

lec03 图像卷积

1. 互相关运算

卷积层其实是互相关运算（cross-correlation），而不是卷积运算。在卷积层中，输入张量和核张量通过(互相关运算)产生输出张量。

忽略通道（第三维）这一情况，则处理二维图像数据和隐藏表示如下图。输入是高度为3、宽度为3的二维张量（即形状为 3×3 ）。卷积核的高度和宽度都是2，而卷积核窗口（或卷积窗口）的形状由内核的高度和宽度决定（即 2×2 ）。

二维互相关运算。阴影部分是第一个输出元素，以及用于计算输出的输入张量元素和核张量元素：.

在二维互相关运算中，卷积窗口从输入张量的左上角开始，从左到右、从上到下滑动。当卷积窗口滑动到新一个位置时，包含在该窗口中的部分张量与卷积核张量进行按元素相乘，得到的张量再求和得到一个单一的标量值，由此得出这一位置的输出张量值。在如上例子中，输出张量的四个元素由二维互相关运算得到，这个输出高度为2、宽度为2，如下所示：

$$\begin{aligned} 0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 &= 19, \\ 1 \times 0 + 2 \times 1 + 4 \times 2 + 5 \times 3 &= 25, \\ 3 \times 0 + 4 \times 1 + 6 \times 2 + 7 \times 3 &= 37, \\ 4 \times 0 + 5 \times 1 + 7 \times 2 + 8 \times 3 &= 43. \end{aligned}$$


注意，输出大小略小于输入大小。这是因为卷积核的宽度和高度大于1，而卷积核只与图像中每个大小完全适合的位置进行互相关运算。所以，输出大小等于输入大小 $n_h \times n_w$ 减去卷积核大小 $k_h \times k_w$ ，即：

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

接下来，尝试用 `corr2d` 函数实现如上过程，该函数接受输入张量 `X` 和卷积核张量 `K`，并返回输出张量 `Y`。

```
In [1]: import torch
        from torch import nn
        from d2l import torch as d2l

In [2]: def corr2d(X, K):  #@save
        """计算二维互相关运算"""
        h, w = K.shape
        Y = torch.zeros((X.shape[0] - h + 1, X.shape[1] - w + 1))
        for i in range(Y.shape[0]):
            for j in range(Y.shape[1]):
                Y[i, j] = (X[i:i + h, j:j + w] * K).sum()
        return Y
```

通过二维互相关运算。阴影部分是第一个输出元素，以及用于计算输出的输入张量元素和核张量元素：.的输入张量 `X` 和卷积核张量 `K`，我们来验证上述二维互相关运算的输出。

```
In [3]: X = torch.tensor([[0.0, 1.0, 2.0],
                          [3.0, 4.0, 5.0],
                          [6.0, 7.0, 8.0]])
        K = torch.tensor([[0.0, 1.0],
                          [2.0, 3.0]])
        corr2d(X, K)

Out[3]: tensor([[19., 25.],
                [37., 43.]])
```

卷积层

卷积层对输入和卷积核权重进行互相关运算，并在添加标量偏置之后产生输出。所以，卷积层中的两个被训练的参数是卷积核权重和标量偏置。就像之前随机初始化全连接层一样，在训练基于卷积层的模型时也随机初始化卷积核权重。

基于上面定义的 `corr2d` 函数实现二维卷积层。在 `__init__` 构造函数中，将 `weight` 和 `bias` 声明为两个模型参数。前向传播函数调用 `corr2d` 函数并添加偏置。

```
In [4]: class Conv2D(nn.Module):
        def __init__(self, kernel_size):
            super().__init__()
            self.weight = nn.Parameter(torch.rand(kernel_size))
            self.bias = nn.Parameter(torch.zeros(1))

        def forward(self, x):
            return corr2d(x, self.weight) + self.bias
```

高度和宽度分别为 h 和 w 的卷积核可以被称为 $h \times w$ 卷积或 $h \times w$ 卷积核。将带有 $h \times w$ 卷积核的卷积层称为 $h \times w$ 卷积层。

