Batch Normalization 学习报告

高鑫辰

程序地址：https://github.com/dearflypig/batchnormalization/tree/master

一、相关理论介绍

1、当采用较深层数的网络进行训练时：

*后层网络需要不停调整来适应输入数据分布的变化，导致网络学习速度的降低；
*当在神经网络中采用饱和激活函数时（例如 sigmoid、tanh 激活函数），网络的训练过程容易陷入梯度饱和区，梯度会变得很小甚至接近于 0，参数的更新速度会减慢；

2、Batch Normalization

*在对输入数据进行激活函数的处理之前，对每个特征进行 normalization，让每个特征都有均值为 0，方差为 1 的分布，让数据位于激活函数的敏感区域；
*计算方法：

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

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

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

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

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

标准化步骤

引入两个可学习的参数γ和β，让神经网络自己去使用和修改这两个参数

二、实现功能：

编写 BN 层的代码，并利用一个简单的数据进行检测，确保自己编写的 BN 层能实现对数据的标准化处理：

1、使用 pytorch 提供的 BatchNorm1d 模块

```
#使用pytorch的BatchNorm1d模块
data = np.array([[1, 2],[3, 4],[5, 6]]).astype(np.float32)
data = torch.from_numpy(data)
print('data: ',data)
bn=nn.BatchNorm1d(num_features=2)
out=bn(data)
print('out: ',out)
```

```
data:  tensor([[1., 2.],
        [3., 4.],
        [5., 6.]])
out:  tensor([[-1.2247, -1.2247],
        [ 0.0000,  0.0000],
        [ 1.2247,  1.2247]], grad_fn=<NativeBatchNormBackward>)
```

2、使用自己编写的 BN 层

```
class MyBN(nn.Module):
    def __init__(self, num_features):
        super(MyBN, self).__init__()
        self.gamma = nn.Parameter(torch.ones((num_features,)),requires_grad=True)
        self.beta = nn.Parameter(torch.zeros((num_features,)),requires_grad=True)
        self.moving_mean = torch.zeros((num_features,))
        self.moving_var = torch.zeros((num_features,))

    def forward(self, x):
        momentum: float
        momentum=0.01
        mean = torch.mean(x,dim=0)
        var=torch.mean(((x-mean)**2),dim=0)
        x_ = (x - mean) / torch.sqrt(var + eps)
        self.moving_mean = momentum * self.moving_mean + (1.0 - momentum) * mean
        self.moving_var = momentum * self.moving_var + (1.0 - momentum) * var
        out = self.gamma * x_ + self.beta
        return out
```

```
#使用自己编写的MyBN层
data = np.array([[1, 2],[3, 4],[5, 6]]).astype(np.float32)
data = torch.from_numpy(data)
print('data: ',data)
my_bn = MyBN(num_features=2)
out_ = my_bn(data)
print('out_:',out_)
```

```
data:  tensor([[1., 2.],
        [3., 4.],
        [5., 6.]])
out_: tensor([[-1.2247, -1.2247],
        [ 0.0000,  0.0000],
        [ 1.2247,  1.2247]], grad_fn=<AddBackward0>)
```

***两者的输出是相同的，说明自己编写的 BN 层可以实现对数据的标准化**

三、体会 BN 层的效果

***数据：**

制作伪数据来模拟一个回归的任务，看在加 BN 层前后神经网络模型的预测能力，采用的数据为 y=a*x^2+b，对 y 数据加上噪声点来更加真实的展示；

***网络模型：**

构造 3 个四层的神经网络，分别是不加 BN 层的网络、使用 pytorch 提供的 BatchNorm1d 模块的网络，使用自己编写的 MyBN 层的网络，对训练数据进行学习，并在测试数据上观察拟合的曲线。

