);
d{i}(:,2:end) = di1 + di2 + 1/m * repmat(d_mu,m,1);
```

在训练的最后一个epoch，要对所有的gamma和beta进行统计，代码很简单就不贴了，完整代码在我的Github上有：https://github.com/happynear/DeepLearnToolbox

1、sigmoid激活函数的过饱和问题

经测试发现算法对sigmoid激活函数的提升非常明显，解决了困扰学术界十几年的sigmoid过饱和的问题，即在深层的神经网络中，前几层在梯度下降时得到的梯度过低，导致深层神经网络变成了前边是随机变换，只在最后几层才是真正在做分类的问题。

下面是使用一个10个隐藏层的nn网络，对mnist进行分类，每层的梯度值：

使用Batch Normalization前：

```
[plain]
epoch:1 iteration:10/300
 3.23e-07 8.3215e-07 3.3605e-06 1.5193e-05 6.4892e-05 0.00027249 0.0011954 0.006295 0.029835 0.12476 0.3894
8
epoch:1 iteration:20/300
 4.4649e-07 1.3282e-06 5.6753e-06 2.5294e-05 0.00010326 0.00043651 0.0019583 0.0096396 0.040469 0.16142 0.5
235
epoch:1 iteration:30/300
 4.6973e-07 1.2993e-06 5.3923e-06 2.3111e-05 9.4839e-05 0.00040398 0.0017893 0.0081367 0.037543 0.1544 0.46
472
epoch:1 iteration:40/300
 4.6986e-07 1.3801e-06 5.677e-06 2.4355e-05 0.00010245 0.00041999 0.0019832 0.0095022 0.043719 0.17696 0.56
134
epoch:1 iteration:50/300
 4.6964e-07 1.6532e-06 7.2543e-06 3.0731e-05 0.00011805 0.00048795 0.0021705 0.0099466 0.042835 0.17993 0.5
319
```

可以看到，最开始的几层只有1e-6到1e-7这个量级的梯度，基本上梯度在最后3层就已经饱和了。

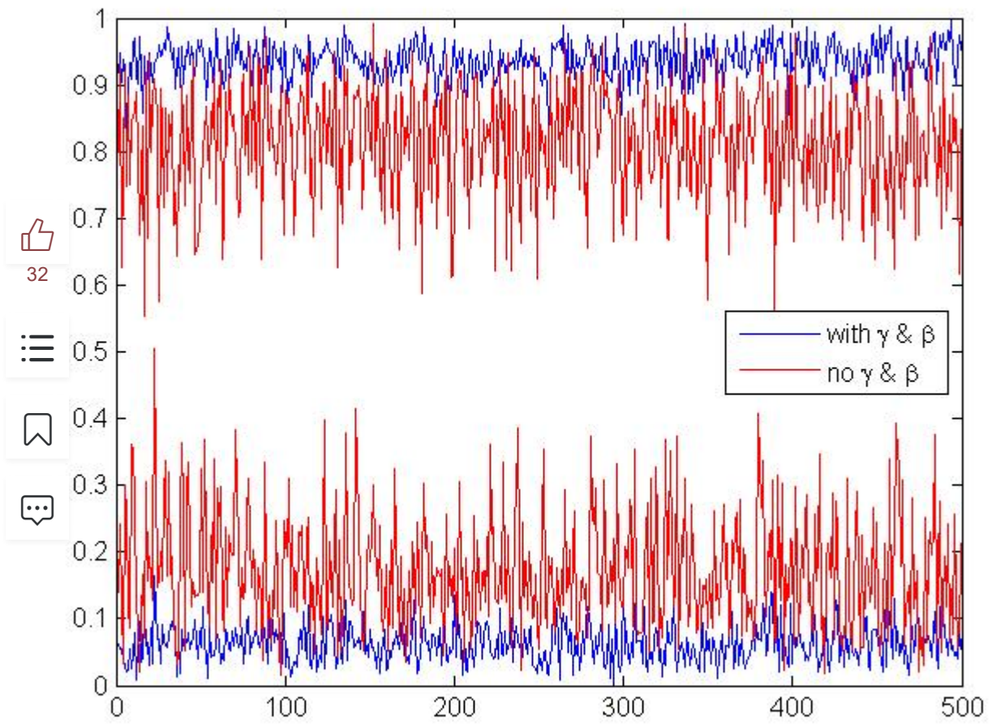
使用Batch Normalization后：

```
[plain]
epoch:1 iteration:10/300
 0.27121 0.15534 0.15116 0.15409 0.15515 0.14542 0.12878 0.13888 0.16607 0.21036 0.76037
epoch:1 iteration:20/300
 0.24567 0.15369 0.14169 0.13183 0.1278 0.13904 0.13546 0.12032 0.14332 0.14868 0.54481
epoch:1 iteration:30/300
 0.30403 0.16365 0.14119 0.14502 0.13916 0.12851 0.11781 0.11424 0.11082 0.1088 0.39574
epoch:1 iteration:40/300
