

实验三 用 Python 编程构建 CNN 实现手写体数字识别

【实验目的】

1. 理解全连接神经网络的缺陷和卷积神经网络的基本原理。
2. 掌握搭建卷积神经网络流程。
3. 掌握全连接层数据的正向传播以及误差反向传播算法的 Python 编程实现。
4. 熟悉卷积层和池化层的数据正向传播和误差反向传播算法的 Python 编程实现。
5. 了解搭建卷积神经网络的常用函数。
6. 熟悉 MNIST 数据集的加载方法以及将二进制编码的手写体标签向独热 (onehot) 类型转换的方法。

【实验器材】

硬件：实验用 PC 机一台

环境：人工智能实验平台

【实验原理】

一、卷积神经网络结构

在搭建卷积神经网络之前，首先要了解卷积神经网络的结构是怎么样的，其和之前的多层感知机的区别是怎么样的，具体是怎样进行计算的。常见的卷积神经网络的表现形式如下：

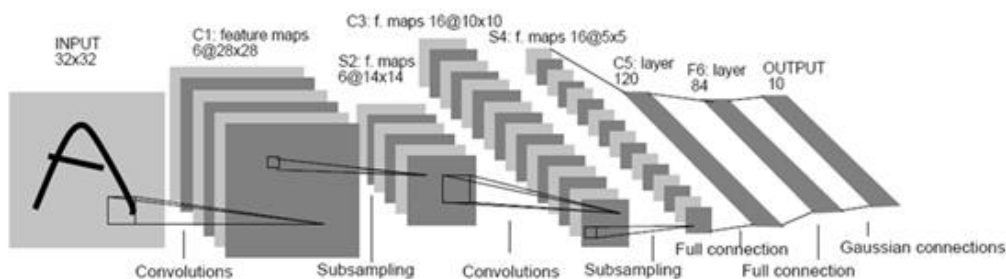


图 1 卷积神经网络结构示意图

(1) 结构图

由于卷积神经网络主要用于处理图像数据，所以卷积神经网络的结构图中，常以矩阵的形式代表二维排列的神经元。因次，虽然图中没画出神经元结构，但图中每个矩形结构中，都是二维排列的神经元。例如输入图像，其大小为 28×28 ，则它里面是二维排列的 $28 \times 28 = 784$ 个神经元，图中其它矩形结构以此类推。

(2) 卷积神经网络结构特点

图 1 是一个经典简单的卷积神经网络结构图，其中“Convolutions”是卷积操作；“feature map”是卷积后生成的隐藏层结果，叫做特征图；“Subsampling”是下采样操作，也就是现在常说的“pooling”池化操作；“Full connection”是全连接操作。由图 1 可以看出，输入图像进

入该卷积神经网络之后，首先进行了卷积操作，再进行了池化操作，接着再进行了一次卷积和池化操作，接着是两次全连接操作，最终输出结果。

卷积神经网络的经典结构公式可以为“ $N(\text{卷积}+\text{池化})+N+\text{全连接层}$ ”。

二、全连接神经网络的缺陷

多层感知机结构是人工神经网络早期使用比较多的一种网络。它的特点是，除输入层外，每个神经元与其上一层的每个神经元之间都有连接，且不存在跨层连接或反向连接。从神经元的连接方式来看，这种结构被称为“全连接结构”，全部由全连接结构组成的神经网络被称作“全连接神经网络”。

全连接网络常见的形式如下：

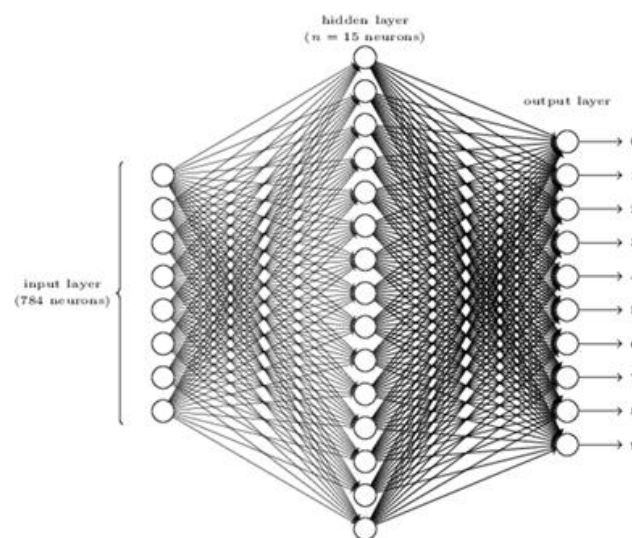


图 2 全连接神经网络结构示意图

全连接的一个特征是连接较密集，一个连接需要需要一个权值参数，因此全连接结构的参数量会比较大。参数量大会造成模型对硬件的计算能力有一定要求，此外也会导致过拟合问题。

在卷积神经网络中，全连接结构主要用于网络的最后几层，用于对提取的特征进行分类。

三、全连接参数过多的缓解

图 3 是一个全连接结构的示意图，左边是输入图像，大小为 1000×1000 ，所以其输入神经元个数为 10^6 。第一层隐藏层神经元也是 10^6 个，那么总的来说，采用全连接之后，两层的权值参数共有 10^6 乘以 10^6 ，也就是 10^{12} 次方。

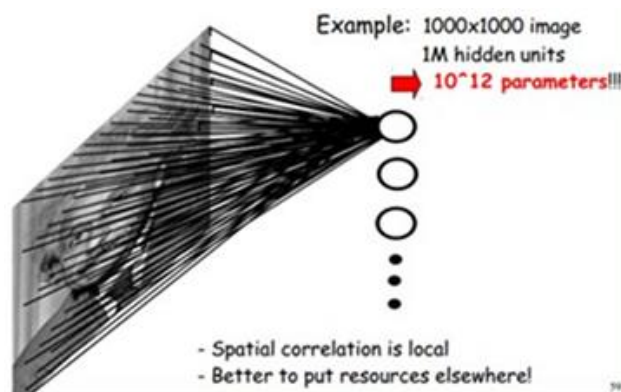


图 3 全连接结构

为了减少连接的参数量，卷积在此基础上做了两个调整：1) 不再使用全连接的结构，改用局部连接的方式。也就是说隐藏层的某个神经元不再与上一层所有神经元相连，仅与一个窗口大小的神经元相连。如下图所示，这里隐藏层的神经元只与输入层的一个 10×10 的窗口大小的神经元相连。这样参数的总数就变成 10 的 2 次方乘以 10 的 6 次方，也就是 10 的 8 次方。参数量比原来的全连接下降了 4 个数量级，但还是比较大。

