5.10 批量归一化

本节我们介绍批量归一化（batch normalization）层，它能让较深的神经网络的训练变得更加容易 [1]。在3.16节（实战Kaggle比赛：预测房价）里，我们对输入数据做了标准化处理：处理后的任意一个特征在数据集中所有样本上的均值为0、标准差为1。标准化处理输入数据使各个特征的分布相近：这往往更容易训练出有效的模型。

通常来说，数据标准化预处理对于浅层模型就足够有效了。随着模型训练的进行，当每层中参数更新时，靠近输出层的输出较难出现剧烈变化。但对深层神经网络来说，即使输入数据已做标准化，训练中模型参数的更新依然很容易造成靠近输出层输出的剧烈变化。这种计算数值的不稳定性通常令我们难以训练出有效的深度模型。

批量归一化的提出正是为了应对深度模型训练的挑战。在模型训练时，批量归一化利用小批量上的均值和标准差，不断调整神经网络中间输出，从而使整个神经网络在各层的中间输出的数值更稳定。**批量归一化和下一节将要介绍的残差网络为训练和设计深度模型提供了两类重要思路。**

5.10.1 批量归一化层

对全连接层和卷积层做批量归一化的方法稍有不同。下面我们将分别介绍这两种情况下的批量归一化。

5.10.1.1 对全连接层做批量归一化

我们先考虑如何对全连接层做批量归一化。通常，我们将批量归一化层置于全连接层中的仿射变换和激活函数之间。设全连接层的输入为$\boldsymbol{u}$，权重参数和偏差参数分别为$\boldsymbol{W}$和$\boldsymbol{b}$，激活函数为ϕ。设批量归一化的运算符为BN。那么，使用批量归一化的全连接层的输出为

$$\phi(\mathrm{BN}(\boldsymbol{x})),$$

其中批量归一化输入$\boldsymbol{x}$由仿射变换

$$\boldsymbol{x} = \boldsymbol{W}\boldsymbol{u} + \boldsymbol{b}$$

得到。考虑一个由m个样本组成的小批量，仿射变换的输出为一个新的小批量$\mathcal{B} = \{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$。它们正是批量归一化层的输入。对于小批量$\mathcal{B}$中任意样本$\boldsymbol{x}^{(i)} \in \mathbb{R}^d, 1 \leq i \leq m$，批量归一化层的输出同样是$d$维向量

$$\boldsymbol{y}^{(i)} = \mathrm{BN}(\boldsymbol{x}^{(i)}),$$

并由以下几步求得。首先，对小批量$\mathcal{B}$求均值和方差：

$$\boldsymbol{\mu}_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} \boldsymbol{x}^{(i)},$$

$$\boldsymbol{\sigma}_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (\boldsymbol{x}^{(i)} - \boldsymbol{\mu}_\mathcal{B})^2,$$

其中的平方计算是按元素求平方。接下来，使用按元素开方和按元素除法对$\boldsymbol{x}^{(i)}$标准化：

$$\hat{\boldsymbol{x}}^{(i)} \leftarrow \frac{\boldsymbol{x}^{(i)} - \boldsymbol{\mu}_\mathcal{B}}{\sqrt{\boldsymbol{\sigma}_\mathcal{B}^2 + \epsilon}},$$

这里$\epsilon > 0$是一个很小的常数，保证分母大于0。在上面标准化的基础上，批量归一化层引入了两个可以学习的模型参数，拉伸（scale）参数 $\boldsymbol{\gamma}$ 和偏移（shift）参数 $\boldsymbol{\beta}$。这两个参数和$\boldsymbol{x}^{(i)}$形状相同，皆为d维向量。它们与$\boldsymbol{x}^{(i)}$分别做按元素乘法（符号⊙）和加法计算：

$$\boldsymbol{y}^{(i)} \leftarrow \boldsymbol{\gamma} \odot \hat{\boldsymbol{x}}^{(i)} + \boldsymbol{\beta}.$$

至此，我们得到了$\boldsymbol{x}^{(i)}$的批量归一化的输出$\boldsymbol{y}^{(i)}$。 值得注意的是，可学习的拉伸和偏移参数保留了不对$\boldsymbol{x}^{(i)}$做批量归一化的可能：此时只需学出$\boldsymbol{\gamma} = \sqrt{\boldsymbol{\sigma}_\mathcal{B}^2 + \epsilon}$和$\boldsymbol{\beta} = \boldsymbol{\mu}_\mathcal{B}$。我们可以对此这样理解：如果批量归一化无益，理论上，学出的模型可以不使用批量归一化。

