

人工神经网络实验报告

作业 3 • BN

周正平, 计 54, 2015011314, zhouzp15@mails.tsinghua.edu.cn

2017 年 11 月 2 日

Contents

1 实验内容	2
2 算法实现	2
2.1 模型搭建	2
2.1.1 MLP	2
2.1.2 CNN	3
2.2 Batch Normalization 实现	4
2.2.1 算法原理	5
2.2.2 MLP	6
2.2.3 CNN	7
3 实验结果	7
3.1 MLP	7
3.2 CNN	8
4 对比分析	10
4.1 MLP/CNN	10
4.2 有无 BN	11
4.2.1 MLP	11
4.2.2 CNN	12
4.3 可视化分析	13
5 测试说明	14

1 实验内容

本次实验利用 TensorFlow 实现 MLP 及 CNN，用于完成 MNIST 手写数字识别任务。

实验中所用 MLP、CNN 网络结构如下：

网络 结构

MLP $input \rightarrow Linear \rightarrow BN \rightarrow ReLU \rightarrow Linear \rightarrow loss$

CNN $input \rightarrow Conv \rightarrow BN \rightarrow ReLU \rightarrow MaxPool \rightarrow Conv \rightarrow BN \rightarrow ReLU \rightarrow MaxPool \rightarrow Linear \rightarrow loss$

总体来说，本次作业需要在给出的框架基础上，实现以下功能：

1. 模型实现：使用 TensorFlow 搭建如上结构的 MLP 及 CNN；
2. BN 优化：使用 batch normalization 将各个 batch 标准化为 $E = 0, Var = 1$ ，从而加速收敛、缓解梯度弥散/梯度爆炸。

2 算法实现

2.1 模型搭建

2.1.1 MLP

本次作业实现的 MLP 结构为 $input \rightarrow Linear \rightarrow BN \rightarrow ReLU \rightarrow Linear \rightarrow loss$ ，使用 TensorBoard 可视化如下（笔者在 ReLu 之后加入了 dropout 层）：

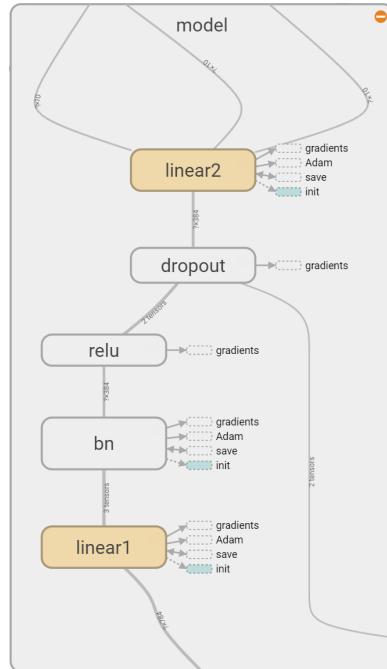


Figure 1: MLP 网络结构

其中关键步骤总结如下：

1. Linear 层

```
self.W1 = weight_variable([28*28, self.num_hidden], name='W1')
self.b1 = bias_variable([self.num_hidden], name='b1')
logits = tf.matmul(self.x_, self.W1) + self.b1
```

2. BN 层

```
logits = batch_normalization_layer(logits, is_train)
```

3. ReLu 层

```
logits = tf.nn.relu(logits)
```

4. Dropout 层

```
logits = tf.nn.dropout(logits, self.keep_prob)
```

2.1.2 CNN

本次作业实现的 CNN 结构为 $input \rightarrow Conv \rightarrow BN \rightarrow ReLU \rightarrow MaxPool \rightarrow Conv \rightarrow BN \rightarrow ReLU \rightarrow MaxPool \rightarrow Linear \rightarrow loss$, 使用 TensorBoard 可视化如下：

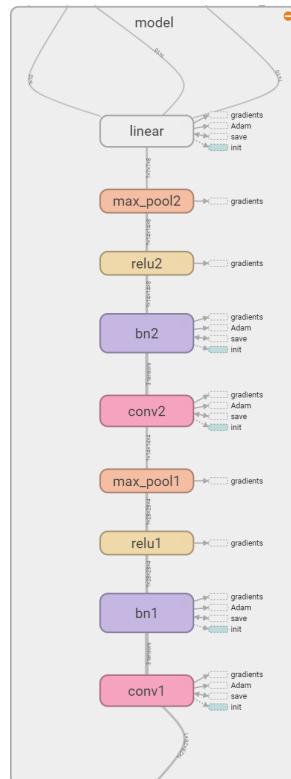


Figure 2: CNN 网络结构

其中关键步骤总结如下：

1. 卷积层

```
W = weight_variable([k, k, c_in, c_out], name='W')
b = bias_variable([c_out], name='b')
return tf.nn.conv2d(inputs, W, strides=[1, 1, 1, 1], padding='SAME') + b
```

