

# 卷积神经网络

目前，我们处理任何结构的信息，都只会选择 **将整个数据展开成一维向量**，这样就会忽略图像的空间结构信息。

为了改进上面的缺点，提出了 **卷积神经网络**，是一个很强大的，为了处理图像数据提出的神经网络。

卷积神经网络的基本元素：卷积层本身、填充、步幅、在相邻区域汇聚信息的汇聚层、在每一层多个通道的使用。

最后会介绍一个完整的 可以运行的leNet模型。（是很早就成功应用的卷积神经网络）

## 1.从全连接层次到卷积层

### 1.1平移不变性和局部性

这是两个CNN里面很重要的想法。

**平移不变性**：不管检测对象出现在图像中的哪个位置，神经网络的前面几层应该对相同的图像区域具有相似的反应，即为“平移不变性”。

这一部分也可以总结为 **不变性**，当我们要检测的物体出现的时候，只要检测到我们就认为它出现了，不去关注是否应该在图片的合理的位置，比如水里面、陆地上这种观点。

**局部性**：神经网络应该只探索 **输入图像里面的局部区域**，而不要在意图像中相隔较远的区域的关系。

## 2.图像卷积

学习一下卷积层的实际应用。

### 2.1互相关运算过程：

输入		核函数		输出																	
<table border="1"><tr><td>0</td><td>1</td><td>2</td></tr><tr><td>3</td><td>4</td><td>5</td></tr><tr><td>6</td><td>7</td><td>8</td></tr></table>	0	1	2	3	4	5	6	7	8	*	<table border="1"><tr><td>0</td><td>1</td></tr><tr><td>2</td><td>3</td></tr></table>	0	1	2	3	=	<table border="1"><tr><td>19</td><td>25</td></tr><tr><td>37</td><td>43</td></tr></table>	19	25	37	43
0	1	2																			
3	4	5																			
6	7	8																			
0	1																				
2	3																				
19	25																				
37	43																				

目前仅仅考虑二维的情况，暂且不讨论第三维度的情况。

**卷积核**上图的高度和宽度都是2，也就是 $\begin{smallmatrix} 0 & 1 \\ 2 & 3 \end{smallmatrix}$ 。

**二维互相关运算过程**：卷积窗口从输入的张量左上角开始，从左往右、从上往下进行滑动。

每一次滑动之后，**包含在该窗口中的部分张量与卷积核张量进行按元素相乘**，得到一个标量值，就是当前位置的输出。

上图的实际计算得到输出的过程为：

```

1  0 × 0 + 1 × 1 + 3 × 2 + 4 × 3 = 19,
2  1 × 0 + 2 × 1 + 4 × 2 + 5 × 3 = 25,
3  3 × 0 + 4 × 1 + 6 × 2 + 7 × 3 = 37,
4  4 × 0 + 5 × 1 + 7 × 2 + 8 × 3 = 43.

```

**尺寸、大小的说明：**输入大小为 $(n_h, n_w)$ ,卷积核大小为 $(k_b, k_w)$ 。  
得到的输出的大小为： $(n_h - k_h + 1, n_w - k_w + 1)$ 。

## 2.1.2手动互相关运算的代码实现：

```

1  import torch
2  from torch import nn
3
4  def corr2d(X, K):  #@save
5      """计算二维互相关运算"""
6      h, w = K.shape
7      Y = torch.zeros((X.shape[0] - h + 1, X.shape[1] - w + 1))
8      #这里直接指明Y的shape,防止越界访问元素
9      for i in range(Y.shape[0]):
10         for j in range(Y.shape[1]):
11             Y[i, j] = (X[i:i + h, j:j + w] * K).sum()
12     return Y
13
14     #声明张量X和卷积核张量K,调用函数进行计算:
15     X = torch.tensor([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]])
16     K = torch.tensor([[0.0, 1.0], [2.0, 3.0]])
17     print(corr2d(X, K))

```

结果为：  
19 25  
37 43

## 2.1.3卷积层的调库实现：

高度和宽度分别为  $h$  和  $w$  的卷积核可以被称为  $h \times w$  卷积或  $h \times w$  卷积核。我们也将带有  $h \times w$  卷积核的卷积层称为  $h \times w$  卷积层。

- **引入目标的边缘检测问题：**

通过找到像素变化的位置，来检测图像中不同颜色的边缘。初始图像为：

```

X = torch.ones((6, 8))
X[:, 2:6] = 0
X

```

```

tensor([[1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.]])

