Artificial Intelligence Experimental Manual

人工智能实验课程手册（下册）

作者：Yunlong Yu

组织：浙江大学信电学院

时间：July 10, 2023

浙江大学

目录

第 1 章 卷积神经网络

1.1 从全连接层到卷积

我们之前讨论的多层感知机十分适合处理表格数据，其中行对应样本，列对应特征。对于表格数据，我们寻找的模式可能涉及特征之间的交互，但是我们不能预先假设任何与特征交互相关的先验结构。此时，多层感知机可能是最好的选择，然而对于高维感知数据，这种缺少结构的网络可能会变得不实用。

例如，在之前猫狗分类的例子中：假设我们有一个足够充分的照片数据集，数据集中是拥有标注的照片，每张照片具有百万级像素，这意味着网络的每次输入都有一百万个维度。即使将隐藏层维度降低到 1000，这个全连接层也将有 $10^6 \times 10^3 = 10^9$ 个参数。想要训练这个模型将不可实现，因为需要有大量的 GPU、分布式优化训练的经验和超乎常人的耐心。

有些人可能会反对这个观点，认为要求百万像素的分辨率可能不是必要的。然而，即使分辨率减小为十万像素，使用 1000 个隐藏单元的隐藏层也可能不足以学习到良好的图像特征，在真实的系统中我们仍然需要数十亿个参数。此外，拟合如此多的参数还需要收集大量的数据。然而，如今人类和机器都能很好地区分猫和狗：这是因为图像中本就拥有丰富的结构，而这些结构可以被人类和机器学习模型使用。卷积神经网络（convolutional neural networks，CNN）是机器学习利用自然图像中一些已知结构的创造性方法。

1.2 图像卷积

1.2.1 互相关运算

严格来说，卷积层是个错误的叫法，因为它所表达的运算其实是互相关运算（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。

首先，我们暂时忽略通道（第三维）这一情况，看看如何处理二维图像数据和隐藏表示。在图1.1中，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）。

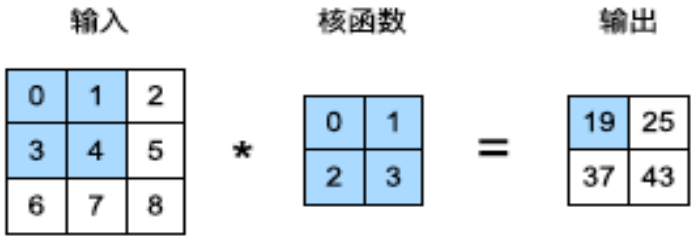


图 1.1: 二维互相关运算。阴影部分是第一个输出元素，以及用于计算输出的输入张量元素和核张量元素：$0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19$。

在二维互相关运算中，卷积窗口从输入张量的左上🄴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，由此我们得出了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：

$$0\times0+1\times1+3\times2+4\times3=19; \tag{1.1}$$

$$1\times0+2\times1+4\times2+5\times3=25; \tag{1.2}$$

$$3\times0+4\times1+6\times2+7\times3=37; \tag{1.3}$$

$$4\times0+5\times1+7\times2+8\times3=43: \tag{1.4}$$

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：

$$(n_h - k_h + 1) \times (n_w - k_w + 1). \tag{1.5}$$

这是因为我们需要足够的空间在图像上“移动”卷积核。稍后，我们将看到如何通过在图像边界周围填充零来保证有足够的空间移动卷积核，从而保持输出大小不变。接下来，我们在 corr2d 函数中实现如上过程，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。

```
import torch
from torch import nn
from d2l import torch as d2l
```

```
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
```

通过图1.1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。

```
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)
```

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

1.2.2 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。就像我们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，我们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。

```
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。我们也将带有 $h \times w$ 卷积核的卷积层称为 $h \times w$ 卷积层。

1.2.3 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）。

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

```
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。

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

现在，我们对参数 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。

```
Y = corr2d(X, K)
Y
```

```
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.]])
```

现在我们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。不出所料，这个卷积核 K 只可以检测垂直边缘，无法检测水平边缘。

```
corr2d(X.t(), K)
```

```
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.]])
```

1.2.4 学习卷积核

如果我们只需寻找黑白边缘，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，我们不可能手动设计滤波器。那么我们是否可以学习由 X 生成 Y 的卷积核呢?

