第 8 章 卷积神经网络

前面几章介绍的都是传统的前馈神经网络结构，也称为全连接层神经网络。传统神经网络在许多应用中都有着不错的性能表现；但是在某些领域中，其性能受限，表现并不完美。本章将介绍一种全新的神经网络结构——卷积神经网络。

卷积神经网络是一类包含卷积计算且具有深度结构的神经网络，是深度学习的代表算法之一。卷积神经网络具有比传统前馈神经网络更强大的学习能力，尤其在计算机视觉、图像处理领域。本章主要介绍卷积神经网络的基本知识和原理，并且详细地讲解如何搭建一个卷积神经网络，以及如何将其应用于实际问题中。

8.1 为什么选择卷积神经网络

在机器视觉、图像识别领域，传统神经网络结构存在以下两个缺点。

第一，输入层维度过大。例如一张 $64\times64\times3$ 的三通道图片，神经网络输入层的维度多达 12 288。如果图片尺寸较大，是一张 $1000\times1000\times3$ 的三通道图片，神经网络输入层的维度将达到三百万，使神经网络权重参数 W 的数量过于庞大。这样会造成两个后果：一是神经网络结构复杂，样本训练集不够，容易出现过拟合；二是训练神经网络所需的内存和计算量都十分庞大，给训练模型带来较大的困难。

第二，传统的前馈神经网络不符合图像特征提取的机制。传统前馈神经网络的做法是将二维或者三维（包含 RGB 三通道）图片拉伸成一维特征，作为神经网络的输入层。这种操作实际上是将图片的各个像素点独立开来，忽略了各个像素点之间的区域性联系。而图片是以区域特

征为基础的，例如图片的这块区域组成了眼睛，那块区域组成了鼻子。人脸的视觉机制也是从边缘、轮廓、局部特征等方面来捕获图片信息的。传统前馈神经网络结构并不具备这样的功能，因此在性能上不会表现得特别突出。

鉴于传统前馈神经网络的这两个缺点，一种新的神经网络结构——卷积神经网络应运而生。接下来，我们就开始详细介绍卷积神经网络。

8.2　卷积神经网络的基本结构

传统前馈神经网络的输入层、隐藏层、输出层都是由一维神经元构成，而卷积神经网络的结构有很大的不同。首先，它的输入是二维矩阵（灰度图片）或者三维矩阵（彩色图片）；其次，整个卷积神经网络由卷积层、池化层、全连接层等组成，如图 8-1 所示。

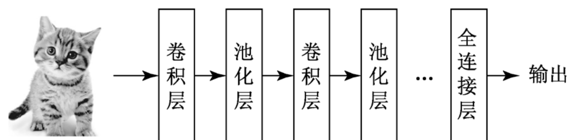


图 8-1　卷积神经网络的基本结构

由图 8-1 可以看出，卷积神经网络的输入是一张图片，包括二维矩阵（灰度图片）或者三维矩阵（彩色图片），由像素值构成。卷积神经网络最基本的结构有三种，分别是卷积层、池化层、全连接层。一般池化层在卷积层之后，成对存在，最后一般是全连接层。最后输出的是卷积神经网络模型的预测结果，相当于传统前馈神经网络的输出层，可以实现单神经元二分类，也可以实现多神经元多分类。

值得注意的是，卷积层、池化层、全连接层出现的次数不唯一，根据设计的具体卷积神经网络的结构来确定。

8.3　卷积层

卷积层（convolutional layer），顾名思义，它实现的是对图片的卷积操作。首先，我们先来了解一下什么是卷积。

8.3.1 卷积

图片的卷积运算非常简单，与我们在数字信号处理中定义的卷积运算稍有不同，更加简化了。卷积运算的具体操作步骤为：已知原图像和模板图像，首先，将模板图像在原图像中移动；然后，每到一个位置，将原图像与模板图像定义域相交的元素进行乘积且求和，得出新的图像点；最后，遍历原图像所有像素点，得到卷积后的图像，整个卷积运算完成。这里的模板图像又称卷积核。

下面，我们用一个简单的例子来说明。假设原图像是一个 3×3 的二维矩阵，卷积核是一个 2×2 的二维矩阵，如图 8-2 所示。

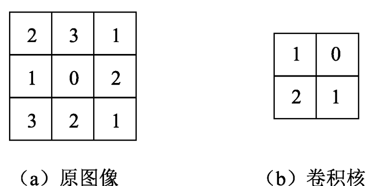


图 8-2 原图像和卷积核

对于图 8-2 中的原图像和卷积核，我们来详细展示其完整的卷积运算过程，如图 8-3 所示。

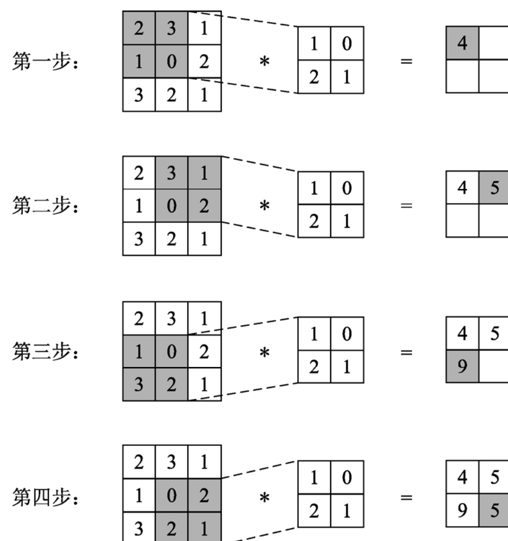
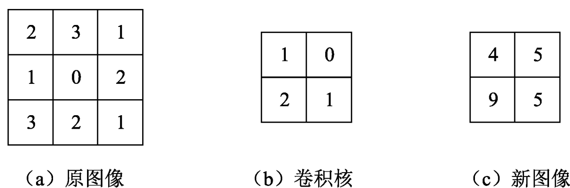


图 8-3 卷积运算过程

图 8-3 清晰地展示了一个完整的卷积运算过程。第一步，卷积核移动至原图像的左上角灰色区域，两者定义域相交的元素进行乘积且求和，得到新的图像点，数值为 4；第二步，卷积核移

动至原图像的右上角灰色区域，两者定义域相交的元素进行乘积且求和，得到新的图像点，数值为 5；第三步，卷积核移动至原图像的左下角灰色区域，两者定义域相交的元素进行乘积且求和，得到新的图像点，数值为 9；第四步，卷积核移动至原图像的右下角灰色区域，两者定义域相交的元素进行乘积且求和，得到新的图像点，数值为 5。至此，卷积核已经遍历完原图像的所有位置，最终得到的卷积后的新图像是一个 2×2 的二维矩阵，如图 8-4 所示。



