第 3 章 卷积神经网络

卷积神经网络(Convolutional Neural Network,CNN)是重要的神经网络模型,最初设计用于图像特征提取,随着模型的不断发展,它在其他领域也展现出卓越的性能和强大的能力,成为深度学习领域不可或缺的模型之一。

卷积神经网络的起源可追溯到 20 世纪 80 年代,日本科学家福岛邦彦提出了 Neocognitron 神经网络模型,这是卷积神经网络模型的雏形。1998 年,Yann LeCun 等提出了 LeNet-5,这是经典的卷积神经网络模型,包含一个输入层,两个卷积层,两个池化层和 3 个全连接层,被应用于手写数字识别,取得了很好的效果,但受限于当时的计算能力,卷积神经网络的训练十分耗时,这在一定程度上制约了卷积神经网络进一步的发展与应用,从而导致它在一段时间内未受到学者的重视。

随着计算机硬件技术的发展,尤其是适用于神经网络模型并行训练的 GPU 的性能提升,卷积神经网络逐渐重新受到学者关注。2012 年,Hinton 和其学生 Alex Krizhevsky 等提出的 AlexNet 模型在 ImageNet 大规模视觉识别挑战赛(ILSVRC)中获得了冠军,标志着卷积神经网络进入快速发展阶段。

此后,关于卷积神经网络模型的研究成果如雨后春笋般不断涌现,学者们相继提出了 GoogLeNet、ResNet 等模型,在图像分类、目标检测、自动驾驶、语音识别等领域提供了强有力的技术支撑。在当前主流的深度学习模型中,卷积、反卷积等操作已经成为常用的特征提取和生成手段。

3.1 模型结构

本节先从设计思路入手,讨论卷积神经网络对生物视觉神经机制的模仿,随后详细介绍卷积层、池化层、全连接层和激活函数的原理与结构。

3.1.1 设计思路

卷积神经网络作为专注于识别图像特征的神经网络分支,其设计思路源于对生物视觉神经系统的模仿。第 2 章讨论了神经网络模型的设计借鉴了生物神经系统机制,而卷积神

经网络在此基础上，进一步围绕视觉神经机制展开。

1. 生物视觉机制中的感受野

在生物视觉系统中，眼睛接收的视野图像传入大脑后，单个神经元并不对整个视野进行响应，而只对视野中的一个特定区域产生反应，这个区域被称为感受野(Receptive Field)。具体而言，视网膜上的神经细胞只对视网膜上的某个小区域的光刺激产生反应。

卷积神经网络借鉴了这一特性。在卷积运算中，卷积核在输入图像上滑动，每次仅对局部区域进行处理。当输入图像的局部区域与卷积核所代表特征相匹配时，卷积运算结果值较大；当局部区域与卷积核的特征不相符时，卷积运算结果值较小，因此卷积运算结果的大小直观地反映了卷积核是否成功识别输入图像的局部特征。

2. 分层提取特征机制

在视觉神经系统中，低级的神经皮层(如 V1 区)负责检测简单特征，深层神经皮层(如 IT 区)则负责检测复杂特征。

卷积神经网络借鉴了分层提取特征机制。与多层感知机类似，卷积神经网络也设计了多层网络结构，浅层网络负责识别简单特征，深层网络负责识别复杂特征。以识别汽车图像模型为例，浅层网络负责识别车轮圆形轮廓、车身的线条等简单特征；中层网络将低级特征组合，形成车轮、车灯等局部部件；深层网络则整合部件特征，对各部件的位置关系建立模型，识别汽车整体结构、车型等信息。

3. 平移不变性与权值共享

生物视觉系统具有平移不变性，即同一物体出现在视野的不同位置时均可被感知，例如无论太阳出现在天空的西边还是东边，人眼都能识别。

卷积神经网络的设计同样符合这一特征。卷积运算的结果只与局部图像与卷积核特征的匹配程度有关，与图像位置无关。这一特性与卷积神经网络的权值共享机制高度契合。不同于全连接神经网络，在卷积神经网络的卷积层中，神经元与上一层神经元之间的连接权重并非都需要单独训练，而是通过卷积计算强制性地使部分权重相等。具体来讲，卷积核在输入图像上滑动时，无论处于图像的哪个位置，均采用同一卷积核的权重进行计算，而非对每个位置单独建模。这种方式极大地减少了卷积神经网络的参数量，显著地提升了计算效率。

4. 空间下采样机制

下采样是指减少数据规模的同时尽可能地保留关键信息。生物神经系统在处理信息时会逐步减少细节信息，只保留关键特征，例如人脑在识别汽车时会忽略纯色车身的颜色信息，保留车轮与地面交界处的轮廓；忽略天空等无关的背景，保留了车身边缘轮廓信息。

卷积神经网络同样使用了下采样机制，通过池化操作降低图像分辨率，压缩数据维度；同时，卷积核负责检测重要信息，而 ReLU 等激活函数通过引入非线性变换，抑制不显著的激活信号，实现降低噪声的效果。

3.1.2 卷积层

卷积层是卷积神经网络的核心，其作用是通过卷积核对图像进行扫描，提取相应的特

征。卷积核可以理解为一种过滤器(也称为图像扫描器、特征扫描器、局部感受野),当图像中的局部区域与卷积核所代表的特征匹配时,卷积运算的结果值较高,表明特征被成功提取;当局部区域与卷积核特征不匹配时,结果值较低,表明未提取到相应特征。卷积的具体计算方法将在下文详细说明,本节从定性角度介绍其核心逻辑。

卷积层的工作方式可通过一个直观的例子理解。假设医生通过 X 光片检测肿瘤,整张 X 光片相当于输入图像,其中的每个像素对应神经网络输入层的神经元。卷积层的输出则反映每个局部区域是否存在肿瘤的判断。假设用 5×5 的卷积核扫描 X 光片,卷积层的每个神经元(对应卷积输出的一个点)只负责判断 X 光片中的一个 5×5 的局部区域是否存在肿瘤细胞。这类似于医生聚焦在 X 光片中的某个 5×5 区域,仅通过该区域的细节判断是否有肿瘤,无须关注整张 X 光片的其他部分。

这种“局部连接”的方式与全连接网络存在明显差异。在全连接网络中,每个输出神经元需要与输入层的所有像素相连,相当于医生在检查每个 5×5 区域是否有肿瘤时,必须先看完整张 X 光片的所有像素,这显然会造成大量无效计算,效率极低,而卷积层的局部连接则让每个神经元只关注对应的局部区域,既提高了细节捕捉的能力,又避免了冗余信息的干扰。

局部连接设计的优势主要体现在两方面。从仿生角度看,生物视觉神经元无须对全部图像进行感知,只需响应局部刺激,例如人眼观察某个物体时,视野中该物体之外的部分会自然虚化,距离较远的元素的相关性比较弱。卷积层的局部连接正是模仿了这一特性,让神经网络无须处理全图信息,只需聚焦局部特征,从而大幅提升了处理高分辨率图像的效率。从计算角度看,卷积操作大幅降低了计算量。以上述 X 光检测肿瘤为例,医生在对每个小区域进行检测时,每次只需关注 X 光片相应位置的 5×5 像素,而无须重复检查整张 X 光片,避免不必要的计算。

此外,CNN 中存在权值共享机制,即同一卷积核在扫描整张图像时,使用的是同一套参数。对应到上述例子,医生在扫描整张 X 光片时,使用的是同一套肿瘤检测标准(例如黑色阴影、圆形),无论检查到左上角还是右下角,判断标准都不会改变,而全连接神经网络并不具备这一特性,每个位置的肿瘤判断标准都需要重新训练。CNN 的这一特性不仅大幅降低了参数量,而且可以确保同一特征无论在图像的哪个位置都能被识别出来,例如无论肿瘤长在左肺还是右肺,卷积核都能通过相同标准检测出来,体现了 CNN 平移不变性的特征。

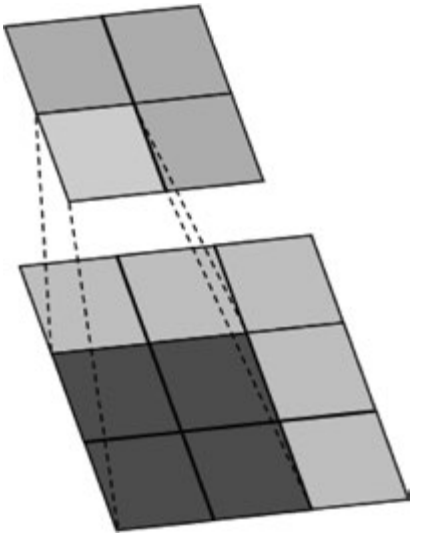

图 3-1　卷积运算的示例

卷积运算的示例如图 3-1 所示,其中下方 3×3 的矩阵为输入图像,上方 2×2 的矩阵为输出图像。将输入图像左下角 2 行 2 列的区域与 2×2 卷积核进

行运算，得到输出图像左下角 1×1 的结果。

3.1.3 池化层

池化又叫子采样，与卷积操作原理类似，其核心目的是降维。以 250×250 像素的图像为例，采用 2×2 大小的池化窗口从上到下、从左到右扫描图像。最大化池化在每个 2×2 的局部区域中，选取像素值最大的点代替此区域，同时舍弃其他 3 像素；平均池化是计算这 4 像素值的平均值代替此区域。需要注意的是，平均池化可看作特殊的卷积操作，此时对应的卷积核中的每个元素的值均为 $1/n$（n 指池化窗口像素总数，此处为 4）。

结合前文 X 光片检测肿瘤的例子来看，假设 X 光片分辨率过高，但实际上并不需要如此清晰的图像完成肿瘤检测，过高的分辨率会增加医生逐像素检查的工作量，影响检测效率。此时，可通过池化降低 X 光片图像的分辨率，仍能满足检测需求。

3.1.4 全连接层

全连接层在 CNN 中扮演着重要角色。在 CNN 内，卷积层和池化层的输入及输出均为二维图像形式，但 CNN 的常见任务是图像分类，这类任务需要的是能直接对应类别判断的输出，而非二维形式。

在处理图像分类问题时，通常预先设定若干类别，输出层配置与类别数量相同的神经元，采用 Softmax 激活函数计算。通过 Softmax 函数，可判断哪个神经元在激活后产生的信号最强烈，进而确定模型输出的预测类别。

由于 Softmax 激活函数仅接收一维输入，因此需要先将卷积层和池化层输出的二维数据展开为一维向量，再通过全连接层将提取的特征信息整合、映射到与类别对应的输出维度，最后接入多层感知机完成分类。

3.1.5 激活函数

卷积层和池化层一般使用 ReLU 激活函数，将显著的信号激活并传递到下一层，将不显著的信号过滤为 0。全连接层一般使用 Softmax 激活函数，因为 Softmax 适合多分类问题，而卷积神经网络一般用于图像分类和识别问题中，属于多分类问题。

