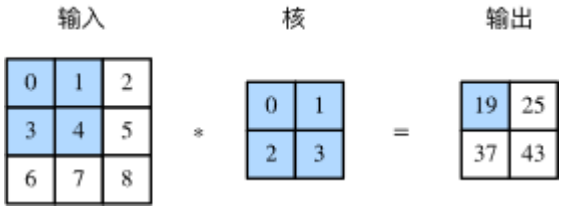
学习笔记-5: Task 5

13 - 卷积神经网络基础

二维卷积层

二维互相关运算

二维互相关(cross-correlation)运算的输入是一个二维输入数组和一个二维核(kernel)数组，输出也是一个二维数组，其中核数组通常称为卷积核或过滤器(filter)。卷积核的尺寸通常小于输入数组，卷积核在输入数组上滑动，在每个位置上，卷积核与该位置处的输入子数组按元素相乘并求和，得到输出数组中相应位置的元素。下图为一个示例。



corr2d函数实现二维互相关运算，它接受输入数组X与核数组K，并输出数组Y

```
import torch
import torch.nn as nn

def corr2d(X, K):
    H, W = X.shape  # H: height, W: width
    h, w = K.shape
    Y = torch.zeros(H - h + 1, W - w + 1)   # 输出数组Y的尺寸
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            # 按元素相乘并求和
            Y[i, j] = (X[i: i + h, j: j + w]  ×  K).sum()
    return Y
```

```
X = torch.tensor([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
K = torch.tensor([[0, 1], [2, 3]])
Y = corr2d(X, K)
print(Y)
>>> tensor([[19., 25.],
        [37., 43.]])
```

二维卷积层

二维卷积层将输入和卷积核做互相关运算，并加上一个标量偏置来得到输出。卷积层的模型参数包括卷积核和标量偏置。

```
class Conv2D(nn.Module):
    def __init__(self, kernel_size):
        super(Conv2D, self).__init__()
        """
        把要学习的参数定义成nn.Parameter
        Parameter是tensor的子类，这样定义可以自动为参数赋上梯度
        """
        self.weight = nn.Parameter(torch.randn(kernel_size))
        self.bias = nn.Parameter(torch.randn(1))

    def forward(self, x):
        # 基于boardcast机制的加法
        return corr2d(x, self.weight) + self.bias
```

互相关运算与卷积运算

卷积层得名于卷积运算，但卷积层中用到的并非卷积运算而是互相关运算。我们将核数组上下翻转、左右翻转，再与输入数组做互相关运算，这一过程就是卷积运算。由于卷积层的核数组是可学习的，所以使用互相关运算与使用卷积运算并无本质区别。

特征图与感受野

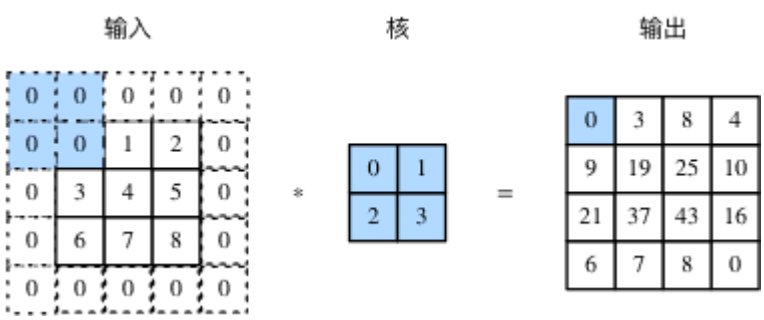
二维卷积层输出的二维数组可以看作是输入在空间维度(宽和高)上某一级的表征，也叫特征图(feature map)。影响元素 x 的前向计算的所有可能输入区域(可能大于输入的实际尺寸)叫做 x 的感受野(receptive field)。
以上面那个示意图为例，输入中阴影部分的四个元素是输出中阴影部分元素的感受野。我们将图中形状为 2×2 的输出记为 Y，将 Y 与另一个形状为 2×2 的核数组做互相关运算，输出单个元素 z。那么，z 在 Y上的感受野包括 Y 的全部四个元素，在输入上的感受野包括其中全部 9 个元素。可见，我们可以通过更深的卷积神经网络使特征图中单个元素的感受野变得更加广阔，从而捕捉输入上更大尺寸的特征。

填充和步幅

填充和步幅是卷积层的两个超参数，它们可以对给定形状的输入和卷积核改变输出形状。

填充

填充(padding)是指在输入高和宽的两侧填充元素(通常是0元素)，下图中原输入高和宽的两侧分别添加了值为0的元素。



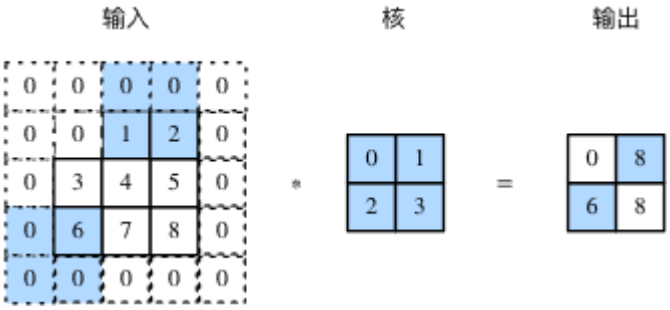
如果原输入的高和宽是n_h和n_w，卷积核的高和宽是k_h和k_w，在高的两侧一共填充p_h行，在宽的两侧一共填充p_w列，则输出形状为：

$$(n_h + p_h - k_h + 1) \times (n_w + p_w - k_w + 1)$$

在卷积神经网络中使用奇数高宽的核，比如 3×3，5×5 的卷积核，对于高度(或宽度)为大小为 $2k + 1$ 的核，令步幅为1，在高(或宽)两侧选择大小为k的填充，便可保持输入与输出尺寸相同。

