参考文献

(1) Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., & Anguelov, D. & Rabinovich, A.(2015). Going deeper with convolutions. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 1-9).

(2) Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167.

(3) Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2016). Rethinking the inception architecture for computer vision. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 2818-2826).

(4) Szegedy, C., Ioffe, S., Vanhoucke, V., & Alemi, A. A. (2017, February). Inception-v4, inception-resnet and the impact of residual connections on learning. In Proceedings of the AAAI Conference on Artificial Intelligence (Vol. 4, p. 12).

> 注：除代码外本节与原书此节基本相同，原书传送门

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。

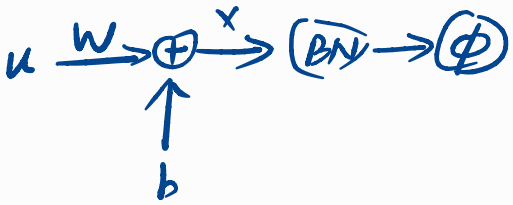
5.10.1.1 对全连接层做批量归一化

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

Batch_size * 输出神经元个数

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

对中间值x做标准化，让其在batch_size上的均值为0，标准差为1 。



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：

m：batch_size大小

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

d维

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$\odot$）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$\hat{\boldsymbol{x}}^{(i)}$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。

即如果两个参数等于上面这样，x就回到了归一化前的值，自己带到上面的式子就知道。

5.10.1.2 对卷积层做批量归一化

CNN → BN → ϕ

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, keepdim=True)
            var = ((X - mean) ** 2).mean(dim=0, keepdim=True).mean(dim=2, keepdim=True).mean(dim=3, keepdim=True)
        # 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
```

标准化的目标：可能是仿射变换的输出，也可能是卷积层的输出。

目标是把X标准化为期望0、方差1的值。

以batch为单位计算均值和方差

先判断X的形状是2还是4；

若是2，则是全连接层仿射变换的输出，则在其第0维上求平均。因为全连接层的维度是m*d，m是batch_size，所以对全连接层就是在其batch_size的维度上求平均。

求出的平均值第1维的值是d

则X是卷积计算后得到的输出，即batch_size*通道数*高*宽

即对batch_size*h*w个元素同时做批量归一化，使用相同的均值和方差。

后→前，先在3rd维上求均值，再在2nd维上求均值，后在0th维上求均值，分别对应在w、h、batch_size上求均值。

公式：

```
超参，自己设置
        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): 全连接层：代表输出神经元的个数；
    def __init__(self, num_features, num_dims): 卷积层的情况下代表通道数；  2或4，对应全连接层和卷积层
        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: # 如果X不在GPU上，就复制到GPU上
            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.eval()后设成false
        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)# 在forward中已经调用了batct_norm了
        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), # 在这一层里加上BatchNorm层即可。
            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)
        )
```

120:输出神经元的个数

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)
        )
```

卷积层的batch_norm

全连接层的batch_norm

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。

参考文献

(1) Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167.

> 注：除代码外本节与原书此节基本相同，原书传送门

5.11 残差网络（RESNET）

让我们先思考一个问题：对神经网络模型添加新的层，充分训练后的模型是否只可能更有效地降低训练误差？理论上，原模型解的空间只是新模型解的空间的子空间。也就是说，如果我们能将新添加的层训练成恒等映射$f(x) = x$，新模型和原模型将同样有效。由于新模型可能得出更优的解来拟合训练数据集，因此添加层似乎更容易降低训练误差。然而在实践中，添加过多的层后训练误差往往不降反升。即使利用批量归一化带来的数值稳定性使训练深层模型更加容易，该问题仍然存在。针对这一问题，何恺明等人提出了残差网络（ResNet）〔1〕。它在2015年的ImageNet图像识别挑战赛夺魁，并深刻影响了后来的深度神经网络的设计。

