第八章 卷积神经网络

到现在为止，我们建立的神经网络都是全连接的。全连接的意思是相邻两层之间的结点都相连，如图 8-1 所示。

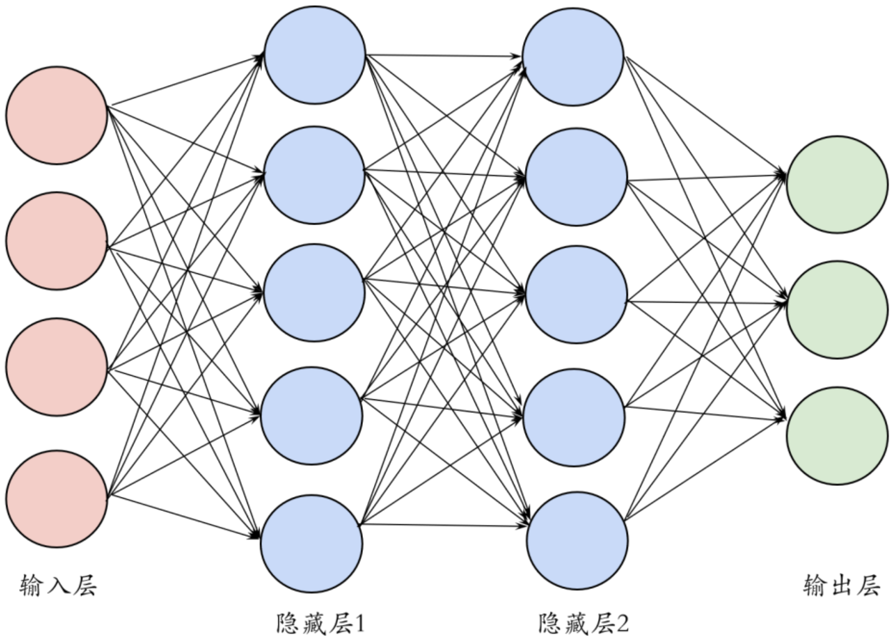

图 8-1 全连接神经网络

这类神经网络可以处理很多一般问题。但是，当数据自变量具有一定的结构时，全连接神经网络通常表现一般。例如，在数字图像处理中，Lenna 是一幅使用广泛的标准图片，如图 8-2。图 8-2 中黑色矩形框的像素点组成了 Lenna 的右眼。这些像素点具备特定的结构，例如，中间黑色的瞳孔，四周眼白，像素点只有这样排列才能组成眼睛。换句话说，这些像素点具有内在关联，是相关的。在实际中，全连接神经网络输入层的结点顺序可以是任意的，不能有效考虑自变量的内在结构。因此，全连接神经网络不能很好地处理图片数据，以及类似的自变量具有一定结构的数据。

图 8-2 数字图像处理领域常用的标准图片

卷积神经网络（CNN）可以简单有效地考虑自变量的内在结构，是深度学习在图片处理等计算机视觉领域实现优异表现的基石。本章将首先介绍卷积神经网络两个最重要的组成部分：卷积（convolution）和池化（pooling）。然后把卷积和池化应用到 mnist 数据中，建立卷积神经网络，使得手写数字图片分类达到更高精度。

8.1 卷积层

8.1.1 卷积运算

卷积层（Convolutional Layer）是卷积神经网络最重要的组成部分。卷积层需要用到卷积运算。在数学上，卷积是通过两个函数f和g生成第三个函数的一种数学算子，通常会在数学分析或者泛函分析中学到。看到不仅有数学分析，还有泛函分析，是不是有点让人摸不着头脑和令人生畏?! 幸运的是，学习卷积神经网络可以完全不需要了解数学分析关于卷积的定义和性质。

在卷积神经网络中，卷积运算只是按照一定规则简单相乘，再求和的运算，卷积运算记为$\circledast$。下面例子中，输入数据为一个3×3的数组，核（kernel）是一个2×2的数组。输入数据和核通过一定的运算规则（运算规则称为卷积）得到输出数据。总的运算规则是：把核从左到右，从上到下与输入数据比对（也就是在输入数据中找与核维度相同的部分），然后把输入数据中的数字和核的对应数字相乘，再求和。请看下面详细说明。

- 首先，把核对应到输入数据左上角的 4 个数字。然后把输入数据左上角的 4 个数字与核的对应位置 4 个数字相乘，再求和，得到输出数据的第 1 个结果。如图 8-3，$0 \times (-1) + 4 \times 0.5 + (-1) \times 0 + 0 \times 1 = 2$。

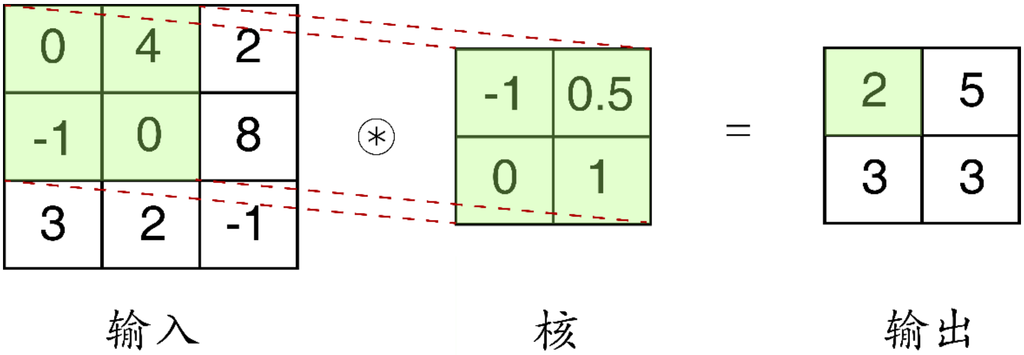

图 8-3 卷积运算（第 1 步）

- 然后把核往右边移动一格，对应到输入数据的右上角的 4 个数字。然后把输入数据右上角的 4 个数字与核的对应位置 4 个数字相乘，再求和，得到输出数据的第 2 个结果。如图 8-4，$4\times(-1)+2\times0.5+0\times0+8\times1=5$。

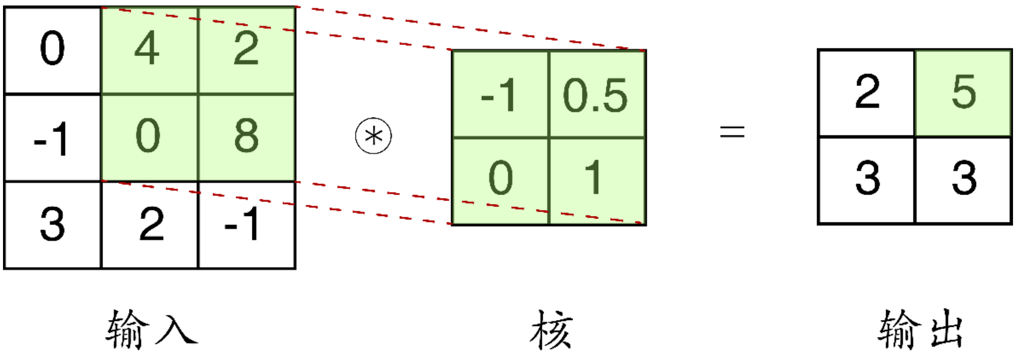

图 8-4 卷积运算（第 2 步）

- 接着把核向下移动一格，且移到最左边，对应到输入数据的左下角的 4 个数字。然后把输入数据左下角的 4 个数字与核的对应位置 4 个数字相乘，再求和，得到输出数据的第 3 个结果。如图 8-5，$(-1)\times(-1)+0\times0.5+3\times0+2\times1=3$。

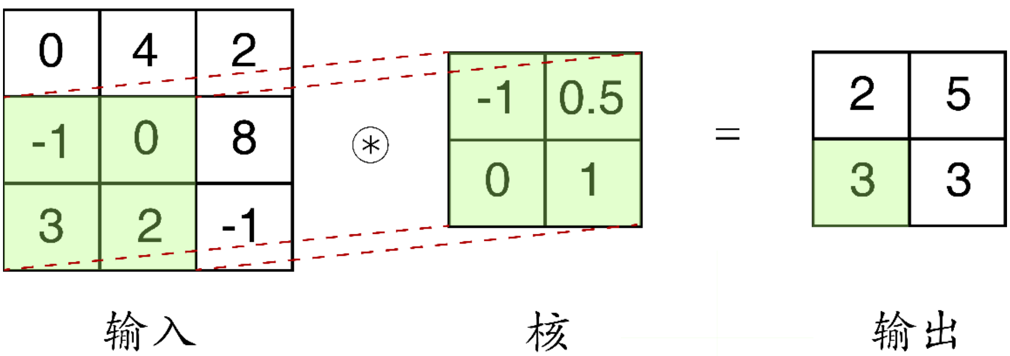

图 8-5 卷积运算（第 3 步）

- 最后把核往右边移动一格，对应到输入数据右下角的 4 个数字。然后把输入数据右下角的 4 个数字与核的对应位置 4 个数字相乘，再求和，得到输出数据的第 4 个结果。如图 8-6，$0 \times (-1) + 8 \times 0.5 + 2 \times 0 + (-1) \times 1 = 3$。

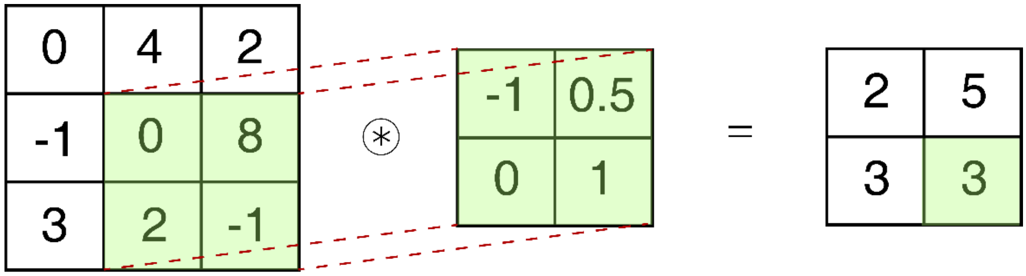

图 8-6 卷积运算（第 4 步）

8.1.2 卷积层运算

对于图片数据，我们之前会把二维的像素点（每个像素点就是一个自变量）转换成一维的向量。这样的方式破坏了像素点之间的关系。在卷积神经网络中，我们将保留二维像素点的形式。例如，在 mnist 数据中，图片表示为28×28的灰度值。

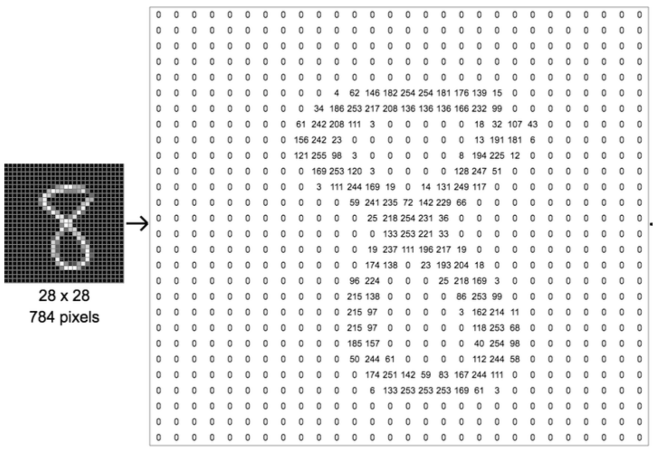

图 8-7 mnist 数据中的一幅图（数字 8）

卷积神经网络可以表示为图 8-8 的形式。卷积神经网络的每一层都可以看做是一个 3 维数组。3 个维度分别为宽度（width）、高度（height）和深度（depth）。例如，图 8-7 数字 8 的宽度为 28，高度为 28，深度为 1。在本章，

