

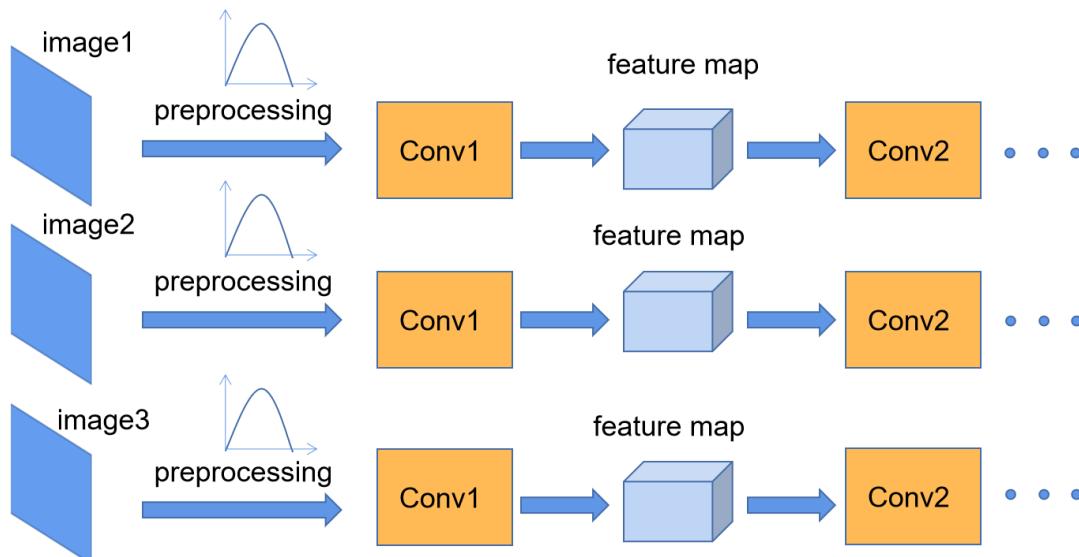
Batch Normalization详解以及pytorch实验

Batch Normalization是google团队在2015年论文《Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift》提出的。通过该方法能够加速网络的收敛并提升准确率。在网上虽然已经有很多相关文章，但基本都是摆上论文中的公式泛泛而谈，bn真正是如何运作的很少有提及。本文主要分为以下几个部分：

- (1) BN的原理
- (2) 使用pytorch验证本文的观点
- (3) 使用BN需要注意的地方 (BN没用好就是个坑)

1. Batch Normalization原理

我们在图像预处理过程中通常会对图像进行标准化处理，这样能够加速网络的收敛，如下图所示，对于Conv1来说输入的就是满足某一分布的特征矩阵，但对于Conv2而言输入的feature map就不一定满足某一分布规律了（注意这里所说满足某一分布规律并不是指某一个feature map的数据要满足分布规律，理论上是指整个训练样本集所对应feature map的数据要满足分布规律）。而我们Batch Normalization的目的就是使我们的feature map满足均值为0，方差为1的分布规律。



https://blog.csdn.net/qq_37541097

看到这里应该还是蒙的，不要慌，喝口水，慢慢来。下面是从原论文中截取的原话，注意标黄的部分：

3 Normalization via Mini-Batch Statistics

Since the full whitening of each layer's inputs is costly and not everywhere differentiable, we make two necessary simplifications. The first is that instead of whitening the features in layer inputs and outputs jointly, we will normalize each scalar feature independently, by making it have the mean of zero and the variance of 1. For a layer with d -dimensional input $\mathbf{x} = (x^{(1)} \dots x^{(d)})$, we will normalize each dimension

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

https://blog.csdn.net/qq_37541097

“对于一个拥有 d 维的输入 \mathbf{x} ，我们将对它的每一个维度进行标准化处理。”假设我们输入的 \mathbf{x} 是RGB三通道的彩色图像，那么这里的 d 就是输入图像的channels即 $d=3$ ，其中就代表我们的R通道所对应的特征矩阵，依此类推。标准化处理也就是分别对我们的R通道，G通道，B通道进行处理。上面的公式不用看，原文提供了更加详细的计算公式：