步幅

在互相关运算中，卷积核在输入数组上滑动，每次滑动的行数与列数即是步幅(stride)。此前使用的步幅都是1，下图展示了在高上步幅为3、在宽上步幅为2的二维互相关运算。



一般来说，当高上步幅为s_h，宽上步幅为s_w时，输出形状为：

$$\lfloor (n_h + p_h - k_h + s_h)/s_h \rfloor \times \lfloor (n_w + p_w - k_w + s_w)/s_w \rfloor$$

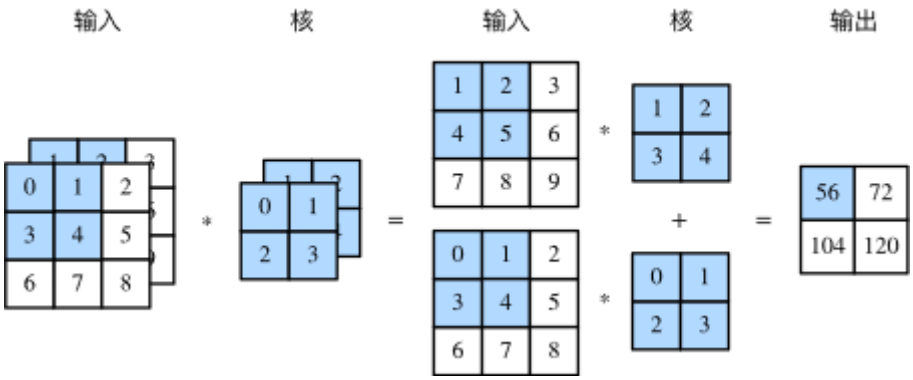
如果$p_h = k_h - 1$，$p_w = k_w - 1$，那么输出形状将简化为$\lfloor (n_h + s_h - 1)/s_h \rfloor \times \lfloor (n_w + s_w - 1)/s_w \rfloor$。更进一步，如果输入的高和宽能分别被高和宽上的步幅整除，那么输出形状将是$(n_h/s_h) \times (n_w/s_w)$。

多输入通道和多输出通道

之前的输入和输出都是二维数组，但真实数据的维度经常更高。例如，彩色图像在高和宽2个维度外还有RGB(红、绿、蓝)3个颜色通道。假设彩色图像的高和宽分别是h和w(像素)，那么它可以表示为一个$3 \times h \times w$的多维数组，我们将大小为3的这一维称为通道(channel)维。

多输入通道

卷积层的输入可以包含多个通道，下图展示了一个含2个输入通道的二维互相关计算的例子。



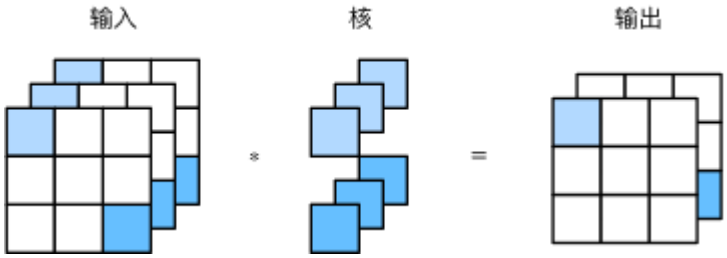
假设输入数据的通道数为c_i，卷积核形状为$k_h \times k_w$，为每个输入通道各分配一个形状为$k_h \times k_w$的核数组，将c_i个互相关运算的二维输出按通道相加，得到一个二维数组作为输出。把c_i个核数组在通道维上连结，即得到一个形状为$c_i \times k_h \times k_w$的卷积核。

多输出通道

卷积层的输出也可以包含多个通道，设卷积核输入通道数和输出通道数分别为c_i和c_o，高和宽分别为k_h和k_w。如果希望得到含多个通道的输出，可以为每个输出通道分别创建形状为$c_i \times k_h \times k_w$的核数组，将它们在输出通道维上连结，卷积核的形状即$c_o \times c_i \times k_h \times k_w$。
对于输出通道的卷积核，提供这样一种理解，一个$c_i \times k_h \times k_w$的核数组可以提取某种局部特征，但是输入可能具有相当丰富的特征，需要有多个这样的$c_i \times k_h \times k_w$的核数组，不同的核数组提取的是不同的特征。

1x1卷积层

通常称包含形状为1×1的卷积核的卷积层为1×1卷积层。下图展示了使用输入通道数为3、输出通道数为2的1×1卷积核的互相关计算。
1×1卷积核可在不改变高宽的情况下，调整通道数。1×1卷积核不识别高和宽维度上相邻元素构成的模式，其主要计算发生在通道维上。假设我们将通道维当作特征维，将高和宽维度上的元素当成数据样本，那么1×1卷积层的作用与全连接层等价。