我们考虑较为简单的情况，输入层的深度为 1（即输入数据表示为一个矩阵或者二维数组），且神经网络中只有 1 个卷积层。

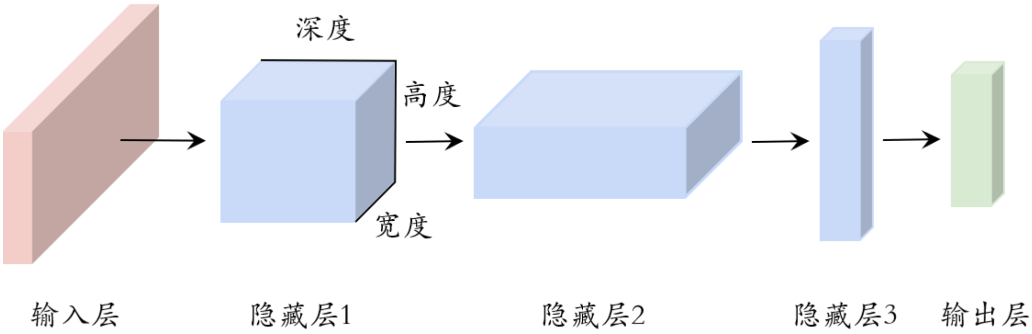

图 8-8 卷积神经网络

图 8-9 中，输入层的维度为28×28，核的维度为5×5，卷积运算时核每次往右边移动一格，那么输出结果行维度为 24（24=28-5+1）；每一行结束之后，核每次往下移动一格，那么输出结果列维度为 24（24=28-5+1）。因此，对于图 8-9 的核，输出为一个24×24的平面。对于一般的情况，记输入层维度为$W \times H$，核的维度为$K \times K$。如果核每次移动一格，卷积运算输出结果维度为$W_{\text{out}} = W - K + 1$和$H_{\text{out}} = H - K + 1$。在实际中，核也可以每次移动超过一格。记每次移动$S$格，$S$称为步幅（stride）；这时卷积运算输出结果的维度为$W_{\text{out}} = (W - K)/S + 1$和$H_{\text{out}} = (H - K)/S + 1$。

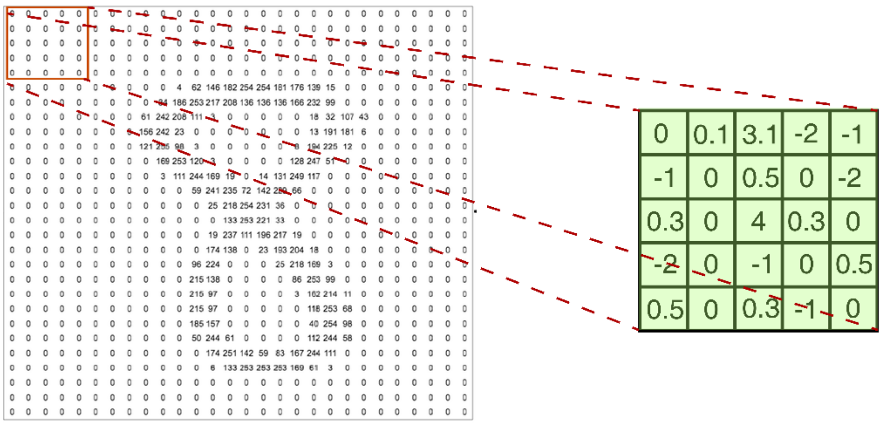

图 8-9 卷积层运算，核的维度为5×5，步幅为 1

在实际中，卷积层使用多个核分别与输入层做卷积运算，每个核与输入层卷积运算的结果为一个平面（每个平面的维度都是宽度×高度），把多个平面放在一起堆叠成一个 3 维数组，平面的数量即为深度。如图 8-10 中，我们有 3 个核。图 8-10 中，假设输出层的维度为28×28，核的维度为5×5，有 3 个核，因此隐

藏层维度为$24 \times 24 \times 3$（宽度×高度×深度），即把 3 个核与输入层卷积运算的结果（24×24）按从左到右排列。

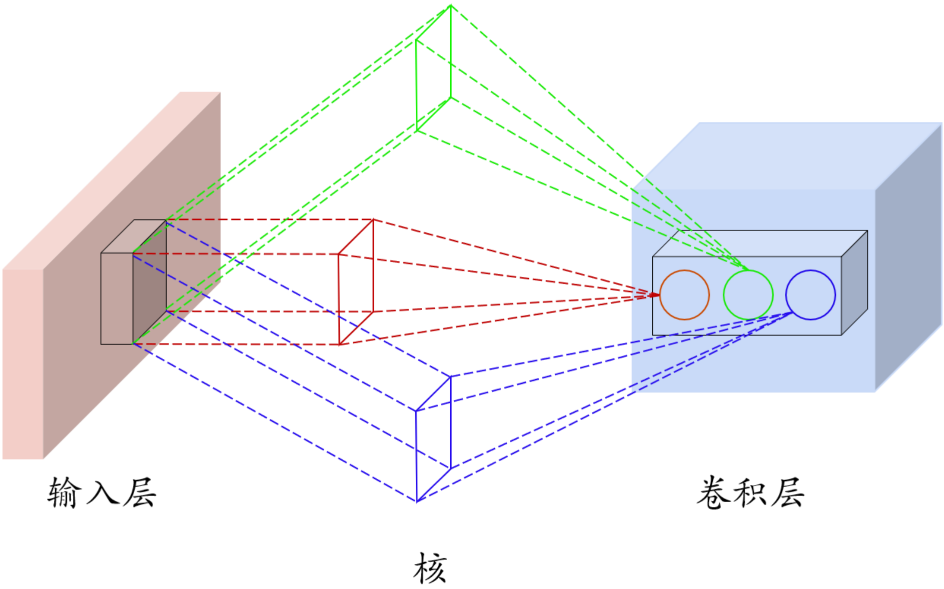

图 8-10 卷积层运算，3 个核

8.1.3 卷积运算的直觉理解

请看图 8-11。在日常生活中，人类大脑对眼睛接收信号的反应速度非常快，但是现在请尽可能地放慢看的速度，认真思考辨认图片的过程。

- 首先我们会看到一些线条、棱角、颜色等基本元素；
- 然后组合这些基本元素，可以识别出眼睛、耳朵、鼻子、嘴巴、尾巴等图形；
- 最后根据眼睛、鼻子等图形分辨出图片是一只小狗。

图 8-11 一只小狗

通常的卷积神经网络通常有多个卷积层，每个卷积层又由多个卷积核产生，每一个卷积核都有其特定功能。卷积运算可以类比为人类的图片识别过程。例如，

- 第一层卷积层的卷积核可能识别图片的一些基本特征，如线条、颜色等；
- 第二层卷积层的卷积核可能综合第一层识别的线条颜色等特征，判断更大范围的特征，如，鼻尖、眼角等；
- 第三层卷积层的卷积核可能又根据第二层得到的特征，识别出鼻子、眼睛等；
- …
- 最后判断图片的类别。

例如，图 8-12 有两个核，虚线框核和实线框核。虚线框核斜向上位置的值都等于 1，左上角和右下角的值都等于 0；实线框核斜向下位置的值都等于 1，左下角和右上角都等于 0。

- 当虚线框核在图 8-12 左图 a 处时，隐藏层中虚线框核对应结点将得到一个较大的值；而虚线框核在其他位置时，得到的值都会比较小，例如左图位置 b。也就是说，虚线框核得到的较大结果说明输入数据对应位置可能具斜向上线段特征。
- 当实线框核在图 8-12 左图 d 处时，隐藏层中实线框核对应结点将得到一个较大值；而实线框核在其他位置时，得到的值都会比较小，例如左图位置 c。也就是说，实线框核得到的较大结果说明输入数据对应位置可能具斜向下线段特征。

实际应用中，卷积神经网络使用多个核作用于同一个输入层。通过训练，不同的核可以识别出图片的不同特征。如果使用多个卷积层，下一步的卷积层将有

可能综合上一层识别的特征，判断范围更大的特征。总的来说，可以认为卷积神经网络的预测或者判断过程和人类大脑的思考过程类似。

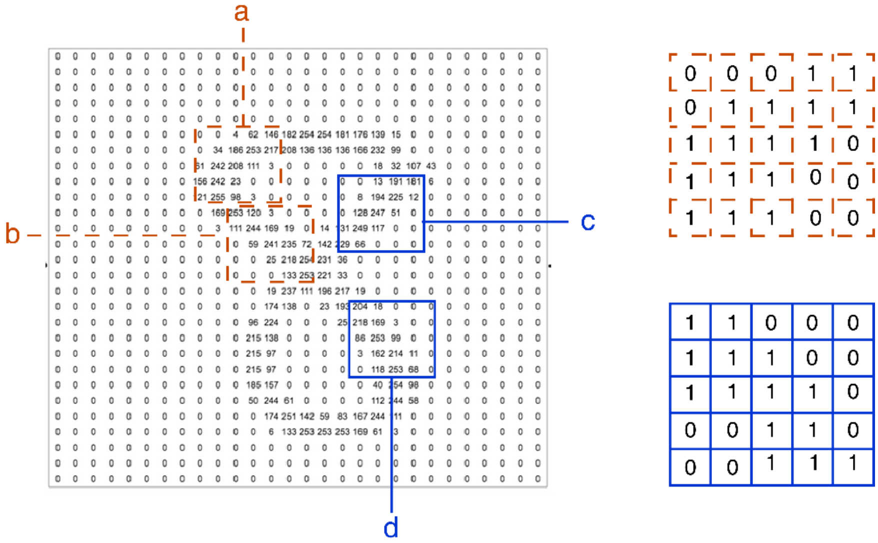

图 8-12 识别出斜向上线段和斜向下线段的核

在机器学习领域，有很多科学家从事卷积神经网络图片识别的解释性研究。图 8-13 展示了 Krizhevsky et al 文章中建立的卷积神经网络第一层卷积层的核。该卷积层有 96 个核，每个核$11 \times 11 \times 3$，输入层的维度$224 \times 224 \times 3$（这是输入层是 3 维，核也是 3 维的情况，因此可以体现出颜色。我们将会在第九章介绍 3 维卷积运算）。从图 8-13 可以看到，不同的核表现出了不同的功能，有些核可以识别出一些角度各不相同的线段（特别是前 3 行），有些可以识别一些花纹，还有一些可以识别颜色特征（特别是后 3 行）。

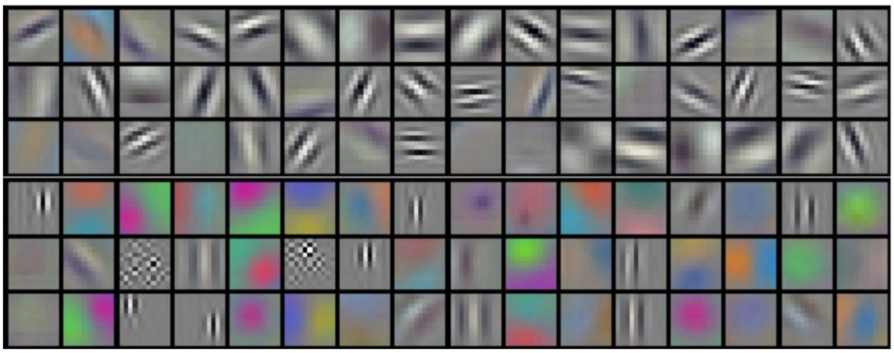

图 8-13 Krizhevsky et al 文章中第一层卷积层图示。该卷积层有 96 个核。

8.1.4 填充

我们已经学习了卷积层的基本运算。不过，8.1.2 节介绍的卷积运算还有两点不足之处。

1. 相对于输入层，卷积层的宽度和高度都变小了，而且变小的速度很快。如图 8-14，输入层维度为4×4，核维度为3×3，每次只移动一格，卷积运算结果的维度为2×2。在这个例子中，输出数据维度只有输入数据维度的一半。卷积运算结果维度下降有两个缺点。
   - 建模过程中需要时刻关注卷积运算结果的维度才能确定下一层的核或者权重矩阵的维度。如果卷积运算结果的维度保持不变，那么确定下一层的核或者权重矩阵的维度将会更简单。
   - 当卷积运算使维度下降很快时，少数几个卷积层便能使维度下降得很小。这样很难建立层数很多的深度学习模型。
2. 卷积运算中，因为输入层边缘只出现在少数的运算中，边缘部分的信息没有很好利用。如图 8-14，输入层左下角的3只出现在一次运算中，容易使该信息在运算中消失。如果输入层的边缘体现了图片特征，这种情况便会降低预测准确率。

