深度学习系列（9）：Batch Normalization

batch normalization(Ioffe and Szegedy, 2015) 是优化深度神经网络中最激动人心的创新之一。实际上它并不是一个优化算法，而是一个自适应的重新参数化 的方法，试图解决训练非常深层模型的困难。
Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift
机器学习领域有一个很重要的假设：iid独立同分布假设，就是假设训练数据和测试数据满足相同分布，这是通过训练数据训练出来的模型能够在测试集上获得好的效果的一个基本保证。Batch Normalization就是在深度神经网络训练过程中使得每一层神经网络的输入保持相同分布。

一、Internal covariate shift

首先给出covariate shift的定义：模型实例集合中的输入值X的分布总是变化，违背了idd独立通同布假设。

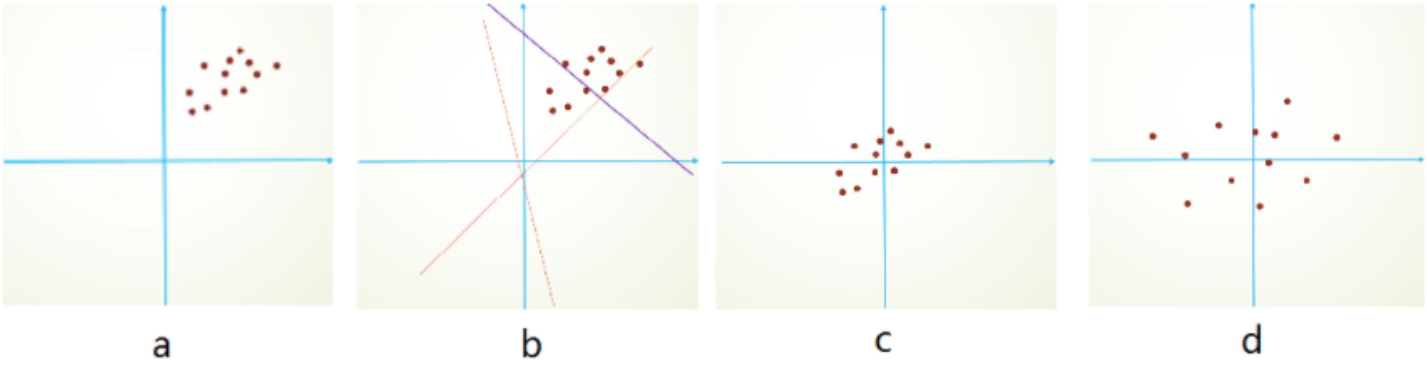
深度学习网络包含很多隐层的网络结构，在训练过程中参数会不断发生改变，导致后续每一层输入的分布也面临着covariate shift，也就是在训练过程中，隐层的输入分布总是发生改变，这就是所谓的Internal covariate shift，Internal指的是深层网络的隐层，covariate shift发生在深度神经网络内部，就被称作Internal covariate shift。

在DNN的实验中，对数据进行预处理时，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。为什么减均值、白化可以加快训练，作如下分析：

首先，图像数据的每一维一般都是0~255之间的数字，因此数据点智慧落在第一象限，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小区域内，形成类似第一个图的狭长分布。

其次，神经网络模型在初始化的时候，权重W都是随机采样生成的，一般都是零均值，因此起初的拟合y=Wx+b，基本过原点附近，如图b红色虚线。因此，网络需要经过多次迭代学习才能逐步达到如紫色实线的拟合，即收敛的比较慢。更何况，这里只是个二维的演示，数据占据四个象限中的一个，但如果是几百、几千、上万维呢？而且数据在第一象限也只是占了很小的一部分区域而已，可想而知若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是就遇到了一个局部最优，导致overfit的问题。如果我们对输入数据先作减均值操作，如图c，数据点就不再只分布在第一象限，这是一个随机分界面落入数据分布的概率增加了2^n倍，大大加快学习。更进一步的，我们对数据再进行去相关操作，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，使得数据更加容易区分，这样又会加快训练，如图d。