5.11.2 残差块

让我们聚焦于神经网络局部。如图5.9所示，设输入为$\boldsymbol{x}$。假设我们希望学出的理想映射为$f(\boldsymbol{x})$，从而作为图5.9上方激活函数的输入。左图虚线框中的部分需要直接拟合出该映射$f(\boldsymbol{x})$，而右图虚线框中的部分则需要拟合出有关恒等映射的残差映射$f(\boldsymbol{x}) - \boldsymbol{x}$。残差映射在实际中往往更容易优化。以本节开头提到的恒等映射作为我们希望学出的理想映射$f(\boldsymbol{x})$。我们只需将图5.9中右图虚线框内上方的加权运算（如仿射）的权重和偏差参数学成0，那么$f(\boldsymbol{x})$即为恒等映射。实际中，当理想映射$f(\boldsymbol{x})$极接近于恒等映射时，残差映射也易于捕捉恒等映射的细微波动。图5.9右图也是ResNet的基础块，即残差块（residual block）。在残差块中，输入可通过跨层的数据线路更快地向前传播。

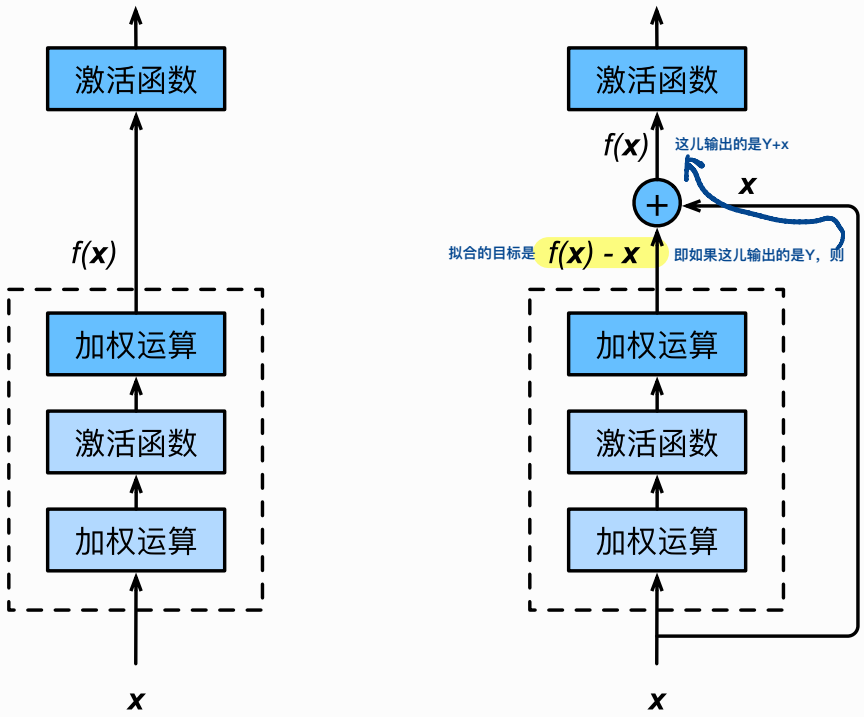


图5.9 普通的网络结构（左）与加入残差连接的网络结构（右）

ResNet沿用了VGG全3×3卷积层的设计。残差块里首先有2个有相同输出通道数的3×3卷积层。每个卷积层后接一个批量归一化层和ReLU激活函数。然后我们将输入跳过这两个卷积运算后直接加在最后的ReLU激活函数前。这样的设计要求两个卷积层的输出与输入形状一样，从而可以相加。如果想改变通道数，就需要引入一个额外的1×1卷积层来将输入变换成需要的形状后再做相加运算。

残差块的实现如下。它可以设定输出通道数、是否使用额外的1×1卷积层来修改通道数以及卷积层的步幅。

```
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')

class Residual(nn.Module):  # 本类已保存在d2lzh_pytorch包中方便以后使用
    def __init__(self, in_channels, out_channels,
use_1x1conv=False, stride=1):
        super(Residual, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels,
kernel_size=3, padding=1, stride=stride)
        self.conv2 = nn.Conv2d(out_channels, out_channels,
kernel_size=3, padding=1)
        if use_1x1conv:
            self.conv3 = nn.Conv2d(in_channels, out_channels,
kernel_size=1, stride=stride)
```

```
        else:
            self.conv3 = None
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.bn2 = nn.BatchNorm2d(out_channels)

重点看forward方法：  def forward(self, X):
        Y = F.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))# 这一步得出的Y是一个结果，如果没有残差，Y就可以直接作为结果。但现在需要把X的形状变成Y，因为残差块最终返回的是Y+x
        if self.conv3:
            X = self.conv3(X) # 改变X的通道数，与Y相同
        return F.relu(Y + X)
```