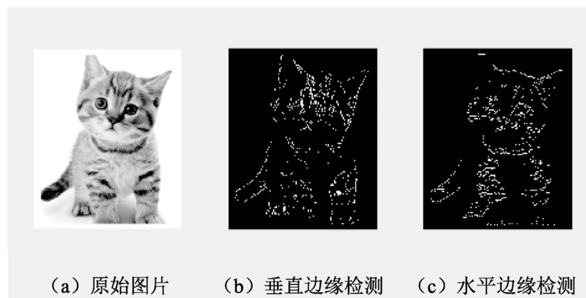
（a）原图像　（b）卷积核　（c）新图像

图 8-4　卷积运算示意

8.3.2　边缘检测

我们刚刚介绍了图片的卷积运算，那么为什么需要进行卷积运算呢？也就是说，图片经过卷积运算之后会有什么效果呢？本小节将以边缘检测为例，来说明图片卷积运算的作用。

边缘检测（edge detection）是图像处理中最常用的算法之一，其目的是检测图片中包含的边缘信息，例如水平边缘、垂直边缘等轮廓信息，如图 8-5 所示。



（a）原始图片　（b）垂直边缘检测　（c）水平边缘检测

图 8-5　边缘检测

图 8-5 展示了垂直边缘检测（vertical edge detection）和水平边缘检测（horizontal edge detection）的处理效果。可以看出，垂直边缘检测会将原始图片中的垂直线条、边缘检测出来，而水平边缘检测会将原始图片中的水平线条、边缘检测出来。

其实边缘检测算法的原理非常简单，只需将图片与相应的边缘检测算子进行卷积操作即可。也就是说，卷积运算实现的是对原始图片进行特征提取。

举例来说，垂直边缘检测和水平边缘检测的滤波器算子如图 8-6 所示，也就是 8.1 小节介绍的卷积核。

（a）垂直滤波器算子

1	0	−1
1	0	−1
1	0	−1

（b）水平滤波器算子

1	1	1
0	0	0
−1	−1	−1

图 8-6　滤波器算子

然后，就可以将原图与相应的滤波器算子进行卷积运算，以水平边缘检测为例，如图 8-7 所示。

（a）原图

10	10	10	10	10	10
10	10	10	10	10	10
10	10	10	10	10	10
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

（b）滤波器算子

1	1	1
0	0	0
−1	−1	−1

（c）水平边缘检测结果

0	0	0	0
30	30	30	30
30	30	30	30
0	0	0	0

图 8-7　水平边缘检测

图 8-7 展示了水平边缘检测的卷积运算过程。可以看到，卷积后的图片像素矩阵确实在原图水平边缘的位置呈现非零值，表示原图片的水平边缘被检测出来了。

这里我们讨论的是简单的水平滤波器算子，除此之外，还有更复杂的、检测各个方向的滤波器算子，能够帮助我们检测原图各种边缘信息。至此，我们已经介绍了图片卷积运算的意义，它可以帮助我们检测出原始图片的各种边缘特征，而这恰恰是卷积神经网络的设计原理之一。

因此，对于卷积层来说，如果我们想检测图片的各种边缘特征，而不仅限于垂直边缘和水平边缘，那么卷积核的具体数值一般需要通过模型训练得到，这与我们前面介绍的传统前馈神经网络训练参数 W 和 b 意义相同。卷积神经网络的主要目的就是求出这些卷积核。确定了卷积核之后，卷积层也就实现了对图片边缘特征的检测。

8.3.3　填充

介绍卷积运算的时候，我们发现一个问题，经过卷积运算之后的矩阵，维度较原图片矩阵减小了，且每次卷积运算都会出现这样的情况。原因很简单，因为卷积核需要在原始图片限定

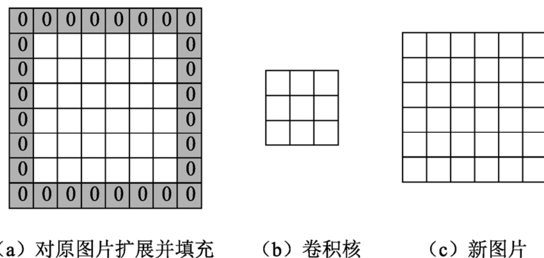
区域内滑动，就造成了有效运算范围减小。

一般来说，如果原始图片的尺寸为$n \times n$，卷积核的尺寸为$f \times f$，则卷积后图片尺寸为：

$$(n-f+1)\times(n-f+1) \tag{8-1}$$

卷积运算造成图片尺寸减小，可能会造成原始图片的边缘信息对输出的贡献少，使输出图片丢失一些边缘信息。

为了解决图片缩小的问题，可以使用填充的方法，即对原始图片的尺寸进行扩展，扩展区域补零，用 p 来表示每个方向扩展的宽度。填充的目的是先让原始图片尺寸增加，然后经过卷积运算后，新的图片尺寸维持原始图片大小，其效果如图 8-8 所示。



（a）对原图片扩展并填充　（b）卷积核　（c）新图片

图 8-8　图片填充

经过填充之后，原始图片尺寸扩展为$(n+2p)\times(n+2p)$，卷积核尺寸为$f \times f$，则卷积运算后的图片尺寸为$(n+2p-f+1)\times(n+2p-f+1)$。若要保证卷积运算前后图片尺寸不变，则$p$应满足：

$$p=\frac{f-1}{2} \tag{8-2}$$

很简单，只要令$n+2p-f+1=n$，就能得到式（8-2）中p的表达式了。

另外，由式（8-2）可以得到，如果f是奇数，能找到整数p，使填充之后卷积运算得到的图片尺寸与原图一样；如果f不是奇数，则不能找到整数p，使填充之后卷积运算得到的图片尺寸与原图一样。因此，卷积层中的卷积核尺寸宽度f一般都是奇数。

8.3.4　步幅

步幅表示卷积核在原图片中每次移动的步长。之前我们默认设置步幅为 1，当然也可以设置成其他值。若步幅为 2，则表示卷积核每次移动步长为 2。

图 8-9 是一个步幅为 2 的卷积运算例子。

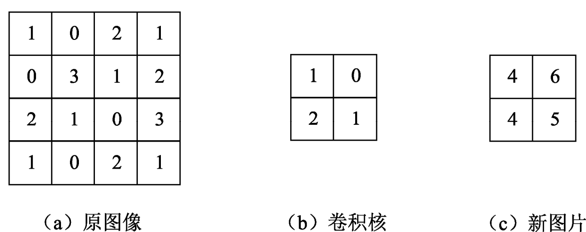


图 8-9　步幅为 2 的卷积运算

我们用 s 表示步幅大小，用 p 表示填充大小，如果原始图片尺寸为$n \times n$，卷积核尺寸为$f \times f$，则卷积运算后的图片尺寸为：

$$(\frac{n+2p-f}{s}+1)\times(\frac{n+2p-f}{s}+1) \tag{8-3}$$

对于图 8-9 中的例子，将 $n=4$、$p=0$、$s=2$、$f=2$ 代入式（8-3）中，就可得到卷积运算后的图片尺寸大小为 2，与实际结果完全吻合。

值得注意的是，如果出现式（8-3）中$(\frac{n+2p-f}{s}+1)$不是整数的情况，则一般的做法是对其进行向下取整。

8.3.5　卷积神经网络卷积

卷积神经网络的输入一般是图片，我们以三通道图片为例，来了解一下卷积神经网络卷积层的结构。

前面介绍的都是二维平面卷积，针对的是单通道图片。如果是三通道的图片，即三维卷积是如何运算的呢？

其实，三维卷积运算的基本原理与二维卷积相同，只是增加了一个维度。三维卷积运算需要使用 3 个卷积核，分别与对应通道的图片进行卷积运算，如图 8-10 所示。

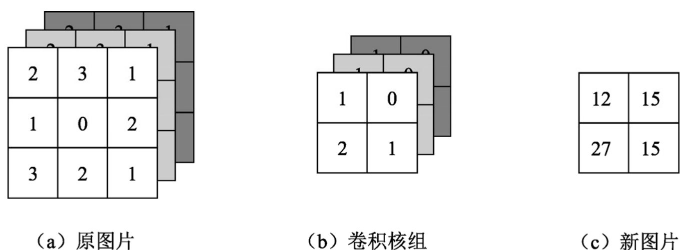


图 8-10　三维卷积运算

图 8-10 中，原图片是由 RGB 三通道组成，分别对应红、绿、蓝三种颜色，维度为$3\times3\times3$。

卷积核组的尺寸大小为 $f=2$，共包含 3 个卷积核，分别对应原图片中的三通道，维度为 $2\times2\times3$。卷积运算时，原图片各个通道分别与其对应的卷积核进行卷积运算，然后将各个通道的结果累加起来作为相应位置的输出。图 8-10 中，$n=3$，$f=2$，$p=0$，$s=1$。根据式（8-3），卷积运算后的图片尺寸为 $\frac{n+2p-f}{s}+1=2$，即新图像的维度为 $2\times2\times1$。