卷积层与全连接层的对比

二维卷积层经常用于处理图像，与此前的全连接层相比，它主要有两个优势：

- 一是全连接层把图像展平成一个向量，在输入图像上相邻的元素可能因为展平操作不再相邻，网络难以捕捉局部信息。而卷积层的设计，天然地具有提取局部信息的能力。
- 二是卷积层的参数量更少。不考虑偏置的情况下，一个形状为(c_i, c_o, h, w)的卷积核的参数量是$c_i \times c_o \times h \times w$，与输入图像的宽高无关。假如一个卷积层的输入和输出形状分别是(c_1, h_1, w_1)和(c_2, h_2, w_2)，如果要用全连接层进行连接，参数数量就是$c_1 \times c_2 \times h_1 \times w_1 \times h_2 \times w_2$。使用卷积层可以以较少的参数数量来处理更大的图像。

卷积层的简洁实现

我们使用Pytorch中的`nn.Conv2d`类来实现二维卷积层，主要关注以下几个构造函数参数：

- `in_channels` (python:int) – Number of channels in the input imag
- `out_channels` (python:int) – Number of channels produced by the convolution
- `kernel_size` (python:int or tuple) – Size of the convolving kernel
- `stride` (python:int or tuple, optional) – Stride of the convolution. Default: 1
- `padding` (python:int or tuple, optional) – Zero-padding added to both sides of the input. Default: 0
- `bias` (bool, optional) – If True, adds a learnable bias to the output. Default: True

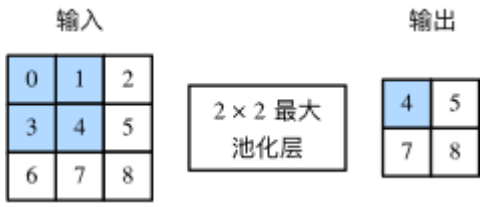
`forward`函数的参数为一个四维张量，形状为$(N, C_{in}, H_{in}, W_{in})$，返回值也是一个四维张量，形状为$(N, C_{out}, H_{out}, W_{out})$，其中$N$是批量大小，$C, H, W$分别表示通道数、高度、宽度。

```
X = torch.rand(4, 2, 3, 5)
print(X.shape)
>>> torch.Size([4, 2, 3, 5])
conv2d = nn.Conv2d(in_channels=2, out_channels=3, kernel_size=(3, 5), stride=1,
padding=(1, 2))
Y = conv2d(X)
print('Y.shape: ', Y.shape)
>>> Y.shape: torch.Size([4, 3, 3, 5])
print('weight.shape: ', conv2d.weight.shape)
>>> weight.shape: torch.Size([3, 2, 3, 5])
print('bias.shape: ', conv2d.bias.shape)
>>> bias.shape: torch.Size([3])
```

池化

二维池化层

池化层主要用于缓解卷积层对位置的过度敏感性。同卷积层一样，池化层每次对输入数据的一个固定形状窗口(池化窗口)中的元素计算输出，池化层直接计算池化窗口内元素的最大值或者平均值，该运算也分别叫做最大池化或平均池化。下图展示了池化窗口形状为2×2的最大池化。



二维平均池化的工作原理与二维最大池化类似，但将最大运算符替换成平均运算符。池化窗口形状为$p \times q$的池化层称为$p \times q$池化层，其中的池化运算叫作$p \times q$池化。
池化层也可以在输入的高和宽两侧填充并调整窗口的移动步幅来改变输出形状。池化层填充和步幅与卷积层填充和步幅的工作机制一样。
在处理多通道输入数据时，池化层对每个输入通道分别池化，但不会像卷积层那样将各通道的结果按通道相加。这意味着池化层的输出通道数与输入通道数相等。

池化层的简洁实现

使用Pytorch中的`nn.MaxPool2d`实现最大池化层，关注以下构造函数参数：

- `kernel_size` – the size of the window to take a max over
- `stride` – the stride of the window. Default value is kernel_size
- `padding` – implicit zero padding to be added on both sides

`forward`函数的参数为一个四维张量，形状为(N, C, H_{in}, W_{in})，返回值也是一个四维张量，形状为(N, C, H_{out}, W_{out})，其中N是批量大小，C, H, W分别表示通道数、高度、宽度。

