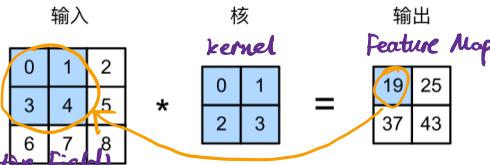


Convolutional Neural Network

→ 含有卷积层的网络

卷积层 → Extract features 在 CNN 中定义为卷积核 (kernel) 或过滤器 (Filter)

更深的网络 → 更广泛的感受野



(图 5.1 二维互相关运算)

- 卷积层中输入与核做互相关运算后加上标量偏差得到输出
- $p \times q$ 卷积层 \Leftrightarrow kernel $p \times q$
- 卷积层可通过重复使用 kernel 有效特征局部空间
- 可以通过数据学习 kernel

输入	核	输出
$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 & 0 \\ 0 & 3 & 4 & 5 & 0 \\ 0 & 6 & 7 & 8 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}$	$= \begin{bmatrix} 0 & 3 & 8 & 4 \\ 9 & 19 & 25 & 10 \\ 21 & 37 & 43 & 16 \\ 6 & 7 & 8 & 0 \end{bmatrix}$

(图 5.2 在输入的高和宽两侧分别填充了 0 元素的二维互相关计算)

填充 (Padding) (p_h, p_w) 默认 0

- 输入 $n_h \times n_w$
- Kernel $k_h \times k_w$
- 输出 $(n_h - k_h + 1) \times (n_w - k_w + 1)$
- 在高两侧共填充 $p_h = k_h - 1$ → 使输入和在宽两侧共填充 $p_w = k_w - 1$ 输出等宽等高

→ Padding 增加高和宽

输入	核	输入	核	输出
$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$	$\begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}$	$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$	$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$	$= \begin{bmatrix} 56 & 72 \\ 104 & 120 \end{bmatrix}$

(图 5.4 含 2 个输入通道的互相关计算)

输入	核	输出
$\begin{bmatrix} x_1 & x_2 \\ x_3 & x_4 \end{bmatrix}$	$\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix}$	$= \begin{bmatrix} o_1 & o_2 \\ o_3 & o_4 \end{bmatrix}$

(图 5.5 1x1 卷积核的互相关计算。输入和输出具有相同的高和宽)

卷积运算

将核数组左右翻转
轻并上下翻转，再与输入数组做互相关运算 (在卷积层中二者等价)

Input	kernel	Output
$\begin{bmatrix} 1 & -1 & -1 \\ -1 & 1 & 1 \\ -1 & -1 & 1 \end{bmatrix}$	\times	$= \begin{bmatrix} 0.11 & 0.11 & 0.11 & 0.11 & 0.11 \\ 0.11 & 0.11 & 0.11 & 0.11 & 0.11 \\ 0.11 & 0.11 & 0.11 & 0.11 & 0.11 \\ 0.11 & 0.11 & 0.11 & 0.11 & 0.11 \\ 0.11 & 0.11 & 0.11 & 0.11 & 0.11 \end{bmatrix}$
$\begin{bmatrix} 1 & -1 & 1 \\ -1 & 1 & -1 \\ 1 & -1 & 1 \end{bmatrix}$	\times	$= \begin{bmatrix} 0.11 & 0.11 & 0.11 & 0.11 & 0.11 \\ 0.11 & 0.11 & 0.11 & 0.11 & 0.11 \\ 0.11 & 0.11 & 0.11 & 0.11 & 0.11 \\ 0.11 & 0.11 & 0.11 & 0.11 & 0.11 \\ 0.11 & 0.11 & 0.11 & 0.11 & 0.11 \end{bmatrix}$
$\begin{bmatrix} -1 & -1 & 1 \\ -1 & 1 & -1 \\ 1 & -1 & -1 \end{bmatrix}$	\times	$= \begin{bmatrix} 0.11 & 0.11 & 0.11 & 0.11 & 0.11 \\ 0.11 & 0.11 & 0.11 & 0.11 & 0.11 \\ 0.11 & 0.11 & 0.11 & 0.11 & 0.11 \\ 0.11 & 0.11 & 0.11 & 0.11 & 0.11 \\ 0.11 & 0.11 & 0.11 & 0.11 & 0.11 \end{bmatrix}$