图 8-10 展示的卷积核只有一组，在实际的卷积神经网络卷积层中，通常会包含多组卷积核，每组卷积核分别与原图片进行卷积运算，最后将各组得到的新图片组合起来，效果如图 8-11 所示。

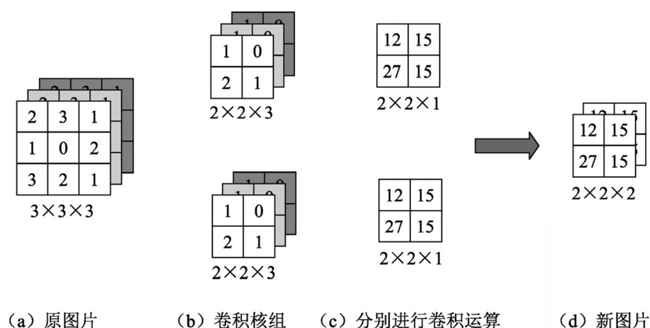


图 8-11　多组卷积核三维卷积运算

图 8-11 与图 8-10 唯一的差别就是包含了两组卷积核，其目的是进行多次卷积运算，实现更多边缘检测。例如，第一个卷积核组实现垂直边缘检测，第二个卷积核组实现水平边缘检测。这样，不同滤波器组卷积就得到不同的输出，最后输出的维度由卷积核组的个数决定。

若输入图片的维度为 $n\times n\times n_c$，卷积核的维度为 $f\times f\times n_c\times n_c^{'}$，卷积后的图片维度为 $(n-f+1)\times(n-f+1)\times n_c^{'}$。其中，$n_c$ 为图片通道数目，$n_c^{'}$ 为卷积核组个数。这里为了计算简便，默认 $p=0$、$s=1$。

接下来，我们需要在卷积层中加入非线性激活层，即在卷积运算之后再引入偏置参数 b，并使用激活函数对输出进行非线性运算，如图 8-12 所示。

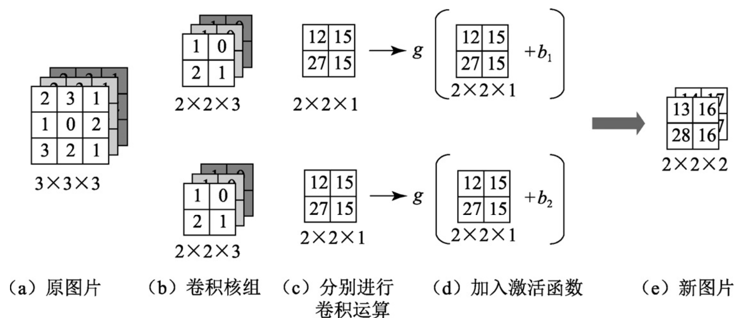


图 8-12　加入激活函数的卷积运算

如图 8-12 所示，在卷积运算之后，对结果加上偏移常数 b，再进行激活函数的非线性运算，用 $g()$ 来表示，最后得到卷积层的输出。

其实，此过程与传统的前馈神经网络单层结构非常类似，传统前馈神经网络单层递归计算公式如下：

$$\begin{cases} Z^{[l]} = \boldsymbol{W}^{[l]} A^{[l-1]} + b^{[l]} \\ A^{[l]} = g(Z^{[l]}) \end{cases} \tag{8-4}$$

在卷积层运算中，卷积核组就相当于式（8-4）中的 $\boldsymbol{W}^{[l]}$，卷积层的线性运算就相当于 $W^{[l]}$ 与 $A^{[l-1]}$ 的乘积再加上偏置参数 $b^{[l]}$，得到 $Z^{[l]}$。然后，经过激活函数 $g()$ 的作用，实现非线性运算，得到最终的输出 $A^{[l]}$。

我们来计算图 8-12 中所有参数的数量：每个卷积核组有 $2\times2\times3=12$ 个参数，还有 1 个偏移参数 b，则每个卷积核组有 $12+1=13$ 个参数，两个卷积核组共包含 $13\times2=26$ 个参数。我们发现，选定卷积核组后，总的参数数量与输入图片尺寸大小无关，所以就避免了由于图片尺寸过大造成参数过多的情况发生。这样大大简化了模型的复杂度，提高了模型的运算速度和性能。这也正是卷积神经网络的优点之一。

下面，我们总结卷积层的标记符号。

输入维度：$n_h^{[l-1]}\times n_w^{[l-1]}\times n_c^{[l-1]}$。

每个卷积核组的维度：$f^{[l]}\times f^{[l]}\times n_c^{[l-1]}$。

权重维度：$f^{[l]}\times f^{[l]}\times n_c^{[l-1]}\times n_c^{[l]}$。

偏置维度：$1\times1\times1\times n_c^{[l]}$。

输出维度：$n_h^{[l]}\times n_w^{[l]}\times n_c^{[l]}$。

其中，l 为当前网络层数；n 的下标 h、w、c 分别表示图片的高度、宽度、通道；$n_c^{[l-1]}$ 表示上一层卷积核组的个数；$n_c^{[l]}$ 表示该层卷积核组的个数。

卷积运算后的结果 $n_h^{[l]}$ 和 $n_w^{[l]}$ 的计算公式如下：

$$\begin{cases} n_h^{[l]} = \dfrac{n_h^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \\ n_w^{[l]} = \dfrac{n_w^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \end{cases} \tag{8-5}$$

一般来说，大部分情况下图片的高度和宽度都是相等的，即 $n_h^{[l-1]} = n_w^{[l-1]}$，且 $n_h^{[l]} = n_w^{[l]}$。