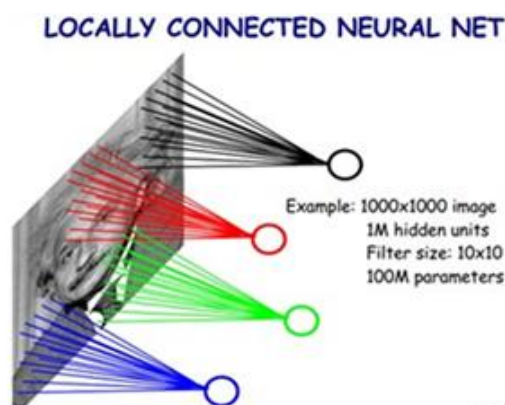


图 4 局部连接结构

若是我们将所有隐藏层的神经元共享一套权值，这样的话，10 的 6 次方个神经元用的都是一套权值，权值的数量就变成看 10×10 ，也就是 10 的 2 次方个。此时隐藏层中的参数量已经下降到一个非常可观的程度，对比原来的全连接结构，采用了“局部感知”和“权值共享”后，参数整体下降了 10 个数量级。

四、单通道卷积结构

在全连接基础上采用的“局部感知”和“权值共享”组成的操作，就是“卷积”操作。“权值共享”共享的参数，就是卷积核，“局部感知”的窗口大小，就是卷积核的尺寸。最终卷积操作呈现的效果，类似于有一张原图，一个卷积核窗口从图上一次扫过，每扫过一部分，就扫过的图片的局部做加权求和的操作。

如下图所示， 5×5 大小的绿色矩阵的是原始图像，其中 3×3 大小的黄色小方框是卷积核每次扫过的区域，黄色方框右下角对应的“ $\times 0$ ”，“ $\times 1$ ”是卷积核中的参数，右侧的红色图像为每次卷积核扫过后生成的结果。

以步长为 1 的大小扫过输入图像，则卷积核会由上到下，由左到右每次移动一个单位，每扫过一个区域，就会得到一个输出，扫完整整图像之后，生成一个 3*3 的特征图。

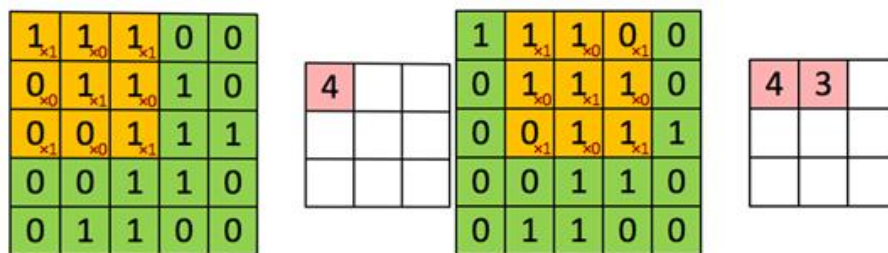


图 5 第一、二窗口卷积计算示意图

以第一个窗口为例，局部输入为 3*3 黄色窗口，输出为 1 个神经元，权值黄色窗口中的红色数字，加权求和的计算过程如下为：

$$1*1 + 1*0 + 1*1 + 0*0 + 1*1 + 1*0 + 0*1 + 0*0 + 1*1 = 4$$

同理，第二个窗口计算如下：

$$1*1 + 1*0 + 0*1 + 1*0 + 1*1 + 1*0 + 0*1 + 1*0 + 1*1 = 3$$

最终卷积结果如下：

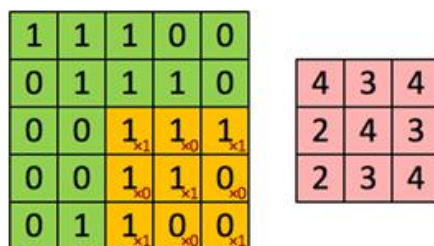


图 6 卷积结果示意图

五、 其他形式的卷积

上述图 6 所示的卷积操作形式存在两个问题：

A、图像越来越小；

B、图像边界信息丢失，即有些图像角落和边界的信息发挥作用较少。

因此需要 padding，也就是填充。填充常见的形式有：零填充，常数填充，镜像填充，重复填充。

根据填充多少不同，有三种卷积形式，valid padding、same padding 和 full padding。

(1) valid padding(有效填充)：

就是输入图像周围没有填充。卷积核从输入图像左上角开始以一定步长依次划过。生成的特征图一般比输入图像小。这种完全不使用填充的方式，我们也叫它“窄卷积”(narrow convolution)。

(2) same padding(相同填充)或者叫 half padding (半填充)

通过填充来保持输出和输入的特征图尺寸相同。相同填充下特征图的尺寸不会缩减但输入像素中靠近边界的部分相比于中间部分对于特征图的影响更小，即存在边界像素的欠表达。

使用相同填充的卷积被称为“等长卷积 (equal-width convolution)”。

(3) 第三种 full padding (全填充)

填充后使得每个像素在每个方向上被访问的次数相同。步长为 1 时，全填充输出的特征图尺寸为 $L+f-1$ ，大于输入值。使用全填充的卷积被称为“宽卷积 (wide convolution)”。

经典的卷积神经网络中见到的卷积都是采用 valid 和 same padding 的形式。而在“语义分割模型”和 GAN 的“生成模型”中，特征图还原成较大的尺寸，所以常会用到 full padding。

六、多通道卷积

在图 1 中，图像输入网络结构后，通道数由原来的单通道变成了 6 个，再变为后面的 16 个，就是通过多通道卷积进行变换的。多通道的计算方式如下：

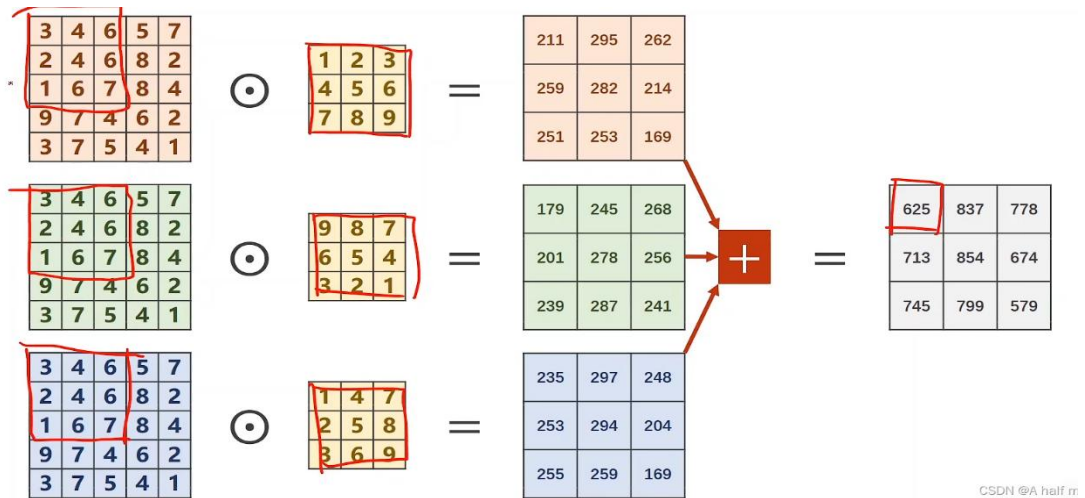


图 7 多通道卷积示意图

多通道图片如 RGB 图像的三通道，卷积核的通道数要和输入图像的通道数相同。多通道卷积过程：将每组卷积核应用到前一层的输入通道上生成一个输出通道，然后将这些通道中相同位置的像素值相加形成单个输出通道。如上图所示，输入层是一个 $5 \times 5 \times 3$ 矩阵，有 3 个通道。滤波器是 $3 \times 3 \times 3$ 矩阵，最后生成一张特征图。