 0.32681 0.19801 0.16792 0.14741 0.13294 0.12805 0.13754 0.12941 0.13288 0.12957 0.50937
epoch:1 iteration:50/300
 0.32358 0.17484 0.16367 0.16605 0.17118 0.14703 0.14458 0.12693 0.13928 0.11938 0.3692
```

我第一次看到的时候，就像之前看到ReLU一样惊艳，终于，sigmoid的饱和问题也得到了解决。不过论文中还有我自己的实验都表明，sigmoid在分类问题上确实没有ReLU好用，大概是因为sigmoid的中间部分太“线性”了，不像ReLU一个很大的转折，在拟合复杂非线性函数的时候可能没那么高效，真的是蛮遗憾的。

2、gamma和beta的作用

在第二章提到，引入gamma和beta两个参数是为了避免数据只用sigmoid的线性部分，这里做了个简单的测试，将用和不用gamma与beta参数训练出的网络的最大/最小激活值显示出来：



可以看到，如果不使用gamma和beta，激活值基本上会在[0.1 0.9]这个近似线性的区域中，这与深度神经网络所要求的“多层非线性函数逼近任意函数”的要求不符，所以引入gamma和beta还是有必要的，深度网络会自动决定使用哪一段函数（这是我自己想的，其具体作用欢迎讨论）。

对于ReLU来说，gamma的作用可能不是很明显，因为relu是分段”线性“的，对数值进行伸缩并不能影响relu取x还是取0。但beta的作用就很大了，试想一下如果没有beta，经过batch normalization层的特征，都具有0均值的期望，这样岂不是强制令ReLU的输出有一半是0一半非0么？这与我们的初衷不太相符，我们希望神经网络自行决定在什么位置去设定这个阈值，而不是增加一个如此强的限制。另外，因为这个beta我曾经还闹了个大笑话，记录在http://blog.csdn.net/happynear/article/details/46583811，请大家引以为戒。

四、总结

Batch Normalization的加速作用体现在两个方面：一是归一化了每层和每维度的scale，所以可以整体使用一个较高的学习率，而不必像以前那样迁就小scale的维度；二是归一化后使得更多的权重分界面落在了数据中，降低了overfit的可能性，因此一些防止overfit但会降低速度的方法，例如dropout和权重衰减就可以不使用或者降低其权重。

截止到目前，还没有哪个机构宣布重现了论文中的结果，不过归一化的用处在理论层面就已经有了保证，以后也许归一化的形式会有所改变，但逐层的归一化应该会成为一种标准。本博客文章仅仅给出了归一化优点的几何解释，希望有更多的理论解释来指导我们使用归一化层。

就目前来看，争议的重点在于归一化的位置，还有gamma与beta参数的引入，从理论上分析，论文中的这两个细节实际上并不符合ReLU的特性：ReLU后，数据分布重新回到第一象限，这时是最应当进行归一化的；gamma与beta对sigmoid函数确实能起到一定的作用（实际也不如固定gamma=2），但对于ReLU这种分段线性的激活函数，并不存在sigmoid的低scale呈线性的现象。期待更多的理论分析，我自己也会持续跟进这个方向。

五、一些资源

本文所用到的matlab代码：https://github.com/happynear/DeepLearnToolbox
Caffe的BN实现：https://github.com/ducha-aiki/caffe/tree/bn
cxxnet的BN实现：https://github.com/antinucleon/cxxnet