批标准化

下面我们可以实现一下简单的一维的情况，也就是神经网络中的情况

```
import torch
```

```
def simple_batch_norm_1d(x, gamma, beta):
    eps = 1e-5
    x_mean = torch.mean(x, dim=0, keepdim=True) # 保留维度进行 broadcast
    x_var = torch.mean((x - x_mean) ** 2, dim=0, keepdim=True)
    x_hat = (x - x_mean) / torch.sqrt(x_var + eps)
    return gamma.view_as(x_mean) * x_hat + beta.view_as(x_mean)
```

我们来验证一下是否对于任意的输入，输出会被标准化

```
x = torch.arange(15).view(5, 3)
gamma = torch.ones(x.shape[1])
beta = torch.zeros(x.shape[1])
print('before bn: ')
print(x)
y = simple_batch_norm_1d(x, gamma, beta)
print('after bn: ')
print(y)
```

```
before bn:

  0   1   2
  3   4   5
  6   7   8
  9  10  11
 12  13  14
[torch.FloatTensor of size 5x3]

after bn:

-1.4142 -1.4142 -1.4142
-0.7071 -0.7071 -0.7071
 0.0000  0.0000  0.0000
 0.7071  0.7071  0.7071
 1.4142  1.4142  1.4142
```

```
[torch.FloatTensor of size 5x3]
```

可以看到这里一共是 5 个数据点，三个特征，每一列表示一个特征的不同数据点，使用批标准化之后，每一列都变成了标准的正态分布

这个时候会出现一个问题，就是测试的时候该使用批标准化吗?

答案是肯定的，因为训练的时候使用了，而测试的时候不使用肯定会导致结果出现偏差，但是测试的时候如果只有一个数据集，那么均值不就是这个值，方差为 0 吗？这显然是随机的，所以测试的时候不能用测试的数据集去算均值和方差，而是用训练的时候算出的移动平均均值和方差去代替

下面我们实现以下能够区分训练状态和测试状态的批标准化方法

```
def batch_norm_1d(x, gamma, beta, is_training, moving_mean, moving_var,
moving_momentum=0.1):
    eps = 1e-5
    x_mean = torch.mean(x, dim=0, keepdim=True) # 保留维度进行 broadcast
    x_var = torch.mean((x - x_mean) ** 2, dim=0, keepdim=True)
    if is_training:
        x_hat = (x - x_mean) / torch.sqrt(x_var + eps)
        moving_mean[:] = moving_momentum * moving_mean + (1. - moving_momentum) *
x_mean
        moving_var[:] = moving_momentum * moving_var + (1. - moving_momentum) * x_var
    else:
        x_hat = (x - moving_mean) / torch.sqrt(moving_var + eps)
    return gamma.view_as(x_mean) * x_hat + beta.view_as(x_mean)
```

下面我们使用上一节课将的深度神经网络分类 mnist 数据集的例子来试验一下批标准化是否有用

```
import numpy as np
from torchvision.datasets import mnist # 导入 pytorch 内置的 mnist 数据
from torch.utils.data import DataLoader
from torch import nn
from torch.autograd import Variable
```

```
# 使用内置函数下载 mnist 数据集
train_set = mnist.MNIST('./data', train=True)
test_set = mnist.MNIST('./data', train=False)

def data_tf(x):
    x = np.array(x, dtype='float32') / 255
    x = (x - 0.5) / 0.5 # 数据预处理，标准化
    x = x.reshape((-1,)) # 拉平
    x = torch.from_numpy(x)
```

```
    return x

train_set = mnist.MNIST('./data', train=True, transform=data_tf, download=True) # 重新载入数据集，申明定义的数据变换
test_set = mnist.MNIST('./data', train=False, transform=data_tf, download=True)
train_data = DataLoader(train_set, batch_size=64, shuffle=True)
test_data = DataLoader(test_set, batch_size=128, shuffle=False)
```

```
class multi_network(nn.Module):
    def __init__(self):
        super(multi_network, self).__init__()
        self.layer1 = nn.Linear(784, 100)
        self.relu = nn.ReLU(True)
        self.layer2 = nn.Linear(100, 10)

        self.gamma = nn.Parameter(torch.randn(100))
        self.beta = nn.Parameter(torch.randn(100))

        self.moving_mean = Variable(torch.zeros(100))
        self.moving_var = Variable(torch.zeros(100))

    def forward(self, x, is_train=True):
        x = self.layer1(x)
        x = batch_norm_1d(x, self.gamma, self.beta, is_train, self.moving_mean, self.moving_var)
        x = self.relu(x)
        x = self.layer2(x)
        return x
```

```
net = multi_network()
```

```
# 定义 loss 函数
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(net.parameters(), 1e-1) # 使用随机梯度下降，学习率 0.1
```