图像中目标的边缘检测

如下是**卷积层的一个简单应用**：通过找到像素变化的位置，来**检测图像中不同颜色的边缘**。首先，我们构造一个 6×8 像素的黑白图像。中间四列为黑色（0），其余像素为白色（1）。

```
In [5]: X = torch.ones((6, 8))
X[:, 2:6] = 0
X

Out[5]: tensor([[1., 1., 0., 0., 0., 0., 1., 1.],
                [1., 1., 0., 0., 0., 0., 1., 1.],
                [1., 1., 0., 0., 0., 0., 1., 1.],
                [1., 1., 0., 0., 0., 0., 1., 1.],
                [1., 1., 0., 0., 0., 0., 1., 1.],
                [1., 1., 0., 0., 0., 0., 1., 1.]])
```

接下来，我们构造一个高度为1、宽度为2的卷积核 **K**。当进行互相关运算时，如果水平相邻的两元素相同，则输出为零，否则输出为非零。

```
In [6]: K = torch.tensor([[1.0, -1.0]])
```

现在，对参数 **X**（输入）和 **K**（卷积核）执行互相关运算。

如下所示，**输出 Y 中的1代表从白色到黑色的边缘，-1代表从黑色到白色的边缘**，其他情况的输出为0。

```
In [7]: Y = corr2d(X, K)
Y

Out[7]: tensor([[ 0.,  1.,  0.,  0.,  0., -1.,  0.],
                [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
                [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
                [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
                [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
                [ 0.,  1.,  0.,  0.,  0., -1.,  0.]])
```

现在将输入的二维图像转置，再进行如上的互相关运算。其输出如下，之前检测到的垂直边缘消失了。

所以，这个**卷积核 K 只可以检测垂直边缘**，无法检测水平边缘。

```
In [8]: corr2d(X.t(), K)

Out[8]: tensor([[0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0.]])
```

学习卷积核

如果只需寻找黑白边缘，那么以上 **[1, -1]** 的边缘检测器足以。然而，当有了更复杂数值的卷积核，或者连续的卷积层时，手动设计滤波器是不可能的。那么是否可以**学习由 X 生成 Y 的卷积核**呢？

现在看看是否可以通过仅查看“输入-输出”对来学习由 **X** 生成 **Y** 的卷积核。

先构造一个卷积层，并将其卷积核初始化为随机张量。接下来，在每次迭代中，比较 **Y** 与卷积层输出的平方误差，然后计算梯度来更新卷积核。为了简单起见，这里使用内置的二维卷积层，并忽略偏置。

```
In [9]: # 构造一个二维卷积层，它具有1个输出通道和形状为（1，2）的卷积核
conv2d = nn.Conv2d(1,1, kernel_size=(1, 2), bias=False)

# 这个二维卷积层使用四维输入和输出格式（批量大小、通道、高度、宽度），
# 其中批量大小和通道数都为1
X = X.reshape((1, 1, 6, 8))
Y = Y.reshape((1, 1, 6, 7))
lr = 3e-2 # 学习率

for i in range(10):
    Y_hat = conv2d(X)
    l = (Y_hat - Y) ** 2
    conv2d.zero_grad()
    l.sum().backward()
    # 迭代卷积核
    conv2d.weight.data[:] -= lr * conv2d.weight.grad
    if (i + 1) % 2 == 0:
        print(f'epoch {i+1}, loss {l.sum():.3f}')

epoch 2, loss 2.768
epoch 4, loss 0.527
epoch 6, loss 0.114
epoch 8, loss 0.030
epoch 10, loss 0.009
```

在10次迭代之后，误差已经降到足够低。现在查看**所学的卷积核的权重张量**。


```
In [10]: conv2d.weight.data.reshape((1, 2))

Out[10]: tensor([[ 1.0006, -0.9829]])
```

