卷积网络

朱明超

deityrayleigh@gmail.com

卷积网络是指那些至少在网络的一层中使用**卷积运算**来**替代一般的矩阵乘法运算**的神经网络。

1 卷积运算

卷积操作是指对**不同的数据窗口数据** (图像) 用**一组固定的权重** (滤波矩阵，可以看做一个恒定的滤波器 `Filter`) **逐个元素相乘再求和** (做内积)。

也可以用如下方式理解，如果想要低噪声估计，一种可行的方法是**对得到的测量结果进行平均**。可以认为**时间上越近**的测量**结果越相关**，所以采用一种加权平均的方法，对于**最近的测量结果赋予更高的权重**。

采用一个加权函数 $w(a)$ 来实现，其中 a 表示测量结果距当前时刻的时间间隔：$s(t)=\int x(a)w(t-a)da$。这就是卷积操作，可以用星号表示：$s(t)=(x*w)(t)$。其中 x 是输入 (`Input`)；w 是核 (`Kernel` 或 `Filter`)；输出的 s 是特征映射或特征图 (`Feature Map`) 或称输出 (`Output`)。

二维卷积运算

上面我们介绍了卷积运算的思想，但我们通常在卷积层中会使用更加直观的互相关 (`Cross-correlation`) 运算。例如，在二维卷积层中，一个二维输入数组 I 和一个二维核数组 K 通过互相关运算输出一个二维数组 O。如图所示，输入是一个高和宽均为 3 的二维数组，核数组 (也称卷积核或过滤器) 的高和宽分别为 2。因此，我们可以说输入数组形状为 $(3,3)$，卷积核窗口 (又称卷积窗口) 的形状为 $(2,2)$。

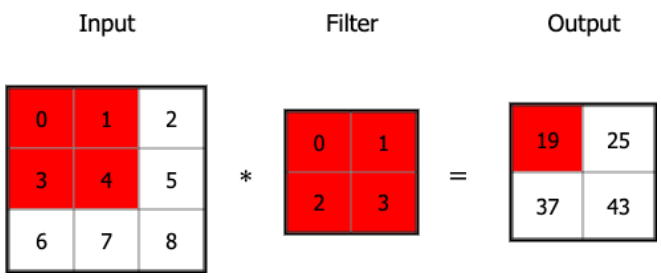

图 1. 二维卷积运算。输入的红色区域与核的红色区域通过运算得到输出的左上角红色区域。

红色区域显示了第一个输出元素的计算过程：$0\times0+1\times1+3\times2+4\times3=19$。

其他输出元素的计算过程同样可得：

$$\begin{aligned}1\times0+2\times1+4\times2+5\times3&=25,\\3\times0+4\times1+6\times2+7\times3&=37,\\4\times0+5\times1+7\times2+8\times3&=43.\end{aligned}\tag{1}$$

卷积背后的直觉

我们将卷积操作方式展开成前馈网络的形式，如图 2 所示，左图为原本的前馈网络，右图为卷积操作。在第六章介绍的深度前馈网络中，权重矩阵中的**每一个元素只会使用一次**，不会在不同的输入位置上共享参数。而在卷积操作中，卷积核的**每一个元素都作用在输入的每一位置**上。这个设计保证了我们只需要学习一个参数集合，而不用对每一位置去学习一个单独的参数。

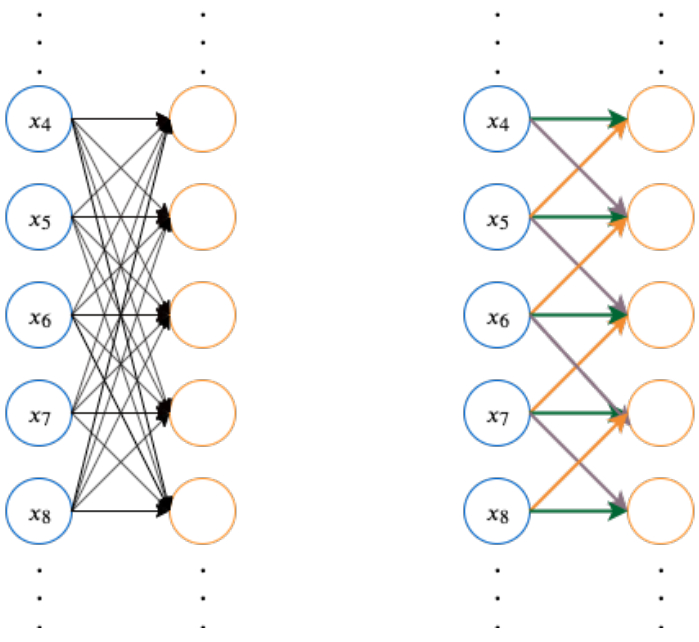

图 2. 卷积操作展开为深度前馈网络形式 (见第六章)，左图为前馈网络，右图为卷积网络展开。

这便是卷积的直觉，稀疏交互 (Sparse Interactions) 与参数共享 (Parameter Sharing)。前馈网络中每一个输出单元与每一个输入单元都产生交互，如果有 m 个输入和 n 个输出，那么矩阵乘法需要 $m \times n$ 个参数并且相应算法的时间复杂度为 $O(m \times n)$；卷积操作中如果我们限制每一个输出拥有的连接数为 k，那么稀疏的连接方法只需要 $k \times n$ 个参数以及 $O(k \times n)$ 的运行时间。

2 池化

在得到卷积特征后，下一步就是用来做分类。理论上，可以用提取到的所有特征训练分类器，但用所有特征的计算量开销会很大，而且也会使分类器倾向于过拟合。

为了解决这个问题，考虑到要描述一个大图像，一个自然的方法是在不同位置处对特征进行汇总统计。例如，可以计算在图像中某一区域的一个特定特征的平均值 (或最大值)，这样概括统计出来的数据，其规模 (相比使用提取到的所有特征) 就低得多，同时也可以改进分类结果 (使模型不易过拟合)。这样的聚集操作称为 “池化”。池化可以保留显著特征，降低特征规模。

池化运算

池化函数是使用某一位置的**相邻输出的总体统计特征**来代替网络在该位置的输出。

- **最大池化函数** (Max Pooling) 给出相邻矩形区域内的最大值；
- **平均池化** (Average Pooling) 计算相邻矩形区域内的平均值；
- 其他常用的池化函数包括 L^2 范数以及基于距中心像素距离的加权平均函数。

如图 3 所示，我们展示最大池化的运算过程：

$$\begin{aligned} \max(0,1,3,4) &= 4, \\ \max(2,0,5,0) &= 5, \\ \max(6,7,0,0) &= 7, \\ \max(8,0,0,0) &= 8. \end{aligned} \tag{2}$$

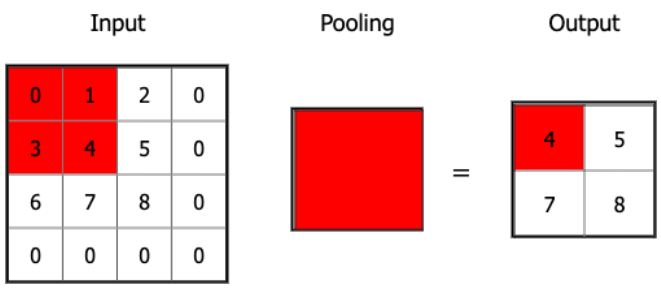

图 3. 最大池化运算。输入的红色区域经过池化得到输出的左上角红色区域。

不管采用什么样的池化函数，当输入作出少量平移时，池化能够帮助输入的表示近似不变。平移的不变性是指当我们对输入进行少量平移时，经过池化函数后的大多数输出并不会发生改变。**局部平移不变性**是一个很有用的性质，**尤其是当我们关心某个特征是否出现而不关心它出现的具体位置时**，后文我们会对这一性质进一步描述。

卷积与池化作为一种无限强的先验：

先验的强或者弱取决于先验中概率密度的集中程度：弱先验具有较高的熵值，例如方差很大的高斯分布，这样的先验允许数据对于参数的改变具有或多或少的自由性；强先验具有较低的熵值，例如方差很小的高斯分布，这样的先验在决定参数最终取值时起着更加积极的作用；无限强的先验需要对一些参数的概率置零并且完全禁止对这些参数赋值，无论数据对于这些参数的值给出了多大的支持。因此，我们可以把卷积的使用当作是对网络中一层的参数引入了一个无限强的先验概率分布，要求该层学到的函数只能包含局部连接关系并且对平移具有等变性。

3 深度学习框架下的卷积：

3.1 多个并行卷积

通常指由多个并行卷积组成的运算，可以在每个位置提取多种类型的特征。

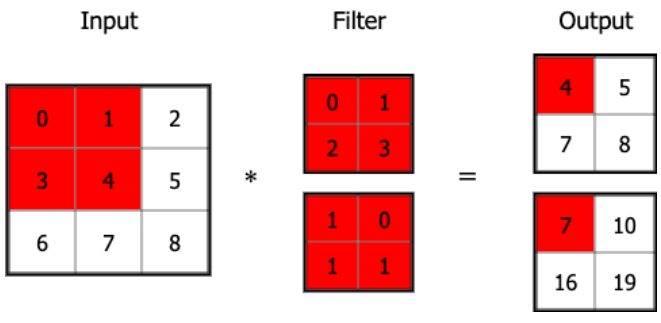

图 4. 多个并行卷积。输入的红色区域分别与两个核经过计算分别得到输出的左上角红色区域。

如图 4 所示，我们可以用两组卷积核去提取不同的特征。

3.2 输入值与核

输入通常也不仅仅是实值的网格，而是由一系列向量的网格，如一幅彩色图像在每一个像素点都会有红绿蓝三种颜色的亮度。当处理图像时，通常把卷积的输入输出都看作是 3 维的张量，其中一个索引用于标明不同的通道 (例如 RGB)，另外两个索引标明在每个通道上的空间坐标。我们可以将输入图像数组写作 V，它的每一个元素是 $\mathsf{V}_{i,j,k}$，表示处在通道 k 中第 i 行第 j 列的值。如下图所示，V 有 3 个通道，每个通道的形状均为 $(3,3)$。此时，我们还要定义卷积核数组 K，它的每一个元素是 $\mathsf{K}_{i,j,k,l}$，表示处在第 l 组卷积核通道 k 中第 i 行第 j 列的值。如图 5 所示，我们假设卷积核窗口的形状为 $(2,2)$，因为输入通道为 3，于是我们要对每个通道都定义一个卷积核，同时我们构造两组卷积核实现并行卷积，所以 K 的形状为 $(2,2,3,2)$。

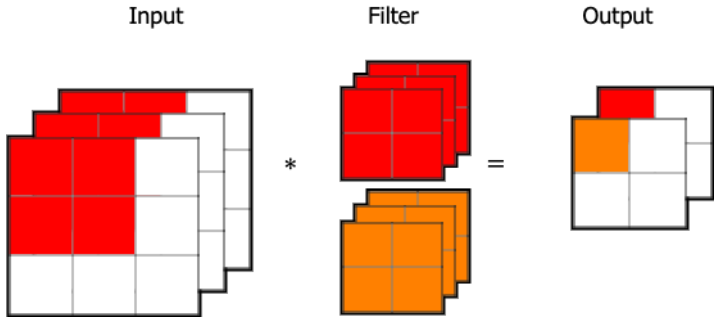

图 5. 输入数组的通道数为 3 (如图像数据)。用两组卷积核 (红色、橙色) 学习特征，而每组卷积核与输入运算再得到输出。

我们再看一下是如何获得输出 Z 值的，我们用 1 组卷积核，并且以输入为 2 通道为例，如下图 6，$\mathsf{Z}_{i,j,l} = \sum_{m,n,k} \mathsf{V}_{i+m,j+n,k}\mathsf{K}_{m,n,k,l}$。

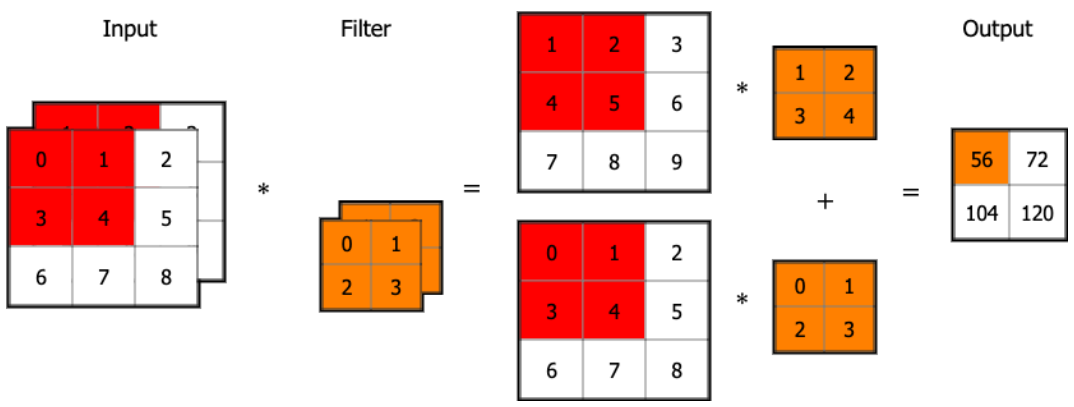

图 6. 输入数组的通道数为 2。展示用一组卷积核学习特征，卷积核的每个通道与输入的对应通道进行运算。再将各个通道的结果相加得到输出。

3.3 填充 (Padding)

假设输入图片的大小为 (m,n)，而卷积核的大小为 (f,f)，则卷积后的输出图片大小为 $(m-f+1, n-f+1)$，由此带来两个问题：

- 每次卷积运算后，输出图片的尺寸缩小。
- 原始图片的角落、边缘区像素点在输出中采用较少，输出图片丢失很多边缘位置的信息。

因此可以在进行卷积操作前，对原始图片在边界上进行填充 (Padding)，以增加矩阵的大小，通常将 0 作为填充值。

设每个方向扩展像素点数量为 p，则填充后原始图片的大小为 $(m+2p, n+2p)$，卷积核大小保持 (f,f) 不变，则输出图片大小 $(m+2p-f+1, n+2p-f+1)$。常用填充的方法有：

- 有效 (valid) 卷积，不填充，直接卷积，结果大小为 $(m-f+1, n-f+1)$。
- 相同 (same) 卷积，用 0 填充，并使得卷积后结果大小与输入一致，这样 $p=(f-1)/2$。
- 全 (full) 卷积，通过填充，使得输出尺寸为 $(m+f-1, n+f-1)$。

我们将之前的例子用 $p=1$ 填充，如图 7 所示。

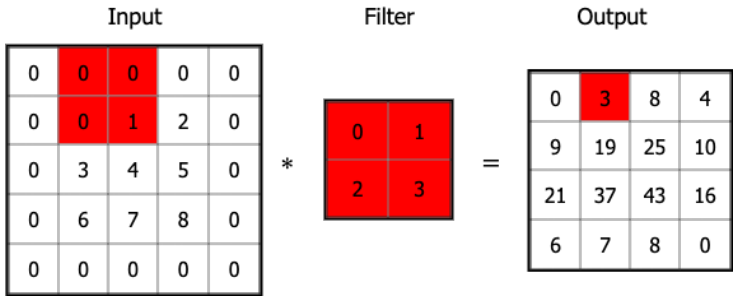

图 7. $p=1$ 填充，将 $(3,3)$ 的输入填充为 $(5,5)$。再与核运算得到输出。

3.4 卷积步幅 (Stride)