3.2 卷积与池化的计算

卷积和反卷积是深度学习中常用的运算，常见于 CNN、GAN 等模型。

3.2.1 卷积的概念

卷积是一种数学运算，记$(f*g)(n)$为 f 与 g 的卷积

连续的卷积定义见式(3-1)：

$$(f*g)(n)=\int_{-\infty}^{\infty} f(\tau)g(n-\tau)\mathrm{d}\tau \tag{3-1}$$

离散的卷积定义见式(3-2)：

$$(f*g)(n)=\sum_{\tau=-\infty}^{\infty}f(\tau)g(n-\tau) \tag{3-2}$$

从定义式中可发现卷积的要点：①卷积的入参为 n；②两个函数遍历所有可能的入参，求乘积后求和；③在此过程中保证两个函数的入参之和为 n。

3.2.2 卷积的例子

下面列举若干实例，介绍卷积运算原理，加深理解。

1. 抛骰子

假设抛出两个骰子，记 $f(x)$和 $g(x)$分别为两个骰子点数为 x 的概率。现在求出两枚骰子点数之和为 4 的概率，则此概率可以表示为卷积，见式(3-3)：

$$(f*g)(4)=\sum_{\tau=1}^{3}f(\tau)g(4-\tau) \tag{3-3}$$

从公式可以看出，g 函数先对 τ 求了相反数，又增加了 4，相当于 g 先做了翻转，骰子点数变成倒序的 6、5、4、3、2、1，再做了平移，平移量就是 4。

2. 复利

假设小明往银行存入 100 元，年利率是 5%，按复利计算，第 n 年的金额会变为 $100\times(1.05)^n$。假设小明每年末都存入 100 元，求第 5 年末小明的银行总存款金额。这一问题也可用卷积表示。可将小明的总存款分解为 6 次存款得到的总资金。

初始存款，将享受 5 年的利息，复利到第 5 年末总资金为 $100\times(1.05)^5$，第一年末存款，将享受 4 年的利息，复利到第 5 年末总资金为 $100\times(1.05)^4$。以此类推，第 5 年末存款，不享受利息，第 5 年总资金为 $100\times(1.05)^0$。记 $f(x)$为第 x 年存入的本金，$g(x)$为经过 x 年得到的利息，则第 5 年末总资金为 $(f*g)(5)=\sum_{\tau=0}^{5}f(\tau)g(5-\tau)$。

3. MOBA 游戏的延迟伤害

在一些 MOBA(Multiplayer Online Battle Arena)类游戏中，有的角色对敌人攻击有特殊的延迟机制，如接下来 5 秒内，敌方英雄每秒都会受到伤害。现在该角色对敌人每秒都攻击 1 次，每次攻击都会在未来 5 秒造成伤害。计算第 3 秒敌方受到的伤害，即为第 0 秒、第 1 秒、第 2 秒、第 3 秒攻击在第 3 秒时造成的伤害之和。记 $f(x)$为第 x 秒攻击造成的伤害，$g(x)$为 x 秒后伤害衰减的比例，则第 3 秒造成伤害之和为 $(f*g)(3)=\sum_{\tau=0}^{3}f(\tau)g(3-\tau)$。

4. 以上例子的归纳

从以上例子可归纳，卷积的主要特点是：函数 f 表示一个输入，随时间变化；输入对输出结果有持续的影响，这个影响用函数 g 表示；最终的结果是若干历史时刻对当前时刻影响的线性叠加，见表 3-1。

表 3-1 卷积例子的归纳

例子	f 的含义	g 的含义	卷积的结果
抛骰子	第 1 个骰子点数概率	在此例子中似乎并不明显，可以这样理解：例如第 1 个骰子的点位是 1，对于最终结果点位之和是 2、3、4、5、6、7 都有影响。第 2 个骰子的概率反映了第 1 个骰子多大程度影响了最终结果	第 1 个骰子点数叠加 g 的影响后得到点数之和为 4 的概率
复利	历年的存款	历年存款对当前金额的影响，即复利计算	历年存款复利到当前金额之和
MOBA 游戏	攻击的伤害	攻击给对手造成延迟伤害	之前所有的攻击在此时造成的伤害之和

3.2.3 卷积的计算

卷积计算是模型的重要特性，本节分析其原理。

1. 计算方法

假设卷积核和原始图像都是二维的，并假设卷积核的大小是 2 行 2 列。在计算卷积时，首先卷积核与原始图像左上角相同大小的部分（前两行前两列）对应元素相乘并求和，得到在该位置卷积的结果。随后卷积核向右移动，与原始图像前两行，第 2 列到第 3 列的部分对应元素相乘并求和，得到在该位置卷积的结果。以此类推，遵循从上到下，从左到右的方向，直到卷积核与图像右下角（最后 2 行、最后 2 列）对应元素计算，得到最后一个数值。最终卷积运算得到的结果是一个矩阵。

原始图像像素值为 $\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$，卷积核为 $\begin{bmatrix} k_{11} & k_{12} \\ k_{21} & k_{22} \end{bmatrix}$，卷积结果为 $\begin{bmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \end{bmatrix}$。

其中，各输出见式(3-4)～式(3-7)：

$$y_{11} = a_{11}k_{11} + a_{12}k_{12} + a_{21}k_{21} + a_{22}k_{22} \tag{3-4}$$

$$y_{12} = a_{12}k_{11} + a_{13}k_{12} + a_{22}k_{21} + a_{23}k_{22} \tag{3-5}$$

$$y_{21} = a_{21}k_{11} + a_{22}k_{12} + a_{31}k_{21} + a_{32}k_{22} \tag{3-6}$$

$$y_{22} = a_{22}k_{11} + a_{23}k_{12} + a_{32}k_{21} + a_{33}k_{22} \tag{3-7}$$

卷积计算过程如图 3-2 所示，可直观地看出 y_{11} 是由原始图像左上角前两行 2 列元素与卷积核元素对应相乘求和得到的。

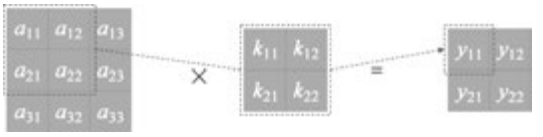

图 3-2 卷积计算过程

2. 图像卷积核与 g 的关系

在深度学习中，卷积核与数学上定义的卷积有所不同。对于二维矩阵形式的函数 $\boldsymbol{f}$ 和 $\boldsymbol{g}$，卷积需要先将函数 $\boldsymbol{g}$ 进行 180°翻转，然后进行对应元素求和，而在 CNN 中，卷积核不需翻转，直接在图像上滑动，进行对应位置求和。

同样使用 3.2.3.1 节的例子，用 $\boldsymbol{g}$ 卷积核对图像的前两行前两列位置（记为 $\boldsymbol{f}$）做卷积运算。见式(3-8)和式(3-9)：

$$\boldsymbol{f}=\begin{bmatrix}a_{11} & a_{12}\\ a_{21} & a_{22}\end{bmatrix} \tag{3-8}$$

$$\boldsymbol{g}=\begin{bmatrix}k_{11} & k_{12}\\ k_{21} & k_{22}\end{bmatrix} \tag{3-9}$$

在数学中的卷积运算 $\boldsymbol{f}*\boldsymbol{g}$ 中，$\boldsymbol{f}$ 的第 1 个元素应与 $\boldsymbol{g}$ 的最后一个元素相乘，$\boldsymbol{f}$ 的第 2 个元素应与 $\boldsymbol{g}$ 的倒数第 2 个元素相乘，即相乘项的坐标之和应相等（本例图形为二维，因此横纵坐标之和分别相等），即卷积运算的结果为 $a_{11}k_{22}+a_{12}k_{21}+a_{21}k_{12}+a_{22}k_{11}$。

而在图像卷积核中，$\boldsymbol{f}$ 与 $\boldsymbol{g}$ 是对应位置相乘的，即图像卷积运算的结果为 $a_{11}k_{11}+a_{12}k_{12}+a_{21}k_{21}+a_{22}k_{22}$。

CNN 中的卷积核参数是通过模型训练自动学习得到的。无论初始的卷积核是否经过翻转，模型都能通过学习找到最适合提取特征的卷积核形态，即使按照数学中卷积的定义使用翻转后的卷积核，模型在训练过程中也会学习到翻转后卷积核的参数，所以可以直接使用未翻转的卷积核进行运算。

3. 在 TensorFlow 中验证卷积计算

以具体数值计算为例，演示卷积运算，并与 TensorFlow 中的结果对比，查看是否一致。假设原始图像元素为 $\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{bmatrix}$，卷积核为 $\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix}$，则左上角位置卷积的结果为 $1\times1+2\times2+4\times3+5\times4=37$，以此类推，全部位置的结果见式(3-10)：

$$\begin{bmatrix}37 & 47\\ 67 & 77\end{bmatrix} \tag{3-10}$$

使用 TensorFlow 的 conv2d 函数验证，代码如下：

```
#chap3/chap3_cnn_val/chap3_val_conv.py
import numpy as np
import tensorflow as tf

x = tf.constant(np.arange(1, 10).reshape([1, 3, 3, 1]), dtype=tf.float32)
filters = tf.constant(np.arange(1, 5).reshape(2, 2, 1, 1), dtype=tf.float32)
print(tf.nn.conv2d(x, filters, [1, 1, 1, 1], "VALID").numpy().reshape([2, 2]))
"""
[[37. 47.]
 [67. 77.]]
"""
```

程序输出结果与计算结果一致。需要说明的是，此处原始图像 x 和卷积核 filters 需要用 reshape 修改形状，因为调用 TensorFlow 的 conv2d 函数需要指定 in_channels(输入通道)和 out_channels(输出通道)，本例为二维图形的计算，因此都指定为 1。

3.2.4 带步长和填充的卷积计算

3.2.3 节展示了卷积运算最基本的原理，为了展示核心的计算逻辑，并未考虑步长和填充，但一般的卷积运算需要考虑步长和填充，本节进行分析。

1. 步长和填充的含义

步长(Stride)和填充(Padding)是卷积运算中重要的参数，它们对卷积的结果和 CNN 的性能有显著影响。

步长是指卷积核在原始图像上滑动时移动的距离。步长直接影响输出的尺寸，步长越大，卷积核扫描图像的速度越快，输出尺寸越小。小步长意味着卷积核在输入信号上有更多重叠，能捕捉到更细微特征，但会增加计算量；大步长则跳过一些位置，减少计算量，但可能会遗漏部分特征。如果将卷积核比作体检时做彩超的仪器，则原始图是身体，输出图像是彩超图。步长为 1 表示医生每次移动仪器的幅度较小，彩超图较为精细；步长较大表示医生每次移动仪器的幅度较大，彩超图相对粗略。

在 3.2.3 节的例子中，用 2×2 卷积核对原始图像左上角 2×2 的位置做卷积运算后，若步长为 1，则卷积核向右平移一个单位，继续对原始图像前两行、第 2～3 列的图像做卷积运算。若步长为 2，则卷积核将向右移动 2 个单位，对原始图像前两行、第 3～4 列的图像做卷积运算。

填充是指在原始图像四周填充上 0。若不进行填充，每次卷积操作后图像尺寸都会缩小，随着卷积层数增加，边缘信息将丢失。填充使卷积核能覆盖到边缘像素，避免边缘信息丢失。

在 3.2.3 节的例子中，以原始图像左上角的元素 a_{11} 为例，在不填充的情况下，从结果表达式看，该元素仅参与一次卷积运算，只影响 y_{11} 的结果，而中间位置的元素 a_{22} 参与了 4 次，影响了所有输出 y_{11}、y_{12}、y_{21}、y_{22} 的结果。在填充后，原始图像所有元素都参与 4 次卷积运算，避免边缘位置元素被弱化。

