第3章　计算机视觉

3.1　卷积神经网络基础

3.1.1　概述

计算机视觉作为一门让机器学会如何去“看”的学科，具体地说，就是让机器去识别摄像机拍摄的图片或视频中的物体，检测出物体所在的位置，并对目标物体进行跟踪，从而理解并描述出图片或视频里的场景和故事，以此来模拟人脑视觉系统。因此，计算机视觉也通常被叫作机器视觉，其目的是建立能够从图像或者视频中“感知”信息的人工系统。

计算机视觉技术经过几十年的发展，已经在交通（车牌识别、道路违章抓拍）、安防（人脸闸机、小区监控）、金融（刷脸支付、柜台的自动票据识别）、医疗（医疗影像诊断）、工业生产（产品缺陷自动检测）等多个领域应用，影响或正在改变人们的日常生活和工业生产方式。未来，随着技术的不断演进，必将涌现出更多的产品和应用，为我们的生活创造更大的便利和更广阔的机会，如图 3.1 所示。

飞桨为计算机视觉任务提供了丰富的 API，并通过底层优化和加速保证了这些 API 的性能。同时，飞桨还提供了丰富的模型库，覆盖图像分类、检测、分割、文字识别和视频理解等多个领域。用户可以直接使用这些 API 组建模型，也可以在飞桨提供的模型库基础上进行二次研发。

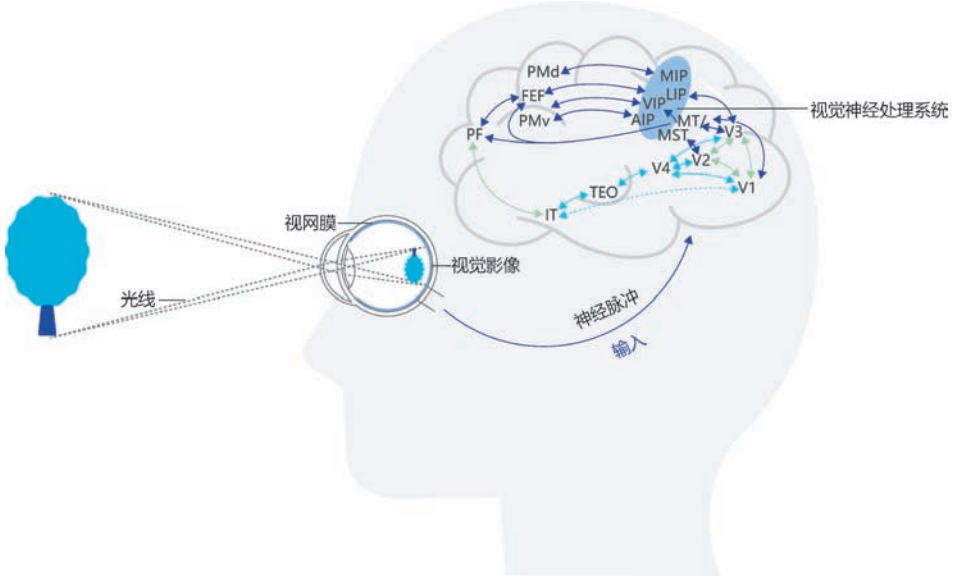
由于篇幅所限，本章将重点介绍计算机视觉的经典模型（卷积神经网络）和图像分类任务，而在下一章介绍目标检测。本章主要涵盖如下内容：

- 卷积神经网络：卷积神经网络（Convolutional Neural Networks，CNN）是计算机视觉技术最经典的模型结构。本书主要介绍卷积神经网络的常用模块，包括：卷积、池化、激活函数、批归一化、Dropout 等。
- 图像分类：介绍图像分类算法的经典模型结构，包括：LeNet、AlexNet、VGG、GoogLeNet、ResNet，并通过眼疾筛查的案例展示算法的应用。

■图 3.1　计算机视觉技术在各领域的应用

计算机视觉的发展历程

计算机视觉的发展历程要从生物视觉讲起。对于生物视觉的起源，目前学术界尚没有形成定论。有研究者认为最早的生物视觉形成于距今约 7 亿年前的水母之中，也有研究者认为生物视觉产生于距今约 5 亿年前寒武纪。经过几亿年的演化，目前人类的视觉系统已经具备非常高的复杂度和强大的功能，人脑中神经元数目达到了 1000 亿个，这些神经元通过网络互相连接，这样庞大的视觉神经网络使得我们可以很轻松地观察周围的世界，如图 3.2 所示。



■图 3.2　人类视觉感知

对人类来说，识别猫和狗是件非常容易的事。但对计算机来说，即使是一个精通编程的高手，也很难轻松写出具有通用性的程序（比如：假设程序认为体型大的是狗，体型小的是猫，但由于拍摄角度不同，可能一张图片上猫占据的像素比狗还多）。那么，如何让计算机也能像人一样看懂周围的世界呢？研究者尝试着从不同的角度去解决这个问题，由此也发展出一系列的子任务，如图 3.3 所示。

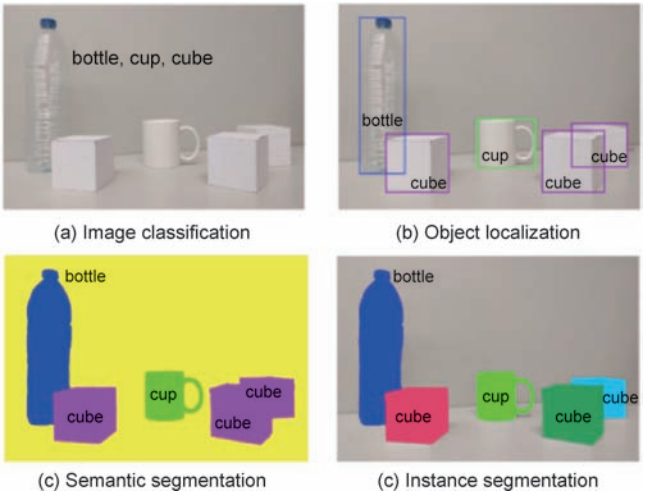


图 3.3 计算机视觉子任务示意图

（1）Image Classification：图像分类，用于识别图像中物体的类别（如：bottle、cup、cube）。

（2）Object Localization：目标检测，用于检测图像中每个物体的类别，并准确标出它们的位置。

（3）Semantic Segmentation：图像语义分割，用于标出图像中每个像素点所属的类别，属于同一类别的像素点用一个颜色标识。

（4）Instance Segmentation：实例分割，值得注意的是，(b)中的目标检测任务只需要标注出物体位置，而(d)中的实例分割任务不仅要标注出物体位置，还需要标注出物体的外形轮廓。

在早期的图像分类任务中，通常是先人工提取图像特征，再用机器学习算法对这些特征进行分类，分类的结果强依赖于特征提取方法，往往只有经验丰富的研究者才能完成，如图 3.4 所示。

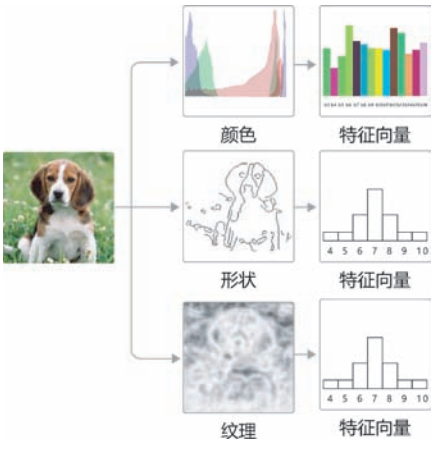
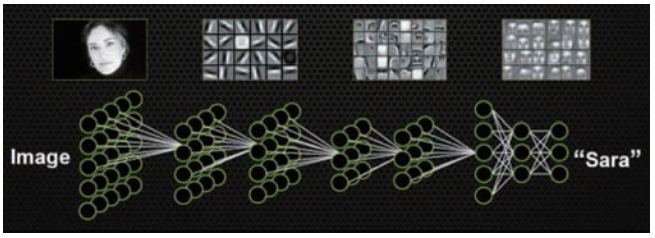


图 3.4 早期的图像分类任务

在这种背景下，基于神经网络的特征提取方法应运而生。Yann LeCun 是最早将卷积神经网络应用到图像识别领域的，其主要逻辑是使用卷积神经网络提取图像特征，并对图像所属类别进行预测，通过训练数据不断调整网络参数，最终形成一套能自动提取图像特征并对这些特征进行分类的网络，如图 3.5 所示。



■图 3.5　早期的卷积神经网络处理图像任务示意

这一方法在手写数字识别任务上取得了极大的成功，但在接下来的时间里，却没有得到很好的发展。其主要原因一方面是数据集不完善，只能处理简单任务，在大尺寸的数据集上效果比较差；另一方面是硬件瓶颈，网络模型复杂时，计算速度会特别慢。

目前，随着互联网技术的不断进步，数据量呈现大规模的增长，越来越丰富的数据集不断涌现。另外，得益于硬件能力的提升，计算机的算力也越来越强大。不断有研究者将新的模型和算法应用到计算机视觉领域。由此催生了越来越丰富的模型结构和更加准确的精度，同时计算机视觉所处理的问题也越来越丰富，包括分类、检测、分割、场景描述、图像生成和风格变换等，甚至还不仅仅局限于 2 维图片，包括视频处理技术和 3D 视觉等。

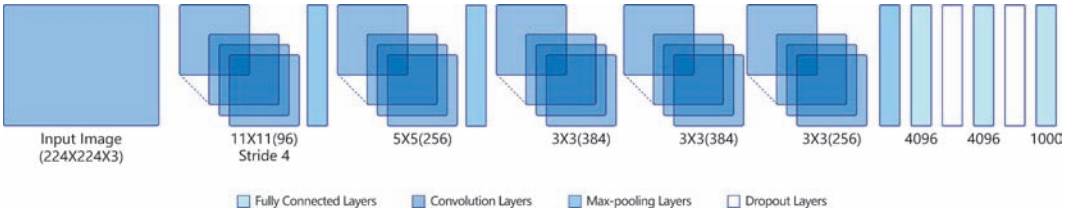
3.1.2　卷积神经网络

卷积神经网络是目前计算机视觉中使用最普遍的模型结构。回顾一下，在上一章“一个案例带你吃透深度学习”中，我们介绍了手写数字识别任务，应用的是全连接层的特征提取，即将一张图片上的所有像素点展开成一个 1 维向量输入网络，存在如下两个问题：

（1）输入数据的空间信息被丢失。空间上相邻的像素点往往具有相似的 RGB 值，RGB 的各个通道之间的数据通常密切相关，但是转化成 1 维向量时，这些信息被丢失。同时，图像数据的形状信息中，可能隐藏着某种本质的模式，但是转变成 1 维向量输入全连接神经网络时，这些模式也会被忽略。

（2）模型参数过多，容易发生过拟合。在手写数字识别案例中，每个像素点都要跟所有输出的神经元相连接。当图片尺寸变大时，输入神经元的个数会按图片尺寸的平方增大，导致模型参数过多，容易发生过拟合。

为了解决上述问题，我们引入卷积神经网络进行特征提取，既能提取到相邻像素点之间的特征模式，又能保证参数的个数不随图片尺寸变化。图 3.6 是一个典型的卷积神经网络结构，多层卷积和池化层组合作用在输入图片上，在网络的最后通常会加入一系列全连接层，ReLU 激活函数一般加在卷积或者全连接层的输出上，网络中通常还会加入 Dropout 来防止过拟合。



■图 3.6　卷积神经网络经典结构

说明：

在卷积神经网络中，计算范围是在像素点的空间邻域内进行的，卷积核参数的数目也远小于全连接层。卷积核本身与输入图片大小无关，它代表了对空间领域内某种特征模式的提取。比如，有些卷积核提取物体边缘特征，有些卷积核提取物体拐角处的特征，图像上不同区域共享同一个卷积核。当输入图片大小不一样时，仍然可以使用同一个卷积核进行操作。

卷积(Convolution)

这一小节将为读者介绍卷积算法的原理和实现方案，并通过具体的案例展示如何使用卷积对图片进行操作，主要涵盖如下内容：

(1) 卷积计算。

(2) 填充(Padding)。

(3) 步幅(Stride)。

(4) 感受野(Receptive Field)。

(5) 多输入通道、多输出通道和批量操作。

(6) 飞桨卷积 API 介绍。

(7) 卷积算子应用举例。

1) 卷积计算

卷积是数学分析中的一种积分变换的方法，在图像处理中采用的是卷积的离散形式。这里需要说明的是，在卷积神经网络中，卷积层的实现方式实际上是数学中定义的互相关(Cross-correlation)运算，与数学分析中的卷积定义有所不同，具体的计算过程如图 3.7 所示。

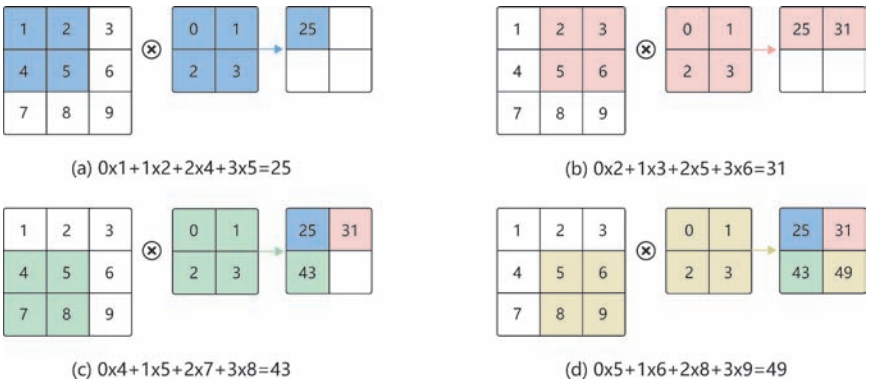


图 3.7 卷积计算过程

说明：

卷积核(kernel)也被叫作滤波器(filter)，假设卷积核的高和宽分别为 k_h 和 k_w，则将称为 $k_h \times k_w$ 卷积，比如 3×5 卷积，就是指卷积核的高为 3，宽为 5。

- 如图 3.7(a)所示：左边的图大小是 3×3，表示输入数据是一个维度为 3×3 的二维数组；中间的图大小是 2×2，表示一个维度为 2×2 的二维数组，我们将这个二维数组称为卷积核。先将卷积核的左上角与输入数据的左上角(即：输入数据的(0，0)位置)对齐，把卷积核的每个元素跟其位置对应的输入数据中的元素相乘，再把所有乘积相加，得到卷积输出的第一个结果

$$0\times1+1\times2+2\times4+3\times5=25 \tag{a}$$

- 如图 3.5(b)所示：将卷积核向右滑动，让卷积核左上角与输入数据中的(0,1)位置对齐，同样将卷积核的每个元素跟其位置对应的输入数据中的元素相乘，再把这 4 个乘积相加，得到卷积输出的第二个结果，

$$0\times2+1\times3+2\times5+3\times6=31 \tag{b}$$

- 如图 3.7(c)所示：将卷积核向下滑动，让卷积核左上角与输入数据中的(1,0)位置对齐，可以计算得到卷积输出的第三个结果，

$$0\times4+1\times5+2\times7+3\times8=43 \tag{c}$$

- 如图 3.7(d)所示：将卷积核向右滑动，让卷积核左上角与输入数据中的(1,1)位置对齐，可以计算得到卷积输出的第四个结果，

$$0\times5+1\times6+2\times8+3\times9=49 \tag{d}$$

卷积核的计算过程可以用下面的数学公式表示，其中 a 代表输入图片，b 代表输出特征图，w 是卷积核参数，它们都是二维数组，$\sum u,v$ 表示对卷积核参数进行遍历并求和。

$$b[i,j]=\sum_{u,v}a[i+u,j+v]\cdot w[u,v]$$

举例说明，假如图 3.7 中卷积核大小是 2×2，则 u 可以取 0 和 1，v 也可以取 0 和 1，也就是说：

$$b[i,j]=a[i+0,j+0]\cdot w[0,0]+a[i+0,j+1]\cdot w[0,1]+a[i+1,j+0]\cdot w[1,0]+a[i+1,j+1]\cdot w[1,1]$$

读者可以自行验证，当[i,j]取不同值时，根据此公式计算的结果与图 3.7 中的例子是否一致。

【思考】 当卷积核大小为 3×3 时，b 和 a 之间的对应关系应该是怎样的？

说明：

在卷积神经网络中，一个卷积算子除了上面描述的卷积过程之外，还包括加上偏置项的操作。例如假设偏置为 1，则上面卷积计算的结果为：

$$0\times1+1\times2+2\times4+3\times5+1=26$$
$$0\times2+1\times3+2\times5+3\times6+1=32$$
$$0\times4+1\times5+2\times7+3\times8+1=44$$
$$0\times5+1\times6+2\times8+3\times9+1=50$$