2. BN 层

```
logits = batch_normalization_layer(logits, is_train)
```

3. ReLu 层

```
logits = tf.nn.relu(logits)
```

4. MaxPooling 层

```
logits = tf.nn.max_pool(logits, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
```

5. Linear 层

```
W_fc = weight_variable([7 * 7 * 8, 10], name='W')
b_fc = bias_variable([10], name='b')
logits = tf.reshape(logits, [-1, 7 * 7 * 8])
logits = tf.matmul(logits, W_fc) + b_fc
```

2.2 Batch Normalization 实现

由于 MLP 与 CNN 在 BN 算法逻辑上较为相似，以下使用 TensorBoard 统一给出其可视化结果：

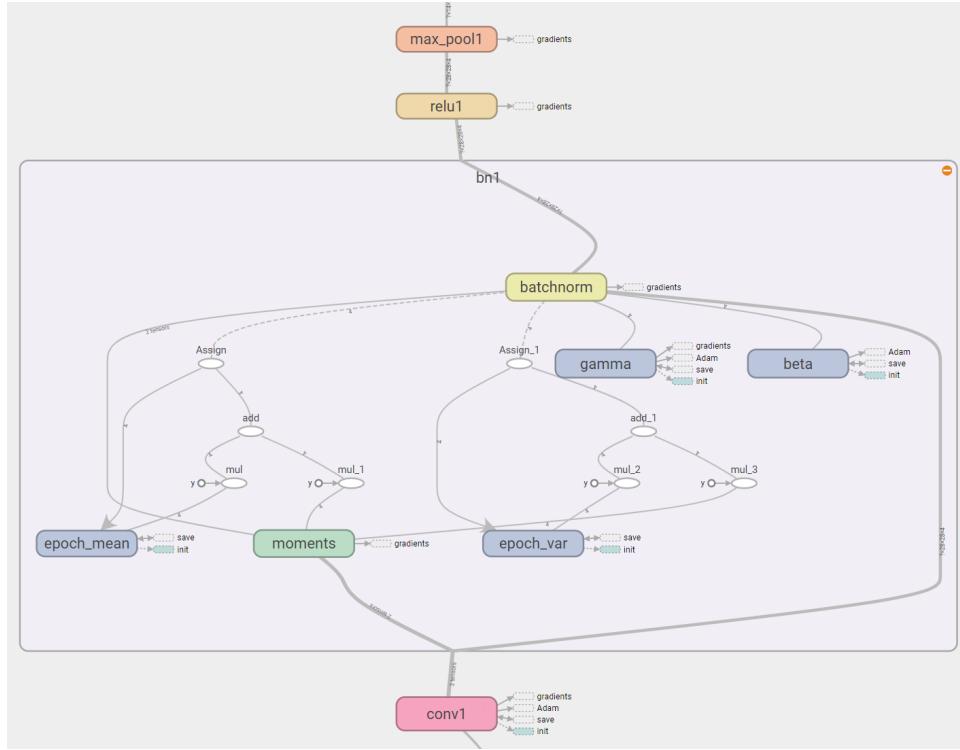


Figure 3: BN 算法可视化流程图

2.2.1 算法原理

论文 *Batch normalization: Accelerating deep network training by reducing internal covariate shift* 中对 batch normalization (下称 BN) 的描述如下:

Input:	Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots m\}$;
	Parameters to be learned: γ, β
Output:	$\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$
	$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i$ // mini-batch mean
	$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2$ // mini-batch variance
	$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$ // normalize
	$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i)$ // scale and shift

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Figure 4: BN 算法

总的来说, BN 将一个 batch 内的数据标准化为期望 $E = 0$, 方差 $Var = 1$ 。这有 3 个显著的优势:

1. 归一化数据分布

由于各个 batch 在分布上存在差异, 并且在神经网络加深的过程中, 各个层会改变输入数据的分布, 故而将数据的期望、方差归一化, 有助于减弱这种差异, 从而提高网络的泛化能力;

