# 卷积神经网络

## 从边缘特征信息聊起

在深度学习中，卷积神经网络（Convolutional Neural Network，CNN）被广泛应用于图像识别和计算机视觉任务。在介绍CNN之前，我们先来了解一下计算机视觉中的图像边缘特征。

边缘特征是图像中重要的视觉特征之一。它们表示了图像中物体边界的位置和形状。通过检测图像中的边缘，我们可以获取关键的形状信息，从而实现图像的分割、识别和分析等任务。

在CNN中，最核心的操作是卷积操作。使用卷积操作，可以提取图像的边缘特征。

图3-1展示了对图片提取边缘特征的结果。



图 ‑1

下一节中，我们将详细介绍卷积操作的原理和步骤，并探讨卷积操作如何提取图像的边缘特征。

## 什么是卷积

### 图解卷积运算

为了直观地理解卷积运算，让我们通过这个例子来感受一下卷积运算是怎样进行的。



图 ‑2

如图3-2所示，我们有两个矩阵参与卷积运算（用表示），得到一个新的矩阵。第一个矩阵为输入矩阵，实践中通常是输入图像的数字表示。第二个矩阵为**卷积核**（Kernel）。卷积核也常常称为滤波器（Filter），但在本章中统称为卷积核。第三个矩阵是卷积运算的结果，我们称其为**特征图**（Feature Map）。

我们首先将卷积核与输入矩阵的左上角对齐，然后逐元素相乘并求和，得到卷积输出的第一个元素。在这个例子中，我们计算，所以特征图左上角的结果就是2。

接着，我们移动卷积核，重复这个过程，继续计算特征图的下一个元素。



图 ‑3

在这里，我们计算，所以特征图的第二个位置是5。

在水平方向上，卷积核已无法再移动了，因此卷积核向下移动一格，并回到最左侧。



图 ‑4

同样地，计算，所以特征图第三个位置是14。

最后，用同样的方式可以得到：



图 ‑5

由此，就得到了特征图，卷积运算就完成了。

接下来，我们看看如何用卷积运算提取图像的边缘特征。

我们现在假设矩阵中的数字表示图像像素的亮度，因此数字越大的地方会越亮。



图 ‑6

在这个例子中，我们使用一个6x6矩阵作为输出，并用一个人工设计的3x3卷积核来进行卷积，最终得到了一个4x4的特征图。

这个卷积核就是一个垂直边缘检测器。使用这个卷积核进行卷积，就可以探测到图像的垂直边缘。原图像在中间有明显的垂直边缘将亮部和暗部分隔开，因此在特征图中，中间部分得到了-30的结果，这就表示原图像的中间部分有垂直边缘。

并且，如果我们将输入矩阵左右翻转：



图 ‑7

可见，所得特征图的中间部分变为了30。这说明，通过这种卷积的方式，不仅可以得知此处存在垂直边缘，还可以得知此处是从亮变暗还是从暗变亮。

### 填充与步幅

之前所用的垂直边缘检测器，即卷积核，是我们人工设计的。传统的边缘检测方法中，卷积核中的数字有多种设计，例如Sobel算子的设计是，Scharr算子的设计是，它们最大的区别就是使用了不同的卷积核设计。

随着深度学习的发展，计算机视觉工程师们的大脑里萌生了一种想法：为什么不让神经网络来学习怎样的卷积核才是最好的？

事实上，CNN就是在做这样的工作，但远远不局限于边缘特征提取。通过多个卷积层的堆叠，CNN可以学习到不同级别的特征。

但是，卷积运算会使得输出的特征图“缩水“。例如，3x3的输入通过2x2的卷积核卷积后，得到的是2x2的特征图。想象一下，要是我们再用2x2卷积核卷积一次，输出尺寸就变成1x1，这就无法继续卷积了！