若有多组卷积核的多通道卷积的计算如图 8 所示：

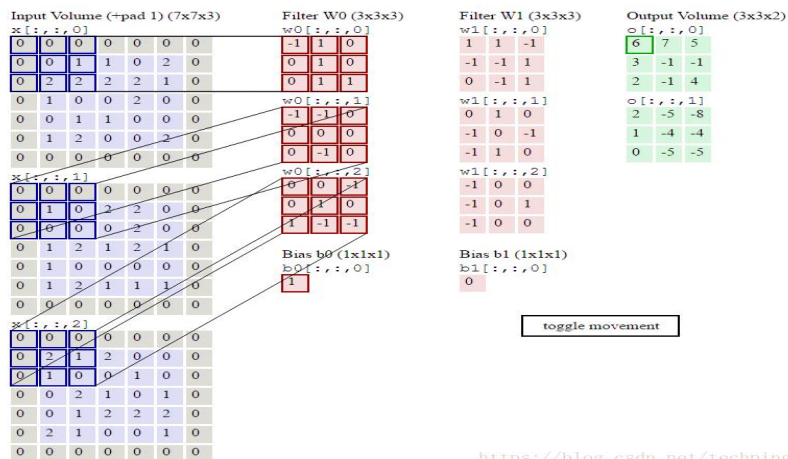


图 8 多通道卷积计算细节

通过上图可以看到，输入图像有 3 通道，总共有两组卷积核，共生成了两个特征图。图 1 中，单通道的输入可以变为 6 个特征图，又可以变为 16 个特征图，就是因为分别使用了 6 组和 16 组卷积核，最终生成了对应个数的特征图。

卷积计算后输出的特征图的计算公式如下：

若输入为 $n \times w_i \times h_i$ 的矩阵（图像）；卷积核为 $m \times (n \times f \times f)$ ，填充为 p ；步长 stride 为 s ，输出特征图像为 $k \times (w_o \times h_o)$ 。其中：

$$\begin{cases} w_o = \frac{w_i - f + 2p}{s} + 1 \\ h_o = \frac{h_i - f + 2p}{s} + 1 \\ k = m \end{cases}$$

七、池化操作

池化操作的作用主要是降维。早期的池化通常作用于图像中不重合的区域，根据实际需要也可以重叠池化。不重叠的池化过程如图 9，池化也有一个池化窗口，窗口按照一定步长扫过特征图，生成降维的特征图。

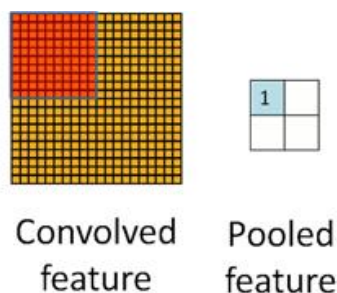


图 9 池化示意图

有两种常见的池化，最大池化和平均池化。最大池化，就是取最大值作为输出值，平均池化就是取平均值作为输出值。

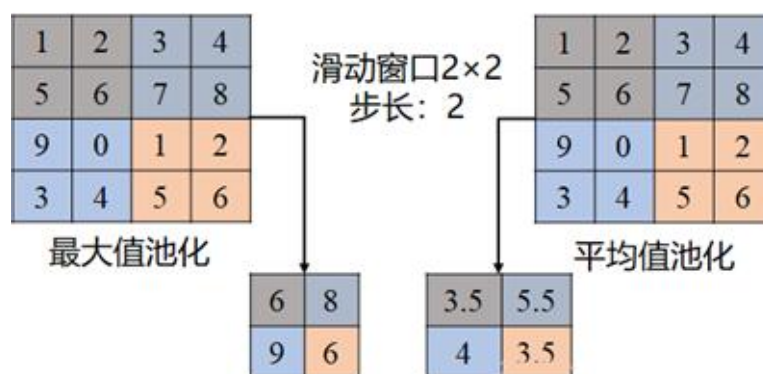


图 10 常见池化计算示意图

八、CNN 实现手写体数字识别流程图

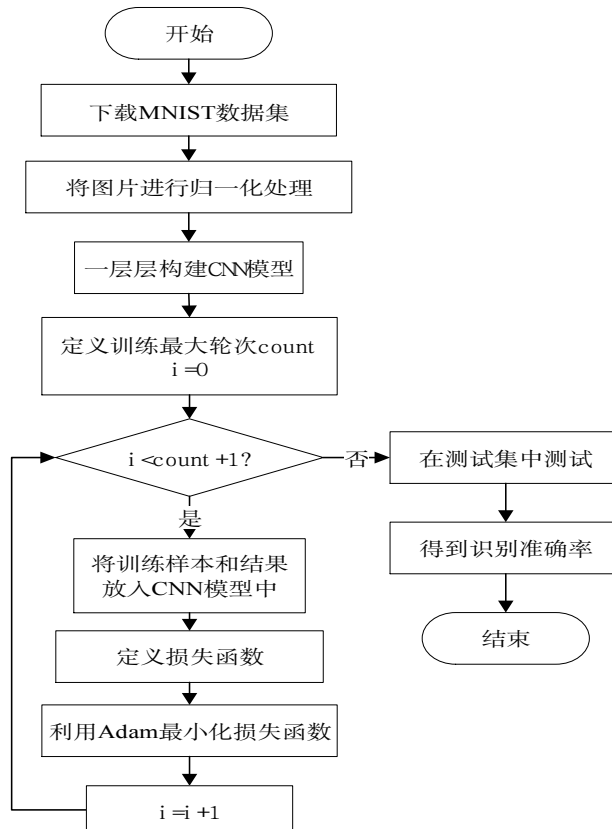


图 11 CNN 实现手写体数字识别流程图

九、数据集介绍

1、MNIST 数据集

MNIST 数据集有两种规格，小数据集（1797 张 8*8 的灰度图）和大数据集（70000 张 28*28 的灰度图）。sklearn 中内置的 digits 手写字体数据集是小数据集。本实验中第一部分，即用简单的 CNN 完成手写体数字识别，用小数据集完成模型的训练和测试。实验的第二部分，即用 Le-Net5 实现手写体数字识别用大数据集。

在计算机视觉领域中，MNIST 数据集是经典的考察和比较基础模型相关性能的数据集，一般网络上提供的 **MNIST 数据集是 gz 格式的压缩包**，在导入模型前，首先要解压缩，之后再对数据进行预处理操作。解压缩的 MNIST 数据集为原始的二进制字节文件，加载二进制数据可以通过 python 的 struct 库进行二进制解码。二进制是计算机底层通用的编码，因此，许多高级数值计算库提供了为二进制进行解码和读取的功能，而且更为高效，如 Numpy。