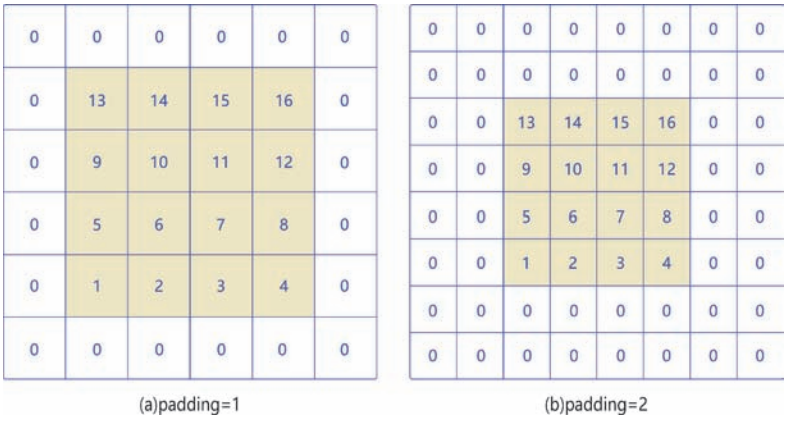
---

2) 填充

在上面的例子中，输入图片尺寸为 3×3，输出图片尺寸为 2×2，经过一次卷积之后，图片尺寸变小。卷积输出特征图的尺寸计算方法如下：

$$H_{out} = H - k_h + 1$$
$$W_{out} = W - k_w + 1$$

如果输入尺寸为4，卷积核大小为3时，输出尺寸为4−3+1=2。读者可以自行检查当输入图片和卷积核为其他尺寸时，上述计算式是否成立。当卷积核尺寸大于1时，输出特征图的尺寸会小于输入图片尺寸。如果经过多次卷积，输出图片尺寸会不断减小。为了避免卷积之后图片尺寸变小，通常会在图片的外围进行填充(padding)，如图3.8所示。



■图3.8 图形填充

- 如图3.8(a)所示：填充的大小为1，填充值为0。填充之后，输入图片尺寸从4×4变成了6×6，使用3×3的卷积核，输出图片尺寸为4×4。
- 如图3.8(b)所示：填充的大小为2，填充值为0。填充之后，输入图片尺寸从4×4变成了8×8，使用3×3的卷积核，输出图片尺寸为6×6。

如果在图片高度方向，在第一行之前填充 p_{h1} 行，在最后一行之后填充 p_{h2} 行；在图片的宽度方向，在第1列之前填充 p_{w1} 列，在最后1列之后填充 p_{w2} 列；则填充之后的图片尺寸为$(H+p_{h1}+p_{h2})\times(W+p_{w1}+p_{w2})$。经过大小为 $k_h\times k_w$ 的卷积核操作之后，输出图片的尺寸为：

$$H_{out} = H + p_{h1} + p_{h2} - k_h + 1$$
$$W_{out} = W + p_{w1} + p_{w2} - k_w + 1$$

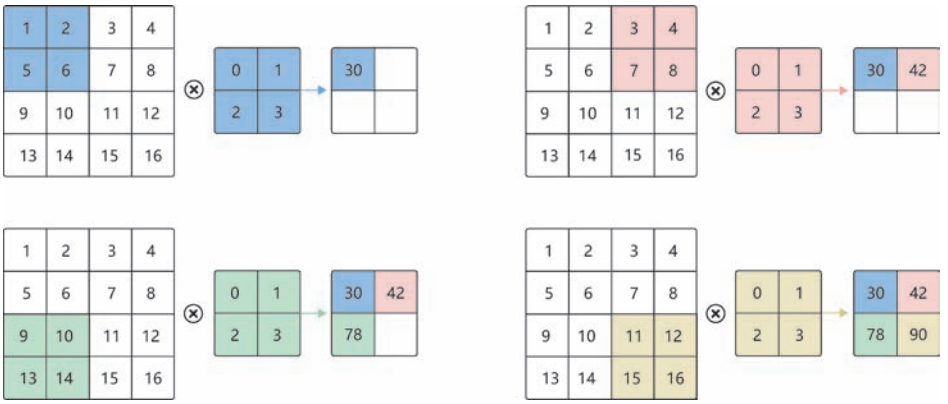
在卷积计算过程中，通常会在高度或者宽度的两侧采取等量填充，即 $p_{h1}=p_{h2}=p_h$，$p_{w1}=p_{w2}=p_w$，上面计算公式也就变为：

$$H_{out} = H + 2p_h - k_h + 1$$
$$W_{out} = W + 2p_w - k_w + 1$$

卷积核大小通常使用1,3,5,7这样的奇数，如果使用的填充大小为 $p_h=(k_h-1)/2$，$p_w=(k_w-1)/2$，则卷积之后图像尺寸不变。例如当卷积核大小为3时，padding大小为1，卷积之后图像尺寸不变；同理，如果卷积核大小为5，使用padding的大小为2，也能保持图像尺寸不变。

3) 步幅

如图3.8所示，中卷积核每次滑动一个像素点，这是步幅为1的特殊情况。如图3.9所示，是步幅为2的卷积过程，卷积核在图片上移动时，每次移动大小为2个像素点。



■图 3.9　步幅为 2 的卷积过程

当宽和高方向的步幅分别为 s_h 和 s_w 时，输出特征图尺寸的计算公式是：

$$H_{out}=\frac{H+2p_h-k_h}{s_h}+1$$

$$W_{out}=\frac{W+2p_w-k_w}{s_w}+1$$

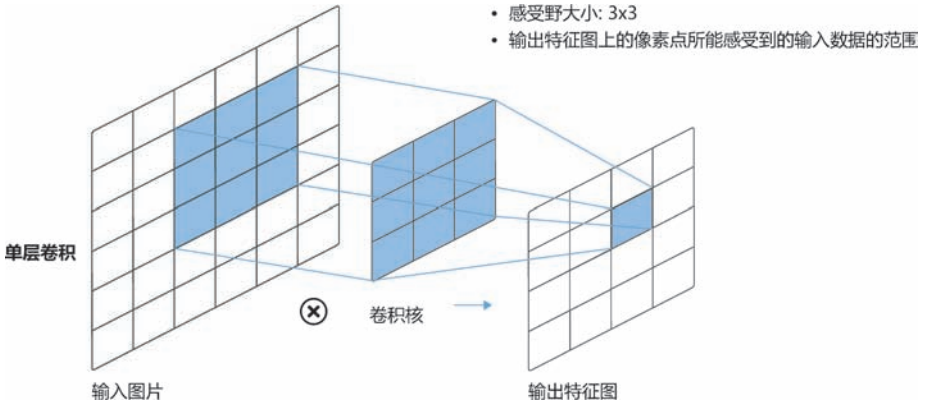
假设输入图片尺寸是 $H\times W=100\times 100$，卷积核大小 $k_h\times k_w=3\times 3$，填充 $p_h=p_w=1$，步幅为 $s_h=s_w=2$，则输出特征图的尺寸为：

$$H_{out}=\frac{100+2-3}{2}+1=50$$

$$W_{out}=\frac{100+2-3}{2}+1=50$$

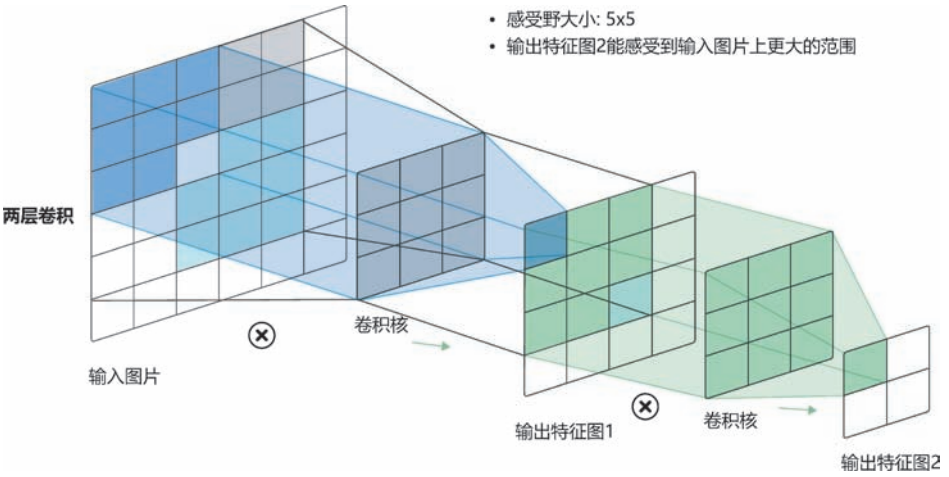
4）感受野

输出特征图上每个点的数值，是由输入图片上大小为 $k_h\times k_w$ 的区域的元素与卷积核每个元素相乘再相加得到的，所以输入图像上 $k_h\times k_w$ 区域内每个元素数值的改变，都会影响输出点的像素值。我们将这个区域叫作输出特征图上对应点的感受野。感受野内每个元素数值的变动，都会影响输出点的数值变化。比如 3×3 卷积对应的感受野大小就是 3×3，如图 3.10 所示。



■图 3.10　感受野为 3×3 的卷积

而当通过两层 3×3 的卷积之后，感受野的大小将会增加到 5×5，如图 3.11 所示。



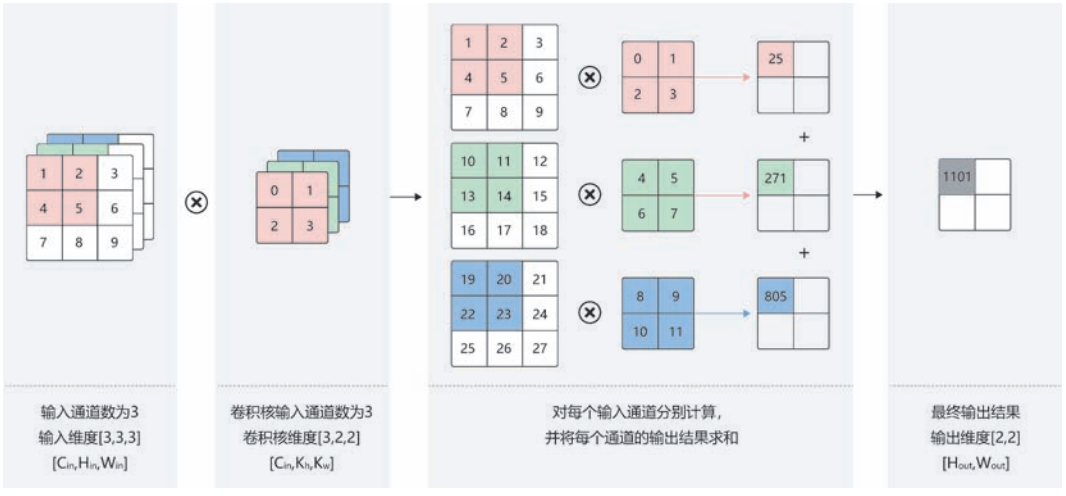
■图 3.11 感受野为 5×5 的卷积

因此，当增加卷积网络深度的同时，感受野将会增大，输出特征图中的一个像素点将会包含更多的图像语义信息。

5）多输入通道、多输出通道和批量操作

前面介绍的卷积计算过程比较简单，实际应用时，处理的问题要复杂得多。例如：对于彩色图片有 RGB 三个通道，需要处理多输入通道的场景。输出特征图往往也会具有多个通道，而且在神经网络的计算中常常是把一个批次的样本放在一起计算，所以卷积算子需要具有批量处理多输入和多输出通道数据的功能，下面将分别介绍这几种场景的操作方式。

（1）多输入通道场景。上面的例子中，卷积层的数据是一个 2 维数组，但实际上一张图片往往含有 RGB 三个通道，要计算卷积的输出结果，卷积核的形式也会发生变化。假设输入图片的通道数为 C_{in}，输入数据的形状是 $C_{\text{in}} \times H_{\text{in}} \times W_{\text{in}}$，计算过程如图 3.12 所示。



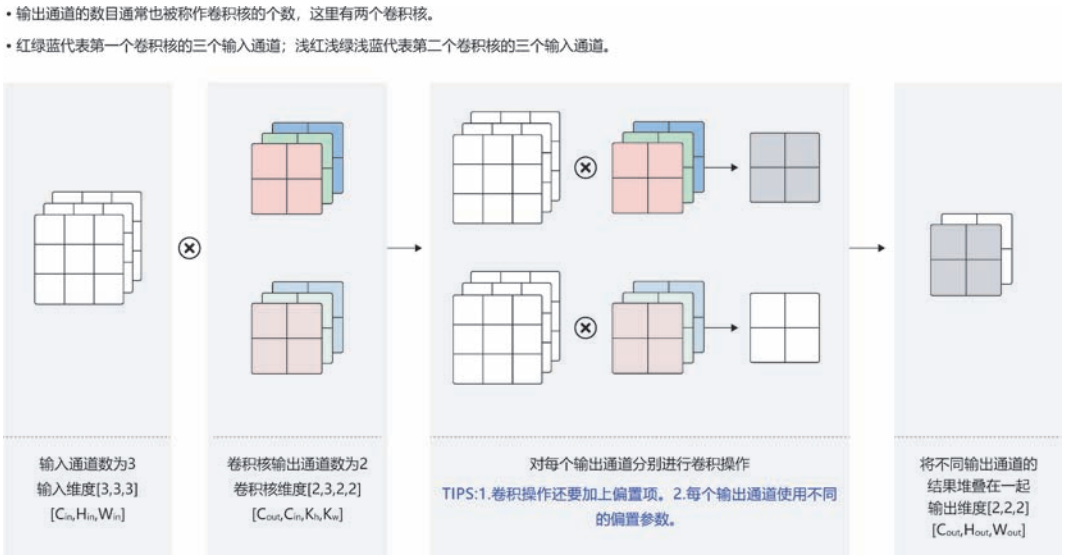
■图 3.12 多输入通道计算过程

对每个通道分别设计一个 2 维数组作为卷积核，卷积核数组的形状是 $C_{in}\times k_h\times k_w$。

对任一通道 $c_{in}\in[0,C_{in})$，分别用大小为 $k_h\times k_w$ 的卷积核在大小为 $H_{in}\times W_{in}$ 的二维数组上做卷积。

将这 C_{in} 个通道的计算结果相加，得到的是一个形状为 $H_{out}\times W_{out}$ 的二维数组。

(2) 多输出通道场景。一般来说，卷积操作的输出特征图也会具有多个通道 C_{out}，这时我们需要设计 C_{out} 个维度为 $C_{in}\times k_h\times k_w$ 的卷积核，卷积核数组的维度是 $C_{out}\times C_{in}\times k_h\times k_w$，如图 3.13 所示。



■图 3.13　多输出通道计算过程

对任一输出通道 $c_{out}\in[0,C_{out})$，分别使用上面描述的形状为 $C_{in}\times k_h\times k_w$ 的卷积核对输入图片做卷积。

将这 C_{out} 个形状为 $H_{out}\times W_{out}$ 的二维数组拼接在一起，形成维度为 $C_{out}\times H_{out}\times W_{out}$ 的三维数组。

说明：

通常将卷积核的输出通道数叫作卷积核的个数。

(3) 批量操作。在卷积神经网络的计算中，通常将多个样本放在一起形成一个 mini-batch 进行批量操作，即输入数据的维度是 $N\times C_{in}\times H_{in}\times W_{in}$。由于会对每张图片使用同样的卷积核进行卷积操作，卷积核的维度与上面多输出通道的情况一样，仍然是 $C_{out}\times C_{in}\times k_h\times k_w$，输出特征图的维度是 $N\times C_{out}\times H_{out}\times W_{out}$，如图 3.14 所示。

6) 飞桨卷积 API 介绍

飞桨卷积算子对应的 API 是 paddle.fluid.dygraph.Conv2D，用户可以直接调用 API 进行计算，也可以在此基础上修改。常用的参数如下：