2. 符号定义

下面定义一组符号，方便下文说明，见表 3-2。

表 3-2 卷积计算符号定义

符　号	含　义
i	输入维度
o	输出维度
k	卷积核维度
s	Stride 的缩写，步长
p	Padding 的缩写，填充的个数
$pu/pd/pl/pr$	分别表示上下左右(up/down/left/right)填充的个数。当需要特别说明具体填充方向时使用此符号

3. 卷积中维度关系

在卷积操作下，维度满足式(3-11)：

$$o=\lfloor (i+2p-k)/s+1 \rfloor \tag{3-11}$$

其中，$\lfloor\ \rfloor$表示向下取整。

从式(3-11)上看，输入维度 i 越高、填充 p 越多，输出图像的维度越大；卷积核维度 k 越高、步长 s 越大，输出图像的维度越小。公式中 $i+2p$ 可理解为填充后的输入维度，卷积操作等价于先填充到 $i+2p$，再用无填充的卷积核计算，例如，若输入 $i=3$，填充 $p=1$，则填充后维度为 $3+2\times1=5$，与直接对填充后 5×5 的图像做无填充卷积的结果一致。

填充可以是非对称的，上式中 $2p$ 表示在两个方向(左右或上下)都填充 p 像素，实际填充时，不同方向的填充个数可不一致，即可将上式的 $2p$ 拆分为 $pl+pr$(左右方向)或 $pu+pd$(垂直方向)。

向下取整是因为$(i+2p-k)$不一定可以被 s 整除。在此情况下，当卷积核滑动到边缘时，若剩余维度小于卷积核维度，则无法进行卷积运算，应将剩余维度图像舍弃，符合实际计算逻辑。

此外，维度关系还可以表示为 $o=\lceil (i+2p-k+1)/s \rceil$，其中$\lceil\ \rceil$表示向上取整。该式与式(3-11)是等价的。

4. TensorFlow 中 Padding 的模式

在 TensorFlow 中，Padding 参数代表了填充的模式，除了传入数值(表示填充的个数)，还可以传入 VALID、SAME 字符串，表示不同的填充策略。

在 VALID 模式下，不做任何填充，即 $p=0$。

在 SAME 模式下，TensorFlow 将自动填充，从而满足 $o=\left\lceil \frac{i}{s} \right\rceil$，即输出尺寸等于输入尺寸除以步长后向上取整。此处的 s 相当于缩放倍数，将输入维度缩放为 s 的整数倍。

得到预期的输出维度 o 后，填充数量应满足 $pl+pr=(o-1)s+k-i$。具体分配到 pl、pr 填充的数量，若上式右侧为偶数，则左右侧各一半，若上式右侧为奇数，则约定右侧比左侧多一个(在垂直方向上，约定下侧比上侧多一个)，见式(3-12)和式(3-13)：

$$pl=\lfloor [(o-1)s+k-i]/2 \rfloor \tag{3-12}$$

$$pr=\lceil [(o-1)s+k-i]/2 \rceil \tag{3-13}$$

3.2.5 池化的计算

池化的计算与卷积类似，同样用卷积核在图像上滑动，区别在于卷积是用卷积核与图像对应元素相乘后相加，池化是取图像对应元素均值(平均池化)或最大值(最大值池化)。池化是用平均特征或典型特征代替区域特征，起到降低图像维度的结果。

例如，对图像 $\begin{bmatrix}1&2&3\\4&5&6\\7&8&9\end{bmatrix}$ 做 2×2 区域的平均池化，每个 2×2 的区域都取均值，结果为

$\begin{bmatrix}3 & 4\\6 & 7\end{bmatrix}$。如果对该图像做最大值池化，则结果为 $\begin{bmatrix}5 & 6\\8 & 9\end{bmatrix}$。

3.3 反卷积的计算

反卷积(Deconvolution)又称转置卷积，是卷积的逆操作，核心作用是通过上采样将小尺寸特征图放大到目标尺寸，被广泛地用于生成模型中。

在生成模型中，上采样和下采样是常见的概念。

下采样是将图像尺寸变小的过程。以卷积操作为例，输入图像后，卷积核在图像上，输出图像尺寸将小于输入，这一过程提取了图像关键特征，实现了图像的压缩。再如最大值池化，从相邻像素中提取最大的像素，丢弃其他不重要的像素，同样实现图像降维。

上采样是将图像维度增大的过程，输入小尺寸的图像后，通过反卷积等运算可生成更大尺寸的图像。上采样可用插值、反卷积等实现。

在生成模型中经常使用反卷积，这是因为隐藏层(特征图)包含最基本的图像特征，一般维度很小，需要使用反卷积升高维度，例如在生成小狗图片的模型中，隐藏层可能只包括眼睛、耳朵等特征的轮廓，需要使用反卷积丰富细节，例如添加上草坪，调整颜色，增加光影等，从而扩大图像维度。

3.3.1 基本思路

反卷积实际上可以通过常规的卷积运算来实现。具体过程可分为两步：首先对输入图像进行内部填充和外部填充以增大其维度，随后将步长设置为1，利用卷积核执行卷积操作。

反卷积的计算演示如图3-3所示。下方为输入的2×2图像，通过在图像中间和周围填充0，使其维度扩充为5×5。之后使用3×3的卷积核对该5×5的图像进行扫描，最终得到上方的输出图像，例如对输入图像左下方3×3的区域用3×3的卷积核进行计算，得到输出图像左下方1×1的区域。输出图像的维度是3×3。

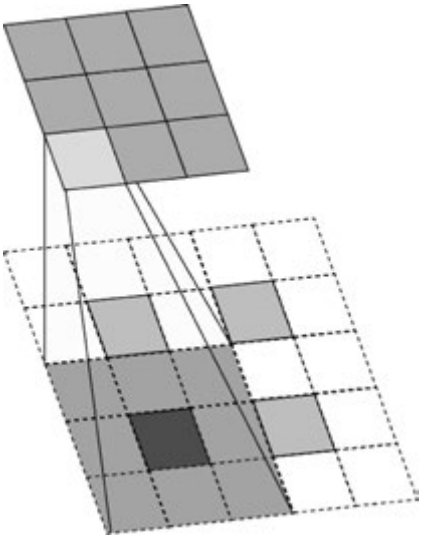

图3-3 反卷积计算演示

由此可见，在反卷积中输入图像先经过填充处理，使其尺寸足够大，再通过卷积操作得到结果。尽管卷积操作对图像起到降维作用，但由于填充使维度提升足够高，所以输出结果维度依然大于输入图像的原始维度，从而实现上采样。

3.3.2 反卷积中的步长和填充

在反卷积中，步长和填充的概念有特定的含义，二者分别决定了图像内部和外部填充0的方式。需要注意的是，这里的步长并非反卷积最后一步卷积运算的步长，最后执行卷积时步长固定为1，而填充的含义与卷积操作的填充类似，用于控制输入图像四周填充0的数量。

卷积操作和反卷积作为逆向过程，可从维度关系上理解。若对反卷积的输出矩阵按照正向卷积的步长和填充进行卷积操作，得到的结果维度应与反卷积的输入矩阵一致，即在卷积操作维度关系中，将输入和输出维度交换，则反卷积的维度应满足 $i=\lfloor(o+2p-k)/s+1\rfloor$。

在TensorFlow中，反卷积的填充模式同样包括SAME、VALID和数值3种方式。

VALID代表上式中 $p=0$，此时 $i=\lfloor(o-k)/s+1\rfloor$。注意，$p=0$ 仅代表输入维度 i 与输出维度 o 满足上式，并不意味着反卷积过程不填充0。为了实现维度升高，反卷积必须进行填充，具体填充方式将在下文计算方式中进行详细说明。

SAME模式的维度关系更直观，当步长为1时输入和输出相同，当步长不为1时，输出维度应为输入维度的 s 倍，即 $o=i\times s$。

由于满足维度关系式的输出维度 o 可能不止一个，因此反卷积需要指定输出的维度大小。在TensorFlow中通过参数output_shape设置，例如当 $p=0,k=2,s=3,i=2$，关系式为 $2=\lfloor(o-2)/3+1\rfloor$，此时 o 为5、6、7均可满足条件，因此output_shape可设置为这3个值中的任意一个。

3.3.3 反卷积计算方法

反卷积的计算过程可分为以下5个步骤：确定输出大小、确定最后一步卷积所需的元素数、元素之间填充0、元素四周填充0、卷积计算。

1. 计算步骤

第1步，确定输出的大小。输出大小output_shape是由用户指定的，需要满足不同填充模式的特定条件。具体条件可参考表3-3。

第2步，确定最后一步卷积操作时需要的元素数。由于反卷积最后一步的卷积运算是按照不填充、步长为1模式计算的，因此可求出填充后的输入图像尺寸应满足 $o+k-1$，由此可知需要填充的总数量为 $o+k-1-i$。

第3步，元素之间填充0。每个相邻元素之间需填充 $(s-1)$ 个0，整个输入图像共有 $(i-1)$ 组相邻元素，因此元素之间的填充数量为 $(s-1)\times(i-1)$ 个。

第4步，元素四周填充0。在不同填充模式下，填充规则有所不同。在VALID和SAME模式下，左侧固定填充数量为 $(k-1)$；在数值模式下，左侧填充数量为 $(k-1-pl)$。在数值模式下，填充数量有可能小于0，此时需对输入矩阵进行裁剪，例如该式值为−1需删除输入矩阵最左侧一列。若裁剪的列数超过了输入图像矩阵的总列数，则矩阵将被整体删除，最终输出为空。右侧填充数量等于总填充数减去左侧填充数和内部填充数。上侧和下

侧的填充规则同理。

第 5 步，进行卷积运算。将卷积核旋转 180°后，按照步长为 1，不填充的模式进行卷积运算。卷积核旋转 180°与卷积核的转置操作有关，下文将详细说明。

反卷积计算的关键步骤见表 3-3。

表 3-3 反卷积计算的关键步骤

关键计算步骤	Padding＝VALID	Padding＝SAME	Padding 为数值
反卷积的输出大小	满足$\lceil (o-k+1)/s \rceil = i$，是不定的	满足$\lceil o/s \rceil = i$，是不定的	满足$\lceil (o+pl+pr-k+1)/s \rceil = i$，是不定的，其中 pl 和 pr 代表在左右侧填充，在上下侧填充同理
最后一步执行卷积操作时需要的元素数	$o+k-1$	$o+k-1$	$o+k-1$
总需要填充个数，记为[1]	$o+k-1-i$	$o+k-1-i$	$o+k-1-i$
输入矩阵元素之间填充 0 个数，记为[2]	$(s-1)\times(i-1)$	$(s-1)\times(i-1)$	$(s-1)\times(i-1)$
输入矩阵左侧填充 0 个数，记为[3]	$k-1$	$k-1$	$k-1-pl$
输入矩阵右侧填充 0 个数	[1]-[2]-[3]	[1]-[2]-[3]	[1]-[2]-[3]
输入矩阵上侧填充 0 个数，记为[4]	$k-1$	$k-1$	$k-1-pu$
输入矩阵下侧填充 0 个数	[1]-[2]-[4]	[1]-[2]-[4]	[1]-[2]-[4]

2. 在 TensorFlow 中验证反卷积计算

假设输入为 $\begin{bmatrix}1&2&3\\4&5&6\\7&8&9\end{bmatrix}$，卷积核为 $\begin{bmatrix}1&0&-1\\0&2&0\\1&0&1\end{bmatrix}$，假设步长为 2，填充模式为 SAME。