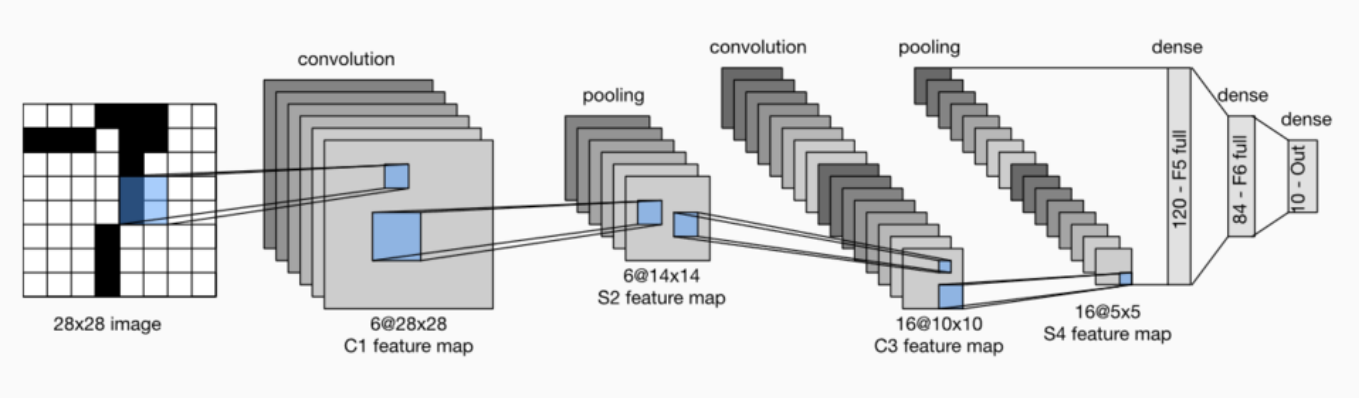
```
X = torch.arange(32, dtype=torch.float32).view(1, 2, 4, 4)
pool2d = nn.MaxPool2d(kernel_size=3, padding=1, stride=(2, 1))
Y = pool2d(X)
```

```
print(X)
>>> tensor([[[[ 0.,  1.,  2.,  3.],
          [ 4.,  5.,  6.,  7.],
          [ 8.,  9., 10., 11.],
          [12., 13., 14., 15.]],

         [[16., 17., 18., 19.],
          [20., 21., 22., 23.],
          [24., 25., 26., 27.],
          [28., 29., 30., 31.]]])
print(Y)
>>> tensor([[[[ 5.,  6.,  7.,  7.],
          [13., 14., 15., 15.]],

         [[21., 22., 23., 23.],
          [29., 30., 31., 31.]]])
```

14 - LeNet

LeNet 模型

LeNet分为卷积层块和全连接层块两个部分。

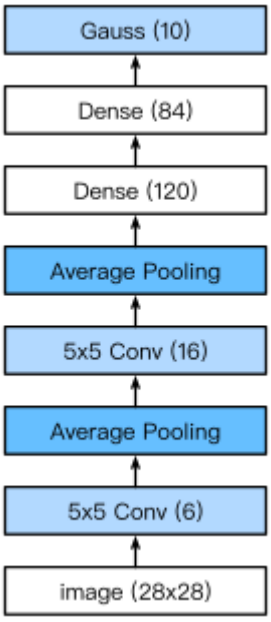


卷积层块里的基本单位是卷积层后接平均池化层：卷积层用来识别图像里的空间模式，如线条和物体局部，之后的平均池化层则用来降低卷积层对位置的敏感性。
卷积层块由两个这样的基本单位重复堆叠构成。在卷积层块中，每个卷积层都使用5×5的窗口，并在输出上使用`sigmoid`激活函数。第一个卷积层输出通道数为6，第二个卷积层输出通道数则增加到16。
全连接层块含3个全连接层。它们的输出个数分别是120、84和10，其中10为输出的类别个数。
下面通过`Sequential`类来实现`LeNet`模型。

```
class Flatten(torch.nn.Module):  # 展平操作
    def forward(self, x):
        return x.view(x.shape[0], -1)

class Reshape(torch.nn.Module): # 将图像大小重定型
    def forward(self, x):
        return x.view(-1,1,28,28)      # (B x C x H x W)

# LeNet
```

```
net = torch.nn.Sequential(
    Reshape(),
    # b × 1 × 28 × 28 => b × 6 × 28 × 28
    nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5, padding=2),
    nn.Sigmoid(),
    # b × 6 × 28 × 28 => b × 6 × 14 × 14
    nn.AvgPool2d(kernel_size=2, stride=2),
    # b × 6 × 14 × 14 => b × 16 × 10 × 10
    nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5),
    nn.Sigmoid(),
    # b × 16 × 10 × 10 => b × 16 × 5 × 5
    nn.AvgPool2d(kernel_size=2, stride=2),
    # b × 16 × 5 × 5 => b × 400
    Flatten(),
    nn.Linear(in_features=16 * 5 * 5, out_features=120),
    nn.Sigmoid(),
    nn.Linear(120, 84),
    nn.Sigmoid(),
    nn.Linear(84, 10)
)
```

在卷积层块中输入的高和宽在逐层减小。卷积层由于使用高和宽均为5的卷积核，从而将高和宽分别减小4，而池化层则将高和宽减半，但通道数则从1增加到16。全连接层则逐层减少输出个数，直到变成图像的类别数10。



15 - 卷积神经网络进阶

深度卷积神经网络 (AlexNet)

LeNet： 在大的真实数据集上的表现并不尽如人意

局限性

1. 神经网络计算复杂

2. 还没有大量深入研究参数初始化和非凸优化算法等诸多领域

特征提取

1. 机器学习的特征提取：手工定义的特征提取函数
2. 神经网络的特征提取：通过学习得到数据的多级表征，并逐级表示越来越抽象的概念或模式

AlexNet

首次证明了学习到的特征可以超越手工设计的特征

AlexNet的特点

1. 8层变换，其中有5层卷积和2层全连接隐藏层，以及1个全连接输出层
2. 将`sigmoid`激活函数改成了更加简单的`ReLU`激活函数
3. 用`Dropout`来控制全连接层的模型复杂度，防止过拟合
4. 引入数据增强，如翻转、裁剪和颜色变化，从而进一步扩大数据集来缓解过拟合