- num_channels (int)——输入图像的通道数。

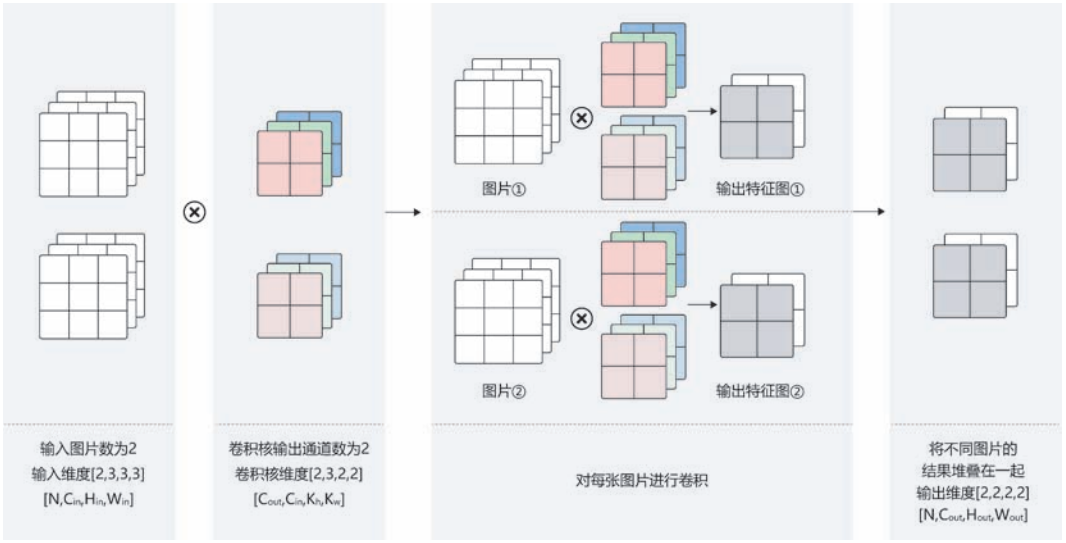


图 3.14 批量操作

- num_fliters (int)——卷积核的个数，和输出特征图通道数相同，相当于上文中的CoutC_{out}Cout。
- filter_size(int|tuple)——卷积核大小，可以是整数，比如 3，表示卷积核的高和宽均为 3；或者是两个整数的 list，例如[3,2]，表示卷积核的高为 3，宽为 2。
- stride(int|tuple)——步幅，可以是整数，默认值为 1，表示垂直和水平滑动步幅均为 1；或者是两个整数的 list，例如[3,2]，表示垂直滑动步幅为 3，水平滑动步幅为 2。
- padding(int|tuple)——填充大小，可以是整数，比如 1，表示竖直和水平边界填充大小均为 1；或者是两个整数的 list，例如[2,1]，表示竖直边界填充大小为 2，水平边界填充大小为 1。
- act(str)——应用于输出上的激活函数，如 Tanh、Softmax、Sigmoid，Relu 等，默认值为 None。

输入数据维度$[N, C_{in}, H_{in}, W_{in}]$，输出数据维度[N，num_filters，$H_{out}$，$W_{out}$]，权重参数 w 的维度[num_filters，C_{in}，filter_size_h，filter_size_w]，偏置参数 b 的维度是[num_filters]。

7）卷积算子应用举例

下面介绍卷积算子在图片中应用的三个案例，并观察其计算结果。

案例 1——简单的黑白边界检测 下面是使用 Conv2D 算子完成一个图像边界检测的任务。图像左边为光亮部分，右边为黑暗部分，需要检测出光亮跟黑暗的分界处。可以设置宽度方向的卷积核为[1,0,−1]，此卷积核会将宽度方向间隔为 1 的两个像素点的数值相减。当卷积核在图片上滑动的时候，如果它所覆盖的像素点位于亮度相同的区域，则左右间隔为 1 的两个像素点数值的差为 0。只有当卷积核覆盖的像素点有的处于光亮区域，有的处在黑暗区域时，左右间隔为 1 的两个点像素值的差才不为 0。将此卷积核作用到图片上，输出特征图上只有对应黑白分界线的地方像素值才不为 0。具体代码如下所示，结果如图 3.15 所示。

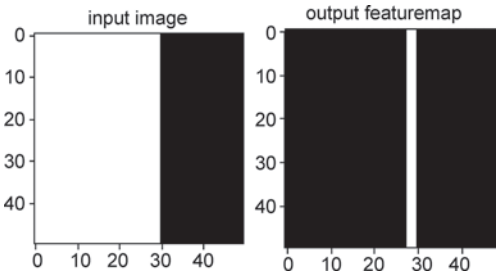


图 3.15 输出结果

```
import matplotlib.pyplot as plt
import numpy as np
import paddle
import paddle.fluid as fluid
from paddle.fluid.dygraph.nn import Conv2D
from paddle.fluid.initializer import NumpyArrayInitializer
%matplotlib inline

with fluid.dygraph.guard():
    # 创建初始化权重参数 w
    w = np.array([1, 0, -1], dtype='float32')
    # 将权重参数调整成维度为[cout, cin, kh, kw]的四维张量
    w = w.reshape([1, 1, 1, 3])
    # 创建卷积算子,设置输出通道数,卷积核大小和初始化权重参数
    # filter_size = [1, 3]表示 kh = 1, kw = 3
    # 创建卷积算子的时候,通过参数属性 param_attr,指定参数初始化方式
    # 这里的初始化方式时,从 numpy.ndarray 初始化卷积参数
    conv = Conv2D(num_channels=1, num_filters=1, filter_size=[1, 3],
            param_attr=fluid.ParamAttr(
              initializer=NumpyArrayInitializer(value=w)))

    # 创建输入图片,图片左边的像素点取值为 1,右边的像素点取值为 0
    img = np.ones([50,50], dtype='float32')
    img[:, 30:] = 0.
    # 将图片形状调整为[N, C, H, W]的形式
    x = img.reshape([1,1,50,50])
    # 将 numpy.ndarray 转化成 paddle 中的 tensor
    x = fluid.dygraph.to_variable(x)
    # 使用卷积算子作用在输入图片上
    y = conv(x)
    # 将输出 tensor 转化为 numpy.ndarray
    out = y.numpy()

f = plt.subplot(121)
f.set_title('input image', fontsize=15)
plt.imshow(img, cmap='gray')

f = plt.subplot(122)
f.set_title('output featuremap', fontsize=15)
# 卷积算子 Conv2D 输出数据形状为[N, C, H, W]形式
# 此处 N, C=1,输出数据形状为[1, 1, H, W],是 4 维数组
```

```
# 但是画图函数 plt.imshow 画灰度图时,只接受 2 维数组
# 通过 numpy.squeeze 函数将大小为 1 的维度消除
plt.imshow(out.squeeze(), cmap = 'gray')
plt.show()

# 查看卷积层的参数
with fluid.dygraph.guard():
    # 通过 conv.parameters()查看卷积层的参数,返回值是 list,包含两个元素
    print(conv.parameters())
    # 查看卷积层的权重参数名字和数值
    print(conv.parameters()[0].name, conv.parameters()[0].numpy())
    # 参看卷积层的偏置参数名字和数值
print(conv.parameters()[1].name, conv.parameters()[1].numpy())
[name conv2d_0.w_0, dtype: VarType.FP32 shape: [1, 1, 1, 3]      lod: {}
    dim: 1, 1, 1, 3
    layout: NCHW
    dtype: float
    data: [1 0 -1]
, name conv2d_0.b_0, dtype: VarType.FP32 shape: [1]      lod: {}
    dim: 1
    layout: NCHW
    dtype: float
    data: [0]
]
conv2d_0.w_0 [[[[ 1.  0. -1.]]]]
conv2d_0.b_0 [0.]
```

案例 2——图像中物体边缘检测 上面展示的是一个人为构造出来的简单图片使用卷积检测明暗分界处的例子,对于真实的图片,也可以使用合适的卷积核对它进行操作,用来检测物体的外形轮廓,观察输出特征图跟原图之间的对应关系,如下代码所示:

```
import matplotlib.pyplot as plt
from PIL import Image
import numpy as np
import paddle
import paddle.fluid as fluid
from paddle.fluid.dygraph.nn import Conv2D
from paddle.fluid.initializer import NumpyArrayInitializer

img = Image.open('./work/images/section1/000000098520.jpg')
with fluid.dygraph.guard():
    # 设置卷积核参数
    w = np.array([[-1,-1,-1], [-1,8,-1], [-1,-1,-1]], dtype='float32')/8
    w = w.reshape([1, 1, 3, 3])
    # 由于输入通道数是 3,将卷积核的形状从[1,1,3,3]调整为[1,3,3,3]
    w = np.repeat(w, 3, axis = 1)
    # 创建卷积算子,输出通道数为 1,卷积核大小为 3×3,
    # 并使用上面设置好的数值作为卷积核权重的初始化参数
    conv = Conv2D(num_channels = 3, num_filters = 1, filter_size = [3, 3],
            param_attr = fluid.ParamAttr(
              initializer = NumpyArrayInitializer(value = w)))
```

```
    # 将读入的图片转化为 float32 类型的 numpy.ndarray
    x = np.array(img).astype('float32')
    # 图片读入成 ndarry 时,形状是[H, W, 3],
    # 将通道这一维度调整到最前面
    x = np.transpose(x, (2,0,1))
    # 将数据形状调整为[N, C, H, W]格式
    x = x.reshape(1, 3, img.height, img.width)
    x = fluid.dygraph.to_variable(x)
    y = conv(x)
    out = y.numpy()

plt.figure(figsize=(20, 10))
f = plt.subplot(121)
f.set_title('input image', fontsize=15)
plt.imshow(img)
f = plt.subplot(122)
f.set_title('output feature map', fontsize=15)
plt.imshow(out.squeeze(), cmap='gray')
plt.show()
```

输出结果如图 3.16 所示。

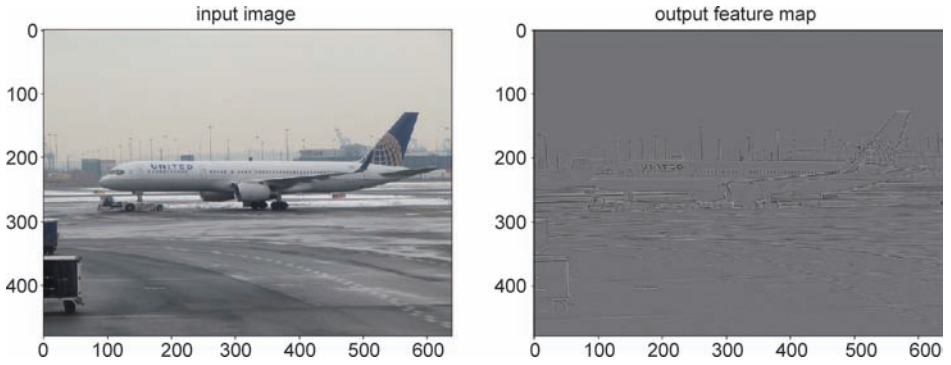


图 3.16 输出结果

案例 3——图像均值模糊 另外一种比较常见的卷积核是用当前像素跟它邻域内的像素取平均,这样可以使图像上噪声比较大的点变得更平滑,如下代码所示:

```
import matplotlib.pyplot as plt
from PIL import Image
import numpy as np
import paddle
import paddle.fluid as fluid
from paddle.fluid.dygraph.nn import Conv2D
from paddle.fluid.initializer import NumpyArrayInitializer

# 读入图片并转成 numpy.ndarray
#img = Image.open('./images/section1/000000001584.jpg')
img = Image.open('./work/images/section1/000000355610.jpg').convert('L')
img = np.array(img)
```

```
    # 换成灰度图
    with fluid.dygraph.guard():
        # 创建初始化参数
        w = np.ones([1, 1, 5, 5], dtype = 'float32')/25
        conv = Conv2D(num_channels=1, num_filters=1, filter_size=[5, 5],
                param_attr=fluid.ParamAttr(
                  initializer=NumpyArrayInitializer(value=w)))
        x = img.astype('float32')
        x = x.reshape(1,1,img.shape[0], img.shape[1])
        x = fluid.dygraph.to_variable(x)
        y = conv(x)
        out = y.numpy()
    plt.figure(figsize=(20, 12))
    f = plt.subplot(121)
    f.set_title('input image')
    plt.imshow(img, cmap='gray')
    f = plt.subplot(122)
    f.set_title('output feature map')
    out = out.squeeze()
    plt.imshow(out, cmap='gray')
    plt.show()
```

输出结果如图 3.17 所示。

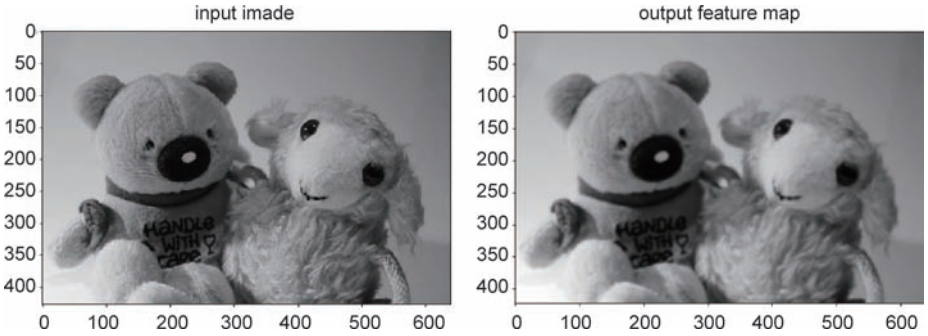


图 3.17 输出结果

3.1.3 作业

计算下面卷积中一共有多少次乘法和加法操作。

输入数据形状是[10,3,224,224]，卷积核 $k_h=k_w=3$，输出通道数为 64，步幅 stride=1，填充 $p_h=p_w=1$。则完成这样一个卷积，一共需要做多少次乘法和加法操作？

提示：

先看输出一个像素点需要做多少次乘法和加法操作，然后再计算总共需要的操作次数。

作业提交方式

请读者扫描图书封底的二维码，在 AI Studio“零基础实践深度学习”课程中的“作业”节点下提交相关作业。

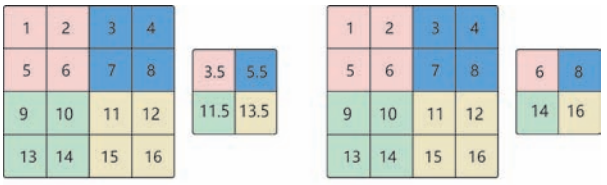
3.2 卷积的四种操作

3.2.1 概述

上一节我们介绍了卷积的基本操作与计算，这一节我们讲解卷积之后一般需要进行的4种操作——池化、激活函数、批归一化和丢弃法。

3.2.2 池化

池化是使用某一位置的相邻输出的总体统计特征代替网络在该位置的输出，其好处是当输入数据做出少量平移时，经过池化函数后的大多数输出还能保持不变。比如：当识别一张图像是否是人脸时，我们需要知道人脸左边有一只眼睛，右边也有一只眼睛，而不需要知道眼睛的精确位置，这时候通过池化某一片区域的像素点来得到总体统计特征会显得很有用。由于池化之后特征图会变得更小，如果后面连接的是全连接层，能有效地减少神经元的个数，节省存储空间并提高计算效率。如图3.18所示，将一个2×2的区域池化成一个像素点。通常有两种方法，平均池化和最大池化。



(a) 平均池化　　(b) 最大池化

图3.18　池化

- 如图3.15(a)：平均池化。这里使用大小为2×2的池化窗口，每次移动的步长也为2，对池化窗口覆盖区域内的像素数值取平均，得到相应的输出特征图的像素值。
- 如图3.15(b)：最大池化。对池化窗口覆盖区域内的像素取最大值，得到输出特征图的像素值。当池化窗口在图片上滑动时，会得到整张输出特征图。池化窗口的大小称为池化大小，用 $k_h \times k_w$ 表示。在卷积神经网络中用的比较多的是窗口大小为2×2，步长也为2的池化。

与卷积核类似，池化窗口在图片上滑动时，每次移动的步长称为步幅，当宽和高方向的移动大小不一样时，分别用 s_w 和 s_h 表示。也可以对需要进行池化的图片进行填充，填充方式与卷积类似，假设在第一行之前填充 p_{h1} 行，在最后一行后面填充 p_{h2} 行。在第一列之前填充 p_{w1} 列，在最后一列之后填充 p_{w2} 列，则池化层的输出特征图大小为：

$$H_{out} = \frac{H + p_{h1} + p_{h2} - k_h}{s_h} + 1$$

$$W_{out} = \frac{W + p_{w1} + p_{w2} - k_w}{s_w} + 1$$

在卷积神经网络中，通常使用 2×2 大小的池化窗口，步幅也使用 2，填充为 0，则输出特征图的尺寸为：

$$H_{\text{out}} = \frac{H}{2}$$

$$W_{\text{out}} = \frac{W}{2}$$

通过这种方式的池化，输出特征图的高和宽都减半，但通道数不会改变。

3.2.3 ReLU 激活函数

前面介绍的网络结构中，普遍使用 Sigmoid 函数做激活函数。在神经网络发展的早期，Sigmoid 函数用得比较多，而目前用得较多的激活函数是 ReLU。这是因为 Sigmoid 函数在反向传播过程中，容易造成梯度的衰减。让我们仔细观察 Sigmoid 函数的形式，就能发现这一问题。