第 1 步，指定输出大小，此处指定输出维度为 6×6。

第 2 步，最后一步做卷积需要的维度为 $o+k-1=6+3-1=8$，需要填充 $8-3=5$。

第 3 步，根据步长，在元素之间填充 0。由于步长 $s=2$，因此元素之间填充 $s-1=1$，得到 $\begin{bmatrix}1&0&2&0&3\\0&0&0&0&0\\4&0&5&0&6\\0&0&0&0&0\\7&0&8&0&9\end{bmatrix}$。

第 4 步，根据填充模式，在元素外部填充 0。由于模式为 SAME，左侧填充 $k-1=2$ 个 0。根据第 2 步和第 3 步的结论，共填充 5 个 0，第 3 步已填充 2 个 0，左侧填充了两个 0，因此右侧还需要填充 1 个 0。上下侧同理，上侧填充 2 个 0，下侧填充 1 个 0。得

到 $\begin{bmatrix}0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&1&0&2&0&3&0\\0&0&0&0&0&0&0&0\\0&0&4&0&5&0&6&0\\0&0&0&0&0&0&0&0\\0&0&7&0&8&0&9&0\\0&0&0&0&0&0&0&0\end{bmatrix}$。

第5步，卷积计算。按照不填充、步长为1进行卷积计算，注意将卷积核旋转180°，即卷积核变为 $\begin{bmatrix}1&0&1\\0&2&0\\-1&0&1\end{bmatrix}$。用此卷积核对填充后的矩阵进行卷积运算，最终结果为 $\begin{bmatrix}1&0&1&0&1&0\\0&2&0&4&0&6\\5&0&2&0&2&0\\0&8&0&10&0&12\\11&0&2&0&2&0\\0&14&0&16&0&18\end{bmatrix}$。

在TensorFlow中验证，代码如下：

```
#chap3/chap3_cnn_val/chap3_val_deconv.py
import tensorflow as tf
import numpy as np

a = tf.constant(np.arange(1, 10).reshape([1, 3, 3, 1]), dtype=tf.float32)
kernel = np.array([1, 0, -1, 0, 2, 0, 1, 0, -1]).reshape([3, 3])

kernel = tf.constant(kernel.reshape([3, 3, 1, 1]), dtype=tf.float32)


res = tf.nn.conv2d_transpose(a, kernel, output_shape=[1, 6, 6, 1],
strides=[1, 2, 2, 1])
print(res.numpy().reshape([6, 6]))
"""
[[ 1. 0. 1. 0. 1. 0.]
 [ 0. 2. 0. 4. 0. 6.]
 [ 5. 0. 2. 0. 2. 0.]
 [ 0. 8. 0. 10. 0. 12.]
 [11. 0. 2. 0. 2. 0.]
 [ 0. 14. 0. 16. 0. 18.]]
"""
```

与上文计算结果一致。按照反卷积的逻辑，手动填充图像后用卷积进行计算，查看结果

是否一致，代码如下：

```
#chap3/chap3_cnn_val/chap3_val_deconv.py
import tensorflow as tf
import numpy as np



b = tf.constant(np.array([[0, 0, 0, 0, 0, 0, 0, 0],
                          [0, 0, 0, 0, 0, 0, 0, 0],
                          [0, 0, 1, 0, 2, 0, 3, 0],
                          [0, 0, 0, 0, 0, 0, 0, 0],
                          [0, 0, 4, 0, 5, 0, 6, 0],
                          [0, 0, 0, 0, 0, 0, 0, 0],
                          [0, 0, 7, 0, 8, 0, 9, 0],
                          [0, 0, 0, 0, 0, 0, 0, 0]
                          ]).reshape([1, 8, 8, 1]), dtype=tf.float32)

kernel = np.array([1, 0, -1, 0, 2, 0, 1, 0, -1]).reshape([3, 3])
kernel2 = np.flip(kernel) #注意反卷积核和正向卷积核是 reverse 的关系
kernel2 = tf.constant(kernel2.reshape([3, 3, 1, 1]), dtype=tf.float32)
res = tf.nn.conv2d(b, kernel2, strides=[1, 1, 1, 1], padding="VALID")
print(res.numpy().reshape([6, 6]))
"""
[[ 1. 0. 1. 0. 1. 0.]
 [ 0. 2. 0. 4. 0. 6.]
 [ 5. 0. 2. 0. 2. 0.]
 [ 0. 8. 0. 10. 0. 12.]
 [11. 0. 2. 0. 2. 0.]
 [ 0. 14. 0. 16. 0. 18.]]
"""
```

程序输出结果与上文运算结果也是一致的。

3. 在 TensorFlow 中自定义反卷积运算

根据前文总结的反卷积计算逻辑，本节自定义实现反卷积的逻辑，并与 TensorFlow 中自带的反卷积函数对比，验证计算逻辑。

第 1 步，引入模块并定义输入矩阵和卷积核，代码如下：

```
#chap3/chap3_cnn_val/chap3_val_custom_deconv.py
import math

import tensorflow as tf
import numpy as np

x2 = tf.constant(np.array([[1, 2],
                           [3, 4]]).reshape([1, 2, 2, 1]), dtype=tf.float32)
x3 = tf.constant(np.array([[1, 2, 3],
                           [4, 5, 6],
```

```
                            [7, 8, 9]]).reshape([1, 3, 3, 1]), dtype=tf.float32)
x4 = tf.constant(np.array([[1, 2, 3, 4],
                            [5, 6, 7, 8],
                            [9, 10, 11, 12],
                            [13, 14, 15, 16]]).reshape([1, 4, 4, 1]), dtype=tf.float32)
k2 = tf.constant(np.array([[1, 2],
                            [3, 4]]).reshape(2, 2, 1, 1), dtype=tf.float32)
k3 = tf.constant(np.array([[1, 2, 3],
                            [4, 5, 6],
                            [7, 8, 9]]).reshape(3, 3, 1, 1), dtype=tf.float32)
k4 = tf.constant(np.array([[1, 2, 3, 4],
                            [5, 6, 7, 8],
                            [9, 10, 11, 12],
                            [13, 14, 15, 16]]).reshape(4, 4, 1, 1), dtype=tf.float32)
```

此处定义了 3 种输入矩阵和 3 种卷积核，分别是 2、3、4 维。

第 2 步，引用 TensorFlow 自带的反卷积函数，代码如下：

```
#chap3/chap3_cnn_val/chap3_val_custom_deconv.py
def conv2d_transpose(x, k, s, o, padding):
    i = len(x.numpy()[0])
    k_length = len(k.numpy()[0])
    print(f"i={i},k={k_length},i={o},padding={padding}")
    print("TensorFlow 计算结果")
    if isinstance(o, list):
        output_shape = [1, o[0], o[1], 1]
    else:
        output_shape = [1, o, o, 1]
    return tf.nn.conv2d_transpose(x, k, output_shape, strides=s,
padding=padding).numpy().reshape([output_shape[1], output_shape[2]])
```

第 3 步，定义零元素填充函数 padding_zero，实现在四周、内部填充。注意反卷积有时需要对输入矩阵进行裁剪，因此定义了 cutting_zero 函数，代码如下：

```
#chap3/chap3_cnn_val/chap3_val_custom_deconv.py
def padding_zero(matrix, direction, n):
    """
    在 matrix 的上、下、左、右及内部补 0
    Args:
        matrix: 原始矩阵
        direction: 方向，可选 left、up、right、down、inside
        n: 补 0 的个数
    Returns:
        补 0 后的矩阵
    Examples:
        >>> a = tf.constant(np.array(
        ...                          [[1, 2, 3],
        ...                           [4, 5, 6],
        ...                           [7, 8, 9]]).reshape([1, 3, 3, 1]), dtype=tf.float32)
```

```
        >>> res = padding_zero(a, "up", 2).numpy()
        >>> res.reshape([len(res[0]), len(res[0][0])])
        [[0. 0. 0.]
         [0. 0. 0.]
         [1. 2. 3.]
         [4. 5. 6.]
         [7. 8. 9.]]
        >>> res = padding_zero(a, "inside", 2).numpy()
        >>> res.reshape([len(res[0]), len(res[0][0])])
        [[1. 0. 0. 2. 0. 0. 3.]
         [0. 0. 0. 0. 0. 0. 0.]
         [0. 0. 0. 0. 0. 0. 0.]
         [4. 0. 0. 5. 0. 0. 6.]
         [0. 0. 0. 0. 0. 0. 0.]
         [0. 0. 0. 0. 0. 0. 0.]
         [7. 0. 0. 8. 0. 0. 9.]]
    """
    if n < 0:
        assert direction != "inside"  #inside 不缩减 0
        return cutting_zero(matrix, direction, -n)
    a = matrix.numpy()                  #维度为 1 *x *y *1
    row_length = matrix.shape[1]
    column_length = matrix.shape[2]
    a = a.reshape([row_length, column_length])
    res = a
    if direction == "up":
        if row_length > 0:
            res = np.pad(a, [[n, 0], [0, 0]], constant_values=0)
            row_length = row_length + n
    elif direction == "down":
        if row_length > 0:
            res = np.pad(a, [[0, n], [0, 0]], constant_values=0)
            row_length = row_length + n
    elif direction == "left":
        if column_length > 0:
            res = np.pad(a, [[0, 0], [n, 0]], constant_values=0)
            column_length = column_length + n
    elif direction == "right":
        if column_length > 0:
            res = np.pad(a, [[0, 0], [0, n]], constant_values=0)
            column_length = column_length + n
    else:
        row_length = (row_length - 1) *n + row_length
        #需要补 x-1 个位置,每个位置补 n 个 0
        column_length = (column_length - 1) *n + column_length
        #需要补 y-1 个位置,每个位置补 n 个 0
        res = np.zeros([row_length, column_length])
        x_position = np.arange(0, len(res), n + 1)      #原矩阵在新矩阵中的位置
```

```
        y_position = np.arange(0, len(res[0]), n + 1)  #原矩阵在新矩阵中的位置
        for i in range(len(x_position)):
            for j in range(len(y_position)):
                res[x_position[i]][y_position[j]] = a[i][j]
    res = tf.constant(np.array(res).reshape([1, row_length, column_length, 1]),
dtype=tf.float32)
    return res


def cutting_zero(matrix, direction, n):
    """
    在 matrix 的上、下、左、右缩减
    Args:
        matrix: 原始矩阵
        direction: 方向,可选 left、up、right、down
        n: 缩减行/列数
    Returns:
        缩减后的矩阵
    Examples:
        >>> a = tf.constant(np.array(
        ...                          [[1, 2, 3],
        ...                           [4, 5, 6],
        ...                           [7, 8, 9]]).reshape([1, 3, 3, 1]), dtype=tf.float32)
        >>> res = cutting_zero(a, "up", 1).numpy()
        >>> res.reshape([len(res[0]), len(res[0][0])])
        [[4. 5. 6.]
         [7. 8. 9.]]
    """
    a = matrix.numpy()  #维度为 1 *x *y *1
    row_length = len(a[0])
    column_length = len(a[0][0])
    a = a.reshape([row_length, column_length])
    if direction == "up":
        res = a[n:, :]
        row_length = max(row_length - n, 0)
    elif direction == "down":
        res = a[:-n, :]
        row_length = max(row_length - n, 0)
    elif direction == "left":
        res = a[:, n:]
        column_length = max(column_length - n, 0)
    else:
        res = a[:, :-n]
        column_length = max(column_length - n, 0)
    res = tf.constant(np.array(res).reshape([1, row_length, column_length, 1]),
dtype=tf.float32)
    return res
```