另一方面，在卷积的过程中，图像边角的元素会更少地在计算中使用，这将导致卷积运算对图像边角的信息提取不充分，无法有效利用。

为了解决这些问题，我们引入**填充**（Padding）的技术。通过填充，我们在输入数据的边缘周围添加额外的像素，从而解决了问题。

如图3-7所示，使用与之前一样的3x3的输入矩阵和2x2的卷积核，但是加入了padding=1（意为在外侧添加1圈）的0填充，实际上就变成了在5x5矩阵上进行卷积，从而我们可以得到一个4x4的特征图作为输出。



图 ‑8

加入填充后，边角的元素也得到了充分的使用，特征图的尺寸也得到了了扩大。

然后，我们引入**步幅**（Stride）的概念。步幅，也称步长，它定义了卷积核在输入上每次滑动的距离。在之前的卷积中，我们使用的是步幅的默认值1，即每次滑动一格距离。但我们也可以设置stride=2，这样卷积核每次将滑动2格距离。

如图3-9所示，我们使用一个4x4的输入矩阵和2x2的卷积核，并设置stride=2，于是卷积核每次会在输入矩阵上滑动2格距离，最终将输出一个2x2的特征图。



图 ‑9

最后，我们给出一个计算输出特征图尺寸的公式：

其中，分别是输出特征图的高度和宽度，分别是卷积核的高度和宽度，是填充，是步幅。

## 构建第一个卷积网络：从MNIST到LeNet

在这一节中，我们将使用NumPy实现一个经典的CNN网络——LeNet。

在4.1节中，我们还将进一步详细地介绍LeNet。现在，让我们先简单地了解一下LeNet。

图3-10与图3-11展示了LeNet的结构。



图 ‑10 LeNet-5结构

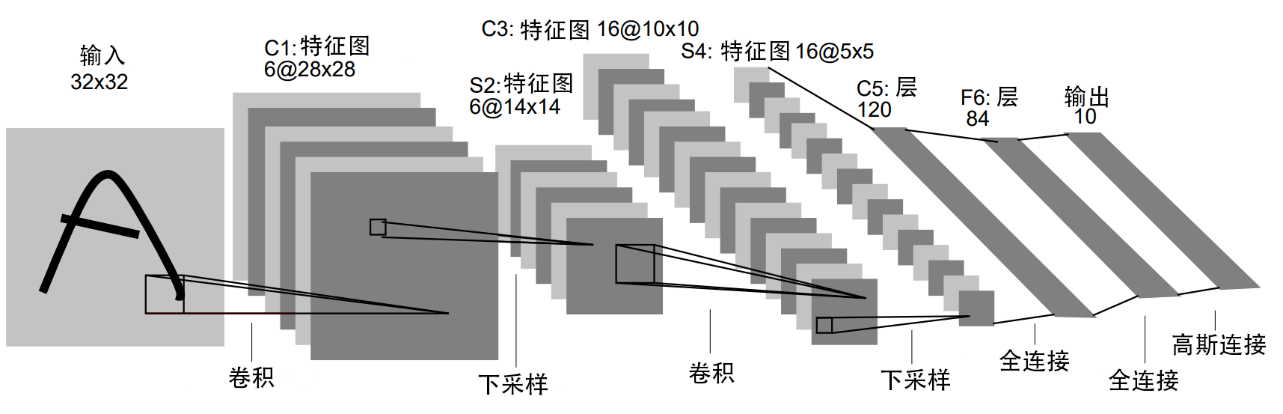


图 ‑11 LeNet-5结构[1]

可见，LeNet主要由三个结构组成：

* 卷积层
* 汇聚层
* 全连接层

接下来，我们将先从了解MNIST数据集开始，逐步了解卷积层、汇聚层、全连接层，并动手实现和训练一个LeNet，并将其用于手写数字识别任务。我们还将对LeNet的结构稍加改造，以便于取得更好的效果。

### MNIST数据集简介

MNIST数据集是一个广泛使用的手写数字识别数据集，常用于机器学习和深度学习的实验和基准测试。它由来自美国国家标准与技术研究所的两个数据集组成：训练集和测试集。训练集包含来自250位不同写者的60,000个样本图像，而测试集包含来自不同写者的10,000个样本图像。