现在让我们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。我们先构造一个卷积层，并将其卷积核初始化为随机张量。接下来，在每次迭代中，我们比较 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，我们在此使用内置的二维卷积层，并忽略偏置。

```
# 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 15.223
epoch 4, loss 5.325
epoch 6, loss 2.029
epoch 8, loss 0.805
epoch 10, loss 0.326
```

在 10 次迭代之后，误差已经降到足够低。现在我们来看看我们所学的卷积核的权重张量。

```
conv2d.weight.data.reshape((1, 2))
```

```
tensor([[ 1.0486, -0.9313]])
```

我们学习到的卷积核权重非常接近我们之前定义的卷积核 K。

1.2.5 特征映射和感受野

图1.1中输出的卷积层有时被称为特征映射（feature map），因为它可以被视为一个输入映射到下一层的空间维度的转换器。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。让我们用图1.1为例来解释感受野：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 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 Y 为输入，输出单个元素 z。在这种情况下，Y 上的 z 的感受野包括 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，我们可以构建一个更深的网络。

1.3 填充和步幅

在前面的例子图1.1中，输入的高度和宽度都为 3，卷积核的高度和宽度都为 2，生成的输出表征的维数为 2×2。假设输入形状为 $n_h\times n_w$，卷积核形状为 $k_h\times k_w$，那么输出形状将是 $(n_h-k_h+1)\times(n_w-k_w+1)$。因此，卷积的输出形状取决于输入形状和卷积核的形状。

还有什么因素会影响输出的大小呢？本节我们将介绍填充（padding）和步幅（stride）。假设以下情景：有时，在应用了连续的卷积之后，我们最终得到的输出远小于输入大小。这是由于卷积核的宽度和高度通常大于 1 所导致的。比如，一个 240×240 像素的图像，经过 10 层 5×5 的卷积后，将减少到 200×200 像素。如此一来，原始图像的边界丢失了许多有用信息。而填充是解决此问题最有效的方法；有时，我们可能希望大幅降低图像的宽度和高度。例如，如果我们发现原始的输入分辨率十分冗余。步幅则可以在这类情况下提供帮助。

1.3.1 填充

如上所述，在应用多层卷积时，我们常常丢失边缘像素。由于我们通常使用小卷积核，因此对于任何单个卷积，我们可能只会丢失几个像素。但随着我们应用许多连续卷积层，累积丢失的像素数就多了。解决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，在图1.1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：$0\times0+0\times1+0\times2+0\times3=0$。

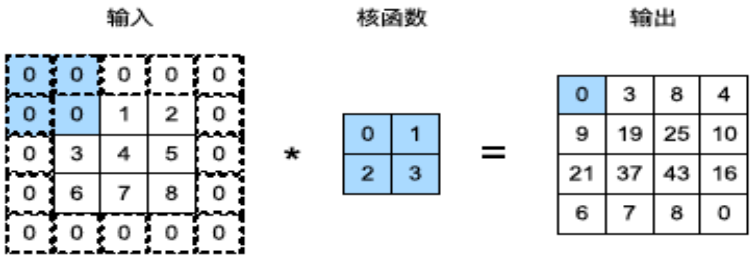


图 1.2: 带填充的二维互相关。

通常，如果我们添加 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:

$$(n_h-k_h+p_h+1)\times(n_w-k_w+p_w+1) \tag{1.6}$$

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，我们需要设置 $p_h=k_h-1$ 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，我们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 $p_h/2$ 行，在底部填充 $p_h/2$ 行。同理，我们填充宽度的两侧。