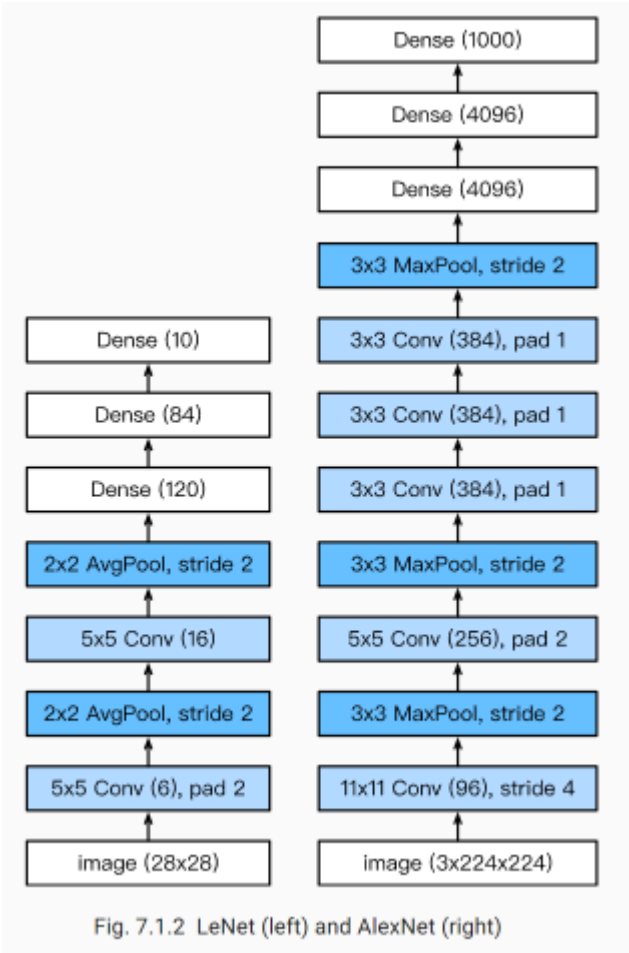


Fig. 7.1.2 LeNet (left) and AlexNet (right)

上图中最后一个卷积层的输出通道数应为**256**

```
class AlexNet(nn.Module):
    def __init__(self):
        super(AlexNet, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(1, 96, 11, 4), # in_channels, out_channels, kernel_size,
stride, padding
```

```
            nn.ReLU(),
            nn.MaxPool2d(3, 2), # kernel_size, stride
            # 减小卷积窗口，使用填充为2来使得输入与输出的高和宽一致，且增大输出通道数
            nn.Conv2d(96, 256, 5, 1, 2),
            nn.ReLU(),
            nn.MaxPool2d(3, 2),
            # 连续3个卷积层，且使用更小的卷积窗口。除了最后的卷积层外，进一步增大了输出
通道数。
            # 前两个卷积层后不使用池化层来减小输入的高和宽
            nn.Conv2d(256, 384, 3, 1, 1),
            nn.ReLU(),
            nn.Conv2d(384, 384, 3, 1, 1),
            nn.ReLU(),
            nn.Conv2d(384, 256, 3, 1, 1),
            nn.ReLU(),
            nn.MaxPool2d(3, 2)
        )
         # 这里全连接层的输出个数比LeNet中的大数倍。使用丢弃层来缓解过拟合
        self.fc = nn.Sequential(
            nn.Linear(256*5*5, 4096),
            nn.ReLU(),
            nn.Dropout(0.5),
            # 若使用CPU计算，为精简网络，可删去该层
            nn.Linear(4096, 4096),
            nn.ReLU(),
            nn.Dropout(0.5),
            # 输出层 由于这里使用Fashion-MNIST，所以用类别数为10，而非论文中的1000
            nn.Linear(4096, 10),
        )

    def forward(self, img):
        # self.conv的输出是一个4维tensor，需要变换成2维的
        feature = self.conv(img)
        output = self.fc(feature.view(img.shape[0], -1))
        return output
```

使用重复元素的网络 (VGG)

VGG：通过重复使用简单的基础块来构建深度模型
Block：数个相同的填充为1、窗口形状为3×3的卷积层，接上一个步幅为2、窗口形状为2×2的最大池化层。
卷积层保持输入的高和宽不变，而池化层则对其减半。