MNIST数据集的图像尺寸为28x28像素，每个像素的灰度级别在0到255之间。每个图像都有一个相应的标签，表示图像中所绘制数字的实际值。标签是0到9的整数，与图像所表示的手写数字对应。

访问<http://yann.lecun.com/exdb/mnist/>以下载我们所需要的MNIST数据集。

接下来，我们开始进行代码实践。

首先导入我们需要的NumPy包：

1. import numpy as np

我们要能够从MNIST数据集中读取数据，因此需要定义加载数据集的load\_mnist()函数：

TODO: 加载数据集

在该函数中，我们通过frombuffer函数将数据集文件读取为NumPy数组。

在数据集中直接读取到的label只有一个数字，而在网络中需要以独热编码的形式参与计算，所以在这里先进行转换。

此外，因为LeNet的输入尺寸为32x32，我们还需要通过pad函数为28x28的MNIST图像添加填充。

通过此函数，我们可以获得四个变量：

train\_images：训练图像数据，类型为NumPy数组，形状为(60000, 32, 32, 1)。

train\_labels：训练图像对应的标签，类型为NumPy数组，形状为(60000, 10)。

test\_images：测试图像数据，类型为NumPy数组，形状为(10000, 28, 28, 1)。

test\_labels：测试图像对应的标签，类型为NumPy数组，形状为(10000, 10)。

这些变量将于后续用在训练与测试当中。

### 辅助函数

在开始搭建CNN的各个部件以前，我们先定义一些之后会使用到的辅助函数。

1. def relu(x):
2. return np.maximum(0, x)
3. def softmax(x):
4. e\_x = np.exp(x - np.max(x, axis=1, keepdims=True))
5. return e\_x / np.sum(e\_x, axis=1, keepdims=True)
6. def relu\_backward(grad\_output, input):
7. return grad\_output \* (input > 0)
8. def rot180(kernel):
9. return np.rot90(kernel, 2, axes=(2, 3))
10. def cross\_entropy\_loss(y, t):
11. # 批量平均损失
12. return -np.sum(t \* np.log(y + 1e-7)) / y.shape[0]
13. def softmax\_backward(y, t):
14. return (y - t) / y.shape[0]

### 卷积层

卷积层（Convolution Layer）是CNN的重要组成部分，它由一系列卷积核（Convolutional Kernel）组成，每个卷积核对输入数据进行卷积运算。我们只需要指定卷积核的尺寸和个数，而卷积核的每个位置的数字都是一个可学习的参数。

输入进入LeNet后，首先进入卷积层。因此，我们先来定义卷积层。

1. class ConvLayer:
2. def \_\_init\_\_(self, in\_channels, out\_channels, kernel\_size, stride=1, padding=0):
3. self.in\_channels = in\_channels
4. self.out\_channels = out\_channels
5. self.kernel\_size = kernel\_size
6. self.stride = stride
7. self.padding = padding
8. # He initialization
9. self.weights = np.random.randn(out\_channels, in\_channels, kernel\_size, kernel\_size) \* np.sqrt(
10. 2.0 / (in\_channels \* kernel\_size \* kernel\_size))
11. self.bias = np.zeros((out\_channels, 1))
12. self.input = None
13. self.grad\_weights = None
14. self.grad\_bias = None
15. self.conv\_output = None

在卷积层的初始化中，我们定义了通道数、卷积核大小、步幅、填充等卷积操作所需的参数变量，并使用He初始化进行了权重和偏差的初始化操作。我们还定义了输入、权重梯度、偏差梯度、卷积结果等变量，这些变量将用于反向传播与梯度更新。

