问题

BN在深度网络训练过程中是非常好用的trick，在笔试中也很常考，而之前只是大概知道它的作用，很多细节并不清楚，因此希望用这篇文章彻底解决揭开BN的面纱。

BN层的由来与概念

讲解BN之前，我们需要了解BN是怎么被提出的。在机器学习领域，数据分布是很重要的概念。如果训练集和测试集的分布很不相同，那么在训练集上训练好的模型，在测试集上应该不奏效（比如用ImageNet训练的分类网络去在灰度医学图像上finetune再测试，效果应该不好）。对于神经网络来说，如果每一层的数据分布都不一样，后一层的网络则需要去学习适应前一层的数据分布，这相当于去做了domain的adaptation，无疑增加了训练难度，尤其是网络越来越深的情况。

实际上，确实如此，不同层的输出的分布是有差异的。BN的那篇论文中指出，不同层的数据分布会往激活函数的上限或者下限偏移。论文称这种偏移为**internal Covariate Shift**，internal指的是网络内部。神经网络一旦训练起来，那么参数就要发生更新，除了输入层的数据外(因为输入层数据，我们已经人为的为每个样本归一化)，后面网络每一层的输入数据分布是一直在发生变化的，因为在训练的时候，前面层训练参数的更新将导致后面层输入数据分布的变化。以网络第二层为例：网络的第二层输入，是由第一层的参数和input计算得到的，而第一层的参数在整个训练过程中一直在变化，因此必然会引起后面每一层输入数据分布的改变, 第一层输出变化了，势必会引起第二层输入分布的改变，模型拟合的效果就会变差，也会影响模型收敛的速度（例如我原本的参数是拟合分布A的，然后下一轮更新的时候，样本都是来自分布B的，对于这组参数来说，这些样本就会很陌生）

BN就是为了解决偏移的，解决的方式也很简单，就是让每一层的分布都normalize到标准高斯分布。（BN是根据划分数据的集合去做Normalization，不同的划分方式也就出现了不同的Normalization，如**GN，LN，IN**）

BN核心公式

$$
\begin{aligned}
Input: B=\{x_{1\dots m}\};\gamma,\beta \quad (\text{这两个是可以训练的参数}) \\
Output:\{y_i=BN_{\gamma,\beta}(x_i)\} \\
\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m}x_i \\
\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i-\mu_B)^2 \\
\tilde{x}_i=\frac{x_i-\mu_B}{\sqrt{\sigma_B^2+\varepsilon}} \quad (\text{分母加}\varepsilon\text{是为了防止方差为}0) \\
y_i=\gamma\tilde{x}_i+\beta
\end{aligned}
\tag{1}
$$

对上述公式的解释： B 即一个**batch**中的数据，先计算 B 的均值与方差，之后将 B 集合的均值、方差变换为0、1即标准正态分布，最后将 B 中的每个元素乘以 γ 再加上 β 然后输出， γ 和 β 是可训练的参数，这两个参数是BN层的精髓所在，为什么这么说呢？

和卷积层，激活层，全连接层一样，BN层也是属于网络中的一层。我们前面提到了，前面的层引起了数据分布的变化，这时候可能有一种思路是说：在每一层输入的时候，再加一个预处理就好。比如归一化到均值为0，方差为1，然后再输入进行学习。基本思路是这样的，然而实际上没有这么简单，如果我们只是使用简单的归一化方式：

$$
\tilde{x}_i=\frac{x_i-\mu_B}{\sqrt{\sigma_B^2+\varepsilon}} \tag{2}
$$

对某一层的输入数据做归一化，然后送入网络的下一层，这样是会影响到本层网络所学习的特征的，比如网络中学习到的数据本来大部分分布在0的右边，经过RELU激活函数以后大部分会被激活，如果直接强制归一化，那么就会有大多数的数据无法激活了，这样学习到的特征不就被破坏掉了么？论文中对上面的方法做了一些改进：**变换重构**，引入了可以学习的参数 γ 和 β，这就是算法的关键之处（这两个希腊字母就是要学习的）。

$$y_i = \gamma \tilde{x}_i + \beta \tag{3}$$

每个batch的每个通道都有这样的一对参数：（看完后面应该就可以理解这句话了）

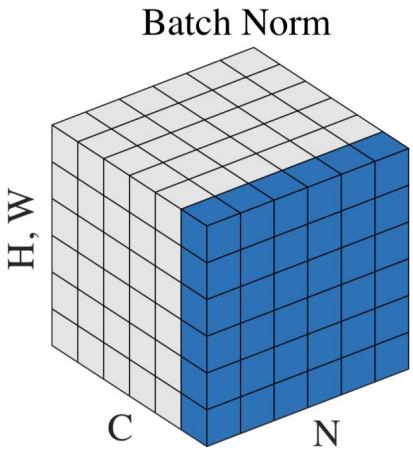
$$\gamma = \sqrt{\sigma_B^2} \quad , \quad \beta = \mu_B \tag{4}$$