除了需要通过填充来避免信息损失，有时也需要通过设置步幅 (Stride) 来压缩一部分信息。步幅表示核在原始图片的水平方向和垂直方向上每次移动的距离。使用卷积步幅，跳过核中的一些位置 (看作对输出的下采样) 来降低计算的开销。如图 8 所示，在高上步幅为 3、在宽上步幅为 2 的

卷积操作。当输出第一列第二个元素时，卷积窗口向下滑动了 3 行，而在输出第一行第二个元素时卷积窗口向右滑动了 2 列。通常我们设置在水平方向和垂直方向的步幅一样，如果步幅设为 s，则输出尺寸为 $(\lfloor \frac{m+2p-f}{s}+1 \rfloor, \lfloor \frac{n+2p-f}{s}+1 \rfloor)$。

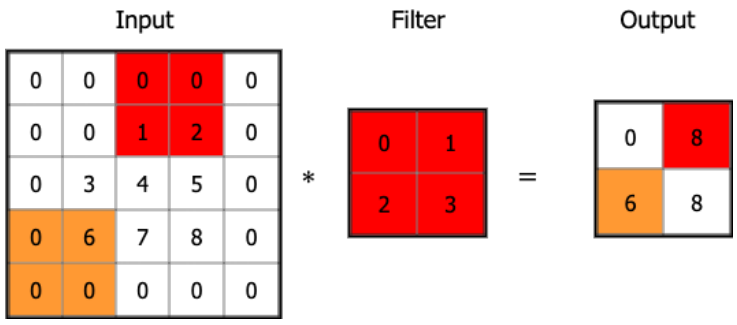

图 8. 考虑步幅下的卷积运算，在高上步幅为 3、在宽上步幅为 2。核第二步与输入的红色区域运算得到输出的右上角红色区域，第三步与输入的橙色区域运算得到输出的左下角橙色区域。

4 更多的卷积策略

4.1 深度可分离卷积 (Depthwise Separable Convolution)

如图所示，对输入 3×3×2 的数组，经过 4 组 2×2×2 的卷积核，得到 2×2×4 的输出。这里我们一共需要 2×2×2×4=32 个参数。我们使用深度可分离卷积，它将卷积过程分成两个步骤。第一步，在 Depthwise Convolution，输入有几个通道就设几个卷积核，如图 9 所示，输入一共 2 个通道，对每个通道分配一个卷积核，这里的每个卷积核只处理一个通道 (对比原始卷积过程每组卷积核处理所有通道)；第二步，在 Pointwise Convolution，由于在上一步不同通道间没有联系，因此这一步用 1×1 的卷积核组来获得不同通道间的联系。我们可以看出，在深度可分离卷积中，我们一共需要 2×2×2+1×1×2×4=16 个参数。

更形式化的，我们假设：

- 输入尺寸：$(\mathrm{H}_{in}, \mathrm{W}_{in}, c_1)$
- 卷积核尺寸：$(\mathrm{K}, \mathrm{K}, c_1)$
- 输出尺寸：$(\mathrm{H}_{out}, \mathrm{W}_{out}, c_2)$

我们需要的标准卷积核参数量为 $\mathrm{K} \times \mathrm{K} \times c_1 \times c_2$。在深度可分离卷积中，第一步需要的参数量为 $\mathrm{K} \times \mathrm{K} \times 1 \times c_1$，第二步需要的参数为 $1 \times 1 \times c_1 \times c_2$，一共 $\mathrm{K} \times \mathrm{K} \times 1 \times c_1 + 1 \times 1 \times c_1 \times c_2$。所以深度可分离卷积参数量是标准卷积的 $\frac{1}{c_2} + \frac{1}{\mathrm{K}^2}$。

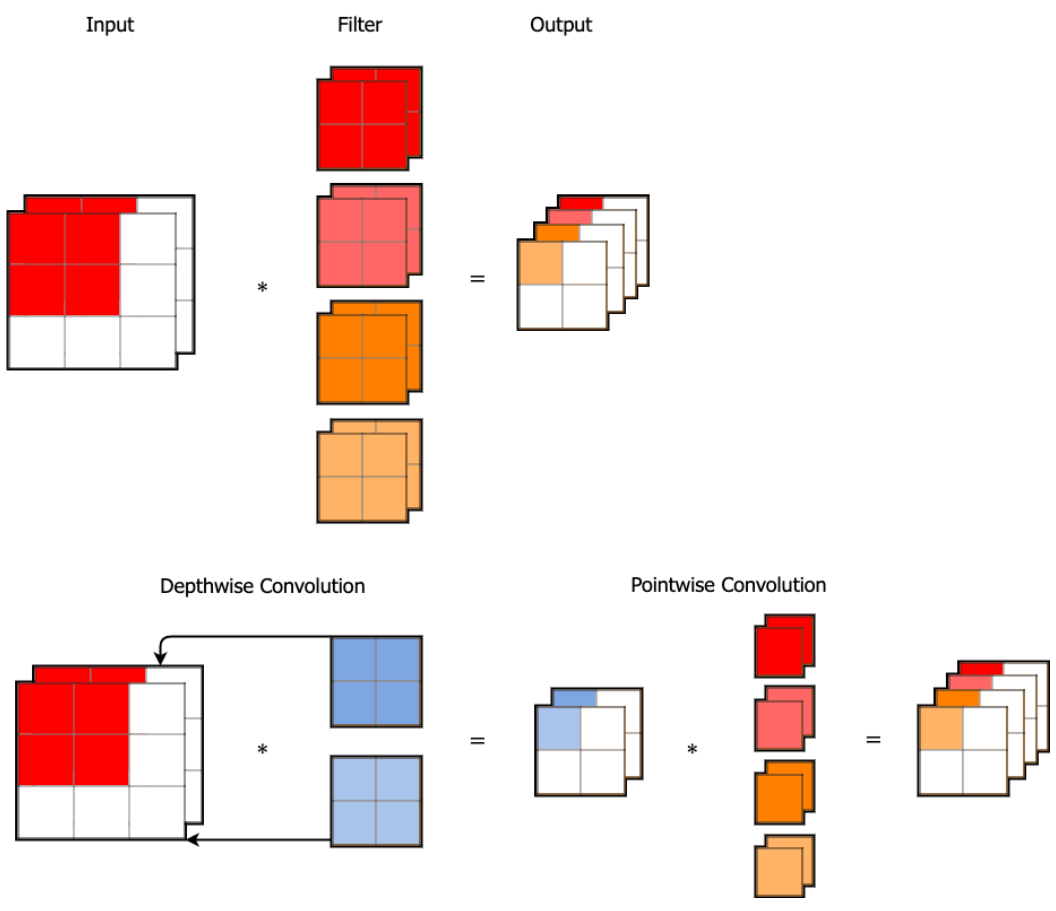

图 9. 上方显示了原始的卷积过程，构造四组通道数为 2 的卷积核组，经过运算得到输出。下方显示了深度可分离卷积，先经过一组卷积核，再经过四组通道数为 2 但更小的卷积核组。

4.2 分组卷积 (Group Convolution)

如图 10 所示，我们先考虑标准卷积，对输入为 3×3×4 的数组，经过 2 组 2×2×4 的卷积核，得到 2×2×2 的输出。这里我们一共需要 2×2×4×2=32 个参数。我们将输入数组依据通道分为 2 组，每组需要 2×2×2 的卷积核就可以得到 2×2×1 的输出，拼接在一起同样得到 2×2×2 的输出。在这个分组卷积中，我们一共需要 2×2×2×2=16 个参数。

更形式化的，我们假设：

- 输入尺寸：$(\mathrm{H}_{in}, \mathrm{W}_{in}, c_1)$
- 卷积核尺寸：$(\mathrm{K}, \mathrm{K}, c_1)$
- 输出尺寸：$(\mathrm{H}_{out}, \mathrm{W}_{out}, c_2)$

我们需要的标准卷积核参数量为 $\mathrm{K} \times \mathrm{K} \times c_1 \times c_2$。在分组卷积中，假设被分为 g 组，则每一组输入的尺寸为 $(\mathrm{H}_{in}, \mathrm{W}_{in}, c_1/g)$，对应该组需要的卷积核组的尺寸为 $(\mathrm{K}, \mathrm{K}, c_1/g, c_2/g)$，输出尺寸为 $(\mathrm{H}_{out}, \mathrm{W}_{out}, c_2/g)$。最后，将 g 组的结果拼接在一起，最终得到 $(\mathrm{H}_{out}, \mathrm{W}_{out}, c_2)$ 大小的输出。在这个过程中，分组卷积需要的卷积核参数量为 $\mathrm{K} \times \mathrm{K} \times (c_1/g) \times (c_2/g) \times g$，是标准卷积的 $\frac{1}{g}$。

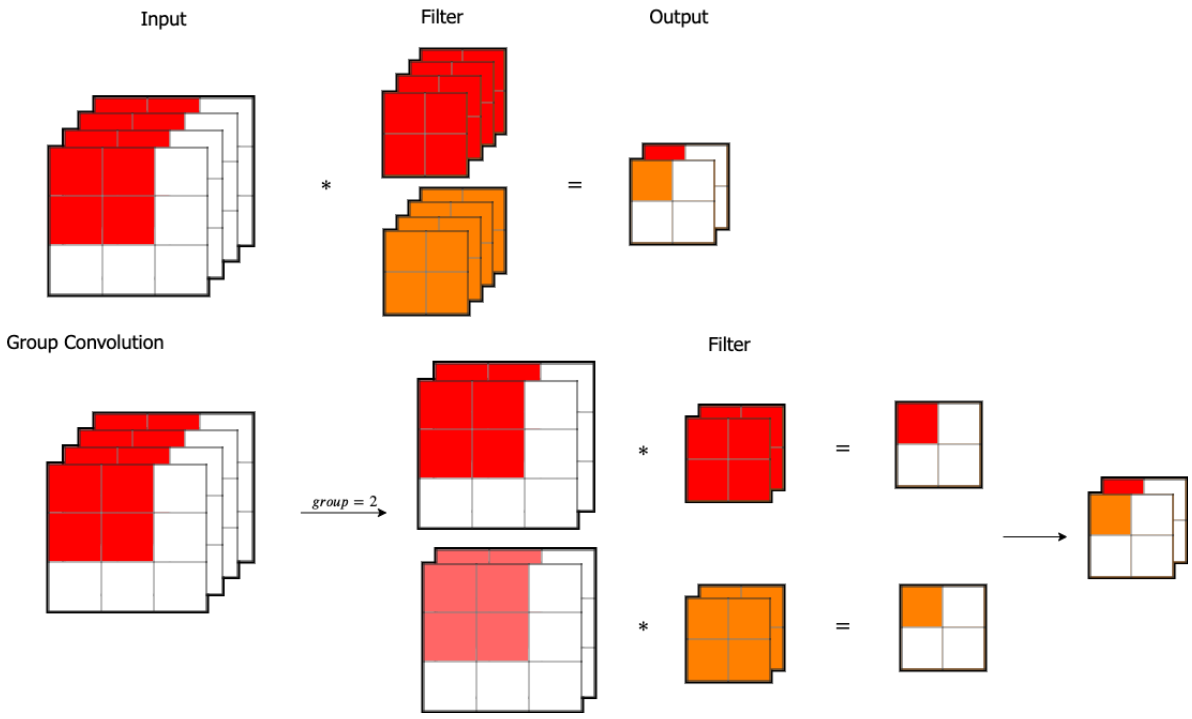

图 10. 上方显示了原始的卷积过程，构造两组通道数为 4 的卷积核组，经过运算得到输出。下方显示了分组卷积，将原始输入分为两组，每组通道数为 2。再分别经过一组通道数为 2 的卷积核。将分别得到的结果拼接得到最终输出。

4.3 扩张卷积 (Dilated Convolution)

扩张卷积，也称空洞卷积，它引入的参数被称为扩张率 (Dilation rate)，其定义了核内值之间的间距。如图 11 所示，图中扩张速率为 2 的 3×3 内核将具有与 5×5 内核相同的视野，但只使用 9 个参数。这种方法能以相同的计算成本，提供更大的感受野。在需要更大的观察范围，且无法承受多个卷积或更大的内核，可以用它。

如果输入图片的大小为 (m, n)，而卷积核的大小为 (f, f)，每个方向扩展像素点数量为 p，步幅设为 s，则标准卷积输出尺寸为 $(\lfloor \frac{m+2p-f}{s}+1 \rfloor, \lfloor \frac{n+2p-f}{s}+1 \rfloor)$。如果扩张率为 r，则扩张卷积输出尺寸为 $(\lfloor \frac{m+2p-[f+(f-1)(r-1)]}{s}+1 \rfloor, \lfloor \frac{n+2p-[f+(f-1)(r-1)]}{s}+1 \rfloor)$。

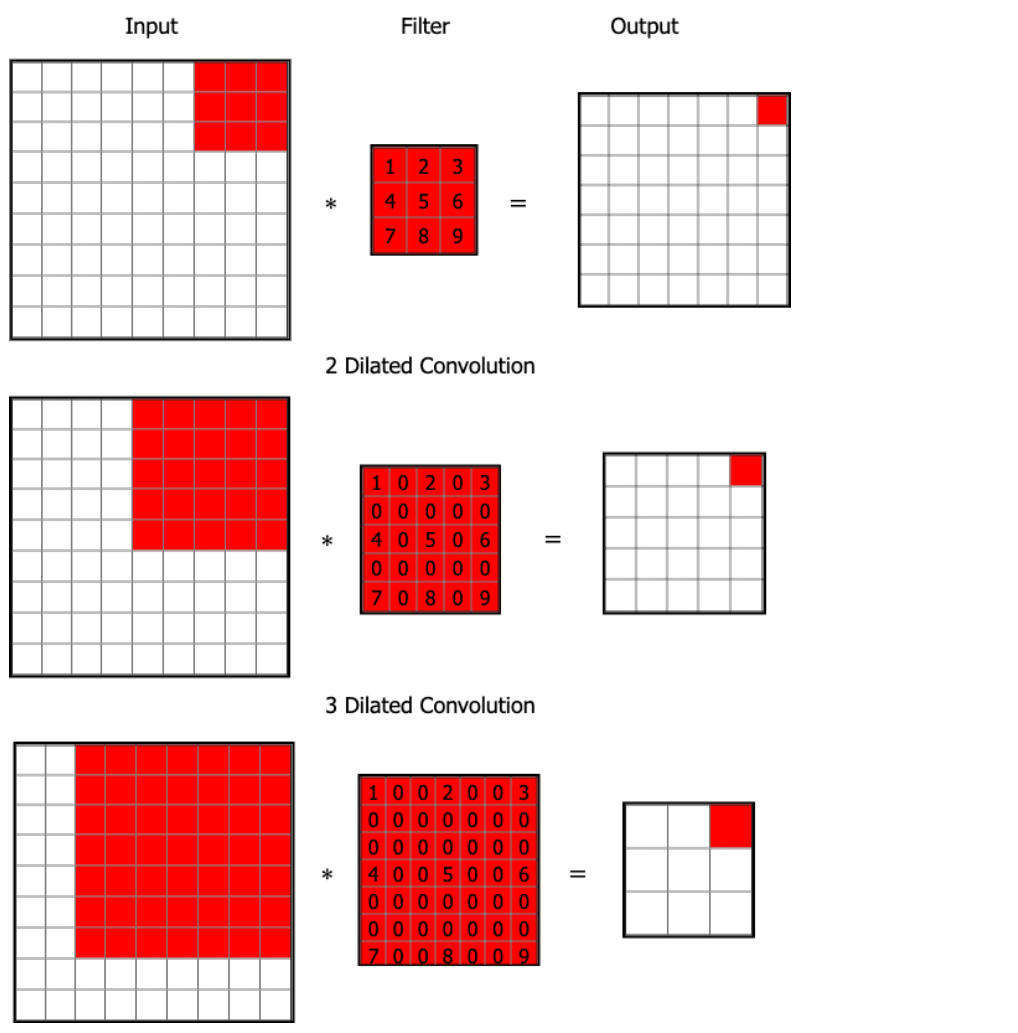

图 11. 分别显示了不考虑扩张率 (默认为 1)，以及考虑扩张率 (为 2 和 3) 的效果。

[1]:
```
import numpy as np
import time
import sys
sys.path.append('../')
from method.optimizer import OptimizerInitializer
from method.weight import WeightInitializer
from method.activation import ActivationInitializer
from chapter6 import LayerBase, CrossEntropy, FullyConnected, minibatch, softmax
from collections import OrderedDict
```