以上是单个样本图片卷积层的标记符号，如果共有 m 个样本，则一般将 m 增加到第 0 维，此时卷积层的标记符号如下。

输入维度：$m\times n_h^{[l-1]}\times n_w^{[l-1]}\times n_c^{[l-1]}$。

每个卷积核组的维度：$f^{[l]}\times f^{[l]}\times n_c^{[l-1]}$。

权重维度：$f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]}$。

偏置维度：$1 \times 1 \times 1 \times n_c^{[l]}$。

输出维度：$m \times n_h^{[l]} \times n_w^{[l]} \times n_c^{[l]}$。

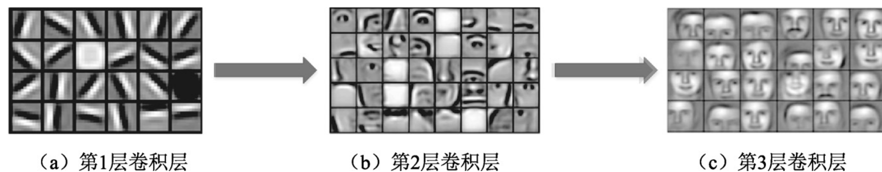
可以看到，权重维度和偏置维度没有维度 m，这与传统的前馈神经网络是一样的。

8.3.6　卷积层的作用

我们刚才介绍了卷积的运算过程，使用卷积实现边缘检测。那么，卷积层在卷积神经网络中到底实现什么功能呢？

实际上，卷积神经网络中通常由浅到深包含多个卷积层，最浅层的卷积层倾向于学习原图片中的点、颜色等基础特征，深层的卷积层开始学习线段、边缘等特征。层数越深，卷积层学习到的特征就越具体、越抽象。

图 8-13 展示了一个很好的人脸识别的例子。可以看出，第 1 层卷积层提取的是人脸的边缘、线条、轮廓等浅层特征；第 2 层卷积层提取的是人脸的一些器官，如眼镜、鼻子、耳朵等；第 3 层卷积层提取的是人脸的整体面部轮廓，这时候整个人脸就看得比较清楚了。



（a）第1层卷积层　（b）第2层卷积层　（c）第3层卷积层

图 8-13　各卷积层提取的特征

总而言之，卷积神经网络不同的卷积层会根据网络深浅提取不同层次的特征。这正是使用多层卷积层的原因，卷积神经网络的强大之处也在于此。

8.4　池化层

池化层（pooling layers）在卷积神经网络中用于减小尺寸，提高运算速度，也能减小噪声的影响，让各特征更具有健壮性。卷积神经网络中，池化层一般出现在卷积层的后面。

池化层比卷积层简单得多，没有卷积运算。最常用的池化层运算是最大池化。最大池化的做法就是选择一个类似于卷积核的滤波器算子，在上一层输出矩阵上滑动；然后，每到一个位置，计算两者定义域相交元素的最大值，作为新的图像点；最后，遍历原图片所有像素点，得

到最大池化之后的图片。整个过程只有比较大小，没有其他数学运算。

最大池化的示意图如图 8-14 所示。

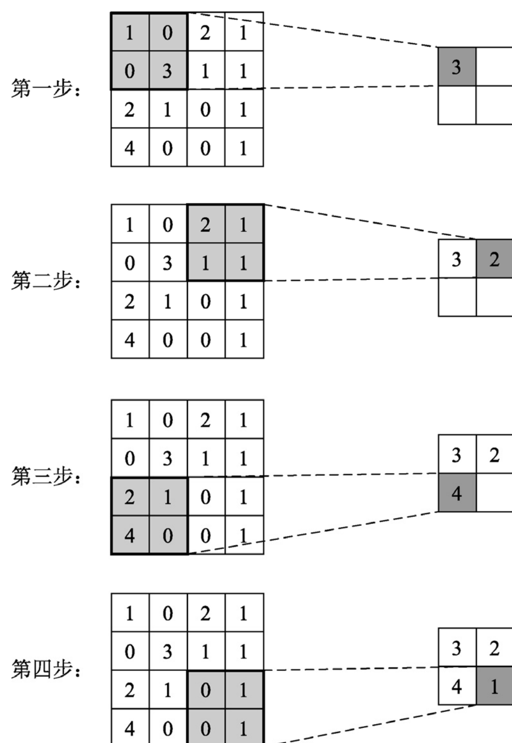


图 8-14　池化层最大池化

通过图 8-14，可以很清晰地看到最大池化的过程，只在每次滑动的区域内寻找最大值。上面显示的是单通道情况，如果是多通道，原理一样，只需要分别对各个通道进行最大池化就可以了。

最大池化有两个参数需要注意：一个是滤波器算子（滑动窗）的尺寸 f，另外一个是滤波器算子每次移动的步幅 s。例如在图 8-14 中，$f = 2$，$s = 2$。注意，填充参数 p 很少在池化层中使用。

最大池化的优点是只保留区域内的最大值（特征），忽略其他值，减小噪声的影响，提高模型的健壮性，而且计算量很小。

除了最大池化之外，池化层的另一种做法是平均池化。顾名思义，平均池化就是在滤波器算子滑动区域计算平均值。

平均池化的示意图如图 8-15 所示。

通过图 8-15，可以很清晰地看到平均池化的过程，只在每次滑动的区域内计算平均值。上

面显示的是单通道情况，如果是多通道，原理一样，只需要分别对各个通道进行平均池化就可以了。

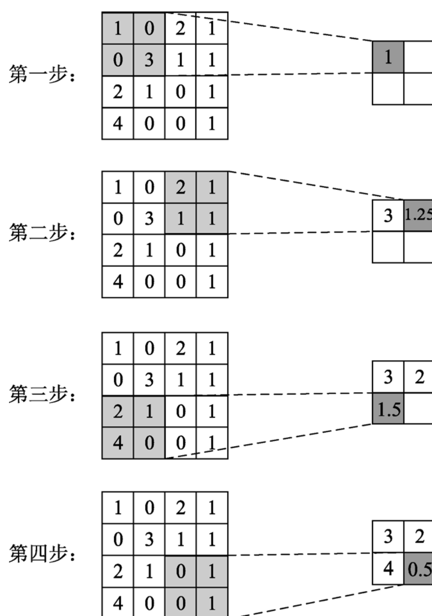


图 8-15　池化层平均池化

图 8-15 中，滤波器算子（滑动窗）的尺寸 $f=2$，滤波器算子每次移动的步幅 $s=2$。

平均池化的优点是顾及每一个像素，选择将所有的像素值都相加然后再平均的做法也会增强模型的抗噪声能力。

实际应用中，最大池化比平均池化更常用一些。输入通道与输出通道个数相同，因为我们对每个通道都做了池化。需要注意的是，池化层没有需要学习的参数。执行反向传播时，反向传播没有参数适用于最大池化。也就是说，池化层没有网络参数 $\boldsymbol{W}$ 和 b，也就不需要对池化层进行反向梯度的优化。

最后，给出一个三通道最大池化的例子，如图 8-16 所示。

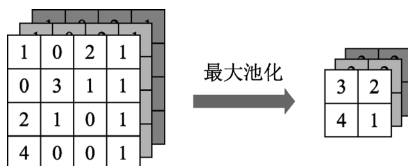


图 8-16　三通道最大池化

8.5 全连接层

卷积层和池化层都是多维矩阵，全连接层更加简单易懂，即将上一个卷积层或池化层的维度拉伸成一维向量。例如，当前卷积层的维度是 10×10×5，则拉伸为一维向量的神经元个数就是 500，后面再连接若干一维神经元层。全连接层实际上就是传统的前馈神经元网络结构，一般出现在卷积神经网络的末端，输出层之前。