Sigmoid 激活函数定义如下：

$$y = \frac{1}{1 + e^{-x}}$$

ReLU 激活函数的定义如下：

$$y = \begin{cases} 0, & (x < 0) \\ x, & (x \geqslant 0) \end{cases}$$

下面的程序画出了 Sigmoid 和 ReLU 函数的曲线图（见图 3.19）：

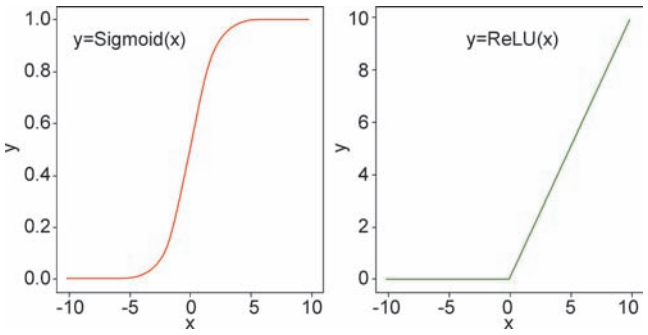


图 3.19 Sigmoid 和 ReLU 函数曲线图

```
# ReLU 和 Sigmoid 激活函数示意图
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
plt.figure(figsize=(10, 5))

# 创建数据 x
x= np.arange(-10, 10, 0.1)

# 计算 Sigmoid 函数
s= 1.0 / (1 + np.exp(0. - x))
```

```
# 计算 ReLU 函数
y = np.clip(x, a_min = 0., a_max = None)

#########################################
# 以下部分为画图代码
f = plt.subplot(121)
plt.plot(x, s, color = 'r')
currentAxis = plt.gca()
plt.text( - 9.0, 0.9, r' $ y = Sigmoid(x) $ ', fontsize = 13)
currentAxis.xaxis.set_label_text('x', fontsize = 15)
currentAxis.yaxis.set_label_text('y', fontsize = 15)
f = plt.subplot(122)
plt.plot(x, y, color = 'g')
plt.text( - 3.0, 9, r' $ y = ReLU(x) $ ', fontsize = 13)
currentAxis = plt.gca()
currentAxis.xaxis.set_label_text('x', fontsize = 15)
currentAxis.yaxis.set_label_text('y', fontsize = 15)

plt.show()
```

梯度消失现象

在神经网络里面，将经过反向传播之后，梯度值衰减到接近于零的现象称作梯度消失现象。

从上面的函数曲线可以看出，当 x 为较大的正数的时候，Sigmoid 函数数值非常接近于 1，函数曲线变得很平滑，在这些区域 Sigmoid 函数的导数接近于零。当 x 为较小的负数的时候，Sigmoid 函数值非常接近于 0，函数曲线也很平滑，在这些区域 Sigmoid 函数的导数也接近于 0。只有当 x 的取值在 0 附近时，Sigmoid 函数的导数才比较大。可以对 Sigmoid 函数求导数，结果如下所示：

$$\frac{\mathrm{d}y}{\mathrm{d}x}=-\frac{1}{(1+\mathrm{e}^{-x})^2}\cdot\frac{\mathrm{d}(\mathrm{e}^{-x})}{\mathrm{d}x}=\frac{1}{2+\mathrm{e}^{x}+\mathrm{e}^{-x}}$$

从上面的式子可以看出，Sigmoid 函数的导数 $\frac{\mathrm{d}y}{\mathrm{d}x}$ 最大值为 $\frac{1}{4}$。前向传播时，$y=\mathrm{Sigmoid}(x)$；而在反向传播过程中，x 的梯度等于 y 的梯度乘以 Sigmoid 函数的导数，如下所示：

$$\frac{\partial L}{\partial x}=\frac{\partial L}{\partial y}\cdot\frac{\partial y}{\partial x}$$

使得 x 的梯度数值最大也不会超过 y 的梯度的 $\frac{1}{4}$。

由于最开始是将神经网络的参数随机初始化的，x 很有可能取值在数值很大或者很小的区域，这些地方都可能造成 Sigmoid 函数的导数接近于 0，导致 x 的梯度接近于 0；即使 x 取值在接近于 0 的地方，按上面的分析，经过 Sigmoid 函数反向传播之后，x 的梯度不超过 y 的梯度的 $\frac{1}{4}$，如果有多层网络使用了 Sigmoid 激活函数，则比较靠后的那些层梯度将衰减到非常小的值。

ReLU 函数则不同，虽然在 $x<0$ 的地方，ReLU 函数的导数为 0。但是在 $x>0$ 的地

方，ReLU 函数的导数为 1，能够将 y 的梯度完整地传递给 x，而不会引起梯度消失。

3.2.4 批归一化

批归一化方法(Batch Normalization，BatchNorm)是由 Ioffe 和 Szegedy 于 2015 年提出的，已被广泛应用在深度学习中，其目的是对神经网络中间层的输出进行标准化处理，使得中间层的输出更加稳定。

通常我们会对神经网络的数据进行标准化处理，处理后的样本数据集满足均值为 0，方差为 1 的统计分布，这是因为当输入数据的分布比较固定时，有利于算法的稳定和收敛。对于深度神经网络来说，由于参数是不断更新的，即使输入数据已经做过标准化处理，但是对于比较靠后的那些层，其接收到的输入仍然是剧烈变化的，通常会导致数值不稳定，模型很难收敛。BatchNorm 能够使神经网络中间层的输出变得更加稳定，并有如下三个优点：

(1) 使学习快速进行(能够使用较大的学习率)。

(2) 降低模型对初始值的敏感性。

(3) 从一定程度上抑制过拟合。

BatchNorm 主要思路是在训练时按 mini-batch 为单位，对神经元的数值进行归一化，使数据的分布满足均值为 0，方差为 1。具体计算过程如下：

1. 计算 mini-batch 内样本的均值

$$\mu_{\mathrm{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$$

其中 $x^{(i)}$ 表示 mini-batch 中的第 i 个样本。

例如输入 mini-batch 包含 3 个样本，每个样本有 2 个特征，分别是：

$$x^{(1)}=(1,2),\quad x^{(2)}=(3,6),\quad x^{(3)}=(5,10)$$

对每个特征分别计算 mini-batch 内样本的均值：

$$\mu_{\mathrm{B0}}=\frac{1+3+5}{3}=3,\quad \mu_{\mathrm{B1}}=\frac{2+6+10}{3}=6$$

则样本均值是：

$$\mu_{\mathrm{B}}=(\mu_{\mathrm{B0}},\mu_{\mathrm{B1}})=(3,6)$$

2. 计算 mini-batch 内样本的方差

$$\sigma_{\mathrm{B}}^{2} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x^{(i)}-\mu_{\mathrm{B}})^{2}$$

上面的计算公式先计算一个批次内样本的均值 μ_{B} 和方差 σ_{B}^{2}，然后再对输入数据做归一化，将其调整成均值为 0，方差为 1 的分布。

对于上述给定的输入数据 $x^{(1)}$，$x^{(2)}$，$x^{(3)}$，可以计算出每个特征对应的方差：

$$\sigma_{\mathrm{B0}}^{2}=\frac{1}{3}\cdot((1-3)^{2}+(3-3)^{2}+(5-3)^{2})=\frac{8}{3}$$

$$\sigma_{\mathrm{B1}}^{2}=\frac{1}{3}\cdot((2-6)^{2}+(6-6)^{2}+(10-6)^{2})=\frac{32}{3}$$

则样本方差是：

$$\sigma_{\mathrm{B}}^2=(\sigma_{\mathrm{B}0}^2,\sigma_{\mathrm{B}1}^2)=\left(\frac{8}{3},\frac{32}{3}\right)$$

3. 计算标准化之后的输出

$$\hat{x}^{(i)}\leftarrow\frac{x^{(i)}-\mu_{\mathrm{B}}}{\sqrt{(\sigma_{\mathrm{B}}^2+\epsilon)}}$$

其中 ϵ 是一个微小值(例如 1e−7),其主要作用是为了防止分母为 0。

对于上述给定的输入数据 $x^{(1)},x^{(2)},x^{(3)}$,可以计算出标准化之后的输出:

$$\hat{x}^{(1)}=\left(\frac{1-3}{\sqrt{\frac{8}{3}}},\quad\frac{2-6}{\sqrt{\frac{32}{3}}}\right)=\left(-\sqrt{\frac{3}{2}},\quad-\sqrt{\frac{3}{2}}\right)$$

$$\hat{x}^{(2)}=\left(\frac{3-3}{\sqrt{\frac{8}{3}}},\quad\frac{6-6}{\sqrt{\frac{32}{3}}}\right)=(0,0)$$

$$\hat{x}^{(1)}=\left(\frac{5-3}{\sqrt{\frac{8}{3}}},\quad\frac{10-6}{\sqrt{\frac{32}{3}}}\right)=\left(\sqrt{\frac{3}{2}},\quad\sqrt{\frac{3}{2}}\right)$$

读者可以自行验证由 $\hat{x}^{(1)},\hat{x}^{(2)},\hat{x}^{(3)}$ 构成的 mini-batch,是否满足均值为 0,方差为 1 的分布。

如果强行限制输出层的分布是标准化的,可能会导致某些特征模式的丢失,所以在标准化之后,BatchNorm 会紧接着对数据做缩放和平移。

$$y_i\leftarrow\gamma\hat{x}_i+\beta$$

其中 γ 和 β 是可学习的参数,可以赋初始值 $\gamma=1,\beta=0$,在训练过程中不断学习调整。

上面列出的是 BatchNorm 方法的计算逻辑,下面针对两种类型的输入数据格式分别进行举例。飞桨支持输入数据的维度大小为 2、3、4、5 四种情况,这里给出的是维度大小为 2 和 4 的示例。

(1) 示例一:当输入数据形状是[N,K]时,一般对应全连接层的输出,示例代码如下所示。

这种情况下会分别对 K 的每一个分量计算 N 个样本的均值和方差,数据和参数对应如下:

- 输入 x,[N, K]
- 输出 y,[N, K]
- 均值 μ_{B},[K,]
- 方差 σ_{B}^2, [K,]
- 缩放参数 γ, [K,]
- 平移参数 β, [K,]

```
# 输入数据形状是 [N, K]时的示例
import numpy as np
import paddle
```

```
import paddle.fluid as fluid
from paddle.fluid.dygraph.nn import BatchNorm
# 创建数据
data = np.array([[1,2,3], [4,5,6], [7,8,9]]).astype('float32')
# 使用 BatchNorm 计算归一化的输出
with fluid.dygraph.guard():
    # 输入数据维度[N, K],num_channels 等于 K
    bn = BatchNorm(num_channels = 3)
    x = fluid.dygraph.to_variable(data)
    y = bn(x)
    print('output of BatchNorm Layer: \n {}'.format(y.numpy()))

# 使用 NumPy 计算均值、方差和归一化的输出
# 这里对第 0 个特征进行验证
a = np.array([1,4,7])
a_mean = a.mean()
a_std = a.std()
b = (a - a_mean) / a_std
print('std {}, mean {}, \n output {}'.format(a_mean, a_std, b))

# 建议读者对第 1 和第 2 个特征进行验证,观察 numpy 计算结果与 paddle 计算结果是否一致
```

(2) 示例二：当输入数据形状是$[N,C,H,W]$时，一般对应卷积层的输出，示例代码如下所示。

这种情况下会沿着 C 这一维度进行展开，分别对每一个通道计算 N 个样本中总共 $N\times H\times W$ 个像素点的均值和方差，数据和参数对应如下：

- 输入 x，$[N, C, H, W]$
- 输出 y，$[N, C, H, W]$
- 均值 μ_B，$[C,]$
- 方差 σ_B^2，$[C,]$
- 缩放参数 γ，$[C,]$
- 平移参数 β，$[C,]$

小窍门：

可能有读者会问："BatchNorm 里面不是还要对标准化之后的结果做仿射变换吗，怎么使用 NumPy 计算的结果与 BatchNorm 算子一致？"这是因为 BatchNorm 算子里面自动设置初始值 $\gamma=1,\beta=0$，这时候仿射变换相当于是恒等变换。在训练过程中这两个参数会不断地学习，这时仿射变换就会起作用。

```
# 输入数据形状是[N, C, H, W]时的 batchnorm 示例
import numpy as np
import paddle
import paddle.fluid as fluid
from paddle.fluid.dygraph.nn import BatchNorm

# 设置随机数种子,这样可以保证每次运行结果一致
```

```
np.random.seed(100)
# 创建数据
data = np.random.rand(2,3,3,3).astype('float32')
# 使用 BatchNorm 计算归一化的输出
with fluid.dygraph.guard():
    # 输入数据维度[N, C, H, W],num_channels 等于 C
    bn = BatchNorm(num_channels = 3)
    x = fluid.dygraph.to_variable(data)
    y = bn(x)
    print('input of BatchNorm Layer: \n {}'.format(x.numpy()))
    print('output of BatchNorm Layer: \n {}'.format(y.numpy()))

# 取出 data 中第 0 通道的数据,
# 使用 numpy 计算均值、方差及归一化的输出
a = data[:, 0, :, :]
a_mean = a.mean()
a_std = a.std()
b = (a - a_mean) / a_std
print('channel 0 of input data: \n {}'.format(a))
print('std {}, mean {}, \n output: \n {}'.format(a_mean, a_std, b))

# 提示: 这里通过 numpy 计算出来的输出
# 与 BatchNorm 算子的结果略有差别,
# 因为在 BatchNorm 算子为了保证数值的稳定性,
# 在分母里面加上了一个比较小的浮点数 epsilon = 1e - 05
```

4. 预测时使用 BatchNorm

上面介绍了在训练过程中使用 BatchNorm 对一批样本进行归一化的方法，但如果使用同样的方法对需要预测的一批样本进行归一化，则预测结果会出现不确定性。

例如样本 A、样本 B 作为一批样本计算均值和方差，与样本 A、样本 C 和样本 D 作为一批样本计算均值和方差，得到的结果一般来说是不同的。那么样本 A 的预测结果就会变得不确定，这对预测过程来说是不合理的。解决方法是在训练过程中将大量样本的均值和方差保存下来，预测时直接使用保存好的值而不再重新计算。实际上，在 BatchNorm 的具体实现中，训练时会计算均值和方差的移动平均值。在飞桨中，默认是采用如下方式计算：

$$\text{saved\_}\mu_B \leftarrow \text{saved\_}\mu_B \times 0.9 + \mu_B \times (1 - 0.9)$$

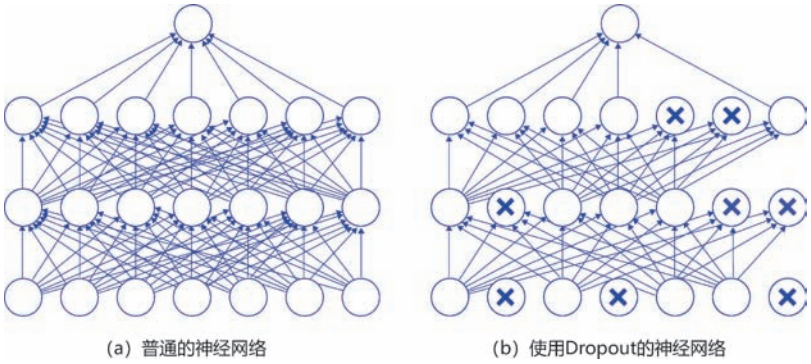
$$\text{saved\_}\sigma_B^2 \leftarrow \text{saved\_}\sigma_B^2 \times 0.9 + \sigma_B^2 \times (1 - 0.9)$$

在训练过程的最开始将 $\text{saved\_}\mu_B$ 和 $\text{saved\_}\sigma_B^2$ 设置为 0，每次输入一批新的样本，计算出 μ_B 和 σ_B^2，然后通过上面的公式更新 $\text{saved\_}\mu_B$ 和 $\text{saved\_}\sigma_B^2$，在训练的过程中不断更新它们的值，并作为 BatchNorm 层的参数保存下来。预测的时候将会加载参数 $\text{saved\_}\mu_B$ 和 $\text{saved\_}\sigma_B^2$，用它们来代替 μ_B 和 σ_B^2。

3.2.5 丢弃法

丢弃法(Dropout)是深度学习中一种常用的抑制过拟合的方法，其做法是在神经网络学习过程中，随机删除一部分神经元。训练时，随机选出一部分神经元，将其输出设置为 0，这些神经元将不对外传递信号。

图 3.20 是 Dropout 示意图，左边是完整的神经网络，右边是应用了 Dropout 之后的网络结构。应用 Dropout 之后，会将标了×的神经元从网络中删除，让它们不向后面的层传递信号。在学习过程中，丢弃哪些神经元是随机决定，因此模型不会过度依赖某些神经元，且在一定程度上能抑制过拟合。



■图 3.20 Dropout 示意图

在预测场景时，会向前传递所有神经元的信号，可能会引出一个新的问题：训练时由于部分神经元被随机丢弃了，输出数据的总大小会变小。比如：计算其 L1 范数会比不使用 Dropout 时变小，但是预测时却没有丢弃神经元，这将导致训练和预测时数据的分布不一样。为了解决这个问题，飞桨支持如下两种方法：

1）downgrade_in_infer

训练时以比例 r 随机丢弃一部分神经元，不向后传递它们的信号；预测时向后传递所有神经元的信号，但是将每个神经元上的数值乘以（1－r）。

2）upscale_in_train

训练时以比例 r 随机丢弃一部分神经元，不向后传递它们的信号，但是将那些被保留的神经元上的数值除以（1－r）；预测时向后传递所有神经元的信号，不做任何处理。

在飞桨 dropout API 中，paddle.fluid.layers.dropout 通过 dropout_implementation 参数来指定用哪种方式对神经元进行操作，dropout_implementation 参数的可选值是'downgrade_in_infer'或'upscale_in_train'，默认值是'downgrade_in_infer'。