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，我们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。

此外，使用奇数的核大小和填充大小也提供了书写上的便利。对于任何二维张量 X，当满足：1. 卷积核的大小是奇数；2. 所有边的填充行数和列数相同；3. 输出与输入具有相同高度和宽度则可以得出：输出 $Y[i,j]$ 是

通过以输入 $X[i,j]$ 为中心，与卷积核进行互相关计算得到的。

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。

```
import torch
from torch import nn
# 为了方便起F，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
```

```
torch.Size([8, 8])
```

当卷积核的高度和宽度不同时，我们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，我们使用高度为 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。

```
conv2d = nn.Conv2d(1, 1, kernel_size=(5, 3), padding=(2, 1))
comp_conv2d(conv2d, X).shape
```

```
torch.Size([8, 8])
```

1.4 步幅

在计算互相关时，卷积窗口从输入张量的左上F开始，向下、向右滑动。在前面的例子中，我们默认每次滑动一个元素。但是，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。

我们将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？图1.2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：$0\times0+0\times1+1\times2+2\times3=8$、$0\times0+6\times1+0\times2+0\times3=6$。

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（除非我们添加另一列填充）。

通常，当垂直步幅为 s_h、水平步幅为 s_w 时，输出形状为

$$(n_h - k_h + p_h + s_h)/s_h \times (n_w - k_w + p_w + s_w)/s_w. \tag{1.7}$$

如果我们设置了 $p_h = k_h - 1$ 和 $p_w = k_w - 1$，则输出形状将简化为 $(n_h + s_h - 1)/s_h \times (n_w + s_w - 1)/s_w$。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，我们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。

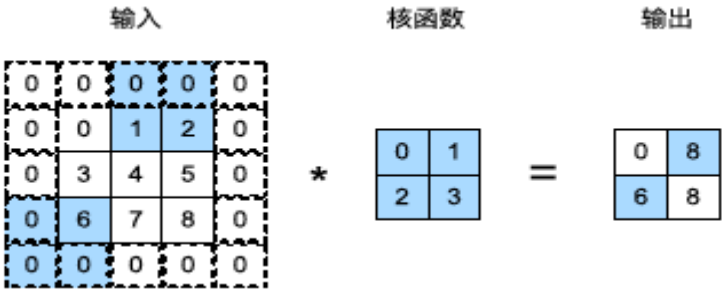


图 1.3: 垂直步幅为 3，水平步幅为 2 的二维互相关运算。

```
conv2d = nn.Conv2d(1, 1, kernel_size=3, padding=1, stride=2)
comp_conv2d(conv2d, X).shape
```

```
torch.Size([4, 4])
```

接下来，看一个稍微复杂的例子。

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

```
torch.Size([2, 2])
```

1.5 多输入多输出通道

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

当我们添加通道时，我们的输入和隐藏的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。本节将更深入地研究具有多输入和多输出通道的卷积核。

1.5.1 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。如果卷积核的窗口形状是 $kh\times kw$，那么当 $c_i=1$ 时，我们可以把卷积核看作形状为 $kh\times kw$ 的二维张量。

然而，当 $c_i>1$ 时，我们卷积核的每个输入通道将包含形状为 $kh\times kw$ 的张量。将这些张量 ci 连结在一起可以得到形状为 $c_i\times kh\times kw$ 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。

在图1.4中，我们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：$(1\times1+2\times2+4\times3+5\times4)+(0\times0+1\times1+3\times2+4\times3)=56$。

为了加深理解，我们实现一下多输入通道互相关运算。简而言之，我们所做的就是对每个通道执行互相关操作，然后将结果相加。

```
import torch
from d2l import torch as d2l
```