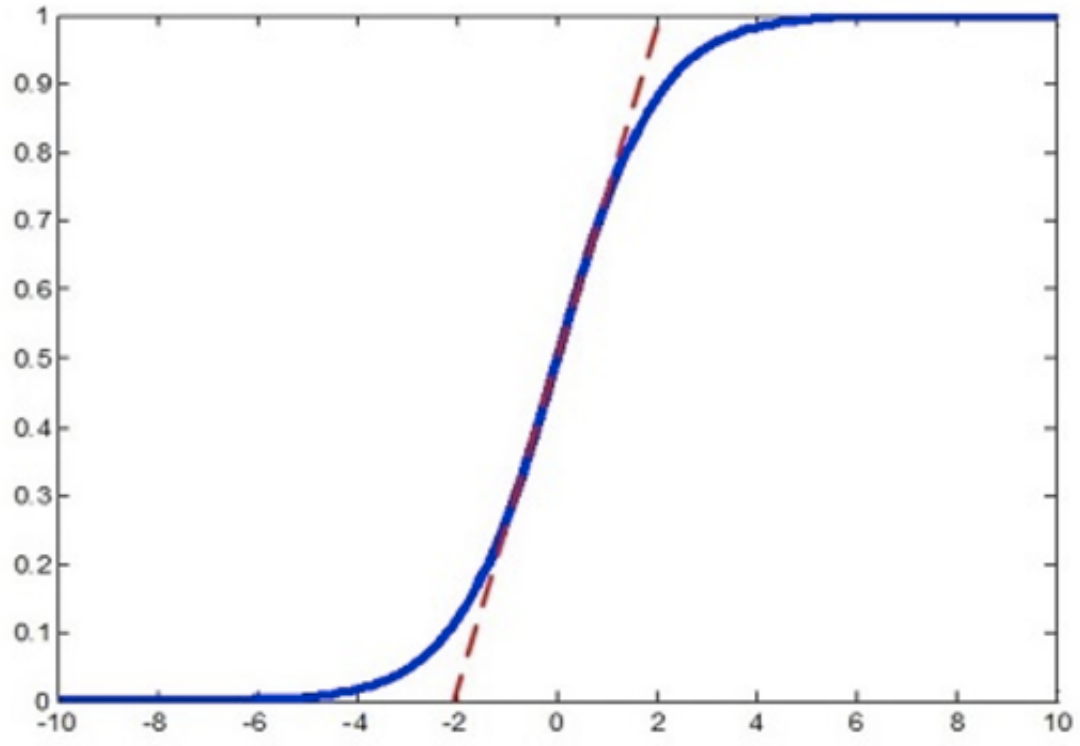
不过计算协方差的特征值太耗时也太耗空间，一般最多只用到z-score处理，即每一维减去自身均值，再除以自身标准差，这样能使数据点在每维上具有相似的宽度，可以起到增大数据分布范围，进而使更多随机分界面有意义的作用。

二、Batch Normalization

2.1 直观解释

Batch Normalization的基本思想其实很直观：因为深层神经网络在做非线性变换前的激活输入值（就是那个x=WU+B,U是输入）随着网络深度加深或者在训练过程中，其分布逐渐发生偏移或者变动，之所以收敛慢，一般是整体分布逐渐往非线性函数的取值区间的上下限两端靠近（对于Sigmoid函数来说，意味着激活输入值WU+B是大的负值或者正值），所以这导致反向传播的时候低层神经网络的梯度消失，这是训练深层神经网络收敛越来越慢的本质原因，而BN就是通过一定的规范化手段，对于每个隐层神经元，把逐渐向非线性函数映射后向取值区间极限饱和区靠拢的输入分布强制拉回到均值为0方差为1的比较标准的正态分布，使得非线性变换函数的输入值落入对输入比较敏感的区域，以此避免梯度消失问题。因为梯度一直都能保持比较大的状态，所以很明显对神经网络的参数调整效率比较高，就是变动大，就是说向损失函数最优值迈动的步子大，也就是说收敛地快。