我们为卷积层加入了偏差。虽然对于CNN而言，即使没有偏差，卷积核也能够工作得很好，但理论上来说偏差能增强模型的表达能力，所以我们还是为卷积层加入了偏差。

1. **正向传播**

我们将卷积层在正向传播中使用的卷积运算操作定义为conv2d函数：

1. def conv2d(input, kernel, \*, padding=0, stride=1):
2. batch\_size, num\_channels, in\_height, in\_width = input.shape
3. out\_channels, \_, kernel\_height, kernel\_width = kernel.shape
4. out\_height = int((in\_height - kernel\_height + 2 \* padding) / stride) + 1
5. out\_width = int((in\_width - kernel\_width + 2 \* padding) / stride) + 1
6. output = np.zeros((batch\_size, out\_channels, out\_height, out\_width))
7. padded\_input = np.pad(input, ((0, 0), (0, 0), (padding, padding), (padding, padding)), 'constant')
8. for b in range(batch\_size):
9. for c in range(out\_channels):
10. for h\_out in range(out\_height):
11. for w\_out in range(out\_width):
12. h\_start = h\_out \* stride
13. h\_end = h\_start + kernel\_height
14. w\_start = w\_out \* stride
15. w\_end = w\_start + kernel\_width
16. receptive\_field = padded\_input[b, :, h\_start:h\_end, w\_start:w\_end]
17. output[b, c, h\_out, w\_out] = np.sum(receptive\_field \* kernel[c])
18. return output

这段代码实现了一个二维卷积操作，使用了4个for循环，逐个计算特征图结果。

在卷积层中，使用ReLU作为激活函数：

LeNet的论文中使用Sigmoid函数作为激活函数，但现代CNN常使用ReLU作为激活函数来避免梯度消失问题，所以我们在这里使用ReLU作为卷积层的激活函数。

正向传播过程：

1. def forward(self, input):
2. self.input = input
3. self.conv\_output = conv2d(input, self.weights, stride=self.stride, padding=self.padding)
4. # conv\_output: (B, OC, OH, OW)
5. # bias: (OC, 1) -> (1, OC, 1, 1)
6. output = relu(self.conv\_output + self.bias.reshape((1, -1, 1, 1)))
7. return output

注释中提供了张量形状及计算过程中的形状变化。B表示批量大小batch\_size，OC表示输出通道out\_channel，OH表示输出高度out\_height，OW表示输出宽度out\_width。

我们将ReLU函数置于卷积层内部，计算卷积结果后加上偏差，通过ReLU函数得到输出的特征图。

1. **反向传播**

卷积层的反向传播过程较为复杂，总体来说基于这两个公式：

其中x表示卷积层的输入，F表示卷积核，y表示未经过激活函数的卷积结果。这两个公式的推导较为复杂，我们不多描述。

函数表示将输入旋转180度。而FullConvolution函数，可将其视作一种从右下到左上的“反卷积”。图3-12展示了这种“反卷积”的计算过程。可见，这种运算与之前所述卷积的不同不仅在于方向，还在于卷积核可以超出输入矩阵之外，只要有交叉的部分即可。这样将翻转的卷积核与梯度经过“反卷积”以后，我们就可以得到与x尺寸相同的梯度了。



图 ‑12 反卷积示例

反向传播过程的代码：

1. def backward(self, grad\_output):
2. # batch\_size, num\_channels, in\_height, in\_width = grad\_output.shape
3. grad\_output = relu\_backward(grad\_output, self.conv\_output)
4. # rotated\_kernel: [OC, IC, KS, KS] -> [IC, OC, KS, KS]
5. # grad\_output: [B, OC, OH, OW]
6. # grad\_input: [IC, B, H, W] -> [B, IC, H, W]
7. rotated\_kernel = rot180(self.weights)
8. rotated\_kernel = np.transpose(rotated\_kernel, (1, 0, 2, 3))
9. grad\_input = full\_conv(rotated\_kernel, grad\_output)
10. grad\_input = np.transpose(grad\_input, (1, 0, 2, 3))
11. # input: [B, IC, H, W] -> [IC, B, H, W]
12. # grad\_output: [B, OC, OH, OW] -> [OC, B, OH, OW]
13. # grad\_weights: [IC, OC, KS, KS] -> [OC, IC, KS, KS]
14. self.grad\_weights = conv2d(np.transpose(self.input, (1, 0, 2, 3)), np.transpose(grad\_output, (1, 0, 2, 3)),
15. padding=self.padding)
16. self.grad\_weights = np.transpose(self.grad\_weights, (1, 0, 2, 3))
17. # grad\_output: [B, OC, OH, OW] -> [OC, 1]
18. self.grad\_bias = np.sum(grad\_output, axis=(0, 2, 3)).reshape(self.out\_channels, 1)
19. return grad\_input

