Batch Normalization 实验

1.问题背景

在神经网络里，深度神经网络涉及到很多层的叠加，而每一层的参数更新会导致上层的输入数据分布发生变化，通过层层叠加，高层的输入分布变化会非常剧烈，这就使得高层需要不断去重新适应底层的参数更新。为了训练好模型，我们需要非常谨慎地设定学习率，初始化权重，以及尽可能细致的参数更新策略。这种现象被称为 *Internal Covariate Shift*。

Internal Covariate Shift 的本质是输入变量的数据分布前后不一致。在神经网络中，将每一层的输入作为一个分布看待，由于底层的参数随着训练更新，导致相同的输入分布得到的输出分布改变了。而机器学习中有个很重要的假设：IID 独立同分布假设，就是假设训练数据和测试数据是满足相同分布的，这是通过训练数据获得的模型能够在测试集获得好的效果的一个基本保障。那么，细化到神经网络的每一层间，每轮训练时分布都是不一致，那么相对的训练效果就得不到保障，所以称为层间的 *covariate shift*。

随着网络层数的加深，输入分布经过多次线性非线性变化，已经改变了，但是对应的标签或者分类还是一致的，即使条件概率一致，边缘概率不同。当每个神经元输入数据不再是“独立同分布”，将会导致上层网络需要不断适应新的输入数据分布，降低学习速度；下层输入的变化可能趋于变大或者变小，导致上层落入饱和区，使得学习过早停止；每层的更新都会影响到其他层，因此每层的参数更新策略需要尽可能的谨慎。

从“怎么使得输入分布满足独立同分布”的角度出发，一种 *Batch Normalization* 的方法高效地解决了这个问题。

2. BN 数学思想和具体算法

2.1 *Batch Normalization* 的数学模型

$$Input: Values\ of\ x\ over\ a\ mini\_batch: B = \{x_1, \dots x_m\};$$

$$parameters\ to\ be\ learned: \boldsymbol{\gamma}, \boldsymbol{\beta}$$

$$Output: (y_i = BN_{\gamma,\beta}(x_i))$$

$$\mu_B = \frac{1}{m}\sum x_i \qquad \#mini-batch\ mean$$

$$\sigma_B^2 = \frac{1}{m}\sum (x_i - \mu_B)^2 \qquad \#mini-batch\ variance$$

$$\widehat{x_i} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}} \qquad \#normalize$$

$$y_i = \gamma \widehat{x_i} + \beta \equiv BN_{\gamma,\beta}(x_i) \qquad \#scale\ and\ shift$$

2.2 *Batch Normalization* 的数学理解

传统的神经网络随着输入和网络结构的复杂度增加，其内部输入逐渐发生偏移或者变动，整体分布逐渐往非线性函数的取值区间的上下两端靠近，导致反向传播时底层的梯度小，从而导致网络收敛慢。而 BN 就是通过一定的规范化手段，把逐渐向函数映射后取值区间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，以避免梯度消失问题。

但是当所有结构都使用 BN 时，神经网络的效果近似于线性函数，即多层的线性函数变化对于深层是没有意义的，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，也意味着神经网络中深度的意义就没有了。所以 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，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移动。BN 的核心思想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，又避免太靠近非线性区两头使得网络收敛速度太慢。

2.3 *Batch Normalization* 具体算法实现:

1. 获得每个 *mini_batch* 以及他们对应的统计量：均值和方差
2. 利用权重滑动平均法来更新统计量
3. 利用数学公式实现 BN 层网络
4. 前向传播，得到最终的损失函数
5. 反向传播，利用 SGD 不断修正 BN 层的两个学习参数，使得两个参数不断逼近最优解

3. 实验操作

3.1 数据集简介

本次实验采用 minist 数据集。该数据集包含 60000 个用于训练的示例和 10000 个用于测试的示例。这些数字已经过尺寸标准化并位于图像中心，图像是固定大小（28*28 像素），其值为 0 到 1 的灰度数据。

图 1.minist 图例

3.2 实验设置

本次实验采用了卷积神经网络 *CNN*。损失函数选择了 MSE 均方差收敛。*CNN* 采用的基本参数是：*learning_rate*=0.001；隐藏层神经元个数为 500；输出层神经元个数为 10；训练次数 *Epoch*=20；批分组大小为 *batch_size*=20。具体的 CNN 神经网络层之间的框架如图所示（从左向右）