[2]:
```
"""
Padding 操作
"""
def calc_pad_dims_sameconv_2D(X_shape, out_dim, kernel_shape, stride, dilation=1):
    """
    当填充方式为相同卷积时，计算 padding 的数目，保证输入输出的大小相同。这里在卷积过程中考虑填充 (Padding)，
    卷积步幅 (Stride)，扩张率 (Dilation rate)。根据扩张卷积的输出公式可以得到 padding 的数目。

    参数说明:
    X_shape: 输入数组，为 (n_samples, in_rows, in_cols, in_ch)
    out_dim: 输出数组维数，为 (out_rows, out_cols)
    kernel_shape: 卷积核形状，为 (fr, fc)
    stride: 卷积步幅，int 型
    dilation: 扩张率，int 型，default=1
    """
    d = dilation
    fr, fc = kernel_shape
    out_rows, out_cols = out_dim
    n_ex, in_rows, in_cols, in_ch = X_shape
    # 考虑扩张率
    _fr, _fc = fr + (fr-1) * (d-1), fc + (fc-1) * (d-1)
    # 计算 padding 维数
    pr = int((stride * (out_rows-1) + _fr - in_rows) / 2)
    pc = int((stride * (out_cols-1) + _fc - in_cols) / 2)
    # 校验，如不等 (right/bottom 处) 添加不对称 0 填充
    out_rows1 = int(1 + (in_rows + 2 * pr - _fr) / stride)
    out_cols1 = int(1 + (in_cols + 2 * pc - _fc) / stride)
    pr1, pr2 = pr, pr
    if out_rows1 == out_rows - 1:
        pr1, pr2 = pr, pr + 1
    elif out_rows1 != out_rows:
        raise AssertionError
    pc1, pc2 = pc, pc
    if out_cols1 == out_cols - 1:
        pc1, pc2 = pc, pc + 1
    elif out_cols1 != out_cols:
        raise AssertionError
    # 返回对 X 的 Padding 维数 (left, right, up, down)
    return (pr1, pr2, pc1, pc2)


def pad2D(X, pad, kernel_shape=None, stride=None, dilation=1):
    """
    二维填充

    参数说明:
    X: 输入数组，为 (n_samples, in_rows, in_cols, in_ch)，
        其中 padding 操作是应用到 in_rows 和 in_cols
    pad: padding 数目，4-tuple, int, 或 'same', 'valid'
```

```
        在图片的左、右、上、下 (left, right, up, down) 0 填充
        若为 int，表示在左、右、上、下均填充数目为 pad 的 0，
        若为 same，表示填充后为相同 (same) 卷积，
        若为 valid，表示填充后为有效 (valid) 卷积
    kernel_shape: 卷积核形状，为 (fr, fc)
    stride: 卷积步幅，int 型
    dilation: 扩张率，int 型，default=1
    """
    p = pad
    if isinstance(p, int):
        p = (p, p, p, p)
    if isinstance(p, tuple):
        X_pad = np.pad(
            X,
            pad_width=((0, 0), (p[0], p[1]), (p[2], p[3]), (0, 0)),
            mode="constant",
            constant_values=0,
        )
    # 'same' 卷积，首先计算 padding 维数
    if p == "same" and kernel_shape and stride is not None:
        p = calc_pad_dims_sameconv_2D(
            X.shape, X.shape[1:3], kernel_shape, stride, dilation=dilation
        )
        X_pad, p = pad2D(X, p)
    if p == "valid":
        p = (0, 0, 0, 0)
        X_pad, p = pad2D(X, p)
    return X_pad, p
```

[3]:
```
def conv2D(X, W, stride, pad, dilation=1):
    """
    二维卷积实现过程。

    参数说明:
    X: 输入数组，为 (n_samples, in_rows, in_cols, in_ch)
    W: 卷积层的卷积核参数，为 (kernel_rows, kernel_cols, in_ch, out_ch)
    stride: 卷积核的卷积步幅，int 型
    pad: padding 数目，4-tuple, int, 或 'same', 'valid' 型
        在图片的左、右、上、下 (left, right, up, down) 0 填充
        若为 int，表示在左、右、上、下均填充数目为 pad 的 0，
        若为 same，表示填充后为相同 (same) 卷积，
        若为 valid，表示填充后为有效 (valid) 卷积
    dilation: 扩张率，int 型，default=1

    输出说明:
    Z: 卷积结果，为 (n_samples, out_rows, out_cols, out_ch)
    """
    s, d = stride, dilation
    X_pad, p = pad2D(X, pad, W.shape[:2], stride=s, dilation=d)
    pr1, pr2, pc1, pc2 = p
    fr, fc, in_ch, out_ch = W.shape
    n_samp, in_rows, in_cols, in_ch = X.shape
    # 考虑扩张率
    _fr, _fc = fr + (fr-1) * (d-1), fc + (fc-1) * (d-1)
    out_rows = int((in_rows + pr1 + pr2 - _fr) / s + 1)
    out_cols = int((in_cols + pc1 + pc2 - _fc) / s + 1)
    Z = np.zeros((n_samp, out_rows, out_cols, out_ch))
    for m in range(n_samp):
        for c in range(out_ch):
```

```
            for i in range(out_rows):
                for j in range(out_cols):
                    i0, i1 = i * s, (i * s) + fr + (fr-1) * (d-1)
                    j0, j1 = j * s, (j * s) + fc + (fc-1) * (d-1)
                    window = X_pad[m, i0 : i1 : d, j0 : j1 : d, :]
                    Z[m, i, j, c] = np.sum(window * W[:, :, :, c])
    return Z
```

5 GEMM 转换

在前面介绍的二维卷积运算代码中我们会有 4 个 for 循环，这需要很大的时间开销，我们能否有更快的计算方法?

矩阵乘 (General Matrix Multiplication，GEMM) 是深度学习的核心。前面的全连接层是通过矩阵乘实现的，卷积层是否也可以这样实现?

卷积核是在输入图像上按步长滑动，每次滑动操作在输入图像上的对应窗口的区域，将卷积核中的各个权值 w_* 与输入图像上对应窗口中的各个值 x_* 相乘，然后相加得到输出特征图上的一个值。卷积核对输入图像的每一次运算，可以看作是两个向量的内积，这意味着卷积操作完全可以转化为矩阵的乘法来实现。

我们将卷积层的卷积操作转化为卷积核的权值矩阵与输入数组转化的输入矩阵进行相乘。我们现在假设：

- 卷积核组：$(c_2, c_1, \mathrm{K}, \mathrm{K})$，单组卷积核 Kernel 的大小为 $(c_1, \mathrm{K}, \mathrm{K})$；
- 输入数组：$(c_1, \mathrm{H}_{in}, \mathrm{W}_{in})$；

转化如图 12 所示：图中单组卷积核 Kernel 在输入图像上滑动得到各个 Patch_*，Patch_* 的大小同卷积核。我们将 Kernel 和 Patch_* 均展开成一维向量。假设有 $n_k (n_k = c_2)$ 组卷积核，并且每组卷积核在输入上所有的滑动可以得到 n_p 个 Patch，于是可以得到输入矩阵 (l_{col}, n_p) 以及权值矩阵 (n_k, l_{col})。

- 权值矩阵：$(c_2, c_1 \times \mathrm{K} \times \mathrm{K})$；
- 输入矩阵：$(c_1 \times \mathrm{K} \times \mathrm{K}, \mathrm{H}_{out} \times \mathrm{W}_{out})$

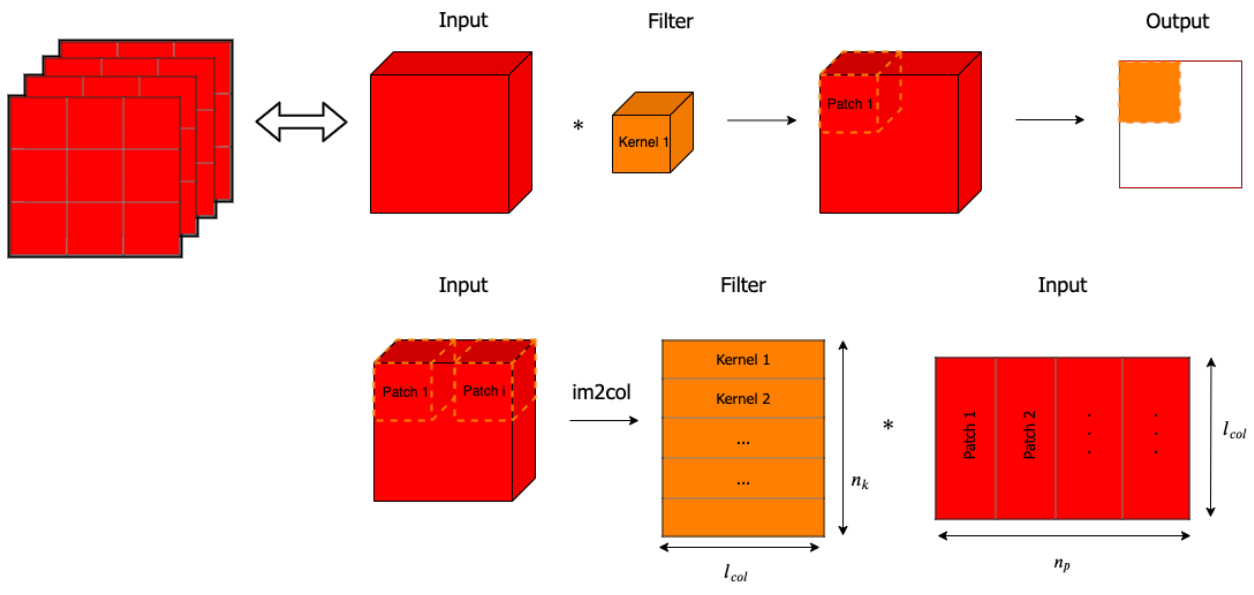

图 12. im2col 操作过程。一组卷积核在三维图像上滑动的过程，可以视作卷积核与输入图像上不同的块分别做运算。基于块可以将输入图像数组转化为 (l_{col}, n_p)。如果有 n_k 组卷积核，也将卷积核组转化为 (n_k, l_{col})。

再具体看一下 Kernel 和 Patch 的展开过程。对于 Kernel，我们可以依图 13 所示展开，图中所示为 1 组卷积核有 3 个通道，将卷积核先依通道再依行依次放入。

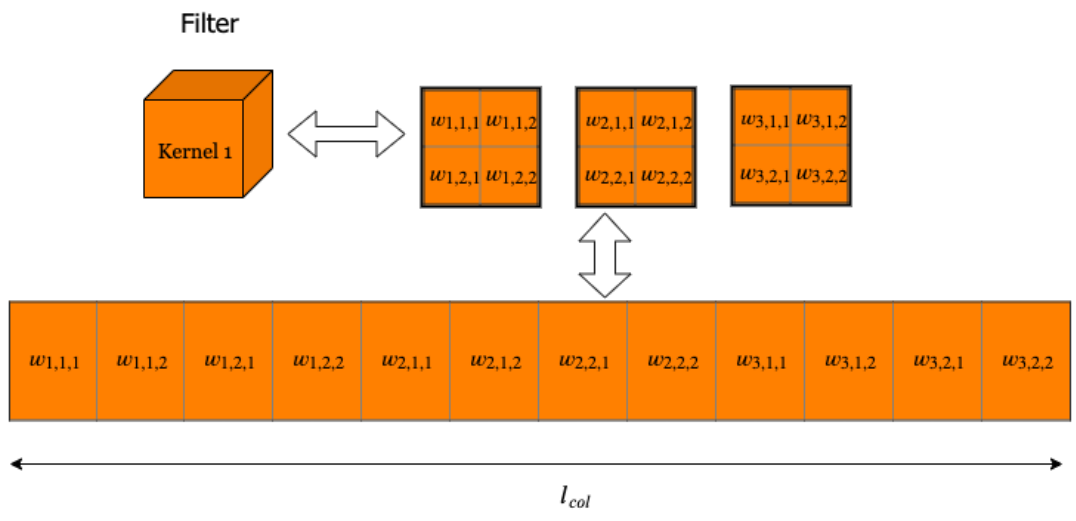

图 13. im2col 操作中过程，将单个卷积核组 (三维) 展开成一列向量的实现过程。

Patch 的展开过程同样。而每个 Patch 是 Kernel 在输入上滑动得到的，如果我们记 Patch 的右上角在图像中坐标为 $(:, i, j)$，则 Patch 中的元素为：$\{x_{c,i:i+h,j:j+h}\}, c = 1, ..., c_1, h = \mathrm{K}$，而 (i, j) 的取值 $i = 1, ..., \mathrm{H}_{out}, j = 1, ..., \mathrm{W}_{out}$。如果考虑扩张卷积，则元素可以写作：$\{x_{c,i:r:i+rh,j:r:j+rh}\}$。