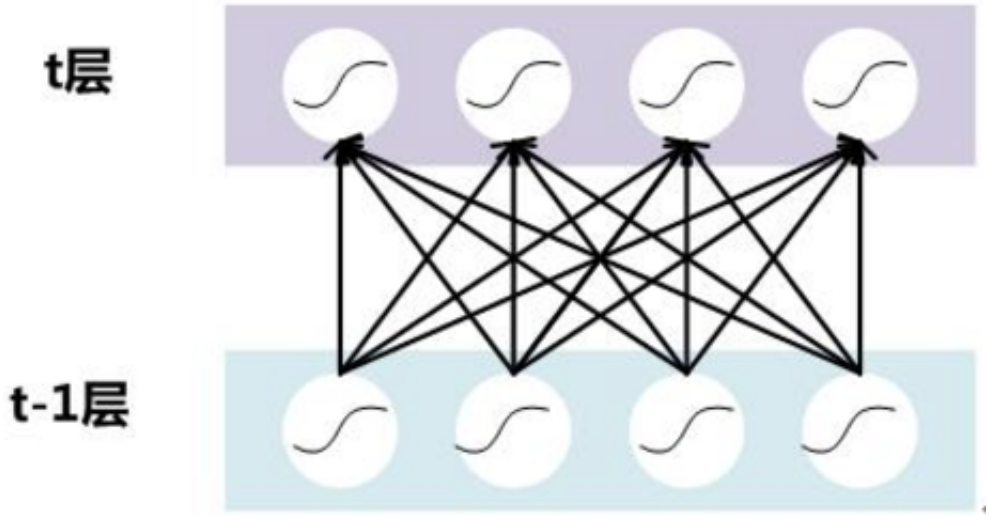
但是这里有个问题，如果都通过Batch Normalization，那么不就跟把非线性函数替换成线性函数效果相同了？我们知道，如果是多层的线性函数变换，其实这个深层是没有意义的，因为多层线性网络跟一层线性网络是等价的。这意味着网络的表达能力下降了，这也意味着深度的意义就没有了。比如下图，在使用sigmoid激活函数的时候，如果把数据限制到零均值单位方差，那么相当于只使用了激活函数中近似线性的部分，这显然会降低模型的表达能力。



BN为了保证非线性的获得，对变换后的满足均值为0方差为1的x又进行了scale加上shift操作(y=scale*x+shift)，每个神经元增加了两个参数scale和shift参数，这两个参数是通过训练学习到的，意思是通过scale和shift把这个值从标准正态分布左移或者由移一点并长胖一点或者变瘦一点，每个实例挪动的程度不一样，这样等价于非线性函数的值从正中心周围的线性区往非线性区动了动，让因训练所需而“刻意”加入的BN能够有可能还原最初的输入。核心思想应该是想找到一个线性和非线性的较好平衡点，既能享受非线性的较强表达能力的好处，又避免太靠非线性区两头使得网络收敛速度太慢。从而保证整个网络的capacity。