注释中依然提供了张量形状变化过程，除去之前说明过的外，IC表示输入通道in\_channel，KS表示卷积核尺寸kernel\_size，H表示输入的高度height，W表示输出的宽度width。

### 汇聚层

**汇聚层**（Pooling Layer），也常叫**池化层**，在LeNet的原论文中称为下采样层（Subsampling Layer）。

尽管不太严谨，但是直观地说，下采样就是让图片“缩水”的过程，即输出尺寸小于输入尺寸。从这个角度来看，卷积当然也可以是一种下采样。

我们介绍一下汇聚操作是如何进行的。对于汇聚操作而言，最常用的有两种方法：平均汇聚（Average Pooling）和最大汇聚（Max Pooling）。

对于汇聚操作而言，有步幅和窗口尺寸两个参数。汇聚层的窗口也常称为卷积核或者滤波器，但与卷积层不同的是，汇聚层并没有可学习的参数，也并不进行卷积操作，而是遵循固定的规则进行操作。

汇聚操作会根据步幅与窗口尺寸将整个输入分为许多区域，就和卷积操作中卷积核在输入上可能会处于的不同位置一样。例如，图3-12展示了一个4x4的输入矩阵，在stride=2, kernel\_size=2时，窗口尺寸为2x2，每次移动2个距离，因此输入会被分割为4个区域。



图 ‑13

1. **平均汇聚**

在平均汇聚下，我们取每个窗口内所有数字的平均值作为该窗口在特征图中的结果。如图3-13，我们将得到一个2x2的特征图。



图 ‑14

比如，在第一个窗口内，。

1. **最大汇聚**

在最大汇聚下，我们取每个窗口内所有数字的最大值作为该窗口在特征图中的结果。如图3-14，我们将得到一个2x2的特征图。



图 ‑15

很明显，我们对每个窗口取了其中最大的数字作为输出特征图中的结果。

我们在实现时，将使用平均汇聚方法。

下面是汇聚层的代码实现：

1. class AvgPool:
2. def \_\_init\_\_(self, kernel\_size, stride=1):
3. self.kernel\_size = kernel\_size
4. self.stride = stride
5. def forward(self, x):
6. batch\_size, num\_channels, in\_height, in\_width = x.shape
7. out\_height = int((in\_height - self.kernel\_size) / self.stride) + 1
8. out\_width = int((in\_width - self.kernel\_size) / self.stride) + 1
9. output = np.zeros((batch\_size, num\_channels, out\_height, out\_width))
10. for b in range(batch\_size):
11. for c in range(num\_channels):
12. for h\_out in range(out\_height):
13. for w\_out in range(out\_width):
14. h\_start = h\_out \* self.stride
15. h\_end = h\_start + self.kernel\_size
16. w\_start = w\_out \* self.stride
17. w\_end = w\_start + self.kernel\_size
18. receptive\_field = x[b, c, h\_start:h\_end, w\_start:w\_end]
19. output[b, c, h\_out, w\_out] = np.mean(receptive\_field)
20. return output
21. # average pooling layer backward
22. def backward(self, grad\_output):
23. batch\_size, num\_channels, out\_height, out\_width = grad\_output.shape
24. in\_height = int((out\_height - 1) \* self.stride + self.kernel\_size)
25. in\_width = int((out\_width - 1) \* self.stride + self.kernel\_size)
26. grad\_input = np.zeros((batch\_size, num\_channels, in\_height, in\_width))
27. for b in range(batch\_size):
28. for c in range(num\_channels):
29. for h\_out in range(out\_height):
30. for w\_out in range(out\_width):
31. h\_start = h\_out \* self.stride
32. h\_end = h\_start + self.kernel\_size
33. w\_start = w\_out \* self.stride
34. w\_end = w\_start + self.kernel\_size
35. grad\_input[b, c, h\_start:h\_end, w\_start:w\_end] = grad\_output[b, c, h\_out, w\_out] / (
36. self.kernel\_size \* self.kernel\_size)
37. return grad\_input