[4]:
```
"""
下面展示 conv2D 的 GEMM 实现过程，将 X 和 W 转化为 2D 矩阵，
这里我们将 X 转化为 (kernel_rows*kernel_cols*in_ch, n_samples*out_rows*out_cols)
W 转化为 (out_ch, kernel_rows*kernel_cols*in_ch)
"""
def _im2col_indices(X_shape, fr, fc, p, s, d=1):
    """
    生成输入矩阵的 (c,h_in,w_in) 三个维度的索引

    输出说明:
    i: 输入矩阵的 i 值，(kernel_rows*kernel_cols*in_ch, out_rows*out_cols)，图示中第二维坐标
    j: 输入矩阵的 j 值，(kernel_rows*kernel_cols*in_ch, out_rows*out_cols)，图示中第三维坐标
    k: 输入矩阵的 c 值，(kernel_rows*kernel_cols*in_ch, 1)，图示中第一维坐标
    """
    pr1, pr2, pc1, pc2 = p
    n_ex, n_in, in_rows, in_cols = X_shape
    # 考虑扩张率
    _fr, _fc = fr + (fr-1) * (d-1), fc + (fc-1) * (d-1)
    out_rows = int((in_rows + pr1 + pr2 - _fr) / s + 1)
    out_cols = int((in_cols + pc1 + pc2 - _fc) / s + 1)
    # i0/i1/j0/j1: 用于得到 i，j，k。i0/j0 过程见图示，i1/j1 由滑动过程得出
    i0 = np.repeat(np.arange(fr), fc)
    i0 = np.tile(i0, n_in) * d
    i1 = s * np.repeat(np.arange(out_rows), out_cols)
    j0 = np.tile(np.arange(fc), fr * n_in) * d
    j1 = s * np.tile(np.arange(out_cols), out_rows)
    i = i0.reshape(-1, 1) + i1.reshape(1, -1)
    j = j0.reshape(-1, 1) + j1.reshape(1, -1)
    k = np.repeat(np.arange(n_in), fr * fc).reshape(-1, 1)
    return k, i, j


def im2col(X, W_shape, pad, stride, dilation=1):
    """
    im2col 实现

    参数说明:
    X: 输入数组，为 (n_samples, in_rows, in_cols, in_ch)，此时还未 0 填充 (padding)
    W_shape: 卷积层的卷积核的形状，为 (kernel_rows, kernel_cols, in_ch, out_ch)
    pad: padding 数目，4-tuple, int, 或 'same', 'valid' 型
        在图片的左、右、上、下 (left, right, up, down) 0 填充
        若为 int，表示在左、右、上、下均填充数目为 pad 的 0，
        若为 same，表示填充后为相同 (same) 卷积，
        若为 valid，表示填充后为有效 (valid) 卷积
    stride: 卷积核的卷积步幅，int 型
    dilation: 扩张率，int 型，default=1

    输出说明:
    X_col: 输出结果，形状为 (kernel_rows*kernel_cols*n_in, n_samples*out_rows*out_cols)
    p: 填充数，4-tuple
    """
    fr, fc, n_in, n_out = W_shape
    s, p, d = stride, pad, dilation
    n_samp, in_rows, in_cols, n_in = X.shape
    X_pad, p = pad2D(X, p, W_shape[:2], stride=s, dilation=d)
    pr1, pr2, pc1, pc2 = p
    # 将输入的通道维数移至第二位
```

```
    X_pad = X_pad.transpose(0, 3, 1, 2)
    k, i, j = _im2col_indices((n_samp, n_in, in_rows, in_cols), fr, fc, p, s, d)
    # X_col.shape = (n_samples, kernel_rows*kernel_cols*n_in, out_rows*out_cols)
    X_col = X_pad[:, k, i, j]
    X_col = X_col.transpose(1, 2, 0).reshape(fr * fc * n_in, -1)
    return X_col, p


def conv2D_gemm(X, W, stride=0, pad='same', dilation=1):
    """
    二维卷积实现过程，依靠“im2col”函数将卷积作为单个矩阵乘法执行。

    参数说明：
    X: 输入数组，为 (n_samples, in_rows, in_cols, in_ch)
    W: 卷积层的卷积核参数，为 (kernel_rows, kernel_cols, in_ch, out_ch)
    stride: 卷积核的卷积步幅，int 型
    pad: padding 数目，4-tuple, int, 或 'same', 'valid' 型
        在图片的左、右、上、下 (left, right, up, down) 0 填充
        若为 int，表示在左、右、上、下均填充数目为 pad 的 0，
        若为 same，表示填充后为相同 (same) 卷积，
        若为 valid，表示填充后为有效 (valid) 卷积
dilation: 扩张率，int 型，default=1

输出说明：
Z: 卷积结果，为 (n_samples, out_rows, out_cols, out_ch)
"""
s, d = stride, dilation
_, p = pad2D(X, pad, W.shape[:2], s, dilation=dilation)
pr1, pr2, pc1, pc2 = p
fr, fc, in_ch, out_ch = W.shape
n_samp, in_rows, in_cols, in_ch = X.shape
# 考虑扩张率
_fr, _fc = fr + (fr-1) * (d-1), fc + (fc-1) * (d-1)
# 输出维数，根据上面公式可得
out_rows = int((in_rows + pr1 + pr2 - _fr) / s + 1)
out_cols = int((in_cols + pc1 + pc2 - _fc) / s + 1)
# 将 X 和 W 转化为 2D 矩阵并乘积
X_col, _ = im2col(X, W.shape, p, s, d)
W_col = W.transpose(3, 2, 0, 1).reshape(out_ch, -1)
Z = (W_col @ X_col).reshape(out_ch, out_rows, out_cols, n_samp).transpose(3, 1, 2, 0)
return Z
```

6 卷积网络的训练

卷积的内部实现实际上是**矩阵相乘**，因此，卷积的**反向传播**过程实际上跟普通的全连接是类似的。

6.1 卷积网络示意图

CNN 的基本层包括卷积层和池化层，二者通常一起使用 (一个池化层紧跟一个卷积层之后)。这两层包括三个级联的函数：卷积，激励函数 (如 sigmoid 函数)，池化。其前向传播和后向传播的示意图如下：

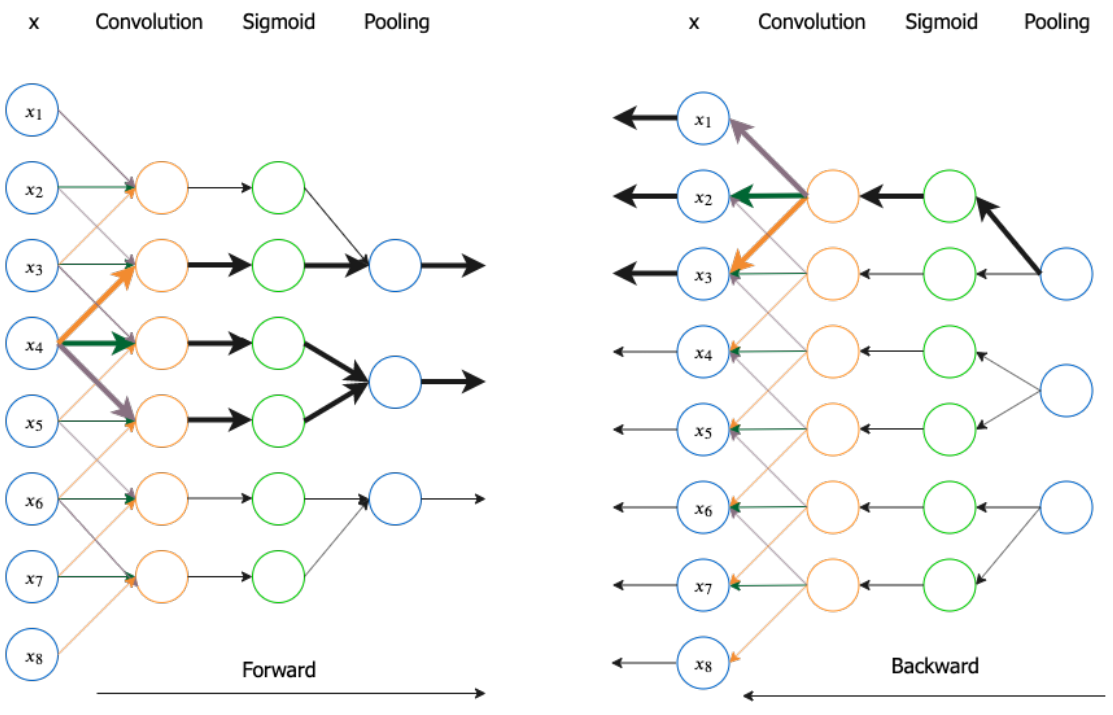

图 14. 卷积网络的前向传播与反向传播示意图 (这里输入为一维向量，即一维卷积)。

6.2 单层卷积层/池化层

6.2.1 卷积函数的导数及反向传播

从一维卷积入手，假设一个卷积过程的输入向量是 $\boldsymbol{x}$，输出向量是 $\boldsymbol{y}$, 参数向量 (卷积算子) 是 $\boldsymbol{w}$。从输入到输出的过程为：

$$\boldsymbol{y} = \boldsymbol{x} * \boldsymbol{w} \tag{3}$$

其中 $\boldsymbol{y}$ 的长度为 $|\boldsymbol{y}|$，$\boldsymbol{y}$ 中每一个元素的计算方法为：

$$y_n = (\boldsymbol{x} * \boldsymbol{w})[n] = \sum_{k=1}^{|\boldsymbol{w}|} x_{n+k-1} w_k = w^\top x_{n:n+|w|-1} \tag{4}$$

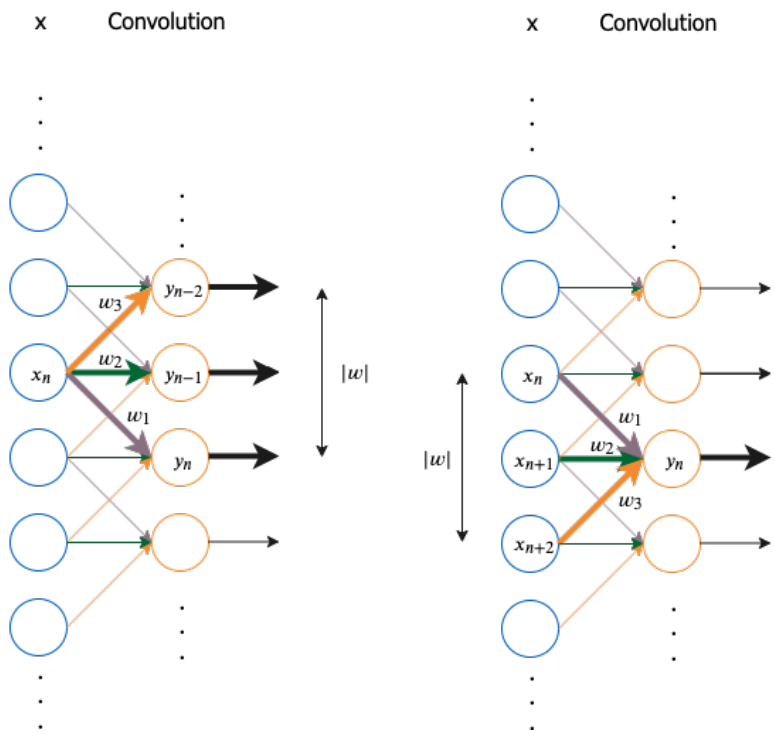

图 15. 卷积函数的前向传播与反向传播示意图 (一维卷积)。

$\boldsymbol{y}$ 中的元素与 $\boldsymbol{x}$ 中的元素有如下导数关系：

$$\frac{\partial y_{n-k+1}}{\partial x_n} = w_k, \quad \frac{\partial y_n}{\partial w_k} = x_{n+k-1}, \quad \text{for } 1 \le k \le |w| \tag{5}$$

进一步可以得到 J 关于 $\boldsymbol{w}$ 和 $\boldsymbol{x}$ 的导数：

$$\begin{aligned}
\delta_n^{(\boldsymbol{x})} = \frac{\partial J}{\partial \boldsymbol{y}} \frac{\partial \boldsymbol{y}}{\partial x_n} = \sum_{k=1}^{|\boldsymbol{w}|} \frac{\partial J}{\partial y_{n-k+1}} \frac{\partial y_{n-k+1}}{\partial x_n} = \sum_{k=1}^{|\boldsymbol{w}|} \delta_{n-k+1}^{(\boldsymbol{y})} w_k = (\delta^{(\boldsymbol{y})} * \text{flip}(\boldsymbol{w}))[n] \\
\delta^{(\boldsymbol{x})} = \delta^{(\boldsymbol{y})} * \text{flip}(\boldsymbol{w}) \\
\frac{\partial}{\partial w_k} J = \frac{\partial J}{\partial \boldsymbol{y}} \frac{\partial \boldsymbol{y}}{\partial w_k} = \sum_{n=1}^{|\boldsymbol{y}|} \frac{\partial J}{\partial y_n} \frac{\partial y_n}{\partial w_k} = \sum_{n=1}^{|\boldsymbol{y}|} \delta_n^{(\boldsymbol{y})} x_{n+k-1} = (\delta^{(\boldsymbol{y})} * \boldsymbol{x})[k] \\
\frac{\partial}{\partial \boldsymbol{w}} J = \delta^{(\boldsymbol{y})} * \boldsymbol{x}
\end{aligned} \tag{6}$$

因此，通过 $\delta^{(y)}$ 与 $\text{flip}(\boldsymbol{w})$ 可得到 J 关于 $\boldsymbol{x}$ 的导数 $\delta^{(x)}$，通过 $\delta^{(y)}$ 与 $\boldsymbol{x}$ 可计算出 $\boldsymbol{w}$ 的梯度 $\frac{\partial}{\partial \boldsymbol{w}} J$。

如果考虑偏置和激活函数，则前向传播时，$\boldsymbol{y}$ 经过激活函数 g 得到 $\boldsymbol{a} = g(\boldsymbol{y}) = g(\boldsymbol{x} * \boldsymbol{w} + \boldsymbol{b})$；反向传播时，$\delta^{(y)} = \delta^{(a)} \odot g'(\boldsymbol{y})$。于是 $\delta^{(x)} = \delta^{(y)} * \text{flip}(\boldsymbol{w}) \odot g'(\boldsymbol{y})$。

至此，我们拆解了卷积核作用在输入上的导数与反向传播过程。映射到二维卷积，这也是得到了一个二维卷积核作用在二维输入数组上的结论。推导二维卷积的导数与反向传播的过程类似。

在实际的卷积核中，我们还需要考虑通道。上文介绍的多通道输入，后一层的每个通道都是由前一层的各个通道经过卷积再求和得到的。我们将这个过程可以理解为全连接，节点为一个通道的输入数组。