2. 缓解梯度弥散/爆炸

由于导数运算的链式法则, 当网络加深时, 梯度的累乘可能引起上溢/下溢: 如果各层梯度多 < 1 , 则发生梯度弥散, 收敛缓慢; 若各层梯度多 > 1 , 则发生梯度爆炸。BN 算法有效地归一化了过大或过小的梯度, 从而对梯度弥散/爆炸有较好的缓解作用;

3. 加速收敛

从直观上看, BN 算法对训练过程作了 2 点优化:

其一, BN 将整个 batch 中的数据减去期望 E , 相当于将所有数据移动到零点。由于模型中参数向量多使用 0 附近的值随机初始化, 而这些参数需逐渐收敛至数据均值附近, 故而将数据整体平移到零点有效地加速了收敛过程。

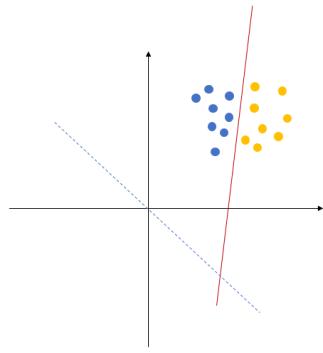


Figure 5: 原始数据

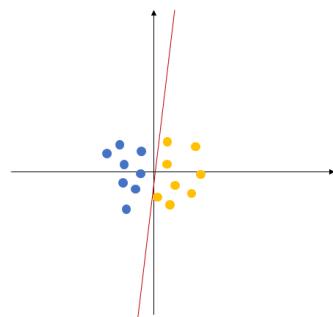


Figure 6: BN 优化后

其二，BN 将数据各个维度上的方差缩放为 1，可以使其分布更加均匀。如果数据样本分布较为狭长，各个维度上方差不均匀，分类的超平面稍有变动便偏离了正确的位置。统一各维度方差为 1 有效地解决了这一问题。

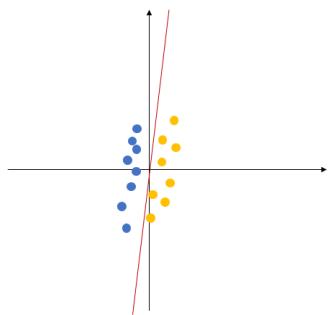


Figure 7: 原始数据

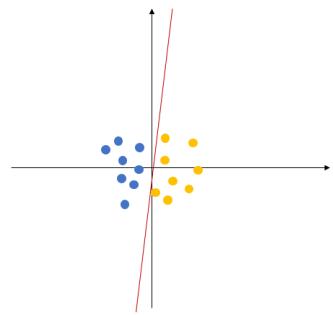


Figure 8: BN 优化后

2.2.2 MLP

结合论文中对算法的说明，代码中实现了以下逻辑：

1. Train

在训练阶段，通过 `tf.nn.moments` 函数对输入向量的 `batch_size` 这一维度取均值与方差，再将其传入 `tf.nn.batch_normalization` 函数求出结果即可（以下给出核心代码）：

```
mean, var = tf.nn.moments(inputs, [0])
return tf.nn.batch_normalization(inputs, mean, var, offset, scale, eps)
```

同时，需要保存训练过程中各个 batch 的期望和方差的平均值，用于在 Test 时使用。实验中采取 `ExponentialMovingAverage` 的方式取平均：

```
update_epoch_mean = tf.assign(epoch_mean, epoch_mean * decay + mean * (1 - decay))
update_epoch_var = tf.assign(epoch_var, epoch_var * decay + var * (1 - decay))
```

2. Test

在测试阶段，考虑测试用例单个输入的情形（即 `batch_size = 1`），则归一化后，所有维度上的输入都将是 0。为解决这一问题，在测试时，应使用训练时保存的全局期望与方差进行归一化：

```
return tf.nn.batch_normalization(inputs, epoch_mean, epoch_var, offset, scale, eps)
```

2.2.3 CNN

CNN 的 BN 算法实现与 MLP 非常类似，唯一的差异在于：在 CNN 中，各个 Feature Map (1 个通道) 作为一个整体计算均值和方差，而在 MLP 中，一张图的不同像素分别对 `batch_size` 计算均值和方差。以 $C_{in} = 32, C_{out} = 64$ 的卷积层为例，其 BN 操作示意如下图：