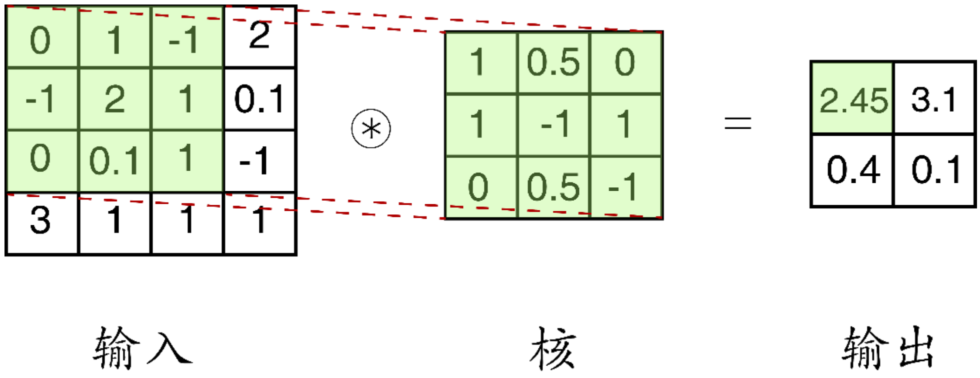

图 8-14 卷积运算（没有填充）

幸运的是，填充（padding）可以有效解决上述两个问题。填充的想法很简单，只需要在输入层的四周填充数字，然后再做卷积运算，便可以使输出结果的维度保持与输入数据维度一致，且输入层边缘可以多次在计算中被使用。因为0在卷积运算中不会增加噪声，通常情况下，我们在输入数据四周填充0。因此，填充也称为零填充（zero-padding）。

下面例子中，输入数据的维度为4×4，在数据的左右两边都填充一列 0，上下两边都填充一行 0；输入数据维度变为6×6。核的维度为3×3，步幅为$S = 1$。最终得到的输出数据维度为4×4，与输入数据一样。而且输入数据中左下角的 3 出现在 4 次运算中，该像素点的信息可以更好地保留下来，如果该像素点对判断该图有重要作用，那么填充便可能提高模型预测准确度。

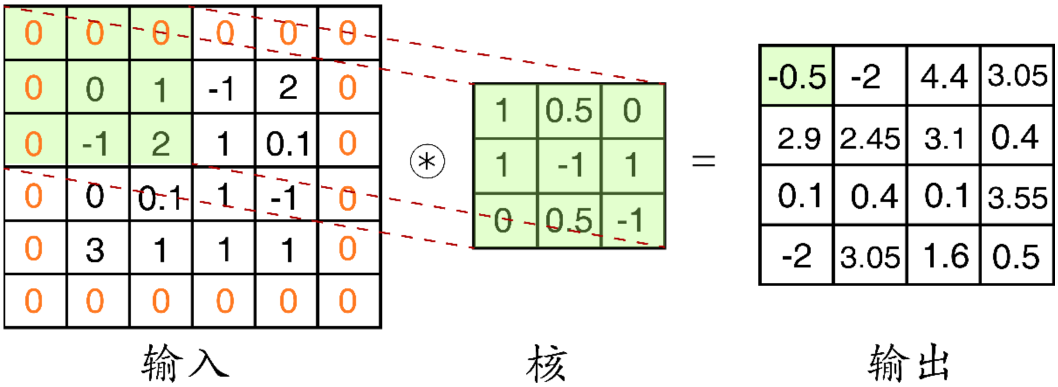

图 8-15 卷积运算(有填充)

零填充时，通常在输入数据每一边都填充相同的行数和列数，记为P。例如，上面例子中，$P=1$。这时卷积运算输出维度为$W_{\text{out}}=(W-K+2P)/S+1$和$H_{\text{out}}=(H-K+2P)/S+1$。例如，在图 8-15 中，$W_{\text{out}}=(4-3+2\times1)/1+1=4$，$H_{\text{out}}=(4-3+2\times1)/1+1=4$，输入层与输出层的维度保持一致。反过来，当我们知道W和H，确定了K和S时，令$W_{\text{out}}=W$，$H_{\text{out}}=H$，便可以通过方程$W_{\text{out}}=(W-K+2P)/S+1$或者$H_{\text{out}}=(H-K+2P)/S+1$得到需要零填充的行数和列数，即$P=((W-1)S-W+K)/2$。

8.1.5 卷积层求导

本节中，我们以一个简单例子说明卷积层求导过程。假设进行如下卷积运算，输入层维度为3×3，核维度为2×2，输出维度为2×2。损失函数记为L。

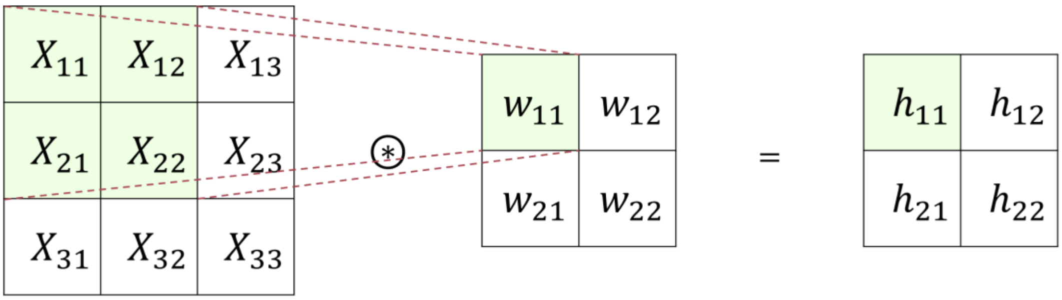

图 8-16 卷积层求导

- 使用正向传播算法计算预测值：在卷积层中，通过卷积运算计算得到h_{11}，h_{12}，h_{21}，h_{22}，再通过一系列运算计算预测值。特别的，卷积运算的结果如下：
  - $h_{11}=X_{11}w_{11}+X_{12}w_{12}+X_{21}w_{21}+X_{22}w_{22}$
  - $h_{12}=X_{12}w_{11}+X_{13}w_{12}+X_{22}w_{21}+X_{23}w_{22}$
  - $h_{21}=X_{21}w_{11}+X_{22}w_{12}+X_{31}w_{21}+X_{32}w_{22}$

- $h_{22} = X_{22}w_{11} + X_{23}w_{12} + X_{32}w_{21} + X_{33}w_{22}$

- 然后计算损失函数L，通过反向传播算法得到L关于h_{11}， h_{12}， h_{21}，h_{22}的偏导数$\frac{\partial L}{\partial h_{11}}$，$\frac{\partial L}{\partial h_{12}}$，$\frac{\partial L}{\partial h_{21}}$，$\frac{\partial L}{\partial h_{22}}$。

- 为了更新核元素w_{11}，w_{12}，w_{21}，w_{22}，需要计算损失函数L关于w_{11}，w_{12}，w_{21}，w_{22}的偏导数，$\frac{\partial L}{\partial w_{11}}$，$\frac{\partial L}{\partial w_{12}}$，$\frac{\partial L}{\partial w_{21}}$，$\frac{\partial L}{\partial w_{22}}$。根据链式法则，
  - $\frac{\partial L}{\partial w_{11}} = \frac{\partial L}{\partial h_{11}}\frac{\partial h_{11}}{\partial w_{11}} + \frac{\partial L}{\partial h_{12}}\frac{\partial h_{12}}{\partial w_{11}} + \frac{\partial L}{\partial h_{21}}\frac{\partial h_{21}}{\partial w_{11}} + \frac{\partial L}{\partial h_{22}}\frac{\partial h_{22}}{\partial w_{11}}$
  $= \frac{\partial L}{\partial h_{11}}X_{11} + \frac{\partial L}{\partial h_{12}}X_{12} + \frac{\partial L}{\partial h_{21}}X_{21} + \frac{\partial L}{\partial h_{22}}X_{22}$
  - $\frac{\partial L}{\partial w_{12}} = \frac{\partial L}{\partial h_{11}}\frac{\partial h_{11}}{\partial w_{12}} + \frac{\partial L}{\partial h_{12}}\frac{\partial h_{12}}{\partial w_{12}} + \frac{\partial L}{\partial h_{21}}\frac{\partial h_{21}}{\partial w_{12}} + \frac{\partial L}{\partial h_{22}}\frac{\partial h_{22}}{\partial w_{12}}$
  $= \frac{\partial L}{\partial h_{11}}X_{12} + \frac{\partial L}{\partial h_{12}}X_{13} + \frac{\partial L}{\partial h_{21}}X_{22} + \frac{\partial L}{\partial h_{22}}X_{23}$
  - $\frac{\partial L}{\partial w_{21}} = \frac{\partial L}{\partial h_{11}}\frac{\partial h_{11}}{\partial w_{21}} + \frac{\partial L}{\partial h_{12}}\frac{\partial h_{12}}{\partial w_{21}} + \frac{\partial L}{\partial h_{21}}\frac{\partial h_{21}}{\partial w_{21}} + \frac{\partial L}{\partial h_{22}}\frac{\partial h_{22}}{\partial w_{21}}$
  $= \frac{\partial L}{\partial h_{11}}X_{21} + \frac{\partial L}{\partial h_{12}}X_{22} + \frac{\partial L}{\partial h_{21}}X_{31} + \frac{\partial L}{\partial h_{22}}X_{32}$
  - $\frac{\partial L}{\partial w_{22}} = \frac{\partial L}{\partial h_{11}}\frac{\partial h_{11}}{\partial w_{22}} + \frac{\partial L}{\partial h_{12}}\frac{\partial h_{12}}{\partial w_{22}} + \frac{\partial L}{\partial h_{21}}\frac{\partial h_{21}}{\partial w_{22}} + \frac{\partial L}{\partial h_{22}}\frac{\partial h_{22}}{\partial w_{22}}$
  $= \frac{\partial L}{\partial h_{11}}X_{22} + \frac{\partial L}{\partial h_{12}}X_{23} + \frac{\partial L}{\partial h_{21}}X_{32} + \frac{\partial L}{\partial h_{22}}X_{33}$

