卷积层的python实现

卷积网络采用卷积层来实现局部连接和参数共享。卷积网络的核心是卷积运算。

1.卷积运算及代码实现

卷积运算是线性滤波，对于图像中的每一个像素，计算以该像素为中心的像素和卷积核的内积，并将其作为该像素的新值。

卷积运算公式：

$$g(i,j)=\sum_{m=-1,n=-1}^{m=1,n=1} f(i+m,j+n)h(m,n)$$

$$g=f^{*}h$$

其中（i，j）是中心像素的坐标，i = 1，2，…，h，j = 1，2，…，w。此处h是图像高度，w是图像宽度。卷积需要遍历整个图像。f是原图像（二维矩阵），g是“新图像”，h是卷积核(3*3).

当对图像边界进行卷积时，卷积核的一部分位于图像外面，无像素与之相乘，此处有两种策略：

（1）舍弃图像边缘：

这样会使得“新图像”尺寸缩小（对于3x3卷积核，每边会减小1）

（2）采用像素填充技巧：

人为指定位于图像外的像素值，使得卷积核能与之相乘。像素填充的两种方式:0填充、复制边缘像素。卷积网络中普遍采用0填充方式。

卷积核经典取值：

图像模糊：[[1, 1, 1], [1, 1, 1],[1, 1, 1]] / 9

图像锐化：[[-1,-1,-1], [-1, 9, -1], [-1, -1, -1]]

边缘检测：

sobel：[[-1, -2, -1], [0, 0, 0], [1, 2, 1]] 及其转置

prewitt: [[-1, -1, -1], [0, 0, 0], [1, 1, 1]] 及其转置

Laplace: [[1, 1, 1], [1, -8, 1], [1, 1, 1]]

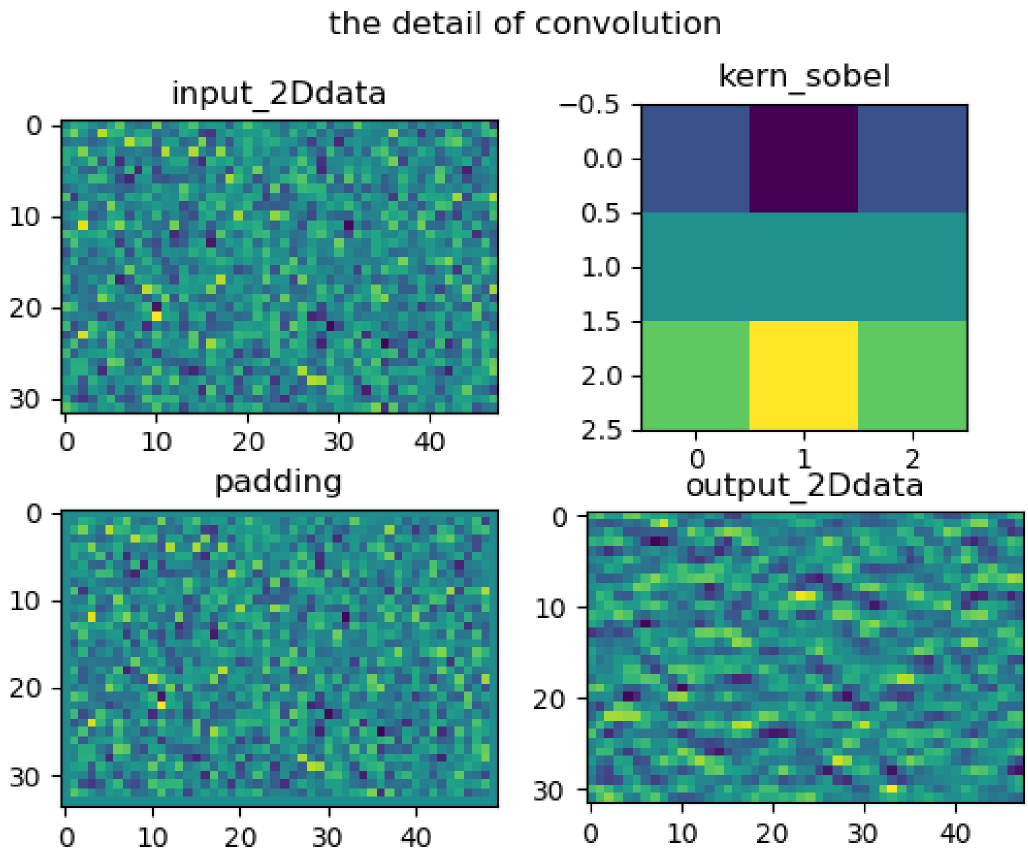
用于边缘检测的卷积核可以检测到图像的边缘，因为在图像的边缘地区，像素值的变化剧烈，而在平滑区域，像素值基本一致。计算局部窗口的像素差能区分边缘和平滑区域，像素差大的为边缘，像素差接近0的为平滑区域。