Convolutional Layer

Input	kernel	Output
$\begin{bmatrix} 0.77 & 0.11 & 0.11 & 0.33 & 0.55 & 0.11 & 0.33 \\ 0.11 & 1.00 & 0.11 & 0.33 & 0.11 & 0.11 & -0.11 \\ 0.11 & -0.11 & 1.00 & -0.33 & 0.11 & -0.11 & 0.55 \\ 0.33 & 0.33 & -0.33 & 0.55 & -0.33 & 0.33 & 0.33 \\ 0.55 & -0.11 & 0.11 & -0.33 & 1.00 & -0.11 & 0.11 \\ -0.11 & 0.11 & -0.11 & 0.33 & -0.11 & 1.00 & 0.11 \\ 0.33 & -0.11 & 0.55 & 0.33 & 0.11 & -0.11 & 0.77 \end{bmatrix}$	\times	$= \begin{bmatrix} 0.77 & 0 & 0.11 & 0.33 & 0.55 & 0 & 0.33 \\ 0 & 1.00 & 0 & 0.33 & 0 & 0.11 & 0 \\ 0.11 & 0 & 1.00 & 0 & 0.11 & 0 & 0.55 \\ 0.33 & 0.33 & 0 & 0.55 & 0 & 0.33 & 0.33 \\ 0.55 & 0 & 0.11 & 0 & 1.00 & 0 & 0.11 \\ 0 & 0.11 & 0 & 0.33 & 0 & 1.00 & 0 \\ 0.33 & 0 & 0.55 & 0.33 & 0.11 & 0 & 0.77 \end{bmatrix}$

Input	Activation Function (Relu)	Output
$\begin{bmatrix} 0.77 & 0.11 & 0.11 & 0.33 & 0.55 & 0 & 0.33 \\ 0 & 1.00 & 0 & 0.33 & 0 & 0.11 & 0 \\ 0.11 & 0 & 1.00 & 0 & 0.11 & 0 & 0.55 \\ 0.33 & 0.33 & 0 & 0.55 & 0 & 0.33 & 0.33 \\ 0.55 & 0 & 0.11 & 0 & 1.00 & 0 & 0.11 \\ 0 & 0.11 & 0 & 0.33 & 0 & 1.00 & 0 \\ 0.33 & 0 & 0.55 & 0.33 & 0.11 & 0 & 0.77 \end{bmatrix}$	\times	$= \begin{bmatrix} 0.77 & 0 & 0.11 & 0.33 & 0.55 & 0 & 0.33 \\ 0 & 1.00 & 0 & 0.33 & 0 & 0.11 & 0 \\ 0.11 & 0 & 1.00 & 0 & 0.11 & 0 & 0.55 \\ 0.33 & 0.33 & 0 & 0.55 & 0 & 0.33 & 0.33 \\ 0.55 & 0 & 0.11 & 0 & 1.00 & 0 & 0.11 \\ 0 & 0.11 & 0 & 0.33 & 0 & 1.00 & 0 \\ 0.33 & 0 & 0.55 & 0.33 & 0.11 & 0 & 0.77 \end{bmatrix}$

(图 5.3 高和宽上步幅分别为 3 和 2 的二维互相关运算)

步幅 (Stride) (S_h, S_w) 默认 1

● 形状

$$[(n_h - k_h + p_h + S_h) / S_h] \times [(n_w - k_w + p_w + S_w) / S_w]$$

$$\downarrow p_h = k_h - 1, p_w = k_w - 1$$

$$[(n_h + S_h - 1) / S_h] \times [(n_w + S_w - 1) / S_w]$$

↓ 输入的高和宽能被高和宽上的步幅整除

$$(n_h / S_h) \times (n_w / S_w)$$

→ Stride 减小高和宽

c_i c_o

多输入通道及多输出通道

- 构造输入通道数相同的卷积核 $c_i \times k_h \times k_w$
- 各通道上互相关运算再相加 → 输出通道总为 1
- 为每个输出通道分别创建 $c_i \times k_h \times k_w$ 的核数组 → 卷积核 $c_o \times c_i \times k_h \times k_w$

1x1 卷积层

无法识别高和宽维度上相邻元素构成的模式 → 在通道维上进

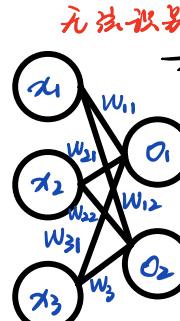
→ 将通道维作特征维，高和宽维度行卷积运算

元素作数据样本 → 1x1 卷积核 \Leftrightarrow 全连接层

高和宽上相同位置不同通道间按权重相加

用于调整网络层间的通道数，并控制模型复杂度

几个 1x1 的 filter 对应几个通道



池化层 (Pooling Layer)