全连接层的结构如图 8-17 所示。

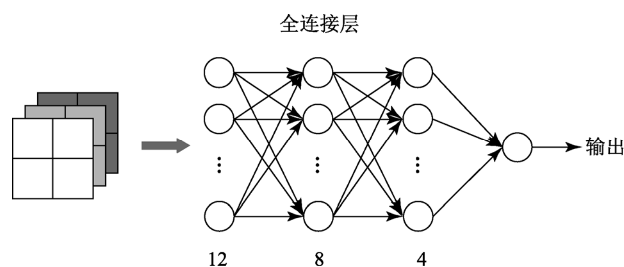


图 8-17 全连接层的结构

图 8-17 中，来自前一层的卷积层或池化层会被展开成一维向量，由 12 个神经元组成。然后是两层前馈神经网络层，各包含 8 个和 4 个神经元。接着就是整个卷积神经网络的输出层，输出层可以是单个神经元（二分类），也可以是 Softmax 层（多个神经元，多分类）。最后输出的是预测的概率值。其实，这里的全连接层与传统前馈神经网络是一样的。

值得一提的是，全连接层的层数和各层包含的神经元个数是不固定的，可以根据情况选择。

为什么卷积神经网络最后需要引入全连接层呢？卷积层提取的是局部特征，全连接层就是把之前按所有的局部特征重新通过权值矩阵组装成完整的图，因为用到了所有的局部特征，所以叫作全连接层。实际上，卷积层提取好特征，由全连接层对特征再次进行完整的学习和训练，最终实现分类的效果。

直观地说，卷积层提取的是局部视野，如果用它类比我们的眼睛的话，就是将外界信息翻译成神经信号的工具，它能将接收的输入中的各个特征提取出来；而全连接层好比是我们的大脑，它能够利用卷积层提取的特征来做出相应的决策，全连接层起到将学习到的分布式特征表示映射到样本标记空间的作用。

事实上，卷积神经网络的强大之处其实就在于其卷积层强大的特征提取能力。我们完全可以利用卷积神经网络将特征提取出来，然后使用全连接层或决策树、支持向量机等各种机器学习算法模型来进行分类。

8.6　卷积神经网络模型

介绍完卷积层、池化层、全连接层之后，我们就可以构建一个完整的卷积神经网络模型，如图 8-18 所示。

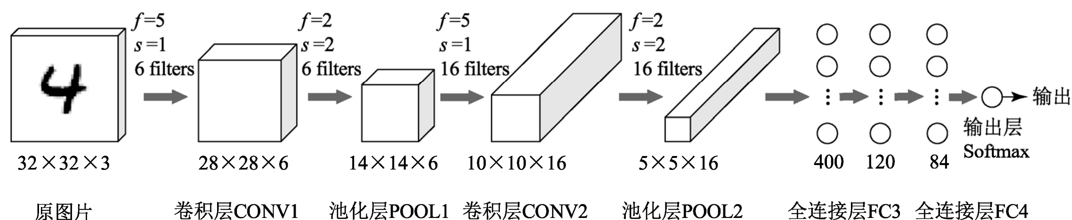


图 8-18　完整的卷积神经网络模型

图 8-18 展示的是一个完整的卷积神经网络模型。从整体上来看，模型的输入是维度为 32×32×3 的图片，输出是一个 Softmax 层，处理的是多分类问题。除了输入之外，模型的主要结构有 7 个，分别是卷积层 CONV1、池化层 POOL1、卷积层 CONV2、池化层 POOL2、全连接层 FC3、全连接层 FC4、输出层 Softmax。

在卷积层 CONV1 中，使用的卷积核尺寸 $f=5$，卷积核的个数为 6 个，步幅 $s=1$，则该层输出的尺寸为 $32-5+1=28$，维度为 $28\times28\times6$。包含的参数个数为 $(5\times5\times3+1)\times6=456$，因为每个卷积核有 $5\times5\times3=75$ 个参数，还有 1 个偏置参数，6 个卷积核。

在池化层 POOL1 中，使用的滤波器算子尺寸 $f=2$，滤波器算子个数为 6 个，步幅 $s=2$，则该层输出的尺寸为 $28\div2=14$，维度为 $14\times14\times6$。包含的参数为 0，因为池化层没有参数。

在卷积层 CONV2 中，使用的卷积核尺寸 $f=5$，卷积核的个数为 16 个，步幅 $s=1$，则该层输出的尺寸为 $14-5+1=10$，维度为 $10\times10\times16$。包含的参数个数为：$(5\times5\times6+1)\times16=2416$，因为每个卷积核有 $5\times5\times6=125$ 个参数，还有 1 个偏置参数，16 个卷积核。

在池化层 POOL2 中，使用的滤波器算子尺寸 $f=2$，滤波器算子个数为 16 个，步幅 $s=2$，则该层输出的尺寸为 $10\div2=5$，维度为 $5\times5\times16$。包含的参数为 0，因为池化层没有参数。

接下来，我们需要将上一个池化层三维矩阵拉伸成一维向量，以便后面与全连接层相连。因为 POOL2 的维度是 $5\times5\times16$，则拉伸为一维向量的神经元个数为 $5\times5\times16=400$。这一层是展开层，没有参数。

在全连接层 PC3 中，它的输入是 400 个神经元，该层包含了 120 个神经元。因此，总共包含的参数 $\boldsymbol{W}$ 和 b 的个数为 $400\times120+120=48\,120$。其中，400×120 是权重参数 $\boldsymbol{W}$ 的数量，120 是偏置参数 b 的数量。

在全连接层 PC4 中，它的输入是 120 个神经元，该层包含了 84 个神经元。因此，总共包含

的参数 $\boldsymbol{W}$ 和 b 的个数为 $120\times84+84=10164$ 。其中，120×84 是权重参数 $\boldsymbol{W}$ 的数量，84 是偏置参数 b 的数量。

在最后的输出层 Softmax，例如输出有 10 个神经元，即实现的是十分类问题。它的输入是 84 个神经元，该层包含了 10 个神经元。因此，总共包含的参数 $\boldsymbol{W}$ 和 b 的个数为 $84\times10+10=850$ 。其中，84×10 是权重参数 $\boldsymbol{W}$ 的数量，10 是偏置参数 b 的数量。

以上就是对该卷积神经网络模型所有细节的描述和参数的统计情况。

了解了卷积神经网络模型的基本结构之后，如何来训练这个模型呢？实际上，训练卷积神经网络模型就是训练优化网络结构中所有的参数值。在这点上，原理与传统的前馈神经网络相同。

卷积神经网络模型的训练过程也是基于梯度下降算法的。开始训练时，卷积神经网络中卷积层的卷积核系数包括全连接层各项参数都是随机初始化的；然后，整个网络进行前向传播，计算模型的损失函数；接着，使用梯度下降算法及各种优化算法进行反向传播，更新各个参数值。经过多次迭代训练之后，各卷积核参数和全连接层的各项参数都将取得最优值，从而使模型具有较高的准确率。

卷积神经网络模型的结构相对比较复杂，训练过程涉及较多的参数和复杂的计算，如果完全使用手动搭建的方式效率会很低。因此，在掌握卷积神经网络基本原理的前提下，建议使用各种成熟的深度学习框架来搭建卷积神经网络模型，例如 PyTorch 和 TensorFlow。8.8 节和 8.9 节将使用 PyTorch 和 TensorFlow 来搭建一个卷积神经网络模型，解决图片分类的问题。