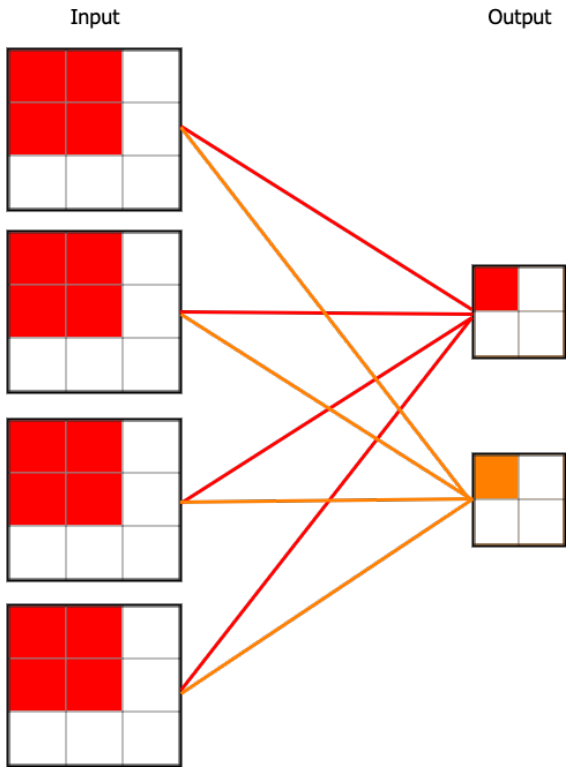

图 16. 考虑多通道下的卷积运算。输入的每个通道先与不同卷积核组相应的卷积核运算，再累加得到输出的对应位置。这个过程可以视作全连接，将单个卷积核视作权重。

[5]:
```python
class Conv2D(LayerBase):

    def __init__(
        self,
        out_ch,
        kernel_shape,
        pad=0,
        stride=1,
        dilation=1,
        acti_fn=None,
        optimizer=None,
        init_w="glorot_uniform",
    ):
        """
        二维卷积

        参数说明:
        out_ch: 卷积核组的数目，int 型
        kernel_shape: 单个卷积核形状，2-tuple
        acti_fn: 激活函数，str 型
        pad: padding 数目，4-tuple, int, 或 'same', 'valid' 型
            在图片的左、右、上、下 (left, right, up, down) 0 填充
            若为 int，表示在左、右、上、下均填充数目为 pad 的 0，
            若为 same，表示填充后为相同 (same) 卷积，
            若为 valid，表示填充后为有效 (valid) 卷积
        stride: 卷积核的卷积步幅，int 型
        dilation: 扩张率，int 型，default=1
        init_w: 权重初始化方法，str 型
        optimizer: 优化方法，str 型
        """
        super().__init__(optimizer)
```

```
        self.pad = pad
        self.in_ch = None
        self.out_ch = out_ch
        self.stride = stride
        self.dilation = dilation
        self.kernel_shape = kernel_shape
        self.init_w = init_w
        self.init_weights = WeightInitializer(mode=init_w)
        self.acti_fn = ActivationInitializer(acti_fn)()
        self.parameters = {"W": None, "b": None}
        self.is_initialized = False

    def _init_params(self):
        fr, fc = self.kernel_shape
        W = self.init_weights((fr, fc, self.in_ch, self.out_ch))
        b = np.zeros((1, 1, 1, self.out_ch))

        self.params = {"W": W, "b": b}
        self.gradients = {"W": np.zeros_like(W), "b": np.zeros_like(b)}
        self.derived_variables = {"Y": []}
        self.is_initialized = True

    def forward(self, X, retain_derived=True):
        """
        卷积层的前向传播，原理见上文。

        参数说明：
        X: 输入数组，形状为 (n_samples, in_rows, in_cols, in_ch)
        retain_derived: 是否保留中间变量，以便反向传播时再次使用，bool 型

        输出说明：
        a: 卷积层输出，形状为 (n_samples, out_rows, out_cols, out_ch)
        """
        if not self.is_initialized:
            self.in_ch = X.shape[3]
            self._init_params()
        W = self.params["W"]
        b = self.params["b"]
        n_samp, in_rows, in_cols, in_ch = X.shape
        s, p, d = self.stride, self.pad, self.dilation
        # 卷积操作
        Y = conv2D(X, W, s, p, d) + b
        a = self.acti_fn(Y)
        if retain_derived:
            self.X.append(X)
            self.derived_variables["Y"].append(Y)
        return a

    def backward(self, dLda, retain_grads=True):
        """
        卷积层的反向传播，原理见上文。

        参数说明：
        dLda: 关于损失的梯度，为 (n_samples, out_rows, out_cols, out_ch)
        retain_grads: 是否计算中间变量的参数梯度，bool 型

        输出说明：
        dXs: 即 dX，当前卷积层对输入关于损失的梯度，为 (n_samples, in_rows, in_cols, in_ch)
```

```
        """
        if not isinstance(dLda, list):
            dLda = [dLda]
        W = self.params["W"]
        b = self.params["b"]
        Ys = self.derived_variables["Y"]
        Xs, d = self.X, self.dilation
        (fr, fc), s, p = self.kernel_shape, self.stride, self.pad
        dXs = []
        for X, Y, da in zip(Xs, Ys, dLda):
            n_samp, out_rows, out_cols, out_ch = da.shape
            X_pad, (pr1, pr2, pc1, pc2) = pad2D(X, p, self.kernel_shape, s, d)
            dY = da * self.acti_fn.grad(Y)
            dX = np.zeros_like(X_pad)
            dW, db = np.zeros_like(W), np.zeros_like(b)
            for m in range(n_samp):
                for i in range(out_rows):
                    for j in range(out_cols):
                        for c in range(out_ch):
                            i0, i1 = i * s, (i * s) + fr + (fr-1) * (d-1)
                            j0, j1 = j * s, (j * s) + fc + (fc-1) * (d-1)
                            wc = W[:, :, :, c]
                            kernel = dY[m, i, j, c]
                            window = X_pad[m, i0:i1:d, j0:j1:d, :]
                            db[:, :, :, c] += kernel
                            dW[:, :, :, c] += window * kernel
                            dX[m, i0:i1:d, j0:j1:d, :] += (
                                wc * kernel
                            )
            if retain_grads:
                self.gradients["W"] += dW
                self.gradients["b"] += db
            pr2 = None if pr2 == 0 else -pr2
            pc2 = None if pc2 == 0 else -pc2
            dXs.append(dX[:, pr1:pr2, pc1:pc2, :])
        return dXs[0] if len(Xs) == 1 else dXs

    @property
    def hyperparams(self):
        return {
            "layer": "Conv2D",
            "pad": self.pad,
            "init_w": self.init_w,
            "in_ch": self.in_ch,
            "out_ch": self.out_ch,
            "stride": self.stride,
            "dilation": self.dilation,
            "acti_fn": str(self.acti_fn),
            "kernel_shape": self.kernel_shape,
            "optimizer": {
                "cache": self.optimizer.cache,
                "hyperparams": self.optimizer.hyperparams,
            },
        }
```

如果我们使用 GEMM 转换实现前向传播和反向传播，权值矩阵 $\boldsymbol{W}$ 形状为 $(c_2, c_1 \times \mathrm{K} \times \mathrm{K})$，输入矩阵 $\boldsymbol{X}$ 形状为 $(c_1 \times \mathrm{K} \times \mathrm{K}, \mathrm{H}_{out} \times \mathrm{W}_{out})$。

于是 $\boldsymbol{y} = \boldsymbol{W}\boldsymbol{X} + \boldsymbol{b}$，$\boldsymbol{b}$ 的形状为 c_2。

我们回顾第六章介绍的 DFN 的反向传播算法，在第 l 个全连接层有：

$$\begin{aligned} \boldsymbol{z}_{l+1} &= \boldsymbol{W}_l \boldsymbol{a}_l + \boldsymbol{b}_l \\ \boldsymbol{a}_{l+1} &= g(\boldsymbol{z}_{l+1}) \end{aligned} \tag{7}$$

我们可以得到：

$$\begin{cases} \delta_l^{(\boldsymbol{z})} = (\boldsymbol{W}_l)^\top \delta_{l+1}^{(\boldsymbol{z})} \odot g'(\boldsymbol{z}_l) \\ \nabla_{\boldsymbol{W}_l} J = \delta_{l+1}^{(\boldsymbol{z})} (\boldsymbol{a}_l)^\top \\ \nabla_{\boldsymbol{b}_l} J = \delta_{l+1}^{(\boldsymbol{z})} \end{cases} \tag{8}$$

同样的，我们这里有：

$$\begin{cases} \delta^{(\boldsymbol{x})} = \boldsymbol{W}^\top \delta^{(\boldsymbol{a})} \odot g'(\boldsymbol{y}) \\ \nabla_{\boldsymbol{W}} J = \delta^{(\boldsymbol{y})} \boldsymbol{X}^\top \\ \nabla_{\boldsymbol{b}} J = \delta^{(\boldsymbol{y})} \end{cases} \tag{9}$$

[6]:
```python
def col2im(X_col, X_shape, W_shape, pad, stride, dilation=0):
    """
    col2im 实现，“col2im” 函数将 2D 矩阵变为 4D 图像

    参数说明:
    X_col: X 经过 im2col 后 (列) 的矩阵，形状为 (Q, Z)，具体形状见上文
    X_shape: 原始的输入数组形状，为 (n_samples, in_rows, in_cols, in_ch)，
             此时还未 0 填充 (padding)
    W_shape: 卷积核组形状，4-tuple 为 (kernel_rows, kernel_cols, in_ch, out_ch)
    pad: padding 数目，4-tuple
            在图片的左、右、上、下 (left, right, up, down) 0 填充
    stride: 卷积核的卷积步幅，int 型
    dilation: 扩张率，int 型，default=1

    输出说明:
    img: 输出结果，形状为 (n_samples, in_rows, in_cols, in_ch)
    """
    s, d = stride, dilation
    pr1, pr2, pc1, pc2 = pad
    fr, fc, n_in, n_out = W_shape
    n_samp, in_rows, in_cols, n_in = X_shape
    X_pad = np.zeros((n_samp, n_in, in_rows + pr1 + pr2, in_cols + pc1 + pc2))
    k, i, j = _im2col_indices((n_samp, n_in, in_rows, in_cols), fr, fc, pad, s, d)
    X_col_reshaped = X_col.reshape(n_in * fr * fc, -1, n_samp)
    X_col_reshaped = X_col_reshaped.transpose(2, 0, 1)
    np.add.at(X_pad, (slice(None), k, i, j), X_col_reshaped)
    pr2 = None if pr2 == 0 else -pr2
    return X_pad[:, :, pr1:pr2, pc1:pc2]
```

[7]:
```python
class Conv2D_gemm(LayerBase):

    def __init__(
        self,
        out_ch,
        kernel_shape,
        pad=0,
        stride=1,
        dilation=1,
        acti_fn=None,
        optimizer=None,
        init_w="glorot_uniform",
    ):
        """
        二维卷积
```

```
        参数说明:
        out_ch: 卷积核组的数目，int 型
        kernel_shape: 单个卷积核形状，2-tuple
        acti_fn: 激活函数，str 型
        pad: padding 数目，4-tuple, int, 或 'same', 'valid' 型
            在图片的左、右、上、下 (left, right, up, down) 0 填充
            若为 int，表示在左、右、上、下均填充数目为 pad 的 0，
            若为 same，表示填充后为相同 (same) 卷积，
            若为 valid，表示填充后为有效 (valid) 卷积
        stride: 卷积核的卷积步幅，int 型
        dilation: 扩张率，int 型，default=1
        init_w: 权重初始化方法，str 型
        optimizer: 优化方法，str 型
        """
        super().__init__(optimizer)
        self.pad = pad
        self.in_ch = None
        self.out_ch = out_ch
        self.stride = stride
        self.dilation = dilation
        self.kernel_shape = kernel_shape
        self.init_w = init_w
        self.init_weights = WeightInitializer(mode=init_w)
        self.acti_fn = ActivationInitializer(acti_fn)()
        self.parameters = {"W": None, "b": None}
        self.is_initialized = False

    def _init_params(self):
        fr, fc = self.kernel_shape
        W = self.init_weights((fr, fc, self.in_ch, self.out_ch))
        b = np.zeros((1, 1, 1, self.out_ch))
        self.params = {"W": W, "b": b}
        self.gradients = {"W": np.zeros_like(W), "b": np.zeros_like(b)}
        self.derived_variables = {"Y": []}
        self.is_initialized = True

    def forward(self, X, retain_derived=True):
        """
        卷积层的前向传播，原理见上文。

        参数说明:
        X: 输入数组，形状为 (n_samples, in_rows, in_cols, in_ch)
        retain_derived: 是否保留中间变量，以便反向传播时再次使用，bool 型

        输出说明:
        a: 卷积层输出，形状为 (n_samples, out_rows, out_cols, out_ch)
        """
        if not self.is_initialized:
            self.in_ch = X.shape[3]
            self._init_params()
        W = self.params["W"]
        b = self.params["b"]
        n_samp, in_rows, in_cols, in_ch = X.shape
        s, p, d = self.stride, self.pad, self.dilation
        # 卷积操作
        Y = conv2D_gemm(X, W, s, p, d) + b
        a = self.acti_fn(Y)
        if retain_derived:
            self.X.append(X)
```

```
            self.derived_variables["Y"].append(Y)
        return a

    def backward(self, dLda, retain_grads=True):
        """
        卷积层的反向传播，原理见上文。

        参数说明：
        dLda：关于损失的梯度，为 (n_samples, out_rows, out_cols, out_ch)
        retain_grads：是否计算中间变量的参数梯度，bool 型

        输出说明：
        dX：当前卷积层对输入关于损失的梯度，为 (n_samples, in_rows, in_cols, in_ch)
        """
        if not isinstance(dLda, list):
            dLda = [dLda]
        dX = []
        X = self.X
        Y = self.derived_variables["Y"]
        for da, x, y in zip(dLda, X, Y):
            dx, dw, db = self._bwd(da, x, y)
            dX.append(dx)
            if retain_grads:
                self.gradients["W"] += dw
                self.gradients["b"] += db
        return dX[0] if len(X) == 1 else dX

    def _bwd(self, dLda, X, Y):
        W = self.params["W"]
        d = self.dilation
        fr, fc, in_ch, out_ch = W.shape
        n_samp, out_rows, out_cols, out_ch = dLda.shape
        (fr, fc), s, p = self.kernel_shape, self.stride, self.pad
        dLdy = dLda * self.acti_fn.grad(Y)
        dLdy_col = dLdy.transpose(3, 1, 2, 0).reshape(out_ch, -1)
        W_col = W.transpose(3, 2, 0, 1).reshape(out_ch, -1).T
        X_col, p = im2col(X, W.shape, p, s, d)
        dW = (dLdy_col @ X_col.T).reshape(out_ch, in_ch, fr, fc).transpose(2, 3, 1, 0)
        db = dLdy_col.sum(axis=1).reshape(1, 1, 1, -1)
        dX_col = W_col @ dLdy_col
        dX = col2im(dX_col, X.shape, W.shape, p, s, d).transpose(0, 2, 3, 1)
        return dX, dW, db

    @property
    def hyperparams(self):
        return {
            "layer": "Conv2D",
            "pad": self.pad,
            "init_w": self.init_w,
            "in_ch": self.in_ch,
            "out_ch": self.out_ch,
            "stride": self.stride,
            "dilation": self.dilation,
            "acti_fn": str(self.acti_fn),
            "kernel_shape": self.kernel_shape,
            "optimizer": {
                "cache": self.optimizer.cache,
                "hyperparams": self.optimizer.hyperparams,
            },
```

```
        }
```

6.2.2 池化函数的导数及后向传播

池化函数是一个下采样函数，对于大小为 m 的池化区域，池化函数及其导数可以定义为：

- 最大池化：$g(x) = \max(x)$，导数为 $\frac{\partial g}{\partial x_i} = \begin{cases} 1 & \text{if } x_i = \max(x) \\ 0 & \text{otherwise} \end{cases}$
- 均值池化：$g(x) = \frac{\sum_{k=1}^{m} x_k}{m}$，导数为 $\frac{\partial g}{\partial x_i} = \frac{1}{m}$
- p 范数池化：$g(x) = \|x\|_p = (\sum_{k=1}^{m} |x_k|^p)^{1/p}$ ，导数为 $\frac{\partial g}{\partial x_i} = (\sum_{k=1}^{m} |x_k|^p)^{1/p-1} |x_i|^{p-1}$