```

中间四列为黑色(0)，其余像素都是白色(1)。

理论情况下，我们需要一个卷积核为：

```
K = torch.tensor([[1.0, -1.0]])
```

**卷积核解释：**shape = (1,2)。如果两个输入元素相同，结果为0。如果输入是[1,0]结果是1.(从白色到黑色的边界)输入是[0,1]，结果为-1。（从黑色到白色的边界）。

上述问题的结果为：

```
tensor([[ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.]])
```

因为没有进行填充，步幅为1，所以最后的结果的宽度会减一。

借助库来实现卷积层：

还是解决上面的问题，定义变量X是输入，Y是输出。

声明一个卷积层，通过迭代让卷积核自己学习参数。

#### • 二维卷积层的声明：

此题需要：构造一个二维卷积层，它具有1个输出通道和形状为（1，2）的卷积核。

```
1 conv2d = nn.Conv2d(1, 1, kernel_size=(1, 2), bias=False)
```

参数解释：

1. nn.Conv2d(1, 1, kernel\_size=(1, 2), bias=False)：

- 1 (输入通道数)：卷积操作的输入特征图的通道数为 1。通常，对于单通道图像（如灰度图像），in\_channels 为 1。
- 1 (输出通道数)：卷积操作将生成一个输出通道（即一个特征图）。所以这里输出的特征图是单通道的。
- kernel\_size=(1, 2)：卷积核的大小为 1x2。这里表示卷积核的高度是 1，宽度是 2。也就是说，每次卷积操作会在输入特征图中选择一个 1 行 2 列的区域进行计算。卷积核的高度和宽度可以分别设置为单独的数值，因此 (1, 2) 表示垂直方向上只跨一行，水平方向上跨两列。
- bias=False：表示不使用偏置项。卷积层的输出通常是卷积核的加权和加上偏置，但这里显式设置了 bias=False，意味着卷积层将不使用偏置。

对于Conv2d官方所有参数解释：

```
Args:
    in_channels (int): Number of channels in the input image
    out_channels (int): Number of channels produced by the convolution
    kernel_size (int or tuple): Size of the convolving kernel
    stride (int or tuple, optional): Stride of the convolution. Default: 1
    padding (int, tuple or str, optional): Padding added to all four sides of
        the input. Default: 0
    padding_mode (string, optional): ``'zeros'``, ``'reflect'``,
        ``'replicate'`` or ``'circular'``. Default: ``'zeros'``
    dilation (int or tuple, optional): Spacing between kernel elements. Default: 1
    groups (int, optional): Number of blocked connections from input
        channels to output channels. Default: 1
    bias (bool, optional): If ``True``, adds a learnable bias to the
        output. Default: ``True``
    """
    .format(**reproducibility_notes, **convolution_notes) + r"""
```

文件位置：

"D:\ProgramData\anaconda3\envs\d21\Lib\site-packages\torch\nn\modules\conv.py"

#### • 完整代码和解释：

```
1 #构造输入和标准输出
2 X = torch.ones((6, 8))
3 X[:, 2:6] = 0
4
5 Y = torch.zeros((6, 7))
6 Y[:, 1] = 1
7 Y[:, -2] = -1
8
```

```

9
10 # 构造一个二维卷积层，它具有1个输出通道和形状为（1，2）的卷积核
11 conv2d = nn.Conv2d(1, 1, kernel_size=(1, 2), bias=False)
12 # 这个二维卷积层使用四维输入和输出格式（批量大小、通道、高度、宽度），
13 # 其中批量大小和通道数都为1
14 X = X.reshape((1, 1, 6, 8))
15 print(X)
16 Y = Y.reshape((1, 1, 6, 7))
17 print(Y)
18 lr = 3e-2 # 学习率
19
20 for i in range(10):
21     Y_hat = conv2d(X)
22     # print(Y_hat)
23     l = (Y_hat - Y) ** 2
24     conv2d.zero_grad()
25     l.sum().backward()
26     # 迭代卷积核
27     conv2d.weight.data[:] -= lr * conv2d.weight.grad
28     if (i + 1) % 2 == 0:
29         print(f'epoch {i+1}, loss {l.sum():.3f}')