下面我们来查看输入和输出形状一致的情况。

```
blk = Residual(3, 3) # 输入通道和输出通道相同
X = torch.rand((4, 3, 6, 6))
blk(X).shape # torch.Size([4, 3, 6, 6])
```

我们也可以在增加输出通道数的同时减半输出的高和宽。

```
                                    当输入和输出通道数不同时，就需要1*1全连接层改变通道数
blk = Residual(3, 6, use_1x1conv=True, stride=2)
blk(X).shape # torch.Size([4, 6, 3, 3])
```

5.11.2 RESNET模型

ResNet的前两层跟之前介绍的GoogLeNet中的一样：在输出通道数为64、步幅为2的7×7卷积层后接步幅为2的3×3的最大池化层。不同之处在于ResNet每个卷积层后增加的批量归一化层。

```
net = nn.Sequential(
        nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3),
        nn.BatchNorm2d(64),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

GoogLeNet在后面接了4个由Inception块组成的模块。ResNet则使用4个由残差块组成的模块，每个模块使用若干个同样输出通道数的残差块。第一个模块的通道数同输入通道数一致。由于之前已经使用了步幅为2的最大池化层，所以无须减小高和宽。之后的每个模块在第一个残差块里将上一个模块的通道数翻倍，并将高和宽减半。

下面我们来实现这个模块。注意，这里对第一个模块做了特别处理。

```
def resnet_block(in_channels, out_channels, num_residuals,
first_block=False):
    if first_block:
        assert in_channels == out_channels # 第一个模块的通道数同输入通
道数一致
    blk = []
    for i in range(num_residuals):
        if i == 0 and not first_block:
            blk.append(Residual(in_channels, out_channels,
use_1x1conv=True, stride=2))
        else:
            blk.append(Residual(out_channels, out_channels))
    return nn.Sequential(*blk)
```

该block中需要包含的残差块，即class Residual

因为1st block的输入和输出通道数一致，所以需要区别对待

若为各block中的第0个残差块

用1*1的CNN，放缩输入通道数至输出通道个数

在非第0个残差块中，保证残差块的输入和输出通道都是out_channels

接着我们为ResNet加入所有残差块。这里每个模块使用两个残差块。

添加4个残差block

```
net.add_module("resnet_block1", resnet_block(64, 64, 2,
first_block=True))
net.add_module("resnet_block2", resnet_block(64, 128, 2))
net.add_module("resnet_block3", resnet_block(128, 256, 2))
net.add_module("resnet_block4", resnet_block(256, 512, 2))
```

最后，与GoogLeNet一样，加入全局平均池化层后接上全连接层输出。

```
net.add_module("global_avg_pool", d2l.GlobalAvgPool2d()) #
GlobalAvgPool2d的输出: (Batch, 512, 1, 1)
net.add_module("fc", nn.Sequential(d2l.FlattenLayer(),
nn.Linear(512, 10)))
```

这里每个模块里有4个卷积层（不计算1×1卷积层），加上最开始的卷积层和最后的全连接层，共计18层。这个模型通常也被称为ResNet-18。通过配置不同的通道数和模块里的残差块数可以得到不同的ResNet模型，例如更深的含152层的ResNet-152。虽然ResNet的主体架构跟GoogLeNet的类似，但ResNet结构更简单，修改也更方便。这些因素都导致了ResNet迅速被广泛使用。

在训练ResNet之前，我们来观察一下输入形状在ResNet不同模块之间的变化。

输入一个例子：

假设1张图是224*224的灰度图

```
X = torch.rand((1, 1, 224, 224))
for name, layer in net.named_children():
    X = layer(X)
    print(name, ' output shape:\t', X.shape)
```

输出：

每一层的大小输出：