下采样的后向传播过程为上采样，其示意图为：

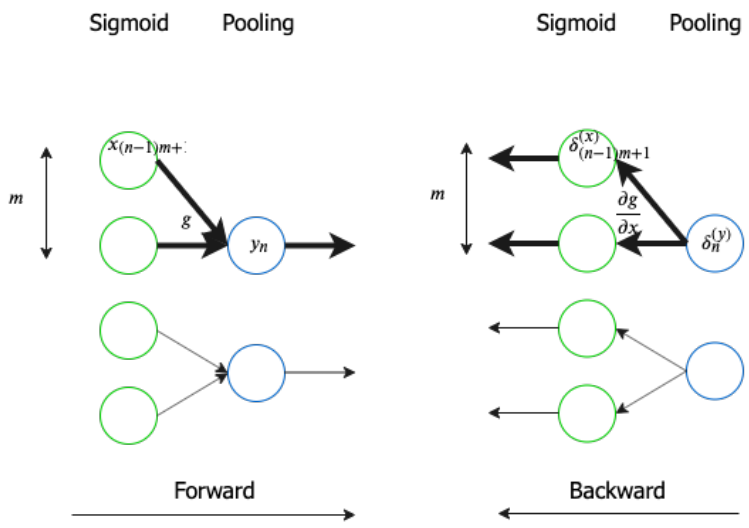

图 17. 池化函数的前向传播与反向传播示意图 (一维卷积)。

该后向传播过程就是利用 g 的导数将误差信号传递到 g 的输入。

$$\begin{aligned} \delta^{(\boldsymbol{x})}_{(n-1)m+1:nm} = \frac{\partial}{\partial x_{(n-1)m+1:nm}} J = \frac{\partial J}{\partial y_n} \frac{\partial y_n}{\partial x_{(n-1)m+1:nm}} = \delta^{(\boldsymbol{y})}_n g'_n \\ \delta^{(\boldsymbol{x})} = upsample(\delta^{(\boldsymbol{y})}, g') \end{aligned} \tag{10}$$

上采样 (`upsampling`) 具体展示如下，假设池化大小为 $2, 2$，步幅为 2，且 $\delta^{(\boldsymbol{y})}$ 第 k 个子矩阵为：

$$\delta^{(y)}_k = \begin{pmatrix} 2 & 4 \\ 6 & 8 \end{pmatrix} \tag{11}$$

先将 $\delta^{(\boldsymbol{y})}_k$ 还原成：

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 2 & 4 & 0 \\ 0 & 6 & 8 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{12}$$

如果是最大池化，假设之前在前向传播时记录的最大值位置分别是左上，右上，左下，右下，则转换后的矩阵为：

$$\begin{pmatrix} 2 & 0 & 0 & 4 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 6 & 0 & 0 & 8 \end{pmatrix} \tag{13}$$

如果是均值池化，根据公式可以得到：

$$\begin{pmatrix} 0.5 & 0.5 & 1.0 & 1.0 \\ 0.5 & 0.5 & 1.0 & 1.0 \\ 1.5 & 1.5 & 2.0 & 2.0 \\ 1.5. & 1.5 & 2.0 & 2.0 \end{pmatrix} \tag{14}$$

[8]:
```
class Pool2D(LayerBase):


    def __init__(self, kernel_shape, stride=1, pad=0, mode="max", optimizer=None):
        """
```

```
        二维池化

        参数说明:
        kernel_shape: 池化窗口的大小，2-tuple
        stride: 和卷积类似，窗口在每一个维度上滑动的步长，int 型
        pad: padding 数目，4-tuple, int, 或 str('same','valid') 型 (default: 0)
            和卷积类似
        mode: 池化函数，str 型 (default: 'max')，可选 {"max","average"}
        optimizer: 优化方法，str 型
        """
        super().__init__(optimizer)
        self.pad = pad
        self.mode = mode
        self.in_ch = None
        self.out_ch = None
        self.stride = stride
        self.kernel_shape = kernel_shape
        self.is_initialized = False

    def _init_params(self):
        self.derived_variables = {"out_rows": [], "out_cols": []}
        self.is_initialized = True

    def forward(self, X, retain_derived=True):
        """
        池化层前向传播

        参数说明:
        X: 输入数组，形状为 (n_samp, in_rows, in_cols, in_ch)
        retain_derived: 是否保留中间变量，以便反向传播时再次使用，bool 型

        输出说明:
        Y: 输出结果，形状为 (n_samp, out_rows, out_cols, out_ch)
        """
        if not self.is_initialized:
            self.in_ch = self.out_ch = X.shape[3]
            self._init_params()
        n_samp, in_rows, in_cols, nc_in = X.shape
        (fr, fc), s, p = self.kernel_shape, self.stride, self.pad
        X_pad, (pr1, pr2, pc1, pc2) = pad2D(X, p, self.kernel_shape, s)
        out_rows = int((in_rows + pr1 + pr2 - fr) / s + 1)
        out_cols = int((in_cols + pc1 + pc2 - fc) / s + 1)
        if self.mode == "max":
            pool_fn = np.max
        elif self.mode == "average":
            pool_fn = np.mean
        Y = np.zeros((n_samp, out_rows, out_cols, self.out_ch))
        for m in range(n_samp):
            for i in range(out_rows):
                for j in range(out_cols):
                    for c in range(self.out_ch):
                        i0, i1 = i * s, (i * s) + fr
                        j0, j1 = j * s, (j * s) + fc
                        xi = X_pad[m, i0:i1, j0:j1, c]
                        Y[m, i, j, c] = pool_fn(xi)
        if retain_derived:
            self.X.append(X)
            self.derived_variables["out_rows"].append(out_rows)
            self.derived_variables["out_cols"].append(out_cols)
```

```
        return Y

    def backward(self, dLdy, retain_grads=True):
        """
        池化层的反向传播，原理见上文。

        参数说明：
        dLdy：关于损失的梯度，为 (n_samples, out_rows, out_cols, out_ch)
        retain_grads：是否计算中间变量的参数梯度，bool 型

        输出说明：
        dXs：即 dX，当前卷积层对输入关于损失的梯度，为 (n_samples, in_rows, in_cols, in_ch)
        """
        if not isinstance(dLdy, list):
            dLdy = [dLdy]
        Xs = self.X
        out_rows = self.derived_variables["out_rows"]
        out_cols = self.derived_variables["out_cols"]
        (fr, fc), s, p = self.kernel_shape, self.stride, self.pad
        dXs = []
        for X, dy, out_row, out_col in zip(Xs, dLdy, out_rows, out_cols):
            n_samp, in_rows, in_cols, nc_in = X.shape
            X_pad, (pr1, pr2, pc1, pc2) = pad2D(X, p, self.kernel_shape, s)
            dX = np.zeros_like(X_pad)
            for m in range(n_samp):
                for i in range(out_row):
                    for j in range(out_col):
                        for c in range(self.out_ch):
                            i0, i1 = i * s, (i * s) + fr
                            j0, j1 = j * s, (j * s) + fc
                            if self.mode == "max":
                                xi = X[m, i0:i1, j0:j1, c]
                                mask = np.zeros_like(xi).astype(bool)
                                x, y = np.argwhere(xi == np.max(xi))[0]
                                mask[x, y] = True
                                dX[m, i0:i1, j0:j1, c] += mask * dy[m, i, j, c]
                            elif self.mode == "average":
                                frame = np.ones((fr, fc)) * dy[m, i, j, c]
                                dX[m, i0:i1, j0:j1, c] += frame / np.prod((fr, fc))
            pr2 = None if pr2 == 0 else -pr2
            pc2 = None if pc2 == 0 else -pc2
            dXs.append(dX[:, pr1:pr2, pc1:pc2, :])
        return dXs[0] if len(Xs) == 1 else dXs

    @property
    def hyperparams(self):
        return {
            "layer": "Pool2D",
            "acti_fn": None,
            "pad": self.pad,
            "mode": self.mode,
            "in_ch": self.in_ch,
            "out_ch": self.out_ch,
            "stride": self.stride,
            "kernel_shape": self.kernel_shape,
            "optimizer": {
                "cache": self.optimizer.cache,
                "hyperparams": self.optimizer.hyperparams,
            },
```

```
        }
```

6.3 多层卷积层/池化层

在第六章介绍了多层神经网络反向传播的推导过程。这里同前面所述，假设损失函数 $J(\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{x}, \boldsymbol{y})$，第 $l+1$ 层的输入和输出分别是 $\boldsymbol{a}_l$ 和 $\boldsymbol{a}_{l+1}$，参数为 $\boldsymbol{W}_l$ 和 $\boldsymbol{b}_l$，仿射结果为中间变量 $\boldsymbol{z}_{l+1} = \boldsymbol{W}_l \boldsymbol{a}_l + \boldsymbol{b}_l$。其中 $\boldsymbol{a}_{l+1} = g(\boldsymbol{z}_{l+1})$，$g$ 为激励函数，第一层的输出 $\boldsymbol{a}_1 = \boldsymbol{x}$，为整个网络的输入，最后一层的输出 $\boldsymbol{a}_L$ 是代价函数的输入。计算得到第 l 层 J 对 $\boldsymbol{z}_l$ 的偏导数和 J 对参数 $\boldsymbol{W}_l$ 和 $\boldsymbol{b}_l$ 的梯度。

$$\begin{cases} \delta_l^{(z)} = (\boldsymbol{W}_l)^\top \delta_{l+1}^{(z)} \odot g'(\boldsymbol{z}_l) \\ \nabla_{\boldsymbol{W}_l} J = \delta_{l+1}^{(z)} (\boldsymbol{a}_l)^\top \\ \nabla_{\boldsymbol{b}_l} J = \delta_{l+1}^{(z)} \end{cases} \tag{15}$$

同样的，可以得到在卷积网络反向传播中，假设在第 $l+1$ 层经过卷积操作 (包括激励函数) 或池化 (激励函数为池化函数)。

1. 对第 l 层计算得到 J 对 z_l 的偏导数；

   - 如果是卷积层：

   $$\delta_l^{(z)} = \delta_{l+1}^{(z)} * \text{flip}(\boldsymbol{W}_l) \odot g'(\boldsymbol{z}_l) \tag{16}$$

   - 如果是池化层：

   $$\delta_l^{(z)} = upsample(\delta_{l+1}^{(z)}, g') \tag{17}$$

2. 对第 l 层计算得到 J 对参数 $\boldsymbol{W}_l$ 和 $\boldsymbol{b}_l$ 的梯度；

$$\begin{aligned} \nabla_{\boldsymbol{W}_l} J &= \delta_{l+1}^{(z)} * \boldsymbol{a}_l \\ \nabla_{\boldsymbol{b}_l} J &= \delta_{l+1}^{(z)} \end{aligned} \tag{18}$$

6.4 Flatten 层 & 全连接层

通过多层卷积层/池化层后我们可以获得很多个特征图 (`Feature Map`)，这些特征图从不同角度得到模型的特征，但是我们无法直接用这些特征图拿来连接到分类。对于一个分类模型，我们既要综合考虑所有的特征图，又要能够连接到分类，可以考虑的方案就是将最后得到的特征图 “拍平” (丢到 Flatten 层)，然后把 Flatten 层的输出放到全连接层里 (一般至少 2 层以保证全连接层能够学到拟合函数)。最后采用 softmax 对其进行分类。

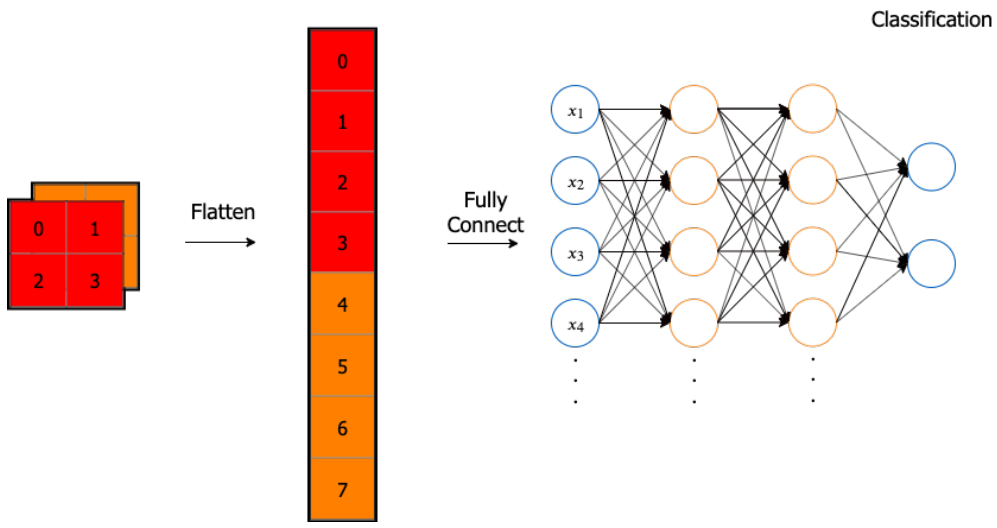

图 18. Flatten 层的实现过程：对最后得到的卷积核组依次展开，每个值作为一个神经元节点。

如果是将最后一层的特征图 Flatten 再放入全连接层，会存在的问题是输入的参数量过大，这会降低训练的速度，同时很容易过拟合。既然是针对这些特征图进行分类，那么另外一种可以考虑的方案是全局平均池化 (`Global Average Pooling`)。如图所示，GAP 的操作是将最后一层的特征图取均值。我们得到的 GAP 层中的每个节点对应不同的特征图，连接 GAP 层和最后的密集层的权重编码了每个特征图对预测目标分类的贡献。