```

- **输入和输出的格式要求：**

输入和输出的格式都要求：**四维输入和输出格式（批量大小、通道、高度、宽度）**。  
所以，我们首先要将X,Y进行reshape。

- 对于一个二维张量X。

如果 $X.shape = (H, W)$ 。

通过执行：`X = X.reshape((1, 1) + X.shape)` 就会对张量 X 进行形状调整。

- `(1, 1)`：这个元组表示要在形状的前两维前加上两个维度，值都为1。这通常是为了将 X 转换为一个有 4 个维度的张量，形状的前两维表示通道数和批次大小。

`(1, 1) + X.shape`：通过将 `(1, 1)` 和 `X.shape` 拼接起来，得到一个新的元组。举例来说，如果 `X.shape` 是 `(H, W)`，则 `X.reshape((1, 1) + X.shape)` 就是将 X 的形状转换为 `(1, 1, H, W)`。如果 `X.shape` 本来就是 `(C, H, W)`，则新的形状会变成 `(1, 1, C, H, W)`。

`X.reshape(...)`：这部分是将 X 重新调整为新的形状，保持数据的内容不变，只是调整了它的维度。

- 整个迭代的过程，以及参数更新的操作，在上述代码。

## 2.2互相关和卷积

互相关和卷积计算并不是一回事，两者有细微的差别。

**more formally:** 只需水平和垂直翻转二维卷积核张量，然后对输入张量执行互相关运算。

**上述过程就是卷积计算。**

这一部分的具体过程可以看 [链接1](#)，[链接2](#)。

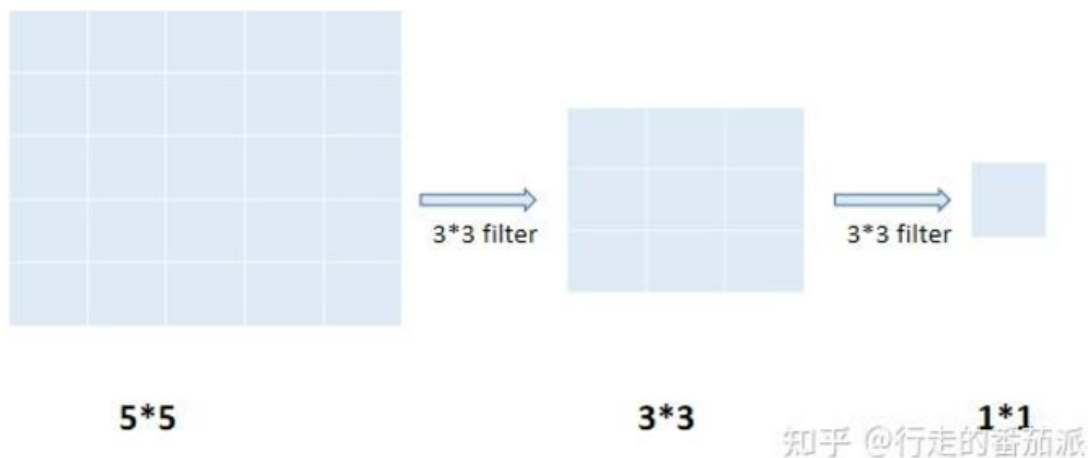
值得注意的是，由于卷积核是从数据中学习到的，因此无论这些层执行严格的卷积运算还是互相关运算，卷积层的输出都不会受到影响。为了说明这一点，假设卷积层执行互相关运算并学习图6.2.1中的卷积核，该卷积核在这里由矩阵 $\mathbf{K}$ 表示。假设其他条件不变，当这个层执行严格的卷积时，学习的卷积核 $\mathbf{K}'$ 在水平和垂直翻转之后将与 $\mathbf{K}$ 相同。也就是说，当卷积层对图6.2.1中的输入和 $\mathbf{K}'$ 执行严格卷积运算时，将得到与互相关运算图6.2.1中相同的输出。

为了与深度学习文献中的标准术语保持一致，我们将继续把“互相关运算”称为卷积运算，尽管严格地说，它们略有不同。此外，对于卷积核张量上的权重，我们称其为元素。

## 2.3感受野

**定义：**对于CNN每一层输出的 **特征图** 上面的每一个元素，它的感受野就是**相对于原图，这个点能够看到的输入图像的区域**。

**举例说明：**



输入的图像shape为 $5 \times 5$ ，经过一次卷积之后，得到 $3 \times 3$ 的特征图，此时每一个特征图里面的元素对应的 **感受野的尺寸为 $3 \times 3$** 。