缓解卷积层对位置的过度敏感性

• 二维最大池化层和平均池化层

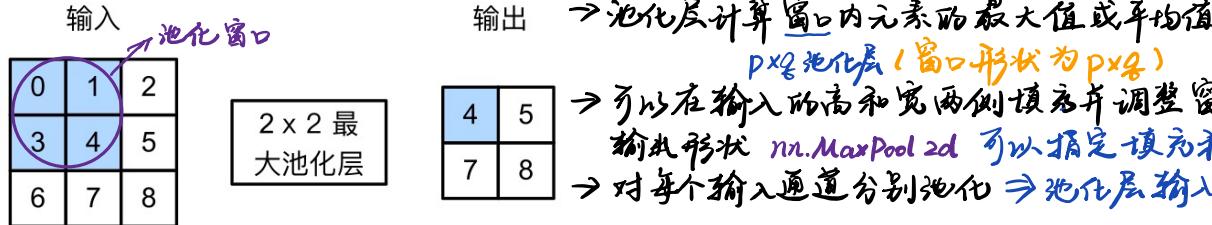


图5.6 池化窗口形状为 2×2 的最大池化

卷积神经网络 (LeNet)

→ 卷积层保留输入形状，图像像素在高和宽方向上的相关性可被有效识别

→ 滑动窗口将同一卷积核不同位置的输入重复计算，避免参数尺寸过大

LeNet 模型

• 分为卷积块和全连接层两个部分

• 卷积块 → 卷积层后接最大池化层 (批量大小、通道、高、宽)

→ 识别图像里的空间模式 (线条和物体局部)

• 5×5 窗口，步长为 1，输出上使用 sigmoid 激活函数

• 卷积层 1 输出通道数为 6 且比上一层窄

卷积层 2 输出通道数为 16 → 增加通道数使参数尺寸类似

最大池化层

→ 降低卷积层对位置敏感性

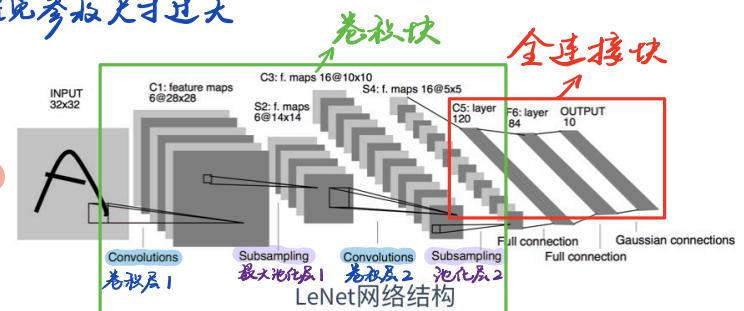
• 2×2 窗口，步幅为 2 → 池化窗口在输入上每次滑动覆盖的区域互不重叠

• 全连接层块

→ 将小批量中每个样本扁平 (flatten)

→ 由二维 (小批量中样本，扁平后向量表示) 长度为通道、高、宽乘积

• 含 3 个全连接层 120, 84, 10 类别个数



```
LeNet(  
    (conv): Sequential(  
        (0): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))  
        (1): Sigmoid()  
        (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)  
        (3): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))  
        (4): Sigmoid()  
        (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)  
    )  
    (fc): Sequential(  
        (0): Linear(in_features=256, out_features=120, bias=True)  
        (1): Sigmoid()  
        (2): Linear(in_features=120, out_features=84, bias=True)  
        (3): Sigmoid()  
        (4): Linear(in_features=84, out_features=10, bias=True)  
    )  
)
```

深度卷积神经网络 (AlexNet)

浅层 NN 和深度 NN 的分界线
端到端 (end-to-end) 直接基于图像原始像素分类

学习特征表示 分级表示特征

缺失要素 • 收据 → ImageNet 数据集

• 硬件 → GPU OpenCL, CUDA 框架

AlexNet 模型 8 层卷积神经网络

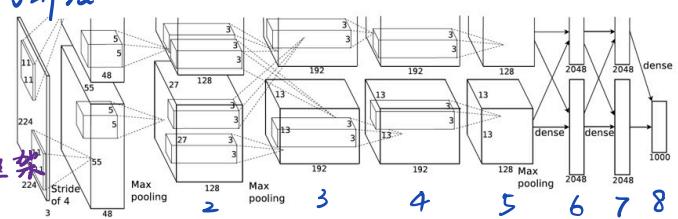
5 层卷积，2 层全连接隐藏层，1 个全连接输出层

卷积层 • 第一层 kernel 11×11，第二层 kernel 5×5，之后全部 3×3 kernel

池化层 • 一、二、五卷积层后使用最大池化层

(3×3, stride=2)