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

Parameters to be learned: γ, β

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

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

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

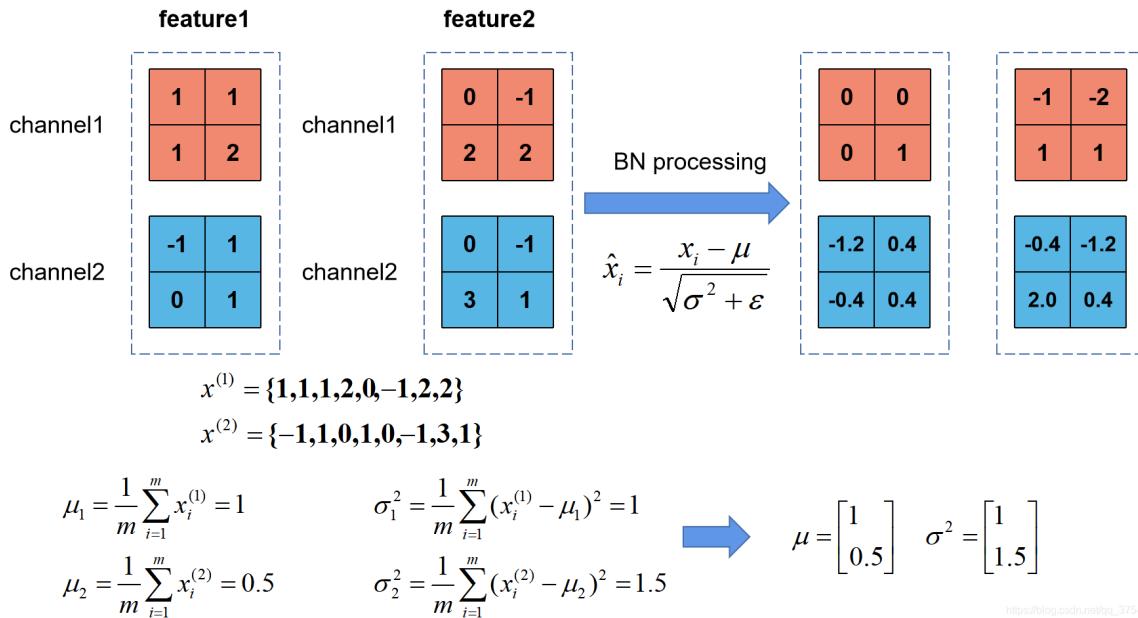
$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{normalize}$$

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

https://blog.csdn.net/qq_37541097

我们刚刚有说让feature map满足某一分布规律，理论上是指整个训练样本集所对应feature map的数据要满足分布规律，也就是说要计算出整个训练集的feature map然后在进行标准化处理，对于一个大型的数据集明显是不可能的，所以论文中说的是Batch Normalization，也就是我们计算一个Batch数据的feature map然后在进行标准化（batch越大越接近整个数据集的分布，效果越好）。我们根据上图的公式可以知道代表着我们计算的feature map每个维度（channel）的均值，注意是一个向量不是一个值，向量的每一个元素代表着一个维度（channel）的均值。代表着我们计算的feature map每个维度

（channel）的标准差，注意是一个向量不是一个值，向量的每一个元素代表着一个维度（channel）的方差，然后根据和计算标准化处理后得到的值。下图给出了一个计算均值和方差的示例：



上图展示了一个batch size为2（两张图片）的Batch Normalization的计算过程，假设feature1、feature2分别是由image1、image2经过一系列卷积池化后得到的特征矩阵，feature的channel为2，那么代表该batch的所有feature的channel1的数据，同理代表该batch的所有feature的channel2的数据。然后分别计算和的均值与方差，得到我们的和两个向量。然后在根据标准差计算公式分别计算每个channel的值（公式中的是一个很小的常量，防止分母为零的情况）。在我们训练网络的过程中，我们是通过一个batch一个batch的数据进行训练的，但是我们在预测过程中通常都是输入一张图片进行预测，此时batch size为1，如果在通过上述方法计算均值和方差就没有意义了。所以我们在训练过程中要去不断的计算每个batch的均值和方差，并使用移动平均(moving average)的方法记录统计的均值和方差，在我们训练完后我们可以近似认为我们所统计的均值和方差就等于我们整个训练集的均值和方差。然后在我们验证以及预测过程中，就使用我们统计得到的均值和方差进行标准化处理。