说明：

不同框架中 dropout 的默认处理方式可能不一样，读者可以查看其 API 以确认用的是哪种方式。

飞桨 dropout API 包含的主要参数如下：

- x，数据类型是 Tensor，需要采用丢弃法进行操作的对象。
- dropout_prob，对 x 中元素进行丢弃的概率，即输入单元设置为 0 的概率，该参数对元素的丢弃概率是对于每一个元素而言而不是对所有的元素而言。举例来说，假设矩阵内有 12 个数字，则经过概率为 0.5 的 dropout 未必一定有 6 个零。
- is_test，是否运行在测试阶段，由于 dropout 在训练和测试阶段表现不一样，通过此

参数控制其表现，默认值为 False。

- dropout_implementation，丢弃法的实现方式，有'downgrade_in_infer'和'upscale_in_train'两种，具体情况请见上面的说明，默认是'downgrade_in_infer'。

下面这段程序展示了经过 dropout 之后输出数据的形式。

```
# dropout 操作
import numpy as np
import paddle
import paddle.fluid as fluid

# 设置随机数种子，这样可以保证每次运行结果一致
np.random.seed(100)
# 创建数据[N, C, H, W]，一般对应卷积层的输出
data1 = np.random.rand(2,3,3,3).astype('float32')
# 创建数据[N, K]，一般对应全连接层的输出
data2 = np.arange(1,13).reshape([-1, 3]).astype('float32')
# 使用 dropout 作用在输入数据上
with fluid.dygraph.guard():
    x1 = fluid.dygraph.to_variable(data1)
    out1_1 = fluid.dygraph.dropout(x1, dropout_prob=0.5, is_test=False)
    out1_2 = fluid.dygraph.dropout(x1, dropout_prob=0.5, is_test=True)

    x2 = fluid.dygraph.to_variable(data2)
    out2_1 = fluid.dygraph.dropout(x2, dropout_prob=0.5, \
                    dropout_implementation='upscale_in_train')
    out2_2 = fluid.dygraph.dropout(x2, dropout_prob=0.5, \
                    dropout_implementation='upscale_in_train', is_test=True)
      print('x1 {}, \n out1_1 \n {}, \n out1_2 \n {}'.format(data1, out1_1.numpy(),  out1_2.
numpy()))
    print('x2 {}, \n out2_1 \n {}, \n out2_2 \n {}'.format(data2, out2_1.numpy(),  out2_2.
numpy()))
```

3.2.6 作业

计算下面网络层的输出数据和参数的形状。网络结构定义如以下代码所示，输入数据形状是[10,3,224,224]，请分别计算每一层的输出数据形状，以及各层包含的参数形状。

```
# 定义 SimpleNet 网络结构
import paddle
import paddle.fluid as fluid
from paddle.fluid.dygraph.nn import Conv2D, Pool2D, Linear
class SimpleNet(fluid.dygraph.Layer):
    def __init__(self, num_classes=1):
        #super(SimpleNet, self).__init__(name_scope)
        self.conv1 = Conv2D(num_channels=3, num_filters=6, filter_size=5, stride=1,
padding=2, act='relu')
        self.pool1 = Pool2D(pool_size=2, pool_stride=2, pool_type='max')
        self.conv2 = Conv2D(num_channels=6, num_filters=16, filter_size=5, stride=1,
padding=2, act='relu')
        self.pool2 = Pool2D(pool_size=2, pool_stride=2, pool_type='max')
        self.fc1 = Linear(input_dim=50176, output_dim=64, act='sigmoid')
        self.fc2 = Linear(input_dim=64, output_dim=num_classes)
```

```
    def forward(self, x):
        x = self.conv1(x)
        x = self.pool1(x)
        x = self.conv2(x)
        x = self.pool2(x)
        x = fluid.layers.reshape(x, [x.shape[0], -1])
        x = self.fc1(x)
        x = self.fc2(x)
        return x
```

提示，第一层卷积 conv1，各项参数如下：

$$C_{in}=3,\quad C_{out}=6,\quad k_h=k_w=5,\quad p_h=p_w=2,\quad \text{stride}=1$$

则卷积核权重参数 w 的形状是：$[C_{out},C_{in},k_h,K_w]=[6,3,5,5]$，个数为

$$6\times3\times5\times5=450$$

偏置参数 b 的形状是：$[C_{out}]$，偏置参数的个数是 6。

输出特征图的大小是：

$$H_{out}=224+2\times2-5+1=224,\quad W_{out}=224+2\times2-5+1=224$$

输出特征图的形状是

$$[N,C_{out},H_{out},W_{out}]=[10,6,224,224]$$

请将表格 3.1 补充完整：

表 3.1 特征图数据

名　　称	w 形状	w 参数个数	b 形状	b 参数个数	输 出 形 状
conv1	[6,3,5,5]	450	[6]	6	[10, 6, 224, 224]
pool1	无	无	无	无	[10, 6, 112, 112]
conv2					
pool2					
fc1					
fc2					

3.3 图像分类

3.3.1 概述

图像分类是根据图像的语义信息对不同类别图像进行区分，是计算机视觉的核心，是物体检测、图像分割、物体跟踪、行为分析、人脸识别等其他高层次视觉任务的基础。图像分类在许多领域都有着广泛的应用，例如：安防领域的人脸识别和智能视频分析、交通领域的交通场景识别、互联网领域基于内容的图像检索和相册自动归类、医学领域的图像识别等。

上一节介绍了卷积神经网络常用的一些基本模块，本节将基于眼疾分类数据集 iChallenge-PM，对图像分类领域的经典卷积神经网络进行剖析，介绍如何应用这些基础模块构建卷积神经网络，解决图像分类问题。涵盖如下卷积神经网络：

(1) LeNet：Yan LeCun 等人于 1998 年第一次将卷积神经网络应用到图像分类任务上，在手写数字识别任务上取得了巨大成功。

(2) AlexNet：Alex Krizhevsky 等人在 2012 年提出了 AlexNet，并应用在大尺寸图片数据集 ImageNet 上，获得了 2012 年 ImageNet 比赛(ImageNet Large Scale Visual Recognition Challenge，ILSVRC)冠军。

(3) VGG：Simonyan 和 Zisserman 于 2014 年提出了 VGG 网络结构，是当前最流行的卷积神经网络之一，由于其结构简单、应用性极强而深受研究者欢迎。

(4) GoogLeNet：Christian Szegedy 等人在 2014 提出了 GoogLeNet，并取得了 2014 年 ImageNet 比赛冠军。

(5) ResNet：Kaiming He 等人在 2015 年提出了 ResNet，通过引入残差模块加深网络层数，在 ImagNet 数据集上的识别错误率降低到 3.6%，超越了人眼识别水平。ResNet 的设计思想深刻影响了后来的深度神经网络的设计。

3.3.2 LeNet

LeNet 是最早的卷积神经网络之一。1998 年，Yan LeCun 第一次将 LeNet 卷积神经网络应用到图像分类上，在手写数字识别任务中取得了巨大成功。LeNet 通过连续使用卷积和池化层的组合提取图像特征，其架构如图 3.21 所示，这里展示的是作者论文中的 LeNet-5 模型：

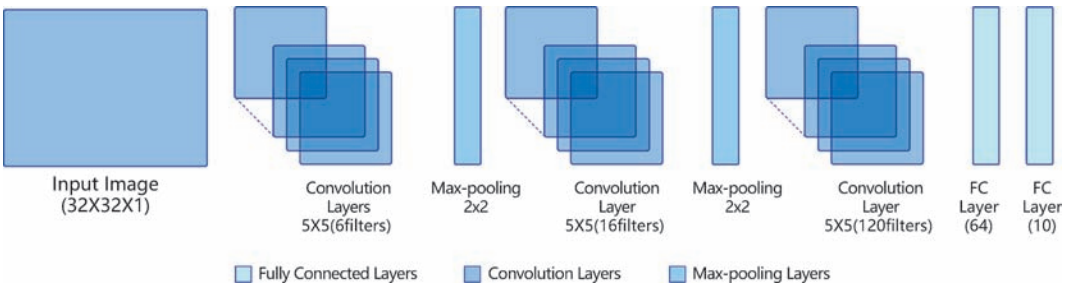


图 3.21 LeNet 模型网络结构示意图

(1) 第一模块：包含 5×5 的 6 通道卷积和 2×2 的池化。卷积提取图像中包含的特征模式(激活函数使用 Sigmoid)，图像尺寸从 32 减小到 28。经过池化层可以降低输出特征图对空间位置的敏感性，图像尺寸减到 14。

(2) 第二模块：和第一模块尺寸相同，通道数由 6 增加为 16。卷积操作使图像尺寸减小到 10，经过池化后变成 5。

(3) 第三模块：包含 5×5 的 120 通道卷积。卷积之后的图像尺寸减小到 1，但是通道数增加为 120。将经过第 3 次卷积提取到的特征图输入到全连接层。第一个全连接层的输出神经元的个数是 64，第二个全连接层的输出神经元个数是分类标签的类别数，对于手写数字识别其大小是 10。然后使用 Softmax 激活函数即可计算出每个类别的预测概率。

提示：

卷积层的输出特征图如何当作全连接层的输入使用呢？

卷积层的输出数据格式是$[N,C,H,W]$，在输入全连接层的时候，会自动将数据拉平，

也就是对每个样本，自动将其转化为长度为 K 的向量。

其中 $K=C\times H\times W$，一个 mini-batch 的数据维度变成了 $N\times K$ 的二维向量。

1. LeNet 在手写数字识别上的应用

LeNet 网络的实现代码如下：

```
# 导入需要的包
import paddle
import paddle.fluid as fluid
import numpy as np
from paddle.fluid.dygraph.nn import Conv2D, Pool2D, Linear

# 定义 LeNet 网络结构
class LeNet(fluid.dygraph.Layer):
    def __init__(self, num_classes=1):
        super(LeNet, self).__init__()

        # 创建卷积和池化层块，每个卷积层使用 Sigmoid 激活函数，后面跟着一个 2×2 的池化
        self.conv1 = Conv2D(num_channels=1, num_filters=6, filter_size=5, act='sigmoid')
        self.pool1 = Pool2D(pool_size=2, pool_stride=2, pool_type='max')
        self.conv2 = Conv2D(num_channels=6, num_filters=16, filter_size=5, act='sigmoid')
        self.pool2 = Pool2D(pool_size=2, pool_stride=2, pool_type='max')
        # 创建第 3 个卷积层
        self.conv3 = Conv2D(num_channels=16, num_filters=120, filter_size=4, act='sigmoid')
        # 创建全连接层，第一个全连接层的输出神经元个数为 64，第二个全连接层输出神经元个数为分类标签的类别数
        self.fc1 = Linear(input_dim=120, output_dim=64, act='sigmoid')
        self.fc2 = Linear(input_dim=64, output_dim=num_classes)
    # 网络的前向计算过程
    def forward(self, x):
        x = self.conv1(x)
        x = self.pool1(x)
        x = self.conv2(x)
        x = self.pool2(x)
        x = self.conv3(x)
        x = fluid.layers.reshape(x, [x.shape[0], -1])
        x = self.fc1(x)
        x = self.fc2(x)
        return x
```

使用随机数作为输入，查看经过 LeNet-5 的每一层作用之后输出数据的形状。

```
# 输入数据形状是 [N, 1, H, W]
# 这里用 np.random 创建一个随机数组作为输入数据
x = np.random.randn(*[3,1,28,28])
x = x.astype('float32')
with fluid.dygraph.guard():
    # 创建 LeNet 类的实例，指定模型名称和分类的类别数目
    m = LeNet(num_classes=10)
```

```
    # 通过调用 LeNet 从基类继承的 sublayers()函数,
    # 查看 LeNet 中所包含的子层
    print(m.sublayers())
    x = fluid.dygraph.to_variable(x)
    for item in m.sublayers():
        # item 是 LeNet 类中的一个子层
        # 查看经过子层之后的输出数据形状
        try:
            x = item(x)
        except:
            x = fluid.layers.reshape(x, [x.shape[0], -1])
            x = item(x)
        if len(item.parameters()) == 2:
            # 查看卷积和全连接层的数据和参数的形状,
            # 其中 item.parameters()[0]是权重参数 w,item.parameters()[1]是偏置参数 b
            print(item.full_name(), x.shape, item.parameters()[0].shape, item.parameters()
[1].shape)
        else:
            # 池化层没有参数
            print(item.full_name(), x.shape)
```

在 LeNet 上完成手写数字的识别。

```
import os
import random
import paddle
import paddle.fluid as fluid
import numpy as np

# 定义训练过程
def train(model):
    print('start training ... ')
    model.train()
    epoch_num = 5
    opt = fluid.optimizer.Momentum(learning_rate=0.001, momentum=0.9, parameter_list=
model.parameters())
    # 使用 Paddle 自带的数据读取器
    train_loader = paddle.batch(paddle.dataset.mnist.train(), batch_size=10)
    valid_loader = paddle.batch(paddle.dataset.mnist.test(), batch_size=10)
    for epoch in range(epoch_num):
        for batch_id, data in enumerate(train_loader()):
            # 调整输入数据形状和类型
            x_data = np.array([item[0] for item in data], dtype='float32').reshape(-1, 1,
28, 28)
            y_data = np.array([item[1] for item in data], dtype='int64').reshape(-1, 1)
            # 将 numpy.ndarray 转化成 Tensor
            img = fluid.dygraph.to_variable(x_data)
            label = fluid.dygraph.to_variable(y_data)
            # 计算模型输出
            logits = model(img)
            # 计算损失函数
            loss = fluid.layers.softmax_with_cross_entropy(logits, label)
```

```
                avg_loss = fluid.layers.mean(loss)
                if batch_id % 1000 == 0:
                    print("epoch: {}, batch_id: {}, loss is: {}".format(epoch, batch_id, avg_
loss.numpy()))
                avg_loss.backward()
                opt.minimize(avg_loss)
                model.clear_gradients()

            model.eval()
            accuracies = []
            losses = []
            for batch_id, data in enumerate(valid_loader()):
                # 调整输入数据形状和类型
                x_data = np.array([item[0] for item in data], dtype='float32').reshape(-1, 1,
28, 28)
                y_data = np.array([item[1] for item in data], dtype='int64').reshape(-1, 1)
                # 将 numpy.ndarray 转化成 Tensor
                img = fluid.dygraph.to_variable(x_data)
                label = fluid.dygraph.to_variable(y_data)
                # 计算模型输出
                logits = model(img)
                pred = fluid.layers.softmax(logits)
                # 计算损失函数
                loss = fluid.layers.softmax_with_cross_entropy(logits, label)
                acc = fluid.layers.accuracy(pred, label)
                accuracies.append(acc.numpy())
                losses.append(loss.numpy())
            print("[validation] accuracy/loss: {}/{}".format(np.mean(accuracies), np.mean
(losses)))
            model.train()

        # 保存模型参数
        fluid.save_dygraph(model.state_dict(), 'mnist')

if __name__ == '__main__':
        # 创建模型
        with fluid.dygraph.guard():
            model = LeNet(num_classes=10)
            #启动训练过程
            train(model)
```

通过运行结果可以看出，LeNet 在手写数字识别 MNIST 验证数据集上的准确率高达 92%以上。那么对于其他数据集效果如何呢？我们通过眼疾识别数据集 iChallenge-PM 验证一下。

2. LeNet 在眼疾识别上的应用

眼疾识别数据集 iChallenge-PM 是百度大脑和中山大学中山眼科中心联合举办的 iChallenge 比赛中，提供的关于病理性近视（Pathologic Myopia，PM）的医疗类数据集，包含 1200 个受试者的眼底视网膜图片，训练、验证和测试数据集各 400 张。下面我们详细介绍 LeNet 在 iChallenge-PM 上的训练过程。

说明：

如今近视已经成为困扰人们健康的一项全球性负担，在近视人群中，有超过35%的人患有重度近视。近视将会导致眼睛的光轴被拉长，有可能引起视网膜或者络网膜的病变。随着近视度数的不断加深，高度近视有可能引发病理性病变，这将会导致以下几种症状：视网膜或者络网膜发生退化、视盘区域萎缩、漆裂样纹损害、Fuchs斑等。因此，及早发现近视患者眼睛的病变并采取治疗，显得非常重要。

1）数据集准备

说明：

请读者在本书的配套教程“零基础实践深度学习课程—图像分类”章节中获取iChallenge-PM数据集。

./data/data19065目录包括如下三个文件，代码解压缩后存放在./work/palm目录下。

- training.zip：包含训练中的图片和标签。
- validation.zip：包含验证集的图片。
- valid_gt.zip：包含验证集的标签。

```
# 初次运行时删除注释，以便解压文件
# 如果已经解压，则不需要运行此段代码，否则文件已经存在解压会报错

# !unzip -o -q -d /home/aistudio/work/palm /home/aistudio/data/data19065/training.zip
# %cd /home/aistudio/work/palm/PALM-Training400/
# !unzip -o -q PALM-Training400.zip
# !unzip -o -q -d /home/aistudio/work/palm /home/aistudio/data/data19065/validation.zip
# !unzip -o -q -d /home/aistudio/work/palm /home/aistudio/data/data19065/valid_gt.zip
```