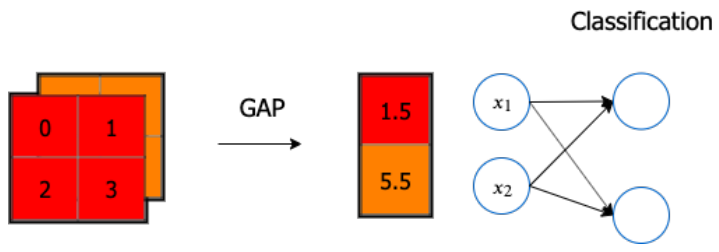

图 19. GAP 层的实现过程：对最后得到的卷积核组，每个卷积核取其均值作为一个神经元节点。

[9]:
```
class Flatten(LayerBase):


    def __init__(self, keep_dim="first", optimizer=None):
        """
```

```
        将多维输入展开

        参数说明:
        keep_dim: 展开形状，str (default : 'first')
                对于输入 X，keep_dim 可选 'first'-> 将 X 重构为 (X.shape[0], -1)，
                'last'-> 将 X 重构为 (-1, X.shape[0])，'none'-> 将 X 重构为 (1,-1)
        optimizer: 优化方法
        """
        super().__init__(optimizer)
        self.keep_dim = keep_dim
        self._init_params()

    def _init_params(self):
        self.X = []
        self.gradients = {}
        self.params = {}
        self.derived_variables = {"in_dims": []}

    def forward(self, X, retain_derived=True):
        """
        前向传播

        参数说明:
        X: 输入数组
        retain_derived: 是否保留中间变量，以便反向传播时再次使用，bool 型
        """
        if retain_derived:
            self.derived_variables["in_dims"].append(X.shape)
        if self.keep_dim == "none":
            return X.flatten().reshape(1, -1)
        rs = (X.shape[0], -1) if self.keep_dim == "first" else (-1, X.shape[-1])
        return X.reshape(*rs)

    def backward(self, dLdy, retain_grads=True):
        """
        反向传播

        参数说明:
        dLdy: 关于损失的梯度
        retain_grads: 是否计算中间变量的参数梯度，bool 型

        输出说明:
        dX: 将对输入的梯度进行重构为原始输入的形状
        """
        if not isinstance(dLdy, list):
            dLdy = [dLdy]
        in_dims = self.derived_variables["in_dims"]
        dX = [dy.reshape(*dims) for dy, dims in zip(dLdy, in_dims)]
        return dX[0] if len(dLdy) == 1 else dX

    @property
    def hyperparams(self):
        return {
            "layer": "Flatten",
            "keep_dim": self.keep_dim,
            "optimizer": {
                "cache": self.optimizer.cache,
                "hyperparams": self.optimizer.hyperparams,
            },
```

```
    }
```

7 平移等变

卷积神经网络具有平移不变性 (`Translation Invariance`) 或称平移等变。平移不变性意味着系统产生完全相同的响应 (输出)，不管它的输入是如何平移的。

- 卷积：图像经过平移，相应的特征图上的表达也是平移的。如图 20，输入图像的左下角有一个人脸，经过卷积后人脸的特征 (眼睛，鼻子) 也位于特征图的左下角。但假如人脸特征在图像的左上角，那么卷积后对应的特征也在特征图的左上角。无论目标出现在图像中的哪个位置，它都会检测到同样的这些特征，输出同样的响应，所以卷积具有平移不变性。

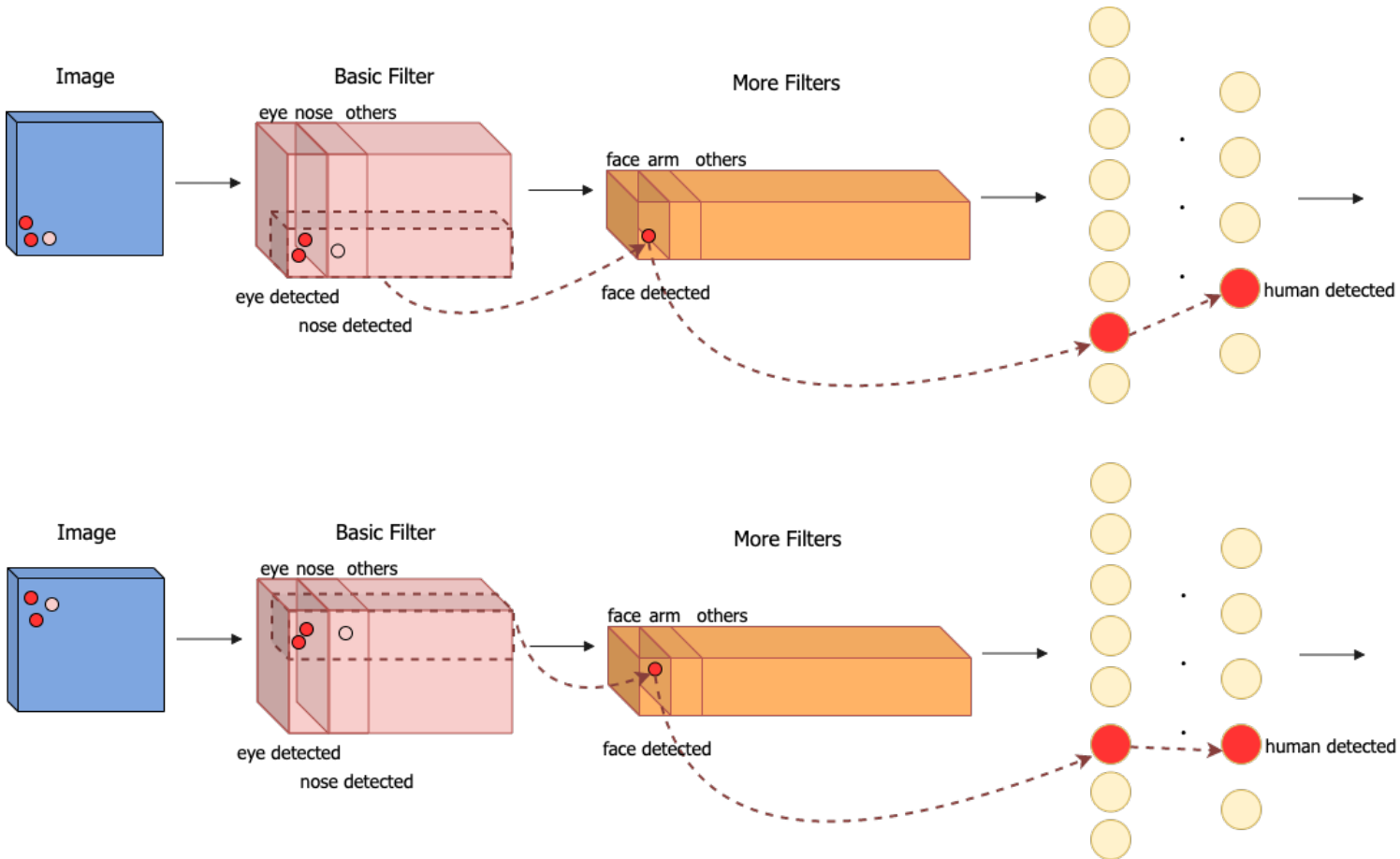

图 20. 卷积的平移不变性。

- 池化：如最大池化，可以返回感受野中的最大值，如果最大值被移动了，但仍然在这个感受野中，那么池化层也仍然会输出相同的最大值，所以卷积也可能具有平移不变性。

8 代表性的卷积神经网络

8.1 卷积神经网络 (LeNet)

LeNet [1] 分为卷积层块和全连接层块两个部分。卷积层块里的基本单位是卷积层后接最大池化层，卷积层块由两个这样的基本单位重复堆叠构成。

在卷积层块中，每个卷积层都使用 5×5 的窗口，并在输出上使用 sigmoid 激活函数。第一个卷积层输出通道数为 6，第二个卷积层输出通道数则增加到 16。这是因为卷积层未使用 0 填充，第二个卷积层比第一个卷积层的输入的高和宽要小，所以增加输出通道使两个卷积层的参数尺寸类似。卷积层块的两个最大池化层的窗口形状均为 2×2，且步幅为 2。由于池化窗口与步幅形状相同，池化窗口在输入上每次滑动所覆盖的区域互不重叠，其池化效果为池化函数的导数及后向传播部分中示例所示。

卷积层块的输出形状为 (批量大小, 通道, 高, 宽)。当卷积层块的输出传入全连接层块时，全连接层块会将小批量中每个样本变平 (Flatten)。即将全连接层的输入形状将变成二维，其中第一维是小批量中的样本，第二维是每个样本变平后的向量表示，且向量长度为通道、高和宽的乘积。全连接层块含 3 个全连接层，它们的输出个数分别是 120、84 和 10，其中 10 为输出的类别个数，如图所示。

我们接下来代码演示的是 LeNet 的实现，但与论文 [1] 中的实现稍有不同：论文 [1] 中的 LeNet 网络在池化层之后再进行非线性处理 (即 sigmoid 激活函数)，现在通用的操作是经过卷积之后就经过非线性处理 (sigmoid 激活函数)，然后再进行池化操作。

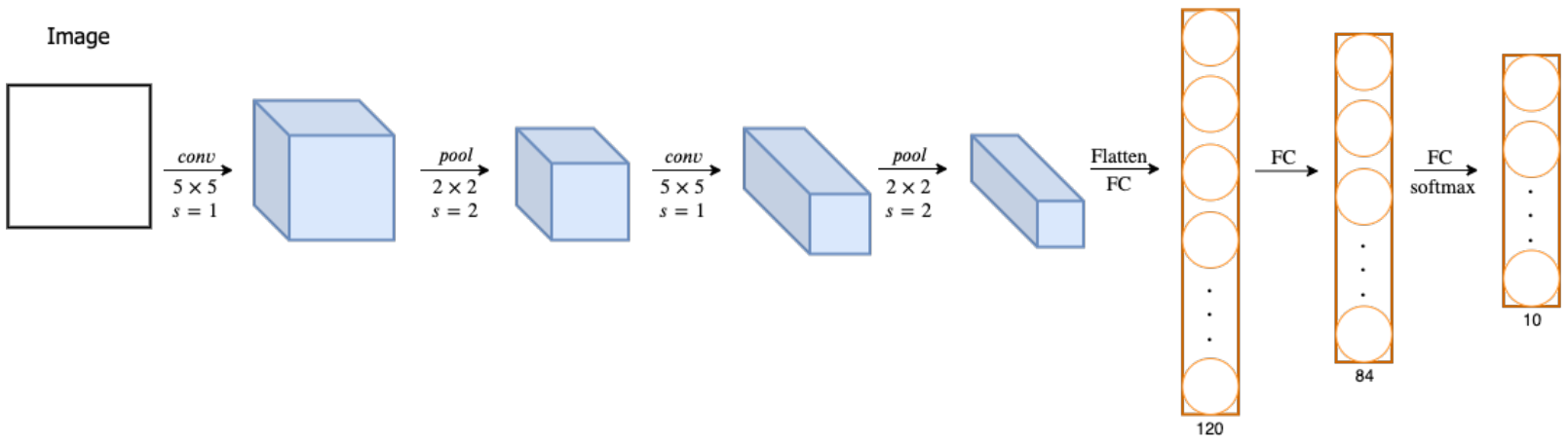

图 21. LeNet 网络实现框架。

[1]: Lecun, Y. , Bottou, L. , Bengio, Y. , & Haffner, P. . (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278-2324.

[10]:
```
class LeNet(object):

    def __init__(
        self,
        fc3_out=128,
        fc4_out=84,
        fc5_out=10,
        conv1_pad=0,
        conv2_pad=0,
        conv1_out_ch=6,
        conv2_out_ch=16,
        conv1_stride=1,
        pool1_stride=2,
        conv2_stride=1,
        pool2_stride=2,
        conv1_kernel_shape=(5, 5),
        pool1_kernel_shape=(2, 2),
        conv2_kernel_shape=(5, 5),
        pool2_kernel_shape=(2, 2),
        optimizer="adam",
        init_w="glorot_normal",
        loss=CrossEntropy()
    ):
        self.optimizer = optimizer
        self.init_w = init_w
        self.loss = loss
        self.fc3_out = fc3_out
        self.fc4_out = fc4_out
        self.fc5_out = fc5_out
        self.conv1_pad = conv1_pad
        self.conv2_pad = conv2_pad
        self.conv1_stride = conv1_stride
        self.conv1_out_ch = conv1_out_ch
        self.pool1_stride = pool1_stride
        self.conv2_out_ch = conv2_out_ch
        self.conv2_stride = conv2_stride
        self.pool2_stride = pool2_stride
        self.conv2_kernel_shape = conv2_kernel_shape
        self.pool2_kernel_shape = pool2_kernel_shape
        self.conv1_kernel_shape = conv1_kernel_shape
        self.pool1_kernel_shape = pool1_kernel_shape
        self.is_initialized = False

    def _set_params(self):
```

```
        """
        函数作用：模型初始化
        Conv1 -> Pool1 -> Conv2 -> Pool2 -> Flatten -> FC3 -> FC4 -> FC5 -> Softmax
        """
        self.layers = OrderedDict()
        self.layers["Conv1"] = Conv2D(
            out_ch=self.conv1_out_ch,
            kernel_shape=self.conv1_kernel_shape,
            pad=self.conv1_pad,
            stride=self.conv1_stride,
            acti_fn="sigmoid",
            optimizer=self.optimizer,
            init_w=self.init_w,
        )
        self.layers["Pool1"] = Pool2D(
            mode="max",
            optimizer=self.optimizer,
            stride=self.pool1_stride,
            kernel_shape=self.pool1_kernel_shape,
        )
        self.layers["Conv2"] = Conv2D(
            out_ch=self.conv1_out_ch,
            kernel_shape=self.conv1_kernel_shape,
            pad=self.conv1_pad,
            stride=self.conv1_stride,
            acti_fn="sigmoid",
            optimizer=self.optimizer,
            init_w=self.init_w,
        )
        self.layers["Pool2"] = Pool2D(
            mode="max",
            optimizer=self.optimizer,
            stride=self.pool2_stride,
            kernel_shape=self.pool2_kernel_shape,
        )
        self.layers["Flatten"] = Flatten(optimizer=self.optimizer)
        self.layers["FC3"] = FullyConnected(
            n_out=self.fc3_out,
            acti_fn="sigmoid",
            init_w=self.init_w,
            optimizer=self.optimizer
        )
        self.layers["FC4"] = FullyConnected(
            n_out=self.fc4_out,
            acti_fn="sigmoid",
            init_w=self.init_w,
            optimizer=self.optimizer
        )
        self.layers["FC5"] = FullyConnected(
            n_out=self.fc5_out,
            acti_fn="affine(slope=1, intercept=0)",
            init_w=self.init_w,
            optimizer=self.optimizer
        )
        self.is_initialized = True

    def forward(self, X_train):
        Xs = {}
        out = X_train
```

```
        for k, v in self.layers.items():
            Xs[k] = out
            out = v.forward(out)
        return out, Xs

    def backward(self, grad):
        dXs = {}
        out = grad
        for k, v in reversed(list(self.layers.items())):
            dXs[k] = out
            out = v.backward(out)
        return out, dXs

    def update(self):
        """
        函数作用：梯度更新
        """
        for k, v in reversed(list(self.layers.items())):
            v.update()
        self.flush_gradients()

    def flush_gradients(self, curr_loss=None):
        """
        函数作用：更新后重置梯度
        """
        for k, v in self.layers.items():
            v.flush_gradients()

    def fit(self, X_train, y_train, n_epochs=20, batch_size=64, verbose=False, epo_verbose=True):
        """
        参数说明：
        X_train：训练数据
        y_train：训练数据标签
        n_epochs：epoch 次数
        batch_size：每次 epoch 的 batch size
        verbose：是否每个 batch 输出损失
        epo_verbose：是否每个 epoch 输出损失
        """
        self.verbose = verbose
        self.n_epochs = n_epochs
        self.batch_size = batch_size
        if not self.is_initialized:
            self.n_features = X_train.shape[1]
            self._set_params()
        prev_loss = np.inf
        for i in range(n_epochs):
            loss, epoch_start = 0.0, time.time()
            batch_generator, n_batch = minibatch(X_train, self.batch_size, shuffle=True)
            for j, batch_idx in enumerate(batch_generator):
                batch_len, batch_start = len(batch_idx), time.time()
                X_batch, y_batch = X_train[batch_idx], y_train[batch_idx]
                out, _ = self.forward(X_batch)
                y_pred_batch = softmax(out)
                batch_loss = self.loss(y_batch, y_pred_batch)
                grad = self.loss.grad(y_batch, y_pred_batch)
                _, _ = self.backward(grad)
                self.update()
                loss += batch_loss
                if self.verbose:
```

```
                fstr = "\t[Batch {}/{}] Train loss: {:.3f} ({:.1f}s/batch)"
                print(fstr.format(j + 1, n_batch, batch_loss, time.time() - batch_start))
        loss /= n_batch
        if epo_verbose:
            fstr = "[Epoch {}] Avg. loss: {:.3f}  Delta: {:.3f} ({:.2f}m/epoch)"
            print(fstr.format(i + 1, loss, prev_loss - loss, (time.time() - epoch_start) / 60.0))
        prev_loss = loss

def evaluate(self, X_test, y_test, batch_size=128):
    acc = 0.0
    batch_generator, n_batch = minibatch(X_test, batch_size, shuffle=True)
    for j, batch_idx in enumerate(batch_generator):
        batch_len, batch_start = len(batch_idx), time.time()
        X_batch, y_batch = X_test[batch_idx], y_test[batch_idx]
        y_pred_batch, _ = self.forward(X_batch)
        y_pred_batch = np.argmax(y_pred_batch, axis=1)
        y_batch = np.argmax(y_batch, axis=1)
        acc += np.sum(y_pred_batch == y_batch)
    return acc / X_test.shape[0]

@property
def hyperparams(self):
    return {
        "init_w": self.init_w,
        "loss": str(self.loss),
        "optimizer": self.optimizer,
        "fc3_out": self.fc3_out,
        "fc4_out": self.fc4_out,
        "fc5_out": self.fc5_out,
        "conv1_pad": self.conv1_pad,
        "conv2_pad": self.conv2_pad,
        "conv1_stride": self.conv1_stride,
        "conv1_out_ch": self.conv1_out_ch,
        "pool1_stride": self.pool1_stride,
        "conv2_out_ch": self.conv2_out_ch,
        "conv2_stride": self.conv2_stride,
        "pool2_stride": self.pool2_stride,
        "conv2_kernel_shape": self.conv2_kernel_shape,
        "pool2_kernel_shape": self.pool2_kernel_shape,
        "conv1_kernel_shape": self.conv1_kernel_shape,
        "pool1_kernel_shape": self.pool1_kernel_shape,
        "components": {k: v.params for k, v in self.layers.items()}
    }
```