- 如果神经网络模型中有多个卷积层，反向传播算法中还需要计算损失函数L关于X_{11}，X_{12}，X_{13}，X_{21}，X_{22}，X_{23}，X_{31}，X_{32}，X_{33}的偏导数，即$\frac{\partial L}{\partial X_{11}}$，$\frac{\partial L}{\partial X_{12}}$，$\frac{\partial L}{\partial X_{13}}$，$\frac{\partial L}{\partial X_{21}}$，$\frac{\partial L}{\partial X_{22}}$，$\frac{\partial L}{\partial X_{23}}$，$\frac{\partial L}{\partial X_{31}}$，$\frac{\partial L}{\partial X_{32}}$，$\frac{\partial L}{\partial X_{33}}$。根据链式法则，
  - $\frac{\partial L}{\partial X_{11}} = \frac{\partial L}{\partial h_{11}}\frac{\partial h_{11}}{\partial X_{11}} = \frac{\partial L}{\partial h_{11}}w_{11}$
  - $\frac{\partial L}{\partial X_{12}} = \frac{\partial L}{\partial h_{11}}\frac{\partial h_{11}}{\partial X_{12}} + \frac{\partial L}{\partial h_{12}}\frac{\partial h_{12}}{\partial X_{12}} = \frac{\partial L}{\partial h_{11}}w_{12} + \frac{\partial L}{\partial h_{12}}w_{11}$
  - $\frac{\partial L}{\partial X_{13}} = \frac{\partial L}{\partial h_{12}}\frac{\partial h_{12}}{\partial X_{13}} = \frac{\partial L}{\partial h_{12}}w_{12}$
  - $\frac{\partial L}{\partial X_{21}} = \frac{\partial L}{\partial h_{11}}\frac{\partial h_{11}}{\partial X_{21}} + \frac{\partial L}{\partial h_{21}}\frac{\partial h_{21}}{\partial X_{21}} = \frac{\partial L}{\partial h_{11}}w_{21} + \frac{\partial L}{\partial h_{21}}w_{11}$
  - $\frac{\partial L}{\partial X_{22}} = \frac{\partial L}{\partial h_{11}}\frac{\partial h_{11}}{\partial X_{22}} + \frac{\partial L}{\partial h_{12}}\frac{\partial h_{12}}{\partial X_{22}} + \frac{\partial L}{\partial h_{21}}\frac{\partial h_{21}}{\partial X_{22}} + \frac{\partial L}{\partial h_{22}}\frac{\partial h_{22}}{\partial X_{22}}$
  $= \frac{\partial L}{\partial h_{11}}w_{22} + \frac{\partial L}{\partial h_{12}}w_{21} + \frac{\partial L}{\partial h_{21}}w_{12} + \frac{\partial L}{\partial h_{22}}w_{11}$
  - $\frac{\partial L}{\partial X_{23}} = \frac{\partial L}{\partial h_{12}}\frac{\partial h_{12}}{\partial X_{23}} + \frac{\partial L}{\partial h_{22}}\frac{\partial h_{22}}{\partial X_{23}} = \frac{\partial L}{\partial h_{12}}w_{22} + \frac{\partial L}{\partial h_{22}}w_{12}$
  - $\frac{\partial L}{\partial X_{31}} = \frac{\partial L}{\partial h_{21}}\frac{\partial h_{21}}{\partial X_{31}} = \frac{\partial L}{\partial h_{21}}w_{21}$
  - $\frac{\partial L}{\partial X_{32}} = \frac{\partial L}{\partial h_{21}}\frac{\partial h_{21}}{\partial X_{32}} + \frac{\partial L}{\partial h_{22}}\frac{\partial h_{22}}{\partial X_{32}} = \frac{\partial L}{\partial h_{21}}w_{22} + \frac{\partial L}{\partial h_{22}}w_{21}$

■ $\frac{\partial L}{\partial X_{33}} = \frac{\partial L}{\partial h_{22}} \frac{\partial h_{22}}{\partial X_{33}} = \frac{\partial L}{\partial h_{22}} w_{22}$

8.1.6 Python 计算卷积层

在本小节中，我们尝试在 Python 中实现卷积层的运算。为了更好地理解卷积层原理，计算都基于基本的矩阵运算，因此只需要用到 Numpy 和 Python 的基本函数。

首先定义函数 get_image_section()，该函数的作用是选取输入数据中与核同样大小的子集。在这里，layer 有 3 个维度：观测点个数、数据宽度、数据高度。参数 row_start，row_end，col_start，col_end 分别表示选取子集的行开始、行结束、列开始、列结束。输出数据维度为 4 维：观测点个数、1、子集宽度、子集高度。第二个维度是为了稍后拼接数据。

```
def get_image_section(layer, row_start, row_end, col_start,
                                                  col_end):

    section = layer[:, row_start:row_end, col_start:col_end]
    return section.reshape(-1, 1, row_end-row_start, \
                              col_end-col_start)
```

然后定义函数 conv_reshape()，该函数的作用是填充输入数据，然后把数据做如图 8-17 的变换。我们使用函数 np.pad() 填充数据，((0,0),(1,1),(1,1))表示观测点个数方向不需要填充，宽度和高度方向都填充 1。接着有两个 for 循环，使用函数 get_image_section()提取输入数据中与核同样大小的子集。函数 np.concatenate()可以把输入数据子集拼接在一起，设置 axis=1 可以使得同一幅图片的输入数据子集放在一起，再把数据变换成列数为核宽度乘以核高度，如图 8-17。函数 conv_reshape()的输出如图 8-17 右图所示：每一行表示一个输入数据中核对应的部分，例如图片 1 阴影矩形部分变换成了右图的第一行；右图的实线竖线部分为一批输入数据中第一幅图片的核对应的所有子集，虚线竖线部分为一批输入数据中第二幅图片的核对应的所有子集。

```
def conv_reshape(image, kernel_rows, kernel_cols):
    # 零填充
    image = np.pad(image, ((0,0),(1,1),(1,1)), mode='constant')
    image_sections = []
    for row_start in range(image.shape[1] - kernel_rows + 1):
        for col_start in range(image.shape[2] - kernel_cols +
1):
            image_sections.append(get_image_section(image, \
                        row_start, row_start+kernel_rows, \
                        col_start, col_start+kernel_cols))
    # 拼接列表元素
    expanded_input = np.concatenate(image_sections, axis=1)
    es = expanded_input.shape
    # 数据变换成列数为核的宽度乘以核的高度
    layer = expanded_input.reshape(es[0]*es[1], -1)
```

```
    return layer
```

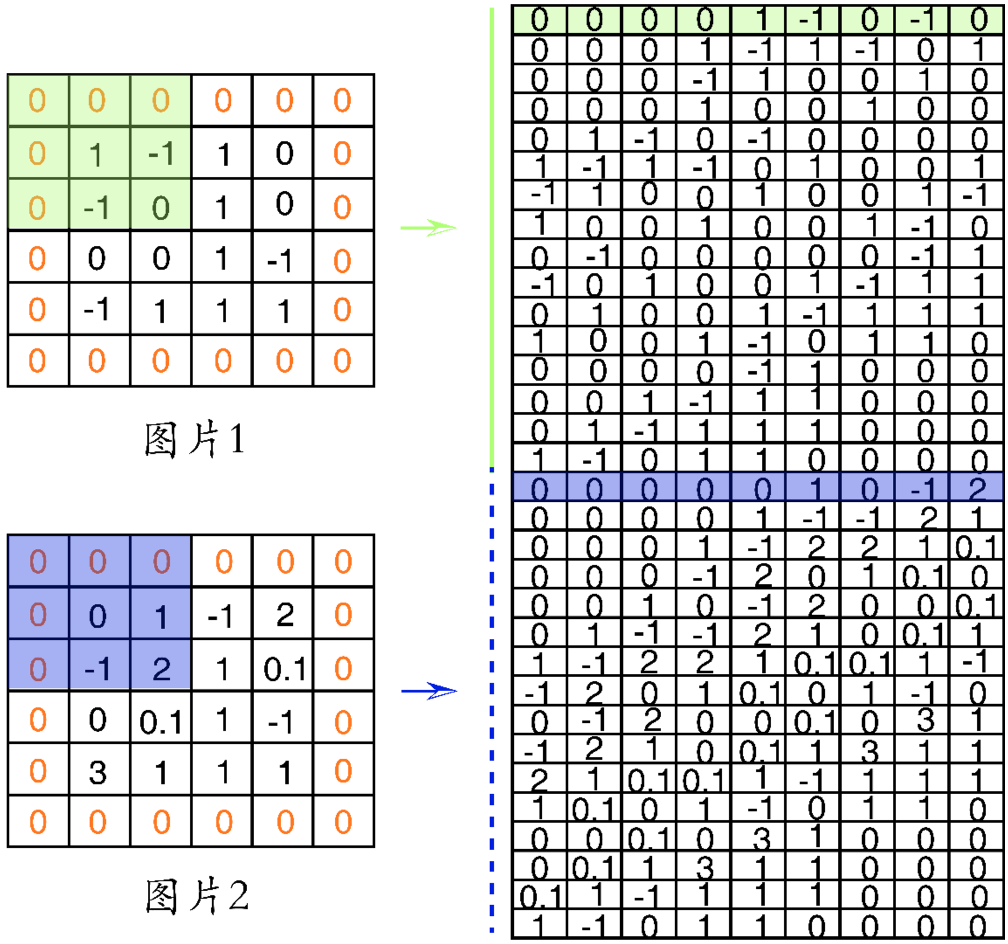

图 8-17 数据填充和变换。在这个例子中，一批数据有两副图片

最后定义函数 conv_2d()，让变换后的输入数据乘以核加上截距项。这里的 kernels 也是经过了变换的，kernels 本来的维度为核宽度、核高度、核的数量。如图 8-18 左图中，核宽度等于 3，核高度等于 3，核的数量等于 2。为了方便计算，我们把 kernels 的维度变换为（核的宽度 × 核的高度）× 核的数量，如图 8-18。在这里，bias 的维度为 1× 核的数量，也就是给每个核增加了一个截距项。本章所有关于核的示意图中，为了画图方便，都没有画截距项。在函数 conv_2d() 中，没有体现 kernels 的变换。该变换在代码中体现在初始化过程中。在这里，参数 kernels 是核变换后的结果，如图 8-18 右图。函数 conv_2d() 的参数 layer 是函数 conv_reshape() 的结果，如图 8-17 的右图。

```
def conv_2d(layer, kernels, bias):
    layer = np.dot(layer, kernels) + bias
    return relu(layer)
```

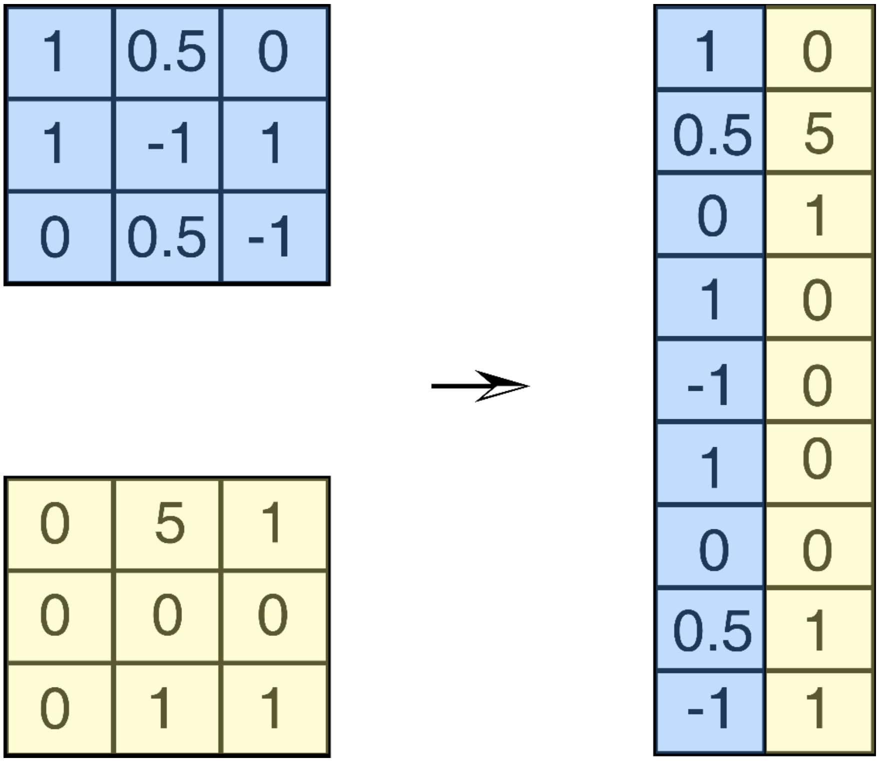

图 8-18 核的变换。两个核，每个核的维度为3×3，变换为右图的维度为9×2的矩阵

函数 `conv_2d()`的作用也可以用图 8-19 更直观的展示。在这个例子中，输入数据有两副图片，也有两个核，输入数据和核都经过了变换，图 8-19 左边两个矩阵分别表示 `layer` 和 `kernels`。函数 `conv_2d()`的结果如图 8-19 等号右边所示，函数 `conv_2d()`的结果为32×2；第一列表示两副图片与第一个核做卷积运算的结果，其中，第一列的前 16 行为第一幅图片与第一个核做卷积运算的结果，第一列的后 16 行为第二幅图片与第一个核做卷积运算的结果；第二列表示两副图片与第二个核做卷积运算的结果，其中，第二列的前 16 行为第一幅图片与第二个核做卷积运算的结果，第二列的后 16 行为第二幅图片与第二个核做卷积运算的结果。