这样的时候可以恢复出原始的某一层学习到的特征的，因此我们引入这个可以学习的参数使得我们的网络可以恢复出原始网络所要学习的特征分布。

我们在一些源码中，可以看到带有BN的卷积层，bias设置为False，就是因为即便卷积之后加上了Bias，在BN中也是要减去的，所以加Bias带来的非线性就被BN一定程度上抵消了。

BN中的均值与方差通过哪些维度计算得到

神经网络中传递的张量数据，其维度通常记为[N, H, W, C]，其中N是batch_size，H、W是行、列，C是通道数。那么上式中BN的输入集合 B 就是下图中蓝色的部分。



均值的计算，就是在一个批次内，将每个通道中的数字单独加起来，再除以 $N * H * W$ 。举个栗子：该批次内有十张图片，每张图片有三个通道RGB，每张图片的高宽是 H、W 那么R通道的均值就是计算这十张图片R通道的像素数值总和再除以 $10 * H * W$ ，其他通道类似，方差的计算也类似。

可训练参数γ 和 β 的维度等于张量的通道数，在上述栗子中，RGB三个通道分别需要一个γ 和 β，所以他们的维度为３。

训练与推理时BN中的均值和方差分别是多少

正确的答案是：

训练时：均值、方差分别是该批次内数据相应维度的均值与方差。

推理时：均值来说直接计算所有训练时batch的 μ_B 的平均值，而方差采用训练时每个batch的 σ_B^2 的无偏估计，公式如下：

$$\begin{aligned} E[x] &\leftarrow E_B[\mu_B] \\ Var[x] &\leftarrow \frac{m}{m-1} E_B[\sigma_B^2] \end{aligned} \tag{5}$$

但在实际实现中，如果训练几百万个Batch，那么是不是要将其均值方差全部储存，最后推理时再计算他们的均值作为推理时的均值和方差？这样显然太过笨拙，占用内存随着训练次数不断上升。为了避免该问题，后面代码实现部分使用了**滑动平均**，储存固定个数Batch的均值和方差，不断迭代更新推理时需要的 $E[x]$ 和 $Var[x]$ 。

为了证明准确性，贴上原论文中的公式（这个图其实我都看不懂.....符号好乱）：

Input: Network N with trainable parameters Θ;
subset of activations $\{x^{(k)}\}_{k=1}^{K}$

Output: Batch-normalized network for inference, $N_{\text{BN}}^{\text{inf}}$

1: $N_{\text{BN}}^{\text{tr}} \leftarrow N$ // Training BN network
2: **for** $k = 1 \ldots K$ **do**
3: Add transformation $y^{(k)} = \text{BN}_{\gamma^{(k)},\beta^{(k)}}(x^{(k)})$ to $N_{\text{BN}}^{\text{tr}}$ (Alg. 1)
4: Modify each layer in $N_{\text{BN}}^{\text{tr}}$ with input $x^{(k)}$ to take $y^{(k)}$ instead
5: **end for**
6: Train $N_{\text{BN}}^{\text{tr}}$ to optimize the parameters $\Theta \cup \{\gamma^{(k)}, \beta^{(k)}\}_{k=1}^{K}$
7: $N_{\text{BN}}^{\text{inf}} \leftarrow N_{\text{BN}}^{\text{tr}}$ // Inference BN network with frozen // parameters
8: **for** $k = 1 \ldots K$ **do**
9: // For clarity, $x \equiv x^{(k)}, \gamma \equiv \gamma^{(k)}, \mu_{\mathcal{B}} \equiv \mu_{\mathcal{B}}^{(k)}$, etc.
10: Process multiple training mini-batches $\mathcal{B}$, each of size m, and average over them:

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

11: In $N_{\text{BN}}^{\text{inf}}$, replace the transform $y = \text{BN}_{\gamma,\beta}(x)$ with $y = \frac{\gamma}{\sqrt{\text{Var}[x]+\epsilon}} \cdot x + \left(\beta - \frac{\gamma\,\text{E}[x]}{\sqrt{\text{Var}[x]+\epsilon}}\right)$
12: **end for**

如上图第11行所示：最后测试阶段，BN采用的公式是：

$$y = \frac{\gamma}{\sqrt{Var[x] + \varepsilon}} * x + (\beta - \frac{\gamma E[x]}{\sqrt{Var[x] + \varepsilon}}) \tag{6}$$

测试阶段的 γ 和 β 是在网络训练阶段已经学习好了的，直接加载进来计算即可。

BN的好处

1. **防止网络梯度消失**：这个要结合sigmoid函数进行理解
2. **加速训练，也允许更大的学习率**：输出分布向着激活函数的上下限偏移，带来的问题就是梯度的降低，（比如说激活函数是sigmoid），通过normalization，数据在一个合适的分布空间，经过激活函数，仍然得到不错的梯度。梯度好了自然加速训练。
3. **降低参数初始化敏感**：以往模型需要设置一个不错的初始化才适合训练，加了BN就不用管这些了，现在初始化方法中随便选择一个用，训练得到的模型就能收敛。
4. **提高网络泛化能力防止过拟合**：所以有了BN层，可以不再使用L2正则化和dropout。可以理解为在训练中，BN的使用使得一个mini-batch中的所有样本都被关联在了一起，因此网络不会从某一个训练样本中生成确定的结果。
5. **可以把训练数据彻底打乱**（防止每批训练的时候，某一个样本都经常被挑选到，文献说这个可以提高1%的精度）。