学习到的卷积核权重非常接近之前定义的卷积核 K 。

特征映射和感受野

在卷积神经网络中，对于某一层的任意元素 x ，其感受野（receptive field）是指在前向传播期间可能影响 x 计算的所有元素（来自所有先前层）。


请注意，感受野可能大于输入的实际大小。同样是对于这张图：二维互相关运算。阴影部分是第一个输出元素，以及用于计算输出的输入张量元素和核张量元素：. 给定 2×2 卷积核，阴影输出元素值19的感受野是输入阴影部分的四个元素。假设之前输出为 \mathbf{Y} ，其大小为 2×2 ，现在我们在其后附加一个卷积层，该卷积层以 \mathbf{Y} 为输入，输出单个元素 z 。在这种情况下， \mathbf{Y} 上的 z 的感受野包括 \mathbf{Y} 的所有四个元素，而输入的感受野包括最初所有九个输入元素。


因此，当一个特征图中的任意元素需要检测更广区域的输入特征时，可以构建一个更深的网络。

2. 填充和步幅

填充

在应用多层卷积时，常常丢失边缘像素。随着许多卷积层连续被应用，丢失的像素数会变得很多。

这个问题的简单方法即为填充（padding）：在输入图像的边界填充元素（通常填充元素是0）。例如，在之前的例子二维互相关运算。阴影部分是第一个输出元素，以及用于计算输出的输入张量元素和核张量元素：.中，我们将 3×3 输入填充到 5×5 ，那么它的输出就增加为 4×4 。阴影部分是第一个输出元素以及用于输出计算的输入和核张量元素： $0 \times 0 + 0 \times 1 + 0 \times 2 + 0 \times 3 = 0$ 。

带填充的二维互相关。

通常，如果添加 p_h 行填充（大约一半在顶部，一半在底部）和 p_w 列填充（左侧大约一半，右侧一半），则输出形状将为

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

这意味着输出的高度和宽度将分别增加 p_h 和 p_w 。

在许多情况下，需要设置 $p_h = k_h - 1$ 和 $p_w = k_w - 1$ ，使输入和输出**具有相同的高度和宽度**。这样可以在构建网络时更容易地预测每个图层的输出形状。

假设 k_h 是奇数，将在高度的两侧填充 $p_h/2$ 行。如果 k_h 是偶数，则一种可能性是在输入顶部填充 $\lceil p_h/2 \rceil$ 行，在底部填充 $\lfloor p_h/2 \rfloor$ 行。填充宽度的两侧同理。

卷积神经网络中卷积核的高度和宽度通常为奇数，例如1、3、5或7。选择奇数的好处是，保持空间维度的同时，可以在顶部和底部填充相同数量的行，在左侧和右侧填充相同数量的列。

比如，在下面的例子中，我们创建一个高度和宽度为3的二维卷积层，并**(在所有侧边填充1个像素)**。给定高度和宽度为8的输入，则输出的高度和宽度也是8。

```
In [11]: # 为了方便起见，我们定义了一个计算卷积层的函数。
# 此函数初始化卷积层权重，并对输入和输出提高和缩减相应的维数
def comp_conv2d(conv2d, X):
    # 这里的 (1, 1) 表示批量大小和通道数都是1
    X = X.reshape((1, 1) + X.shape)
    Y = conv2d(X)
    # 省略前两个维度：批量大小和通道
    return Y.reshape(Y.shape[2:])

# 请注意，这里每边都填充了1行或1列，因此总共添加了2行或2列
conv2d = nn.Conv2d(1, 1, kernel_size=3, padding=1)
X = torch.rand(size=(8, 8))
comp_conv2d(conv2d, X).shape

Out[11]: torch.Size([8, 8])
```

当卷积核的高度和宽度不同时，可以**填充不同的高度和宽度**，使输出和输入具有相同的高度和宽度。在如下示例中，使用高度为5，宽度为3的卷积核，高度和宽度两边的填充分别为2和1。

```
In [12]: conv2d = nn.Conv2d(1, 1, kernel_size=(5, 3), padding=(2, 1))
comp_conv2d(conv2d, X).shape


Out[12]: torch.Size([8, 8])
```