MNIST 数据集分为训练集和测试集两部分，共 4 个文件，训练集图像（train-images-idx3-ubyte.gz）、训练集标签（train-labels-idx1-ubyte.gz）、测试集图像（t10k-images-idx3-ubyte.gz）、测试集标签（t10k-labels-idx1-ubyte.gz）。训练集包括 60000 张数字图像，训练集标签有与之对应的 60000 个数字 0~9 之间的标签。测试集 10000 张图像，测试集标签有与之对应的 10000 个数字 0~9 之间的标签。训练集和测试集中的图像的大小为 28 像素*28 像素，因此每张图对应 784 个字节的数据。训练集图像数据和测试集图像数据中前 16 个字节偏移量为无关数据，其后为图像数据。标签数据集中前 8 个字节为无关数据，

其后每一个字节就是一个标签。

由以上信息可知，MNIST 数据集的加载方法是，对于图像而言，首先读取所有字节，跳过前 16 个无关字节，将余下的数据按照 784 个字节进行分割，每一组 784 字节就是一幅图像。对标签集而言，首先读取所有字节，跳过前 8 个无关字节，将余下的数据按照一个字节一个标签进行分割。

2、手写体图片的特点

手写体数据集中的手写体图片是 28 像素*28 像素的灰度图，每个像素由 8 位二进制编码构成，可表现 0-255 级的灰度级。

【实验内容】

本实验用 Python 编程实现 CNN，完成手写体识别。**(特别提示：要求完全用 Python 编程实现 CNN，不可以调用 Pytorch 等深度学习框架。)** 具体内容包括以下五部分：1) im2col 与 col2im 的实现；2) 卷积层的编程实现；3) 池化层的编程实现；4) 全连接层的编程实现；5) CNN 的实践---用于手写体识别(小数据集)；6) Le-Net5 的实践---用于手写体识别（大数据集），Le-Net5 的池化层采用最大池化。**激活函数：1)卷积层、全连接层激活函数是 sigmoid；2)最后输出层的激活函数是 Softmax；3)池化层激活函数采用恒等函数。**

程序实现过程中用到的变量较多，归纳与表 1 中。

表 1 变量名一览表

| 变量名 | 说明 | 变量名 | 说明 |
|-----|---------|-----|----------|
| B | 批次的大小 | C | 输入图像的通道数 |
| Ih | 输入图像的高度 | Iw | 输入图像的宽度 |
| M | 过滤器的数量 | Fh | 过滤器的高度 |
| Fw | 过滤器的宽度 | Oh | 输出图像的高度 |
| Ow | 输出图像的宽度 | P | 池化区域的大小 |

一、im2col 与 col2im 的实现

为了让卷积层和池化层的代码简洁并较快执行，本实验引入 im2col 与 col2im 两种算法。im2col (image to columns) 和 col2im (columns to image) 分别实现表示图像的二维数组与矩阵的相互转换。以卷积层为例说明 im2col 与 col2im。

在实际应用中，要考虑到批次以及图像的多通道，因此，输入图像是 4 维数组，即四阶张量 (B, C, I_h, I_w)。产生的输出图像也是相同的张量格式，如图 12 中的 (1) 所示。卷积层中的过滤器一般为多个，每个过滤器的通道数与图像的通道数相同。因此，过滤器的整体示意图如图 12 中的 (2) 所示。过滤器也是一个四维数组，即四阶张量。在编程实现卷积层时，必须实现对多层重叠在一起的多维数组的处理。

使用 im2col 与 col2im 算法进行变换的示意图如图 13 所示。

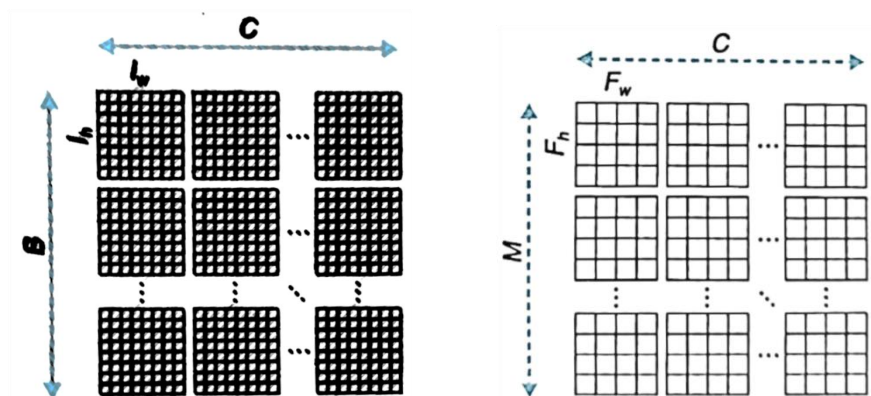


图 12 (1) 输入卷积的图像 (考虑批次和通道) (2) 过滤器的整体示意图

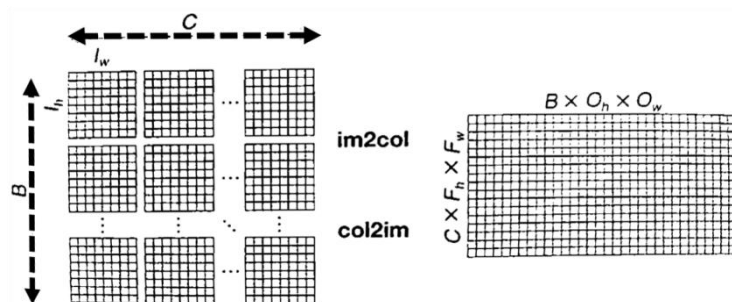


图 13 使用 im2col 与 col2im 变换的示意图

1、im2col 的算法实现

包括 1) 图像的矩阵化, 2) 滤波器的矩阵化以及 3) 卷积运算的实现。

图像的矩阵化。卷积运算可以通过使用矩阵乘法运算来编写简洁的代码实现, 因此需要将输入图像的形状变换为适合矩阵运算的形状。如图 14 所示。由图 14 (1) 可以看出, 图像中的区域是从左上方向右侧滑动, 区域中的元素被转行成矩阵的一个列。当区域范围到达最右边时, 下移一格, 然后继续向右滑动。最终, 图像被转换成图 14 中 (2) 所示的矩阵。由于滑动区域有重叠, 最后得到的矩阵的元素数量要比原图中的像素数多。

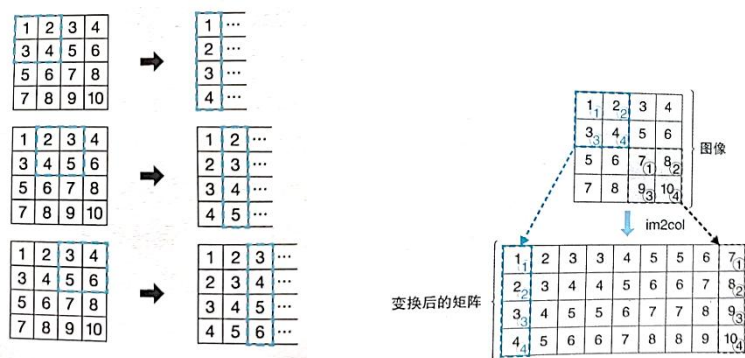


图 14 单通道图像使用 im2col 进行变换

卷积运算时, 过滤器在输入图像上相互重叠的区域之间滑动, 因此, 产生的矩阵的列数就是过滤器所重叠的位置的数量, 与输出图像的像素数 $O_h \times O_w$ 相等。过滤器的数量要与通道数保持一致, 输出图像的数量与批次大小相等。因此考虑通道数和批次, 使用 im2col 进行变化后得到的矩阵形状如图 15 所示。