有一个问题：与传统的前馈神经网络相比，为什么卷积神经网络在图片分类问题上的性能表现更优呢？其实，我们在 8.1 节已经有所提及。首先，卷积神经网络参数的数目要少得多。一方面特征检测器（如垂直边缘检测）对图片某块区域有用，同时也可能作用在图片其他区域；另一方面因为滤波器算子尺寸的限制，每一层的每个输出只与输入部分区域内有关。这样就使卷积核能够在图片的很多区域内发生作用，节省了不必要的参数。其次，由于卷积神经网络参数的数目较小，所需的训练样本也相对较少，在一定程度上不容易发生过拟合现象。而且，卷积神经网络比较擅长捕捉区域位置偏移，也就是说卷积神经网络进行物体检测时，受物体所处图片位置的影响较小，增加了检测的准确性和系统的健壮性。

8.7 典型的卷积神经网络模型

前几节已经介绍了卷积神经网络的基本结构，本节将主要介绍几个具有代表性的卷积神经网络模型。

8.7.1　LeNet-5

LeNet-5 模型是 Yann LeCun 于 1998 年提出来的，它是第一个成功应用于数字识别问题的卷积神经网络。在 MNIST 数据中，它的准确率达到大约 99.2%。典型的 LeNet-5 结构包含卷积层、池化层和全连接层，顺序一般是：卷积层→池化层→卷积层→池化层→全连接层→全连接层→输出层。

图 8-19 展示了 LeNet-5 的基本结构。输入是二维的灰度图像，尺寸为32×32；C1 是卷积层，尺寸为6×28×28，注意这里把通道数放在第 0 个维度；S2 是池化层，尺寸为6×14×14；C3 是卷积层，尺寸为16×10×10；S4 是池化层，尺寸为16×5×5；C5 是全连接层，包含 120 个神经元；F6 是全连接层，包含 84 个神经元；OUTPUT 是输出层，一般是 Softmax 层。

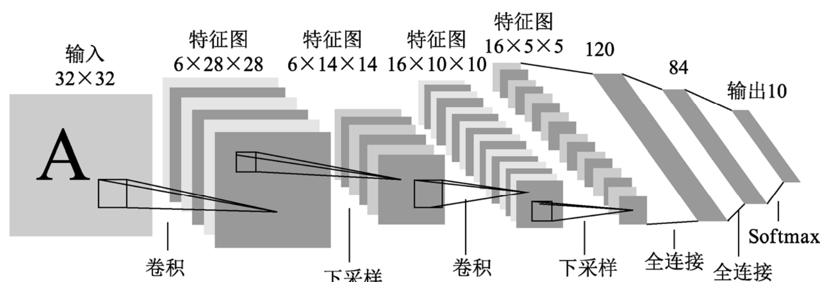


图 8-19　LeNet-5 的基本结构

其实 LeNet-5 网络与图 8-18 中的卷积神经网络模型基本是一样的，只是图 8-18 的卷积神经网络模型的输入图像是三通道的，而这里的 LeNet-5 是单通道的。可以说，LeNet-5 是简单但又功能强大的卷积神经网络模型之一。

8.7.2　AlexNet

AlexNet 是 2012 年 ImageNet 竞赛冠军获得者 Hinton 和他的学生 Alex Krizhevsky 设计的。AlexNet 可以直接对彩色的大图片进行处理，对于传统的机器学习分类算法而言，它的性能相当出色。

AlexNet 是由 5 个卷积层和 3 个全连接层组成，顺序一般是：卷积层→池化层→卷积层→池化层→卷积层→卷积层→卷积层→池化层→全连接层→全连接层→输出层。AlexNet 的基本结构如图 8-20 所示。

AlexNet 的输入是二维的彩色图片，尺寸为224×224×3。可以看到，AlexNet 比 LeNet-5 更加复杂，而且输入图片的尺寸更大，还是三通道的彩色图片。这也决定了 AlexNet 的性能更好，模型更强大。

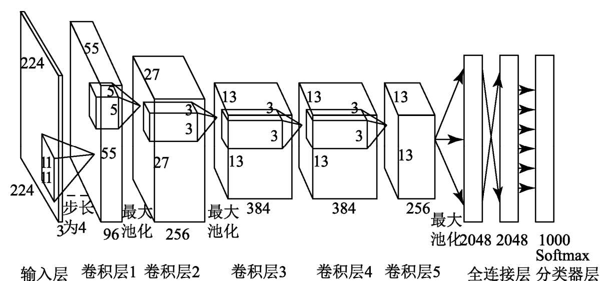


图 8-20　AlexNet 的基本结构

8.8　卷积神经网络模型的 PyTorch 实现

对卷积神经网络模型有了比较深入的了解之后，本节将使用 PyTorch 来搭建卷积神经网络模型，并进行手写数字图片的识别和分类。

8.8.1　准备数据

我们选用的数据集是非常有名的手写数据集 MNIST。MNIST 数据集来自美国国家标准与技术研究所（National Institute of Standards and Technology，NIST），包含数字 0～9 的手写版。

MNIST 数据集由 60 000 张训练图片和 10 000 万张测试图片构成，每张图片的尺寸都是 28×28×1（灰度图像，单通道）。MNIST 数据集如图 8-21 所示。

图 8-21　MNIST 数据集

MNIST 数据集一般由以下四个部分组成。

训练图片：train-images-idx3-ubyte。

训练图片标签：train-labels-idx1-ubyte。

测试图片：t10k-images-idx3-ubyte。

测试图片标签：t10k-labels-idx1-ubyte。

可以看到，MNIST 数据集采用 ubyte 格式存储，便于压缩和节省空间。

在 PyTorch 中，下载和导入 MNIST 数据集非常简单，可以使用 torchvision 库来完成。关于

torchvision，我们在 2.2 节安装 PyTorch 的时候就简单介绍过并完成了 torchvision 的安装。其实，torchvision 是一个专门进行图形处理的库，可加载比较常见的数据库，如 ImageNet、CIFAR10、MNIST 等。使用 torchvision 的好处是避免了重复编写数据集加载代码，让数据集的加载更加简单。

图片的数据转换采用 torchvision.datasets 和 torch.utils.data.DataLoader 即可完成。下载并导入 MNIST 数据集的代码如下：

```
import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import matplotlib.pyplot as plt
import numpy as np

transform = transforms.Compose(
   [transforms.ToTensor()])

# 训练集
trainset = torchvision.datasets.MNIST(
          root='./datasets/ch08/pytorch',       # 选择数据的根目录
          train=True,
          download=True,                        # 从网络上下载图片
          transform=transform)
trainloader = torch.utils.data.DataLoader(
          trainset,
          batch_size=4,
          shuffle=True,
          num_workers=2)
# 测试集
testset = torchvision.datasets.MNIST(
          root='./datasets/ch08/pytorch',       # 选择数据的根目录
          train=False,
          download=True,                        # 从网络上下载图片
          transform=transform)
testloader = torch.utils.data.DataLoader(
          testset,
          batch_size=4,
          shuffle=False,
          num_workers=2)
```

在上面的代码中，torchvision.datasets.MNIST 会自动在网络上下载 MNIST 数据集，参数 root 设置数据集在本地存放的目录，可自由选择。注意，对于训练集，参数 train 设置为 True；对于测试集，参数 train 设置为 False。关于参数 download，如果是第一次运行该代码，则将其设置为 True，表示从网络上下载 MNIST 数据集；如果已经下载了数据集，就可以将其设置为 False。参