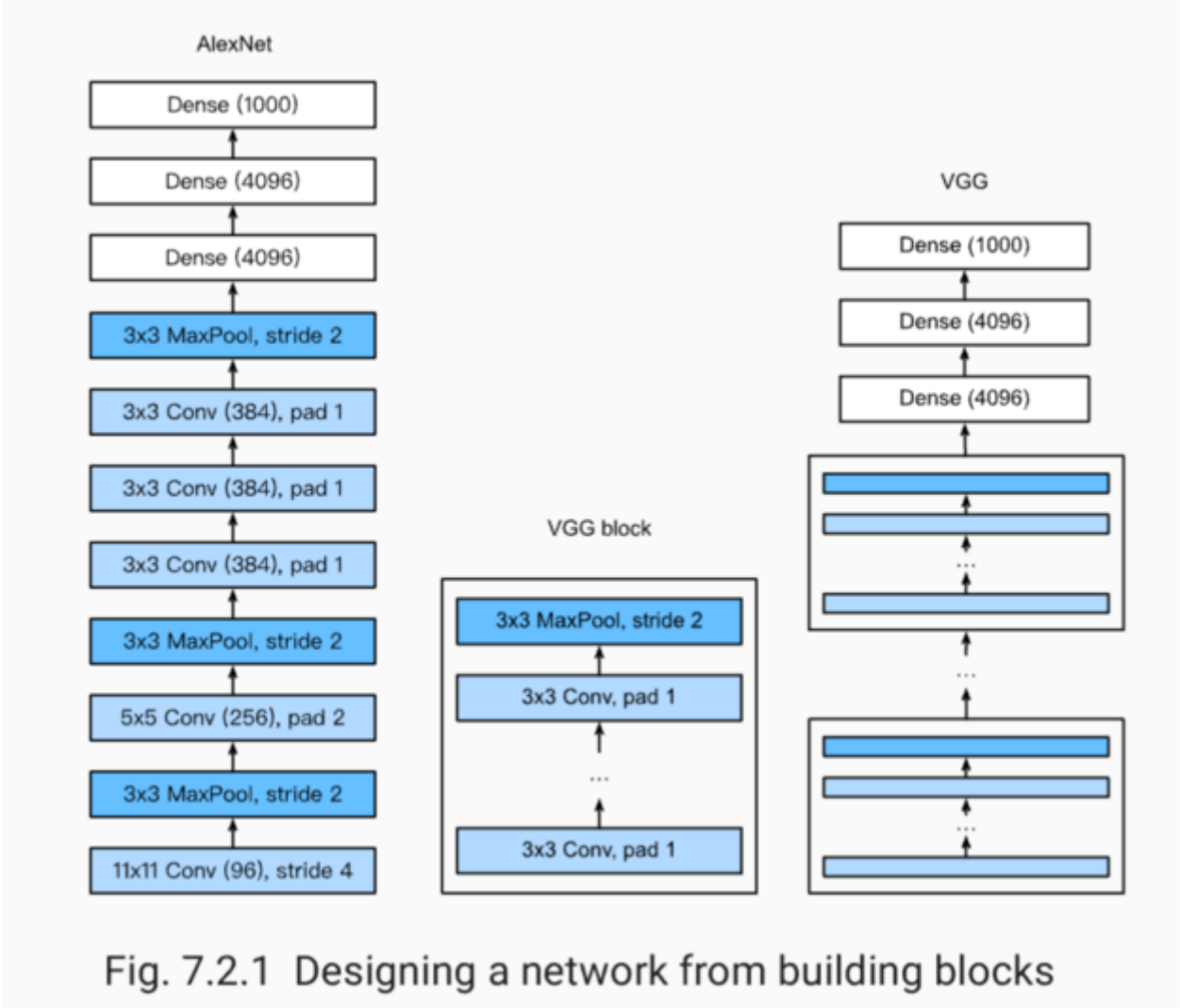


Fig. 7.2.1 Designing a network from building blocks

上图中**AlexNet**最后一个卷积的输出通道数应为**256**，**VGG block**的池化层应为**2×2**的**MaxPool**

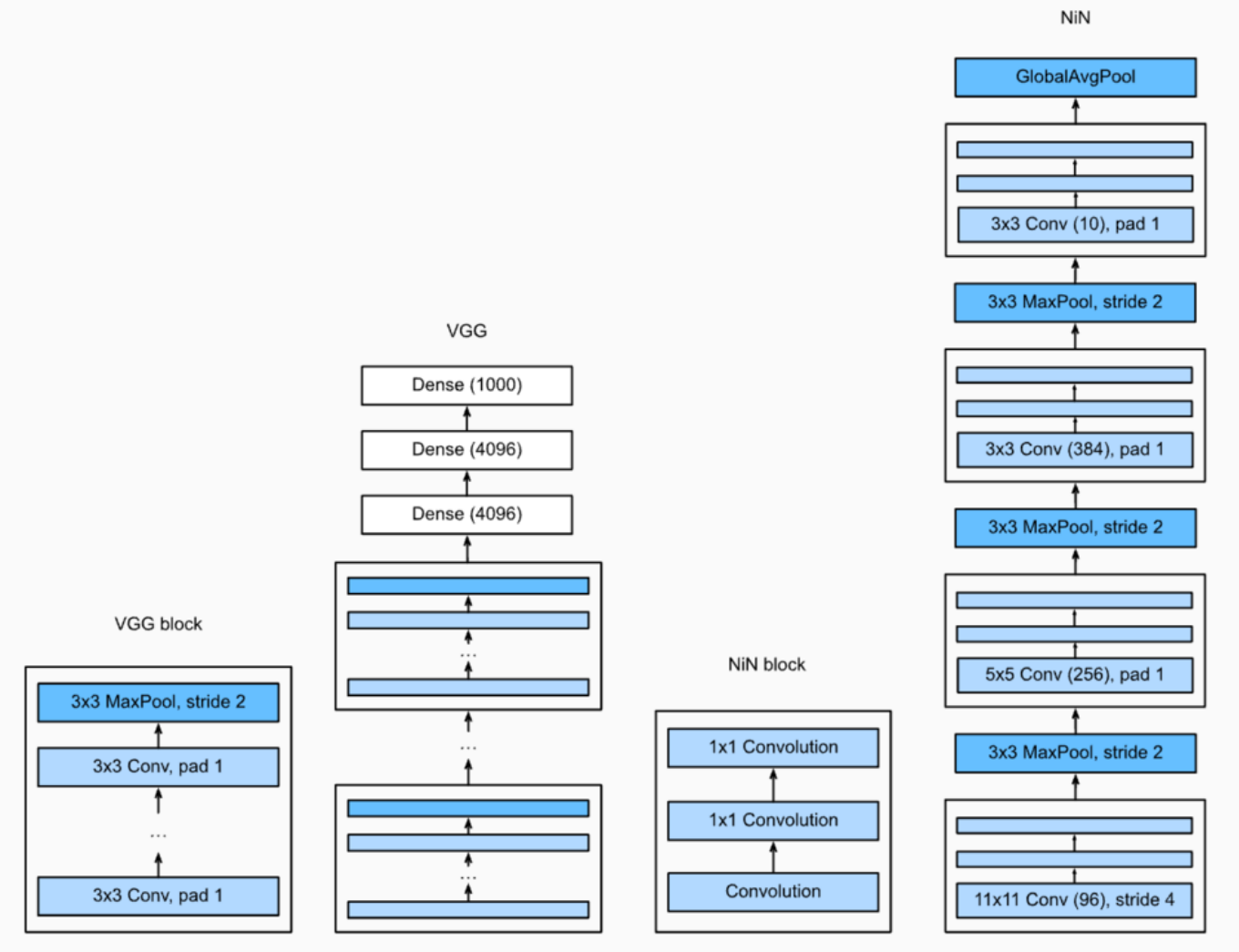
在VGG block中，卷积层保持feature map的宽和高不变，池化层使feature map的宽和高减半

```
def vgg_block(num_convs, in_channels, out_channels):
    """
    Args:
    num_convs: 卷积层个数
    in_channels: 输入通道数
    out_channels: 输出通道数
    """
    blk = []
    for i in range(num_convs):
        if i == 0:
            # 在第一个卷积层就完成输入通道数到输出通道数的转变
            blk.append(nn.Conv2d(in_channels, out_channels, kernel_size=3,
padding=1))
        else:
            blk.append(nn.Conv2d(out_channels, out_channels, kernel_size=3,
padding=1))
        blk.append(nn.ReLU())
    blk.append(nn.MaxPool2d(kernel_size=2, stride=2)) # 这里会使宽高减半
    return nn.Sequential(*blk)
```

```
def vgg(conv_arch, fc_features, fc_hidden_units=4096):
    net = nn.Sequential()
    # 卷积层部分
    for i, (num_convs, in_channels, out_channels) in enumerate(conv_arch):
        # 每经过一个vgg_block都会使宽高减半
        net.add_module("vgg_block_" + str(i+1), vgg_block(num_convs, in_channels,
out_channels))
    # 全连接层部分
    net.add_module("fc", nn.Sequential(d2l.FlattenLayer(),
