# 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。

我们现在假设输入矩阵中的数字表示每个像素的亮度，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0与10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。

## 卷积与自相关的区别

如果你有学习过信号处理或者数学的相关部分，你也许会发现，我们之前描述的卷积运算，严格来说其实是互相关运算，而不是卷积运算。这的确是错误的，但这种错误的叫法已经是行业内的约定俗成了。

自相关是一种在信号处理和统计学中常用的操作，用于衡量信号与其自身在不同时间点上的相似度。自相关操作通过在信号上应用滞后和加权求和来计算自相关系数。它在时间序列分析、信号处理和模式识别等领域具有广泛的应用。

在CNN中，卷积操作与自相关有一些相似之处，但也存在一些关键的区别：

卷积操作的目标是从输入数据中提取特征，以用于后续的任务，如图像分类或目标检测。而自相关的目标是衡量信号自身的相似性，通常用于分析信号的周期性、重复性或相关性。

自相关操作中使用滞后和加权求和的方式来计算自相关系数。而在卷积操作中，卷积核是通过与输入数据的对应位置进行逐元素相乘和求和来生成输出特征图，而没有使用滞后或加权的概念。

在CNN中，卷积核的权重是通过反向传播算法进行学习的。通过大量的训练数据和优化算法，CNN可以自动学习到最佳的卷积核权重，从而提取最有用的特征。而自相关操作通常不涉及学习，而是通过事先定义的滞后和加权方式进行计算。

在CNN中，卷积操作的主要作用是从输入数据中提取特征，捕捉局部模式和结构。通过多个卷积层的堆叠和参数共享机制，CNN可以逐渐提取更加抽象和高级的特征，从而实现更准确的分类和预测。

相比之下，自相关操作更多地用于信号分析和模式识别领域，用于衡量信号自身的相似性和相关性。

## 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

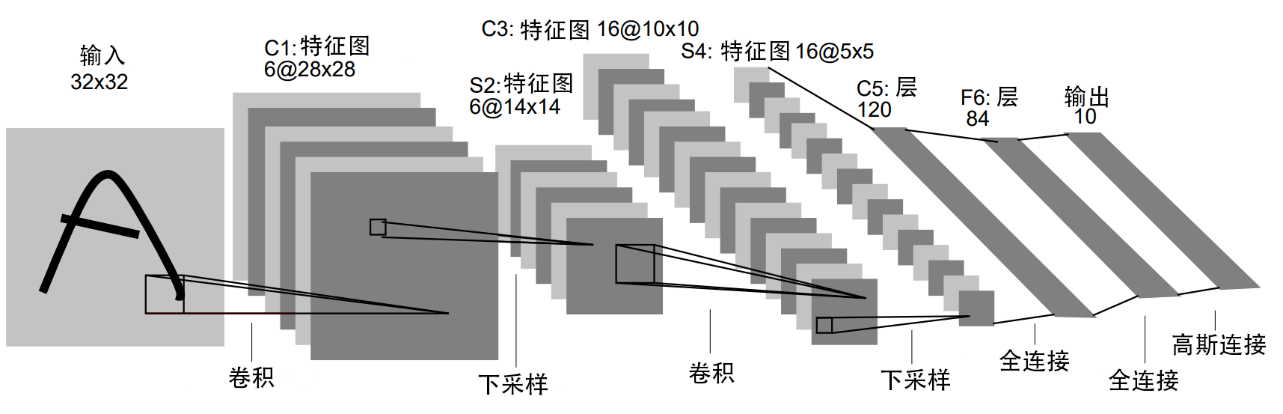


图 ‑11 LeNet-5结构[1]

可见，LeNet主要由三个部分组成：

* 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。

### 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：

1. # 加载mnist数据集
2. def load\_mnist():
3. # 加载训练集
4. with open('train-images-idx3-ubyte', 'rb') as f:
5. train\_images = np.frombuffer(f.read(), dtype=np.uint8, offset=16).reshape(-1, 28, 28, 1)
6. with open('train-labels-idx1-ubyte', 'rb') as f:
7. train\_labels = np.frombuffer(f.read(), dtype=np.uint8, offset=8)
8. # 加载测试集
9. with open('t10k-images-idx3-ubyte', 'rb') as f:
10. test\_images = np.frombuffer(f.read(), dtype=np.uint8, offset=16).reshape(-1, 28, 28, 1)
11. with open('t10k-labels-idx1-ubyte', 'rb') as f:
12. test\_labels = np.frombuffer(f.read(), dtype=np.uint8, offset=8)
13. return train\_images, train\_labels, test\_images, test\_labels

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(N, 28, 28, 1)，其中N是训练样本的数量。

train\_labels：训练图像对应的标签，类型为NumPy数组，形状为(N,)，其中N是训练样本的数量。

test\_images：测试图像数据，类型为NumPy数组，形状为(M, 28, 28, 1)，其中M是测试样本的数量。

test\_labels：测试图像对应的标签，类型为NumPy数组，形状为(M,)，其中M是测试样本的数量。

这些变量将于后续用在训练与结果评估当中。

### 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。

卷积操作我们在之前的章节中已经介绍过了，因此我们直接来看如何用代码实现卷积层：

首先，我们定义ReLU与Softmax激活函数：

1. def relu(x):
2. return np.maximum(x, 0)
3. def softmax(x):
4. exp\_x = np.exp(x - np.max(x, axis=1, keepdims=True))
5. return exp\_x / np.sum(exp\_x, axis=1, keepdims=True)