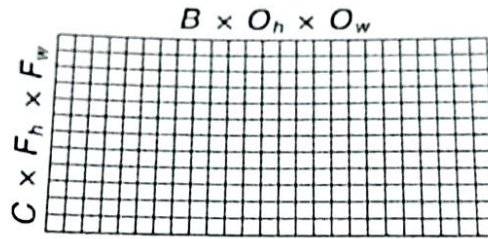


图 15 使用 Im2col 生成的包含批次和通道的矩阵

滤波器的矩阵化。为了实现图像矩阵和滤波器进行乘法运算，需要将多个滤波器转换到同一个矩阵中。如图 16(1)所示,单通道的多个滤波器集中存放的矩阵与图 16(2)所示多通道的多个滤波器集中存放的矩阵。

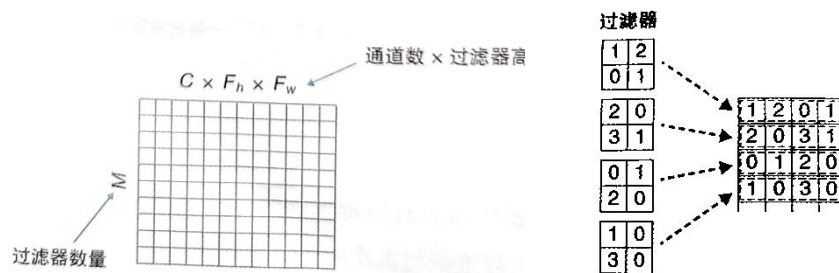


图 16 (1) 多个滤波器转换的矩阵 (2) 包含全部滤波器的矩阵

矩阵乘法运算。包含全部滤波器的矩阵和图像的矩阵进行乘法运算，可以一次性完成对图像的卷积运算。如图 17 所示，乘号前面的矩阵中各个行对应的是各个滤波器，乘号后面的矩阵中各个列对应的是滤波器覆盖的区域。经过矩阵相乘得到的形状是 $(M, B \times O_h \times O_w)$ ，将其转换为 (B, M, O_h, O_w) ，得到卷积层输出数据的形状。

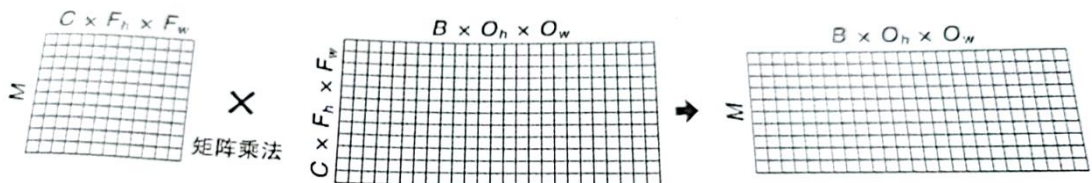


图 17 使用矩阵相乘进行卷积运算

使用图 18 左边的代码实现 im2col, 该代码存在一个明显的问题, 需要执行 $out_h \times out_w$ 次循环, 当输出图像的尺寸较大时, 代码的执行速度很慢。这是由 NumPy 的特性决定的。需要降低对 NumPy 数组的访问次数。对代码作改进, 如图 18 右图所示。

```

import numpy as np

def im2col(image, flt_h, flt_w, out_h, out_w): # 输入图像, 过滤器的高
    img_h, img_w = image.shape # 输入图像的高度和宽度
    cols = np.zeros((flt_h*flt_w, out_h*out_w)) # 生成的矩阵的尺寸

    for h in range(out_h):
        h_lim = h + flt_h
        for w in range(out_w):
            w_lim = w + flt_w
            cols[:, h*out_w:w] = img[h:h_lim, w:w_lim].reshape(-1)

    return cols

img = np.array([[1, 2, 3, 4],
                [5, 6, 7, 8],
                [9, 10, 11, 12],
                [13, 14, 15, 16]])
cols = im2col(img, 2, 2, 3, 3)
print(cols)

```

```

import numpy as np

def im2col(image, flt_h, flt_w, out_h, out_w):
    img_h, img_w = image.shape
    cols = np.zeros((flt_h, flt_w, out_h, out_w))

    for h in range(flt_h):
        h_lim = h + out_h
        for w in range(flt_w):
            w_lim = w + out_w
            cols[h, w, :, :] = img[h:h_lim, w:w_lim]

    cols = cols.reshape(flt_h*flt_w, out_h*out_w)

    return cols

img = np.array([[1, 2, 3, 4],
                [5, 6, 7, 8],
                [9, 10, 11, 12],
                [13, 14, 15, 16]])
cols = im2col(img, 2, 2, 3, 3)
print(cols)

```

图 18 im2col 函数

支持批次和多通道的实用化的 im2col 函数, 代码如下:

```

def im2col(images, flt_h, flt_w, out_h, out_w):
    n_bt, n_ch, img_h, img_w = images.shape # 批次尺寸, 通道数, 输入图像的高度和宽度
    cols = np.zeros((n_bt, n_ch, flt_h, flt_w, out_h, out_w))

    for h in range(flt_h):
        h_lim = h + out_h
        for w in range(flt_w):
            w_lim = w + out_w
            cols[:, :, h, w, :, :] = images[:, :, h:h_lim, w:w_lim]

    cols = cols.transpose(1, 2, 3, 0, 4, 5).reshape(n_ch*flt_h*flt_w, n_bt*out_h*out_w)
    return cols

img = np.array([[[[1, 2, 3, 4],
                  [5, 6, 7, 8],
                  [9, 10, 11, 12],
                  [13, 14, 15, 16]]]])
cols = im2col(img, 2, 2, 3, 3)
print(cols)

```

在支持批次和多通道的实用化的 im2col 函数的基础上, 写出支持填充与步长处理的 im2col 函数, 并输出对原图像填充的结果以及 im2col 转换后的 cols。

2、col2im 的算法实现

col2im 实现的是 im2col 的逆变换, 是将矩阵转换为图像的算法, 在卷积层和池化层的反向传播中会使用到。图 19 是 col2im 算法的示意图。将矩阵的每一列恢复到滤波器覆盖的区域, 并在转换时对重复的位置进行求和处理。实现将矩阵转换为图像。