***其他：**

激活函数：tanh

Epoch：10

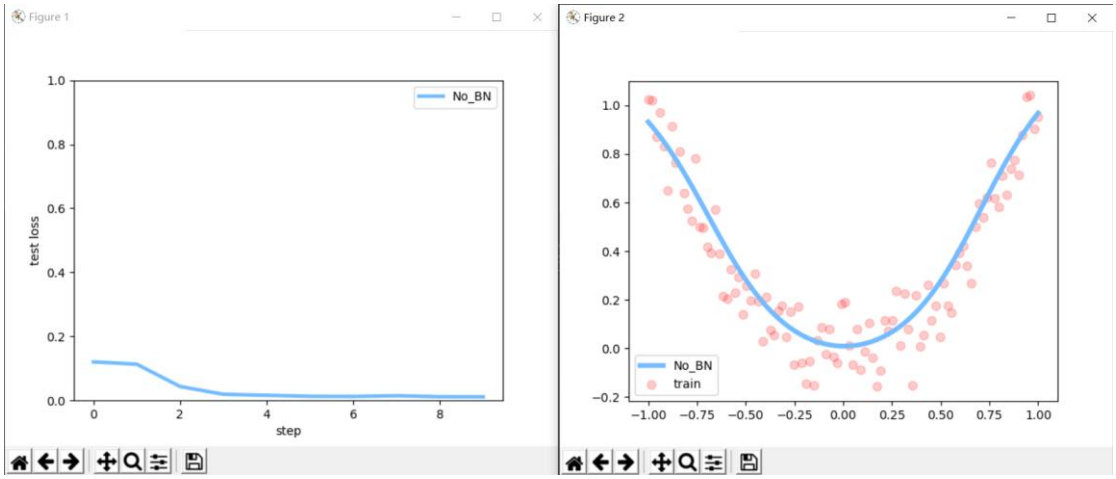
（一） 当输入数据 x 的范围在（-1,1）区间，对比三种模型的效果：

*数据：

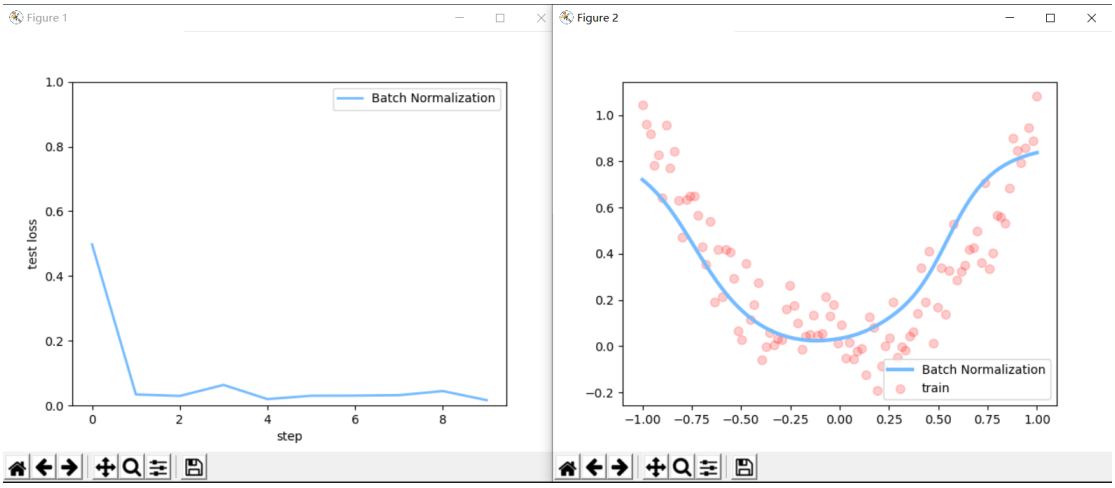
```
#train data
x = np.linspace(-1, 1, 1000)[:, np.newaxis]
y = np.square(x) + np.random.normal(0, 0.1, x.shape)
train_x = torch.from_numpy(x).float()
train_y = torch.from_numpy(y).float()

# test data
test_x = np.linspace(-1, 1, 100)[:, np.newaxis]
test_y = np.square(test_x)  + np.random.normal(0, 0.1, test_x.shape)
test_x = torch.from_numpy(test_x).float()
test_y = torch.from_numpy(test_y).float()
```