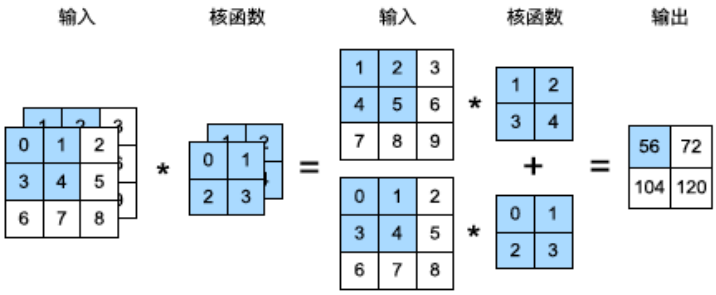


图 1.4: 两个输入通道的互相关计算。

```
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
    return sum(d2l.corr2d(x, k) for x, k in zip(X, K))
```

我们可以构造与图1.4中的值相对应的输入张量 X 和核张量 K，以验证互相关运算的输出。

```
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)
```

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

1.5.2 多输出通道

到目前为止，不论有多少输入通道，我们还只有一个输出通道。然而，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，我们常会增加输出通道的维数，通过减少空间分辨率以获得更大的通道深度。直观地说，我们可以将每个通道看作对不同特征的响应。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，我们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。

```
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。

```
K = torch.stack((K, K + 1, K + 2), 0)
K.shape
```

```
torch.Size([3, 2, 2, 2])
```

下面，我们对输入张量 X 与卷积核张量 K 执行互相关运算。现在的输出包含 3 个通道，第一个通道的结果与先前输入张量 X 和多输入单输出通道的结果一致。

```
corr2d_multi_in_out(X, K)
```

```
tensor([[[ 56., 72.],
[104., 120.]],
[[ 76., 100.],
[148., 172.]],
[[ 96., 128.],
[192., 224.]]])
```

1.5.3 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。下面，让我们详细地解读一下它的实际作用。

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。其实 1×1 卷积的唯一计算发生在通道上。

图1.5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。我们可以将 1×1 卷积层看作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。

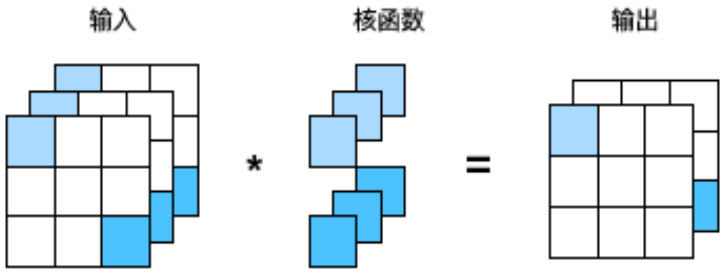


图 1.5: 互相关计算使用了具有 3 个输入通道和 2 个输出通道的 1×1 卷积核。其中，输入和输出具有相同的高度和宽度。

下面，我们使用全连接层实现 1×1 卷积。请注意，我们需要对输入和输出的数据形状进行调整。

```
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。让我们用一些样本数据来验证这一点。

```
X = torch.normal(0, 1, (3, 3, 3))
K = torch.normal(0, 1, (2, 3, 1, 1))
```

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

1.6 汇聚层

通常当我们处理图像时，我们希望逐渐降低隐藏表示的空间分辨率、聚集信息，这样随着我们在神经网络中层叠的上升，每个神经元对其敏感的感受野（输入）就越大。

而我们的机器学习任务通常会跟全局图像的问题有关（例如，“图像是否包含一只猫呢?”），所以我们最后一层的神经元应该对整个输入的全局敏感。通过逐渐聚合信息，生成越来越粗糙的映射，最终实现学习全局表示的目标，同时将卷积图层的所有优势保留在中间层。

此外，当检测较底层的特征时，我们通常希望这些特征保持某种程度上的平移不变性。例如，如果我们拍摄黑白之间轮廓清晰的图像 X，并将整个图像向右移动一个像素，即 $Z[i,j] = X[i,j+1]$，则新图像 Z 的输出可能大不相同。而在现实中，随着拍摄F度的移动，任何物体几乎不可能发生在同一像素上。即使用三脚架拍摄一个静止的物体，由于快门的移动而引起的相机振动，可能会使所有物体左右移动一个像素（除了高端相机配备了特殊功能来解决这个问题）。