第 4 步，实现自定义的反卷积运算函数 conv2d_transpose2，代码如下：

```
#chap3/chap3_cnn_val/chap3_val_custom_deconv.py
def conv2d_transpose2(x, filters, s, o, padding):
    print("手动计算结果")
    i = len(x.numpy()[0])
    k = len(filters.numpy()[0])
    if isinstance(o, list):
        output_shape = [1, o[0], o[1], 1]
    else:
        output_shape = [1, o, o, 1]
    #检查传入的 i 是否合适
    if padding == "VALID":
        assert math.ceil((o - k + 1) / s) == i
    elif padding == "SAME":
        assert o == i * s
    else:
        assert i == math.ceil((output_shape[1] + sum(padding[1]) - k + 1) / s)
        assert i == math.ceil((output_shape[2] + sum(padding[2]) - k + 1) / s)
    #在 x 中间补 0 的个数
    num_of_zero_inside = s - 1
    if padding in ["SAME", "VALID"]:
        #做卷积需要的维度数
        i_padding = o + k - 1
        #左、上补 0 的个数:k-1
        num_of_zero_left = k - 1
        num_of_zero_up = k - 1
        #右、下补 0 的个数
        num_of_zero_right = i_padding - i - num_of_zero_inside * (i - 1) - num_of_
zero_left
        num_of_zero_down = i_padding - i - num_of_zero_inside * (i - 1) - num_of_
zero_up
    else:
        #做卷积需要的维度数
        i_padding_row = output_shape[1] + k - 1
        i_padding_column = output_shape[2] + k - 1
        #左、上补 0 的个数:k-1-pl 或 pu
        pad_top = padding[1][0]
        pad_left = padding[2][0]
        num_of_zero_left = k - 1 - pad_left
        num_of_zero_up = k - 1 - pad_top
        #右、下补 0 的个数
        num_of_zero_right = i_padding_column - i - num_of_zero_inside * (i - 1) -
num_of_zero_left
        num_of_zero_down = i_padding_row - i - num_of_zero_inside * (i - 1) - num_
of_zero_up

    x_new = padding_zero(x, "inside", num_of_zero_inside)
    x_new = padding_zero(x_new, "left", num_of_zero_left)
```

```
    x_new = padding_zero(x_new, "up", num_of_zero_up)
    x_new = padding_zero(x_new, "right", num_of_zero_right)
    x_new = padding_zero(x_new, "down", num_of_zero_down)
    #对卷积核做 reverse
    filters_numpy = filters.numpy()  #维度为 n*n*1*1
    dim_filters = len(filters_numpy)
    filters_reverse = np.flip(filters_numpy.reshape([dim_filters,
dim_filters]))  #翻转
    filters_reverse = tf.constant(filters_reverse.reshape([dim_filters,
dim_filters, 1, 1]), dtype=tf.float32)
    #此处 s=1,p=0,判断做卷积时 i-k+1 应大于 0,避免极端情况例如 x_new 为空
    if x_new.shape[1] - filters_reverse.shape[0] + 1 > 0 and x_new.shape[2] -
filters_reverse.shape[1] + 1 > 0:
        res = tf.nn.conv2d(x_new, filters_reverse, strides=1, padding="VALID")
        return res.numpy().reshape([output_shape[1], output_shape[2]])
    else:
        return []
```

第 5 步,使用各种参数验证自定义的反卷积函数与 TensorFlow 自带的反卷积函数的计算结果是否一致,代码如下:

```
#chap3/chap3_cnn_val/chap3_val_custom_deconv.py
"""
VALID 模式
"""
print("VALID 模式")
print(conv2d_transpose(x3, k2, 1, 4, "VALID"))
print(conv2d_transpose2(x3, k2, 1, 4, "VALID"))

print(conv2d_transpose(x4, k2, 1, 5, "VALID"))
print(conv2d_transpose2(x4, k2, 1, 5, "VALID"))

print(conv2d_transpose(x2, k2, 2, 4, "VALID"))
print(conv2d_transpose2(x2, k2, 2, 4, "VALID"))

print(conv2d_transpose(x2, k2, 2, 5, "VALID"))
print(conv2d_transpose2(x2, k2, 2, 5, "VALID"))

print(conv2d_transpose(x3, k2, 2, 6, "VALID"))
print(conv2d_transpose2(x3, k2, 2, 6, "VALID"))

print(conv2d_transpose(x2, k2, 3, 6, "VALID"))
print(conv2d_transpose2(x2, k2, 3, 6, "VALID"))

print(conv2d_transpose(x3, k2, 4, 10, "VALID"))
print(conv2d_transpose2(x3, k2, 4, 10, "VALID"))

print(conv2d_transpose(x3, k2, 4, 11, "VALID"))
```

```
print(conv2d_transpose2(x3, k2, 4, 11, "VALID"))

print(conv2d_transpose(x3, k2, 4, 12, "VALID"))
print(conv2d_transpose2(x3, k2, 4, 12, "VALID"))

print(conv2d_transpose(x3, k2, 4, 13, "VALID"))
print(conv2d_transpose2(x3, k2, 4, 13, "VALID"))

print(conv2d_transpose(x2, k3, 8, 17, "VALID"))
print(conv2d_transpose2(x2, k3, 8, 17, "VALID"))

"""
SAME 模式
"""
print("SAME 模式")
print(conv2d_transpose(x3, k2, 1, 3, "SAME"))
print(conv2d_transpose2(x3, k2, 1, 3, "SAME"))

print(conv2d_transpose(x3, k2, 2, 6, "SAME"))
print(conv2d_transpose2(x3, k2, 2, 6, "SAME"))

print(conv2d_transpose(x3, k2, 3, 9, "SAME"))
print(conv2d_transpose2(x3, k2, 3, 9, "SAME"))

print(conv2d_transpose(x3, k2, 4, 12, "SAME"))
print(conv2d_transpose2(x3, k2, 4, 12, "SAME"))

print(conv2d_transpose(x3, k3, 8, 24, "SAME"))
print(conv2d_transpose2(x3, k3, 8, 24, "SAME"))

"""
数值模式
"""

pad_top = 8
pad_bottom = 0
pad_left = 0
pad_right = 0
print(conv2d_transpose(x2, k3, 3, [0, 6], [[0, 0], [pad_top, pad_bottom],
                                    [pad_left, pad_right], [0, 0]]))
print(conv2d_transpose2(x2, k3, 3, [0, 6], [[0, 0], [pad_top, pad_bottom],
                                     [pad_left, pad_right], [0, 0]]))

pad_top = 1
pad_bottom = 2
pad_left = 3
pad_right = 2
print(conv2d_transpose(x2, k3, 3, [3, 1], [[0, 0], [pad_top, pad_bottom],
                                    [pad_left, pad_right], [0, 0]]))
print(conv2d_transpose2(x2, k3, 3, [3, 1], [[0, 0], [pad_top, pad_bottom],
                                     [pad_left, pad_right], [0, 0]]))
```

```
pad_top = 1
pad_bottom = 5
pad_left = 3
pad_right = 4
#(i+k-1)/s=2
print(conv2d_transpose(x2, k4, 3, [1, 1], [[0, 0], [pad_top, pad_bottom],
                                    [pad_left, pad_right], [0, 0]]))
print(conv2d_transpose2(x2, k4, 3, [1, 1], [[0, 0], [pad_top, pad_bottom],
                                     [pad_left, pad_right], [0, 0]]))
```

代码对比了各种模式、各种参数下 TensorFlow 的结果与自己手动计算的结果，例如其中一处输出的结果，代码如下：

```
#chap3/chap3_cnn_val/chap3_val_custom_deconv.py
"""
SAME 模式
i=3,k=2,i=3,padding=SAME
TensorFlow 计算结果
[[ 1.  4.  7.]
 [ 7. 23. 33.]
 [19. 53. 63.]]
手动计算结果
[[ 1.  4.  7.]
 [ 7. 23. 33.]
 [19. 53. 63.]]
"""
```

经过检查，所有情况下的自定义函数的计算结果与 TensorFlow 自带函数的计算结果均一致。

3.3.4 延伸讨论

本节从棋盘效应等方面对反卷积进一步地进行讨论。

1. 棋盘效应

反卷积中可能出现一种常见问题——棋盘效应，即生成的图像上呈现深浅不一的棋盘状或格子状图案。出现这种现象的原因是，当反卷积中卷积核维度与步长无法整除时，图像的不同位置参与计算的原始元素数量存在差异，从而导致输出明暗不均。

假设卷积核为 3，步长为 2，填充模式为不填充(不对四周填充)。原始输入图像为三维，按照反卷积的填充规则，元素之间填充 1 个 0，维度将填充为 5 维，见式(3-14)：

$$\begin{bmatrix} a_{11} & 0 & a_{12} & 0 & a_{13} \\ 0 & 0 & 0 & 0 & 0 \\ a_{21} & 0 & a_{22} & 0 & a_{23} \\ 0 & 0 & 0 & 0 & 0 \\ a_{31} & 0 & a_{32} & 0 & a_{33} \end{bmatrix} \tag{3-14}$$

卷积核旋转 180°后，见式(3-15)：

$$\begin{bmatrix} k_{33} & k_{32} & k_{31} \\ k_{23} & k_{22} & k_{21} \\ k_{13} & k_{12} & k_{11} \end{bmatrix} \tag{3-15}$$

用旋转后的卷积核对填充后的图像进行卷积操作，得到 3×3 维度的结果。

考虑输出的第 1 行第 1 列的结果，应为 $y_{11}=a_{11}k_{33}+a_{12}k_{31}+a_{21}k_{13}+a_{22}k_{11}$，由原始图像中的 4 个元素构成，而输出第 2 行第 2 列的结果 $y_{22}=a_{22}k_{22}$，仅由原始图像的 1 个元素参与计算。这种参与元素数量的不均衡导致输出图像明暗交替，呈现棋盘的形状。

避免棋盘效应主要有 3 种方法，第 1 种是合理选择卷积核与步长，使二者可以整除；第 2 种是调整卷积核权重，通过训练使卷积核部分位置数值较大，部分数值较小，平衡不同位置的输出亮度，但这种方式需要额外的训练成本，效果难以控制；第 3 种是用插值替代填充零操作。反卷积中原始的零填充将导致部分位置计算时缺失原始元素，若在填充 0 的位置用插值补充合理数值，则可使每个输出位置参与计算的原始元素数量一致，使输出图像深浅一致。

2. 反卷积为什么也叫转置卷积

图 3-2 展示了卷积计算示例，其中卷积核是 2×2 的形式，输入图像是 3×3 的形式，步长为 1。卷积操作可转换为矩阵操作，其中卷积核转变成了一个 4×9 的稀疏矩阵(稀疏是指其中很多元素是 0)，将输入图像展开，使其成一列形式 9×1，得到输出图像为 4×1 的形式，重新整理成 2×2 形式即为输出的图像，如图 3-4 所示。