卷积运算细节的python实现：

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
```

```python
import numpy as np
import matplotlib.pyplot as plt
h = 32 # 图像高度
w = 48 # 图像深度

input_2Ddata = np.random.randn(h, w) # 随机生成二维输入数据
output_2Ddata = np.zeros(shape=(h, w)) # 卷积输出尺寸与输入一样
kern = np.random.randn(3, 3) # 3x3卷积核
kern_sobel = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]], dtype=np.float64)# sobel卷积核

padding = np.zeros(shape=(h+2, w+2)) # 图像边缘使用0填充
padding[1:-1, 1:-1] = input_2Ddata # 填充图像边缘
for i in range(h):
 for j in range(w):
 window = padding[i:i+3, j:j+3] # 中心像素（i, j）的局部窗口
 output_2Ddata[i, j] = np.sum(kern_sobel*window) # 卷积运算，并将结果存入output_2Ddata中

# 使用matplotlib画图
plt.subplot(2,2,1)
plt.imshow(input_2Ddata)
plt.title('input_2Ddata')

plt.subplot(2,2,2)
plt.imshow(kern_sobel)
plt.title('kern_sobel')

plt.subplot(2,2,3)
plt.imshow(padding)
plt.title('padding')

plt.subplot(2,2,4)
plt.imshow(output_2Ddata)
plt.title('output_2Ddata')
plt.suptitle('the detail of convolution')

