论文：《Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift》

参考链接：

https://blog.csdn.net/hjimce/article/details/50866313#commentBox

https://blog.csdn.net/qq_25737169/article/details/79048516

https://www.cnblogs.com/guoyaohua/p/8724433.html

1.背景意义

BN算法的强大之处在于：

1. 可以选择较大的初始学习率，提高训练速度。以前需要慢慢调整学习率，甚至在网络训练到一半的时候，还需要想着学习率进一步调小的比例选择多少合适，现在可以采用初始很大的学习率，然后学习率衰减速度也很大，因为这个算法收敛很快。
2. 不用理会过拟合中dropout、l2正则项参数的选择问题，采用BN后，可以移除这两项的参数，或者可以选择更小的l2正则约束参数，因为BN具有提高网络泛化能力的特性
3. 不需使用局部响应归一化层（Alexnet用到的方法），因为BN本身就是一个归一化网络层
4. 可以把训练数据彻底打乱（防止每批训练的时候，某一个样本经常被挑选到，文献说可以提高1%的精度）

先来思考一个问题：我们知道在神经网络训练开始前，都要对输入数据做一个归一化处理，那么具体为什么需要归一化呢？归一化后有什么好处呢？原因在于神经网络学习过程本质

就是为了学习数据分布，一旦训练数据与测试数据的分布不同，那么网络的泛化能力也大大降低；另外一方面，一旦每批训练数据的分布各不相同(batch 梯度下降)，那么网络就要在每次迭代都去学习适应不同的分布，这样将会大大降低网络的训练速度，这也正是为什么我们需要对数据都要做一个归一化预处理的原因。

我们知道网络一旦train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。我们把网络中间层**在训练过程中，数据分布的改变称之为**：“**Internal Covariate Shift**”。Paper所提出的算法，就是要解决在训练过程中，中间层数据分布发生改变的情况，于是就有了Batch Normalization，这个牛逼算法的诞生。

2.初识BN

2.1 BN概述

就像激活函数层、卷积层、全连接层、池化层一样，BN也是网络的一层。该算法本质是：在网络的每一层输入的时候，又插入了一个归一化层，也就是先做一个归一化处理，然后再进入网络的下一层。但是文献的归一化层不像想象的那么简单，它是一个可学习、有参数的网络层。既然说到数据预处理，就先来复习一下最强的预处理方法：白化。

2.2 预处理操作

说到神经网络输入数据预处理，最好的算法莫过于白化预处理。然而白化计算量太大了，很不划算，还有就是白化不是处处可微的，所以在深度学习中，其实很少用到白化。经过白化预处理后，数据满足条件：a、特征之间的相关性降低，这个就相当于pca；b、数据均值、标准差归一化，也就是使得每一维特征均值为0，标准差为1。如果数据特征维数比较大，要进行PCA，也就是实现白化的第1个要求，是需要计算特征向量，计算量非常大，于是为了简化计算，作者忽略了第1个要求，仅仅使用了下面的公式进行预处理，也就是近似白化预处理：

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

需要注意的是，我们训练过程中采用batch 随机梯度下降，上面的E(xk)指的是每一批训练数据神经元xk的平均值；然后分母就是每一批数据神经元xk激活度的一个标准差了。

3.BN算法实现

3.1 BN算法概述

其实如果是仅仅使用上面的归一化公式，对网络某一层A的输出数据做归一化，然后送入网络下一层B，这样是会影响到本层网络A所学习到的特征的。打个比方，比如我网络中间某一层学习到特征数据本身就分布在S型激活函数的两侧，你强制把它给我归一化处理、标准差也限制在了1，把数据变换成分布于s函数的中间部分，这样就相当于我这一层网络所学习到的特征