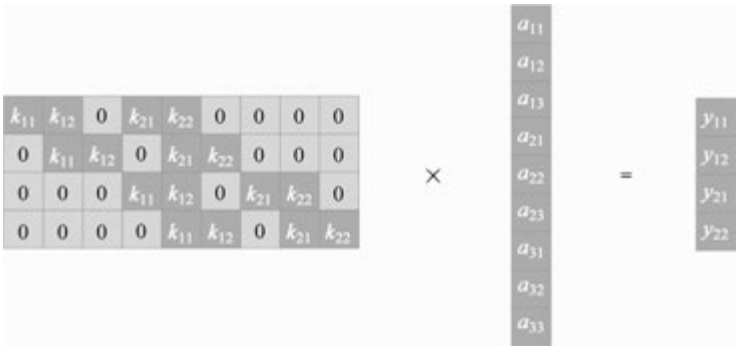

图 3-4 反卷积的矩阵表示形式

图 3-4 所示的过程可表示为 $\boldsymbol{Y}=\boldsymbol{C}\boldsymbol{X}$，将左右两侧同时左乘 $\boldsymbol{C}^{\mathrm{T}}$，得到 $\boldsymbol{C}^{\mathrm{T}}\boldsymbol{Y}=\boldsymbol{C}^{\mathrm{T}}\boldsymbol{C}\boldsymbol{X}$。即稀疏矩阵 $\boldsymbol{C}$ 的转置(维度为 9×4)乘输出的图像 $\boldsymbol{Y}$(维度为 4×1)，得到了结果(维度为 9×1)，从而实现了从 4×1 图像升维到 9×1 图像。注意反卷积不是可逆的，因为右式中 $\boldsymbol{C}^{\mathrm{T}}\boldsymbol{C}$ 不一定等于单位矩阵，即 $\boldsymbol{C}$ 不一定是正交矩阵。等式右边只是维度与输入的 $\boldsymbol{X}$ 相同而已。

因此反卷积又叫作转置卷积，因为反卷积的操作类似于对卷积核对应的稀疏矩阵 $\boldsymbol{C}$ 求转置。

3.4 卷积梯度的计算

根据 3.3.4 节的分析，从本质上看，卷积运算可以视为一种特殊的全连接计算，区别是其权重矩阵具有稀疏性，即大部分位置的权重固定为 0，仅在与卷积核对应的位置存在非零权重，因此卷积操作的梯度计算与全连接神经网络的梯度计算原理一致，它们都是基于链式法则经推导得出的。

在全连接网络中，梯度计算需考虑损失函数对每个权重参数的偏导数，通过反向传播依次计算输出层到输入层的梯度。对于卷积操作而言，尽管权重参数在空间上具有共享性（同一卷积核被用于输入图像的多个位置），但梯度计算仍遵循链式法则，只是梯度的传播路径被扩展到了输入图像的多个位置。

3.4.1 卷积结果对输入的梯度

仍对上文卷积计算中的例子进行分析，卷积结果本质上是输入元素 a 与卷积核元素 k 相乘求和的结果，因此卷积对于输入元素 a 的梯度应为对应的卷积核元素 k，例如，a_{11} 参与了 y_{11} 的计算，系数为 k_{11}，因此梯度为 k_{11}。以此类推，计算 y_{11} 对所有输入的梯度，见表 3-4。

表 3-4 卷积对输入的梯度

输出对象梯度	输出对象第 1 列梯度	输出对象第 2 列梯度	输出对象第 3 列梯度
输出对象第 1 行梯度	$\frac{\partial y_{11}}{\partial a_{11}}=k_{11}=1$	$\frac{\partial y_{11}}{\partial a_{12}}=k_{12}=2$	$\frac{\partial y_{11}}{\partial a_{13}}=0$
输出对象第 2 行梯度	$\frac{\partial y_{11}}{\partial a_{21}}=k_{21}=3$	$\frac{\partial y_{11}}{\partial a_{22}}=k_{22}=4$	$\frac{\partial y_{11}}{\partial a_{23}}=0$
输出对象第 3 行梯度	$\frac{\partial y_{11}}{\partial a_{31}}=0$	$\frac{\partial y_{11}}{\partial a_{32}}=0$	$\frac{\partial y_{11}}{\partial a_{33}}=0$

在 TensorFlow 中验证，代码如下：

```
#chap3/chap3_cnn_val/chap3_val_conv_gradient.py
import numpy as np
import tensorflow as tf


@tf.function
def compute_gradient(x, filters):
    with tf.GradientTape() as tape:
        tape.watch(x)  #监视 x
        y = tf.nn.conv2d(x, filters, [1, 1, 1, 1], "VALID")
        return tape.jacobian(y, x)  #计算 y 相对于 x 的梯度


x = tf.constant(np.arange(1, 10).reshape([1, 3, 3, 1]), dtype=tf.float32)
```

```
filters = tf.constant(np.arange(1, 5).reshape(2, 2, 1, 1), dtype=tf.float32)
print(compute_gradient(x, filters).numpy().reshape([4, 3, 3]))
"""
[[[1. 2. 0.]
  [3. 4. 0.]
  [0. 0. 0.]]

 [[0. 1. 2.]
  [0. 3. 4.]
  [0. 0. 0.]]

 [[0. 0. 0.]
  [1. 2. 0.]
  [3. 4. 0.]]

 [[0. 0. 0.]
  [0. 1. 2.]
  [0. 3. 4.]]]
"""
```

程序输出的结果与上文的计算结论一致。

注意 TensorFlow 中计算每个分量的梯度(例如 y_{11} 对 a_{11} 的梯度),应该使用 jacobian 函数,而不能使用 gradient 函数。这是因为 gradient 函数返回的是所有输入梯度的总和,无法单独获取每个分量的梯度信息。

3.4.2 卷积结果对卷积核的梯度

与 3.4.1 节类似,卷积结果对卷积核的梯度是对应输入图像 a,例如 k_{11} 参与了 y_{11} 的计算,系数为 a_{11},因此梯度为 a_{11}。计算所有输出 y 对于 k_{11} 的梯度,见表 3-5。

表 3-5 卷积对卷积核的梯度

输出对卷积梯度	输出第 1 列对卷积梯度	输出第 2 列对卷积梯度
输出第 1 行对卷积梯度	$\frac{\partial y_{11}}{\partial k_{11}}=a_{11}=1$	$\frac{\partial y_{12}}{\partial k_{11}}=a_{12}=2$
输出第 2 行对卷积梯度	$\frac{\partial y_{21}}{\partial k_{11}}=a_{21}=4$	$\frac{\partial y_{22}}{\partial k_{11}}=a_{22}=5$

在 TensorFlow 中验证,代码如下:

```
#chap3/chap3_cnn_val/chap3_val_conv_gradient.py
import numpy as np
import tensorflow as tf


@tf.function
def compute_gradient(x, filters):
    with tf.GradientTape() as tape:
        tape.watch(filters)  #监视 filters
```

```
        y = tf.nn.conv2d(x, filters, [1, 1, 1, 1], "VALID")
        return tape.jacobian(y, filters)


x = tf.constant(np.arange(1, 10).reshape([1, 3, 3, 1]), dtype=tf.float32)
filters = tf.constant(np.arange(1, 5).reshape(2, 2, 1, 1), dtype=tf.float32)
print(compute_gradient(x, filters).numpy().reshape([4, 2, 2]))
"""
[[[1. 2.]
  [4. 5.]]

 [[2. 3.]
  [5. 6.]]

 [[4. 5.]
  [7. 8.]]

 [[5. 6.]
  [8. 9.]]]
"""
```

程序输出的结果与上文计算结论一致。

3.5 反卷积梯度的计算

反卷积的计算相当于对输入图像先进行填充再进行卷积，因此反卷积的梯度等价于对填充后的输入矩阵进行卷积运算所产生的梯度。

3.5.1 反卷积结果对输入的梯度

仍对 3.4 节的例子进行分析，假设 Padding 为 VALID 模式(不填充)，步长 $s=1$(内部元素不填充)，$k=2$，填充 0 后，得到式(3-16)：

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & a_{11} & a_{12} & a_{13} & 0 \\ 0 & a_{21} & a_{22} & a_{23} & 0 \\ 0 & a_{31} & a_{32} & a_{33} & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{3-16}$$

翻转卷积核，得到 $\begin{bmatrix} k_{11} & k_{12} \\ k_{21} & k_{22} \end{bmatrix}$。

对二者做卷积运算，得到 4×4 的输出。

在计算梯度时，输出维度是 4×4，输入维度是 3×3，每个输出对每个输入都计算梯度，因此计算梯度的矩阵(Jacobian 矩阵)维度是 $4\times4\times3\times3$。

以 y_{23} 对 a_{22} 的梯度为例，由于 $y_{32}=a_{21}k_{22}+a_{22}k_{21}+a_{31}k_{12}+a_{32}k_{11}$，$a_{22}$ 前的系数为 k_{21}，因此梯度为 $k_{21}=3$。

在 TensorFlow 中验证，代码如下：

```
#chap3/chap3_cnn_val/chap3_val_conv_gradient.py
import numpy as np
import tensorflow as tf


def conv2d_transpose(x, filters):
    return tf.nn.conv2d_transpose(x, filters, [1, 4, 4, 1], strides=1, padding=
"VALID")


@tf.function
def compute_conv2d_transpose_i_gradient(x, filters):
    with tf.GradientTape() as tape:
        tape.watch(x)
        y = conv2d_transpose(x, filters)
        return tape.jacobian(y, x)


x = tf.constant(np.arange(1, 10).reshape([1, 3, 3, 1]), dtype=tf.float32)
filters = tf.constant(np.arange(1, 5).reshape(2, 2, 1, 1), dtype=tf.float32)
print(compute_conv2d_transpose_i_gradient(x, filters).numpy().reshape([4, 4,
3, 3])[2][1][1][1])  #注意下标是从 0 开始的,[2][1]代表 y32,[1][1]代表 a22
"""
3
"""
```

程序输出的结果与上文的计算结论一致。

3.5.2 反卷积结果对卷积核的梯度

与 3.5.1 节类似，反卷积结果对卷积核的梯度等价于对填充后的输入矩阵做卷积时的梯度，例如计算 y_{23} 对 k_{21} 的梯度，由于在 y_{23} 的表达式中 k_{21} 前的系数为 a_{22}，因此梯度为 $a_{22}=5$。

在 TensorFlow 中验证，代码如下：

```
#chap3/chap3_cnn_val/chap3_val_conv_gradient.py
import numpy as np
import tensorflow as tf


def conv2d_transpose(x, filters):
    return tf.nn.conv2d_transpose(x, filters, [1, 4, 4, 1], strides=1, padding="VALID")
```

```
@tf.function
def compute_conv2d_transpose_k_gradient(x, filters):
    with tf.GradientTape() as tape:
        tape.watch(filters)
        y = conv2d_transpose(x, filters)
        return tape.jacobian(y, filters)


x = tf.constant(np.arange(1, 10).reshape([1, 3, 3, 1]), dtype=tf.float32)
filters = tf.constant(np.arange(1, 5).reshape(2, 2, 1, 1), dtype=tf.float32)
print(compute_conv2d_transpose_k_gradient(x, filters).numpy().reshape([4, 4,
2, 2])[2][1][1][0])
"""
5
"""
```

程序输出的结果与上文的计算结论一致。

3.5.3 反卷积等价于误差反向传播

在 TensorFlow 中，反卷积函数的源代码实际上调用了误差反向传播函数 conv2d_backprop_input，这意味着反卷积等价于将输入作为卷积下层的误差进行反向传播，本节对此进行分析。