有关汇聚层的反向传播，对于平均汇聚而言，我们只要将梯度填充为输入尺寸，然后将梯度平均分到扩展后的空间即可。

如图3-15的例子，若要将4x4的矩阵通过步幅为2、窗口尺寸为2x2的平均汇聚层后得到2x2的输出。反向传播传来的梯度尺寸也为2x2，将其填充为4x4后，将梯度平均分配到其所对应的输入矩阵的区域中。



图 ‑16

### 全连接层

下面是LeNet的最后一块拼图：全连接层。

全连接层就像是多层感知机中的隐层，它做的事情很简单。从S4到C5，我们将5x5x16的特征图拉伸为一维的尺寸为400的向量，然后将这个向量输入到含有若干神经元的全连接层中。C5输出的结果不必再经过拉伸，直接输入到F6再经过一次全连接。

不过，在进入全连接层以前，我们需要将特征图铺平到一维，因此需要先定义Flatten类：

1. class Flatten:
2. def \_\_init\_\_(self):
3. self.input = None
4. def forward(self, x):
5. self.input = x
6. batch\_size = x.shape[0]
7. return x.reshape(batch\_size, -1)
8. def backward(self, grad\_output):
9. return grad\_output.reshape(self.input.shape)

Flatten类实现的功能很简单，只是改变了输入张量的形状而已。有了Flatten类以后，我们就可以进一步实现全连接层：

1. class Linear:
2. def \_\_init\_\_(self, in\_features, out\_features):
3. self.in\_features = in\_features
4. self.out\_features = out\_features
5. self.weights = np.random.randn(in\_features, out\_features) \* np.sqrt(2.0 / (in\_features + out\_features))
6. self.bias = np.zeros((1, out\_features))
7. self.input = None
8. self.grad\_weights = None
9. self.grad\_bias = None
10. self.output = None
11. def forward(self, input):
12. self.input = input
13. self.output = np.dot(input, self.weights) + self.bias
14. return self.output
15. def backward(self, grad\_output):
16. # grad\_weights = input.T \* grad\_output
17. # input: [B, I] -> [I, B]
18. # grad\_output: [B, O]
19. self.grad\_weights = np.dot(self.input.T, grad\_output)
20. # grad\_bias
21. self.grad\_bias = np.sum(grad\_output, axis=0, keepdims=True)
22. # grad\_input = grad\_output \* weights.T
23. # grad\_output: [B, O]
24. # weights: [I, O] -> [O, I]
25. # grad\_input: [B, I]
26. return np.dot(grad\_output, self.weights.T)

在反向传播部分提供了张量形状变化的注释。I表示in\_features，O表示out\_features。

Linear类完成了全连接层的工作。在这里，我们没有把激活函数定义在类内部，这是因为我们的LeNet的三个全连接层中，前两个使用ReLU作为激活函数，最后一个使用Softmax作为激活函数。