再经过一次卷积之后，最后得到一个元素，此时，这个元素的感受野的尺寸是  **$5 \times 5$** 。也就是整个输入的图片。

```
(d21) D:\Users\gxy\OneDrive\directory\graduate_study\DEEP_learning\Code_of_book\pytorch\chapter_convolutional-neural-networks>python
Python 3.9.20 (main, Oct 3 2024, 07:38:01) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> |
```

## 3.填充和步幅

卷积的输出形状取决于输入形状和卷积核的形状。

公式为：

**尺寸、大小的说明：**输入大小为 $(n_h, n_w)$ ,卷积核大小为 $(k_h, k_w)$ 。  
得到的输出的大小为： $(n_h - k_h + 1, n_w - k_w + 1)$ 。

### 3.1填充(包括填充的调库实现)

根据上面公式，当我们使用多层卷积时，会丢失边缘像素。

虽然当我们使用一个很小的卷积核时，并不会减少很多的像素，但是如果我们使用多层次的，每一层都会减少。就会导致 **累积丢失很多像素**。

**为了解决这个问题**，可以使用**填充**：在输入图像的边界填充元素。（填充的元素通常都是0）。

- 填充举例：

- 没有填充之前:

输入		核函数		输出																	
<table border="1" style="border-collapse: collapse;"> <tr><td>0</td><td>1</td><td>2</td></tr> <tr><td>3</td><td>4</td><td>5</td></tr> <tr><td>6</td><td>7</td><td>8</td></tr> </table>	0	1	2	3	4	5	6	7	8	*	<table border="1" style="border-collapse: collapse;"> <tr><td>0</td><td>1</td></tr> <tr><td>2</td><td>3</td></tr> </table>	0	1	2	3	=	<table border="1" style="border-collapse: collapse;"> <tr><td>19</td><td>25</td></tr> <tr><td>37</td><td>43</td></tr> </table>	19	25	37	43
0	1	2																			
3	4	5																			
6	7	8																			
0	1																				
2	3																				
19	25																				
37	43																				

- 填充之后: 将  $3 \times 3$  输入填充到  $5 \times 5$ , 那么它的输出就增加为  $4 \times 4$ 。

输入		核函数		输出																																													
<table border="1" style="border-collapse: collapse;"> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>1</td><td>2</td><td>0</td></tr> <tr><td>0</td><td>3</td><td>4</td><td>5</td><td>0</td></tr> <tr><td>0</td><td>6</td><td>7</td><td>8</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> </table>	0	0	0	0	0	0	0	1	2	0	0	3	4	5	0	0	6	7	8	0	0	0	0	0	0	*	<table border="1" style="border-collapse: collapse;"> <tr><td>0</td><td>1</td></tr> <tr><td>2</td><td>3</td></tr> </table>	0	1	2	3	=	<table border="1" style="border-collapse: collapse;"> <tr><td>0</td><td>3</td><td>8</td><td>4</td></tr> <tr><td>9</td><td>19</td><td>25</td><td>10</td></tr> <tr><td>21</td><td>37</td><td>43</td><td>16</td></tr> <tr><td>6</td><td>7</td><td>8</td><td>0</td></tr> </table>	0	3	8	4	9	19	25	10	21	37	43	16	6	7	8	0
0	0	0	0	0																																													
0	0	1	2	0																																													
0	3	4	5	0																																													
0	6	7	8	0																																													
0	0	0	0	0																																													
0	1																																																
2	3																																																
0	3	8	4																																														
9	19	25	10																																														
21	37	43	16																																														
6	7	8	0																																														

图6.3.1: 带填充的二维互相关。

- 如果我们一共添加  $p_h$  行,  $p_w$  列。一般添加的时候一半在一边、另外一半在另一边。此时最后的尺寸为:  
 $n_h - k_h - p_h + 1, n_w - k_w + p_w + 1$ 。  
 在许多情况下, 我们需要设置  $p_h = k_h - 1$  和  $p_w = k_w - 1$ , 使输入和输出具有相同的高度和宽度。这样可以在构建网络时更容易地预测每个图层的输出形状。假设  $k_h$  是奇数, 我们将在高度的两侧填充  $p_h/2$  行。如果  $k_h$  是偶数, 则一种可能性是在输入顶部填充  $\lceil p_h/2 \rceil$  行, 在底部填充  $\lfloor p_h/2 \rfloor$  行。同理, 我们填充宽度的两侧。  
 卷积神经网络中卷积核的高度和宽度通常为奇数, 例如1、3、5或7。选择奇数的好处是, 保持空间维度的同时, 我们可以在顶部和底部填充相同数量的行, 在左侧和右侧填充相同数量的列。