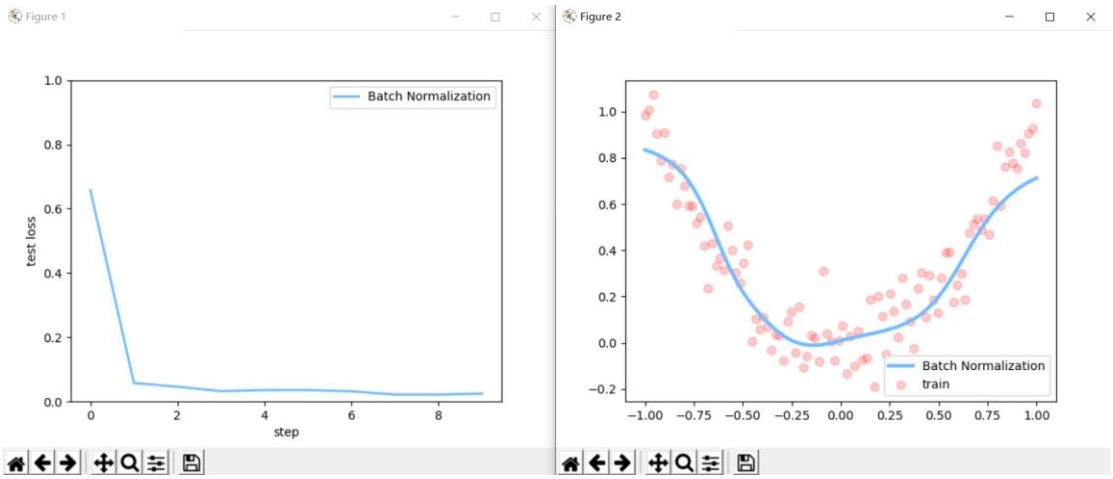
1、不使用 BN 层，左图展示的是每个 epoch 的误差，右图是测试数据及模型拟合的曲线：