本节将介绍汇聚（pooling）层，它具有双重目的：降低卷积层对位置的敏感性，同时降低对空间降采样表示的敏感性。

1.6.1 最大汇聚层和平均汇聚层

与卷积层类似，汇聚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（有时称为汇聚窗口）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，汇聚层不包含参数。相反，池运算是确定性的，我们通常计算汇聚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汇聚层（maximum pooling）和平均汇聚层（average pooling）。

在这两种情况下，与互相关运算符一样，汇聚窗口从输入张量的左上F开始，从左往右、从上往下的在输入张量内滑动。在汇聚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汇聚层还是平均汇聚层。

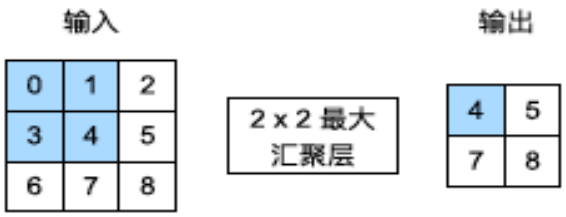


图 1.6: 汇聚窗口形状为 2×2 的最大汇聚层。着色部分是第一个输出元素，以及用于计算这个输出的输入元素：$max(0;1;3;4) = 4$。

图1.6中的输出张量的高度为 2，宽度为 2。这四个元素为每个汇聚窗口中的最大值：

$$max(0;1;3;4) = 4; \tag{1.8}$$

$$max(1;2;4;5) = 5; \tag{1.9}$$

$$max(3;4;6;7) = 7; \tag{1.10}$$

$$max(4;5;7;8) = 8 : \tag{1.11}$$

汇聚窗口形状为 $p \times q$ 的汇聚层称为 $p \times q$ 汇聚层，汇聚操作称为 $p \times q$ 汇聚。

回到本节开头提到的对象边缘检测示例，现在我们将使用卷积层的输出作为 2×2 最大汇聚的输入。设置卷积层输入为 X，汇聚层输出为 Y。无论 $X[i, j]$ 和 $X[i, j+1]$ 的值相同与否，或 $X[i, j+1]$ 和 $X[i, j+2]$ 的值相同与否，汇聚层始终输出 $Y[i, j] = 1$。也就是说，使用 2×2 最大汇聚层，即使在高度或宽度上移动一个元素，卷积层仍然可以识别到模式。

在下面的代码中的 pool2d 函数，我们实现汇聚层的前向传播。

```
import torch
from torch import nn
from d2l import torch as d2l
```

```
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
```

我们可以构建图1.6中的输入张量 X，验证二维最大汇聚层的输出。

```
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))
```

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

此外，我们还可以验证平均汇聚层。

```
pool2d(X, (2, 2), 'avg')
```

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

1.6.2 填充和步幅

与卷积层一样，汇聚层也可以改变输出形状。和以前一样，我们可以通过填充和步幅以获得所需的输出形状。下面，我们用深度学习框架中内置的二维最大汇聚层，来演示汇聚层中填充和步幅的使用。我们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。

```
X = torch.arange(16, dtype=torch.float32).reshape((1, 1, 4, 4))
X
```

```
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汇聚窗口的大小相同。因此，如果我们使用形状为 (3, 3) 的汇聚窗口，那么默认情况下，我们得到的步幅形状为 (3, 3)。

```
pool2d = nn.MaxPool2d(3)
pool2d(X)
```

```
tensor([[[[10.]]]])
```

填充和步幅可以手动设定。

```
pool2d = nn.MaxPool2d(3, padding=1, stride=2)
pool2d(X)
```

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

当然，我们可以设定一个任意大小的矩形汇聚窗口，并分别设定填充和步幅的高度和宽度。