步幅

在计算互相关时，，有时候为了高效计算或是缩减采样次数，卷积窗口可以跳过中间位置，每次滑动多个元素。

每次滑动元素的数量称为**步幅**（stride）。到目前为止，我们只使用过高度或宽度为1的步幅，那么如何使用较大的步幅呢？ 下图是垂直步幅为3，水平步幅为2的二维互相关运算。 着色部分是输出元素以及用于输出计算的输入和内核张量元素：
 $0 \times 0 + 0 \times 1 + 1 \times 2 + 2 \times 3 = 8$ 、 $0 \times 0 + 6 \times 1 + 0 \times 2 + 0 \times 3 = 6$ 。

可以看到，为了计算输出中第一列的第二个元素和第一行的第二个元素，卷积窗口分别向下滑动三行和向右滑动两列。但是，当卷积窗口继续向右滑动两列时，没有输出，因为输入元素无法填充窗口（除非添加另一列填充）。

垂直步幅为 \$3\$，水平步幅为 \$2\$ 的二维互相关运算。通常，当垂直步幅为 s_h 、水平步幅为 s_w 时，输出形状为

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

如果设置了 $p_h = k_h - 1$ 和 $p_w = k_w - 1$ ，则输出形状将简化为 $\lfloor (n_h + s_h - 1) / s_h \rfloor \times \lfloor (n_w + s_w - 1) / s_w \rfloor$ 。更进一步，如果输入的高度和宽度可以被垂直和水平步幅整除，则输出形状将为 $(n_h / s_h) \times (n_w / s_w)$ 。

下面，**将高度和宽度的步幅设置为2**，从而将输入的高度和宽度减半。

```
In [13]: conv2d = nn.Conv2d(1, 1, kernel_size=3, padding=1, stride=2)
         comp_conv2d(conv2d, X).shape
```

```
Out[13]: torch.Size([4, 4])
```

一个稍微复杂的例子。

```
In [14]: conv2d = nn.Conv2d(1, 1, kernel_size=(3, 5), padding=(0, 1), stride=(3, 4))
         comp_conv2d(conv2d, X).shape
```

```
Out[14]: torch.Size([2, 2])
```

在实践中，我们很少使用不一致的步幅或填充，也就是说，我们通常有 $p_h = p_w$ 和 $s_h = s_w$ 。

3. 多输入多输出通道

到目前为止，仅展示了单个输入和单个输出通道的简化例子。于是可以将输入、卷积核和输出看作二维张量。

当添加通道时，输入和隐藏都表示都变成了三维张量。

例如，每个RGB输入图像具有 $3 \times h \times w$ 的形状。我们将这个大小为3的轴称为**通道**（channel）维度。

多输入通道

当输入包含多个通道时，需要构造一个与输入数据具有相同输入通道数的卷积核，以便与输入数据进行互相关运算。

假设输入的通道数为 c_i ，那么卷积核的输入通道数也需要为 c_i 。

当 $c_i > 1$ 时，卷积核的每个输入通道将包含形状为 $k_h \times k_w$ 的张量。将这些张量 c_i 连结在一起可以得到形状为 $c_i \times k_h \times k_w$ 的卷积核。由于输入和卷积核都有 c_i 个通道，我们可以对每个通道输入的二维张量和卷积核的二维张量进行互相关运算，再对通道求和（将 c_i 的结果相加）得到二维张量。这是多通道输入和多输入通道卷积核之间进行二维互相关运算的结果。

下图中演示了一个具有两个输入通道的二维互相关运算的示例。阴影部分是第一个输出元素以及用于计算这个输出的输入和核张量元素：
 $(1 \times 1 + 2 \times 2 + 4 \times 3 + 5 \times 4) + (0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3) = 56$ 。

两个输入通道的互相关计算。

为了加深理解**实现一下多输入通道互相关运算**。

```
In [15]: def corr2d_multi_in(X, K):
         # 先遍历 “X” 和 “K” 的第0个维度（通道维度），再把它们加在一起
         for i in range(X.shape[0]):
             if i == 0:
                 res = corr2d(X[i], K[i])
             else:
                 res += corr2d(X[i], K[i])
         return res
         # return sum(corr2d(x, k) for x, k in zip(X, K)) # corr2d
```

```
In [16]: X = torch.tensor([[[[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]],
                             [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]]]])
         K = torch.tensor([[[[0.0, 1.0], [2.0, 3.0]], [[1.0, 2.0], [3.0, 4.0]]]])

         corr2d_multi_in(X, K)
```

```
Out[16]: tensor([[ 56.,  72.],
               [104., 120.]])
```