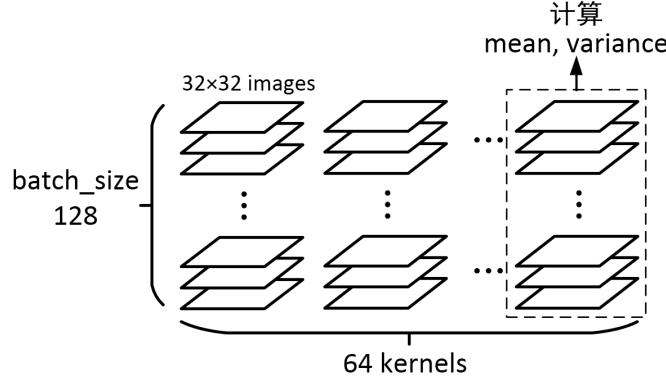


Figure 9: CNN 的 BN 示意图

具体体现到代码中，区别在于：CNN 对输入的前 3 个轴计算期望与方差，而 MLP 只涉及第 1 个轴：

```
# MLP
mean, var = tf.nn.moments(inputs, [0])

# CNN
mean, var = tf.nn.moments(inputs, [0, 1, 2])
```

3 实验结果

3.1 MLP

本次实验中经过 BN 优化的 MLP 网络，在 one-by-one test 中获得精度 98.47%。

网络超参数配置如下：

(网络结构： $input \rightarrow Linear \rightarrow BN \rightarrow ReLU \rightarrow Linear \rightarrow loss$)

超参数	取值	备注
隐层节点数	392	
learning_rate	0.001	学习率
learning_rate_decay_factor	0.9995	动态调整学习率的幅度
BN- ϵ	1e-20	BN 算法中防止除零
BN-ema_decay	0.9	BN 算法中 ExponentialMovingAverage 的 decay 参数

使用 TensorBoard 绘制训练集及验证集上的 loss-iter 曲线如下：

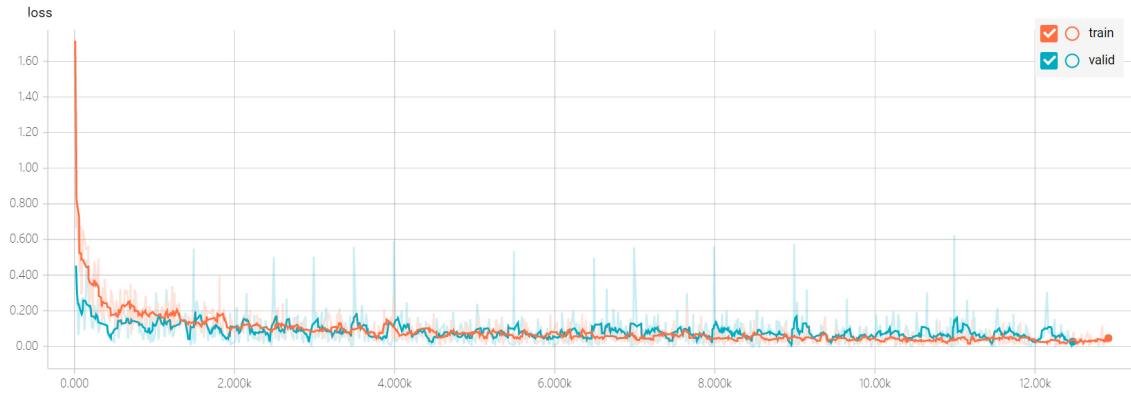


Figure 10: loss-iter 曲线

acc-iter 曲线如下：



Figure 11: acc-iter 曲线

3.2 CNN

本次实验中经过 BN 优化的 CNN 网络，在 one-by-one test 中获得精度 99.28%。

网络超参数配置如下：

(网络结构： $input \rightarrow Conv \rightarrow BN \rightarrow ReLU \rightarrow MaxPool \rightarrow Conv \rightarrow BN \rightarrow ReLU \rightarrow MaxPool \rightarrow Linear \rightarrow loss$ ， 使用 GPU 进行训练)

位置	超参数	取值
训练超参	learning_rate	0.001
	learning_rate_decay_factor	0.9995
	BN- ϵ	1e-10
	BN-ema_decay	1 - 1e-3
Conv1	C_{in}	1
	C_{out}	32
	ksize	5
MaxPool1	ksize	2
Conv2	C_{in}	32
	C_{out}	64
	ksize	7
MaxPool2	ksize	2
Linear	输入节点数	$64 \times 7 \times 7$
	输出节点数	10