全连接层 • 最后一个卷积层后两个输出为 496 的全连接层
1 GB 模型参数



激活函数 Sigmoid → ReLU

使用 Dropout 控制全连接层模型复杂度

→ 浅层 NN 和深度 NN 的分界线

→ 没有提供简单规则设计新网络

(ReLU)
卷积层 + AF
+ (池化层)

```
AlexNet(  
    (conv): Sequential(  
        (0): Conv2d(1, 96, kernel_size=(11, 11), stride=(4, 4))  
        (1): ReLU()  
        (2): MaxPool2d(kernel_size=3, stride=2, padding=1)  
        (3): Conv2d(96, 256, kernel_size=(5, 5), stride=(2, 2))  
        (4): ReLU()  
        (5): MaxPool2d(kernel_size=3, stride=2, padding=1)  
        (6): Conv2d(256, 384, kernel_size=(3, 3), stride=(1, 1))  
        (7): ReLU()  
        (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1))  
        (9): ReLU()  
        (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))  
        (11): ReLU()  
        (12): MaxPool2d(kernel_size=3, stride=2, padding=1)  
    )  
    (fc): Sequential(  
        (0): Linear(in_features=6400, out_features=4096)  
        (1): ReLU()  
        (2): Dropout(p=0.5)  
        (3): Linear(in_features=4096, out_features=4096)  
        (4): ReLU()  
        (5): Dropout(p=0.5)  
        (6): Linear(in_features=4096, out_features=10)  
    )  
)
```

使用重复元素的网络 (VGG)

→ 提出可以通过重复使用简单基础块构建深度模型

VGG 块

- 连续使用两个相同的 Padding 为 1, kernel 为 3×3 的卷积层，保持输入高宽不变。后接 stride 为 2, kernel 为 2×2 的最大池化层，池化层对高宽减半。
- Vgg-block 指定卷积层数量和输入输出通道数。对于给定的感受野（与输出有关的输入图片的局部大小），采用堆积的小卷积核优于采用大的卷积核，因为可以增加网络深度来保证学习更复杂的模式，而且代价还比较小（参数更少）。例如，在 VGG 中，使用了 3 个 3×3 卷积核来代替 7×7 卷积核，使用了 2 个 3×3 卷积核来代替 5×5 卷积核，这样做的主要目的是在保证具有相同感知野的条件下，提升了网络的深度，在一定程度上提升了神经网络的效果。

VGG 网络 → 卷积层模块后接全连接层模块

卷积层模块

```
nv_arch = ((1, 1, 64), (1, 64, 128), (2, 128, 256), (2, 256, 512), (2, 512, 512))
经过5个vgg_block, 宽高会减半5次, 变成 224/32 = 7
```

- 串联多个 vgg-block
- 由 conv_arch 定义超参数，指定每个 VGG 块中卷积层个数和输入输出通道数
- 全连接 + ReLU + Dropout
- VGG-11 通过 5 个卷积块，3 个全连接层构造网络
→ 根据每块里卷积层个数和输出通道数定义不同的 VGG 模型
5 个 Vgg-block 含有 11 个卷积层