分布被你搞坏了，这可怎么办？于是文献使出了一招惊天地泣鬼神的招式：**变换重构，引入了可学习参数γ、β，这就是算法关键之处：**

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

每一个神经元xk都会有一对这样的参数γ、β。这样其实当：

$$\gamma^{(k)} = \sqrt{\mathrm{Var}[x^{(k)}]}$$

$$\beta^{(k)} = \mathrm{E}[x^{(k)}]$$

是可以恢复出原始的某一层所学到的特征的。因此我们引入了这个可学习重构参数γ、β，让我们的网络可以学习恢复出原始网络所要学习的特征分布。最后Batch Normalization网络层的前向传导过程公式就是：

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

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

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

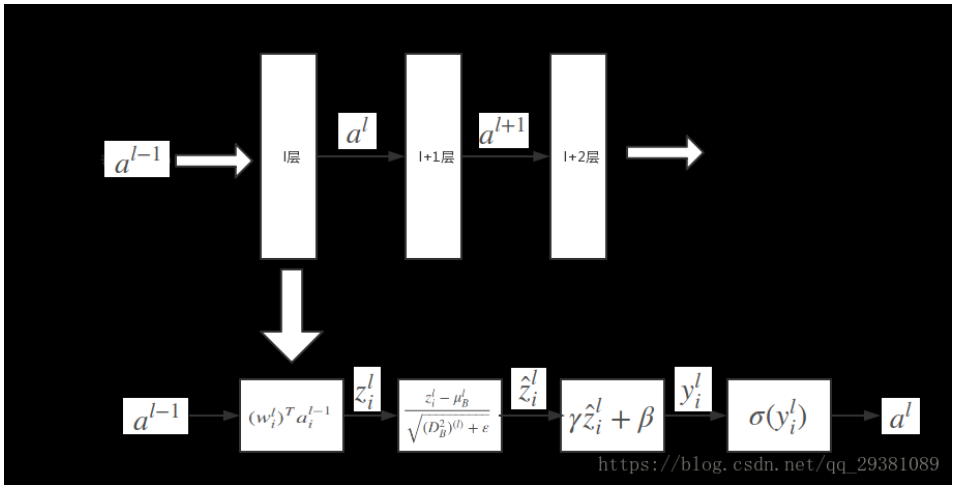
$$y_i \leftarrow \gamma\hat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

上面的公式中m指的是mini-batch size。

3.2 前向传播

3.1所说的过程，详细解释如下：

注意以下字母皆为矩阵形式,i为batch里面的索引



设有minimini_batchbatch的batch_size为m,注意这里的i为batch里面的索引,x和a为特征矩阵

由于当前层的输入等于上一层的输出,那么设第l层输出为alal,则第l层的输入：

$x_i^l = a_i^{l-1}$, i =0,1,2...m

前向传播过程:

1.全连接则乘权重,卷积层则对x卷积

$$z_i^l = (w_i^l)^T a_i^{l-1} \quad //偏重放后面加 \tag{1.1}$$

2.计算batch_size个z的均值

$$\mu_B^l = \frac{1}{m}\sum_{i=1}^{m} z_i^l \tag{1.2}$$

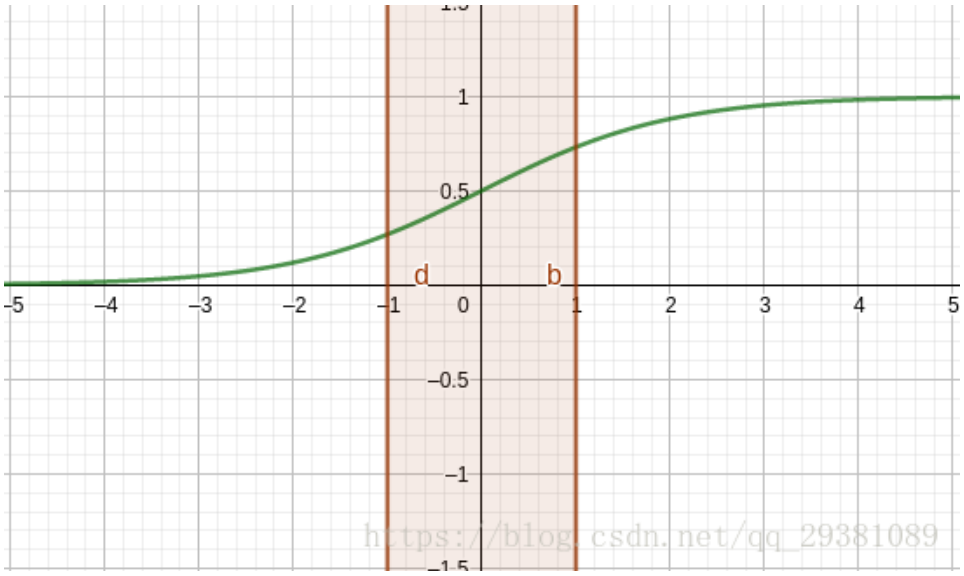
3)计算batch_size个z的方差:

$$(D_B^2)^{(l)} = \frac{1}{m}\sum_{i=1}^{m}(z_i^l - \mu_B^l)^2 \tag{1.3}$$

3)将batch_size个z,归一化成均值为0,方差为1的分布:

$$BN = \hat{z}_i^l = \frac{z_i^l - \mu_B^l}{\sqrt{(D_B^2)^{(l)} + \varepsilon}} \tag{1.4}$$

经过这个操作后数据就被分布在0为圆心,1为半径的范围内了,这样以上 问题就被成功解决了问题



4)放缩和迁移:

$$y_i^l = \hat{z}_i^l + \beta \tag{1.5}$$

这步的作用在于,以逻辑函数为例,经过1.3后,数据主要分布在线性区域,非线性表达能力会受到影响,所以通过对数据放大或缩小和迁移来进入非线性区域范围

5)激活,输出

$$a_i^l = \sigma(y_i^l) \tag{1.6}$$

3.2 源码实现

```
m=K.mean(X,axis=-1,keepdims=True) #计算均值
std=K.std(X,axis=-1,keepdims=True) #计算标准差
X_normed=(X-m)/(std+self.epsilon) #归一化
out=self.gamma * X_normed + self.beta #重构变换
```

上面的x是一个二维矩阵，对于源码的实现就几行代码而已，轻轻松松。

```
def Batchnorm_simple_for_train(x, gamma, beta, bn_param):
"""
param:x    : 输入数据，设shape(B,L)
param:gama : 缩放因子  γ
param:beta : 平移因子  β
param:bn_param   : batchnorm所需要的一些参数
      eps      : 接近0的数，防止分母出现0
      momentum : 动量参数，一般为0.9， 0.99， 0.999
      running_mean ： 滑动平均的方式计算新的均值，训练时计算，为测试数据做准备
      running_var  : 滑动平均的方式计算新的方差，训练时计算，为测试数据做准备
"""
      running_mean = bn_param['running_mean']  #shape = [B]
   running_var = bn_param['running_var']    #shape = [B]
      results = 0. # 建立一个新的变量

      x_mean=x.mean(axis=0)  # 计算x的均值
   x_var=x.var(axis=0)    # 计算方差
   x_normalized=(x-x_mean)/np.sqrt(x_var+eps)       # 归一化
   results = gamma * x_normalized + beta            # 缩放平移

   running_mean = momentum * running_mean + (1 - momentum) * x_mean
   running_var = momentum * running_var + (1 - momentum) * x_var

   #记录新的值
   bn_param['running_mean'] = running_mean
   bn_param['running_var'] = running_var

      return results , bn_param
```

首先计算均值和方差，然后归一化，然后缩放和平移，完事！但是这是在训练中完成的任务，每次训练给一个批量，然后计算批量的均值方差，但是在测试的时候可不是这样，测试的时候每次只输入一张图片，这怎么计算批量的均值和方差，于是，就有了代码中下面两行，在训练的时候实现计算好mean var测试的时候直接拿来用就可以了，不用计算均值和方差。

```
running_mean = momentum * running_mean + (1 - momentum) * x_mean
running_var = momentum * running_var + (1 - momentum) * x_var
```