使用 TensorBoard 绘制训练集及验证集上的 loss-iter 曲线如下：

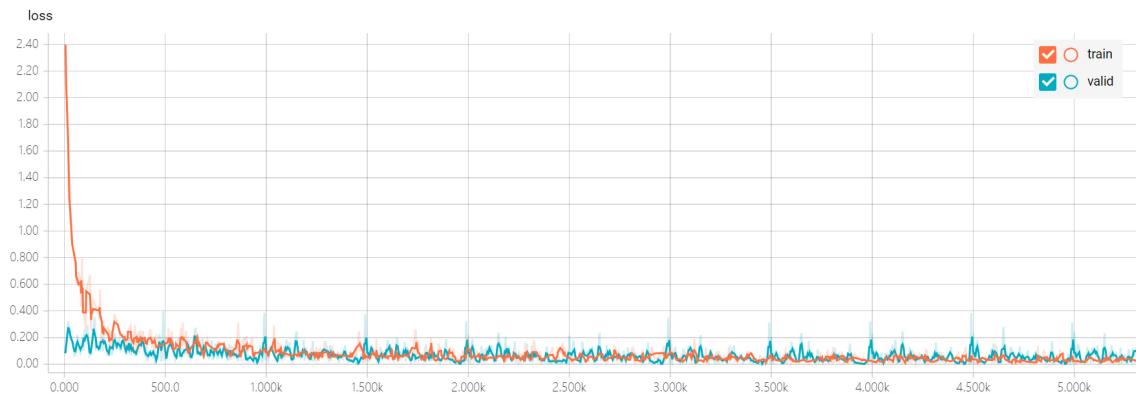


Figure 12: loss-iter 曲线

acc-iter 曲线如下：



Figure 13: acc-iter 曲线

4 对比分析

4.1 MLP/CNN

本节中对比加入 BN 优化的 MLP 与 CNN，从训练时间、收敛性、测试精度 3 个方面加以分析：

网络类型	训练时间	收敛性	测试精度
MLP	在 CPU 上约 5s/epoch	收敛较慢	98.47%
CNN	在 GPU 上约 15s/epoch	收敛较快	99.28%