网络中的网络 (NiN) 使用 1×1 卷积层替代全连接层
串联多个由卷积层和全连接层的小网络构建深层网络

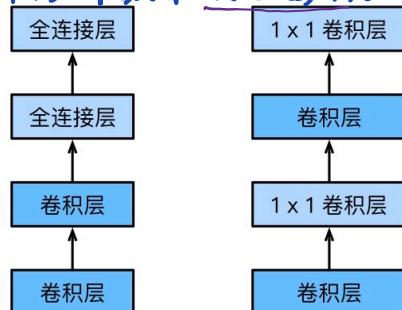


图 5.7 左图是 AlexNet 和 VGG 的网络结构局部，右图是 NiN 的网络结构局部

```
net = nn.Sequential(
    nin_block(1, 96, kernel_size=11, stride=4, padding=0),
    nn.MaxPool2d(kernel_size=3, stride=2),
    nin_block(96, 256, kernel_size=5, stride=1, padding=2),
    nn.MaxPool2d(kernel_size=3, stride=2),
    nin_block(256, 384, kernel_size=3, stride=1, padding=1),
    nn.MaxPool2d(kernel_size=3, stride=2),
    nn.Dropout(0.5),
    # 标签类别数是 10
    nin_block(384, 10, kernel_size=3, stride=1, padding=1),
    GlobalAvgPool2d(),
    # 将四维的输出转成二维的输出，其形状为(批量大小, 10)
    d2l.FlattenLayer()
```

```

Vb1 C1 64 {
  (0): Conv2d(1, 64, kernel_size=(3, 3),
  (1): ReLU()
  (2): MaxPool2d(kernel_size=2, stride=2)
}

(vgg_block_1): Sequential(
  (0): Conv2d(64, 128, kernel_size=(3, 3),
  (1): ReLU()
  (2): MaxPool2d(kernel_size=2, stride=2)
)

Vb2 C2 128 {
  (0): Conv2d(128, 128, kernel_size=(3, 3),
  (1): ReLU()
  (2): MaxPool2d(kernel_size=2, stride=2)
}

(vgg_block_2): Sequential(
  (0): Conv2d(128, 256, kernel_size=(3, 3),
  (1): ReLU()
  (2): MaxPool2d(kernel_size=2, stride=2)
)

Vb3 C3 256 {
  (0): Conv2d(256, 512, kernel_size=(3, 3),
  (1): ReLU()
  (2): Conv2d(512, 512, kernel_size=(3, 3),
  (3): ReLU()
  (4): MaxPool2d(kernel_size=2, stride=2)
)

(vgg_block_3): Sequential(
  (0): Conv2d(512, 512, kernel_size=(3, 3),
  (1): ReLU()
  (2): Conv2d(512, 512, kernel_size=(3, 3),
  (3): ReLU()
  (4): MaxPool2d(kernel_size=2, stride=2)
)

Vb4 C4 512 {
  (0): Conv2d(512, 512, kernel_size=(3, 3),
  (1): ReLU()
  (2): Conv2d(512, 512, kernel_size=(3, 3),
  (3): ReLU()
  (4): MaxPool2d(kernel_size=2, stride=2)
)

(vgg_block_4): Sequential(
  (0): Conv2d(512, 512, kernel_size=(3, 3),
  (1): ReLU()
  (2): Conv2d(512, 512, kernel_size=(3, 3),
  (3): ReLU()
  (4): MaxPool2d(kernel_size=2, stride=2)
)

Vb5 C5 512 {
  (0): Conv2d(512, 512, kernel_size=(3, 3),
  (1): ReLU()
  (2): Conv2d(512, 512, kernel_size=(3, 3),
  (3): ReLU()
  (4): MaxPool2d(kernel_size=2, stride=2)
)

(vgg_block_5): Sequential(
  (0): Conv2d(512, 512, kernel_size=(3, 3),
  (1): ReLU()
  (2): Conv2d(512, 512, kernel_size=(3, 3),
  (3): ReLU()
  (4): MaxPool2d(kernel_size=2, stride=2)
)

Vb6 C6 512 {
  (0): Conv2d(512, 512, kernel_size=(3, 3),
  (1): ReLU()
  (2): Conv2d(512, 512, kernel_size=(3, 3),
  (3): ReLU()
  (4): MaxPool2d(kernel_size=2, stride=2)
)

(vgg_block_6): Sequential(
  (0): Conv2d(512, 512, kernel_size=(3, 3),
  (1): ReLU()
  (2): Conv2d(512, 512, kernel_size=(3, 3),
  (3): ReLU()
  (4): MaxPool2d(kernel_size=2, stride=2)
)

Vb7 C7 512 {
  (0): Conv2d(512, 512, kernel_size=(3, 3),
  (1): ReLU()
  (2): Conv2d(512, 512, kernel_size=(3, 3),
  (3): ReLU()
  (4): MaxPool2d(kernel_size=2, stride=2)
)

(vgg_block_7): Sequential(
  (0): Conv2d(512, 512, kernel_size=(3, 3),
  (1): ReLU()
  (2): Conv2d(512, 512, kernel_size=(3, 3),
  (3): ReLU()
  (4): MaxPool2d(kernel_size=2, stride=2)
)

Vb8 C8 512 {
  (0): Conv2d(512, 512, kernel_size=(3, 3),
  (1): ReLU()
  (2): Conv2d(512, 512, kernel_size=(3, 3),
  (3): ReLU()
  (4): MaxPool2d(kernel_size=2, stride=2)
)

(vgg_block_8): Sequential(
  (0): Conv2d(512, 512, kernel_size=(3, 3),
  (1): ReLU()
  (2): Conv2d(512, 512, kernel_size=(3, 3),
  (3): ReLU()
  (4): MaxPool2d(kernel_size=2, stride=2)
)

F9 F10 F11