```
0  output shape:     torch.Size([1, 64, 112, 112])
1  output shape:     torch.Size([1, 64, 112, 112])
2  output shape:     torch.Size([1, 64, 112, 112])
3  output shape:     torch.Size([1, 64, 56, 56])
resnet_block1  output shape:     torch.Size([1, 64, 56, 56])
resnet_block2  output shape:     torch.Size([1, 128, 28, 28])
resnet_block3  output shape:     torch.Size([1, 256, 14, 14])
resnet_block4  output shape:     torch.Size([1, 512, 7, 7])
global_avg_pool  output shape:   torch.Size([1, 512, 1, 1])
fc  output shape:     torch.Size([1, 10])
```

5.11.3 获取数据和训练模型

下面我们在Fashion-MNIST数据集上训练ResNet。

```
batch_size = 256
# 如出现“out of memory”的报错信息，可减小batch_size或resize
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size,
resize=96)

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 0.0015, train acc 0.853, test acc 0.885, time 31.0 sec
epoch 2, loss 0.0010, train acc 0.910, test acc 0.899, time 31.8 sec
epoch 3, loss 0.0008, train acc 0.926, test acc 0.911, time 31.6 sec
epoch 4, loss 0.0007, train acc 0.936, test acc 0.916, time 31.8 sec
epoch 5, loss 0.0006, train acc 0.944, test acc 0.926, time 31.5 sec
```

小结

- 残差块通过跨层的数据通道从而能够训练出有效的深度神经网络。
- ResNet深刻影响了后来的深度神经网络的设计。

参考文献

(1) He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).

(2) He, K., Zhang, X., Ren, S., & Sun, J. (2016, October). Identity mappings in deep residual networks. In European Conference on Computer Vision (pp. 630-645). Springer, Cham.

> 注：除代码外本节与原书此节基本相同，原书传送门

5.12 稠密连接网络（DENSENET）

ResNet中的跨层连接设计引申出了数个后续工作。本节我们介绍其中的一个：稠密连接网络（DenseNet）（1）。它与ResNet的主要区别如图5.10所示。

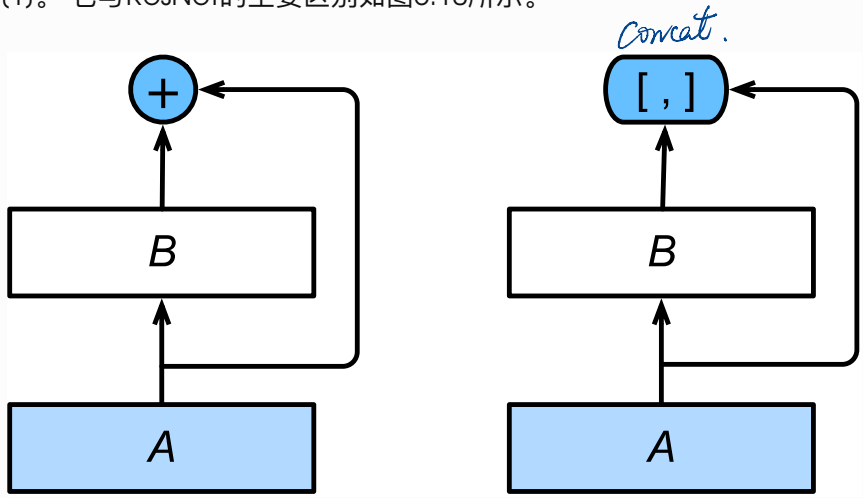


图5.10 ResNet（左）与DenseNet（右）在跨层连接上的主要区别：使用相加和使用连结

图5.10中将部分前后相邻的运算抽象为模块A和模块B。与ResNet的主要区别在于，DenseNet里模块B的输出（假设A有10个通道，B有20个通道，则连接之后的输出是30个通道。）不是像ResNet那样和模块A的输出相加，）而是在通道维上连结。这样模块A的输出可以直接传入模块B后面的层。在这个设计里，模块A直接跟模块B后面的所有层连接在了一起。这也是它被称为“稠密连接”的原因。

DenseNet的主要构建模块是稠密块（dense block）和过渡层（transition layer）。前者定义了输入和输出是如何连结的，后者则用来控制通道数，使之不过大。因为A和B不断连接，那么通道数会越来越大。
也就是上右图看到的部分

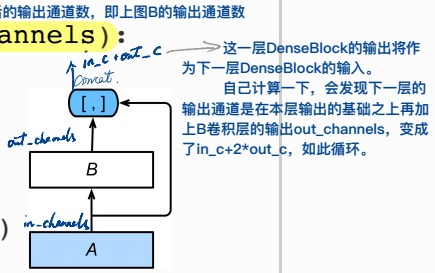
5.12.1 稠密块

DenseNet使用了ResNet改良版的“批量归一化、激活和卷积”结构，我们首先在`conv_block`函数里实现这个结构。