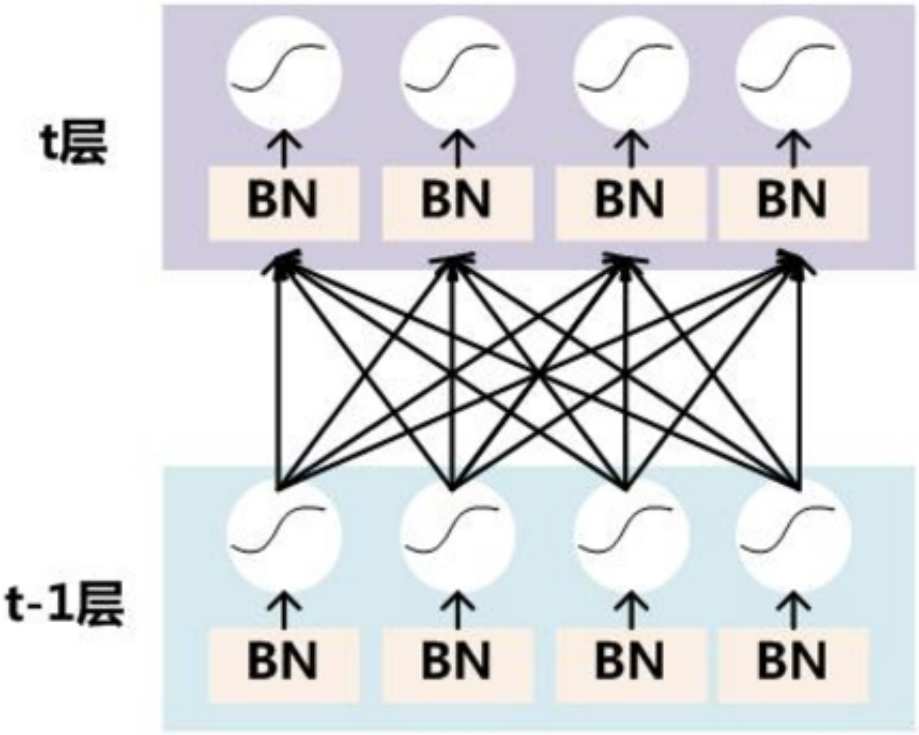
2.2 算法过程

假设对于一个深层神经网络来说，其中两层结构如下：



要对每个隐藏神经元的激活值做BN，可以想象成每个隐层又加上了一层BN操作层，它位于

X=WY+B激活值获得之后，非线性函数变换之前，其图示如下：



对Mini-Batch SGD来说，一次训练过程里面包含m个训练实例，其具体BN操作就是对于隐层内每个神经元的激活值来说，进行如下变换：

$$\hat{x}^{(k)} = \frac{x^{(k)} - E\left[x^{(k)}\right]}{\sqrt{var\left[x^{(k)}\right]}}$$

要注意，这里t层某个神经元的$x(k)$不是指原始输入，就是说不是$t-1$层每个神经元的输出，而是t曾这个神经元的激活$x = WU + B$，这里的U才是$t-1$层神经元的输出。
还有一点，上述公式中用到了均值和方差，在理想情况下均值和方差是针对整个数据集的，但显然这是不现实的，因此，作者做了简化，用一个Batch的均值和方差作为对整个数据集均值和方差的估计。
这个变换就是：某个神经元对应的原始的激活x减去Mini-Batch内m个激活x求得的均值$E(x)$并除以求得的方差$Var(x)$来进行转换。

上文说过经过这个变换后某个神经元的激活x形成了均值为0，方差为1的正态分布，目的是把值往后续要进行的非线性变换的线性区拉动，增大梯度，增强反向传播信息流行性，加快训练收敛速度。但是这样会导致网络表达能力下降，为了防止这一点，每个神经元增加两个调节参数（scale和shift），这俩个参数是通过训练来学习的，用来对变换后的激活反变换，使得网络表达能力增强，即对变换后的激活进行如下的scle和shift操作，这其实是变换的反操作：

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

其整个算法流程如下：

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
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.

2.3 推理过程

BN在训练的时候可以根据Mini-Batch数据里可以得到的统计量，那就想其他办法来获得这个统计量，就是均值和方差。可以用从所有训练实例中获得的统计量来代替Mini-Batch里面m个训练实例获得的均值和方差统计量，因为本来就打算用全局的统计量，知识因为计算量等太大所以才会用Mini-Batch这种简化方式的，那么在推理的时候直接用全局统计量即可。

决定了获得统计量的数据范围，那么接下来的问题就是如何获得均值和方差的问题。很简单，因为每次做Mini-Batch训练时，都会有那个Mini-Batch里m个训练实例获得的均值和方差，现在要全局统计量，只要把每个Mini-batch的均值和方差统计量记住，然后对这些均值和方差求其对应的数学期望即可得出全局统计量，即：

$$E[x] \leftarrow E_\beta\left[\mu_\beta\right]$$

$$Var[x] \leftarrow \frac{m}{m-1}E_\beta\left[\sigma_\beta^2\right]$$

有了均值和方差，每个隐藏神经元也已经有对应训练好的Scaling参数和Shift参数，就可以在推

导的时候对每个神经元的激活数据计算BN进行变换了，在推理过程中进行BN采取如下方式：

$$y = \frac{\gamma}{\sqrt{Var\left[x\right] + \epsilon}} \cdot x + \left(\beta - \frac{\gamma E\left[x\right]}{\sqrt{Var\left[x\right] + \epsilon}} \right)$$