多输出通道

对于深度神经网络来说，每一层有多个输出通道是至关重要的。

在最流行的神经网络架构中，随着神经网络层数的加深，输出通道的维数常会增加，通过减少空间分辨率以获得更大的通道深度。直观地说可以将每个通道看作是对不同特征的响应。而现实可能更为复杂一些，因为每个通道不是独立学习的，而是为了共同使用而优化的。因此，多输出通道并不仅是学习多个单通道的检测器。

用 c_i 和 c_o 分别表示输入和输出通道的数目，并让 k_h 和 k_w 为卷积核的高度和宽度。为了获得多个通道的输出，可以为每个输出通道创建一个形状为 $c_i \times k_h \times k_w$ 的卷积核张量，这样卷积核的形状是 $c_o \times c_i \times k_h \times k_w$ 。在互相关运算中，每个输出通道先获取所有输入通道，再以对应该输出通道的卷积核计算出结果。

如下所示，我们实现一个**计算多个通道的输出的互相关函数**。

```
In [17]: def corr2d_multi_in_out(X, K):
# 迭代“K”的第0个维度，每次都对输入“X”执行互相关运算。
# 最后将所有结果都叠加在一起
return torch.stack([corr2d_multi_in(X, k) for k in K], 0)
```

通过将核张量 **K** 与 **K+1** （**K** 中每个元素加1）和 **K+2** 连接起来，构造了一个具有3个输出通道的卷积核。

```
In [18]: K = torch.stack((K, K + 1, K + 2), 0)
K.shape
```

```
Out[18]: torch.Size([3, 2, 2, 2])
```

```
In [19]: corr2d_multi_in_out(X, K)
```

```
Out[19]: tensor([[[ 56.,  72.],
                  [104., 120.]],

                [[ 76., 100.],
                  [148., 172.]],

                [[ 96., 128.],
                  [192., 224.]])
```

1 × 1 卷积层

1 × 1卷积，即 $k_h = k_w = 1$ ，看起来似乎没有多大意义。

毕竟，卷积的本质是有效提取相邻像素间的相关特征，而1 × 1卷积显然没有此作用。 尽管如此，1 × 1仍然十分流行，经常包含在复杂深层网络的设计中。

因为使用了最小窗口，1 × 1卷积失去了卷积层的特有能力——在高度和宽度维度上，识别相邻元素间相互作用的能力。所以1 × 1卷积的唯一计算发生在通道上。

下图展示了使用1 × 1卷积核与3个输入通道和2个输出通道的互相关计算。

这里输入和输出具有相同的高度和宽度，输出中的每个元素都是从输入图像中同一位置的元素的线性组合。 可以将1 × 1卷积层看作是在每个像素位置应用的全连接层，以 c_i 个输入值转换为 c_o 个输出值。 因为这仍然是一个卷积层，所以跨像素的权重是一致的。同时，1 × 1卷积层需要的权重维度为 $c_o \times c_i$ ，再额外加上一个偏置。

互相关计算使用了具有3个输入通道和2个输出通道的 1×1 卷积核。其中，输入和输出具有相同的高度和宽度。

下面，使用全连接层实现1 × 1卷积。 请注意，输入和输出的数据形状需要进行调整。

```
In [20]: def corr2d_multi_in_out_1x1(X, K):
c_i, h, w = X.shape
c_o = K.shape[0]
X = X.reshape((c_i, h * w))
K = K.reshape((c_o, c_i))
# 全连接层中的矩阵乘法
Y = torch.matmul(K, X)
return Y.reshape((c_o, h, w))
```