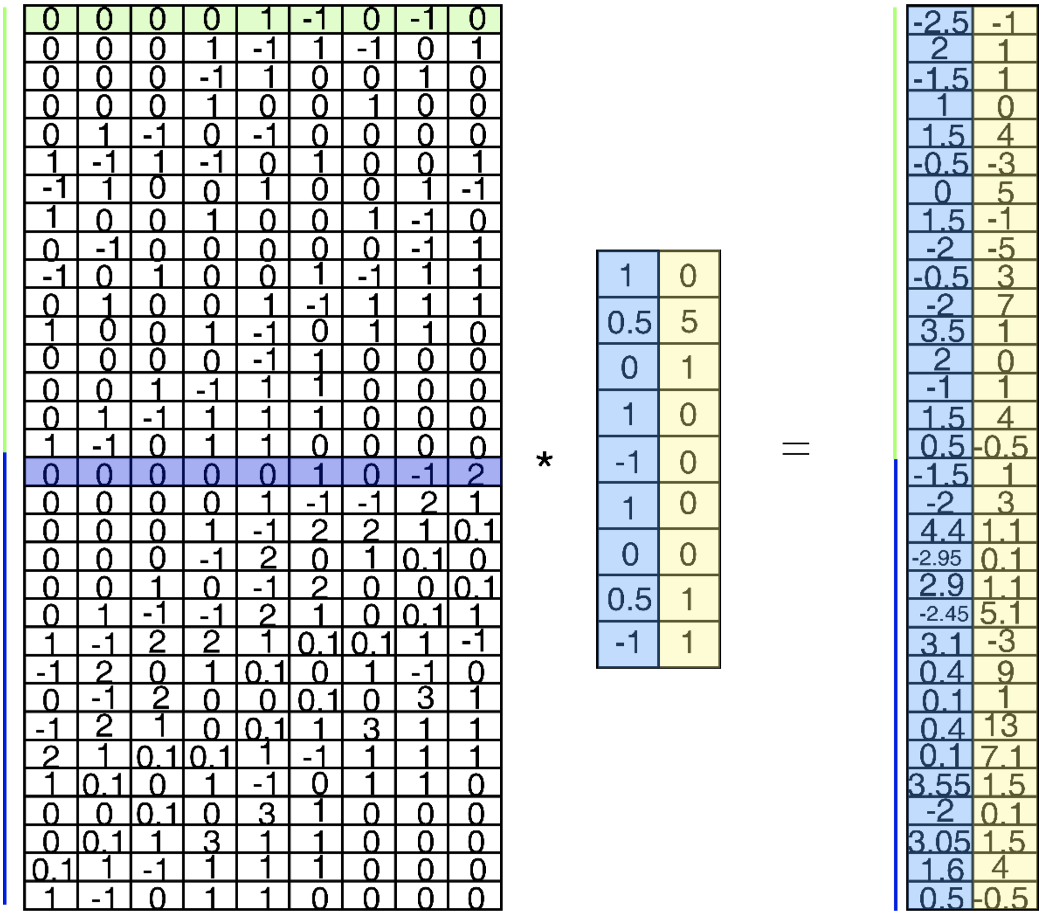

图 8-19 layer 和 kernels 变换后的矩阵运算

8.2 池化层

在卷积神经网络中，通常在一个或者多个卷积层后添加一层池化层（Pool Layer）。池化层运算与卷积层类似，但是核不需要具体数字，而是计算核对应输入数据的元素的最大值或者平均值。常用池化层的核维度为2×2，步幅$S=2$。我们使用一个简单例子学习池化运算。

8.2.1 池化运算

- 首先把核对应到输入数据左上角的 4 个数字，如图 8-20。输出结果为 4 个数字的最大数字，$\max(-0.5,-2,2.9,2.45)=2.9$。

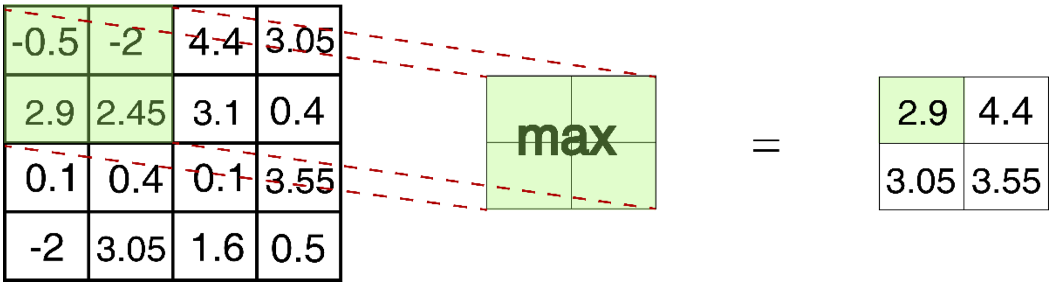

图 8-20 池化运算（第一步）

- 然后把核对应到输入数据右上角的 4 个数字，如图 8-21。输出结果为 4 个数字的最大数字，$\max(4.4,3.05,3.1,0.4) = 4.4$。

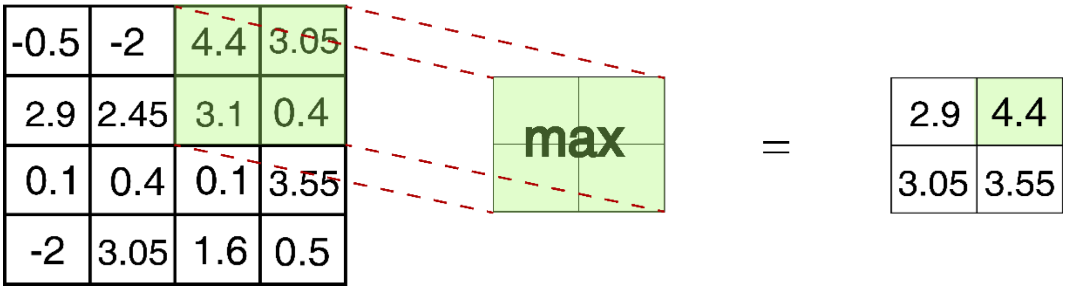

图 8-21 池化运算（第二步）

- 接着把核对应到输入数据的左下角的 4 个数字，如图 8-22，输出结果为 4 个数字的最大数字，$\max(0.1,0.4,-2,3.05) = 3.05$。

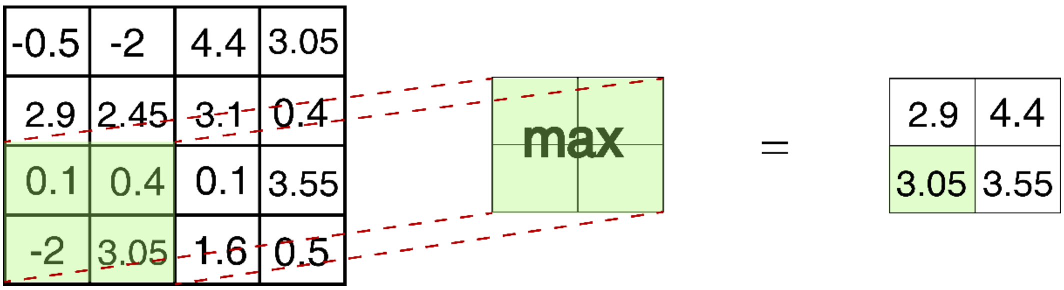

图 8-22 池化运算（第三步）

- 最后，把核对应到输入数据的右下角的 4 个数字，如图 8-23，输出结果为 4 个数字的最大数字，$\max(0.1,3.55,1.6,0.5) = 3.55$。

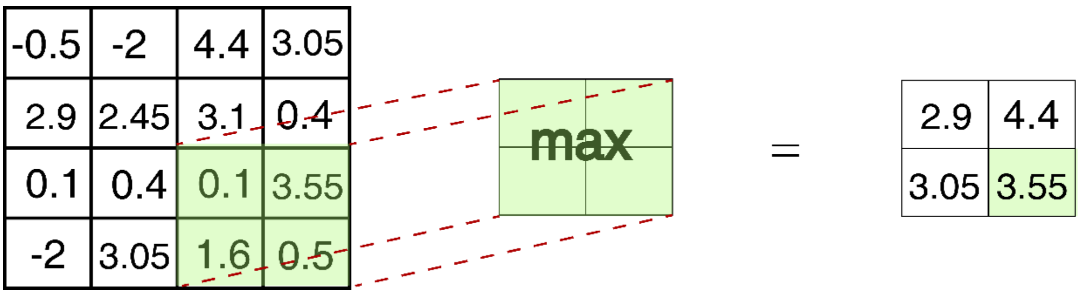

图 8-23 池化运算（第四步）

在上面例子中，池化运算输出结果的维度为2×2，而原先卷积层的维度为4×4。池化运算使得池化层的维度大幅变小。在上面例子中，核维度为2×2，步幅$S = 2$，池化层的宽和高将缩小一半。卷积层通常都是 3 维数组，我们对卷积层深度的每一层做池化运算。如果池化层的核维度为2×2，步幅$S = 2$，那么，池化层的宽和高缩小一半，深度保持不变，如图 8-24。

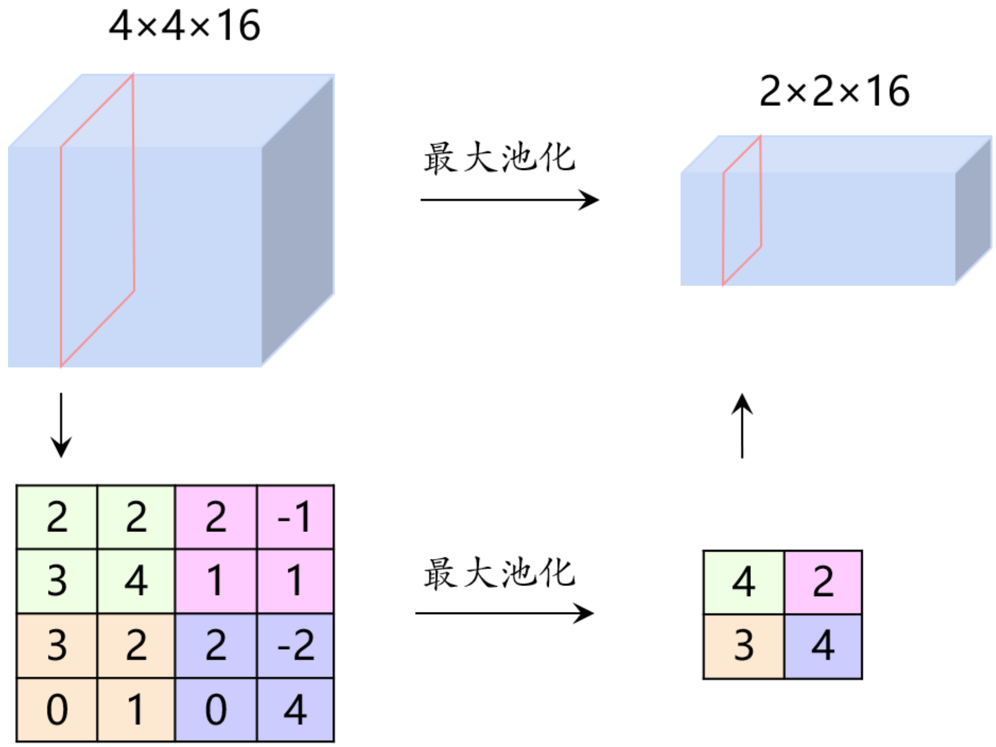

图 8-24 三维数组的池化运算

上面的池化运算称为最大池化（max pooling）。在实际中，还有另一种池化方法，称为平均池化（average pooling）。平均池化的工作原理和最大池化类似，但将求最大值替换成求平均值，如图 8-25。直观的看，最大池化保留卷积层小块区域的最大值，可以保留最大的信息；而平均池化可以让卷积层的结果更加平稳。