细心的同学会发现，在原论文公式中不是还有，两个参数吗？是的，是用来调整数值分布的方差大小，是用来调节数值均值的位置。这两个参数是在反向传播过程中学习得到的，的默认值是1，的默认值是0。

2. 使用pytorch进行试验

你以为你都懂了？不一定哦。刚刚说了在我们训练过程中，均值和方差是通过计算当前批次数据得到的记为和，而我们的验证以及预测过程中所使用的均值方差是一个统计量记为和。和的具体更新策略如下，其中momentum默认取0.1：

$$\mu_{statistic+1} = (1 - momentum) * \mu_{statistic} + momentum * \mu_{now}$$

$$\sigma_{statistic+1}^2 = (1 - momentum) * \sigma_{statistic}^2 + momentum * \sigma_{now}^2$$

这里要注意一下，在pytorch中对当前批次feature进行bn处理时所使用的是总体标准差，计算公式如下：

$$\sigma_{now}^2 = \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{now})^2$$

在更新统计量时采用的是样本标准差，计算公式如下：

$$\sigma_{now}^2 = \frac{1}{m-1} \sum_{i=1}^m (x_i - \mu_{now})^2$$

下面是我使用pytorch做的测试，代码如下：

- (1) bn_process函数是自定义的bn处理方法验证是否和使用官方bn处理方法结果一致。在bn_process中计算输入batch数据的每个维度（这里的维度是channel维度）的均值和标准差（标准差等于方差开平方），然后通过计算得到的均值和总体标准差对feature每个维度进行标准化，然后使用均值和样本标准差更新统计均值和标准差。
- (2) 初始化统计均值是一个元素为0的向量，元素个数等于channel深度；初始化统计方差是一个元素为1的向量，元素个数等于channel深度，初始化=1，=0。

```

import numpy as np
import torch.nn as nn
import torch

def bn_process(feature, mean, var):
    feature_shape = feature.shape
    for i in range(feature_shape[1]):
        # [batch, channel, height, width]
        feature_t = feature[:, i, :, :]
        mean_t = feature_t.mean()
        # 总体标准差
        std_t1 = feature_t.std()
        # 样本标准差
        std_t2 = feature_t.std(ddof=1)

        # bn process
        feature[:, i, :, :] = (feature[:, i, :, :] - mean_t) / std_t1
        # update calculating mean and var
        mean[i] = mean[i]*0.9 + mean_t*0.1
        var[i] = var[i]*0.9 + (std_t2**2)*0.1
    print(feature)

# 随机生成一个batch为2, channel为2, height=width=2的特征向量
# [batch, channel, height, width]
feature1 = torch.randn(2, 2, 2, 2)
# 初始化统计均值和方差
calculate_mean = [0.0, 0.0]
calculate_var = [1.0, 1.0]
# print(feature1.numpy())

# 注意要使用copy()深拷贝
bn_process(feature1.numpy().copy(), calculate_mean, calculate_var)

bn = nn.BatchNorm2d(2)
output = bn(feature1)
print(output)

```

首先我在最后设置了一个断点进行调试，查看下官方bn对feature处理后得到的统计均值和方差。我们可以发现官方提供的bn的running_mean和running_var和我们自己计算的calculate_mean和calculate_var是一模一样的（只是精度不同）。

通过bn_process得到的输出

```
[[[[ 0.8514141 -1.3069117 ]
   [-0.13720813  0.6130738 ]]

 [[ 1.0037296  1.3679621 ]
   [ 0.7113629 -0.3972618 ]]

 [[[ 1.1189657  0.8377364 ]
   [-0.19351949 -1.7835506 ]]

 [[[-1.1295396  0.73421925]
   [-1.2434236 -1.0470493 ]]]]
```