#### • 代码实现:

创建一个高度和宽度为3的二维卷积层。

给定高度和宽度为8的输入, 在输入图像的所有侧边都填充一个像素, 最后输出的高度和宽度还是8。

- 调库使用卷积层:

```
1 conv2d = nn.Conv2d(1, 1, kernel_size=3, padding=1)
```

参数分别表示: 输入通道数为1, 输出通道数为1。卷积核的shape是  $3 \times 3$ 。  
`padding = 1` 表示每一个侧边都填充一个像素。

- 创建输入变量: `X = torch.rand(size=(8, 8))`
- 进行卷积操作:

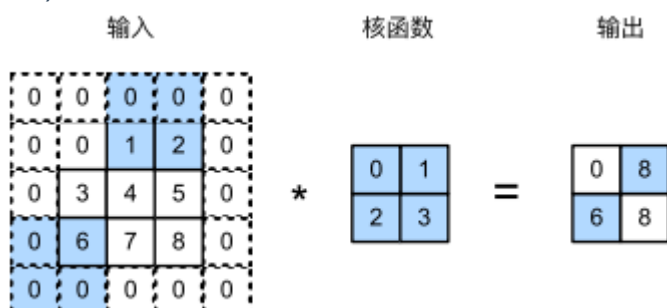
```
1 #输入图像, 调整为四维的输入格式。
2 X = X.reshape((1, 1) + X.shape)
3 #卷积操作得到Y
4 Y = conv2d(X)
5 #得到的Y此时也是四维格式, 我们只保留最后两个维度的信息
6 Y.reshape(Y.shape[2:])
```

**需要注意的是:** `nn.conv2d` 函数, 标准输入输出格式都是四个维度。  
 (input\_channel,output\_channel,h,w) 。

## 3.2步幅(包括步幅的调库实现)

之前所有的例子里面，我们的卷积窗口从输入张量的左上角开始，向下、向右滑动。  
每一次滑动平移的位置都是1，**每一次滑动元素的数量是步幅**，现在我们允许使用较大的步幅。

比如当我们将上述问题修改为 **垂直步幅为3，水平步幅为2**：(如果想要复现这个结果，需要将bias设置为False)。



一旦不能满足下一次滑动之后卷积窗口都有元素，就会停止滑动。

此时最后输出的形状为：

$(n_h - k_h + p_h + s_h)/s_h, (n_w - k_w + p_w + s_w)/s_w$ 。注意都是向下取整。

如果我们设置了  $p_h = k_h - 1$ 、 $p_w = k_w - 1$ ，则输出形状将简化为  $(n_h/s_h, n_w/s_w)$ 。

- 上述图示步幅的代码实现：

原先：

```
1 conv2d = nn.Conv2d(1, 1, kernel_size=3, padding=1)
```

现在：

```
1 conv2d = nn.Conv2d(1, 1, kernel_size=3, padding=1, stride = (3,2), bias = False)
```

步幅的设置，第一个维度是垂直方向，第二个是水平方向。

- 指定核函数的值：

```
1 kernel = torch.tensor([[[[0, 1], [2, 3]]]], dtype=torch.float) # 1 个输入通道, 1 个输出通道,
    2x2 核
2 with torch.no_grad():
3     conv2d.weight = nn.Parameter(kernel)
```

- 完整代码：

```
1 import torch
2 import torch.nn as nn
3
4 # 定义输入张量
5 A = torch.tensor([[0, 1, 2],
6                   [3, 4, 5],
7                   [6, 7, 8]], dtype=torch.float)
8
9 # 定义卷积核 (核函数)
10 kernel = torch.tensor([[[[0, 1],
11                           [2, 3]]]], dtype=torch.float) # 1 个输入通道, 1 个输出通道, 2x2 核
12
13 # 创建卷积层
14 conv2d = nn.Conv2d(in_channels=1, out_channels=1, kernel_size=2, padding=1,
15                     bias=False, stride = (3,2))
```



```

16 # 手动设置卷积核权重
17 with torch.no_grad():
18     conv2d.weight = nn.Parameter(kernel)
19
20 # 将输入张量 A 调整为符合 Conv2d 的输入形状 (batch_size, channels, height, width)
21 A = A.reshape((1, 1) + A.shape)
22
23 # 执行卷积操作
24 B = conv2d(A)
25
26 # 打印结果
27 print("输入: ")
28 print(A)
29 print("卷积核: ")
30 print(kernel)
31 print("输出: ")
32 print(B)