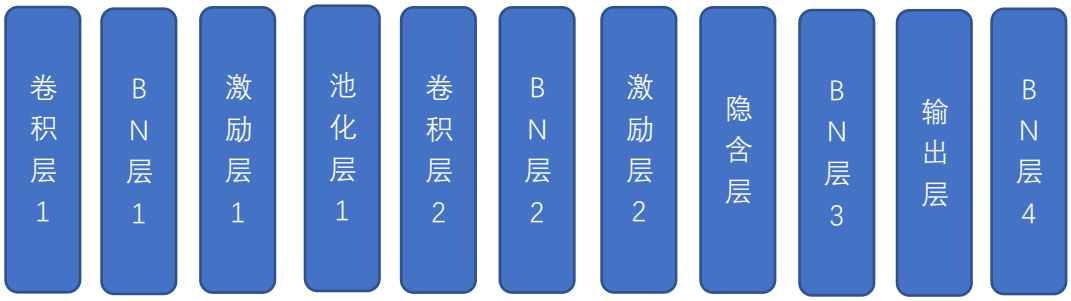

图 2 CNN 卷积神经网络的结构图

3.3 实验结果对比

通过 visdom 模块，将输出的结果可视化，得到损失函数的结果

3.3.1 没有使用 *Batch Normalization* 的网络的训练结果

```
Train Epoch: 20 [14848/60000 (25%)] Loss: 2.303385
Train Epoch: 20 [30208/60000 (50%)] Loss: 2.302585
Train Epoch: 20 [45568/60000 (75%)] Loss: 2.298289

Test set: Average loss: 2.2999, Accuracy: 1135/10000 (11%)
```

图 3 无 BN 网络的最终 loss 值

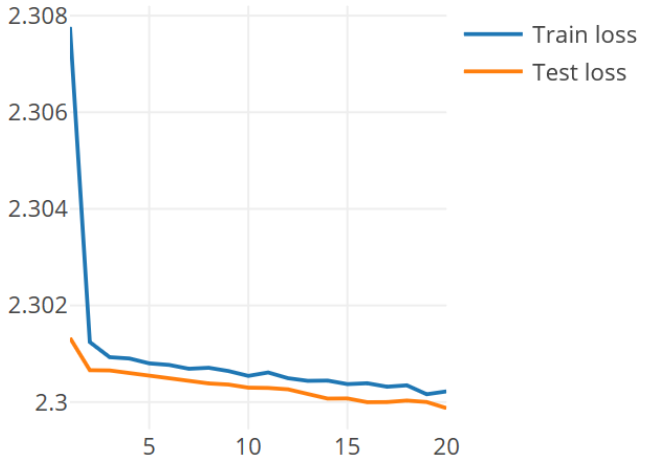

图 4 无 BN 网络的 Loss 衰减曲线

3.3.2 使用 ZJK_BN 的网络的训练结果

```
Train Epoch: 20 [14848/60000 (25%)] Loss: 0.840055
Train Epoch: 20 [30208/60000 (50%)] Loss: 0.836669
Train Epoch: 20 [45568/60000 (75%)] Loss: 0.822424

Test set: Average loss: 0.8150, Accuracy: 9210/10000 (92%)
```

图 5 带 ZJK_BN 网络的最终 Loss 值

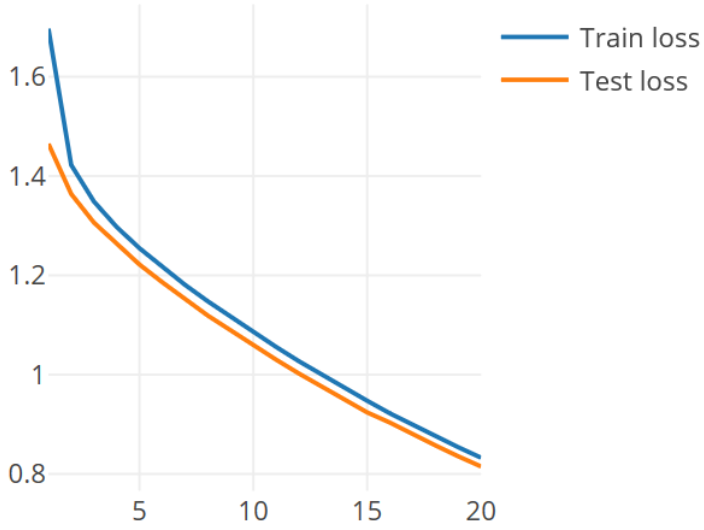

图 6 带 ZJK_BN 网络的 Loss 衰减曲线

3.3.3 使用 pytorch_BN 的网络的训练结果

```
Train Epoch: 20 [14848/60000 (25%)] Loss: 0.457489
Train Epoch: 20 [30208/60000 (50%)] Loss: 0.451923
Train Epoch: 20 [45568/60000 (75%)] Loss: 0.471015

Test set: Average loss: 0.4444, Accuracy: 9565/10000 (96%)
```