```
pool2d = nn.MaxPool2d((2, 3), stride=(2, 3), padding=(0, 1))
pool2d(X)
```

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

1.6.3 多个通道

在处理多通道输入数据时，汇聚层在每个输入通道上单独运算，而不是像卷积层一样在通道上对输入进行汇总。这意味着汇聚层的输出通道数与输入通道数相同。下面，我们将在通道维度上连结张量 X 和 X + 1，以构建具有 2 个通道的输入。

```
X = torch.cat((X, X + 1), 1)
X
```

```
tensor([[[[ 0., 1., 2., 3.],
[ 4., 5., 6., 7.],
[ 8., 9., 10., 11.],
[12., 13., 14., 15.]],
[[ 1., 2., 3., 4.],
[ 5., 6., 7., 8.],
[ 9., 10., 11., 12.],
[13., 14., 15., 16.]]]])
```

如下所示，汇聚后输出通道的数量仍然是 2。

```
pool2d = nn.MaxPool2d(3, padding=1, stride=2)
pool2d(X)
```

```
tensor([[[[ 5., 7.],
[13., 15.]],
[[ 6., 8.],
[14., 16.]]]])
```

1.7 卷积神经网络（LeNet）

通过之前几节，我们学习了构建一个完整卷积神经网络的所需组件。回想一下，之前我们将 softmax 回归模型和多层感知机模型应用于 Fashion-MNIST 数据集中的服装图片。为了能够应用 softmax 回归和多层感知机，我们首先将每个大小为 28×28 的图像展平为一个 784 维的固定🄴度的一维向量，然后用全连接层对其进行处理。而现在，我们已经掌握了卷积层的处理方法，我们可以在图像中保留空间结构。同时，用卷积层代替全连接层的另一个好处是：模型更简洁、所需的参数更少。

本节将介绍 LeNet，它是最早发布的卷积神经网络之一，因其在计算机视觉任务中的高效性能而受到广泛关注。这个模型是由 AT&T 贝尔实验室的研究员 Yann LeCun 在 1989 年提出的（并以其命名），目的是识别图像中的手写数字。当时，Yann LeCun 发表了第一篇通过反向传播成功训练卷积神经网络的研究，这项工作代表了十多年来神经网络研究开发的成果。

当时，LeNet 取得了与支持向量机（support vector machines）性能相媲美的成果，成为监督学习的主流方法。LeNet 被广泛用于自动取款机（ATM）机中，帮助识别处理支票的数字。时至今日，一些自动取款机仍在运行 Yann LeCun 和他的同事 Leon Bottou 在上世纪 90 年代写的代码呢！

1.7.1 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。

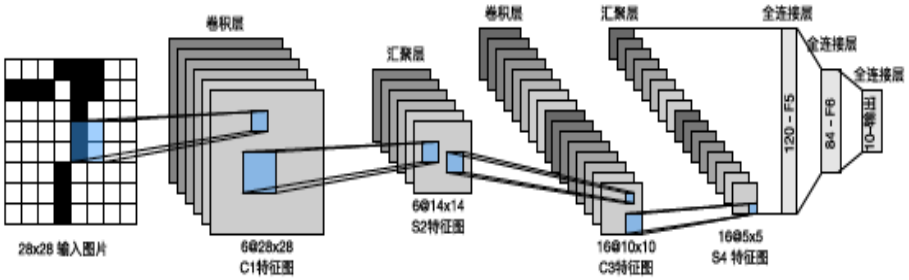


图 1.7: LeNet 中的数据流。输入是手写数字，输出为 10 种可能结果的概率。

每个卷积块中的基本单元是一个卷积层、一个 sigmoid 激活函数和平均汇聚层。请注意，虽然 ReLU 和最大汇聚层更有效，但它们在 20 世纪 90 年代还没有出现。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（步幅 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，我们必须在小批量中展平每个样本。换言之，我们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。

通过下面的 LeNet 代码，可以看出用深度学习框架实现此类模型非常简单。我们只需要实例化一个 Sequential 块并将需要的层连接在一起。