注意：

valid_gt.zip文件解压缩之后，需要将./work/palm/PALM-Validation-GT/目录下的PM_Label_and_Fovea_Location.xlsx文件转存成csv格式，在AI Studio本节代码示例中已经将文件转成labels.csv格式，无需读者操作。

2）查看数据集图片

iChallenge-PM中既有病理性近视患者的眼底图片，也有非病理性近视患者的图片，命名规则如下：

（1）病理性近视（PM）：文件名以P开头。

（2）非病理性近视（non-PM）：

- 高度近视（high myopia）：文件名以H开头。
- 正常眼睛（normal）：文件名以N开头。

我们将病理性患者的图片作为正样本，标签为1；非病理性患者的图片作为负样本，标签为0。从数据集中选取两张图片，通过LeNet提取特征，构建分类器，对正负样本进行分

类，并将图片显示出来（见图 3.22）。代码如下所示：

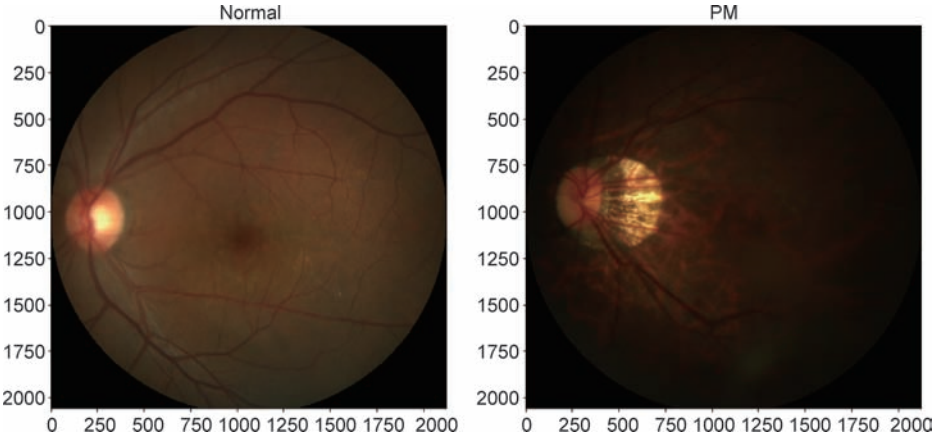


图 3.22 正常眼睛和病理性近视

```
import os
import numpy as np
import matplotlib.pyplot as plt
% matplotlib inline
from PIL import Image

DATADIR = '/home/aistudio/work/palm/PALM - Training400/PALM - Training400'
# 文件名以 N 开头的是正常眼底图片，以 P 开头的是病变眼底图片
file1 = 'N0012.jpg'
file2 = 'P0095.jpg'

# 读取图片
img1 = Image.open(os.path.join(DATADIR, file1))
img1 = np.array(img1)
img2 = Image.open(os.path.join(DATADIR, file2))
img2 = np.array(img2)

# 画出读取的图片
plt.figure(figsize = (16, 8))
f = plt.subplot(121)
f.set_title('Normal', fontsize = 20)
plt.imshow(img1)
f = plt.subplot(122)
f.set_title('PM', fontsize = 20)
plt.imshow(img2)
plt.show()

# 查看图片形状
img1.shape, img2.shape
```

```
((2056, 2124, 3), (2056, 2124, 3))
```

3）定义数据读取器

使用 OpenCV 从磁盘读入图片，将每张图缩放到 224×224 大小，并且将像素值调整到[-1,1]上，代码如下所示：

```
import cv2
import random
import numpy as np

# 对读入的图像数据进行预处理
def transform_img(img):
    # 将图片尺寸缩放到 224×224
    img = cv2.resize(img, (224, 224))
    # 读入的图像数据格式是[H, W, C]
    # 使用转置操作将其变成[C, H, W]
    img = np.transpose(img, (2,0,1))
    img = img.astype('float32')
    # 将数据范围调整到[-1.0, 1.0]之间
    img = img / 255.
    img = img * 2.0 - 1.0
    return img
```

读取训练数据。

```
# 定义训练集数据读取器
def data_loader(datadir, batch_size=10, mode = 'train'):
    # 将 datadir 目录下的文件列出来,每条文件都要读入
    filenames = os.listdir(datadir)
    def reader():
        if mode == 'train':
            # 训练时随机打乱数据顺序
            random.shuffle(filenames)
        batch_imgs = []
        batch_labels = []
        for name in filenames:
            filepath = os.path.join(datadir, name)
            img = cv2.imread(filepath)
            img = transform_img(img)
            if name[0] == 'H' or name[0] == 'N':
                # H 开头的文件名表示高度近视,N 开头的文件名表示正常视力
                # 高度近视和正常视力的样本,都不是病理性的,属于负样本,标签为 0
                label = 0
            elif name[0] == 'P':
                # P 开头的是病理性近视,属于正样本,标签为 1
                label = 1
            else:
                raise('Not excepted file name')
            # 每读取一个样本的数据,就将其放入数据列表中
            batch_imgs.append(img)
            batch_labels.append(label)
            if len(batch_imgs) == batch_size:
                # 当数据列表的长度等于 batch_size 的时候,
                # 把这些数据当作一个 mini-batch,并作为数据生成器的一个输出
                imgs_array = np.array(batch_imgs).astype('float32')
                labels_array = np.array(batch_labels).astype('float32').reshape(-1, 1)
                yield imgs_array, labels_array
                batch_imgs = []
```

```
                batch_labels = []

        if len(batch_imgs) > 0:
            # 剩余样本数目不足一个 batch_size 的数据,一起打包成一个 mini-batch
            imgs_array = np.array(batch_imgs).astype('float32')
            labels_array = np.array(batch_labels).astype('float32').reshape(-1, 1)
            yield imgs_array, labels_array

    return reader
```

读取验证集数据。

```
# 定义验证集数据读取器
def valid_data_loader(datadir, csvfile, batch_size=10, mode='valid'):
    # 训练集读取时通过文件名来确定样本标签,验证集则通过 csvfile 来读取每个图片对应的
标签
    # 请查看解压后的验证集标签数据,观察 csvfile 文件里面所包含的内容
    # csvfile 文件所包含的内容格式如下,每一行代表一个样本,
    # 其中第一列是图片 id,第二列是文件名,第三列是图片标签,
    # 第四列和第五列是 Fovea 的坐标,与分类任务无关
    # ID,imgName,Label,Fovea_X,Fovea_Y
    # 1,V0001.jpg,0,1157.74,1019.87
    # 2,V0002.jpg,1,1285.82,1080.47
    # 打开包含验证集标签的 csvfile,并读入其中的内容
    filelists = open(csvfile).readlines()
    def reader():
        batch_imgs = []
        batch_labels = []
        for line in filelists[1:]:
            line = line.strip().split(',')
            name = line[1]
            label = int(line[2])
            # 根据图片文件名加载图片,并对图像数据作预处理
            filepath = os.path.join(datadir, name)
            img = cv2.imread(filepath)
            img = transform_img(img)
            # 每读取一个样本的数据,就将其放入数据列表中
            batch_imgs.append(img)
            batch_labels.append(label)
            if len(batch_imgs) == batch_size:
                # 当数据列表的长度等于 batch_size 的时候,
                # 把这些数据当作一个 mini-batch,并作为数据生成器的一个输出
                imgs_array = np.array(batch_imgs).astype('float32')
                labels_array = np.array(batch_labels).astype('float32').reshape(-1, 1)
                yield imgs_array, labels_array
                batch_imgs = []
                batch_labels = []

        if len(batch_imgs) > 0:
            # 剩余样本数目不足一个 batch_size 的数据,一起打包成一个 mini-batch
            imgs_array = np.array(batch_imgs).astype('float32')
            labels_array = np.array(batch_labels).astype('float32').reshape(-1, 1)
```

```
            yield imgs_array, labels_array

    return reader
```

查看数据形状。

```
DATADIR = '/home/aistudio/work/palm/PALM-Training400/PALM-Training400'
train_loader = data_loader(DATADIR,
                           batch_size=10, mode='train')
data_reader = train_loader()
data = next(data_reader)
data[0].shape, data[1].shape
```

```
((10, 3, 224, 224), (10, 1))
```

4）启动训练

加载相关类库。

```
# LeNet 识别眼疾图片
import os
import random
import paddle
import paddle.fluid as fluid
import numpy as np

DATADIR = '/home/aistudio/work/palm/PALM-Training400/PALM-Training400'
DATADIR2 = '/home/aistudio/work/palm/PALM-Validation400'
CSVFILE = '/home/aistudio/work/palm/PALM-Validation-GT/labels.csv'
```

定义训练过程。

```
def train(model):
    with fluid.dygraph.guard():
        print('start training ... ')
        model.train()
        epoch_num = 5
        # 定义优化器
        opt = fluid.optimizer.Momentum(learning_rate=0.001, momentum=0.9, parameter_
list=model.parameters())
        # 定义数据读取器,训练数据读取器和验证数据读取器
        train_loader = data_loader(DATADIR, batch_size=10, mode='train')
        valid_loader = valid_data_loader(DATADIR2, CSVFILE)
        for epoch in range(epoch_num):
            for batch_id, data in enumerate(train_loader()):
                x_data, y_data = data
                img = fluid.dygraph.to_variable(x_data)
                label = fluid.dygraph.to_variable(y_data)
                # 运行模型前向计算,得到预测值
                logits = model(img)
                # 进行 loss 计算
                loss = fluid.layers.sigmoid_cross_entropy_with_logits(logits, label)
```

```
                avg_loss = fluid.layers.mean(loss)

                if batch_id % 10 == 0:
                    print("epoch: {}, batch_id: {}, loss is: {}".format(epoch, batch_id,
avg_loss.numpy()))
                # 反向传播,更新权重,清除梯度
                avg_loss.backward()
                opt.minimize(avg_loss)
                model.clear_gradients()

            model.eval()
            accuracies = []
            losses = []
            for batch_id, data in enumerate(valid_loader()):
                x_data, y_data = data
                img = fluid.dygraph.to_variable(x_data)
                label = fluid.dygraph.to_variable(y_data)
                # 运行模型前向计算,得到预测值
                logits = model(img)
                # 二分类,sigmoid计算后的结果以0.5为阈值分两个类别
                # 计算sigmoid后的预测概率,进行loss计算
                pred = fluid.layers.sigmoid(logits)
                loss = fluid.layers.sigmoid_cross_entropy_with_logits(logits, label)
                # 计算预测概率小于0.5的类别
                pred2 = pred * (-1.0) + 1.0
                # 得到两个类别的预测概率,并沿第一个维度级联
                pred = fluid.layers.concat([pred2, pred], axis=1)
                acc = fluid.layers.accuracy(pred, fluid.layers.cast(label, dtype='int64'))
                accuracies.append(acc.numpy())
                losses.append(loss.numpy())
            print("[validation] accuracy/loss: {}/{}".format(np.mean(accuracies), np.mean
(losses)))
            model.train()

        # save params of model
        fluid.save_dygraph(model.state_dict(), 'mnist')
        # save optimizer state
        fluid.save_dygraph(opt.state_dict(), 'mnist')
```

定义评估过程。

```
def evaluation(model, params_file_path):
    with fluid.dygraph.guard():
        print('start evaluation .......')
        #加载模型参数
        model_state_dict, _ = fluid.load_dygraph(params_file_path)
        model.load_dict(model_state_dict)

        model.eval()
        eval_loader = data_loader(DATADIR,
                                  batch_size=10 , mode='eval')
```

```
        acc_set = []
        avg_loss_set = []
        for batch_id, data in enumerate(eval_loader()):
            x_data, y_data = data
            img = fluid.dygraph.to_variable(x_data)
            label = fluid.dygraph.to_variable(y_data)
            y_data = y_data.astype(np.int64)
            label_64 = fluid.dygraph.to_variable(y_data)
            # 计算预测和精度
            prediction, acc = model(img, label)
            # 计算损失函数值
            loss = fluid.layers.cross_entropy(input=prediction, label=label)
            avg_loss = fluid.layers.mean(loss)
            acc_set.append(float(acc.numpy()))
            avg_loss_set.append(float(avg_loss.numpy()))
        # 求平均精度
        acc_val_mean = np.array(acc_set).mean()
        avg_loss_val_mean = np.array(avg_loss_set).mean()

        print('loss={}, acc={}'.format(avg_loss_val_mean, acc_val_mean))
```

网络训练。

```
import paddle
import paddle.fluid as fluid
import numpy as np
from paddle.fluid.dygraph.nn import Conv2D, Pool2D, Linear

# 定义 LeNet 网络结构
class LeNet(fluid.dygraph.Layer):
    def __init__(self, num_classes=1):
        super(LeNet, self).__init__()

        # 创建卷积和池化层块,每个卷积层使用 Sigmoid 激活函数,后面跟着一个 2×2 的池化
        self.conv1 = Conv2D(num_channels=3, num_filters=6, filter_size=5, act='sigmoid')
        self.pool1 = Pool2D(pool_size=2, pool_stride=2, pool_type='max')
        self.conv2 = Conv2D(num_channels=6, num_filters=16, filter_size=5, act=
'sigmoid')
        self.pool2 = Pool2D(pool_size=2, pool_stride=2, pool_type='max')
        # 创建第 3 个卷积层
        self.conv3 = Conv2D(num_channels=16, num_filters=120, filter_size=4, act=
'sigmoid')
        # 创建全连接层,第一个全连接层的输出神经元个数为 64,第二个全连接层的输出神经元
个数为分类标签的类别数
        self.fc1 = Linear(input_dim=300000, output_dim=64, act='sigmoid')
        self.fc2 = Linear(input_dim=64, output_dim=num_classes)
    # 网络的前向计算过程
    def forward(self, x):
        x = self.conv1(x)
        x = self.pool1(x)
        x = self.conv2(x)
        x = self.pool2(x)
```

```
            x = self.conv3(x)
            x = fluid.layers.reshape(x, [x.shape[0], -1])
            x = self.fc1(x)
            x = self.fc2(x)
            return x

    if __name__ == '__main__':
        # 创建模型
        with fluid.dygraph.guard():
            model = LeNet(num_classes=1)

        train(model)
```

通过运行结果可以看出，在眼疾筛查数据集 iChallenge-PM 上，LeNet 的 loss 很难下降，模型没有收敛。这是因为 MNIST 数据集的图片尺寸比较小(28×28)，但是眼疾筛查数据集图片尺寸比较大(原始图片尺寸约为 2000×2000，经过缩放之后变成 224×224，LeNet 模型很难进行有效分类。这说明在图片尺寸比较大时，LeNet 在图像分类任务上存在局限性。

3.3.3 AlexNet

通过上面的实际训练可以看到，虽然 LeNet 在手写数字识别数据集上取得了很好的结果，但在更大的数据集上表现却并不好。自从 1998 年 LeNet 问世以来，接下来十几年的时间里，神经网络并没有在计算机视觉领域取得很好的结果，反而一度被其他算法超越，原因主要有两方面，一是神经网络的计算比较复杂，对当时计算机的算力来说，训练神经网络是件非常耗时的事情；另一方面，当时还没有专门针对神经网络做算法和训练技巧的优化，神经网络的收敛是件非常困难的事情。

随着技术的进步和发展，计算机的算力越来越强大，尤其是在 GPU 并行计算能力的推动下，复杂神经网络的计算也变得更加容易实施。另一方面，互联网上涌现出越来越多的数据，极大地丰富了数据库。同时也有越来越多的研究人员开始专门针对神经网络做算法和模型的优化，Alex Krizhevsky 等人提出的 AlexNet 以很大优势获得了 2012 年 ImageNet 比赛的冠军。这一成果极大地激发了产业界对神经网络的兴趣，开创了使用深度神经网络解决图像问题的途径，随后也在这一领域涌现出越来越多的优秀成果。

AlexNet 与 LeNet 相比，具有更深的网络结构，包含 5 层卷积和 3 层全连接，同时使用了如下三种方法改进模型的训练过程：

(1) 数据增强：深度学习中常用的一种处理方式，通过对训练随机加一些变化，比如平移、缩放、裁剪、旋转、翻转或者增减亮度等，产生一系列跟原始图片相似但又不完全相同的样本，从而扩大训练数据集。通过这种方式，可以随机改变训练样本，避免模型过度依赖于某些属性，能从一定程度上抑制过拟合。