plt.show()

```

运行结果：



2.卷积层及代码初级实现

卷积网络一般为3D特征图，3D特征图表示为[H*W*D]，其中H是高度，W是宽度，D是深度。3D特征图可以看作D个2D数据，每个2D数据的尺寸均为[HxW],称为特征图，3D特征图共有D个特征图。

升级做法为：每个特征图都分别与一个卷积核进行卷积运算，这样就得到D个特征图，这D个特征图先进行矩阵相加，得到一个特征图。再给该特征图的每个元素加一个相同的偏置，最终得到一个新的特征图。最终需要得到3D特征图，故上述过程执行多次，成为一个完整的卷积层操作。

从上面的过程中可看出，为了获得每个输出特征图，需要D个卷积核，我们把这D个卷积核称为一个卷积核组，它是一个3D矩阵。

例：卷积层操作示意图：

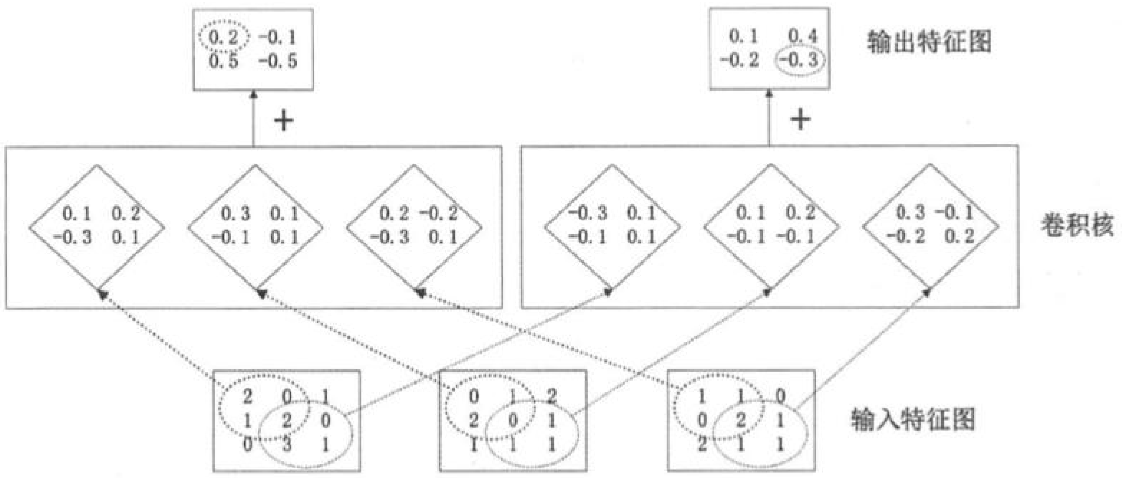


图 4.3　卷积层操作示意图

3D卷积层运算的代码实现：

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import numpy as np

def conv2D(input_2Ddata, kern):
 (h, w) = input_2Ddata.shape
 (kern_h, kern_w) = kern.shape
 padding_h = (kern_h - 1) // 2
 padding_w = (kern_w - 1) // 2
  padding = np.zeros(shape=(h + 2 * padding_h, w + 2 * padding_w))
  padding[padding_h:-padding_h, padding_w:-padding_w] = input_2Ddata
  output_2Ddata = np.zeros(shape=(h, w))
  for i in range(h):
  for j in range(w):
  window = padding[i:i + kern_h, j:j + kern_w]
  output_2Ddata[i, j] = np.sum(kern * window)
  return output_2Ddata


 def conv3D():
  h = 32 # 输入数据的高度
  w = 48 # 输入数据的宽度
  in_d = 12 # 输入数据的深度
  out_d = 24 # 输出数据的深度
  input_3Ddata = np.random.randn(h, w, in_d)
```

```
output_3Ddata = np.zeros(shape=(h, w, out_d))
(kern_h, kern_w) = (3, 3)
kerns = np.random.randn(out_d, kern_h, kern_w, in_d) # 4d卷积核
bias = np.random.randn(out_d) # 1D偏置

for m in range(out_d):
for k in range(in_d):
input_2Ddata = input_3Ddata[:, :, k] # 第k个输入2D数据
kern = kerns[m, :, :, k]
output_3Ddata[:, :, m] += conv2D(input_2Ddata, kern)
output_3Ddata[:, :, m] += bias[m]
return output_3Ddata


if __name__ == '__main__':
output_3Ddata = conv3D()
```

首先定义conv2D函数实现2D卷积运算，卷积核kern的尺寸为奇数， 一般是正方形。padding使用0填充来实现边缘处理。

然后定义输入和输出的3D特征图。卷积运算没有改变特征图的空间尺寸，但深度维度可能会增加。每个输出特征图需要累加输入3D特征图的每个2D特征图的卷积结果，最后加一个偏置。注意卷积核是四维矩阵，共有out_d个卷积核组，每个卷积核组的尺寸是[kern_h x kern_w x in_d]，每次和输入2D特征图进行卷积运算的二维卷积核的取值都不相同。四维卷积核和一维偏置，就是卷积层需要学习的参数。

卷积层运算需要的参数:

卷积核四维矩阵：out_d * kern_h * kern_w * in_d

偏置向量：out_d

其中参数数量与输入和输出3D特征图的深度成正比，与卷积核的面积成正比，而与特征图的空间尺寸无关。

下采样:

若需要缩小输入特征图的空间尺寸（一般缩小为原来的四分之一）这样需要进行下采样，下采样间隔称为步长S。S=1时输出特征图的空间尺寸和输入特征图一致。

包含步长S的程序:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
@File : the detail of c layer with s.py
@Time : 2019/7/12 11:20
@Author : Fan Yuheng
```

```
"""

import numpy as np


def conv2D(input_2Ddata, kern, in_size, out_size, kern_size=3, stride=1):
    (h1, w1) = in_size
    (h2, w2) = out_size
    output_2Ddata = np.zeros(shape=out_size)
    for i2,i1 in zip(range(h2), range(0, h1, stride)): # 输入数据进行步长
    for j2, j1, in zip(range(w2), range(0, w1, stride)):

    window = input_2Ddata[i1:i1+kern_size, j1:j1+kern_size]
    output_2Ddata[i2, j2] = np.sum(kern*window)
    return output_2Ddata

def conv3D():
    h1 = 32 # 输入数据高度
    w1 = 48 # 输入数据宽度
    d1 = 12 # 输入数据深度
    input_3Ddata = np.random.randn(h1, w1, d1)
    # 超参数
    S = 2 # 步长
    F = 3 # 卷积核尺寸
    d2 = 24 # 输出数据深度

    P = (F - 1) // 2 # 填充尺寸
    h2 = (h1 - F + 2 * P) // S + 1 # 输出数据高度
    w2 = (w1 - F + 2 * P) // S + 1 # 输出数据宽度

    padding = np.zeros(shape=(h1+2*P, w1+2*P, d1))
    padding[P:-P, P:-P, :] = input_3Ddata

    output_3Ddata = np.zeros(shape=(h2,w2,d2))

    kerns = np.random.randn(d2, F,F,d1)
    bias = np.random.randn(d2)

    for m in range(d2):
    for k in range(d1):
```

```
        input_2Ddata = padding[:, :,k]
        kern = kerns[m, :, :, k]
        output_3Ddata[:, :, m] += conv2D(input_2Ddata, kern, in_size=(h1, w1), out_size=(h2, w2), kern_size=F, stride=S)
    output_3Ddata[:, :, m] += bias[m]
    return output_3Ddata