```

## 4.多输入多输出通道

之前的研究中，我们认为输入的图像都是二维的张量，添加上通道之后，就变为了三维的张量。此时输入的图像的形狀为  $3 \times h \times w$ 。这一节我们研究多输入、多输出的卷积核。

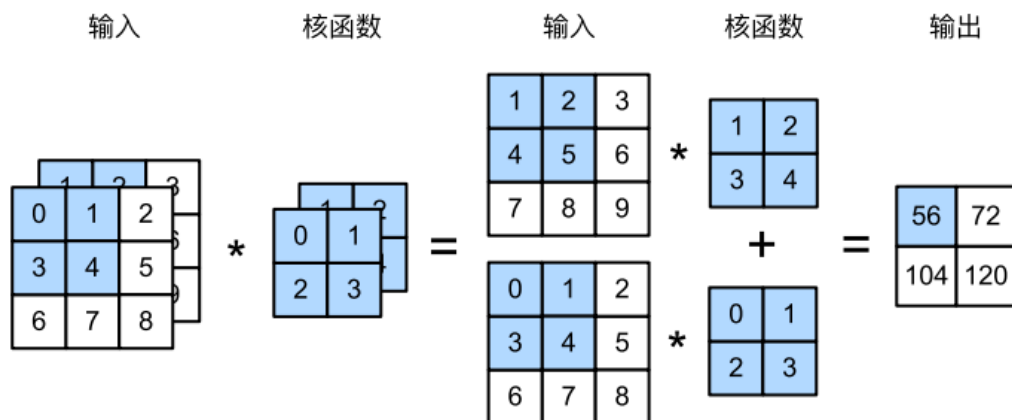
### 4.1多输入、单输出的卷积核：

当输入包含多个通道时，需要构造一个与输入数据具有相同输入通道数的卷积核，以便与输入数据进行互相关运算。

此时：输入通道数目 == 卷积核输入通道数目。

假设输入的通道数目为  $c_i$ ，当  $c_i > 1$  时，将这些张量连结在一起可以得到形状为  $c_i \times k_h \times k_w$  的卷积核。

- 计算过程举例：



演示了一个具有两个输入通道的二维互相关运算的示例。阴影部分是第一个输出元素以及用于计算这个输出的输入和核张量元素：

```

1 (1 * 1 + 2 * 2 + 4 * 3 + 5 * 4) + (0 * 0 + 1 * 1 + 3 * 2 + 4 * 3) =
  56。

```



- 一句话总结多输入通道的计算过程：  
因为输入的通道数目 = 卷积核通道数目。  
所以可以对每个通道输入的二维张量和卷积核的二维张量进行互相关运算，之后再对 $c_i$ 个结果进行求和。得到一个二维张量，就是一个输出。

## 4.2手动实现多输入、单输出的卷积核计算：

```

1  import torch
2  # from d2l import torch as d2l
3
4  #声明输入数据：
5  X = torch.tensor([[[[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]],
6                    [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]]]])
7  #声明卷积核数据：
8  K = torch.tensor([[[[0.0, 1.0], [2.0, 3.0]], [[1.0, 2.0], [3.0, 4.0]]]])
9
10
11 def corr2d(X, K):
12     #计算X 和 K 的卷积结果
13     #此时 X 和 K 都是二维张量。
14     # print('X.shape = ', X.shape)
15     # print('K.shape = ', K.shape)
16     h, w = K.shape
17     Y = torch.zeros((X.shape[0] - h + 1, X.shape[1] - w + 1))
18     for i in range(Y.shape[0]):
19         for j in range(Y.shape[1]):
20             Y[i, j] = (X[i:i + h, j:j + w] * K).sum()
21     return Y
22
23 def corr2d_multi_in(X, K):
24     # 先遍历“X”和“K”的每一个维度，再把它们加在一起
25     return sum(corr2d(x, k) for x, k in zip(X, K))
26
27 corr2d_multi_in(X, K)

```

手动代码实现可以帮助理解。

不过在实际中往往会直接调用nn.conv2d来声明一个卷积层。

- 结果：

```

1  tensor([[ 56.,  72.],
2          [104., 120.]])