当执行1 × 1卷积运算时，上述函数相当于先前实现的互相关函数 **corr2d_multi_in_out** 。

```
In [21]: X = torch.normal(0, 1, (3, 3, 3))
K = torch.normal(0, 1, (2, 3, 1, 1))
```

```
In [22]: Y1 = corr2d_multi_in_out_1x1(X, K)
Y2 = corr2d_multi_in_out(X, K)
assert float(torch.abs(Y1 - Y2).sum()) < 1e-6
```

3. 池化层

最大池化层和平均池化层

与卷积层类似，池化层运算符由一个固定形状的窗口组成，该窗口根据其步幅大小在输入的所有区域上滑动，为固定形状窗口（有时称为**池化窗口**）遍历的每个位置计算一个输出。

然而，不同于卷积层中的输入与卷积核之间的互相关计算，池化层**不包含参数**。

相反，池运算是确定性的，我们通常计算池化窗口中所有元素的最大值或平均值。这些操作分别称为**最大池化层**（maximum pooling）和**平均池化层**（average pooling）。

在这两种情况下，与互相关运算符一样，池化窗口从输入张量的左上角开始，从左往右、从上往下的在输入张量内滑动。在池化窗口到达的每个位置，它计算该窗口中输入子张量的最大值或平均值。计算最大值或平均值是取决于使用了最大池化层还是平均池化层。

池化窗口形状为 2×2 的最大池化层。着色部分是第一个输出元素，以及用于计算这个输出的输入元素: $\max(0, 1, 3, 4)=4$ 。

上图中的输出张量的高度为2，宽度为2。这四个元素为每个池化窗口中的最大值：

$$\begin{aligned}\max(0, 1, 3, 4) &= 4, \\ \max(1, 2, 4, 5) &= 5, \\ \max(3, 4, 6, 7) &= 7, \\ \max(4, 5, 7, 8) &= 8.\end{aligned}$$


池化窗口形状为 $p \times q$ 的池化层称为 $p \times q$ 池化层，池化操作称为 $p \times q$ 池化。

下面的代码中实现 `pool2d` 函数的**池化层的前向传播**。

这类似于之前例子中的 `corr2d` 函数。然而，这里没有卷积核，输出为输入中每个区域的最大值或平均值。

```
In [23]: def pool2d(X, pool_size, mode='max'):
        p_h, p_w = pool_size
        Y = torch.zeros((X.shape[0] - p_h + 1, X.shape[1] - p_w + 1))
        for i in range(Y.shape[0]):
            for j in range(Y.shape[1]):
                if mode == 'max':
                    Y[i, j] = X[i: i + p_h, j: j + p_w].max()
                elif mode == 'avg':
                    Y[i, j] = X[i: i + p_h, j: j + p_w].mean()
        return Y
```

我们可以构建

池化窗口形状为 2×2 的最大池化层。着色部分是第一个输出元素，以及用于计算这个输出的输入元素: $\max(0, 1, 3, 4)=4$ 。中的输入张量 `X`，**验证二维最大池化层的输出**。

```
In [24]: X = torch.tensor([[0.0, 1.0, 2.0],
                          [3.0, 4.0, 5.0],
                          [6.0, 7.0, 8.0]])
        pool2d(X, (2, 2))
```

```
Out[24]: tensor([[4., 5.],
                [7., 8.]])
```

此外，还可以验证**平均池化层**。

```
In [25]: pool2d(X, (2, 2), 'avg')
```

```
Out[25]: tensor([[2., 3.],
                [5., 6.]])
```

填充和步幅

与卷积层一样，池化层也可以改变输出形状。和以前一样，可以通过填充和步幅以获得所需的输出形状。

下面，用深度学习框架中内置的二维最大池化层，来演示汇聚层中填充和步幅的使用。

首先构造了一个输入张量 `X`，它有四个维度，其中样本数和通道数都是1。

```
In [26]: X = torch.arange(16, dtype=torch.float32).reshape((1, 1, 4, 4))
        X
```

```
Out[26]: tensor([[[[ 0.,  1.,  2.,  3.],
                    [ 4.,  5.,  6.,  7.],
                    [ 8.,  9., 10., 11.],
                    [12., 13., 14., 15.]]]])
```