如果你不了解ReLU和Softmax，就先简单地记住：ReLU是一种激活函数，Softmax用于多分类任务的输出。

然后，我们定义卷积层：

1. class ConvLayer:
2. def \_\_init\_\_(self, input\_channels, output\_channels, kernel\_size, stride, padding):
3. self.input\_channels = input\_channels
4. self.output\_channels = output\_channels
5. self.kernel\_size = kernel\_size
6. self.stride = stride
7. self.padding = padding
8. self.weights = np.random.randn(output\_channels, input\_channels, kernel\_size, kernel\_size) / np.sqrt(input\_channels \* kernel\_size \* kernel\_size)
9. def forward(self, x):
10. batch\_size, input\_height, input\_width, \_ = x.shape
11. output\_height = (input\_height - self.kernel\_size + 2 \* self.padding) // self.stride + 1
12. output\_width = (input\_width - self.kernel\_size + 2 \* self.padding) // self.stride + 1
13. # 对输入进行填充
14. padded\_x = np.pad(x, ((0, 0), (self.padding, self.padding), (self.padding, self.padding), (0, 0)), mode='constant')
15. # 初始化输出特征图
16. output = np.zeros((batch\_size, output\_height, output\_width, self.output\_channels))
17. # 对每个输入图像进行卷积操作
18. for b in range(batch\_size):
19. for c\_out in range(self.output\_channels):
20. for h\_out in range(output\_height):
21. for w\_out in range(output\_width):
22. h\_start = h\_out \* self.stride
23. h\_end = h\_start + self.kernel\_size
24. w\_start = w\_out \* self.stride
25. w\_end = w\_start + self.kernel\_size
26. receptive\_field = padded\_x[b, h\_start:h\_end, w\_start:w\_end, :]
27. output[b, h\_out, w\_out, c\_out] = np.sum(receptive\_field \* self.weights[c\_out]) # 点乘卷积
28. return output

在卷积层类的实现中，我们定义了以下属性：

* input\_channels：输入通道数，表示输入特征图的通道数。
* output\_channels：输出通道数，表示卷积操作后得到的特征图的通道数。
* kernel\_size：卷积核大小，表示卷积核的宽度和高度。
* stride：步幅大小，表示卷积操作时卷积核的滑动步幅。
* padding：填充大小，表示在输入特征图周围添加的零填充的宽度和高度。
* weights：卷积核权重，用于卷积操作。

### 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）。

Todo: 介绍下采样操作

定义汇聚层：

1. class MaxPoolingLayer:
2. def \_\_init\_\_(self, kernel\_size):
3. self.kernel\_size = kernel\_size
4. def forward(self, x):
5. batch\_size, num\_channels, height, width = x.shape
6. pool\_height = height // self.kernel\_size
7. pool\_width = width // self.kernel\_size
8. pooled = np.zeros((batch\_size, num\_channels, pool\_height, pool\_width))
9. for i in range(pool\_height):
10. for j in range(pool\_width):
11. start\_h = i \* self.kernel\_size
12. end\_h = start\_h + self.kernel\_size
13. start\_w = j \* self.kernel\_size
14. end\_w = start\_w + self.kernel\_size
15. pool\_region = x[:, :, start\_h:end\_h, start\_w:end\_w]
16. pooled[:, :, i, j] = np.amax(pool\_region, axis=(2, 3))
17. return pooled

### 全连接层

Todo: 简要介绍全连接层

下面是LeNet的最后一块拼图，我们定义全连接层：

1. class FullyConnectedLayer:
2. def \_\_init\_\_(self, input\_size, output\_size):
3. self.weights = np.random.randn(input\_size, output\_size) / np.sqrt(input\_size)
4. self.biases = np.zeros(output\_size)
5. def forward(self, x):
6. batch\_size = x.shape[0]
7. x = x.reshape(batch\_size, -1) # 展平特征图为一维向量
8. # 计算线性变换
9. linear\_output = np.dot(x, self.weights) + self.biases
10. # 应用激活函数
11. output = relu(linear\_output)
12. return output

在全连接层类的实现中，我们定义了以下属性和方法：

* input\_size：输入大小，表示全连接层的输入特征向量的维度。
* output\_size：输出大小，表示全连接层的输出特征向量的维度。
* weights：权重矩阵，用于对输入特征向量进行线性变换。
* biases：偏置项，用于偏移线性变换的结果。
* forward方法：前向传播方法，接受输入特征向量并返回经过线性变换和激活函数处理后的输出特征向量。

### 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 = ConvolutionalLayer(in\_channels=1, out\_channels=6, kernel\_size=5)
4. self.pool1 = MaxPoolingLayer(kernel\_size=2)
5. self.conv2 = ConvolutionalLayer(in\_channels=6, out\_channels=16, kernel\_size=5)
6. self.pool2 = MaxPoolingLayer(kernel\_size=2)
7. self.fc1 = FullyConnectedLayer(input\_size=16 \* 4 \* 4, output\_size=120)
8. self.fc2 = FullyConnectedLayer(input\_size=120, output\_size=84)
9. self.fc3 = FullyConnectedLayer(input\_size=84, output\_size=10)
10. def forward(self, x):
11. x = self.conv1.forward(x)
12. x = self.pool1.forward(x)
13. x = self.conv2.forward(x)
14. x = self.pool2.forward(x)
15. x = self.fc1.forward(x)
16. x = self.fc2.forward(x)
17. x = self.fc3.forward(x)
18. return x

### 训练与测试

Todo: 跑个结果出来