使用 TensorBoard 绘制训练 loss-iter 曲线如下：

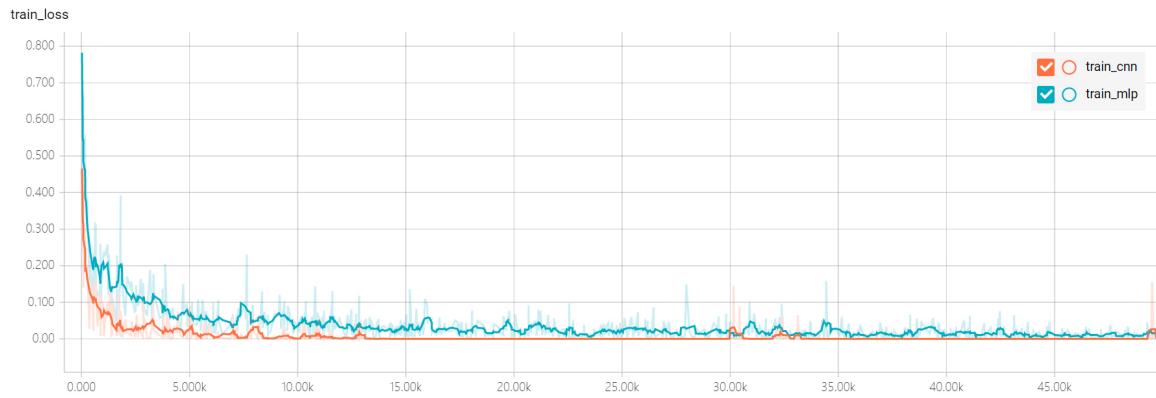


Figure 14: loss-iter 曲线

acc-iter 曲线如下：

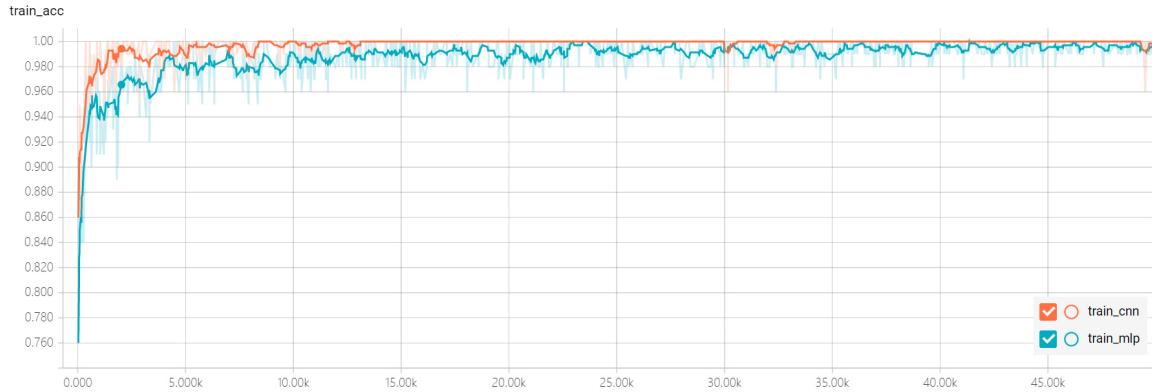


Figure 15: acc-iter 曲线

从图中可以看出，CNN 随迭代次数收敛的速度显著快于 MLP，最终测试精度也更高。然而训练 CNN 需要更多的计算资源，笔者曾尝试在 CPU 上训练上节描述的 CNN 网络，发现训练速度极慢，最后不得不改用 GPU 进行训练，获得了可以接受的训练速度。在实际应用中需考虑分类效果与计算资源之间的权衡取舍。

此外，可以注意到，CNN 在训练后期的训练精度已达到 1.0 的水平，这说明 CNN 在图像分类问题上有着强大的特征提取能力。然而这也说明网络存在过拟合，可以通过 dropout 等技术予以解决。

4.2 有无 BN

本节中分别对比加入与未加入 BN 优化的 MLP 与 CNN，从训练时间、收敛性、测试精度 3 个方面加以分析。

4.2.1 MLP

对于 MLP 网络，加入与未加入 BN 的训练效果对比如下：

是否有 BN	训练时间	收敛性	测试精度
有 BN	在 CPU 上约 5s/epoch	收敛较快	98.47%
无 BN	在 CPU 上约 5s/epoch	收敛略慢	98.40%