为了方便，训练函数已经定义在外面的 utils.py 中，跟前面训练网络的操作是一样的，感兴趣的同学可以去看看

```
from utils import train
train(net, train_data, test_data, 10, optimizer, criterion)
```

```
Epoch 0. Train Loss: 0.308139, Train Acc: 0.912797, Valid Loss: 0.181375, Valid Acc:
0.948279, Time 00:00:07
Epoch 1. Train Loss: 0.174049, Train Acc: 0.949910, Valid Loss: 0.143940, Valid Acc:
0.958267, Time 00:00:09
Epoch 2. Train Loss: 0.134983, Train Acc: 0.961587, Valid Loss: 0.122489, Valid Acc:
0.963904, Time 00:00:08
Epoch 3. Train Loss: 0.111758, Train Acc: 0.968317, Valid Loss: 0.106595, Valid Acc:
0.966278, Time 00:00:09
Epoch 4. Train Loss: 0.096425, Train Acc: 0.971915, Valid Loss: 0.108423, Valid Acc:
0.967563, Time 00:00:10
Epoch 5. Train Loss: 0.084424, Train Acc: 0.974464, Valid Loss: 0.107135, Valid Acc:
0.969838, Time 00:00:09
Epoch 6. Train Loss: 0.076206, Train Acc: 0.977645, Valid Loss: 0.092725, Valid Acc:
0.971420, Time 00:00:09
Epoch 7. Train Loss: 0.069438, Train Acc: 0.979661, Valid Loss: 0.091497, Valid Acc:
0.971519, Time 00:00:09
Epoch 8. Train Loss: 0.062908, Train Acc: 0.980810, Valid Loss: 0.088797, Valid Acc:
0.972903, Time 00:00:08
Epoch 9. Train Loss: 0.058186, Train Acc: 0.982309, Valid Loss: 0.090830, Valid Acc:
0.972310, Time 00:00:08
```

这里的 γ 和 β 都作为参数进行训练，初始化为随机的高斯分布，`moving_mean` 和 `moving_var` 都初始化为 0，并不是更新的参数，训练完 10 次之后，我们可以看看移动平均和移动方差被修改为了多少

```
# 打出 moving_mean 的前 10 项
print(net.moving_mean[:10])
```

```
Variable containing:
 0.5505
 2.0835
 0.0794
-0.1991
-0.9822
-0.5820
 0.6991
-0.1292
 2.9608
 1.0826
[torch.FloatTensor of size 10]
```

可以看到，这些值已经在训练的过程中进行了修改，在测试过程中，我们不需要再计算均值和方差，直接使用移动平均和移动方差即可

作为对比，我们看看不使用批标准化的结果

```
no_bn_net = nn.Sequential(
    nn.Linear(784, 100),
    nn.ReLU(True),
    nn.Linear(100, 10)
)

optimizer = torch.optim.SGD(no_bn_net.parameters(), 1e-1) # 使用随机梯度下降，学习率 0.1
train(no_bn_net, train_data, test_data, 10, optimizer, criterion)
```

```
Epoch 0. Train Loss: 0.402263, Train Acc: 0.873817, Valid Loss: 0.220468, Valid Acc:
0.932852, Time 00:00:07
Epoch 1. Train Loss: 0.181916, Train Acc: 0.945379, Valid Loss: 0.162440, Valid Acc:
0.953817, Time 00:00:08
Epoch 2. Train Loss: 0.136073, Train Acc: 0.958522, Valid Loss: 0.264888, Valid Acc:
0.918216, Time 00:00:08
Epoch 3. Train Loss: 0.111658, Train Acc: 0.966551, Valid Loss: 0.149704, Valid Acc:
0.950752, Time 00:00:08
Epoch 4. Train Loss: 0.096433, Train Acc: 0.970732, Valid Loss: 0.116364, Valid Acc:
0.963311, Time 00:00:07
Epoch 5. Train Loss: 0.083800, Train Acc: 0.973914, Valid Loss: 0.105775, Valid Acc:
0.968058, Time 00:00:08
Epoch 6. Train Loss: 0.074534, Train Acc: 0.977129, Valid Loss: 0.094511, Valid Acc:
0.970728, Time 00:00:08
Epoch 7. Train Loss: 0.067365, Train Acc: 0.979311, Valid Loss: 0.130495, Valid Acc:
0.960146, Time 00:00:09
Epoch 8. Train Loss: 0.061585, Train Acc: 0.980894, Valid Loss: 0.089632, Valid Acc:
0.974090, Time 00:00:08
Epoch 9. Train Loss: 0.055352, Train Acc: 0.982892, Valid Loss: 0.091508, Valid Acc:
0.970431, Time 00:00:08
```

可以看到虽然最后的结果两种情况一样，但是如果我们看前几次的情况，可以看到使用批标准化的情况能够更快的收敛，因为这只是一个小网络，所以用不用批标准化都能够收敛，但是对于更加深的网络，使用批标准化在训练的时候能够很快地收敛