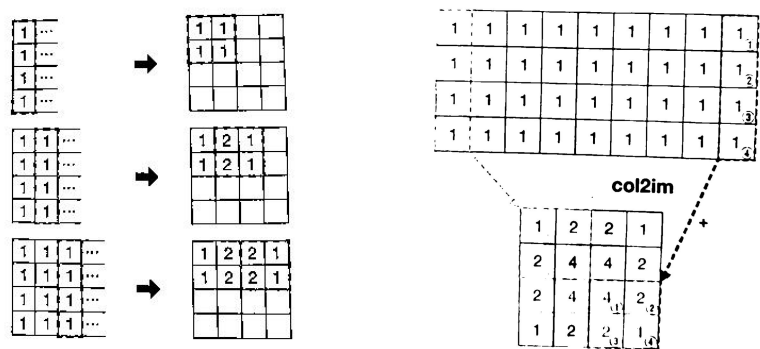


图 19 使用 col2im 转换（支持批次和多通道）

由于滤波器经过的区域有重叠，因此得到的图像的像素总数要比矩阵的元素总数少。而且，对位于图像边缘的像素的加法运算次数也相应减少。如果考虑批次和多通道问题，使用 col2im 进行处理前矩阵的形状为 (CF_hF_w, BO_hO_w) 。然后，将矩阵按照图 20 所示的方式转换为图像。转换得到的图像是与输入图像相同形状的四阶张量（四维数组）。

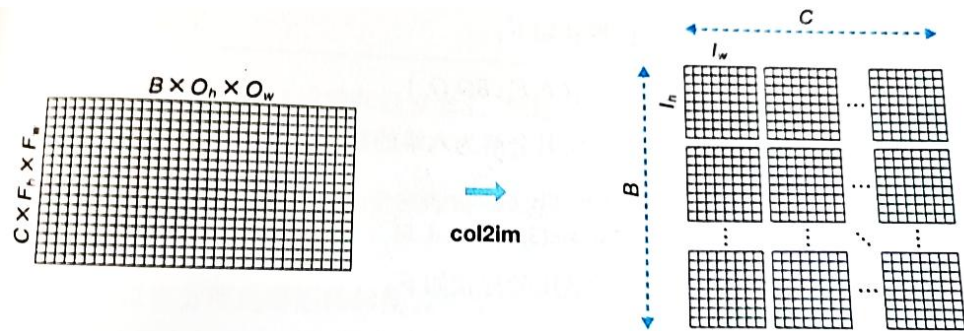


图 20 使用 col2im 进行转换(支持批次和多通道)

请将输入的形状为 (CF_hF_w, BO_hO_w) 的矩阵使用 col2im 的算法转换为形状为 (B, C, I_h, I_w) 的图像。并将输出图像及代码贴到实验报告中。

二、卷积层的编程实现

本模块包括三部分内容：卷积层类的构建、正向传播方法的构建以及反向传播类的构建。

1、卷积层类的构建

卷积层中具体的数据处理流程如图 21 所示。图 21 上方是正向传播的流程，cols 是经过 im2col 变换得到的矩阵，cols 与滤波器的矩阵相乘进行卷积处理，并将加上偏置后经过激励函数处理产生的结果作为输出数据。图 21 下方是反向传播的流程，使用激励函数对下层网络传播回来的输出梯度进行微分得到的结果为 δ ，偏置的梯度就变成 δ 。滤波器的梯度则可以通过 δ 与 cols 的矩阵乘法运算得到。随后 δ 与滤波器的矩阵乘积作为 cols 的形状，再经过 col2im 处理就被恢复成图像形状数组，最后这个数组就被作为输入的梯度。

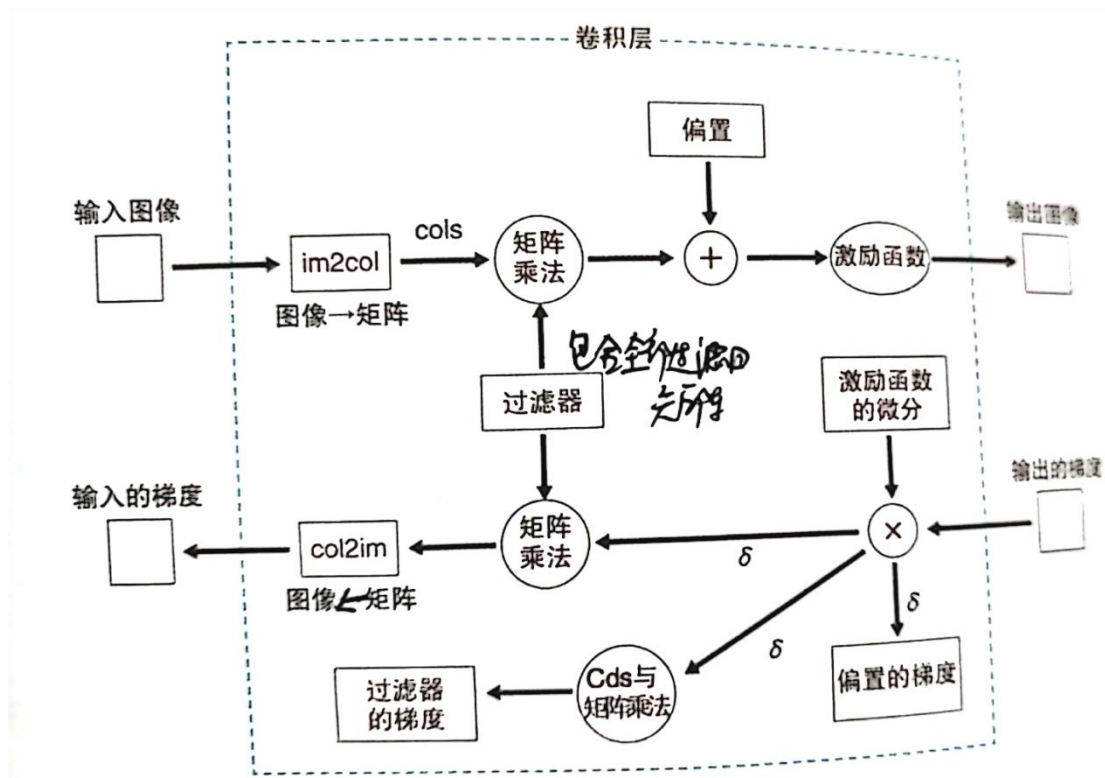


图 21 卷积层中具体的数据处理流程

请依据图 21 的流程，使用如下的类来对卷积层进行定义。随后，在这个类中实现正向传播和反向传播的类方法编写。并截屏到实验报告上。

-- 卷积网络层 --

class ConvLayer:

n_bt: 批次尺寸, x_ch: 输入的通道数量, x_h: 输入图像的高度, x_w: 输入图像的宽度
nflt: 过滤器的数量, flt_h: 过滤器的高度, flt_w: 过滤器的宽度
stride: 步长的幅度, pad: 填充的幅度
y_ch: 输出的通道数量, y_h: 输出的高度, y_w: 输出的宽度

def __init__(self, x_ch, x_h, x_w, nflt, flt_h, flt_w, stride, pad):

将参数集中保存

self.params = (x_ch, x_h, x_w, nflt, flt_h, flt_w, stride, pad)

2、正向传播

卷积层中所使用的正向传播处理可以用如下公式表示，其中 f 是激励函数。

$cols = im2col(\text{输入数据})$

$\text{输出数据} = f(\text{滤波器与 cols 的矩阵乘积} + \text{偏置})$

根据图 21 的上部分流程，对卷积层正向传播的处理进行编程，并将程序截屏到实验报告：

- 1) 使用 `im2col` 将输入的图像转换成矩阵。
- 2) 将多个滤波器转换到同一个矩阵中去。
- 3) 对表示输入图像的矩阵和表示滤波器的矩阵进行矩阵乘法运算。
- 4) 与偏置相加。