                                       nn.Linear(fc_features, fc_hidden_units),
                                       nn.ReLU(),
                                       nn.Dropout(0.5),
                                       nn.Linear(fc_hidden_units, fc_hidden_units),
                                       nn.ReLU(),
                                       nn.Dropout(0.5),
                                       nn.Linear(fc_hidden_units, 10)
                                      ))
    return net
```

网络中的网络 (NiN)

LeNet、AlexNet和VGG：先以由卷积层构成的模块充分抽取空间特征，再以由全连接层构成的模块来输出分类结果。
NiN：串联多个由卷积层和“全连接”层构成的小网络来构建一个深层网络。
用了输出通道数等于标签类别数的NiN块，然后使用全局平均池化层对每个通道中所有元素求平均并直接用于分类。



上图中**NiN**第二个卷积层的**pad**应为**2**，**VGG block**的池化层应为**2×2**的**MaxPool**

1×1卷积核作用

1. 放缩通道数：通过控制卷积核的数量达到通道数的放缩
2. 增加非线性：1×1卷积核的卷积过程相当于全连接层的计算过程，并且还加入了非线性激活函数，从而可以增加网络的非线性
3. 计算参数少

```
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
```

```
class GlobalAvgPool2d(nn.Module):
    # 全局平均池化层可通过将池化窗口形状设置成输入的高和宽实现
```

```
    def __init__(self):
        super(GlobalAvgPool2d, self).__init__()
    def forward(self, x):
        return F.avg_pool2d(x, kernel_size=x.size()[2:])

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())
```