首先定义符号，见表 3-6。注意符号与多层感知机一章的误差反向传播相同，尤其需要注意符号 δ，这个符号代表最终的误差对当前层输出（激活前）的梯度，即从上一层传播回的误差。

表 3-6 卷积反向传播符号的含义

符号	含 义
a_{ij}	卷积层的输入，下标表示第 i 行第 j 列
y_{ij}	卷积层的输出（激活前），即卷积运算结果，下标表示第 i 行第 j 列
L	损失函数
δ	最终误差对当前层输出（激活前）的梯度，代表从上一层传播回的误差，即最终损失对 y 的梯度

根据梯度的链式法则，损失函数对输入的梯度等于上一层传播到本层所有输出的误差乘卷积结果对输入的梯度之和。

以计算损失函数 L 对 a_{12} 的梯度为例，a_{12} 参与了 y_{11} 和 y_{12} 两个输出，见式(3-17)和式(3-18)：

$$y_{11}=a_{11}k_{11}+a_{12}k_{12}+a_{21}k_{21}+a_{22}k_{22} \tag{3-17}$$

$$y_{12}=a_{12}k_{11}+a_{13}k_{12}+a_{22}k_{21}+a_{23}k_{22} \tag{3-18}$$

由于这两个输出都会参与损失函数的计算，因此其梯度包含从 y_{11} 和 y_{12} 两个输出传播回的误差，见式(3-19)：

$$\frac{\partial L}{\partial a_{12}}=\frac{\partial L}{\partial y_{11}}\frac{\partial y_{11}}{\partial a_{12}}+\frac{\partial L}{\partial y_{12}}\frac{\partial y_{12}}{\partial a_{12}}=\delta_{11}k_{12}+\delta_{12}k_{11} \tag{3-19}$$

以此类推，可计算损失函数对所有输入的梯度，如式(3-20)～式(3-28)所示。

$$\frac{\partial L}{\partial a_{11}}=\delta_{11}k_{11} \tag{3-20}$$

$$\frac{\partial L}{\partial a_{12}}=\delta_{11}k_{12}+\delta_{12}k_{11} \tag{3-21}$$

$$\frac{\partial L}{\partial a_{13}}=\delta_{11}k_{12}+\delta_{12}k_{12} \tag{3-22}$$

$$\frac{\partial L}{\partial a_{21}}=\delta_{11}k_{21}+\delta_{21}k_{11} \tag{3-23}$$

$$\frac{\partial L}{\partial a_{22}}=\delta_{11}k_{22}+\delta_{12}k_{21}+\delta_{21}k_{12}+\delta_{22}k_{11} \tag{3-24}$$

$$\frac{\partial L}{\partial a_{23}}=\delta_{12}k_{22}+\delta_{22}k_{12} \tag{3-25}$$

$$\frac{\partial L}{\partial a_{31}}=\delta_{21}k_{22} \tag{3-26}$$

$$\frac{\partial L}{\partial a_{32}}=\delta_{21}k_{22}+\delta_{22}k_{21} \tag{3-27}$$

$$\frac{\partial L}{\partial a_{33}}=\delta_{22}k_{22} \tag{3-28}$$

这种计算结果等价于求卷积的过程，如图 3-5 所示。

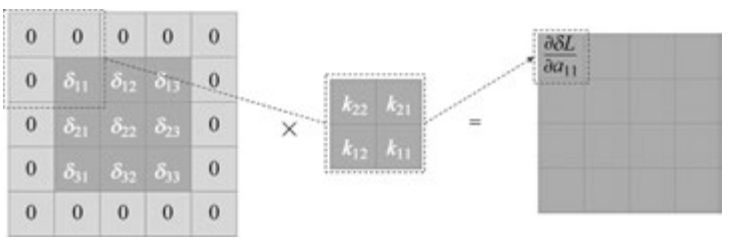

图 3-5　反卷积等价于误差反向传播演示

这正是反卷积的计算过程，因此反卷积可使用卷积的误差反向传播进行计算。

在 TensorFlow 中验证，代码如下：

```
#chap3/chap3_cnn_val/chap3_val_conv_gradient.py
def conv2d_transpose(x, filters):
    return tf.nn.conv2d_transpose(x, filters, [1, 4, 4, 1], strides=1, padding=
"VALID")


x = tf.constant(np.arange(1, 10).reshape([1, 3, 3, 1]), dtype=tf.float32)
filters = tf.constant(np.arange(1, 5).reshape(2, 2, 1, 1), dtype=tf.float32)
print("反卷积结果:")
```

```
print(conv2d_transpose(x, filters).numpy().reshape([4, 4]))
#卷积反向传播
x = tf.constant(np.array([[0, 0, 0, 0, 0],
                          [0, 1, 2, 3, 0],
                          [0, 4, 5, 6, 0],
                          [0, 7, 8, 9, 0],
                          [0, 0, 0, 0, 0]]).reshape([1, 5, 5, 1]),
dtype=tf.float32)
filters = tf.constant(np.array([[4, 3],
                                [2, 1]]).reshape(2, 2, 1, 1), dtype=tf.float32)
print("卷积反向传播结果:")
print(tf.nn.conv2d(x, filters, [1, 1, 1, 1], "VALID").numpy().reshape(4, 4))
"""
反卷积结果:
[[ 1.  4.  7.  6.]
 [ 7. 23. 33. 24.]
 [19. 53. 63. 42.]
 [21. 52. 59. 36.]]
卷积反向传播结果:
[[ 1.  4.  7.  6.]
 [ 7. 23. 33. 24.]
 [19. 53. 63. 42.]
 [21. 52. 59. 36.]]
"""
```

从程序可知,反卷积函数与误差反向传播的结果一致。

3.6 模型的优势

CNN 的提出具有重要意义,其中的卷积层已几乎成为现代深度学习模型的标配模块。CNN 主要具备以下优势。

1. 卷积核具备强大的特征提取能力

CNN 通过卷积层和池化层的组合,可以从数据中学习到不同层次的特征。以图像识别任务为例,卷积核在图像上滑动时,可提取局部的边缘、纹理等低级特征;随着网络层数的增加,后续层将这些低级特征组合成更高级、更抽象的特征,如物体的形状、类别等。这种层次化的特征学习能力,使卷积神经网络能够有效地处理图像、语音等具有复杂结构的数据。

2. 减少参数数量

传统的神经网络在处理图像等数据时,由于输入数据维度高,参数数量过大,所以容易出现过拟合问题。CNN 引入了权值共享和局部连接机制,同一卷积核的参数在输入图像的不同位置共享参数,大幅减少了模型的参数数量。这不仅降低了计算开销,还提高了模型的泛化能力,使模型在有限的数据上也能有较好的表现。

3. 高效的计算性能

由于卷积神经网络引入了权值共享和局部连接,这使模型的并行计算成为可能,所以适

合在 GPU 等并行计算设备上进行加速计算。卷积核在不同图像区域的计算相互独立，可同时对多个位置进行计算，这显著地提高了模型的训练和预测速度，满足在实际应用中对实时性和效率的要求。

3.7 模型的不足

CNN 模型的不足主要体现在以下方面。

1. 对标注数据依赖较大

CNN 的训练通常需要大量高质量的标注数据，标注数据的质量和数量直接影响模型的性能。在实际应用中，获取高质量的标注数据往往需要耗费大量的人力和时间，并且标注的准确性也可能存在误差。如果数据量不足，则模型难以学习到足够的特征，从而导致性能下降。

2. 可解释性差

卷积神经网络内部的计算过程较复杂，难以直观地解释模型的决策过程和依据。这降低了卷积神经网络的可信度，同时限制了其在医疗诊断、金融风险评估等对可解释性要求较高的场景应用。

3.8 案例 1：卷积运算识别简单形状

本案例将使用卷积运算识别简单形状，证明 CNN 中的卷积核可识别图像中的特定形状特征。

绘制正方形和直角三角形的图形，并使用直线卷积核对图形进行卷积运算，查看在正方形和直角三角形横线处卷积结果较大，非横线处卷积结果较小，从而判断卷积核可识别相应形状。

引入相关模块，生成空白图像，代码如下：

```
#chap3/chap3.1_line_kernel.py
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf

plt.rcParams['font.family'] = ['SimSun', 'Times New Roman']
#生成图像尺寸
image_size = 100
image = np.zeros((image_size, image_size), dtype=np.float32)
```

绘制正方形，代码如下：

```
#chap3/chap3.1_line_kernel.py
#绘制正方形
```

```
square_size = 20
square_x = 20
square_y = 40
image[square_y, square_x:square_x + square_size] = 1
image[square_y + square_size - 1, square_x:square_x + square_size] = 1
image[square_y:square_y + square_size, square_x] = 1
image[square_y:square_y + square_size, square_x + square_size - 1] = 1
```

绘制直角三角形，注意斜边使用循环绘制，代码如下：

```
#chap3/chap3.1_line_kernel.py
#绘制直角三角形
triangle_size = 20
triangle_x = 70
triangle_y = 40
image[triangle_y:triangle_y + triangle_size, triangle_x] = 1
image[triangle_y + triangle_size, triangle_x:triangle_x + triangle_size] = 1
for i in range(triangle_size):
    image[triangle_y + i, triangle_x + i] = 1
```

在 TensorFlow 中进行卷积运算时需要传入四维张量(Tensor)，4 个维度分别是训练批次(batch_size)、高度(height)、宽度(width)、通道(channels)。由于本例中只需单个样本，因此第一维仅有一个元素；此外，由于本例中图像为黑白色，通道数为 1，因此最后一维仅有一个元素，代码如下：

```
#chap3/chap3.1_line_kernel.py
#调整图像维度以适配 TensorFlow
image = image.reshape(1, image.shape[0], image.shape[1], 1)
```

将卷积核定义为直线，使其专门识别横线，代码如下：

```
#chap3/chap3.1_line_kernel.py
#定义卷积核大小
kernel_size = 10
#创建直线卷积核，更精细化设置
kernel = np.zeros((kernel_size, kernel_size), dtype=np.float32)
kernel[0, :] = 1    #横线卷积核
#kernel[:, 0]=1     #竖线卷积核
```

需要注意，在 TensorFlow 中进行卷积运算时，卷积核也是四维张量，各维度分别是高度、宽度、输入通道数、输出通道数。本例中，如果原始图像通道数为 1，则第三维仅有一个元素；如果卷积运算输出的矩阵只需 1 个，即输出通道数为 1，则第四维仅有一个元素，因此只需保证前两个维度为卷积核的高和宽，后两个维度为 1，代码如下：

```
#chap3/chap3.1_line_kernel.py
kernel = kernel.reshape(kernel.shape[0], kernel.shape[1], 1, 1)
```

进行卷积计算，注意结果为四维张量，将所得结果转换成二维数组，便于绘制图形，代码如下：

```
#chap3/chap3.1_line_kernel.py
#进行卷积计算
convolved_image = tf.nn.conv2d(image, kernel, strides=[1, 1, 1, 1], padding=
'VALID')
#将卷积结果转换为 NumPy 数组(第二维和第三维)
convolved_image = convolved_image.numpy()[0, :, :, 0]
```

绘制图像，将原始的正方形和直角三角形绘制于左图，将卷积结果绘制于右图，便于对比，代码如下：

```
#chap3/chap3.1_line_kernel.py
#显示结果
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(image[0, :, :, 0])
plt.title('原始图形')
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(convolved_image)
plt.title('卷积结果')
plt.axis('off')
plt.show()
```