数 transform 实现将输入图像转化为张量，并将数值归一化到[0,1]范围内。

torch.utils.data.DataLoader 实现的是对数据集的处理。参数 batch_size 表示每个小批量样本集中的样本数量。参数 shuffle 表示是否在每个 epoch 中随机打乱数据集，这样做的目的是使每个 epoch 数据集的次序都不一样，保证每个小批量样本集尽可能不一样，提高接下来的训练效果。参数 num_workers 表示使用多少个子进程来导入数据。

接下来，运行上面的代码，程序会自动下载 MNIST 数据集，并将其存放在目录“./datasets/ch08/pytorch”下。

我们可以来看一下 trainset 和 testset 的内容：

```
print(trainset)
Dataset MNIST
    Number of datapoints: 60000
    Split: train
    Root Location: ./datasets/ch08
    Transforms (if any): Compose(
                             ToTensor()
                         )
Target Transforms (if any): None


print(testset)
Dataset MNIST
    Number of datapoints: 10000
    Split: test
    Root Location: ./datasets/ch08
    Transforms (if any): Compose(
                             ToTensor()
                         )
    Target Transforms (if any): None
```

下面的代码展示了一个小批量样本集的训练集图片以及标注正确标签的过程。

```
def imshow(img):
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))

# 选择一个 batch 的图片
dataiter = iter(trainloader)
images, labels = dataiter.next()
# 显示图片
imshow(torchvision.utils.make_grid(images))
plt.show()
# 打印 标签
print(' '.join('%11s' % labels[j].numpy() for j in range(4)))
```

运行上面的代码，得到一个小批量样本集的图片和对应的标签，如图 8-22 所示。

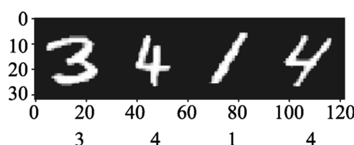


图 8-22　一个小批量样本集的图片和对应的标签

8.8.2　定义卷积神经网络模型

此处，我们使用之前介绍的 LeNet-5 网络，其结构如图 8-19 所示。

在 PyTorch 中构建卷积神经网络模型非常简单，代码如下：

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # 1 个输入图片通道，6 个输出通道，5×5 卷积核
        self.conv1 = nn.Conv2d(1, 6, 5)
        # max pooling，2×2
        self.pool1 = nn.MaxPool2d(2, 2)
        # 6 个输入图片通道，16 个输出通道，5×5 卷积核
        self.conv2 = nn.Conv2d(6, 16, 5)
        # max pooling，2×2
        self.pool2 = nn.MaxPool2d(2,2)
        # 拉伸成一维向量，全连接层
        self.fc1 = nn.Linear(16 * 4 * 4, 120)
        # 全连接层
        self.fc2 = nn.Linear(120, 84)
        # 全连接层，输出层 Softmax，10 个数字
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool1(x)
        x = F.relu(self.conv2(x))
        x = self.pool2(x)
        # 拉伸成一维向量
        x = x.view(-1, 16 * 4 * 4)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

上面的代码是构建卷积神经网络模型的核心部分。我们发现 PyTroch 构建卷积神经网络模型的过程非常简单，只需要简单的几行语句。在类 Net 的初始化函数中，直接搭建卷积层、池化层和全连接层。其中，nn.Conv2d(1, 6, 5)里的“1”代表输入图片的通道数，因为是灰度图片，

所以通道为 1；“6”表示第一层卷积核的组数；“5”表示卷积核的尺寸大小。nn.Conv2d(6, 16, 5) 的含义以此类推。nn.MaxPool2d(2, 2) 表示池化层采用最大池化，2 表示尺寸大小。nn.Linear(16 * 4 * 4, 120) 表示第一个全连接层。16*4*4 表示前一池化层展开为一维矩阵的长度。下面解释 16*4*4 是怎样得来的。

MNIST 图片的尺寸为28×28×1，经过第一层卷积层和池化层后，尺寸为：

$$\frac{28-5}{1}+1=24$$

$$\frac{24}{2}=12$$

经过第二层卷积层和池化层后，尺寸为：

$$\frac{12-5}{1}+1=8$$

$$\frac{8}{2}=4$$

由于该池化层滤波器组个数为 16，则拉伸一维数组的长度就是 16*4*4。

函数 forward(self, x)定义了卷积神经网络的前向传播过程。接下来，我们可以建立一个 Net 对象，并查看它的网络结构。

```
net = Net()
print(net)
Net(
  (conv1): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
  (pool1):MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (pool2):MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=256, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

可见，整个 Net 结构非常直观，可以完整、清晰地查看我们构建的卷积神经网络模型的结构。

8.8.3 损失函数与梯度优化

该项目是一个分类问题，所以损失函数使用交叉熵，PyTorch 中用 nn.CrossEntropyLoss 表示交叉熵。如果是回归问题，损失函数一般使用均方差，PyTorch 中用 nn.MSELoss 表示均方差。

在卷积神经网络模型的反向传播中，仍然是基于梯度下降算法来优化参数的。我们在第 7 章介绍过很多梯度优化算法，如 RMSprop、Adam 等，这些梯度优化算法同样可以应用到卷积神

经网络模型中。使用方法非常简单，直接调用 PyTorch 中的 torch.optim 模块即可。例如，torch.optim.RMSprop 表示 RMSprop 优化，torch.optim.Adam 表示 Adam 优化。

定义损失函数和梯度优化的代码如下：

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(net.parameters(), lr=0.0001)
```

这里，我们使用的梯度优化算法是 Adam，学习率设置为 0.0001。

8.8.4　训练模型

定义好模型、损失函数和梯度优化算法之后，就可以训练卷积神经网络模型了。我们选择的 epoch 数目为 5，代码如下：

```
num_epoches = 5       # 设置 epoch 数目
cost = []             # 损失函数累加

for epoch in range(num_epoches):

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # 输入样本和标签
        inputs, labels = data
        # 每次训练梯度清零
        optimizer.zero_grad()

        # 正向传播、反向传播和优化过程
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # 打印训练情况
        running_loss += loss.item()
        if (i+1) % 2000 == 0:  # 每隔 2000 个小批量样本打印一次
            print('[epoch: %d, mini-batch: %5d] loss: %.3f' %
                 (epoch + 1, i + 1, running_loss / 2000))
            cost.append(running_loss / 2000)
            running_loss = 0.0
```

上述代码中需要注意的是，每次迭代训练时都要先把所有梯度清零，即执行 optimizer.zero_grad()。否则，梯度会累加，造成训练错误和失效。PyTorch 中的.backward()可自动完成所有梯度计算。

训练的过程中，每隔 2000 个小批量样本，会将损失打印出来。整个训练过程中，running_loss 的变化趋势如图 8-23 所示。

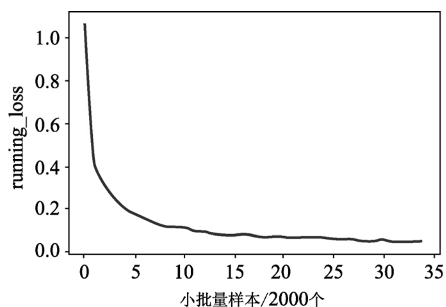


图 8-23 running_loss 的变化趋势

很明显，在训练的过程中，running_loss 是逐渐减小的，说明我们的训练是有效的。

8.8.5 测试模型

接下来就是最后一步，使用训练好的模型进行测试，验证模型的效果。

首先，验证模型在训练集上的效果，代码如下：

```
correct = 0
total = 0
with torch.no_grad():
    for data in trainloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy on the 60000 train images: %.3f %%' %
    (100 * correct / total))
```