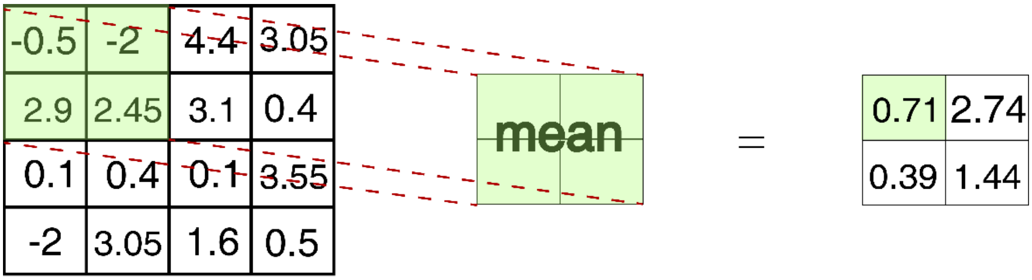

图 8-25 平均池化

在卷积神经网络中，通常在一个或者多个卷积层中会添加一层池化层。池化运算有如下作用。

- 从直观角度看，在实际应用中，感兴趣的物体通常不会出现在图片的固定位置；即使在连续拍摄的图片中，同一个物体也有可能出现像素位置上的偏移；这将会导致同一特征可能出现在卷积层的不同位置，对进一步的识别造成不便。池化运算只保留卷积输出相邻位置的最大值，因此可以减少物体位移对图片预测的影响。
- 从算法角度看，池化可以减少参数数量和减少计算量，因此可以控制过拟合。

在实际中，最常见的池化核的维度为2×2，步幅$S = 2$。有时候也会使用维度为3×3的核，步幅$S = 2$。

8.2.2 池化层求导

假设建立图 8-26 的卷积神经网络，包括输入层、卷积层、池化层和输出层。损失函数记为L。

- 正向传播算法，池化层结点m的值可以计算如下，$m = \max(a, b, c, d)$，进而计算输出层，最终计算损失函数L。
- 利用反向传播算法，可以计算损失函数L关于池化层的导数，其中包括L关于m的导数，$\frac{\partial L}{\partial m}$。
- 现在，我们计算损失函数L关于a，b，c，d的导数，$\frac{\partial L}{\partial a}$，$\frac{\partial L}{\partial b}$，$\frac{\partial L}{\partial c}$，$\frac{\partial L}{\partial d}$。

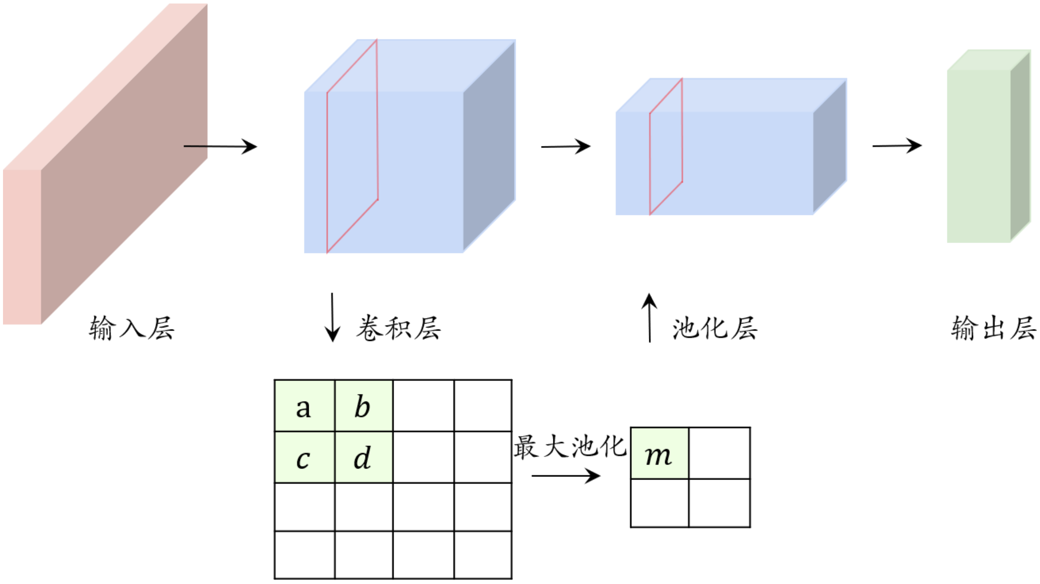

图 8-26 卷积神经网络，包括输入层、卷积层、池化层和输出层

- 如果$a > b$，$a > c$，$a > d$，
  - 那么，$m = a$，因为$\frac{\partial m}{\partial a} = 1$，$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial m}\frac{\partial m}{\partial a} = \frac{\partial L}{\partial m}$。
  - 这时，m和b，c，d没有关系，因此$\frac{\partial m}{\partial b} = \frac{\partial m}{\partial c} = \frac{\partial m}{\partial d} = 0$。进一步的，$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial m}\frac{\partial m}{\partial b} = 0$，$\frac{\partial L}{\partial c} = \frac{\partial L}{\partial m}\frac{\partial m}{\partial c} = 0$，$\frac{\partial L}{\partial d} = \frac{\partial L}{\partial m}\frac{\partial m}{\partial d} = 0$。
- 同理，
  - 如果$m = b$，$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial m}$，$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial c} = \frac{\partial L}{\partial d} = 0$。
  - 如果$m = c$，$\frac{\partial L}{\partial c} = \frac{\partial L}{\partial m}$，$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial b} = \frac{\partial L}{\partial d} = 0$。
  - 如果$m = d$，$\frac{\partial L}{\partial d} = \frac{\partial L}{\partial m}$，$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial b} = \frac{\partial L}{\partial c} = 0$。

再看一个例子。如图 8-27，正向传播算法对卷积层做了池化运算，使得结点的维度由4×4变为了2×2。在反向传播算法中，右下角2×2数组是损失函数关于池化层的导数。损失函数关于卷积层的导数大部分是 0，只有 25%的位置的导数不为 0。

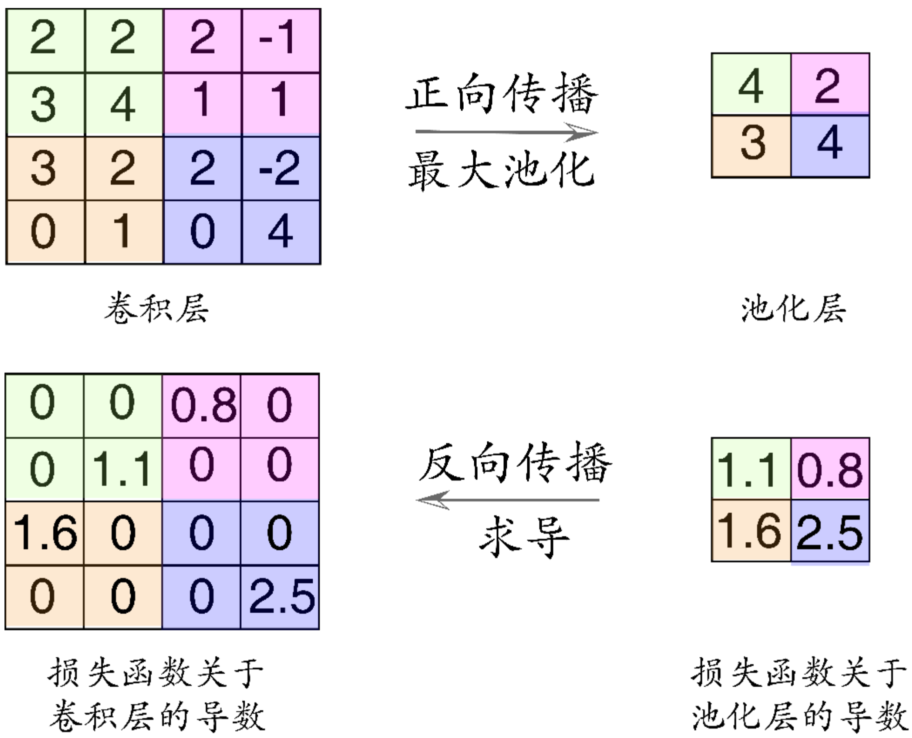

图 8-27 池化层求导

8.2.2 Python 计算池化层

下面定义函数 `maxpool()`，该函数的作用是实现池化运算。函数 `maxpool()` 的参数为:

- `Layer`：函数 `conv_2d()` 的结果，是一个 2 维数组
- `pool_rows`：池化层核的宽度，默认为 2
- `pool_cols`：池化层核的高度，默认为 2
- `batch_size`：每批数据的观测点个数
- `input_rows`：卷积层的宽度
- `input_cols`：卷积层的高度
- `num_kernels`：卷积层的深度

首先，把卷积层 layer 变换成 4 维的数组，维度分别为 batch size, 卷积层宽度, 卷积层高度, 卷积层深度；池化层的维度也是 4 维，宽度和高度为卷积层宽度和高度的一半，其他两个维度，batch size 和深度，保持不变，即池化层的维度为 batch_size, int（input_rows/2）, int（input_cols/2）, num_kernels。在这里，卷积层是一个 4 维数组，为了简便，我们逐层处理深度方向的每一层。在函数 `maxpool()` 中，最外层的 for 循环是卷积层的深度方向。对每一个深度方向的层，用函数 `get_image_section()` 提取出 pool_rows×pool_cols 大小的子集，然后用函数

np.concatenate()把卷积层的子集拼接在一起，接着，把数据列数变换为 pool_rows 乘以 pool_cols，如图 8-17。最后的结果为变换后二维数组每一行的最大值。同时，我们用 argmax_out 记录了每一行最大值的坐标。argmax_out 将用于求池化运算的导数。

```
def maxpool(layer, pool_rows, pool_cols, batch_size, \
                         input_rows, input_cols, num_kernels):

    # 变换卷积层维度
    layer = layer.reshape(batch_size, input_rows, input_cols, \
                                                  num_kernels)

    # layer_out_shape 为池化层结果的维度
    layer_out_shape = (layer.shape[0], int(layer.shape[1]/2), \
                       int(layer.shape[2]/2), layer.shape[3])
    # layer_out 为池化层结果的初始值
    layer_out = np.zeros(layer_out_shape)
    argmax_out = []
    for k in range(num_kernels):
        layer_sections = []
        for row_start in range(0, layer.shape[1] - pool_rows +
1, 2):
            for col_start in range(0, layer.shape[2] - \
                                            pool_cols + 1, 2):
                layer_sections.append(get_image_section(    \
                    layer[:,:,:,k], row_start,   \
                    row_start+pool_rows, col_start, \
                    col_start+pool_cols))
        layer_temp = np.concatenate(layer_sections, axis=1)
        layer_temp = layer_temp.reshape(layer_temp.shape[0]* \
                                        layer_temp.shape[1],  -1)

        # out 为 layer_temp 每一行最大值
        out = np.max(layer_temp, axis=1)
        # argmax_out 为 layer_temp 每一行最大值的坐标
        argmax_out.append(np.argmax(layer_temp, axis=1))

        layer_out[:,:,:,k] = out.reshape(layer_out_shape[:-1])
    return layer_out, argmax_out
```

接着定义函数 maxpool_2_deriv()，该函数的作用主要是在反向传播算法中计算最大池化的导数。函数 maxpool_2_deriv()的参数为:

- delta：损失函数关于池化层的导数
- pool_argmax：最大值位置坐标
- pool_rows：池化层核的宽度
- pool_cols：池化层核的高度
- batch_size：每批数据包含观测点个数
- input_rows：卷积层宽度