从上面可以看到，我们自己实现了 2 维情况的批标准化，对应于卷积的 4 维情况的标准化是类似的，只需要沿着通道的维度进行均值和方差的计算，但是我们自己实现批标准化是很累的，pytorch 当然也为我们内置了批标准化的函数，一维和二维分别是 `torch.nn.BatchNorm1d()` 和 `torch.nn.BatchNorm2d()`，不同于我们的实现，pytorch 不仅将 γ 和 β 作为训练的参数，也将 `moving_mean` 和 `moving_var` 也作为参数进行训练

下面我们在卷积网络下试用一下批标准化看看效果

```
def data_tf(x):
    x = np.array(x, dtype='float32') / 255
    x = (x - 0.5) / 0.5 # 数据预处理，标准化
    x = torch.from_numpy(x)
    x = x.unsqueeze(0)
    return x

train_set = mnist.MNIST('./data', train=True, transform=data_tf, download=True) # 重新载入数据集，申明定义的数据变换
test_set = mnist.MNIST('./data', train=False, transform=data_tf, download=True)
train_data = DataLoader(train_set, batch_size=64, shuffle=True)
test_data = DataLoader(test_set, batch_size=128, shuffle=False)
```

```
# 使用批标准化
class conv_bn_net(nn.Module):
    def __init__(self):
        super(conv_bn_net, self).__init__()
        self.stage1 = nn.Sequential(
            nn.Conv2d(1, 6, 3, padding=1),
            nn.BatchNorm2d(6),
            nn.ReLU(True),
            nn.MaxPool2d(2, 2),
            nn.Conv2d(6, 16, 5),
            nn.BatchNorm2d(16),
            nn.ReLU(True),
            nn.MaxPool2d(2, 2)
        )

        self.classfy = nn.Linear(400, 10)
    def forward(self, x):
        x = self.stage1(x)
        x = x.view(x.shape[0], -1)
        x = self.classfy(x)
        return x

net = conv_bn_net()
optimizer = torch.optim.SGD(net.parameters(), 1e-1) # 使用随机梯度下降，学习率 0.1
```

```
train(net, train_data, test_data, 5, optimizer, criterion)
```

```
Epoch 0. Train Loss: 0.160329, Train Acc: 0.952842, Valid Loss: 0.063328, Valid Acc:
0.978441, Time 00:00:33
Epoch 1. Train Loss: 0.067862, Train Acc: 0.979361, Valid Loss: 0.068229, Valid Acc:
0.979430, Time 00:00:37
Epoch 2. Train Loss: 0.051867, Train Acc: 0.984625, Valid Loss: 0.044616, Valid Acc:
0.985265, Time 00:00:37
Epoch 3. Train Loss: 0.044797, Train Acc: 0.986141, Valid Loss: 0.042711, Valid Acc:
0.986056, Time 00:00:38
Epoch 4. Train Loss: 0.039876, Train Acc: 0.987690, Valid Loss: 0.042499, Valid Acc:
0.985067, Time 00:00:41
```

```
# 不使用批标准化
class conv_no_bn_net(nn.Module):
    def __init__(self):
        super(conv_no_bn_net, self).__init__()
        self.stage1 = nn.Sequential(
            nn.Conv2d(1, 6, 3, padding=1),
            nn.ReLU(True),
            nn.MaxPool2d(2, 2),
            nn.Conv2d(6, 16, 5),
            nn.ReLU(True),
            nn.MaxPool2d(2, 2)
        )

        self.classfy = nn.Linear(400, 10)
    def forward(self, x):
        x = self.stage1(x)
        x = x.view(x.shape[0], -1)
        x = self.classfy(x)
        return x

net = conv_no_bn_net()
optimizer = torch.optim.SGD(net.parameters(), 1e-1) # 使用随机梯度下降，学习率 0.1
```

```
train(net, train_data, test_data, 5, optimizer, criterion)
```

```
Epoch 0. Train Loss: 0.211075, Train Acc: 0.935934, Valid Loss: 0.062950, Valid Acc:
0.980123, Time 00:00:27
Epoch 1. Train Loss: 0.066763, Train Acc: 0.978778, Valid Loss: 0.050143, Valid Acc:
0.984375, Time 00:00:29
Epoch 2. Train Loss: 0.050870, Train Acc: 0.984292, Valid Loss: 0.039761, Valid Acc:
0.988034, Time 00:00:29
Epoch 3. Train Loss: 0.041476, Train Acc: 0.986924, Valid Loss: 0.041925, Valid Acc:
0.986155, Time 00:00:29
Epoch 4. Train Loss: 0.036118, Train Acc: 0.988523, Valid Loss: 0.042703, Valid Acc:
0.986452, Time 00:00:29
```

之后介绍一些著名的网络结构的时候，我们会慢慢认识到批标准化的重要性，使用 pytorch 能够非常方便地添加批标准化层