所以，测试的时候是这样的：

```
def Batchnorm_simple_for_test(x, gamma, beta, bn_param):
"""
param:x    : 输入数据，设shape(B,L)
param:gama : 缩放因子  γ
param:beta : 平移因子  β
param:bn_param   : batchnorm所需要的一些参数
      eps      : 接近0的数，防止分母出现0
      momentum : 动量参数，一般为0.9， 0.99， 0.999
      running_mean ： 滑动平均的方式计算新的均值，训练时计算，为测试数据做准备
      running_var  : 滑动平均的方式计算新的方差，训练时计算，为测试数据做准备
"""
      running_mean = bn_param['running_mean']  #shape = [B]
   running_var = bn_param['running_var']    #shape = [B]
      results = 0. # 建立一个新的变量

   x_normalized=(x-running_mean )/np.sqrt(running_var +eps)       # 归一化
   results = gamma * x_normalized + beta            # 缩放平移

      return results , bn_param
```

3.3 实战使用

1.可能学完了上面的算法，你只是知道它的一个训练过程，一个网络一旦训练完了，就没有了min-batch这个概念了。测试阶段我们一般只输入一个测试样本，看看结果而已。因此测试样本，前向传导的时候，上面的均值u、标准差σ 要哪里来？其实网络一旦训练完毕，参数都是固定的，这个时候即使是每批训练样本进入网络，那么BN层计算的均值u、和标准差都是固定不变的。我们可以采用这些数值来作为测试样本所需要的均值、标准差，于是最后测试阶段的u和σ 计算公式如下：

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

上面简单理解就是：对于均值来说直接计算所有batch u值的平均值；然后对于标准偏差采用每个batch σB的无偏估计。最后测试阶段，BN的使用公式就是：

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

2. 根据文献说，BN可以应用于一个神经网络的任何神经元上。文献主要是把BN变换，置于网络激活函数层的前面。在没有采用BN的时候，激活函数层是这样的：

z=g(Wu+b)

也就是我们希望一个激活函数，比如s型函数s(x)的自变量x是经过BN处理后的结果。因此前向传导的计算公式就应该是：

z=g(BN(Wu+b))

其实因为偏置参数b经过BN层后其实是没有用的，最后也会被均值归一化，当然BN层后面还有个β参数作为偏置项，所以b这个参数就可以不用了。因此最后把BN层+激活函数层就变成了：

z=g(BN(Wu))

4. BN在CNN中的使用

通过上面的学习，我们知道BN层是对每个神经元做归一化处理，甚至只需要对某一个神经元进行归一化，而不是对一整层网络的神经元进行归一化。既然BN是对单个神经元的计算，那么在CNN中卷积层上要怎么搞？假如某一层卷积层有6个特征图，每个特征图的大小是100*100，这样就相当于这一层网络有6*100*100个神经元，如果采用BN，就会有6*100*100个参数γ、β，这样岂不是太恐怖了。因此卷积层上的BN使用，其实也是使用了**类似权值共享的策略，把一整张特征图当做一个神经元进行处理。**

卷积神经网络经过卷积后得到的是一系列的特征图，如果min-batch sizes为m，那么网络某一层输入数据可以表示为四维矩阵(m,f,p,q)，m为min-batch sizes，f为特征图个数，p、q分别为特征图的宽高。在cnn中我们可以**把每个特征图看成是一个特征处理**（一个神经元），因此在使用Batch Normalization，mini-batch size 的大小就是：m*p*q，**于是对**