代码实现BN层

完整代码见参考资料3

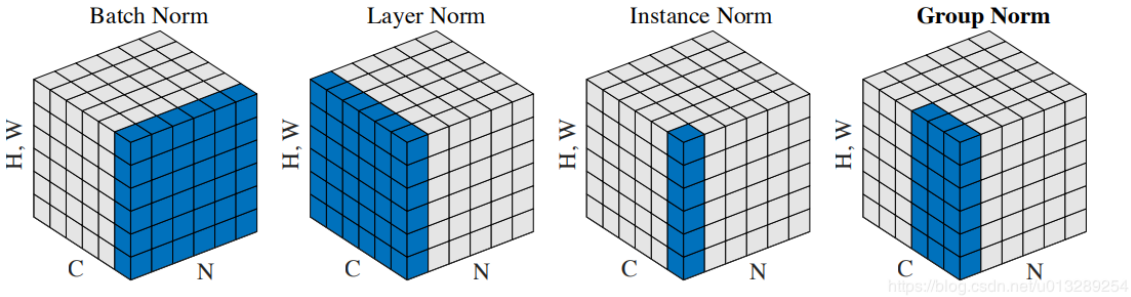
```
def batch_norm(is_training, X, gamma, beta, moving_mean, moving_var, eps, momentum):
    # 判断当前模式是训练模式还是推理模式
    if not is_training:
        # 如果是在推理模式下，直接使用传入的移动平均所得的均值和方差
        X_hat = (X - moving_mean) / torch.sqrt(moving_var + eps)
    else:
        assert len(X.shape) in (2, 4)
        if len(X.shape) == 2:
            # 使用全连接层的情况，计算特征维上的均值和方差
            mean = X.mean(dim=0)
            var = ((X - mean) ** 2).mean(dim=0)
        else:
            # 使用二维卷积层的情况，计算通道维上（axis=1）的均值和方差。这里我们需要保持X的形状以便后面可以做广播运算
            # torch.Tensor 高维矩阵的表示： （nSample）x C x H x W，所以对C维度外的维度求均值
            mean = X.mean(dim=0, keepdim=True).mean(dim=2, keepdim=True).mean(dim=3, keepdim=True)
            var = ((X - mean) ** 2).mean(dim=0, keepdim=True).mean(dim=2, keepdim=True).mean(dim=3, keepdim=True)
        # 训练模式下用当前的均值和方差做标准化
        X_hat = (X - mean) / torch.sqrt(var + eps)
        # 更新移动平均的均值和方差
        moving_mean = momentum * moving_mean + (1.0 - momentum) * mean
        moving_var = momentum * moving_var + (1.0 - momentum) * var
    Y = gamma * X_hat + beta  # 拉伸和偏移（变换重构）
    return Y, moving_mean, moving_var

class BatchNorm(nn.Module):
    def __init__(self, num_features, num_dims):  # num_features就是通道数
        super(BatchNorm, self).__init__()
        if num_dims == 2:
            shape = (1, num_features)
        else:
            shape = (1, num_features, 1, 1)
        # 参与求梯度和迭代的拉伸和偏移参数，分别初始化成0和1
```

```
        self.gamma = nn.Parameter(torch.ones(shape))
        self.beta = nn.Parameter(torch.zeros(shape))
        # 不参与求梯度和迭代的变量，全在内存上初始化成0
        self.moving_mean = torch.zeros(shape)
        self.moving_var = torch.zeros(shape)

    def forward(self, X):
        # 如果X不在内存上，将moving_mean和moving_var复制到X所在显存上
        if self.moving_mean.device != X.device:
            self.moving_mean = self.moving_mean.to(X.device)
            self.moving_var = self.moving_var.to(X.device)
        # 保存更新过的moving_mean和moving_var, Module实例的traning属性默认为true, 调用.eval()后设成false
        Y, self.moving_mean, self.moving_var = batch_norm(self.training,
            X, self.gamma, self.beta, self.moving_mean,
            self.moving_var, eps=1e-5, momentum=0.9)
        return Y

```

问题延伸

当batch size越小，BN的表现效果也越不好，因为计算过程中所得到的均值和方差不能代表全局

其实深度学习中有挺多种归一化的方法，除BN外，还有LN、IN、GN和SN四种，其他四种大致了解下就行了，大同小异，这里推荐篇博客：[深度学习中的五种归一化（BN、LN、IN、GN和SN）方法简介]



参考资料

[1、六问透彻理解BN(Batch Normalization）]
[2、神经网络之BN层]
[3、BN层pytorch实现]
[4、BatchNorm的个人解读和Pytorch中BN的源码解析]
[5、对于BN层的理解]