if __name__ == '__main__':
    conv3D()
```

以上代码中，步长stride为2， conv2D函数需要知道输入和输出特征图的空间尺寸，该函数的输入特征图为0填
充后的特征图。

算法的超参数是卷积核尺寸F，步长S和输出特征图的深度d2。

F不需要太大，3和5最常见。因为卷积核参数数量和F平方成正比，F太大会导致参数数量急剧增加，计算量增加。

运算量，但输出特征图空间尺寸会急剧减小，导致丢失信息，因此S在实践中不会超过2.

深度d2和常规神经网络的隐含层宽度超参数类似，增大d2，学习容量增加，但运算量也增加。在实践中采用试错法。

卷积层操作的数学表示：

$$g(i,j) = \mathrm{bias} + \sum_{k=1}^{k=d1} \sum_{m=-F/2,n=-F/2}^{m=F/2,n=F/2} f_k(i'+m, j'+n) h_k(m,n)$$

其中g(i, j)是输出特征图的一个元素，需要以步长S遍历整个输入特征图：

$$i' = 1, 1+S, 1+2S, \cdots$$
$$j' = 1, 1+S, 1+2S, \cdots$$

卷积层参数总结：

（1）输入3D特征图的尺寸 H1 x W1 x D1

（2）三个超参数：

卷积核组数量：K

卷积核的空间尺寸： F

步长：S

（3）零填充数量 P = （F - 1）// 2

（4）输出3D特征图的尺寸为 H2 x W2 x D2

H2 = （H1-F+2P）//S + 1

W2 = （W1-F+2P）//S + 1

D2 = K

（5）卷积层一共有K x F x F x D1个权重和K个偏置，常见超参数设置为：F=3， S=1 or F=1， S=1

使用矩阵乘法实现卷积层运算：

卷积成的基本运算时卷积核组和输入特征图的局部区域做内积。

将输入3D特征图转化为矩阵X：

每个区域均拉伸为行向量，例如输入特征图是227*227*3，要与尺寸为11*11*3的卷积核以步长为4进行卷积，则首先取局部区域11*11*3数据块，将其拉伸成11*11*3 = 363 的行向量。然后扫描每个局部区域，均拉伸为行向量，因为步长为4，不使用0填充，输出的宽高为（227-11）//4+1=55，共有55*55=3025个行向量，因此输出矩阵的尺寸是3025*363。由于局部区域有重叠，输入特征图的元素在输出矩阵的不同行有重复。

卷积核组转化为矩阵W：

卷积核组拉伸成里列向量。例如尺寸为11*11*3的卷积核组拉伸为11*11*3=363的列向量，如果输出特征图的深度是96，则有96个列向量，生成一个矩阵W，尺寸为363*96

卷积层操作和矩阵乘法np.dot(X, W)是等价的，即得到每个卷积核组和每个局部区域数据的内积。

矩阵重新变为输出3D特征图的尺寸55*55*96，每行就是输出3D的一个深度向量，或每列就是输出3D特征图的一个特征图。

这个方法运算高效，但由于输入特征图中的元素在X中会复制很多次，会占用较多内存。

例：使用矩阵乘法实现卷积层运算示意图：

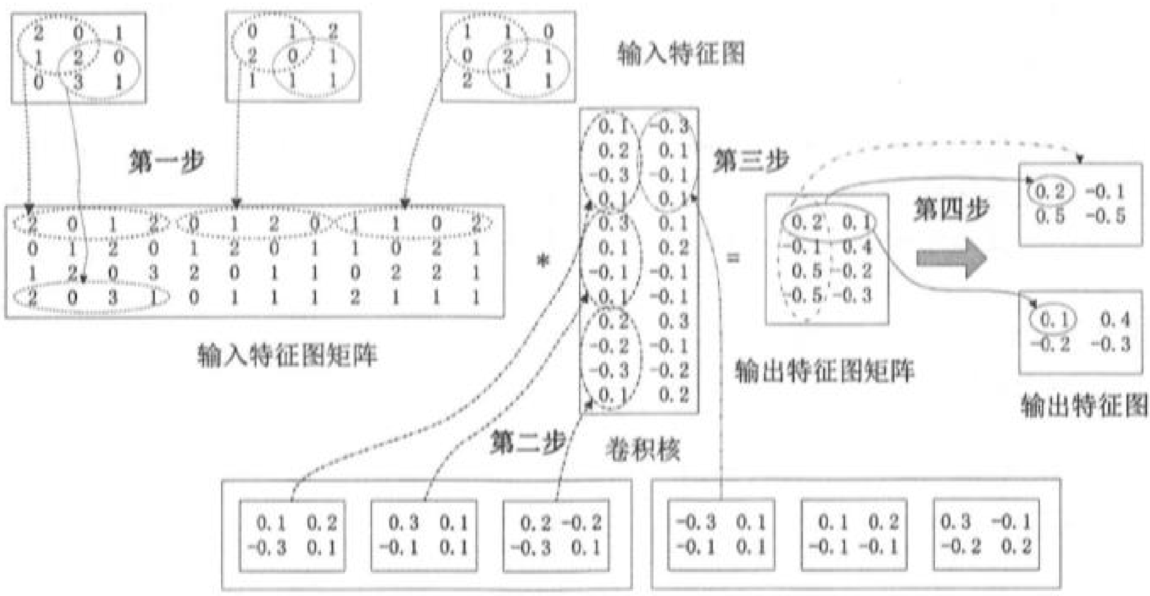


图 4.4 使用矩阵乘法实现卷积层运算的示意图