使用 TensorBoard 绘制训练 loss-iter 曲线如下：

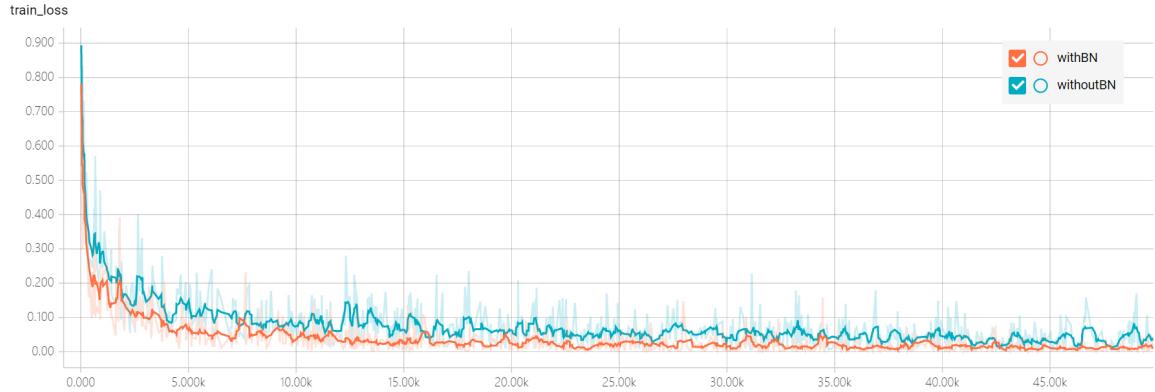


Figure 16: loss-iter 曲线

acc-iter 曲线如下：

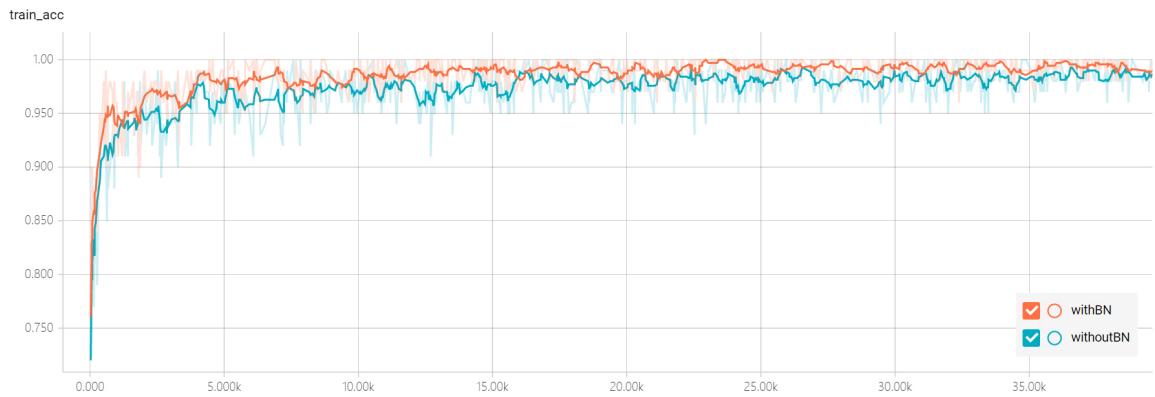


Figure 17: acc-iter 曲线

可以看出，有 BN 时的 MLP 网络相比无 BN 时，收敛更快，精度更高。这是因为 BN 算法将各个 batch 之间的数据分布归一化，根据前文的分析，这使得网络的收敛速度、测试精度均有所提高。

4.2.2 CNN

对于 CNN 网络，加入与未加入 BN 的训练效果对比如下：

是否有 BN	训练时间	收敛性	测试精度
有 BN	在 CPU 上约 15s/epoch	收敛性无显著区别	99.28%
无 BN	在 CPU 上约 15s/epoch	收敛性无显著区别	99.24%

使用 TensorBoard 绘制训练 loss-iter 曲线如下：

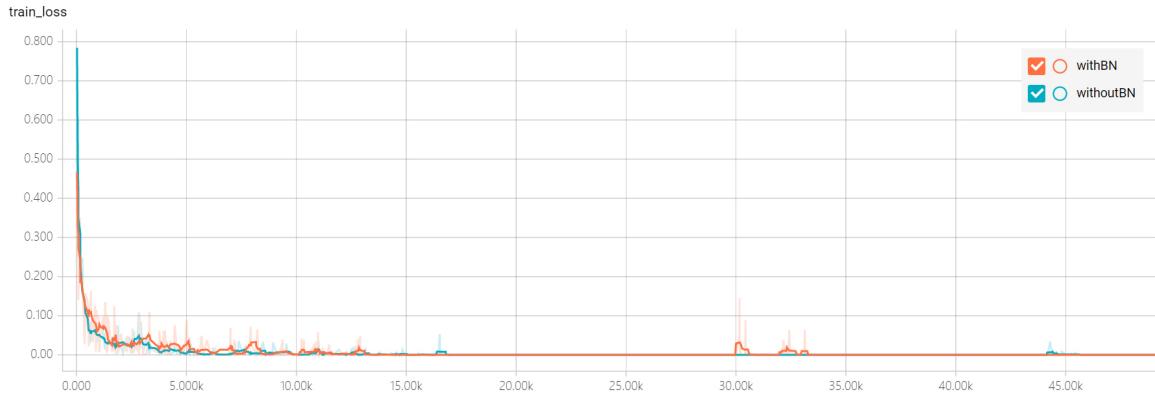


Figure 18: loss-iter 曲线

acc-iter 曲线如下：



Figure 19: loss-acc 曲线

从图中可以看出，在笔者的实验中，对于 CNN 网络而言，是否加入 BN 对收敛速度影响不大，加入 BN 时精度稍有提高。

4.3 可视化分析

以 MLP 网络为例，使用 TensorBoard 绘制第一个 Linear 层之后的数据分布直方图，以便直观分析 BN 的作用：

（图中，横轴表示向量中元素数值的大小，纵轴表示时间。叠加的直方图反映了随时间推移，不同 batch 之间的数据分布变化情况）

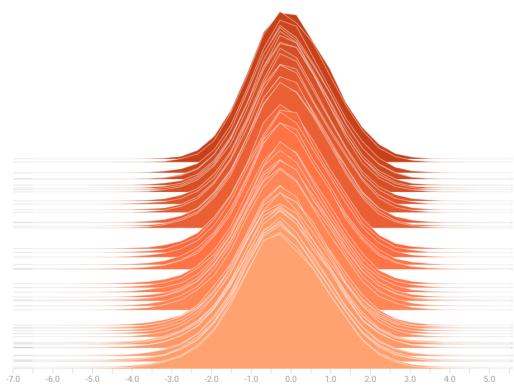
model/output_after_linear
train

Figure 20: 有 BN(train)

