# 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，我们先来了解一下图像处理中的边缘特征以及它们在卷积神经网络中的重要性。

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，通过卷积操作，CNN可以自动学习提取图像中的特征，并用于分类、目标检测和图像生成等任务。卷积操作可以有效地捕捉局部特征，并通过多个卷积层的堆叠来提取更加抽象和高级的特征。

在后续的小节中，我们将详细介绍卷积操作的原理和步骤，并探讨如何构建一个简单的卷积神经网络来解决图像分类问题。通过理解卷积操作的基本概念和作用，我们可以更好地理解CNN的工作原理，并在实践中灵活运用它们。

## 什么是卷积

CNN通常由卷积层、池化层、全连接层构成。图3-1展示了一个经典的CNN——LeNet的结构。

（插图）

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。卷积运算是一种局部感知的运算方式，它通过滑动窗口的方式在输入数据上提取局部特征。卷积核由一组权重构成，与输入数据进行逐元素乘积，并将结果相加得到卷积输出。

为了更直观地理解卷积运算，让我们看看这个例子，来感受一下卷积运算是怎样进行的。

假设我们有一个3x3的输入矩阵和一个2x2的卷积核。输入矩阵的每个元素表示图像的像素值，卷积核的元素表示卷积核的权重。

（插图）

如图3-2所示，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。重复这个过程，移动卷积核，继续计算卷积输出的下一个元素，直到将卷积核移动到输入矩阵的最后一个位置，我们得到了一个新的矩阵，称为特征图（Feature Map）。

然而，这种卷积会使得输出的特征图“缩水“。3x3的输入经过卷积后变成了2x2。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！

为了解决这个问题，我们引入填充（Padding）的概念。通过填充，在输入数据的边缘周围添加额外的像素，从而可以控制输出特征图的大小。

如图3-3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的填充，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。

（插图）

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。之前的卷积中，我们使用的是步幅默认值1，即每次滑动一个距离。但我们也可以设置stride=2，这样卷积核每次将滑动2个距离。

如图3-4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，于是将输出一个2x2的特征图。

（插图）

于是，我们可以得到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

CNN中的卷积，严格来说其实是一种互相关运算，而不是卷积运算。这种错误的叫法沿用下来，是历史遗留的原因。在卷积神经网络（CNN）的发展初期，人们将这种操作误称为卷积，并将其与信号处理中的连续信号卷积操作混淆。

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（lag）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

现在让我们来实践一下，从简单的MNIST数据集开始构建我们的第一个卷积神经网络（CNN）。我们将逐步构建LeNet模型，这是一个经典的卷积神经网络模型，我们将其用于手写数字识别任务。

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。

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，我们获得了四个变量：

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。

接着，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。

再定义池化层：

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

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。

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