- 5) 调整最终输出的张量形状。
- 6) 使用激励函数进行处理 (ReLU 函数)

```
def forward(self, x):
    n_bt = x.shape[0]
    x_ch, x_h, x_w, nflt, flt_h, flt_w, stride, pad = self.params
    y_ch, y_h, y_w = self.y_ch, self.y_h, self.y_w
```

3、反向传播

卷积层中的反向传播处理的是, 接收下层网络的输入数据的梯度 (本层的输出数据的梯度), 并对滤波器的梯度、偏置的梯度、该网络层的输入数据的梯度 (上层网络输出数据的梯度) 进行求解。然后, 将这个网络层的输入数据的梯度传播给上层网络。与一般神经网络不同的是, 这里**求取的是滤波器的梯度而不是权重**。

卷积神经网络中的梯度计算与全连接神经网络中对中间层的各个梯度计算类似。以下公式供参考。

图 22 是图 21 中截取的, 通过图 22 所示的方式对卷积层中的各个梯度进行求解。在反向传播处理中, 需要使用正向传播中用到的滤波器, 以及在进行正向传播处理时 im2col 生成的 cols 变量。根据图 22, 对卷积层中反向传播按照如下流程进行:

- 1) $\delta_{out} = \text{输出梯度} * \text{激励函数微分}$ 。
- 2) $\delta_{cols} = \text{cols 与 } \delta_{out} \text{ 的矩阵乘积}$ 。
- 3) $\delta_{bias} = \delta_{out}$ 。
- 4) $\delta_{cols} = \delta_{cols} \text{ 与 滤波器的矩阵乘积}$ 。
- 5) $\delta_{in} = \text{col2im}(\delta_{cols} \text{ 的梯度})$ 。

请按照如上内容的提示, 完成卷积层的反向传播的编程实现 (已提供前三行代码)。

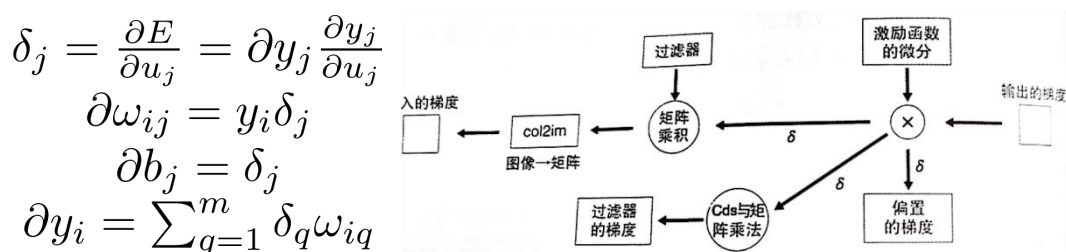


图 22 卷积层中的反向传播

```
def backward(self, grad_y):
    n_bt = grad_y.shape[0]
    x_ch, x_h, x_w, nflt, flt_h, flt_w, stride, pad = self.params
    y_ch, y_h, y_w = self.y_ch, self.y_h, self.y_w
```

三、池化层的编程实现

本模块包括三部分内容: 池化层类的构建、正向传播方法的构建以及反向传播类的构建。池化层中进行的处理流程如图 23 所示。

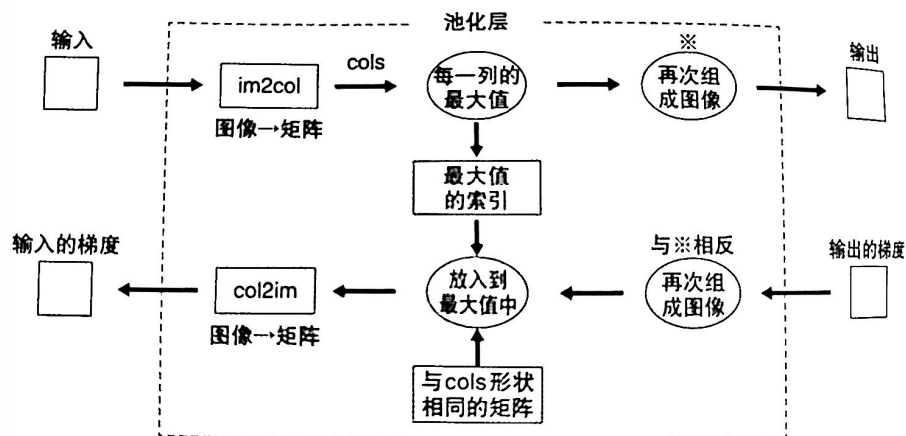


图 23 池化层进行的处理流程

池化层的构建。池化层的输入输出图像和卷积层的输入输出具有相同的形式，也是包含批次和多通道的四阶张量。**本实验采用最大池化处理**，因此正向传播中，选区图像各区域的最大值来生成缩小的图像。使用 `im2col` 将输入图像转换成矩阵，将滤波器的宽度和步长的幅值设置成相同的值（即池化的区域），将图像分割成正方形区域，被划分出来的区域转换成矩阵的列。如图 24 所示。然后使用转换出来的矩阵中的每一列的最大值组成新的图像，这样就得到了使用图像中各个区域中的最大值生成的图像，如图 25 所示。

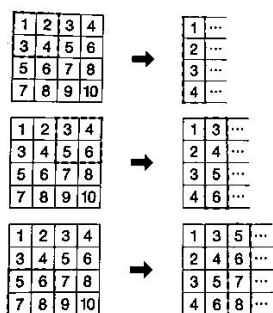


图 24 使用 `im2col` 转换为矩阵

图 25 最大值的抽取与图像重构

池化层是封装为类来实现的。池化层的构造器中，将不需要改动的参数和不需要从外部访问的参数集中保存到 `self.params` 变量中。为了允许从外部对输出图像的通道数、高度、宽度等信息进行访问，将其添加到 `self` 的例变量中。

之后将在这个类中实现正向传播和反向传播。

按照上述内容的提示，完成池化层的类的构造。（已提供前三行代码），并截屏与实验报告中。


```

# -- 池化层 --
class PoolingLayer:

    # n_bt: 批次尺寸, x_ch: 输入的通道数量, x_h: 输入图像的高度, x_w: 输入图像的宽
    # pool: 池化区域的尺寸, pad: 填充的幅度
    # y_ch: 输出的通道数量, y_h: 输出的高度, y_w: 输出的宽度

    def __init__(self, x_ch, x_h, x_w, pool, pad):

        # 将参数集中保存
        self.params = (x_ch, x_h, x_w, pool, pad)

        # 输出图像的尺寸
        self.y_ch = x_ch # 输出的通道数量
        self.y_h = x_h//pool if x_h%pool==0 else x_h//pool+1 # 输出的高度
        self.y_w = x_w//pool if x_w%pool==0 else x_w//pool+1 # 输出的宽度