5.10.1.2 对卷积层做批量归一化

对卷积层来说，批量归一化发生在卷积计算之后、应用激活函数之前。如果卷积计算输出多个通道，我们需要对这些通道的输出分别做批量归一化，且**每个通道都拥有独立的拉伸和偏移参数，并均为标量**。设小批量中有m个样本。在单个通道上，假设卷积计算输出的高和宽分别为p和q。我们需要对该通道中$m \times p \times q$个元素同时做批量归一化。对这些元素做标准化计算时，我们使用相同的均值和方差，即该通道中$m \times p \times q$个元素的均值和方差。

5.10.1.3 预测时的批量归一化

使用批量归一化训练时，我们可以将批量大小设得大一点，从而使批量内样本的均值和方差的计算都较为准确。将训练好的模型用于预测时，我们希望模型对于任意输入都有确定的输出。因此，单个样本的输出不应取决于批量归一化所需要的随机小批量中的均值和方差。一种常用的方法是通过移动平均估算整个训练数据集的样本均值和方差，并在预测时使用它们得到确定的输出。可见，和丢弃层一样，批量归一化层在训练模式和预测模式下的计算结果也是不一样的。

5.10.2 从零开始实现

下面我们自己实现批量归一化层。

```
import time
import torch
from torch import nn, optim
import torch.nn.functional as F

import sys
sys.path.append("..")
import d2lzh_pytorch as d2l
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

def batch_norm(is_training, X, gamma, beta, moving_mean, moving_var, eps, momentum):
    # 判断当前模式是训练模式还是预测模式
    if not is_training:
        # 如果是在预测模式下，直接使用传入的移动平均所得的均值和方差
        X_hat = (X - moving_mean) / torch.sqrt(moving_var + eps)
    else:
        assert len(X.shape) in (2, 4)
        if len(X.shape) == 2:
            # 使用全连接层的情况，计算特征维上的均值和方差
            mean = X.mean(dim=0)
            var = ((X - mean) ** 2).mean(dim=0)
        else:
            # 使用二维卷积层的情况，计算通道维上（axis=1）的均值和方差。这里我们需要保持
            # X的形状以便后面可以做广播运算
            mean = X.mean(dim=0, keepdim=True).mean(dim=2, keepdim=True).mean(dim=3,
            var = ((X - mean) ** 2).mean(dim=0, keepdim=True).mean(dim=2, keepdim=Tru
        # 训练模式下用当前的均值和方差做标准化
        X_hat = (X - mean) / torch.sqrt(var + eps)
        # 更新移动平均的均值和方差
        moving_mean = momentum * moving_mean + (1.0 - momentum) * mean
        moving_var = momentum * moving_var + (1.0 - momentum) * var
    Y = gamma * X_hat + beta  # 拉伸和偏移
    return Y, moving_mean, moving_var
```

接下来，我们自定义一个 `BatchNorm` 层。它保存参与求梯度和迭代的拉伸参数 `gamma` 和偏移参数 `beta` ，同时也维护移动平均得到的均值和方差，以便能够在模型预测时被使用。 `BatchNorm` 实例所需指定的 `num_features` 参数对于全连接层来说应为输出个数，对于卷积层来说则为输出通道数。该实例所需指定的 `num_dims` 参数对于全连接层和卷积层来说分别为2和4。

```
class BatchNorm(nn.Module):
    def __init__(self, num_features, num_dims):
        super(BatchNorm, self).__init__()
        if num_dims == 2:
            shape = (1, num_features)
        else:
            shape = (1, num_features, 1, 1)
        # 参与求梯度和迭代的拉伸和偏移参数，分别初始化成0和1
        self.gamma = nn.Parameter(torch.ones(shape))
        self.beta = nn.Parameter(torch.zeros(shape))
        # 不参与求梯度和迭代的变量，全在内存上初始化成0
        self.moving_mean = torch.zeros(shape)
        self.moving_var = torch.zeros(shape)

    def forward(self, X):
        # 如果X不在内存上，将moving_mean和moving_var复制到X所在显存上
        if self.moving_mean.device != X.device:
            self.moving_mean = self.moving_mean.to(X.device)
            self.moving_var = self.moving_var.to(X.device)
        # 保存更新过的moving_mean和moving_var, Module实例的traning属性默认为true, 调用.e
        Y, self.moving_mean, self.moving_var = batch_norm(self.training,
            X, self.gamma, self.beta, self.moving_mean,
            self.moving_var, eps=1e-5, momentum=0.9)
        return Y
```