```
import torch
from torch import nn
from d2l import torch as d2l
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))
```

我们对原始模型做了一点小改动，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，我们将一个大小为 28×28 的单通道（黑白）图像通过 LeNet。通过在每一层打印输出的形状，我们可以检查模型，以确保其操作与我们期望的图1.8一致。

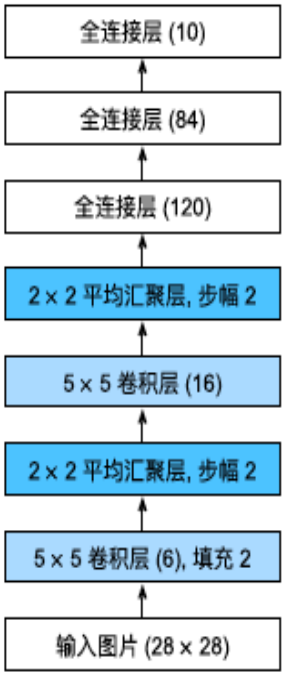


图 1.8: LeNet 的简化版。

```
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])
```

请注意，在整个卷积块中，与上一层相比，每一层特征的高度和宽度都减小了。第一个卷积层使用 2 个像素的填充，来补偿 5×5 卷积核导致的特征减少。相反，第二个卷积层没有填充，因此高度和宽度都减少了 4 个像素。随着层叠的上升，通道的数量从输入时的 1 个，增加到第一个卷积层之后的 6 个，再到第二个卷积层之后的 16 个。同时，每个汇聚层的高度和宽度都减半。最后，每个全连接层减少维数，最终输出一个维数与结果分类数相匹配的输出。

1.7.2 模型训练

现在我们已经实现了 LeNet，让我们看看 LeNet 在 Fashion-MNIST 数据集上的表现。

```
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。通过使用 GPU，可以用它加快训练。

为了进行评估，我们需要对之前中描述的 evaluate_accuracy 函数进行轻微的修改。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，我们需要将其复制到显存中。

```
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"""
    if isinstance(net, nn.Module):
        net.eval() # 设置为评估模式
        if not device:
            device = next(iter(net.parameters())).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]
```

与全连接层一样，我们使用交叉熵损失函数和小批量随机梯度下降训练模型。

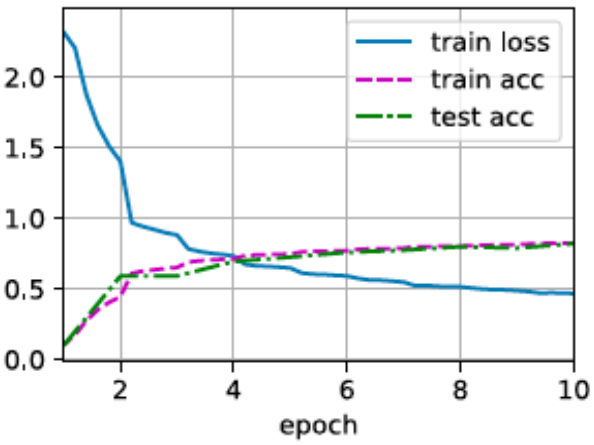
```
#@save
def train_ch6(net, train_iter, test_iter, num_epochs, lr, device):
    """用GPU训练模型(在第六章定义)"""
    def init_weights(m):
        if type(m) == nn.Linear or type(m) == nn.Conv2d:
            nn.init.xavier_uniform_(m.weight)
    net.apply(init_weights)
    print('training on', device)
    net.to(device)
    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)
        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，我们训练和评估 LeNet-5 模型。

```
lr, num_epochs = 0.9, 10
train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.467, train acc 0.825, test acc 0.821
88556.9 examples/sec on cuda:0
```


train loss
train acc
test acc
2.0
1.5
1.0
0.5
0.0
2
4
6
8
10
epoch