(2) 使用 Dropout 抑制过拟合。

(3) 使用 ReLU 激活函数减少梯度消失现象。

AlexNet 的具体结构如图 3.23 所示。

AlexNet 在眼疾筛查数据集 iChallenge-PM 上具体实现的代码如下所示：

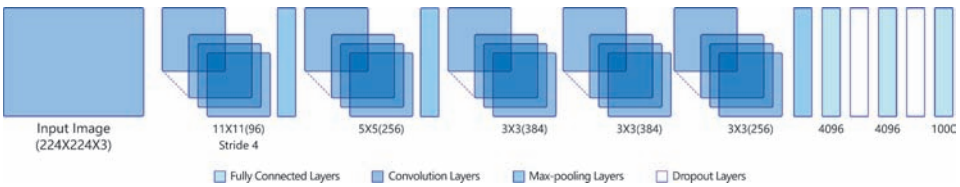


图 3.23　AlexNet 模型网络结构示意图

```
import paddle
import paddle.fluid as fluid
import numpy as np
from paddle.fluid.dygraph.nn import Conv2D, Pool2D, Linear

# 定义 AlexNet 网络结构
class AlexNet(fluid.dygraph.Layer):
    def __init__(self, num_classes=1):
        super(AlexNet, self).__init__()

        # AlexNet 与 LeNet 一样也会同时使用卷积和池化层提取图像特征
        # 与 LeNet 不同的是激活函数换成了 relu
        self.conv1 = Conv2D(num_channels=3, num_filters=96, filter_size=11, stride=4,
padding=5, act='relu')
        self.pool1 = Pool2D(pool_size=2, pool_stride=2, pool_type='max')
        self.conv2 = Conv2D(num_channels=96, num_filters=256, filter_size=5, stride=
1, padding=2, act='relu')
        self.pool2 = Pool2D(pool_size=2, pool_stride=2, pool_type='max')
        self.conv3 = Conv2D(num_channels=256, num_filters=384, filter_size=3, stride=
1, padding=1, act='relu')
        self.conv4 = Conv2D(num_channels=384, num_filters=384, filter_size=3, stride=
1, padding=1, act='relu')
        self.conv5 = Conv2D(num_channels=384, num_filters=256, filter_size=3, stride=
1, padding=1, act='relu')
        self.pool5 = Pool2D(pool_size=2, pool_stride=2, pool_type='max')

        self.fc1 = Linear(input_dim=12544, output_dim=4096, act='relu')
        self.drop_ratio1 = 0.5
        self.fc2 = Linear(input_dim=4096, output_dim=4096, act='relu')
        self.drop_ratio2 = 0.5
        self.fc3 = Linear(input_dim=4096, output_dim=num_classes)

    def forward(self, x):
        x = self.conv1(x)
        x = self.pool1(x)
        x = self.conv2(x)
        x = self.pool2(x)
        x = self.conv3(x)
        x = self.conv4(x)
        x = self.conv5(x)
        x = self.pool5(x)
        x = fluid.layers.reshape(x, [x.shape[0], -1])
        x = self.fc1(x)
        # 在全连接之后使用 dropout 抑制过拟合
```

```
        x = fluid.layers.dropout(x, self.drop_ratio1)
        x = self.fc2(x)
        # 在全连接之后使用 dropout 抑制过拟合
        x = fluid.layers.dropout(x, self.drop_ratio2)
        x = self.fc3(x)
        return x
```

启动模型训练。

```
with fluid.dygraph.guard():
    model = AlexNet()

train(model)
```

通过运行结果可以发现，在眼疾筛查数据集 iChallenge-PM 上使用 AlexNet，loss 能有效下降，经过 5 个 Epoch 的训练，在验证集上的准确率可以达到 94%左右。

3.3.4 VGG

VGG 是当前最流行的 CNN 模型之一，2014 年由 Simonyan 和 Zisserman 提出，其命名来源于论文作者所在的实验室 Visual Geometry Group。AlexNet 模型通过构造多层网络，取得了较好的效果，但是并没有给出深度神经网络设计的方向。VGG 通过使用一系列大小为 3×3 的小尺寸卷积核和池化层构造深度卷积神经网络，并取得了较好的效果。VGG 模型因为结构简单、应用性极强而广受研究者欢迎，尤其是它的网络结构设计方法，为构建深度神经网络提供了方向。

如图 3.24 所示，是 VGG-16 的网络结构示意图，有 13 层卷积和 3 层全连接层。VGG 网络的设计严格使用 3×3 的卷积层和池化层来提取特征，并在网络的最后面使用三层全连接层，将最后一层全连接层的输出作为分类的预测。在 VGG 中每层卷积将使用 ReLU 作为激活函数，在全连接层之后添加 dropout 来抑制过拟合。使用小的卷积核能够有效地减少参数的个数，使得训练和测试变得更加有效。比如使用两层 3×3 卷积层，可以得到感受野为 5 的特征图，而比使用 5×5 的卷积层需要更少的参数。由于卷积核比较小，可以堆叠更多的卷积层，加深网络的深度，这对于图像分类任务来说是有利的。VGG 模型的成功证明了增加网络的深度，可以更好地学习图像中的特征模式。

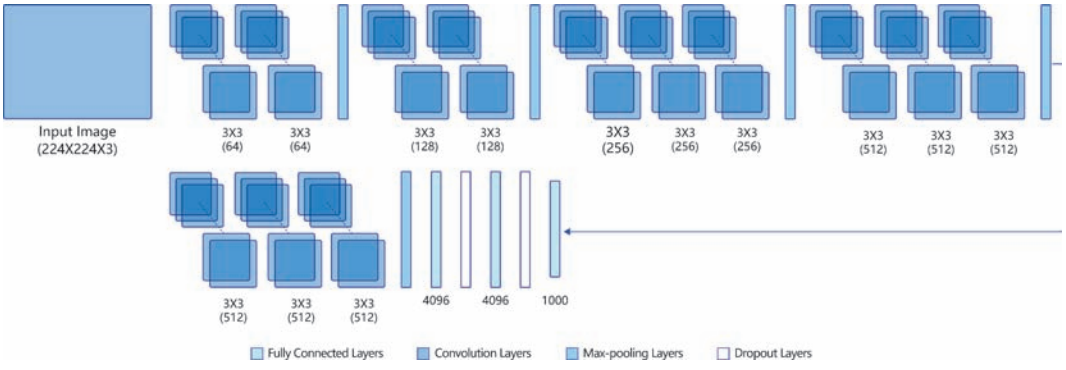


图 3.24 VGG 模型网络结构示意图

VGG 在眼疾识别数据集 iChallenge-PM 上的具体实现如下代码所示：

```
import numpy as np
import paddle
import paddle.fluid as fluid
from paddle.fluid.dygraph.nn import Conv2D, Pool2D, BatchNorm, Linear
from paddle.fluid.dygraph.base import to_variable

# 定义 vgg 块,包含多层卷积和 1 层 2×2 的最大池化层
class vgg_block(fluid.dygraph.Layer):
    def __init__(self, num_convs, in_channels, out_channels):
        """
        num_convs, 卷积层的数目
        num_channels, 卷积层的输出通道数,在同一个 Inception 块内,卷积层输出通道数是一样的
        """
        super(vgg_block, self).__init__()
        self.conv_list = []
        for i in range(num_convs):
            conv_layer = self.add_sublayer('conv_' + str(i), Conv2D(num_channels = in_
channels, num_filters = out_channels, filter_size = 3, padding = 1, act = 'relu'))
            self.conv_list.append(conv_layer)
            in_channels = out_channels
        self.pool = Pool2D(pool_stride = 2, pool_size = 2, pool_type = 'max')
    def forward(self, x):
        for item in self.conv_list:
            x = item(x)
        return self.pool(x)

class VGG(fluid.dygraph.Layer):
    def __init__(self, conv_arch = ((2, 64),
                                    (2, 128), (3, 256), (3, 512), (3, 512))):
        super(VGG, self).__init__()
        self.vgg_blocks = []
        iter_id = 0
        # 添加 vgg_block
        # 这里一共 5 个 vgg_block,每个 block 里面的卷积层数目和输出通道数由 conv_arch 指定
        in_channels = [3, 64, 128, 256, 512, 512]
        for (num_convs, num_channels) in conv_arch:
            block = self.add_sublayer('block_' + str(iter_id),
                    vgg_block(num_convs, in_channels = in_channels[iter_id],
                              out_channels = num_channels))
            self.vgg_blocks.append(block)
            iter_id += 1
        self.fc1 = Linear(input_dim = 512 * 7 * 7, output_dim = 4096,
                          act = 'relu')
        self.drop1_ratio = 0.5
        self.fc2 = Linear(input_dim = 4096, output_dim = 4096,
                          act = 'relu')
        self.drop2_ratio = 0.5
        self.fc3 = Linear(input_dim = 4096, output_dim = 1)

    def forward(self, x):
        for item in self.vgg_blocks:
            x = item(x)
        x = fluid.layers.reshape(x, [x.shape[0], -1])
        x = fluid.layers.dropout(self.fc1(x), self.drop1_ratio)
        x = fluid.layers.dropout(self.fc2(x), self.drop2_ratio)
```

```
        x = self.fc3(x)
        return x

with fluid.dygraph.guard():
    model = VGG()

train(model)
```

通过运行结果可以发现，在眼疾筛查数据集 iChallenge-PM 上使用 VGG，loss 能有效的下降，经过 5 个 epoch 的训练，在验证集上的准确率可以达到 94%左右。

3.3.5 GoogLeNet

GoogLeNet 是 2014 年 ImageNet 比赛的冠军，它的主要特点是网络不仅有深度，还在横向上具有“宽度”。由于图像信息在空间尺寸上的巨大差异，如何选择合适的卷积核大小来提取特征就显得比较困难了。空间分布范围更广的图像信息适合用较大的卷积核来提取其特征，而空间分布范围较小的图像信息则适合用较小的卷积核来提取其特征。为了解决这个问题，GoogLeNet 提出了一种被称为 Inception 模块的方案，如图 3.25 所示。

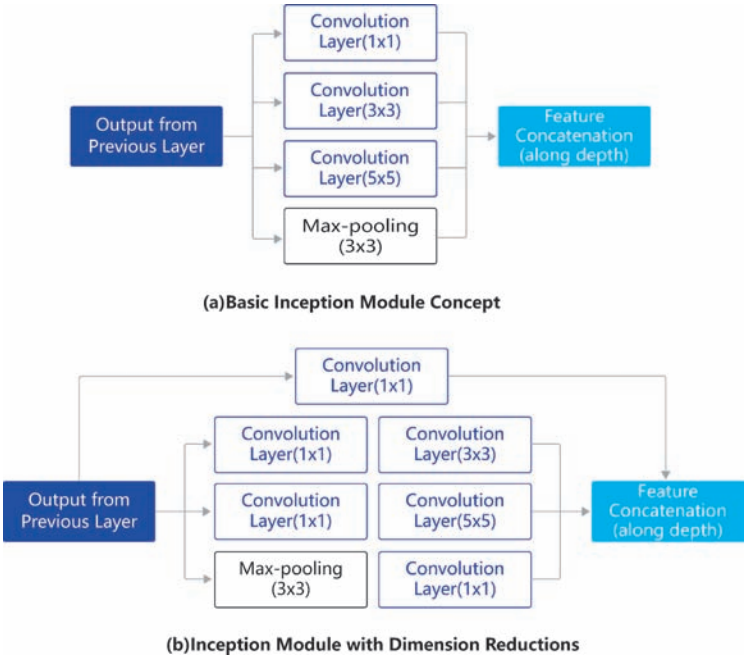


图 3.25 Inception 模块结构示意图

说明：

- Google 的研究人员为了向 LeNet 致敬，特地将模型命名为 GoogLeNet。
- Inception 一词来源于电影《盗梦空间》(Inception)。

图 3.25(a)是 Inception 模块的设计思想，使用 3 个不同大小的卷积核对输入图片进行卷积操作，并附加最大池化，将这 4 个操作的输出沿着通道这一维度进行拼接，构成的输出

特征图将会包含经过不同大小的卷积核提取出来的特征。Inception 模块采用多通路(multi-path)的设计形式,每个支路使用不同大小的卷积核,最终输出特征图的通道数是每个支路输出通道数的总和,这将会导致输出通道数变得很大,尤其是使用多个 Inception 模块串联操作的时候,模型参数量会变得非常巨大。为了减小参数量,Inception 模块使用了图 3.25(b)中的设计方式,在每个 3×3 和 5×5 的卷积层之前,增加 1×1 的卷积层来控制输出通道数;在最大池化层后面增加 1×1 卷积层减小输出通道数。基于这一设计思想,形成了图 3.25(b)中所示的结构。下面这段程序是 Inception 块的具体实现方式,可以对照图 3.25(b)和代码一起阅读。

提示:

可能有读者会问,经过 3×3 的最大池化之后图像尺寸不会减小吗,为什么还能跟另外 3 个卷积输出的特征图进行拼接?这是因为池化操作可以指定窗口大小 $k_h=k_w=3$,pool_stride=1 和 pool_padding=1,输出特征图尺寸可以保持不变。

Inception 模块的具体实现如以下代码所示:

```
class Inception(fluid.dygraph.Layer):
    def __init__(self, c1, c2, c3, c4, **kwargs):
        '''
        Inception 模块的实现代码,
        c1,图 3.25(b)中第一条支路 1×1 卷积的输出通道数,数据类型是整数
        c2,图 3.25(b)中第二条支路卷积的输出通道数,数据类型是 tuple 或 list,
               其中 c2[0]是 1×1 卷积的输出通道数,c2[1]是 3×3
        c3,图 3.25(b)中第三条支路卷积的输出通道数,数据类型是 tuple 或 list,
               其中 c3[0]是 1×1 卷积的输出通道数,c3[1]是 3×3
        c4,图 3.25(b)中第一条支路 1×1 卷积的输出通道数,数据类型是整数
        '''
        super(Inception, self).__init__()
        # 依次创建 Inception 块每条支路上使用到的操作
        self.p1_1 = Conv2D(num_filters=c1,
                           filter_size=1, act='relu')
        self.p2_1 = Conv2D(num_filters=c2[0],
                           filter_size=1, act='relu')
        self.p2_2 = Conv2D(num_filters=c2[1],
                           filter_size=3, padding=1, act='relu')
        self.p3_1 = Conv2D(num_filters=c3[0],
                           filter_size=1, act='relu')
        self.p3_2 = Conv2D(num_filters=c3[1],
                           filter_size=5, padding=2, act='relu')
        self.p4_1 = Pool2D(pool_size=3,
                           pool_stride=1,  pool_padding=1,
                           pool_type='max')
        self.p4_2 = Conv2D(num_filters=c4,
                           filter_size=1, act='relu')

    def forward(self, x):
        # 支路 1 只包含一个 1×1 卷积
        p1 = self.p1_1(x)
        # 支路 2 包含 1×1 卷积 + 3×3 卷积
        p2 = self.p2_2(self.p2_1(x))
```

```
        # 支路 3 包含 1×1 卷积 + 5x5 卷积
        p3 = self.p3_2(self.p3_1(x))
        # 支路 4 包含 最大池化和 1×1 卷积
        p4 = self.p4_2(self.p4_1(x))
        # 将每个支路的输出特征图拼接在一起作为最终的输出结果
        return fluid.layers.concat([p1, p2, p3, p4], axis=1)
```

GoogLeNet 的架构如图 3.21 所示，在主体卷积部分中使用 5 个模块(block)，每个模块之间使用步幅为 2 的 3×3 最大池化层来减小输出高宽。

(1) 第一模块使用一个 64 通道的 7×7 卷积层。

(2) 第二模块使用 2 个卷积层：首先是 64 通道的 1×1 卷积层，然后是将通道增大 3 倍的 3×3 卷积层。

(3) 第三模块串联 2 个完整的 Inception 块。

(4) 第四模块串联了 5 个 Inception 块。

(5) 第五模块串联了 2 个 Inception 块。

(6) 第六模块的前面紧跟输出层，使用全局平均池化层来将每个通道的高和宽变成 1，最后接上一个输出个数为标签类别数的全连接层。

说明：

本书在原作者的论文中添加了图 3.26 所示的 Softmax1 和 Softmax2 两个辅助分类器，如图 3.26 所示，训练时将三个分类器的损失函数进行加权求和，以缓解梯度消失现象。这里的程序作了简化，没有加入辅助分类器。