- `input_cols`：卷积层高度
- `num_kernels`：卷积层深度

我们在卷积层深度方向逐层计算导数。求导数的主要思想是：在卷积层深度方向每一层与池化核对应的区域中，最大值对应的导数为损失函数关于对应池化层结点的导数，其余导数都设为 0。

```
def maxpool_2_deriv(delta, pool_argmax, pool_rows, pool_cols, \
                batch_size, input_rows, input_cols, num_kernels):

    # 初始化卷积层的 delta
    delta_conv = np.zeros((batch_size, input_rows, input_cols,\
                          num_kernels))

    # 池化层的高和宽
    after_pool_rows = int(input_rows/pool_rows)
    after_pool_cols = int(input_cols/pool_cols)

    # 变换池化层维度
    delta = delta.reshape(batch_size, after_pool_rows, \
                          after_pool_cols, num_kernels)
    for k in range(num_kernels):
        # delta_k 为卷积层 delta 在深度方向的一层
        # delta_k 全部元素初始化的值都为 0
        delta_k = np.zeros((int(batch_size*after_pool_rows*  \
                     after_pool_cols), pool_rows*pool_cols))
        # delta_k 每一行中池化运算最大值位置设为损失函数关于池化层的导数
        # 函数.flatten()可以把数组变为一个向量
        delta_k[:, pool_argmax[k]] = delta[:,:,:,k].flatten()
        # 变换 delta_k 的维度为:
        # batch_size, after_pool_rows*after_pool_cols,
        # pool_rows*pool_cols
        delta_k = delta_k.reshape(batch_size,after_pool_rows*\
                     after_pool_cols, pool_rows*pool_cols)

        # for 循环中，把 delta_k 变换成合适的维度，并放入 delta_conv 中
        delta_k_row = 0
        for row_start in range(0, input_rows - pool_rows + 1,\
                                              pool_rows):

            for col_start in range(0, input_cols - pool_cols +\
                                            1,  pool_cols):

                delta_conv[:,row_start:(row_start+pool_rows), \
                          col_start:(col_start+pool_cols),k] =\
                          delta_k[:,delta_k_row,:].reshape( \
                          batch_size, pool_rows, pool_cols)
                delta_k_row += 1

    return delta_conv
```

8.3 卷积神经网络

现在把 8.1 节和 8.2 节学过的关于卷积层和池化层的知识应用到 mnist 数据中。我们建立如图 8-27 的卷积神经网络（CNN），包括输入层、卷积层、池化层、全连接层和输出层。

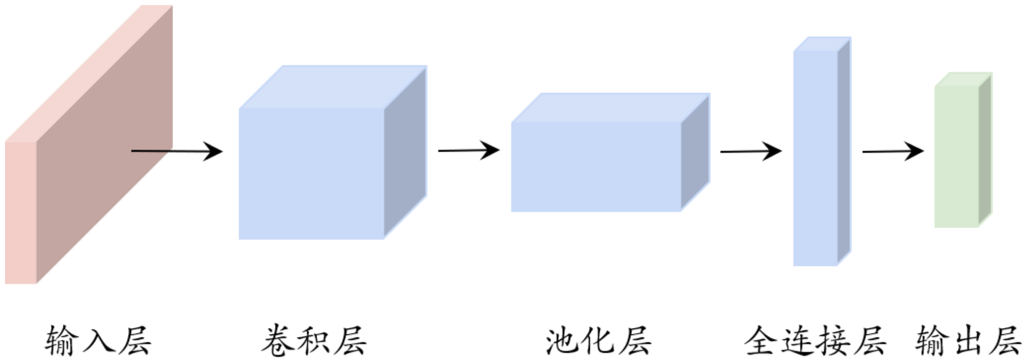

图 8-27 卷积神经网络：包括输入层、卷积层、池化层、全连接层和输出层

池化层到全连接层的计算中，为了计算方便，把 3 维数组变换成一个 1 维数组，如图 8-28 中的虚线框。

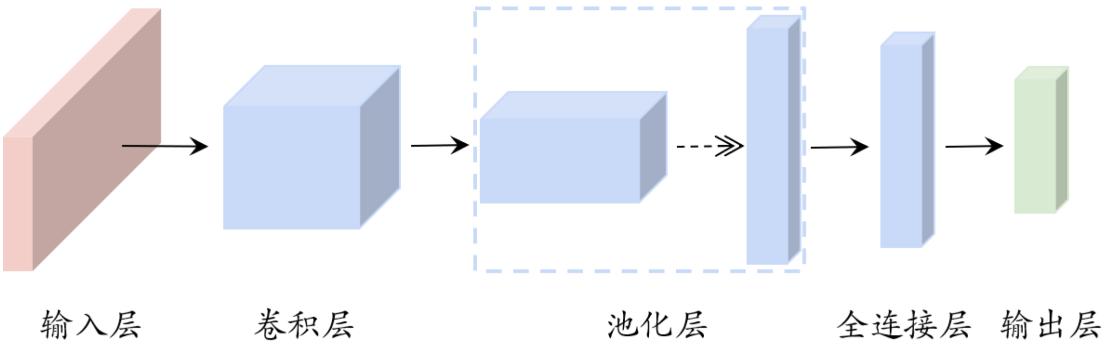

图 8-28 池化层从 3 维数组变换成 1 维数组（虚线框部分）

- 首先加载所需的包和数据，并对数据进行预处理。和第七章一样，把数据分成三个部分，训练数据（50000 幅图片）、验证数据（10000 幅图片）和测试数据（10000 幅图片）。

```
"""
载入需要用到的包和数据
"""
%config InlineBackend.figure_format = 'retina'
import idx2numpy
import matplotlib.pyplot as plt
import numpy as np

x_train = idx2numpy.convert_from_file(\
                      './data/mnist/train-images.idx3-ubyte')
y_train  = idx2numpy.convert_from_file(\
```

```
                     './data/mnist/train-labels.idx1-ubyte')
x_test = idx2numpy.convert_from_file(\
                      './data/mnist/t10k-images.idx3-ubyte')
y_test = idx2numpy.convert_from_file(\
                      './data/mnist/t10k-labels.idx1-ubyte')
"""
获得数据，并对因变量进行 one-hot 编码
"""
np.random.seed(1)

train_images, train_labels = (x_train/255, y_train)

# 训练数据的 one-hot 编码
one_hot_labels = np.zeros((len(train_labels), 10))
for i,j in enumerate(train_labels):
    one_hot_labels[i][j] = 1
train_labels = one_hot_labels


"""
把训练数据(60000*728)分成训练数据(50000*728)和验证数据(10000*728)
"""
index = np.arange(len(train_images))
np.random.shuffle(index)

# 验证数据
valid_images, valid_labels = train_images[index[-10000:]], \
                                 train_labels[index[-10000:]]
# 训练数据
train_images, train_labels = train_images[index[:50000]], \
                                  train_labels[index[:50000]]

test_images = x_test/255                          # 测试数据自变量矩阵
test_labels = np.zeros((len(y_test), 10))   # 测试数据 one-hot 编码

for i,j in enumerate(y_test):
    test_labels[i][j] = 1
```

- 使用 relu 函数为激活函数，定义函数 `relu()`和 `relu2deriv()`，分别计算 relu 函数的函数值和导数。同时，定义函数 `softmax()`，用于得到最终的输出值。

```
def relu(x):
    return (x>0) * x

def relu2deriv(x):
    return (x>0)

def softmax(x):
    temp = np.exp(x)
    return temp/np.sum(temp, axis=1, keepdims=True)
```

- 接着，为了计算卷积层，定义 3 个函数：`get_image_section()`，`conv_reshape()` 和 `conv_2d()`。为了计算池化层，定义函数 `maxpool()`；为了计算池化层的导数，定义函数 `maxpool_2_deriv()`。

```
def get_image_section(layer, row_start, row_end, col_start, 
col_end):
    section = layer[:, row_start:row_end, col_start:col_end]
    return section.reshape(-1, 1, row_end-row_start, \
                                     col_end-col_start)

def conv_reshape(image, kernel_rows, kernel_cols):
    image = np.pad(image, ((0,0),(1,1),(1,1)), mode='constant')
    image_sections = []
    for row_start in range(image.shape[1] - kernel_rows + 1):
        for col_start in range(image.shape[2] - kernel_cols + 
1):
            image_sections.append(get_image_section(image, \
                           row_start, row_start+kernel_rows, \
                           col_start, col_start+kernel_cols))
    expanded_input = np.concatenate(image_sections, axis=1)
    es = expanded_input.shape
    layer = expanded_input.reshape(es[0]*es[1], -1)
    return layer

def conv_2d(layer, kernels, bias):
    layer = np.dot(layer, kernels) + bias
    return relu(layer)

def maxpool(layer,pool_rows,pool_cols,batch_size,input_rows, \
                                      input_cols, num_kernels):
    layer = layer.reshape(batch_size, input_rows, input_cols,\
                                                  num_kernels)

    layer_out_shape = (layer.shape[0], int(layer.shape[1]/2), \
                       int(layer.shape[2]/2), layer.shape[3])
    layer_out = np.zeros(layer_out_shape)
    argmax_out = []
    #layer_2_deriv = np.zeros(layer.shape)
    for k in range(num_kernels):
        layer_sections = []
        for row_start in range(0, layer.shape[1] - pool_rows + 
1, 2):
            for col_start in range(0, layer.shape[2] - \
                                        pool_cols + 1, 2):
                layer_sections.append(get_image_section(\
                layer[:,:,:,k], row_start, row_start+pool_rows,
                    col_start, col_start+pool_cols))
        layer_temp = np.concatenate(layer_sections, axis=1)
        layer_temp = layer_temp.reshape(layer_temp.shape[0]* \
                                  layer_temp.shape[1], -1)
        out = np.max(layer_temp, axis=1)
        argmax_out.append(np.argmax(layer_temp, axis=1))
        layer_out[:,:,:,k] = out.reshape(layer_out_shape[:-1])
    return layer_out, argmax_out

def maxpool_2_deriv(delta, pool_argmax, pool_rows, pool_cols, \
              batch_size, input_rows, input_cols, num_kernels):

    delta_conv = np.zeros((batch_size, input_rows, input_cols,\
```

```
                             num_kernels))

    after_pool_rows = int(input_rows/pool_rows)
    after_pool_cols = int(input_cols/pool_cols)

    delta=delta.reshape(batch_size, after_pool_rows, \
                             after_pool_cols, num_kernels)
    for k in range(num_kernels):
        delta_k = np.zeros((int(batch_size*after_pool_rows* \
                       after_pool_cols), pool_rows*pool_cols))
        delta_k[:, pool_argmax[k]] = delta[:,:,:,k].flatten()
        delta_k = delta_k.reshape(batch_size, \
                 after_pool_rows*after_pool_cols, \
                 pool_rows*pool_cols)
        delta_k_row = 0
        for row_start in range(0, input_rows - pool_rows + 1,
2):
            for col_start in range(0, input_cols - pool_cols +
1, 2):
                delta_conv[:,row_start:(row_start+pool_rows),\
                         col_start:(col_start+pool_cols),k]
                = delta_k[:,delta_k_row,:].reshape( \
                       batch_size, pool_rows, pool_cols)
                delta_k_row += 1

    return delta_conv
```