```

## 4.3多输入、多输出(包括多输入、多输出的手动实现)：

- 多输入、多输出的必要性：  
在最流行的神经网络架构中，随着神经网络层数的加深，我们常会增加输出通道的维数，通过减少空间分辨率以获得更大的通道深度。
- 多输入、多输出的模式下，卷积核的结构：  
用 $c_i, c_o$ 分别表示输入和输出通道的数目，为了获得多个通道的输出，我们可以为每个输出通道创建一个形状为 $c_i \times k_h \times k_w$ 卷积核张量，这样卷积核的形状是 $c_o \times c_i \times k_h \times k_w$ 。  
不过，对于nn.conv2d函数里面的参数的顺序，第一个是input\_channel\_number，第二个是output\_channel\_number。

- 手动实现多输入、多输出网络结构：

```
1 import torch
2 #声明输入数据：
3 X = torch.tensor([[[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]],
4                   [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]]])
5 #声明卷积核数据：
6 K = torch.tensor([[[0.0, 1.0], [2.0, 3.0]], [[1.0, 2.0], [3.0, 4.0]]])
7
8 #创建具有三个输出通道的卷积核：
9 K = torch.stack((K, K + 1, K + 2), 0)
10
11 #遍历每一个输出通道的卷积核，每一个卷积核都得到一个输出，最后将三个输出通道的输出堆叠在一起。
12 Y = torch.stack([corr2d_multi_in(X, k) for k in K], 0)
13 print(Y)
```

此处调用的`corr2d_multi_in()`函数，就是上文在 多输入、单输出 中实现的函数。

- 结果：

```
1 tensor([[ 56.,  72.],
2         [104., 120.]])
3 tensor([[[ 56.,  72.],
4           [104., 120.]],
5
6         [[ 76., 100.],
7           [148., 172.]],
8
9         [[ 96., 128.],
10          [192., 224.]])
```

## 5.汇聚层

### 5.1汇聚层原理

- 汇聚层原理：  
其作用是对卷积层输出的特征图进行下采样（缩小特征图大小），从而减少计算量和参数数量，同时提取更为重要的特征。
- 汇聚层两种基本实现：  
通常有两种方式：最大池化（Max Pooling）和平均池化（Average Pooling）。  
最大池化层的操作是在滑动窗口内寻找最大值作为该区域的输出.平均池化层的操作是在滑动窗口内计算均值作为该区域的输出。
- 汇聚层的主要作用有以下几个方面：
  - 减少计算量和参数数量，使得模型更容易训练；
  - 降低特征图的分辨率，减少过拟合；
  - 提取更为重要的特征，因为汇聚层只选择最大值或平均值，这些值往往包含更多的信息
- 在卷积网络体系结构中，在连续卷积层之间周期性地插入汇集层是常见的。它的功能是逐步减少表示的空间大小，以减少网络中的参数和计算量，从而也控制过拟合。

## 5.2最大汇聚层、平均汇聚层计算过程

- 与卷积层类似，汇聚层运算符由一个固定形状的窗口组成，该窗口根据其步幅大小在输入的所有区域上滑动，为固定形状窗口（有时称为汇聚窗口）遍历的每个位置计算一个输出。  
简单来说，就是计算汇聚窗口中所有元素的最大值或平均值。
- 举例：(步幅为1)



- 互相关运算符一样，汇聚窗口从输入张量的左上角开始，从左往右、从上往下的在输入张量内滑动。在汇聚窗口到达的每个位置，它计算该窗口中输入子张量的最大值或平均值。

## 5.3手动实现汇聚层

```
1 import torch
2
3 def pool2d(X, pool_size, mode = 'max'):
4     p_h , p_w = pool_size
5     #直接指定最后的输出的shape, 方便运算的实现
6     Y = torch.zeros( (X.shape[0] - p_h + 1 , X.shape[1] - p_w + 1) )
7     print(Y.shape)
8
9     for i in range(Y.shape[0]):
10        for j in range(Y.shape[1]):
11            if mode == 'max':
12                Y[i, j] = X[i:i+p_h , j:j+p_w].max()
13            elif mode == 'avg':
14                Y[i, j] = X[i:i+p_h , j:j+p_w].mean()
15            print(Y[i, j])
16        return Y
17
18 #创建输入数据:
19 X = torch.tensor([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]])
20 print(X.shape)
21
22 #进行汇聚操作
23 print(pool2d(X, (2, 2), 'max'))
```

- 这里手动实现的代码和第一次手动实现 [互相关运算](#) 的代码基本一致。  
同时更改最后的max 为 avg，就是 [平均汇聚层](#) 的操作。

## 5.4调库实现最大汇聚层

- 创建数据：

```
1 X = torch.arange(16, dtype=torch.float32).reshape((1, 1, 4, 4))
2 X
```

结果：

```

1  tensor([[[[ 0.,  1.,  2.,  3.],
2      [ 4.,  5.,  6.,  7.],
3      [ 8.,  9., 10., 11.],
4      [12., 13., 14., 15.]]]])