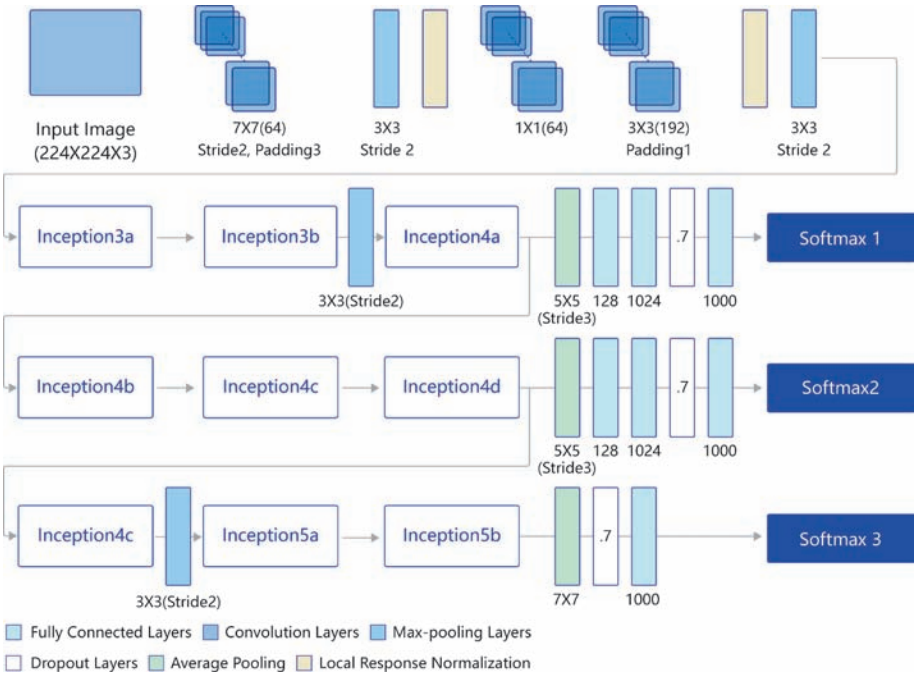


图 3.26 GoogLeNet 模型网络结构示意图

GoogLeNet 的具体实现如下代码所示：

```
import numpy as np
import paddle
import paddle.fluid as fluid
from paddle.fluid.layer_helper import LayerHelper
from paddle.fluid.dygraph.nn import Conv2D, Pool2D, BatchNorm, Linear
from paddle.fluid.dygraph.base import to_variable

class GoogLeNet(fluid.dygraph.Layer):
    def __init__(self):
        super(GoogLeNet, self).__init__()
        # GoogLeNet 包含五个模块,每个模块后面紧跟一个池化层
        # 第一个模块包含 1 个卷积层
        self.conv1 = Conv2D(num_channels=3, num_filters=64, filter_size=7,
                            padding=3, act='relu')
        # 3×3 最大池化
        self.pool1 = Pool2D(pool_size=3, pool_stride=2,
                            pool_padding=1, pool_type='max')
        # 第二个模块包含 2 个卷积层
        self.conv2_1 = Conv2D(num_channels=64, num_filters=64,
                              filter_size=1, act='relu')
        self.conv2_2 = Conv2D(num_channels=64, num_filters=192,
                              filter_size=3, padding=1, act='relu')
        # 3×3 最大池化
        self.pool2 = Pool2D(pool_size=3, pool_stride=2,
                            pool_padding=1, pool_type='max')
        # 第三个模块包含 2 个 Inception 块
        self.block3_1 = Inception(192, 64, (96, 128), (16, 32), 32)
        self.block3_2 = Inception(256, 128, (128, 192), (32, 96), 64)
        # 3×3 最大池化
        self.pool3 = Pool2D(pool_size=3, pool_stride=2,
                            pool_padding=1, pool_type='max')
        # 第四个模块包含 5 个 Inception 块
        self.block4_1 = Inception(480, 192, (96, 208), (16, 48), 64)
        self.block4_2 = Inception(512, 160, (112, 224), (24, 64), 64)
        self.block4_3 = Inception(512, 128, (128, 256), (24, 64), 64)
        self.block4_4 = Inception(512, 112, (144, 288), (32, 64), 64)
        self.block4_5 = Inception(528, 256, (160, 320), (32, 128), 128)
        # 3×3 最大池化
        self.pool4 = Pool2D(pool_size=3, pool_stride=2,
                            pool_padding=1, pool_type='max')
        # 第五个模块包含 2 个 Inception 块
        self.block5_1 = Inception(832, 256, (160, 320), (32, 128), 128)
        self.block5_2 = Inception(832, 384, (192, 384), (48, 128), 128)
        # 全局池化,尺寸用的是 global_pooling,pool_stride 不起作用
        self.pool5 = Pool2D(pool_stride=1,
                            global_pooling=True, pool_type='avg')
        self.fc = Linear(input_dim=1024, output_dim=1, act=None)

    def forward(self, x):
        x = self.pool1(self.conv1(x))
        x = self.pool2(self.conv2_2(self.conv2_1(x)))
```

```
        x = self.pool3(self.block3_2(self.block3_1(x)))
        x = self.block4_3(self.block4_2(self.block4_1(x)))
        x = self.pool4(self.block4_5(self.block4_4(x)))
        x = self.pool5(self.block5_2(self.block5_1(x)))
        x = fluid.layers.reshape(x, [x.shape[0], -1])
        x = self.fc(x)
        return x

with fluid.dygraph.guard():
    model = GoogLeNet()

train(model)
```

通过运行结果可以发现，使用 GoogLeNet 在眼疾筛查数据集 iChallenge-PM 上，*Loss* 能有效下降，经过 5 个 epoch 的训练，在验证集上的准确率可以达到 95%左右。

3.3.6 ResNet

ResNet 是 2015 年 ImageNet 比赛的冠军，将识别错误率降低到了 3.6%，这个结果甚至超出了正常人眼识别的精度。

通过前面几个经典模型学习，我们可以发现随着深度学习的不断发展，模型的层数越来越多，网络结构也越来越复杂。那么是否加深网络结构，就一定会得到更好的效果呢？从理论上来说，假设新增加的层都是恒等映射，只要原有的层学出跟原模型一样的参数，那么深模型结构就能达到原模型结构的效果。换句话说，原模型的解只是新模型的解的子空间，在新模型解的空间里应该能找到比原模型解对应的子空间更好的结果。但是实践表明，增加网络的层数之后，训练误差往往不降反升。

Kaiming He 等人提出了残差网络 ResNet 来解决上述问题，其基本思想如图 3.27 所示。

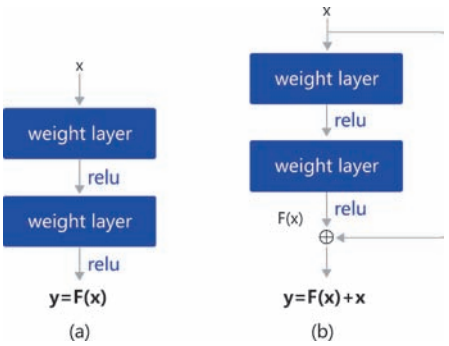


图 3.27 残差块设计思想

(1) 图 3.22(a)：表示增加网络的时候，将 x 映射成 $y=F(x)$ 输出。

(2) 图 3.22(b)：对图 3.27(a)作了改进，输出 $y=F(x)+x$。这时不是直接学习输出特征 y 的表示，而是学习 $y-x$。

- 如果想学习出原模型的表示，只需将 $F(x)$ 的参数全部设置为 0，则 $y=x$ 是恒等映射。
- $F(x)=y-x$ 也称作残差项，如果 $x\to y$ 的映射接近恒等映射，图 3.27(b)中通过学习残差项也比图 3.27(a)学习完整映射形式更加容易。

图 3.27(b)的结构是残差网络的基础，这种结构也叫作残差块(residual block)。输入 x 通过跨层连接，能更快地向前传播数据，或者向后传播梯度。残差块的具体设计方案如图 3.28 所示，这种设计方案也称作瓶颈结构(BottleNeck)。

如图 3.29 所示，表示出了 ResNet-50 的结构，一共包含 49 层卷积和 1 层全连接，所以被称为 ResNet-50。

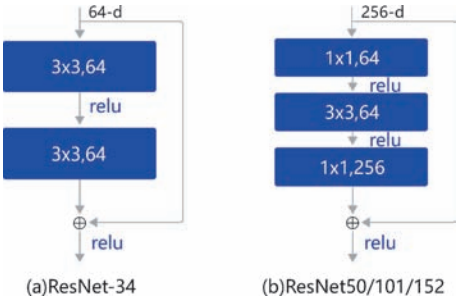


图 3.28 残差块结构示意图

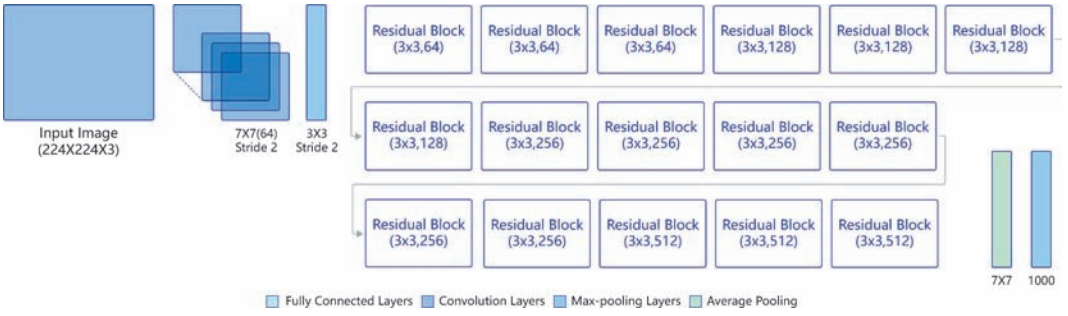


图 3.29 ResNet-50 模型网络结构示意图

ResNet-50 的具体实现如下代码所示。

1. 定义卷积批归一化

```
import numpy as np
import paddle
import paddle.fluid as fluid
from paddle.fluid.layer_helper import LayerHelper
from paddle.fluid.dygraph.nn import Conv2D, Pool2D, BatchNorm, Linear
from paddle.fluid.dygraph.base import to_variable

# ResNet 中使用了 BatchNorm 层,在卷积层的后面加上 BatchNorm 以提升数值稳定性
class ConvBNLayer(fluid.dygraph.Layer):
    def __init__(self,
                 num_channels,
                 num_filters,
                 filter_size,
                 stride = 1,
                 groups = 1,
                 act = None):
        """

        num_channels, 卷积层的输入通道数
        num_filters, 卷积层的输出通道数
        stride, 卷积层的步幅
        groups, 分组卷积的组数,默认 groups = 1 不使用分组卷积
        act, 激活函数类型,默认 act = None 不使用激活函数
        """
        super(ConvBNLayer, self).__init__()
```

```
        # 创建卷积层
        self._conv = Conv2D(
            num_channels=num_channels,
            num_filters=num_filters,
            filter_size=filter_size,
            stride=stride,
            padding=(filter_size - 1) // 2,
            groups=groups,
            act=None,
            bias_attr=False)

        # 创建 BatchNorm 层
        self._batch_norm = BatchNorm(num_filters, act=act)

    def forward(self, inputs):
        y = self._conv(inputs)
        y = self._batch_norm(y)
        return y
```

2. 定义残差块

```
# 每个残差块会对输入图片做三次卷积,然后跟输入图片进行短接
# 如果残差块中第三次卷积输出特征图的形状与输入不一致,则对输入图片做 1×1 卷积,将其输出形状调整成一致
class BottleneckBlock(fluid.dygraph.Layer):
    def __init__(self,
                 num_channels,
                 num_filters,
                 stride,
                 shortcut=True):
        super(BottleneckBlock, self).__init__()
        # 创建第一个卷积层 1×1
        self.conv0 = ConvBNLayer(
            num_channels=num_channels,
            num_filters=num_filters,
            filter_size=1,
            act='relu')
        # 创建第二个卷积层 3×3
        self.conv1 = ConvBNLayer(
            num_channels=num_filters,
            num_filters=num_filters,
            filter_size=3,
            stride=stride,
            act='relu')
        # 创建第三个卷积 1×1,但输出通道数乘以 4
        self.conv2 = ConvBNLayer(
            num_channels=num_filters,
            num_filters=num_filters * 4,
            filter_size=1,
            act=None)

        # 如果 conv2 的输出跟此残差块的输入数据形状一致,则 shortcut=True
```

```
        # 否则 shortcut = False,添加 1 个 1×1 的卷积作用在输入数据上,使其形状变成跟
conv2 一致
        if not shortcut:
            self.short = ConvBNLayer(
                num_channels = num_channels,
                num_filters = num_filters * 4,
                filter_size = 1,
                stride = stride)

        self.shortcut = shortcut

        self._num_channels_out = num_filters * 4

    def forward(self, inputs):
        y = self.conv0(inputs)
        conv1 = self.conv1(y)
        conv2 = self.conv2(conv1)

        # 如果 shortcut = True,直接将 inputs 跟 conv2 的输出相加
        # 否则需要对 inputs 进行一次卷积,将形状调整成跟 conv2 输出一致
        if self.shortcut:
            short = inputs
        else:
            short = self.short(inputs)

        y = fluid.layers.elementwise_add(x = short, y = conv2)
        layer_helper = LayerHelper(self.full_name(), act = 'relu')
        return layer_helper.append_activation(y)
```

3. 定义 ResNet 网络

```
class ResNet(fluid.dygraph.Layer):
    def __init__(self, layers = 50, class_dim = 1):
        """

        layers, 网络层数,可以是 50, 101 或者 152
        class_dim,分类标签的类别数
        """
        super(ResNet, self).__init__()
        self.layers = layers
        supported_layers = [50, 101, 152]
        assert layers in supported_layers, \
            "supported layers are {} but input layer is {}".format(supported_layers, layers)

        if layers == 50:
            #ResNet50 包含多个模块,其中第 2 到第 5 个模块分别包含 3、4、6、3 个残差块
            depth = [3, 4, 6, 3]
        elif layers == 101:
            #ResNet101 包含多个模块,其中第 2 到第 5 个模块分别包含 3、4、23、3 个残差块
            depth = [3, 4, 23, 3]
        elif layers == 152:
            #ResNet152 包含多个模块,其中第 2 到第 5 个模块分别包含 3、8、36、3 个残差块
```

```
            depth = [3, 8, 36, 3]

        # 残差块中使用到的卷积的输出通道数
        num_filters = [64, 128, 256, 512]

        # ResNet 的第一个模块,包含 1 个 7×7 卷积,后面跟着 1 个最大池化层
        self.conv = ConvBNLayer(
            num_channels = 3,
            num_filters = 64,
            filter_size = 7,
            stride = 2,
            act = 'relu')
        self.pool2d_max = Pool2D(
            pool_size = 3,
            pool_stride = 2,
            pool_padding = 1,
            pool_type = 'max')

        # ResNet 的第二到第五个模块 c2、c3、c4、c5
        self.bottleneck_block_list = []
        num_channels = 64
        for block in range(len(depth)):
            shortcut = False
            for i in range(depth[block]):
                bottleneck_block = self.add_sublayer(
                    'bb_%d_%d' % (block, i),
                    BottleneckBlock(
                        num_channels = num_channels,
                        num_filters = num_filters[block],
                        stride = 2 if i == 0 and block != 0 else 1, # c3、c4、c5 将会在第一个
残差块使用 stride = 2; 其余所有残差块 stride = 1
                        shortcut = shortcut))
                num_channels = bottleneck_block._num_channels_out
                self.bottleneck_block_list.append(bottleneck_block)
                shortcut = True

        # 在 c5 的输出特征图上使用全局池化
        self.pool2d_avg = Pool2D(pool_size = 7, pool_type = 'avg', global_pooling = True)

        # stdv 用来作为全连接层随机初始化参数的方差
        import math
        stdv = 1.0 / math.sqrt(2048 * 1.0)

        # 创建全连接层,输出大小为类别数目
        self.out = Linear(input_dim = 2048, output_dim = class_dim,
                          param_attr = fluid.param_attr.ParamAttr(
                              initializer = fluid.initializer.Uniform(-stdv, stdv)))

    def forward(self, inputs):
        y = self.conv(inputs)
        y = self.pool2d_max(y)
        for bottleneck_block in self.bottleneck_block_list:
            y = bottleneck_block(y)
```

```
        y = self.pool2d_avg(y)
        y = fluid.layers.reshape(y, [y.shape[0], -1])
        y = self.out(y)
        return y
```

```
with fluid.dygraph.guard():
    model = ResNet()

train(model)
```

通过运行结果可以发现，使用 ResNet 在眼疾筛查数据集 iChallenge-PM 上，*Loss* 能有效下降，经过 5 个 Epoch 的训练，在验证集上的准确率可以达到 95%左右。

3.3.7 小结

在这一节里，给读者介绍了几种经典的图像分类模型，分别是 LeNet、AlexNet、VGG、GoogLeNet 和 ResNet，并将它们应用到眼疾筛查数据集上。除了 LeNet 不适合大尺寸的图像分类问题之外，其他几个模型在此数据集上损失函数都能显著下降，在验证集上的预测精度在 90%左右。如果读者有兴趣的话，可以进一步调整学习率和训练轮数等超参数，观察是否能够得到更高的精度。

3.3.8 作业

如果将 LeNet 中间层的激活函数 Sigmoid 换成 ReLU，在眼底筛查数据集上将会得到什么样的结果？Loss 是否能收敛，ReLU 和 Sigmoid 之间的区别是引起结果不同的原因吗？

作业提交方式

请读者扫描图书封底的二维码，在 AI Studio“零基础实践深度学习”课程中的“作业”节点下提交相关作业。