程序输出结果如图 3-6 所示，从图中可知横线卷积核在正方形上下两条横向边和直角三角形横向底边处结果较大，在其他位置结果较小，表明其识别出了横线。

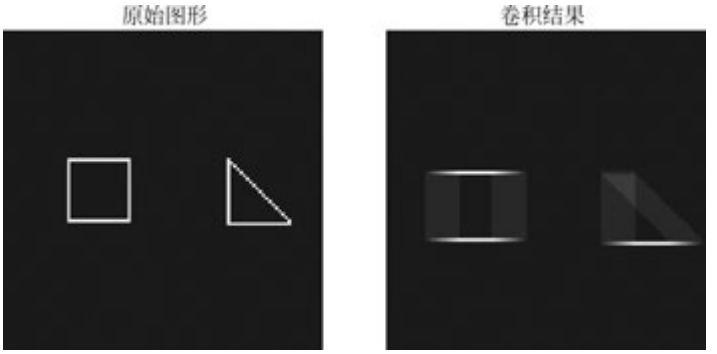

图 3-6　卷积核识别物体形状

3.9　案例 2：卷积运算识别物体边缘

案例 1 表明，通过设计特定参数的卷积核，卷积运算可实现对图像中特定几何形状（如横线、圆形等）进行识别，其本质是利用卷积核与目标形状的空间特性进行匹配，而本案例则

进一步揭示，卷积核的功能不仅限于捕捉具体、直观的形状特征，在更多场景下，它能够通过提取像素间的局部相关性与变化性，实现对抽象特征进行识别。

以物体边缘为例，卷积核可通过权重矩阵的巧妙设计（如 Sobel 算子的梯度计算逻辑），检测图像中亮度的突变区域，从而实现对物体边缘的检测，即使这些区域不构成完整的形状，仅表现为相邻像素的颜色突变，卷积核仍能通过加权求和运算将其转换为可量化的特征值。

引入相关模块，生成空白图像，代码如下：

```
#chap3/chap3.2_edge_detection.py
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf

plt.rcParams['font.family'] = ['SimSun', 'Times New Roman']

#生成图像尺寸
image_size = 100
image = np.zeros((image_size, image_size), dtype=np.float32)
```

与案例 1 不同，本案例生成内部填充的正方形，分析卷积核是否能够识别出正方形的边缘，代码如下：

```
#chap3/chap3.2_edge_detection.py
#绘制内部填充的正方形
square_size = 20
square_x = 20
square_y = 40
image[square_y:square_y + square_size, square_x:square_x + square_size - 1] = 1
```

同样，需要将图像转换为四维张量，上一案例已详细描述，此处不再赘述，代码如下：

```
#chap3/chap3.2_edge_detection.py
#调整图像维度以适配 TensorFlow
image = image.reshape(1, image.shape[0], image.shape[1], 1)
```

定义卷积核，其作用是识别物体边缘，代码如下：

```
#chap3/chap3.2_edge_detection.py
#定义卷积核
kernel = np.array([
        [0, -1, 0],
        [-1, 4, -1],
        [0, -1, 0]
    ], dtype=np.float32)
kernel = kernel.reshape(kernel.shape[0], kernel.shape[1], 1, 1)
```

使用卷积核对原始图形进行卷积运算，此处代码复用前一案例的相应代码即可，代码如下：

```
#chap3/chap3.2_edge_detection.py
#进行卷积计算
convolved_image = tf.nn.conv2d(image, kernel, strides=[1, 1, 1, 1], padding=
'VALID')
#将卷积结果转换为 NumPy 数组(第二维和第三维)
convolved_image = convolved_image.numpy()[0, :, :, 0]

#显示结果
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(image[0, :, :, 0])
plt.title('原始图形')
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(convolved_image)
plt.title('卷积结果')
plt.axis('off')

plt.show()
```

程序的运行结果如图 3-7 所示。从图中可以看出,物体边缘的卷积结果较大,非物体边缘(物体内部和物体外部)的卷积结果较小,符合预期。

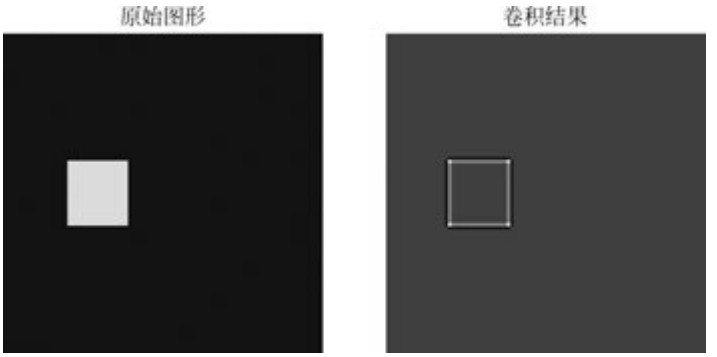

图 3-7 卷积核识别物体边缘

下面分析本案例的卷积核可以识别物体边缘的原因。该卷积核的维度是 3×3,假设中心点的坐标为(x,y),$f(x,y)$表示该点与原始图像像素的乘积,则使用该卷积核进行卷积运算的公式为 $f(x+1,y)+f(x,y+1)+f(x-1,y)+f(x,y-1)-4f(x,y)$,即使用该中心点的相邻上、下、左、右 4 个点的乘积运算之和减去中心点乘积运算的 4 倍。如果图像中某个区域中心点及相邻 4 个点的元素值没有变化(例如在正方形内部元素值都为 1,或者在正方形外部元素值都为 0),则相邻 4 个点乘积运算与中心点乘积运算抵消,结果为 0;如果中心点及相邻 4 个点的元素值有变化(例如以正方形左侧边为界,右侧处于正方形内部,则元素值为 1;左侧处于正方形外部,元素值为 0),则相邻 4 个点乘积运算与中心点乘

积运算无法抵消，结果不为0，从而实现了对物体边缘进行检测。

除了物体边缘，卷积核还能够对纹理等抽象特征提取特征，这使卷积核成为计算机视觉中从低层像素信息向高层语义特征过渡的核心工具，为后续的目标检测、图像分割等复杂任务奠定了基础。

以上两个案例通过人工指定卷积核权重，直观地展现了卷积核具备特征识别的能力，而在CNN的训练过程中，卷积核的权重并非人为预设的，而是通过数据驱动的方式自动学习及优化。CNN基于大量图像数据，通过反向传播算法不断地调整卷积核参数，最终生成与图像特征匹配度最高的卷积核，实现从底层边缘、纹理到高层特征的多层次识别。这种自动化的权重训练机制，使CNN能够自适应不同类型的图像数据，显著地提升了特征提取的灵活性与准确性。

3.10 案例3：用CNN识别手写数字

本案例将使用经典的MNIST手写数字数据集，训练出识别数字的模型。用户在程序提供的画板中手写一个数字，模型需要识别出用户所写的数字是0～9数字中的哪一个。

引入相关模块，代码如下：

```
#chap3/chap3.3_number_recognition.py
import tkinter as tk

import numpy as np
import tensorflow as tf
from PIL import Image, ImageDraw
from tensorflow.keras import layers, models
```

定义训练模型的函数，加载MNIST数据集，构建含有两层卷积层、两层最大化池化层、两层全连接层的CNN模型，代码如下：

```
#chap3/chap3.3_number_recognition.py
#训练模型的函数
def train_model():
    #加载 MNIST 数据集
    mnist = tf.keras.datasets.mnist  #第 1 次运行将从互联网下载数据集，需保持网络连
                                     #接稳定
    (train_images, train_labels), (test_images, test_labels) = mnist.load_data()

    #数据预处理
    train_images = train_images.reshape((60000, 28, 28, 1)).astype('float32') / 255
    test_images = test_images.reshape((10000, 28, 28, 1)).astype('float32') / 255

    #构建卷积神经网络
    model = models.Sequential([
        layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
```

```
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(64, (3, 3), activation='relu'),
        layers.MaxPooling2D((2, 2)),
        layers.Flatten(),
        layers.Dense(64, activation='relu'),
        layers.Dense(10, activation='softmax')
    ])

    #编译模型,由于是分类问题,所以损失函数使用交叉熵
    model.compile(optimizer='adam',
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])
    #训练模型
    model.fit(train_images, train_labels, epochs=5)
    #评估模型
    test_loss, test_acc = model.evaluate(test_images, test_labels)
    print(f'Test accuracy: {test_acc}')
    return model
```

训练模型,并使用 Tkinter 模块加载画布,支持用户用鼠标在画布中绘制图形。由于 Tkinter 并非本章的重点,因此不详细说明,代码如下:

```
#chap3/chap3.2_edge_detection.py
#训练模型
model = train_model()

#创建主窗口
root = tk.Tk()
root.title("手写数字识别")

#创建画布
canvas_width = 200
canvas_height = 200
canvas = tk.Canvas(root, width=canvas_width, height=canvas_height, bg="white")
canvas.pack()

#创建一个空白的 PIL 图像,用于保存绘制的内容
image = Image.new("L", (canvas_width, canvas_height), 255)
draw = ImageDraw.Draw(image)

#初始化绘图状态
is_drawing = False
last_x, last_y = None, None


#鼠标按下事件处理函数
def start_drawing(event):
    global is_drawing, last_x, last_y
```

```
    is_drawing = True
    last_x, last_y = event.x, event.y


#鼠标移动事件处理函数
def draw_on_canvas(event):
    global is_drawing, last_x, last_y
    if is_drawing:
        canvas.create_line((last_x, last_y, event.x, event.y), fill="black",
width=5)
        draw.line((last_x, last_y, event.x, event.y), fill=0, width=5)
        last_x, last_y = event.x, event.y


#鼠标释放事件处理函数
def stop_drawing(event):
    global is_drawing
    is_drawing = False


#绑定鼠标事件
canvas.bind("<ButtonPress-1>", start_drawing)
canvas.bind("<B1-Motion>", draw_on_canvas)
canvas.bind("<ButtonRelease-1>", stop_drawing)


#预测函数
def predict_digit():
    #将图像大小调整为 28×28
    img = image.resize((28, 28), Image.Resampling.LANCZOS)
    img_array = np.array(img)
    img_array = img_array.reshape(1, 28, 28, 1).astype('float32') / 255
    img_array = 1 - img_array  #因为白色的 RGB 值是 255,黑色的 RGB 值是 0,所以用 1 减后
                               #画出的黑色线数值更高,白色更低
    prediction = model.predict(img_array, verbose=False)
    digit = np.argmax(prediction)
    result_label.config(text=f"预测结果: {digit}")


#清空画布函数
def clear_canvas():
    canvas.delete("all")
    draw.rectangle((0, 0, canvas_width, canvas_height), fill=255)
    result_label.config(text="预测结果: ")


#创建预测按钮
predict_button = tk.Button(root, text="预测", command=predict_digit)
predict_button.pack()
```

```
#创建清空按钮
clear_button = tk.Button(root, text="清空", command=clear_canvas)
clear_button.pack()

#创建结果标签
result_label = tk.Label(root, text="预测结果: ")
result_label.pack()

#运行主循环
root.mainloop()
```

程序运行后,用户在画布中手写一个数字,模型识别所写的数字,程序输出结果如图 3-8 所示。从图中可知,程序正确地识别了用户所绘制的数字。

图 3-8　手写数字识别程序运行演示