默认情况下，**深度学习框架中的步幅与池化窗口的大小相同**。因此，如果使用形状为 (3, 3) 的池化窗口，那么默认情况下，得到的步幅形状为 (3, 3)。

```
In [27]: pool2d = nn.MaxPool2d(3)
         pool2d(X)

Out[27]: tensor([[[[10.]]]])
```

填充和步幅可以手动设定。

```
In [28]: pool2d = nn.MaxPool2d(3, padding=1, stride=2)
         pool2d(X)

Out[28]: tensor([[[[ 5.,  7.],
                    [13., 15.]]]])
```

可以设定一个任意大小的矩形池化窗口，并分别设定填充和步幅的高度和宽度。

```
In [29]: pool2d = nn.MaxPool2d((2, 3), stride=(2, 3), padding=(0, 1))
         pool2d(X)

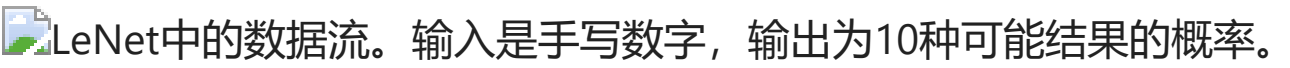
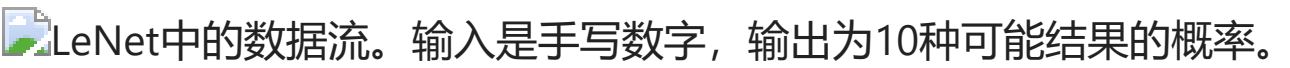
Out[29]: tensor([[[[ 5.,  7.],
                    [13., 15.]]]])
```

4. 卷积神经网络（LeNet）

总体来看，**LeNet（LeNet-5）由两个部分组成：**

- 卷积编码器：由两个卷积层组成;
- 全连接层密集块：由三个全连接层组成。

该架构如下图所示。



每个卷积块中的基本单元是一个卷积层、一个sigmoid激活函数和平均池化层。

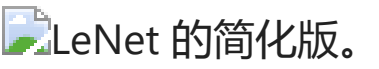
每个卷积层使用 5×5 卷积核和一个sigmoid激活函数。这些层将输入映射到多个二维特征输出，通常同时增加通道的数量。第一卷积层有6个输出通道，而第二个卷积层有16个输出通道。每个 2×2 池操作（步骤2）通过空间下采样将维数减少4倍。卷积的输出形状由批量大小、通道数、高度、宽度决定。

为了将卷积块的输出传递给稠密块，每个样本必须在小批量中展平。换言之，将这个四维输入转换成全连接层所期望的二维输入。这里的二维表示的第一个维度索引小批量中的样本，第二个维度给出每个样本的平面向量表示。LeNet的稠密块有三个全连接层，分别有120、84和10个输出。因为我们在执行分类任务，所以输出层的10维对应于最后输出结果的数量。

现在使用深度学习框架实现LeNet

```
In [30]: net = nn.Sequential(
         nn.Conv2d(1, 6, kernel_size=5, padding=2), nn.Sigmoid(),
         nn.AvgPool2d(kernel_size=2, stride=2),
         nn.Conv2d(6, 16, kernel_size=5), nn.Sigmoid(),
         nn.AvgPool2d(kernel_size=2, stride=2),
         nn.Flatten(),
         nn.Linear(16 * 5 * 5, 120), nn.Sigmoid(),
         nn.Linear(120, 84), nn.Sigmoid(),
         nn.Linear(84, 10))
```