2、使用 pytorch 提供的 BatchNorm1d，左图展示的是每个 epoch 的误差，右图是测试数据及模型拟合的曲线：



3、使用自己编写的 MyBN 层，左图展示的是每个 epoch 的误差，右图是测试数据及模型拟合的曲线：



分析：当数据分布在（-1,1）的区间时，由于处在激活函数的敏感区，所以增加 BN 层的效果不明显

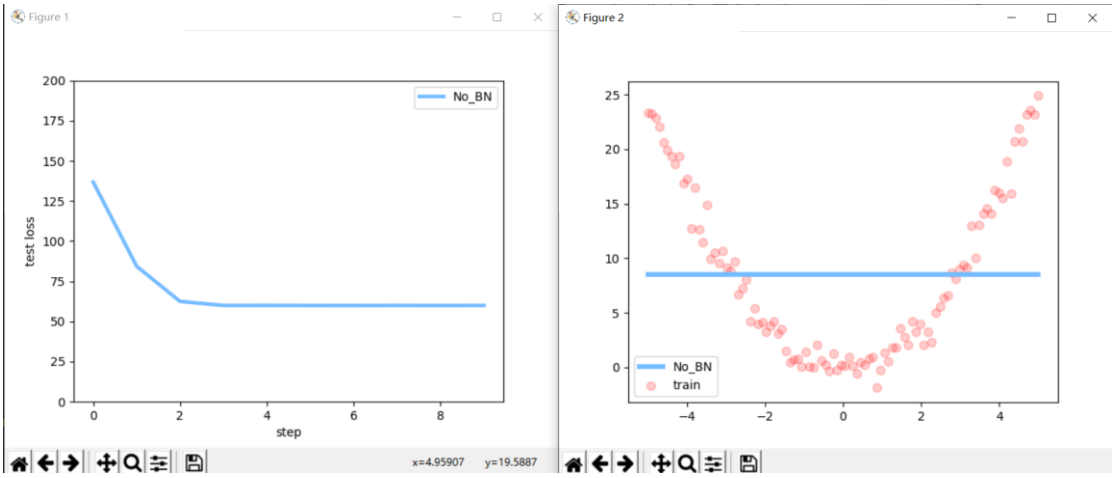
（二）输入数据 x 的范围在（-5,5）时，对比三种模型的效果：

*数据：

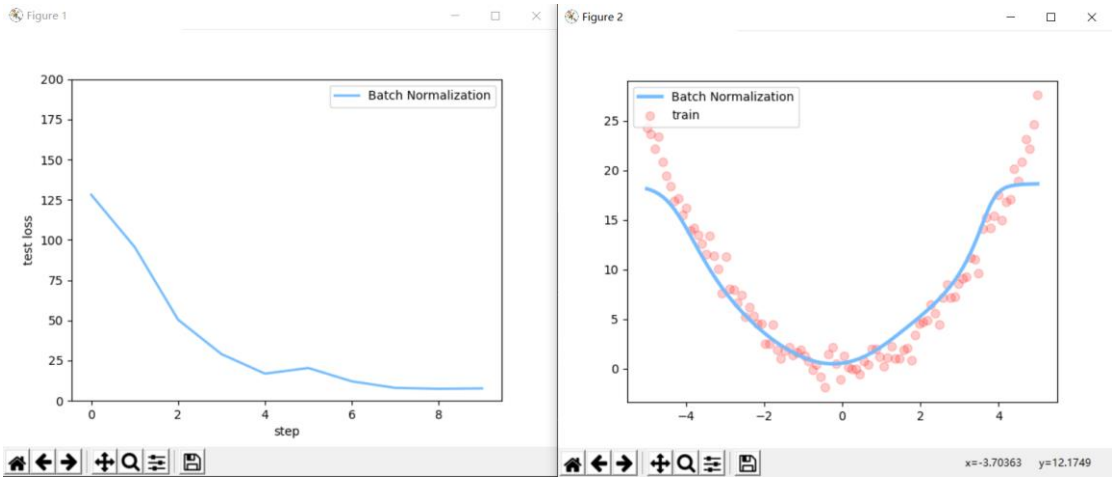
```
#train data
x = np.linspace(-5, 5, 1000)[:, np.newaxis]
y = np.square(x) + np.random.normal(0, 1, x.shape)
train_x = torch.from_numpy(x).float()
train_y = torch.from_numpy(y).float()

# test data
test_x = np.linspace(-5, 5, 100)[:, np.newaxis]
test_y = np.square(test_x)  + np.random.normal(0, 1, test_x.shape)
test_x = torch.from_numpy(test_x).float()
test_y = torch.from_numpy(test_y).float()
```

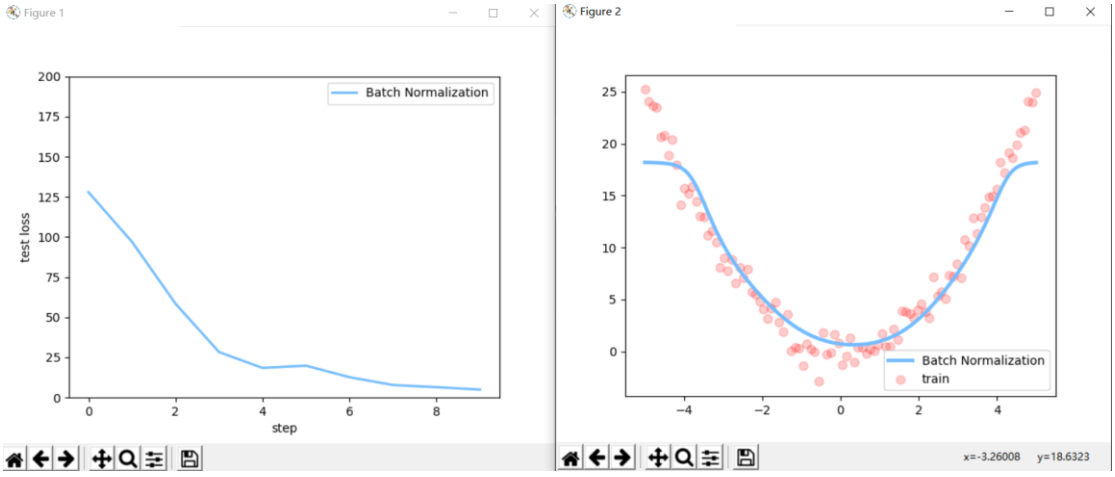
1、不使用 BN 层，左图为 loss，右图为拟合曲线



2、使用 pytorch 提供的 BatchNorm1d，左图为 loss，右图为拟合曲线：



3、使用自己编写的 MyBN 层，左图为 loss，右图为拟合曲线：



分析：当输入的数据在（-5,5）时，大量数据位于激活函数的饱和区，此时如果无 BN 作用，该神经网络将几乎没有学习能力。而加上 pytorch 提供的 BatchNorm1d 模块和

自定义的 MyBN 层都有不错的效果

四、自定义 BN 层时遇到的问题

1、我初始的想法是，对于 BN 层的 forward 和 backward 都由自己编写，利用 dL/dy 以及正向过程存储的变量求出 dL/dγ、dL/dβ、dL/dx，但是由于要利用后层的梯度，并且对前层参数的更新有影响，而在 pytorch 搭建的网络里，其他层的反向传播更新参数的过程都是自动完成的，所以一开始在 bn 层参数的更新上遇到了麻烦；但后来查资料，可以把自定义 BN 层的参数放入网络的 param_groups 中，由 pytorch 来对我需要更新的参数进行更新，所以只需要编写 forward 部分即可；

2、由于在训练模型时是基于 minibatch 的，而测试的时候数据无需分批，所以要区分这两个过程，参考了网上的资料，用两个参数记录全局的 mean 和 var。