通过官方bn得到的输出

```
tensor([[[[ 0.8514, -1.3069],
          [-0.1372,  0.6131]],

         [[ 1.0037,  1.3680],
          [ 0.7114, -0.3973]]],

        [[[ 1.1190,  0.8377],
          [-0.1935, -1.7835]],

         [[-1.1295,  0.7342],
          [-1.2434, -1.0470]]]], grad_fn=<NativeBatchNormBackward>)
```

https://blog.csdn.net/qq_37541097

然后我们打印出通过自定义bn_process函数得到的输出以及使用官方bn处理得到输出，明显结果是一样的（只是精度不同）：

```
emp\test.py
34     bn = nn.BatchNorm2d(2)  bn: BatchNorm2d(2, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
35     output = bn(feature1)  output: tensor([[[[ 0.8514, -1.3069], [-0.1372,  0.6131]]], ...
36     print(output)
```

Variables

- bn = (BatchNorm2d) BatchNorm2d(2, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
 - affine = (bool) True
 - bias = (Parameter: 2) Parameter containing: tensor([0., 0.], requires_grad=True)
 - dump_patches = (bool) False
 - eps = (float) 1e-05
 - momentum = (float) 0.1
 - num_batches_tracked = (Tensor) tensor(1)
 - num_features = (int) 2
 - running_mean = (Tensor: 2) tensor([0.0255, 0.0386])
 - running_var = (Tensor: 2) tensor([0.9694, 0.9689])
 - track_running_stats = (bool) True
 - training = (bool) True
 - weight = (Parameter: 2) Parameter containing: tensor([1., 1.], requires_grad=True)
- Protected Attributes
- calculate_mean = (list: 2) [0.025544130802154542, 0.038603505492210394]
- calculate_var = (list: 2) [0.9693954348968763, 0.9689165265913872]
- feature1 = (Tensor: 2) tensor([[[0.9189, -0.7630], [0.1485, 0.7332]], [[1.1655, 1.4483], [0.9384, 0.0775]]], [[1.0037, 1.3680], [0.7114, -0.3973]]], [[1.1190, 0.8377], [-0.1935, -1.7835]]], ...)
- output = (Tensor: 2) tensor([[[[0.8514, -1.3069], [-0.1372, 0.6131]]], ...]]]
- Special Variables

https://blog.csdn.net/qq_37541097

3. 使用BN时需要注意的问题

(1) 训练时要将traning参数设置为True，在验证时将trainning参数设置为False。在pytorch中可通过创建模型的model.train()和model.eval()方法控制。

(2) batch size尽可能设置大点，设置小后表现可能很糟糕，设置的越大求的均值和方差越接近整个训练集的均值和方差。

(3) 建议将bn层放在卷积层（Conv）和激活层（例如Relu）之间，且卷积层不要使用偏置bias，因为没有用，参考下图推理，即使使用了偏置bias求出的结果也是一样的

- $\mu(x^b) = \mu(x) + b$
- $y_i = \frac{x_i - \mu(x)}{\sqrt{\sigma^2(x)}}$
- $y_i^b = \frac{x_i^b - \mu(x^b)}{\sqrt{\sigma^2(x^b)}}$
- $x_i^b = x_i + b$

→

- $\sigma^2(x^b) = \frac{1}{m} \sum_{i=1}^m [x_i^b - \mu(x^b)]^2$
 $= \frac{1}{m} \sum_{i=1}^m [x_i + b - \mu(x) - b]^2$
 $= \frac{1}{m} \sum_{i=1}^m [x_i - \mu(x)]^2$
 $= \sigma^2(x)$
- $y_i^b = \frac{x_i^b - \mu(x^b)}{\sqrt{\sigma^2(x^b)}} = \frac{x_i + b - \mu(x) - b}{\sqrt{\sigma^2(x)}} = y_i$

https://log.csdn.net/qq_37541097

最后给出李宏毅老师关于batch normalization的视频讲解：

<https://www.bilibili.com/video/av9770302?p=10>