这个模型和原始模型相比，做了一点小改动，去掉了最后一层的高斯激活。除此之外，这个网络与最初的LeNet-5一致。



下面，将一个大小为 28×28 的单通道（黑白）图像通过LeNet。

```
In [31]: X = torch.rand(size=(1, 1, 28, 28), dtype=torch.float32)
         for layer in net:
             X = layer(X)
             print(layer.__class__.__name__, 'output shape: \t', X.shape)
```

```
Conv2d output shape:      torch.Size([1, 6, 28, 28])
Sigmoid output shape:     torch.Size([1, 6, 28, 28])
AvgPool2d output shape:   torch.Size([1, 6, 14, 14])
Conv2d output shape:      torch.Size([1, 16, 10, 10])
Sigmoid output shape:     torch.Size([1, 16, 10, 10])
AvgPool2d output shape:   torch.Size([1, 16, 5, 5])
Flatten output shape:     torch.Size([1, 400])
Linear output shape:      torch.Size([1, 120])
Sigmoid output shape:     torch.Size([1, 120])
Linear output shape:      torch.Size([1, 84])
Sigmoid output shape:     torch.Size([1, 84])
Linear output shape:      torch.Size([1, 10])
```

模型训练

现在已经实现了LeNet，使用**LeNet在Fashion-MNIST数据集上进行训练**。

```
In [32]: batch_size = 256
         train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size=batch_size)
```

虽然卷积神经网络的参数较少，但与深度的多层感知机相比，它们的计算成本仍然很高，因为每个参数都参与更多的乘法。如果有机会使用GPU，可以用它加快训练。

由于完整的数据集位于内存中，因此在模型使用GPU计算数据集之前，需要将其复制到显存中。

```
In [33]: def evaluate_accuracy_gpu(net, data_iter, device=None): #@save
         """使用GPU计算模型在数据集上的精度"""
         net.eval() # 设置为评估模式
         if not device:
             device = net.to(device)
         # 正确预测的数量，总预测的数量
         metric = d2l.Accumulator(2)
         with torch.no_grad():
             for X, y in data_iter:
                 if isinstance(X, list):
                     # BERT微调所需的（之后将介绍）
                     X = [x.to(device) for x in X]
                 else:
                     X = X.to(device)
                 y = y.to(device)
                 metric.add(d2l.accuracy(net(X), y), y.numel())
         return metric[0] / metric[1]
```

与之前的训练函数不同，在进行正向和反向传播之前，需要将每一小批量数据移动到我们指定的设备（例如GPU）上。

如下训练函数 `train`。

由于实现的是多层神经网络，因此将主要使用高级API。

```
In [34]: def train(net, train_iter, test_iter, num_epochs, lr, device):
         """用GPU训练模型"""
         def init_weights(m):
             if type(m) == nn.Linear or type(m) == nn.Conv2d:
                 nn.init.xavier_uniform_(m.weight)
         net.apply(init_weights)
         print('training on', device)
         net.to(device)
         optimizer = torch.optim.SGD(net.parameters(), lr=lr)
         loss = nn.CrossEntropyLoss()
         animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs],
                                 legend=['train loss', 'train acc', 'test acc'])
         timer, num_batches = d2l.Timer(), len(train_iter)
         for epoch in range(num_epochs):
             # 训练损失之和，训练准确率之和，样本数
             metric = d2l.Accumulator(3)
             net.train()
             for i, (X, y) in enumerate(train_iter):
                 timer.start()
                 optimizer.zero_grad()
                 X, y = X.to(device), y.to(device)
                 y_hat = net(X)
                 l = loss(y_hat, y)
                 l.backward()
                 optimizer.step()
                 with torch.no_grad():
                     metric.add(l * X.shape[0], d2l.accuracy(y_hat, y), X.shape[0])
                 timer.stop()
                 train_l = metric[0] / metric[2]
                 train_acc = metric[1] / metric[2]
                 if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
                     animator.add(epoch + (i + 1) / num_batches,
                                   (train_l, train_acc, None))
             test_acc = evaluate_accuracy_gpu(net, test_iter, device)
             animator.add(epoch + 1, (None, None, test_acc))
```



```
print(f'loss {train_l:.3f}, train acc {train_acc:.3f}, '
      f'test acc {test_acc:.3f}')
print(f'{metric[2] * num_epochs / timer.sum():.1f} examples/sec '
      f'on {str(device)}')
```

现在, **训练和评估LeNet-5模型**。

```
In [ ]: lr, num_epochs = 0.9, 10
        train(net, train_iter, test_iter, num_epochs, lr, 'cuda:0')
```