5.10.2.1 使用批量归一化层的LeNet

下面我们修改5.5节（卷积神经网络（LeNet））介绍的LeNet模型，从而应用批量归一化层。我们在所有的卷积层或全连接层之后、激活层之前加入批量归一化层。

```
net = nn.Sequential(
            nn.Conv2d(1, 6, 5), # in_channels, out_channels, kernel_size
            BatchNorm(6, num_dims=4),
            nn.Sigmoid(),
            nn.MaxPool2d(2, 2), # kernel_size, stride
            nn.Conv2d(6, 16, 5),
            BatchNorm(16, num_dims=4),
            nn.Sigmoid(),
```

```
            nn.MaxPool2d(2, 2),
            d2l.FlattenLayer(),
            nn.Linear(16*4*4, 120),
            BatchNorm(120, num_dims=2),
            nn.Sigmoid(),
            nn.Linear(120, 84),
            BatchNorm(84, num_dims=2),
            nn.Sigmoid(),
            nn.Linear(84, 10)
        )
```

下面我们训练修改后的模型。

```
batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size=batch_size)

lr, num_epochs = 0.001, 5
optimizer = torch.optim.Adam(net.parameters(), lr=lr)
d2l.train_ch5(net, train_iter, test_iter, batch_size, optimizer, device, num_epochs)
```

输出：

```
training on  cuda
epoch 1, loss 0.0039, train acc 0.790, test acc 0.835, time 2.9 sec
epoch 2, loss 0.0018, train acc 0.866, test acc 0.821, time 3.2 sec
epoch 3, loss 0.0014, train acc 0.879, test acc 0.857, time 2.6 sec
epoch 4, loss 0.0013, train acc 0.886, test acc 0.820, time 2.7 sec
epoch 5, loss 0.0012, train acc 0.891, test acc 0.859, time 2.8 sec
```

最后我们查看第一个批量归一化层学习到的拉伸参数 `gamma` 和偏移参数 `beta` 。

```
net[1].gamma.view((-1,)), net[1].beta.view((-1,))
```

输出：

```
(tensor([ 1.2537,  1.2284,  1.0100,  1.0171,  0.9809,  1.1870], device='cuda:0'),
 tensor([ 0.0962,  0.3299, -0.5506,  0.1522, -0.1556,  0.2240], device='cuda:0'))
```

5.10.3 简洁实现

与我们刚刚自己定义的 `BatchNorm` 类相比，Pytorch中 `nn` 模块定义的 `BatchNorm1d` 和 `BatchNorm2d` 类使用起来更加简单，二者分别用于全连接层和卷积层，都需要指定输入的 `num_features` 参数值。下面我们用PyTorch实现使用批量归一化的LeNet。

```
net = nn.Sequential(
            nn.Conv2d(1, 6, 5), # in_channels, out_channels, kernel_size
            nn.BatchNorm2d(6),
            nn.Sigmoid(),
            nn.MaxPool2d(2, 2), # kernel_size, stride
            nn.Conv2d(6, 16, 5),
            nn.BatchNorm2d(16),
            nn.Sigmoid(),
            nn.MaxPool2d(2, 2),
            d2l.FlattenLayer(),
            nn.Linear(16*4*4, 120),
            nn.BatchNorm1d(120),
            nn.Sigmoid(),
            nn.Linear(120, 84),
            nn.BatchNorm1d(84),
            nn.Sigmoid(),
            nn.Linear(84, 10)
        )
```

使用同样的超参数进行训练。

```
batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size=batch_size)

lr, num_epochs = 0.001, 5
optimizer = torch.optim.Adam(net.parameters(), lr=lr)
d2l.train_ch5(net, train_iter, test_iter, batch_size, optimizer, device, num_epochs)
```

输出：

```
training on  cuda
epoch 1, loss 0.0054, train acc 0.767, test acc 0.795, time 2.0 sec
epoch 2, loss 0.0024, train acc 0.851, test acc 0.748, time 2.0 sec
epoch 3, loss 0.0017, train acc 0.872, test acc 0.814, time 2.2 sec
epoch 4, loss 0.0014, train acc 0.883, test acc 0.818, time 2.1 sec
epoch 5, loss 0.0013, train acc 0.889, test acc 0.734, time 1.8 sec
```

小结

- 在模型训练时，批量归一化利用小批量上的均值和标准差，不断调整神经网络的中间输出，从而使整个神经网络在各层的中间输出的数值更稳定。
- 对全连接层和卷积层做批量归一化的方法稍有不同。
- 批量归一化层和丢弃层一样，在训练模式和预测模式的计算结果是不一样的。
- PyTorch提供了BatchNorm类方便使用。