图 7 带 pytorch_BN 网络的最终 Loss 值

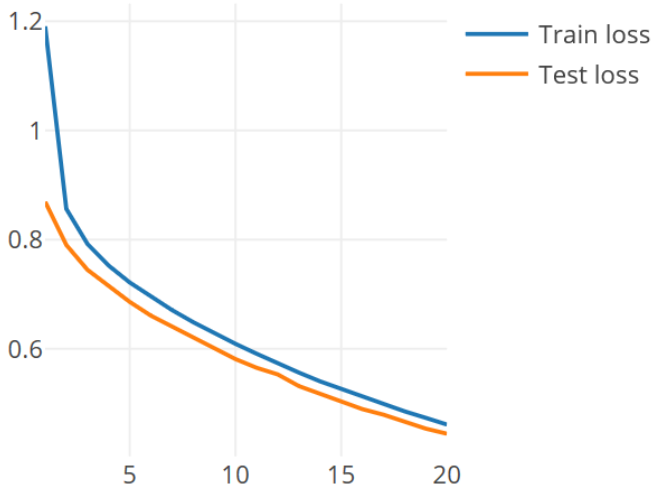

图 8 带 pytorchBN 网络的 Loss 衰减曲线

由图 3 图 4 可以看出，当不使用 BN 时，损失函数虽然也在不断减小，但是衰减速率很慢，需要很大的训练次数才能达到要求；由图 6 和图 8 可以看出，当加入 BN 网络结构后，其网络的损失函数可以迅速得到收敛，并且在次数较少时就已经达到 90%以上的准确率，可以看出 BN 的使用对于整个网络的

效率和运行速度都有很大的提升。

3.4 ZJK_BN 和 pytorchBN 的实现的对比，实验结果分析

由图对比得知，pytorch 的运行结果比 ZJK_BN 好。从准确性角度而言，同样训练 20 次，ZJKBN 得到的准确率是 92%，而 pytorch 的准确率是 96%；从损失函数的结果来看，训练 20 次后，ZJKBN 的 Loss 结果在 0.8 附近，而 pytorchBN 的 Loss 结果在 0.5 左右；从 Loss 函数的收敛速度上来看，ZJK_BN 的收敛速度没有 pytorchBN 的收敛速度快。

ZJK_BN 与 pytorch 的函数差别总结为以下两点：

1. *momentum* 参数的选择（主）

再次给出 BN 的数学公式：

$$y_i = \gamma \hat{x_i} + \beta \ ; \ \hat{x_i} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$$

从 BN 的数学公式中可以看出，每次计算时的均值和方差部分决定了输出的大小，所以每次均值和方差的更新就显得极为重要。在 ZJK_BN 的实现中，选择了权重滑动平均法来更新均值和方差：

$$moving\_mean = momentum * moving\_mean + (1 - momentum) * mean$$

momentum 本质含义是衰减率，即 *moving_mean* 的每次更新都信任之前的 mean，并且取其 *momentum* 的加权。随着批次的增加，最开始的批次的均值所占的比例越来越小，即衰减得越来越小。这种处理使得均值和方差的数值更平滑也更容易信任，过滤掉了一些干扰因子。所以 *momentum* 的选择就很重要。ZJK_BN 里面的 *momentum* 取值为 0.1 当调整 momentum 的参数后会得到更好的 loss 终值和更好的网络结构。所以 *momentum* 的参数选择是影响 ZJK_BN 和 pyBN 之间的差距的一个因素

2. 函数内部的优化差别（次）

虽然 ZJK_BN 的原理是按照 BN 的经典逻辑，但是其内部运行机制和 Pytorch 的函数运行机制还是有区别，其内部运行的优化原理也不同，会使得反向传播的梯度方向也会有所调整，也会导致 loss 的衰减速率不同。

4. 实现代码

https://github.com/junjunjun-study/pytorch_BN/

包含两个文件，myBN.py 是实现 BN 函数的代码，MNIST_with_BN.py 是实现网络代码