这里的反向传播公式是通过计算图推导得来的：

### LeNet类

最后，我们使用定义好的这些类和函数，搭建一个LeNet：

1. class LeNet:
2. def \_\_init\_\_(self):
3. self.conv1 = ConvLayer(in\_channels=1, out\_channels=6, kernel\_size=5)
4. self.avgpool1 = AvgPool(kernel\_size=2, stride=2)
5. self.conv2 = ConvLayer(in\_channels=6, out\_channels=16, kernel\_size=5)
6. self.avgpool2 = AvgPool(kernel\_size=2, stride=2)
7. self.flatten = Flatten()
8. self.linear1 = Linear(16 \* 5 \* 5, 120)
9. self.linear2 = Linear(120, 84)
10. self.linear3 = Linear(84, 10)
11. self.loss = []
12. def forward(self, x):
13. x = self.conv1.forward(x)
14. x = self.avgpool1.forward(x)
15. x = self.conv2.forward(x)
16. x = self.avgpool2.forward(x)
17. x = self.flatten.forward(x)
18. x = self.linear1.forward(x)
19. x = relu(x)
20. x = self.linear2.forward(x)
21. x = relu(x)
22. x = self.linear3.forward(x)
23. x = softmax(x)
24. return x
25. def backward(self, grad\_output):
26. # softmax的反向传播直接通过y-t传入即可，必须使用交叉熵损失函数
27. grad\_output = self.linear3.backward(grad\_output)
28. grad\_output = relu\_backward(grad\_output, self.linear2.output)
29. grad\_output = self.linear2.backward(grad\_output)
30. grad\_output = relu\_backward(grad\_output, self.linear1.output)
31. grad\_output = self.linear1.backward(grad\_output)
32. grad\_output = self.flatten.backward(grad\_output)
33. grad\_output = self.avgpool2.backward(grad\_output)
34. grad\_output = self.conv2.backward(grad\_output)
35. grad\_output = self.avgpool1.backward(grad\_output)
36. grad\_output = self.conv1.backward(grad\_output)
37. return grad\_output

### 训练与测试

定义好了网络，我们就可以开始训练了。

1. def train(model, train\_images, train\_labels, epochs, learning\_rate, batch\_size):
2. for epoch in range(epochs):
3. correct = 0
4. loss\_list = []
5. batch\_num = 0
6. for i in range(0, len(train\_images), batch\_size):
7. batch\_num += 1
8. batch\_images = train\_images[i:i + batch\_size]
9. batch\_labels = train\_labels[i:i + batch\_size]
10. # forward
11. output = model.forward(batch\_images)
12. loss = cross\_entropy\_loss(output, batch\_labels)
13. loss\_list.append(loss)
14. # backward
15. grad\_output = softmax\_backward(output, batch\_labels)
16. model.backward(grad\_output)
17. # update parameters
18. for layer in [model.conv1, model.conv2, model.linear1, model.linear2, model.linear3]:
19. layer.weights -= learning\_rate \* layer.grad\_weights
20. layer.bias -= learning\_rate \* layer.grad\_bias
21. # calculate accuracy
22. pred = np.argmax(output, axis=1)
23. correct += np.sum(pred == np.argmax(batch\_labels, axis=1))
24. print(f'iter {i / batch\_size}, loss: {loss}')
25. model.loss.append(loss\_list)
26. avg\_loss = np.mean(np.array(loss\_list))
27. print('Epoch %d/%d, avg\_loss: %.3f, train accuracy: %.3f' % (
28. epoch + 1, epochs, avg\_loss, correct / len(train\_images)))