打印的结果如下：

```
Accuracy on the 60000 train images: 98.733 %
```

然后，验证模型在测试集上的效果，代码如下：

```
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
```

```
print('Accuracy on the 10000 test images: %.3f %%' %
    (100 * correct / total))
```

打印的结果如下：

```
Accuracy on the 10000 train images: 98.600 %
```

可以看出，训练集的准确率达到 98.733%，测试集的准确率达到 98.600%。从结果来看，该卷积神经网络模型的性能是相当不错的。

8.9　卷积神经网络模型的 TensorFlow 实现

本节将使用 TensorFlow 来搭建卷积神经网络模型，并对 MNIST 手写数字图片进行识别和分类。

8.9.1　准备数据

对于 MNIST 数据集，我们在 8.8.1 节已有所介绍，这里就不赘述了。在 TensorFlow 中下载和导入 MNIST 数据集也非常简单，代码如下：

```
from tensorflow.examples.tutorials.mnist import input_data
import tensorflow as tf

mnist = input_data.read_data_sets(
      './datasets/ch08/MNIST',one_hot=True)
```

很简单，只需要一行语句 input_data.read_data_sets()就可以自动从官网上下载 MNIST 数据集，可以指定下载目录。one_hot=True 表示设置输出为独热编码。

上述代码中的 mnist 包含了训练集、验证集和测试集。我们可以使用以下语句来查看各数据集的维度。

```
print(mnist.train.images.shape)
print(mnist.train.labels.shape)
(55000, 784)
(55000, 10)


print(mnist.validation.images.shape)
print(mnist.validation.labels.shape)
(5000, 784)
(5000, 10)
```

```
print(mnist.test.images.shape)
print(mnist.test.labels.shape)
(10000, 784)
(10000, 10)
```

可见，原本 MNIST 训练集包含 60 000 个样本，现在分成训练集（55 000 个样本）和验证集（5 000 个样本）。划分验证集的好处是可以用来验证模型，但在本例中不会用到它。

8.9.2 定义卷积神经网络模型

TensorFlow 中定义卷积神经网络模型并不像 PyTorch 中那样简便，首先需要定义几个创建模型所需要的函数。

```
# input 代表输入，filter 代表卷积核
# 卷积层
def conv2d(x, filter):
    return tf.nn.conv2d(x,
                        filter,
                        strides=[1,1,1,1],
                        padding='SAME')

# 池化层
def max_pool(x):
    return tf.nn.max_pool(x,
                          ksize=[1,2,2,1],
                          strides=[1,2,2,1],
                          padding='SAME')

# 初始化卷积核或者权重数组的值
def weight_variable(shape):
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial)

# 初始化 bias 的值
def bias_variable(shape):
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)
```

上述代码中，conv2d()是定义卷积层的函数；x 表示上一层输入；filter 表示卷积核尺寸；stride 表示步幅；padding='SAME'表示进行填充，保证卷积运算前后尺寸不变。

max_pool()是定义池化层的函数，各参数与 conv2d()是一样的。

weight_variable()和 bias_variable()是参数 $\boldsymbol{W}$ 和 b 的初始化函数，其中，$\boldsymbol{W}$ 进行随机初始化，方差为 0.1，b 初始化为常量 0.1。

1. CONV1

下面，首先来定义卷积层 1（包含池化层），代码如下：

```
# None 代表图片数量未知
x = tf.placeholder(tf.float32, [None,784])/255
# 将 input 重新调整结构，适用于卷积神经网络的特征提取
x_image = tf.reshape(x, [-1,28,28,1])

# 卷积核尺寸：5×5，输入通道：1，输出通道：32
W_conv1 = weight_variable([5, 5, 1, 32])
b_conv1 = bias_variable([32])
# 输出尺寸：28×28×32
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
# 输出尺寸：14×14×32
h_pool1 = max_pool(h_conv1)
```

因为 x 是训练集，所以需要用到 TensorFlow 中的占位符 tf.placeholder()，输入经过数值归一化之后，将其维度转换成图片维度。

该层采用的卷积核尺寸为$5\times5\times1\times32$，1 表示输入图片的通道数，32 表示该层输出的通道数。最后得到的 h_pool1 就是经过第 1 个卷积层和池化层后的输出，尺寸为$14\times14\times32$。

2. CONV2

接下来定义卷积层 2（包含池化层），代码如下：

```
# 卷积核尺寸：5×5，输入通道：32，输出通道：64
W_conv2 = weight_variable([5, 5, 32, 64])
b_conv2 = bias_variable([64])
# 输出尺寸：14×14×64
h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
# 输出尺寸：7×7×64
h_pool2 = max_pool(h_conv2)
```

该层采用的卷积核尺寸为$5\times5\times32\times64$，32 表示上一层的通道数，64 表示该层输出的通道数。最后得到的 h_pool2 就是经过第 2 个卷积层和池化层后的输出，尺寸为$7\times7\times64$。

3. FC1

经过两次卷积层和池化层之后，接下来定义全连接层，代码如下：

```
W_fc1 = weight_variable([7*7*64, 1024])
b_fc1 = bias_variable([1024])
h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])  # 展开为一维向量
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)
```

由于上一层的输出维度是$7\times7\times64$，因此在全连接层之前需要将其展开为一维向量作为输入，FC1 包含的神经元个数为 1024，使用的激活函数是 ReLU。这里的运算方法与传统的前馈

神经网络是类似的。

4. FC2

最后，定义第二个全连接层，即 Softmax 输出层，代码如下：

```
W_fc2 = weight_variable([1024, 10])
b_fc2 = bias_variable([10])
prediction = tf.nn.softmax(tf.matmul(h_fc1, W_fc2) + b_fc2)
```

Softmax 输出层包含的神经元个数为 10 个，因为数据集包含 10 个数字。

8.9.3 损失函数与优化算法

对于分类问题，损失函数使用交叉熵，梯度下降算法使用 Adam，学习率设置为$1e-4$。

```
# y是最终预测的结果
y = tf.placeholder(tf.float32, [None,10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y * tf.log(prediction),reduction_
indices=[1]))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
```

还需要定义一个计算准确率的函数，代码如下：

```
# 判断预测标签和实际标签是否匹配
correct_prediction = tf.equal(tf.argmax(prediction,1), tf.argmax(y,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction,"float"))
```

8.9.4 训练并测试

定义好模型、损失函数和梯度优化算法之后，就可以训练卷积神经网络模型了。

```
sess = tf.Session()
init = tf.global_variables_initializer()
sess.run(init)

for i in range(1000):
   batch_x, batch_y = mnist.train.next_batch(100)
   sess.run(train_step, feed_dict={x: batch_x, y: batch_y})
   if (i+1) % 50 == 0:
      print("train accuracy %.3f" % accuracy.eval(session = sess,
             feed_dict = {x:batch_x, y:batch_y}))
print("test accuracy %.3f" % accuracy.eval(session = sess,
   feed_dict = {x:mnist.test.images, y:mnist.test.labels}))
```

上述代码中，选择的每个小批量样本集中样本的数量为 100，迭代训练 1000 次。每隔 50 次，将训练集的准确率打印出来，最终计算模型对整个测试集的准确率并打印。

运行程序，得到测试集的准确率 0.971，可以说效果非常好。