- 给定超参数值和初始化参数。卷积层核的宽度，高度和深度分别设为3，3，16。池化层核的宽度和高度分别设为2，2。全连接层的结点数为 512。所有层的截距项都设为 0，卷积层核，池化层到全连接层的权重，全连接层到输出层的权重都从0.1到0.2的均匀分布中产生。

```
lr, epoches = 0.5, 100
pixels_per_image, num_labels = 784, 10

input_rows, input_cols = 28, 28
# 卷积层核的宽度，高度和深度
kernel_rows, kernel_cols, num_kernels = 3, 3, 16
pool_rows, pool_cols = 2, 2
FC_size = 512

batch_size = 100
num_batch = int(len(train_images)/batch_size)

hidden_maxpool = int((input_rows/2)*(input_cols/2)*num_kernels)


b_kernels = np.zeros((1, num_kernels))
b_2_3 = np.zeros((1, FC_size))    # b_0_1 初始值,全部为 0
b_3_4 = np.zeros((1, num_labels)) # b_1_2 初始值,全部为 0

kernels = 0.2 * np.random.random((kernel_rows*kernel_cols, \
                             num_kernels)) - 0.1 # 卷积核的维度
w_2_3 = 0.2 * np.random.random((hidden_maxpool, FC_size)) - 0.1
w_3_4 = 0.2 * np.random.random((FC_size, num_labels)) - 0.1
```

- 最后，正式开始训练卷积神经网络模型。整个过程分成 4 个部分。
  - 第一部分：使用正向传播算法计算每一层结点值
  - 第二部分：使用反向传播算法计算损失函数关于权重和截距项的导数，并更新权重和截距项
  - 第三部分：每隔 20 epoch 计算一次验证误差
  - 第四部分：计算测试误差

```
for epoch in range(epoches):
    correct_cnt, val_correct_cnt = 0, 0
    error, val_error =0.0, 0.0

    for i in range(num_batch):
# ------------------------------------------------------------#
# 第一部分：正向传播算法
        batch_start, batch_end = i*batch_size, (i+1)*batch_size
        batch_image = train_images[batch_start:batch_end]
        batch_label = train_labels[batch_start:batch_end]
        layer_0 = conv_reshape(batch_image, kernel_rows, \
                                            kernel_cols)

        layer_1 = conv_2d(layer_0, kernels, b_kernels)
        layer_1_shape = layer_1.shape

        layer_2, pool_argmax = maxpool(layer_1, pool_rows, \
                        pool_cols, batch_size, \
                        input_rows, input_cols, num_kernels)
        layer_2 = layer_2.reshape(batch_size, -1)

        layer_3 = relu(np.dot(layer_2, w_2_3) + b_2_3)
        layer_4 = softmax(np.dot(layer_3, w_3_4) + b_3_4)
# ------------------------------------------------------------#
# 第二部分：反向传播算法，更新权重
        for k in range(batch_size):
            error -= np.log(layer_4[k:k+1, np.argmax(\
                            batch_label[k:k+1])])

            correct_cnt += int(np.argmax(layer_4[k:k+1]) == \
                               np.argmax(batch_label[k:k+1]))

        layer_4_delta = (layer_4 - batch_label)/batch_size
        layer_3_delta = np.dot(layer_4_delta, w_3_4.T)* \
                        relu2deriv(layer_3)

        layer_2_delta = np.dot(layer_3_delta, w_2_3.T)
        layer_2_delta = maxpool_2_deriv(layer_2_delta,    \
               pool_argmax, pool_rows, pool_cols, batch_size,\
               input_rows, input_cols, num_kernels)
        layer_1_delta = layer_2_delta.reshape(layer_1_shape)* \
                        relu2deriv(layer_1)


        b_3_4 -= lr * np.sum(layer_4_delta, axis=0, \
                                          keepdims=True)
```

```
        b_2_3 -= lr * np.sum(layer_3_delta, axis=0, \
                                          keepdims=True)
        b_kernels -= lr * np.sum(layer_1_delta, axis=0, \
                       keepdims=True)/(input_rows * input_cols)

        w_3_4 -= lr * layer_3.T.dot(layer_4_delta)
        w_2_3 -= lr * layer_2.T.dot(layer_3_delta)
        kernels -= lr * layer_0.T.dot(layer_1_delta)/  \
                                   (input_rows  *  input_cols)

# ------------------------------------------------------------#
# 第三部分：每隔 20 epoch，计算一次验证误差
    if (epoch % 20==0 or epoch==epoches-1):
# 每 100 epochs 输出训练误差，测试误差等信息
        layer_0 = conv_reshape(valid_images, kernel_rows,\
                                           kernel_cols)
        layer_1 = conv_2d(layer_0, kernels, b_kernels)
        layer_1 = layer_1.reshape(len(valid_images), -1)

        layer_2, _ = maxpool(layer_1, pool_rows, pool_cols, \
                              len(valid_images), input_rows, \
                              input_cols, num_kernels)

        layer_3  =  relu(np.dot(layer_2.reshape(len(valid_images),
                        -1), w_2_3) + b_2_3)
        layer_4 = softmax(np.dot(layer_3, w_3_4) + b_3_4)

        for k in range(len(valid_images)):
            val_error -= np.log(layer_4[k:k+1, \
                                  np.argmax(valid_labels[k:k+1])])
            val_correct_cnt += int(np.argmax(layer_4[k:k+1]) ==
                                  np.argmax(valid_labels[k:k+1]))
        print("e:%3d; Train_Err: %0.2f; Train_Acc: %0.2f; \
               Val_Err: %0.2f; Val_Acc: %0.3f" % \
              (epoch, error/float(len(train_images)), \
              correct_cnt/float(len(train_images)), \
              val_error/float(len(valid_images)), \
              val_correct_cnt/float(len(valid_images))))

# ------------------------------------------------------------#
# 第四部分：计算测试误差
num_test_train = len(test_images)

layer_0 = conv_reshape(test_images, kernel_rows, kernel_cols)
layer_1 = conv_2d(layer_0, kernels, b_kernels)
layer_1 = layer_1.reshape(num_test_train, -1)

layer_2, _ = maxpool(layer_1, pool_rows, pool_cols,  \
             num_test_train,input_rows,  input_cols,  num_kernels)
layer_2 = layer_2.reshape(num_test_train, -1)

layer_3 = relu(np.dot(layer_2, w_2_3) + b_2_3)
layer_4 = softmax(np.dot(layer_3, w_3_4) + b_3_4)

test_error, test_correct_cnt = 0, 0
```

```
for k in range(num_test_train):
    test_error -= np.log(layer_4[k:k+1, \
                            np.argmax(test_labels[k:k+1])])

    test_correct_cnt += int(np.argmax(layer_4[k:k+1]) == \
                            np.argmax(test_labels[k:k+1]))

print("Test Loss: %0.3f  Test Acc: %0.3f"% \
       (test_error/num_test_train,    \
       test_correct_cnt/num_test_train))

e: 0; Train_Err: 0.43; Train_Acc: 0.88; Val_Err: 0.21; Val_Acc:
0.940
e:20; Train_Err: 0.01; Train_Acc: 1.00; Val_Err: 0.05; Val_Acc:
0.987
e:40; Train_Err: 0.00; Train_Acc: 1.00; Val_Err: 0.05; Val_Acc:
0.988
e:60; Train_Err: 0.00; Train_Acc: 1.00; Val_Err: 0.05; Val_Acc:
0.988
e:80; Train_Err: 0.00; Train_Acc: 1.00; Val_Err: 0.06; Val_Acc:
0.988
e:99; Train_Err: 0.00; Train_Acc: 1.00; Val_Err: 0.06; Val_Acc:
0.988

Test Loss: 0.052  Test Acc: 0.987
```

从上面运行结果可以看出，该卷积神经网络模型收敛得很快，20 次循环之后，训练准确率就达到了 0.999，验证准确率达到了 0.987。卷积神经网络最终测试准确率达到了 0.987，比第七章建立的使用 dropout 方法的全连接神经网络的准确率还高大约0.5%。

8.4 本章小结

在本章，我们学习了卷积神经网络模型，着重介绍了卷积神经网络模型的两个重要组成部分：卷积层和池化层。卷积层特殊的运算方式使得卷积神经网络有如下优点。

- 每一个核作用于输入层不同小区域，使得卷积神经网络可以提取出不同小区域的特征。
- 核（核可以认为是卷积运算的权重矩阵）的维度通常较小，与全连接的网络相比，减少了参数数量。在一定程度上，卷积运算也控制了过拟合。

池化层的作用在于进一步减少噪声和参数的数量，使得卷积神经网络可以更好控制过拟合。虽然本章实现卷积层、池化层以及池化层导数的代码较为复杂，你可以尝试自行推导卷积层和池化层的求导公式，自己编程实现卷积神经网络，这样可以帮助你加深对卷积神经网络的理解。本章代码主要是为了说明卷积神经网络的工作原理和流程，运行效率较低。我们将在第十章介绍如何使用 TensorFlow 框架建立高效的卷积神经网络。

习题

1. 分析 mnist 数据，尝试自己编程实现本章学习的卷积神经网络，尝试不同的初始值、学习步长和卷积层核的维度。
2. 分析 Fasion-mnist 数据，建立卷积神经网络，计算测试准确率，和 TensorFlow 建立的普通神经网络相比，卷积神经网络的测试准确率提高了吗？
3. 分析 Fasion-mnist 数据，建立卷积神经网络，尝试不同的初始值、学习步长和卷积层核的维度。
4. 分析 Fasion-mnist 数据，建立卷积神经网络，尝试去除全连接层，如 8-29。测试准确率降低还是提高了？

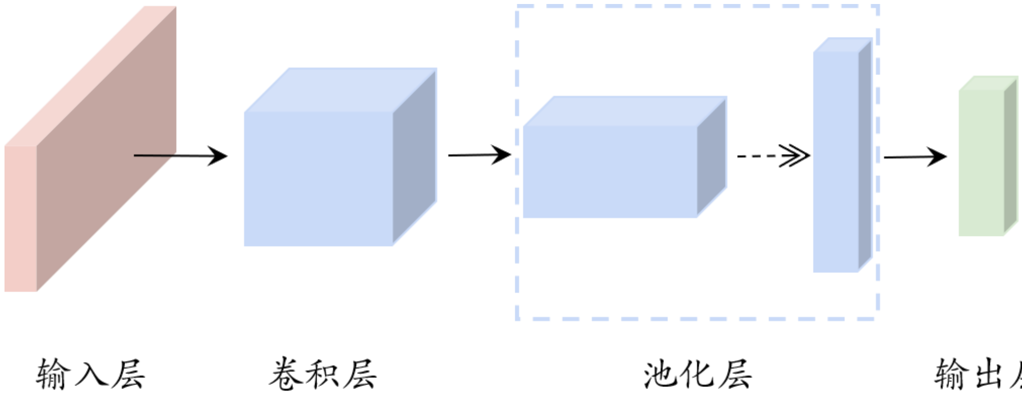

图 8-29 卷积神经网络（没有全连接层）

5. 分析 Fasion-mnist 数据，建立卷积神经网络，尝试去除池化层，如 8-30。测试准确率降低还是提高了？

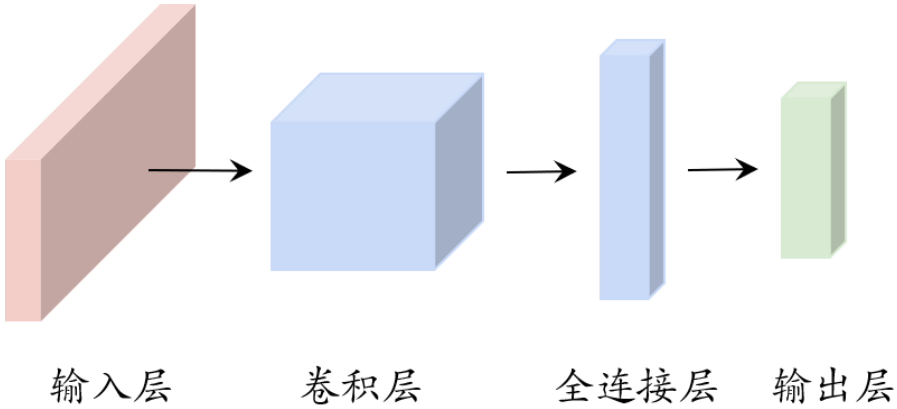

图 8-30 卷积神经网络（没有池化层）