用 LeNet，MNIST 数据集测试

[11]:
```
"""
载入数据，取 10000 条数据用以训练，测试集不变
"""
def load_data(path="../data/mnist/mnist.npz"):
    f = np.load(path)
    X_train, y_train = f['x_train'], f['y_train']
    X_test, y_test = f['x_test'], f['y_test']
    f.close()
    return (X_train, y_train), (X_test, y_test)

(X_train, y_train), (X_test, y_test) = load_data()
y_train = np.eye(10)[y_train.astype(int)]
y_test = np.eye(10)[y_test.astype(int)]
X_train = X_train.reshape(-1, X_train.shape[1], X_train.shape[2], 1).astype('float32')
```

```
X_test = X_test.reshape(-1, X_train.shape[1], X_train.shape[2], 1).astype('float32')
print(X_train.shape, y_train.shape)
N = 10000   # 取 10000 条数据用以训练，测试集不变
indices = np.random.permutation(range(X_train.shape[0]))[:N]
X_train, y_train = X_train[indices], y_train[indices]
print(X_train.shape, y_train.shape)
X_train /= 255
X_train = (X_train - 0.5) * 2
X_test /= 255
X_test = (X_test - 0.5) * 2
```

```
(60000, 28, 28, 1) (60000, 10)
(10000, 28, 28, 1) (10000, 10)
```

[12]:
```
model = LeNet()
model.fit(X_train, y_train, n_epochs=15, batch_size=64, epo_verbose=True)
print("Test Accuracy:{}".format(model.evaluate(X_test, y_test)))
```

```
[Epoch 1] Avg. loss: 2.265  Delta: inf (17.67m/epoch)
[Epoch 2] Avg. loss: 1.369  Delta: 0.896 (17.33m/epoch)
[Epoch 3] Avg. loss: 0.703  Delta: 0.666 (17.33m/epoch)
[Epoch 4] Avg. loss: 0.489  Delta: 0.214 (17.35m/epoch)
[Epoch 5] Avg. loss: 0.389  Delta: 0.100 (17.41m/epoch)
[Epoch 6] Avg. loss: 0.330  Delta: 0.059 (17.31m/epoch)
[Epoch 7] Avg. loss: 0.287  Delta: 0.043 (17.39m/epoch)
[Epoch 8] Avg. loss: 0.255  Delta: 0.032 (17.31m/epoch)
[Epoch 9] Avg. loss: 0.230  Delta: 0.025 (17.30m/epoch)
[Epoch 10] Avg. loss: 0.209  Delta: 0.022 (17.38m/epoch)
[Epoch 11] Avg. loss: 0.192  Delta: 0.017 (17.31m/epoch)
[Epoch 12] Avg. loss: 0.178  Delta: 0.014 (17.31m/epoch)
[Epoch 13] Avg. loss: 0.163  Delta: 0.015 (17.32m/epoch)
[Epoch 14] Avg. loss: 0.155  Delta: 0.008 (17.32m/epoch)
[Epoch 15] Avg. loss: 0.143  Delta: 0.012 (17.32m/epoch)
Test Accuracy:0.959
```

以下我们再用 **GEMM** 转换的卷积计算实现 **LeNet**，比较两者的时间差值。

[13]:
```
class LeNet_gemm(object):

    def __init__(
        self,
        fc3_out=128,
        fc4_out=84,
        fc5_out=10,
        conv1_pad=0,
        conv2_pad=0,
        conv1_out_ch=6,
        conv2_out_ch=16,
        conv1_stride=1,
        pool1_stride=2,
        conv2_stride=1,
        pool2_stride=2,
        conv1_kernel_shape=(5, 5),
        pool1_kernel_shape=(2, 2),
        conv2_kernel_shape=(5, 5),
        pool2_kernel_shape=(2, 2),
        optimizer="adam",
        init_w="glorot_normal",
        loss=CrossEntropy()
    ):
        self.optimizer = optimizer
```

```
        self.init_w = init_w
        self.loss = loss
        self.fc3_out = fc3_out
        self.fc4_out = fc4_out
        self.fc5_out = fc5_out
        self.conv1_pad = conv1_pad
        self.conv2_pad = conv2_pad
        self.conv1_stride = conv1_stride
        self.conv1_out_ch = conv1_out_ch
        self.pool1_stride = pool1_stride
        self.conv2_out_ch = conv2_out_ch
        self.conv2_stride = conv2_stride
        self.pool2_stride = pool2_stride
        self.conv2_kernel_shape = conv2_kernel_shape
        self.pool2_kernel_shape = pool2_kernel_shape
        self.conv1_kernel_shape = conv1_kernel_shape
        self.pool1_kernel_shape = pool1_kernel_shape
        self.is_initialized = False

    def _set_params(self):
        """
        函数作用: 模型初始化
        Conv1 -> Pool1 -> Conv2 -> Pool2 -> Flatten -> FC3 -> FC4 -> FC5 -> Softmax
        """
        self.layers = OrderedDict()
        self.layers["Conv1"] = Conv2D_gemm(
            out_ch=self.conv1_out_ch,
            kernel_shape=self.conv1_kernel_shape,
            pad=self.conv1_pad,
            stride=self.conv1_stride,
            acti_fn="sigmoid",
            optimizer=self.optimizer,
            init_w=self.init_w,
        )
        self.layers["Pool1"] = Pool2D(
            mode="max",
            optimizer=self.optimizer,
            stride=self.pool1_stride,
            kernel_shape=self.pool1_kernel_shape,
        )
        self.layers["Conv2"] = Conv2D_gemm(
            out_ch=self.conv1_out_ch,
            kernel_shape=self.conv1_kernel_shape,
            pad=self.conv1_pad,
            stride=self.conv1_stride,
            acti_fn="sigmoid",
            optimizer=self.optimizer,
            init_w=self.init_w,
        )
        self.layers["Pool2"] = Pool2D(
            mode="max",
            optimizer=self.optimizer,
            stride=self.pool2_stride,
            kernel_shape=self.pool2_kernel_shape,
        )
        self.layers["Flatten"] = Flatten(optimizer=self.optimizer)
        self.layers["FC3"] = FullyConnected(
            n_out=self.fc3_out,
            acti_fn="sigmoid",
```

```
            init_w=self.init_w,
            optimizer=self.optimizer
        )
        self.layers["FC4"] = FullyConnected(
            n_out=self.fc4_out,
            acti_fn="sigmoid",
            init_w=self.init_w,
            optimizer=self.optimizer
        )
        self.layers["FC5"] = FullyConnected(
            n_out=self.fc5_out,
            acti_fn="affine(slope=1, intercept=0)",
            init_w=self.init_w,
            optimizer=self.optimizer
        )
        self.is_initialized = True

    def forward(self, X_train):
        Xs = {}
        out = X_train
        for k, v in self.layers.items():
            Xs[k] = out
            out = v.forward(out)
        return out, Xs

    def backward(self, grad):
        dXs = {}
        out = grad
        for k, v in reversed(list(self.layers.items())):
            dXs[k] = out
            out = v.backward(out)
        return out, dXs

    def update(self):
        """
        函数作用：梯度更新
        """
        for k, v in reversed(list(self.layers.items())):
            v.update()
        self.flush_gradients()

    def flush_gradients(self, curr_loss=None):
        """
        函数作用：更新后重置梯度
        """
        for k, v in self.layers.items():
            v.flush_gradients()

    def fit(self, X_train, y_train, n_epochs=20, batch_size=64, verbose=False, epo_verbose=True):
        """
        参数说明：
        X_train：训练数据
        y_train：训练数据标签
        n_epochs：epoch 次数
        batch_size：每次 epoch 的 batch size
        verbose：是否每个 batch 输出损失
        epo_verbose：是否每个 epoch 输出损失
        """
        self.verbose = verbose
```

```
        self.n_epochs = n_epochs
        self.batch_size = batch_size
        if not self.is_initialized:
            self.n_features = X_train.shape[1]
            self._set_params()
        prev_loss = np.inf
        for i in range(n_epochs):
            loss, epoch_start = 0.0, time.time()
            batch_generator, n_batch = minibatch(X_train, self.batch_size, shuffle=True)
            for j, batch_idx in enumerate(batch_generator):
                batch_len, batch_start = len(batch_idx), time.time()
                X_batch, y_batch = X_train[batch_idx], y_train[batch_idx]
                out, _ = self.forward(X_batch)
                y_pred_batch = softmax(out)
                batch_loss = self.loss(y_batch, y_pred_batch)
                grad = self.loss.grad(y_batch, y_pred_batch)
                _, _ = self.backward(grad)
                self.update()
                loss += batch_loss
                if self.verbose:
                    fstr = "\t[Batch {}/{}] Train loss: {:.3f} ({:.1f}s/batch)"
                    print(fstr.format(j + 1, n_batch, batch_loss, time.time() - batch_start))
            loss /= n_batch
            if epo_verbose:
                fstr = "[Epoch {}] Avg. loss: {:.3f}  Delta: {:.3f} ({:.2f}m/epoch)"
                print(fstr.format(i + 1, loss, prev_loss - loss, (time.time() - epoch_start) / 60.0))
            prev_loss = loss

    def evaluate(self, X_test, y_test, batch_size=128):
        acc = 0.0
        batch_generator, n_batch = minibatch(X_test, batch_size, shuffle=True)
        for j, batch_idx in enumerate(batch_generator):
            batch_len, batch_start = len(batch_idx), time.time()
            X_batch, y_batch = X_test[batch_idx], y_test[batch_idx]
            y_pred_batch, _ = self.forward(X_batch)
            y_pred_batch = np.argmax(y_pred_batch, axis=1)
            y_batch = np.argmax(y_batch, axis=1)
            acc += np.sum(y_pred_batch == y_batch)
        return acc / X_test.shape[0]

    @property
    def hyperparams(self):
        return {
            "init_w": self.init_w,
            "loss": str(self.loss),
            "optimizer": self.optimizer,
            "fc3_out": self.fc3_out,
            "fc4_out": self.fc4_out,
            "fc5_out": self.fc5_out,
            "conv1_pad": self.conv1_pad,
            "conv2_pad": self.conv2_pad,
            "conv1_stride": self.conv1_stride,
            "conv1_out_ch": self.conv1_out_ch,
            "pool1_stride": self.pool1_stride,
            "conv2_out_ch": self.conv2_out_ch,
            "conv2_stride": self.conv2_stride,
            "pool2_stride": self.pool2_stride,
            "conv2_kernel_shape": self.conv2_kernel_shape,
            "pool2_kernel_shape": self.pool2_kernel_shape,
```

```
            "conv1_kernel_shape": self.conv1_kernel_shape,
            "pool1_kernel_shape": self.pool1_kernel_shape,
            "components": {k: v.params for k, v in self.layers.items()}
        }
```

[14]:
```
model = LeNet_gemm()
model.fit(X_train, y_train, n_epochs=15, batch_size=64, epo_verbose=True)
print("Test Accuracy:{}".format(model.evaluate(X_test, y_test)))
```

```
[Epoch 1] Avg. loss: 2.297  Delta: inf (5.20m/epoch)
[Epoch 2] Avg. loss: 1.569  Delta: 0.728 (5.24m/epoch)
[Epoch 3] Avg. loss: 0.751  Delta: 0.818 (5.62m/epoch)
[Epoch 4] Avg. loss: 0.484  Delta: 0.267 (5.52m/epoch)
[Epoch 5] Avg. loss: 0.370  Delta: 0.114 (5.61m/epoch)
[Epoch 6] Avg. loss: 0.310  Delta: 0.059 (5.82m/epoch)
[Epoch 7] Avg. loss: 0.271  Delta: 0.040 (5.62m/epoch)
[Epoch 8] Avg. loss: 0.241  Delta: 0.030 (5.28m/epoch)
[Epoch 9] Avg. loss: 0.217  Delta: 0.024 (5.28m/epoch)
[Epoch 10] Avg. loss: 0.201  Delta: 0.017 (5.28m/epoch)
[Epoch 11] Avg. loss: 0.183  Delta: 0.017 (5.30m/epoch)
[Epoch 12] Avg. loss: 0.170  Delta: 0.013 (5.27m/epoch)
[Epoch 13] Avg. loss: 0.157  Delta: 0.013 (5.25m/epoch)
[Epoch 14] Avg. loss: 0.147  Delta: 0.010 (5.26m/epoch)
[Epoch 15] Avg. loss: 0.138  Delta: 0.009 (5.24m/epoch)
Test Accuracy:0.9567
```

更多的卷积神经网络及其应用将在第十二章呈现

[15]:
```
import numpy


print("numpy:", numpy.__version__)
```

```
numpy: 1.14.5
```