这个公式其实和训练时

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

是等价的，通过简单的合并计算推导就可以得出这个结论。在实际运行时，按照这种变体形式可以减少计算量，因为对每一个隐节点来说：$\frac{\gamma}{\sqrt{Var[x]+\epsilon}}$和$\frac{\gamma E[x]}{\sqrt{Var[x]+\epsilon}}$都是固定值，这样两个值可以实现算好存起来，在推理的时候直接用就行了，比原始的公式每一步骤都少了出发的运算过程，乍一看也没少多少计算量，但是如果隐层节点个数多的话节省的计算量就比较多了。

2.4 参数训练

以上是对算法原理的讲述，在反向传导的时候，我们需要求最终的损失函数对γ和β两个参数的导数，还要求损失函数对Wx+b中x的导数，一遍使误差继续向后传播。几个主要的公式如下，主要用到了链式法则。

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

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

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

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

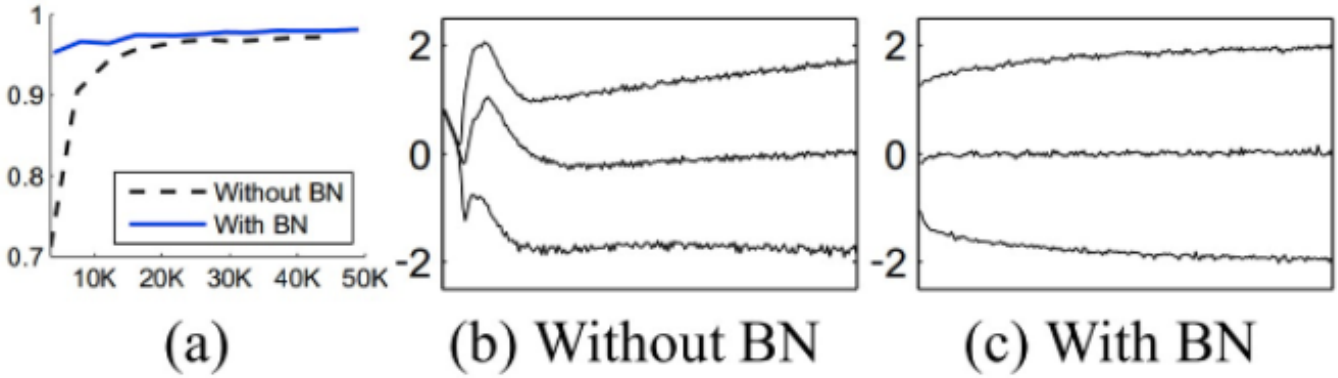
$$\frac{\partial \ell}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i} \cdot \widehat{x}_i$$

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

三、Experiments

作者在文章中也做了很多实验对比，这里简要说明两个：

下图a说明，BN可以加速训练。图b和c分别展示了训练过程中输入数据分布的变化情况。



下表是一个实验结果的对比，需要注意的是在使用BN的过程中，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，但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。

Model	Steps to 72.2%	Max accuracy
Inception	$31.0 \cdot 10^6$	72.2%
BN-Baseline	$13.3 \cdot 10^6$	72.7%
BN-x5	$2.1 \cdot 10^6$	73.0%
BN-x30	$2.7 \cdot 10^6$	74.8%
BN-x5-Sigmoid		69.8%

四、算法优势

论文中罗列了Batch Normalization的很多作用，一一列举如下：

- 1）可以使用很高的学习率。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
- 2）移除或使用较低的dropout。dropout是常用的防止overfitting的方法，而导致overfitting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，overfitting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。

说完BN的优势，自然可以知道什么时候用BN比较好。例如，在神经网络训练时遇到收敛速度很慢，或梯度爆炸等无法训练的状况时可以尝试BN来解决。另外，在一般使用情况下也可以加入BN来加快训练速度，提高模型精度。