总结

- NiN重复使用由卷积层和代替全连接层的1×1卷积层构成的NiN块来构建深层网络
- NiN去除了容易造成过拟合的全连接输出层，而是将其替换成输出通道数等于标签类别数的NiN块和全局平均池化层
- NiN的以上设计思想影响了后面一系列卷积神经网络的设计

GoogLeNet

1. 由Inception基础块组成
2. Inception块相当于一个有4条线路的子网络。它通过不同窗口形状的卷积层和最大池化层来并行抽取信息，并使用1×1卷积层减少通道数从而降低模型复杂度
3. 可以自定义的超参数是每个层的输出通道数，我们以此来控制模型复杂度

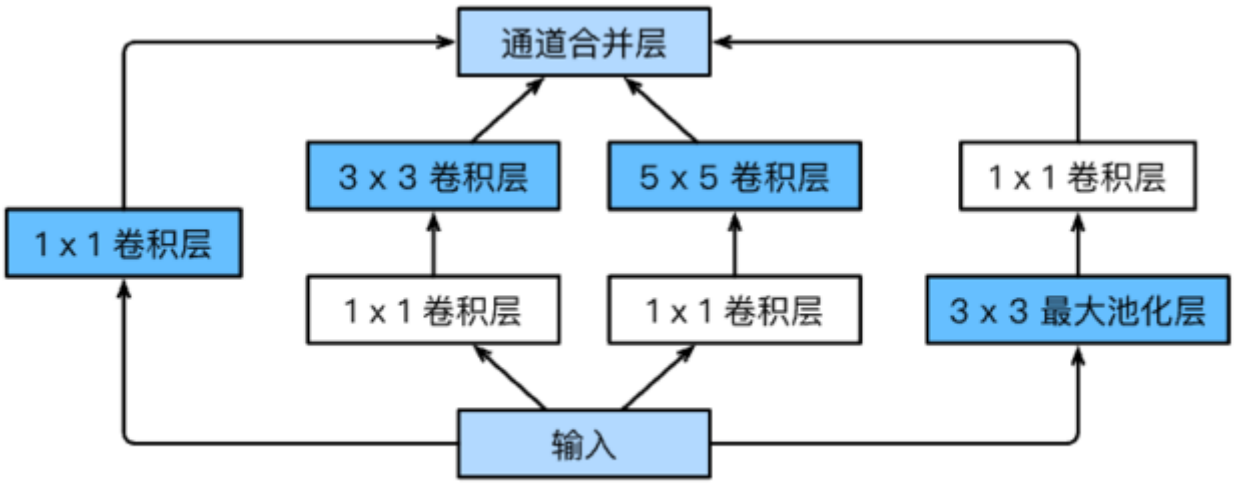


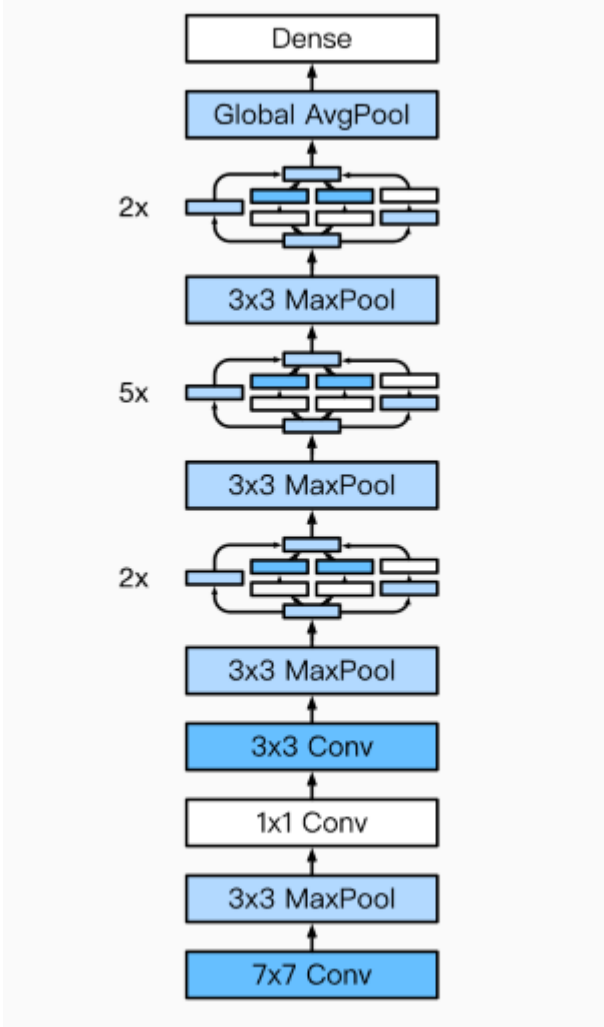
图5.8 Inception块的结构

```
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1 x 1卷积层
        self.p1_1 = nn.Conv2d(in_c, c1, kernel_size=1)
        # 线路2，1 x 1卷积层后接3 x 3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3，1 x 1卷积层后接5 x 5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4，3 x 3最大池化层后接1 x 1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
```

GoogLeNet模型

完整模型结构



```
# b1用来缩小feature map的宽和高
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))

# 从1×1 Conv到3×3 MaxPool
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))

# 2个Inception块
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))

# 5个Inception块
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))

# 2个Inception块
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))
```