于每个特征图都只有一对可学习参数：γ、β。说白了吧，这就是相当于求取所有样本所对应的一个特征图的所有神经元的平均值、方差，然后对这个特征图神经元做归一化。

5.BN作用（加快收敛和防止过拟合）

Batch Normalization有两个功能，一个是可以加快训练和收敛速度，另外一个是可以防止过拟合。

BN算法是如何加快训练和收敛速度的呢？

BN算法在实际使用的时候会把特征给强制性的归到均值为0，方差为1的数学模型下。深度网络在训练的过程中，如果每层的数据分布都不一样的话，将会导致网络非常难收敛和训练，而如果能把每层的数据转换到均值为0，方差为1的状态下，**一方面，数据的分布是相同的，训练会比较容易收敛，另一方面，均值为0，方差为1的状态下，在梯度计算时会产生比较大的梯度值**，可以加快参数的训练，更直观的来说，是把**数据从饱和区直接拉到非饱和区**。更进一步，**这也可以很好的控制梯度爆炸和梯度消失现象，因为这两种现象都和梯度有关。**

BN最大的优点为允许网络使用较大的学习速率进行训练，加快网络的训练速度。

BN算法时如何防止过拟合的？

在这里摘录一段国外大神的解释：

大概意思是：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。

这句话什么意思呢？意思就是**同样一个样本的输出不再仅仅取决于样本本身，也取决于跟这个样本属于同一个mini-batch的其它样本**。同一个样本跟不同的样本组成一个mini-batch，它们的输出是不同的（仅限于训练阶段，在inference阶段是没有这种情况的）。我把这个理解成一种数据增强：**同样一个样本在超平面上被拉扯，每次拉扯的方向的大小均有不同**。不同于数据增强的是，这种拉扯是贯穿数据流过神经网络的整个过程的，**意味着神经网络每一层的输入都被数据增强处理了。**

相比于Dropout、L1、L2正则化来说，BN算法防止过拟合效果没那末明显。

6.tensorflow中

tf.layers.batch_normalization()

BN在如今的CNN结果中已经普遍应用，在tensorflow中可以通过tf.layers.batch_normalization()这个op来使用BN。该op隐藏了对BN的mean var alpha beta参数的显示申明，因此在训练和部署测试中需要特征注意正确使用BN的姿势。

1.原理

公式如下：

$$y=\gamma(x-\mu)/\sigma+\beta$$

其中x是输入，y是输出，μ是均值，σ是方差，γ和β是缩放（scale）、偏移（offset）系数。

一般来讲，这些参数都是基于channel来做的，比如输入x是一个16*32*32*128(NWHC格式)的feature map，那么上述参数都是128维的向量。其中γ和β是可有可无的，有的话，就是一个可以学习的参数（参与前向后向），没有的话，就简化成y=(x-μ)/σ。而μ和σ，在训练的时候，使用的是batch内的统计值，测试/预测的时候，采用的是训练时计算出的滑动平均值。

2.tensorflow中使用

tensorflow中batch normalization的实现主要有下面三个：

tf.nn.batch_normalization

tf.layers.batch_normalization

tf.contrib.layers.batch_norm

封装程度逐个递进，建议使用tf.layers.batch_normalization或tf.contrib.layers.batch_norm，因为在tensorflow官网的解释比较详细。我平时多使用tf.layers.batch_normalization，因此下面的步骤都是基于这个。

3.训练

训练的时候需要注意两点，(1)输入参数training=True，(2)计算loss时，要添加以下代码（即添加update_ops到最后的train_op中）。这样才能计算μ和σ的滑动平均（测试时会用到）

```
update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
with tf.control_dependencies(update_ops):
  train_op = optimizer.minimize(loss)
```

4.测试

测试时需要注意一点，输入参数training=False，其他就没了

5.预测

预测时比较特别，因为这一步一般都是从checkpoint文件中读取模型参数，然后做预测。一般来说，保存checkpoint的时候，不会把所有模型参数都保存下来，因为一些无关数据会增大模型的尺寸，常见的方法是只保存那些训练时更新的参数（可训练参数），如下：

```
var_list = tf.trainable_variables()
saver = tf.train.Saver(var_list=var_list, max_to_keep=5)
```