```

```

def nin_block(in_channels, out_channels, kernel_size, stride, padding):
    blk = nn.Sequential(nn.Conv2d(in_channels, out_channels, kernel_size, stride,
                               padding), nn.ReLU(),
                       nn.Conv2d(out_channels, out_channels, kernel_size=1),
                       nn.ReLU(), nn.Conv2d(out_channels, out_channels, kernel_size=1),
                       nn.ReLU())
    return blk

```

NiN 块 NiN 中的基础块

由一个卷积层加两个充当全连接层的 1×1 卷积层串联
可修改超参数 一般固定

NiN 模型 卷积层设置与 AlexNet 类似

- 卷积窗口分别为 11×11 、 5×5 、 3×3 ，相应输出通道数与 AlexNet 中一致
- 每个 NiN 块后接一个 stride 为 2, kernel 为 3×3 的最大池化层
- 去掉 AlexNet 最后 3 个全连接层 → 容易过拟合
→ 使用输出通道数等于标签类别数的 NiN 块
→ 全局平均池化层对每个通道中所有元均求平均并直接用于分类 窗口形状等于输入空间维形状的平均池化层

→ 显著减小模型参数尺寸，缓解过拟合。
→ 有时造成训练时间的增加

LeNet → AlexNet → Vgg

→ NiN → GoogleNet

↓
ResNet

↓
DenseNet

合并行连接的网络 (GoogleNet)

- 利用NIN中串联网络的思想加以改进
Inception块
- 4条并行线路 并行抽取信息
 - $1 \times 1, 3 \times 3, 5 \times 5$ 抽取不同空间尺寸下的信息
 - 将每条线路输出在通道维连接并输入后面的层中

GoogleNet 模型 → 在主体卷积部分中使用

5个模块(block), 模块间使用步幅为2的最大池化层来减小输出高宽

- 第一模块 64通道 7×7 卷积层

```
python
b1 = nn.Sequential(nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3),
                   nn.ReLU(),
                   nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

- 第二模块 64通道 1×1 卷积层
(Inception P2) 192通道 3×3 卷积层

```
python
b2 = nn.Sequential(nn.Conv2d(64, 64, kernel_size=1),
                   nn.Conv2d(64, 192, kernel_size=3, padding=1),
                   nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

- 第三模块 联接2个 Inception 块

```
python
b3 = nn.Sequential(Inception(192, 64, (96, 128), (16, 32), 32),
                   Inception(256, 128, (128, 192), (32, 96), 64),
                   nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

- 第四模块 联接5个 Inception 块

```
python
b4 = nn.Sequential(Inception(480, 192, (96, 208), (16, 48), 64),
                   Inception(512, 160, (112, 224), (24, 64), 64),
                   Inception(512, 128, (128, 256), (24, 64), 64),
                   Inception(512, 112, (144, 288), (32, 64), 64),
                   Inception(528, 256, (160, 320), (32, 128), 128),
                   nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