```

- 调库进行计算：

**默认情况下，步幅和汇聚窗口的大小一致。**

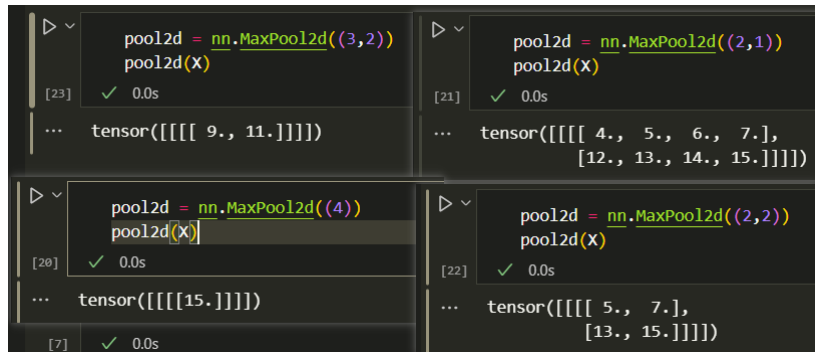
```

1  pool2d = nn.MaxPool2d((3))
2  pool2d(X)

```

此时结果为10.

一些运算结果：



## 5.5 nn.MaxPool2d()函数参数说明

位置：`d:\ProgramData\anaconda3\envs\d2l\Lib\site-packages\torch\nn\modules\pooling.py`

- 官方解释：

```

Args:
    kernel_size: the size of the window to take a max over
    stride: the stride of the window. Default value is :attr:`kernel_size`
    padding: implicit zero padding to be added on both sides
    dilation: a parameter that controls the stride of elements in the window
    return_indices: if ``True``, will return the max indices along with the outputs.
    Useful for :class:`torch.nn.MaxUnpool2d` later
    ceil_mode: when True, will use `ceil` instead of `floor` to compute the output shape

```

**默认情况下，stride = kernel。**

## 5.6 汇聚层使用填充和步幅

- 没有指定填充和步幅参数时，默认填充 = 0， stride = kernel\_size。
- 填充和步幅可以手动设定：

```

1  pool2d = nn.MaxPool2d(3, padding=1, stride=2)
2  pool2d(X)

```

结果：

```

1  tensor([[[[ 5.,  7.],
2      [13., 15.]]]])

```

- 也可以以设定一个任意大小的矩形汇聚窗口，并分别设定填充和步幅的高度和宽度。

```

1 pool2d = nn.MaxPool2d((2, 3), stride=(2, 3), padding=(0, 1))
2 pool2d(X)

```

结果：

```

1 tensor([[[[ 5.,  7.],
2          [13., 15.]]]])

```

## 5.7 汇聚层的多通道：

- 之前的卷积层，每一个通道都需要将所有输入通道的计算结果进行叠加。但是从最大值、平均值的角度，叠加完全没有必要。所有对于汇聚层，不会进行叠加。**输入通道数目 = 输出通道数目**。不同通道之间的计算是独立的。
- 在处理多通道输入数据时，汇聚层在每个输入通道上单独运算，而不是像卷积层一样在通道上对输入进行汇总。这意味着汇聚层的输出通道数与输入通道数相同。
- 在通道维度上连结张量X和X + 1，以构建具有2个通道的输入。

```

1 X = torch.cat((X, X + 1), 1)

```

然后进行最大汇聚：

```

1 pool2d = nn.MaxPool2d(3, padding=1, stride=2)
2 pool2d(X)

```

此时X为：

```

tensor([[[[ 0.,  1.,  2.,  3.],
           [ 4.,  5.,  6.,  7.],
           [ 8.,  9., 10., 11.],
           [12., 13., 14., 15.]],

          [[ 1.,  2.,  3.,  4.],
           [ 5.,  6.,  7.,  8.],
           [ 9., 10., 11., 12.],
           [13., 14., 15., 16.]]]])

```

结果为：

```

tensor([[[[ 5.,  7.],
           [13., 15.]],

          [[ 6.,  8.],
           [14., 16.]]]])

```

- 可以看到，前后的通道数目是一致的。**输入通道数目 = 输出通道数目**。