```
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 conv_block(in_channels, out_channels): # 这个函数将BN、ReLU、Conv2d打包起来
    blk = nn.Sequential(nn.BatchNorm2d(in_channels),
                        nn.ReLU(),
                        nn.Conv2d(in_channels, out_channels,
kernel_size=3, padding=1))
    return blk
```

稠密块由多个`conv_block`组成，每块使用相同的输出通道数。但在前向计算时，我们将每块的输入和输出在通道维上连结。

```
class DenseBlock(nn.Module):
    def __init__(self, num_convs, in_channels, out_channels):
        super(DenseBlock, self).__init__()
        net = []
        for i in range(num_convs):
            in_c = in_channels + i * out_channels
            net.append(conv_block(in_c, out_channels))
        self.net = nn.ModuleList(net)
        self.out_channels = in_channels + num_convs * out_channels
# 计算输出通道数

    def forward(self, X):
        for blk in self.net:
            Y = blk(X) # 每个X都经过一个卷积层
            X = torch.cat((X, Y), dim=1)  # 在通道维上将输入和输出连结
        return X
```



在下面的例子中，我们定义一个有2个输出通道数为10的卷积块。使用通道数为3的输入时，我们会得到通道数为$3+2\times10=23$的输出。卷积块的通道数控制了输出通道数相对于输入通道数的增长，因此也被称为增长率（growth rate）。

```
blk = DenseBlock(2, 3, 10) # 依据上一块代码的右图的解释，这儿输出通道数为2*10+3=23
X = torch.rand(4, 3, 8, 8)
Y = blk(X)
Y.shape # torch.Size([4, 23, 8, 8]) # 没有过度层的输出通道数是23
```

batch_size 通道数 高 宽

通道数

5.12.2 过渡层

由于每个稠密块都会带来通道数的增加，使用过多则会带来过于复杂的模型。过渡层用来控制模型复杂度。它通过1×1卷积层来减小通道数，并使用步幅为2的平均池化层减半高和宽，从而进一步降低模型复杂度。

```
def transition_block(in_channels, out_channels):
    blk = nn.Sequential(
            nn.BatchNorm2d(in_channels),
            nn.ReLU(),
            nn.Conv2d(in_channels, out_channels, kernel_size=1),
            nn.AvgPool2d(kernel_size=2, stride=2))# 减半高、宽
    return blk
```

对上一个例子中稠密块的输出使用通道数为10的过渡层。此时输出的通道数减为10，高和宽均减半。

```
                                  接上上块未接过度层的输出通道
blk = transition_block(23, 10)
blk(Y).shape # torch.Size([4, 10, 4, 4])
```

5.12.3 DENSENET模型

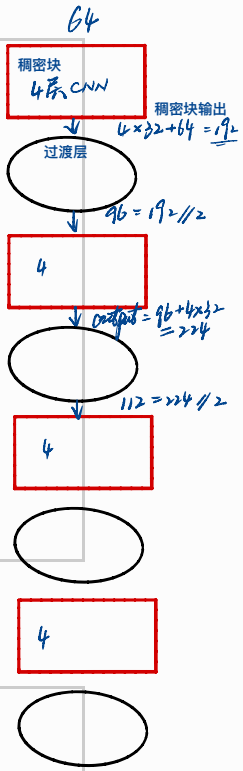
我们来构造DenseNet模型。DenseNet首先使用同ResNet一样的单卷积层和最大池化层。