设置batch\_size为32，学习率为0.01。我的模型在完整训练了一个epoch以后，在测试集上得到了81.9%的准确度结果。

如图3-17，我们随机从测试集中选择9张图片进行测试，将图片与结果一同绘制，可更直观地看到模型的检测结果。

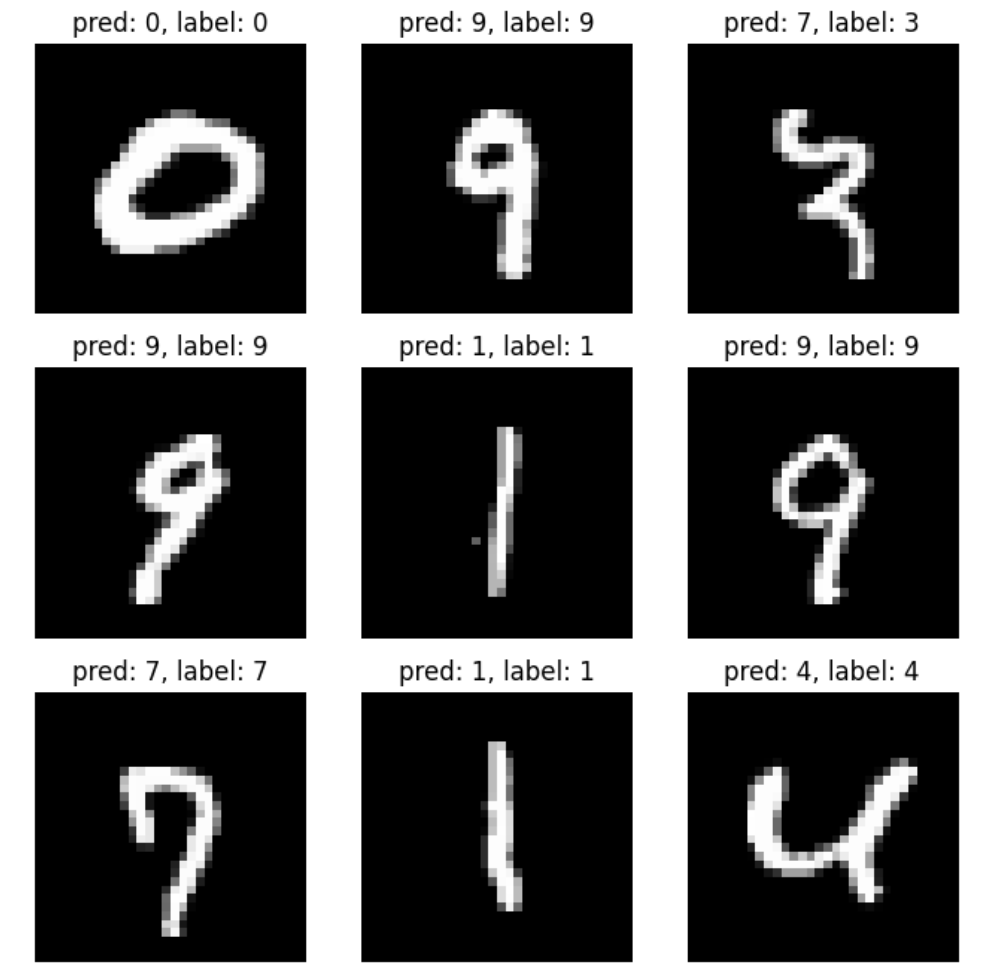


图 ‑17

我们在实现卷积操作时，使用了4重for循环，这种方法虽然实现起来比较容易，但计算速度却十分缓慢，因此我只为此模型训练了一个epoch。现代深度学习框架（如PyTorch, TensorFlow等）对这些计算算法进行了很多优化，并且支持GPU并行计算，因而今后如果使用深度学习框架进行训练，速度会快很多。

## 小结

在本章中，我们展示了卷积网络是如何工作的，卷积操作是如何进行的。我们介绍了卷积网络的各个部分的工作原理，并基于NumPy实现了一个LeNet模型，实现了手写数字识别任务。

我们提供了Jupyter Notebook形式的源代码文件，其中有更加详尽的注释。

## 参考文献

[1] Y. Lecun, L. Bottou, Y. Bengio和P. Haffner, 《Gradient-based learning applied to document recognition》, *Proc. IEEE*, 卷 86, 期 11, 页 2278–2324, 11月 1998, doi: 10.1109/5.726791.

[2] P. Solai, 《Convolutions and Backpropagations》, *Medium*, 2018年4月18日. https://pavisj.medium.com/convolutions-and-backpropagations-46026a8f5d2c (见于 2023年5月24日).