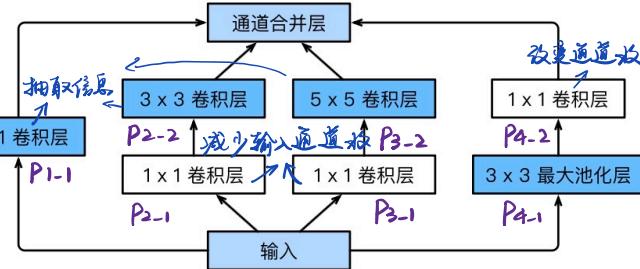


图5.8 Inception块的结构

class Inception(nn.Module):

```

# c1 ~ c4为每条线路里的层的输出通道数
def __init__(self, in_c, c1, c2, c3, c4):
    super(Inception, self).__init__()
    # 线路1, 单1x1卷积层
    self.p1_1 = nn.Conv2d(in_c, c1, kernel_size=1)
    # 线路2, 1x1卷积层后接3x3卷积层
    self.p2_1 = nn.Conv2d(in_c, c2[0], kernel_size=1)
    self.p2_2 = nn.Conv2d(c2[0], c2[1], kernel_size=3, padding=1)
    # 线路3, 1x1卷积层后接5x5卷积层
    self.p3_1 = nn.Conv2d(in_c, c3[0], kernel_size=1)
    self.p3_2 = nn.Conv2d(c3[0], c3[1], kernel_size=5, padding=2)
    # 线路4, 3x3最大池化层后接1x1卷积层
    self.p4_1 = nn.MaxPool2d(kernel_size=3, stride=1, padding=1)
    self.p4_2 = nn.Conv2d(in_c, c4, kernel_size=1)

    def forward(self, x):
        p1 = F.relu(self.p1_1(x))
        p2 = F.relu(self.p2_2(F.relu(self.p2_1(x))))
        p3 = F.relu(self.p3_2(F.relu(self.p3_1(x))))
        p4 = F.relu(self.p4_2(self.p4_1(x)))
        return torch.cat((p1, p2, p3, p4), dim=1) # 在通道维上连结输出

```

• 第五模块 2个 Inception 块

紧跟输出层 全局平均池化 通过高宽为

python

```
b5 = nn.Sequential(Inception(832, 256, (160, 320), (32, 128), 128),
                   Inception(832, 384, (192, 384), (48, 128), 128),
                   d2l.GlobalAvgPool2d())
```

```
net = nn.Sequential(b1, b2, b3, b4, b5,
                    d2l.FlattenLayer(), nn.Linear(1024, 10))
```

将输出变成二维数组后接上一个输出个数为标签类别数的全连接层

批量归一化层

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使整个神经网络在各层的中间输出的数值更稳定

全连接层

BN+AF

→ BN层置于仿射变换与AF之间

$$\begin{aligned} \alpha &= Wx + b \\ \sigma &= \phi(BN(x)) \end{aligned}$$

$$\beta = \{x^{(1)}, \dots, x^{(m)}\}$$

$$\mu_B \leftarrow \frac{1}{m} \sum x^{(i)} \text{ 均值}$$

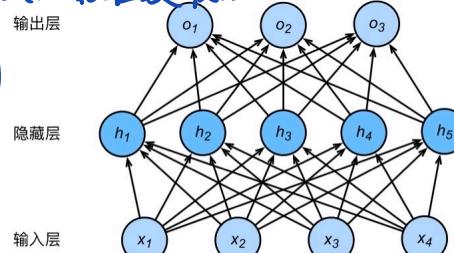
$$G_B^2 \leftarrow \frac{1}{m} \sum (x^{(i)} - \mu_B)^2 \text{ 方差}$$

$$x^{(i)} \leftarrow \frac{x^{(i)} - \mu_B}{\sqrt{G_B^2 + \epsilon}} \text{ 标准化}$$

$$\text{拉伸参数 } \gamma > 0, \text{ 保证 } \gamma > 0$$

$$y^{(i)} \leftarrow \gamma \sigma^{(i)} + \beta \text{ 偏移参数}$$

$$y^{(i)} = BN(x^{(i)}) \text{ 偏移参数 (shift)}$$



$$H = \phi(XW_h + b_h),$$

$$O = HW_o + b_o,$$

如果BN无益，模型可以不使用BN

卷积层 CONV+BN+AF

→ 在卷积计算之后，应用AF之前

- 多输出通道 → 分别做 BN
- 通道有独立 μ , σ^2 且均为标量
- 标准化时使用相同 μ 和 σ^2

($m \times p \times q$ 个参数)

预测 & 训练

- 训练时，Batch_size 设大
→ 批量内 μ 和 σ^2 更准确
- 预测时，单个样本的输出不应取决于小批量的 μ 和 σ^2

→ 移动平均估算整个训练数据集的样本 μ 和 σ^2

残差网络 (ResNet)

理想映射: $f(x) = x$ (恒等映射)

添加过多的层后训练误差不降反升 \rightarrow ResNet

残差块 (Residual Block)

- $f(x) - x$ 的权重和偏差置为0 $\rightarrow f(x)$ 恒等映射
此时残差块也易于捕捉恒等映射的细微波动
- 输入可通过跨层数据线缆更快向前进播
- 沿用 VGG 全 3×3 卷积层设计
2个有相同输出通道数的 3×3 卷积层
每层后接一个 BN 层和 ReLU 激活
输入跳过 conv 运算后, 加在 ReLU 前 \rightarrow 两层输出与输入形状一样