```
net = nn.Sequential(
        nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3),  # 高、宽减半
        nn.BatchNorm2d(64),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=3, stride=2, padding=1))  # 高、宽再减半
```

类似于ResNet接下来使用的4个残差块，DenseNet使用的是4个稠密块。同ResNet一样，我们可以设置每个稠密块使用多少个卷积层。这里我们设成4，从而与上一节的ResNet-18保持一致。稠密块里的卷积层通道数（即增长率）设为32，所以每个稠密块将增加128个通道。

ResNet里通过步幅为2的残差块在每个模块之间减小高和宽。这里我们则使用过渡层来减半高和宽，并减半通道数。

```
num_channels, growth_rate = 64, 32  # num_channels为当前的通道数
num_convs_in_dense_blocks = [4, 4, 4, 4]
for i, num_convs in enumerate(num_convs_in_dense_blocks):
    DB = DenseBlock(num_convs, num_channels, growth_rate)
    net.add_module("DenseBlosk_%d" % i, DB)
    # 上一个稠密块的输出通道数
    num_channels = DB.out_channels
    # 在稠密块之间加入通道数减半的过渡层
    if i != len(num_convs_in_dense_blocks) - 1:
        net.add_module("transition_block_%d" % i,
transition_block(num_channels, num_channels // 2))
        num_channels = num_channels // 2
```



同ResNet一样，最后接上全局池化层和全连接层来输出。

```
net.add_module("BN", nn.BatchNorm2d(num_channels))
net.add_module("relu", nn.ReLU())
net.add_module("global_avg_pool", d2l.GlobalAvgPool2d()) #
GlobalAvgPool2d的输出: (Batch, num_channels, 1, 1)
net.add_module("fc", nn.Sequential(d2l.FlattenLayer(),
nn.Linear(num_channels, 10)))
```

我们尝试打印每个子模块的输出维度确保网络无误：

```
X = torch.rand((1, 1, 96, 96))
for name, layer in net.named_children():
    X = layer(X)
    print(name, ' output shape:\t', X.shape)
```

输出：

```
0  output shape:     torch.Size([1, 64, 48, 48])
1  output shape:     torch.Size([1, 64, 48, 48])
2  output shape:     torch.Size([1, 64, 48, 48])
3  output shape:     torch.Size([1, 64, 24, 24])
DenseBlosk_0  output shape:  torch.Size([1, 192, 24, 24])
transition_block_0  output shape:    torch.Size([1, 96, 12, 12])
DenseBlosk_1  output shape:  torch.Size([1, 224, 12, 12])
transition_block_1  output shape:    torch.Size([1, 112, 6, 6])
DenseBlosk_2  output shape:  torch.Size([1, 240, 6, 6])
transition_block_2  output shape:    torch.Size([1, 120, 3, 3])
DenseBlosk_3  output shape:  torch.Size([1, 248, 3, 3])
BN  output shape:    torch.Size([1, 248, 3, 3])
relu  output shape:  torch.Size([1, 248, 3, 3])
global_avg_pool  output shape:   torch.Size([1, 248, 1, 1])
fc  output shape:    torch.Size([1, 10])
```

5.12.4 获取数据并训练模型

由于这里使用了比较深的网络，本节里我们将输入高和宽从224降到96来简化计算。

```
batch_size = 256
# 如出现“out of memory”的报错信息，可减小batch_size或resize
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size,
resize=96)

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 0.0020, train acc 0.834, test acc 0.749, time 27.7 sec
epoch 2, loss 0.0011, train acc 0.900, test acc 0.824, time 25.5 sec
epoch 3, loss 0.0009, train acc 0.913, test acc 0.839, time 23.8 sec
epoch 4, loss 0.0008, train acc 0.921, test acc 0.889, time 24.9 sec
epoch 5, loss 0.0008, train acc 0.929, test acc 0.884, time 24.3 sec
```

小结

- 在跨层连接上，不同于ResNet中将输入与输出相加，DenseNet在通道维上连结输入与输出。
- DenseNet的主要构建模块是稠密块和过渡层。

参考文献

(1) Huang, G., Liu, Z., Weinberger, K. Q., & van der Maaten, L. (2017). Densely connected convolutional networks. In Proceedings of the IEEE conference on computer vision and pattern recognition (Vol. 1, No. 2).

> 注：除代码外本节与原书此节基本相同，原书传送门