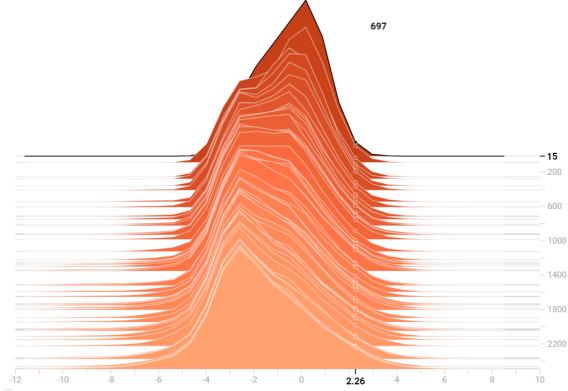
model/output_after_linear
train

Figure 21: 无 BN(train)

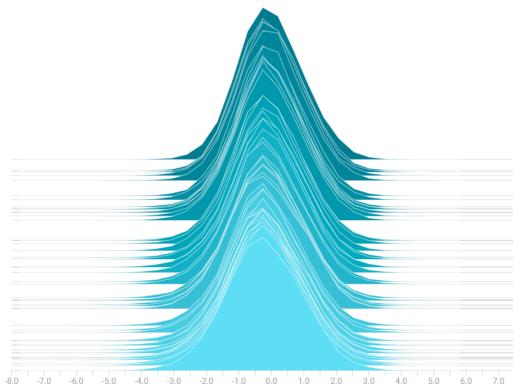
model/output_after_linear
valid

Figure 22: 有 BN(valid)

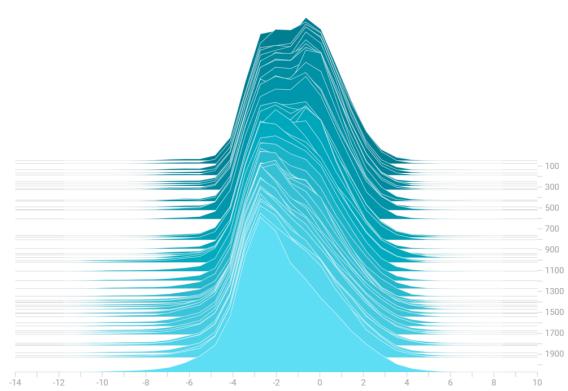
model/output_after_linear
valid

Figure 23: 无 BN(valid)

从上图中可以看出，BN 使得线性层之后的输出不随着时间推移，或是 batch 之间的差异而发生大的变化，较为稳定。这是因为 BN 归一化了各个 batch 的期望与方差，从而提高了网络的泛化能力，也使得收敛速度、测试精度得到提高。

5 测试说明

本节叙述 one-by-one test 的结果、过程中出现的问题及其解决方案。

在开始时，笔者在 train 和 test 时使用一致的公式进行计算。在训练过程中，test 准确率稳定在与 train 接近的水平（95% 以上）：

```
Epoch 99 of 100 took 4.83893108368s
learning rate: 0.000994514
training loss: 0.0390911390866
validation loss: 0.0838553162898
validation accuracy: 0.981717180724
best epoch: 73
best validation accuracy: 0.983232335009
```

```
test loss: 0.0887230747341
test accuracy: 0.980909099483
Epoch 100 of 100 took 5.18315696716s
learning rate: 0.000994514
training loss: 0.0410657270224
validation loss: 0.0907345924777
validation accuracy: 0.982121221947
best epoch: 73
best validation accuracy: 0.983232335009
test loss: 0.0887230747341
test accuracy: 0.980909099483
```

然而，当笔者使用命令行参数 `--is_train=False` 来进行 one-by-one test 时，却发现精度很低，只有 10% 上下。分析原因，这是因为当输入仅有 1 个样本时，BN 算法会将输入向量的所有维度全部置零，导致最终的结果与输入无关，准确率极低。为了解决这一问题，在实现 BN 算法时，需按照 train/test 分类讨论，在 test 时使用 train 时所有 batch 的期望、方差的平均值进行计算。

经过修改之后（详见算法实现一节），笔者实现的 MLP 达到了 98.47% 的测试精度，CNN 达到了 99.28% 的测试精度。

References

- [1] Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift
- [2] <http://www.jianshu.com/p/0312e04e4e83>
- [3] <http://blog.csdn.net/hjimce/article/details/50866313>
- [4] <http://blog.csdn.net/woolseyyyy/article/details/74712946>
- [5] <https://r2rt.com/implementing-batch-normalization-in-tensorflow.html>
- [6] https://github.com/aymericdamien/TensorFlow-Examples/blob/master/examples/4_Utils/tensorboard_basic.py