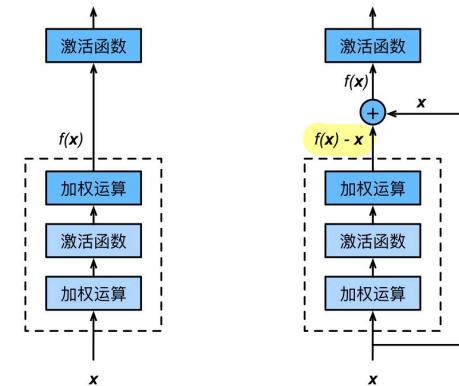


图5.9 普通的网络结构（左）与加入残差连接的网络结构（右）

可引入 1×1 卷积层改变输入形状后再做相加

```
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)

    def forward(self, X):
        Y = F.relu(self.bn1(self.conv1(X))) AF+BN+CONV
        Y = self.bn2(self.conv2(Y))
        if self.conv3:
            X = self.conv3(X)
        return F.relu(Y + X)

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)
```

net = nn.Sequential(前两层与 GoogLeNet 一样 \nwarrow
Conv+BN nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3),
+AF+NP nn.BatchNorm2d(64), 与 GoogLeNet 不同
 nn.ReLU(),
 nn.MaxPool2d(kernel_size=3, stride=2, padding=1))

ResNet 模型 (2×1 conv)

- 前两层与 GoogLeNet 相同
 \nearrow 但在 conv 及后加 BN 层 ($4 \times 2 \times 2$ conv)
- 后面使用 4 个残差块组成的模块
每个模块使用若干个同样输出通道数的残差块
 \rightarrow 第一个模块的通道数与输入通道数相同之前已使用 MaxPool, 无疑减小高宽
 \rightarrow 之后每个模块在第一个残差块里将上一个模块的通道数翻倍, 高和宽减半
 \rightarrow 每个模块包含 2 个残差块

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))

• 最后加入全局平均池化层和全连接输出

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)))

- 每个模块 4 个卷积层, 最开始的卷积层和最后的全连接层, 共 18 层 \rightarrow ResNet-18

配置不同通道数和模块里的残差块 \rightarrow 不同的 ResNet 模型

- 架构与 GoogLeNet 类似, 但结构更简单, 修改更方便

稠密连接网络 (DenseNet)

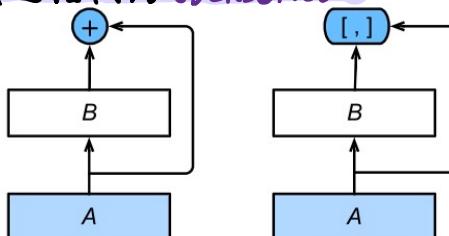


图5.10 ResNet (左) 与 DenseNet (右) 在跨层连接上的主要区别: 使用相加和使用连结

```
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)
            X = torch.cat((X, Y), dim=1) # 在通道维上将输入和输出连结
        return X
```

- \rightarrow DenseNet 模块 B 的输出直接在通道维上和 A 的连结
A 的输出可以直接进入 B 后面的层 \rightarrow 稠密连接
- 由 **稠密块 (Dense Block)** 和 **过渡层 (Transition Layer)** 构成
定义输入输出如何连结 控制通道数, 使之不过大
- \rightarrow 使用 ResNet 改良版 BN+AF+CONV 结构

```
def conv_block(in_channels, out_channels):
    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 **conv_block** 构成,
实现 BN、AF、CONV 结构
- 每块使用相同的输出通道数
输入和输出在通道维上连结
- 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 (增长率)

```
blk = DenseBlock(2, 3, 10)
X = torch.randn(4, 3, 8, 8)
Y = blk(X)
Y.shape # torch.Size([4, 23, 8, 8])
```

```

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))
    return blk

```

```

blk = transition_block(23, 10)
blk(Y).shape # torch.Size([4, 10, 4, 4])
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))
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))

```

db + transition

过渡层 → 控制模型复杂度

- 稠密块 → 通道数增加 使用过多导致复杂度上升
- 1x1卷积层减小通道数
Stride为2的Avg Pool及减半高和宽

DenseNet 模型

- 卷积层和最大池化层
- 4个稠密块

- 可以设置每块多少卷积层
- 增长率 每个conv及增加的通道数

- 全局池化层和全连接层

```

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, 24, 24])
3 output shape: torch.Size([1, 192, 24, 24])
DenseBlosk_0 output shape: torch.Size([1, 96, 12, 12])
DenseBlosk_1 output shape: torch.Size([1, 224, 12, 12])
transition_block_0 output shape: torch.Size([1, 112, 6, 6])
DenseBlosk_2 output shape: torch.Size([1, 240, 6, 6])
transition_block_1 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])

```