```

正向传播。进行正向传播时，是对 cols 的每一列中的最大值提取出来，同时保存每一个最大值的索引。根据图 23，将按照如下流程对正向传播进行编程实现：

- 1) 实验 `im2col` 函数将输入的图像转换为矩阵。
- 2) 对每一列求取最大值。
- 3) 使用求得的最大值对图像重构，并将其作为输出数据。
- 4) 保存每列中的最大值的索引。

将实现上述流程的类方法的代码截屏到实验报告中。（提供前三行代码）

```

def forward(self, x):
    n_bt = x.shape[0]
    x_ch, x_h, x_w, pool, pad = self.params
    y_ch, y_h, y_w = self.y_ch, self.y_h, self.y_w

```

反向传播。池化层的反向传播，同样是对输入的梯度进行传播。不过，池化层的反向传播只对每个区域中具有最大值的像素进行误差传播。因此，需要使用正向传播时保存的每个区域中最大值的索引。其实现的过程是，首先创建一个与正在进行正向传播所取得的各个区域中最大值的索引矩阵具有相同形状的矩阵。然后，将输出的梯度保存在这个矩阵的每一列中相应的最大值的元素内，其余的元素设置为 0。然后使用 `col2im` 函数将此矩阵恢复为图像，并作为输入的梯度。如图 26 和图 27 所示。

池化层中不进行学习处理，因此就不存放权重和偏置等参数。在反向传播中需要计算的只有输入的梯度。池化层反向传播流程图如图 28 所示。

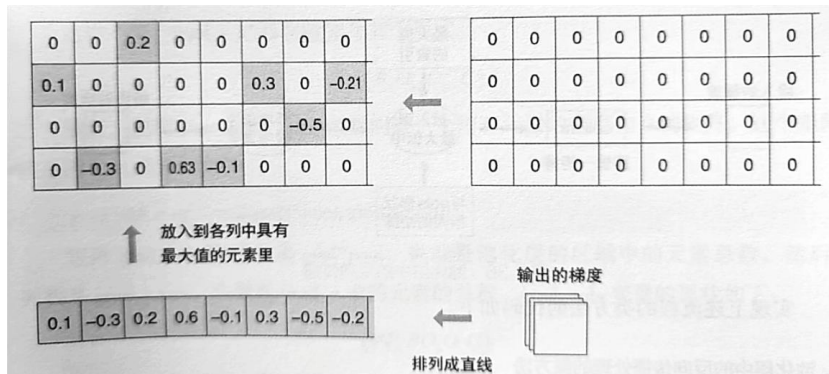


图 26 将输出的梯度保存到各列中具有最大值的元素内



图 27 使用 col2im 函数将矩阵恢复为图像

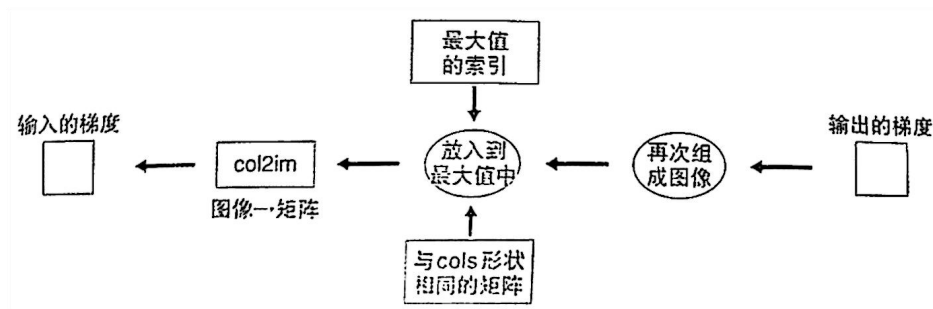


图 28 池化层中的反向传播

根据图 28，按照下面的流程对池化层的反向传播处理进行编程。

- 1) 将输出的梯度从图像的形状转换为直线的形状。
- 2) 创建与 cols 相同尺寸的矩阵。
- 3) 将输出的梯度放入到这个矩阵的每列中具有最大值的元素内。
- 4) 使用 col2im 函数将矩阵转换为图像的形状，并将其作为输入的梯度。

按照如上流程实现其类的代码，并截图与实验报告。（提供前三行代码）

```
def backward(self, grad_y):
    n_bt = grad_y.shape[0]
    x_ch, x_h, x_w, pool, pad = self.params
    y_ch, y_h, y_w = self.y_ch, self.y_h, self.y_w

    # 对输出的梯度的坐标轴进行切换
    grad_y = grad_y.transpose(0, 2, 3, 1)

    # 创建新的矩阵，只对每个列中具有最大值的元素所处位置中放入输出的梯度
```

四、全连接层的编程实现

全连接层与一般的神经网络中的神经元层完全相同。全连接层包括中间层和输出层。中间层的激励函数使用 ReLU 函数，输出层的激励函数使用 SoftMax 函数。损失函数用的是交叉熵误差函数。

补全代码构建包含中间层和输出层的全连接层。（给出两行代码）

```
# -- 全链接层的祖先类 --
class BaseLayer:
    def __init__(self, n_upper, n):
        self.w = wb_width * np.random.randn(n_upper, n)
        self.b = wb_width * np.random.randn(n)
```

```
# -- 全链接的中间层 --
class MiddleLayer(BaseLayer):
    def forward(self, x):
        self.x = x
        self.u = np.dot(x, self.w) + self.b
        self.y = np.where(self.u <= 0, 0, self.u)
```

```
# -- 全链接的输出层 --
class OutputLayer(BaseLayer):
    def forward(self, x):
        self.x = x
        u = np.dot(x, self.w) + self.b
        self.y = np.exp(u) / np.sum(np.exp(u), axis=1).reshape(-1, 1)

    def backward(self, t):
        delta = self.y - t
```

五、卷积神经网络的实践

要构建的 CNN 网络结构如图 29 所示。输入层后连接的是卷积层和池化层，经过中间层连接到最后的输出层。输出层的输出数据还是 10 个，每个输出的数据代表的是分类到对应数值的概率。其中卷积层和中间全连接层的激励函数是 ReLU；输出层的激励函数是 SoftMax 函数；损失函数是交叉熵函数；最优化算法 adaGRad(见图 30)；批次尺寸为 8。

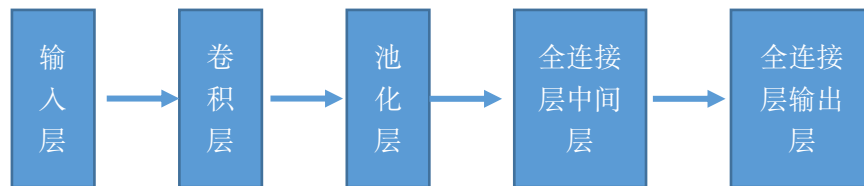


图 29 构建的 CNN 网络结构

```
class BaseLayer:
    def __init__(self, n_upper, n):
        self.w = wb_width * np.random.randn(n_upper, n)  # 权重 (矩阵)
        self.b = wb_width * np.random.randn(n)            # 偏置 (向量)

        self.h_w = np.zeros((n_upper, n)) + 1e-8
        self.h_b = np.zeros(n) + 1e-8

    def update(self, eta):
        self.h_w += self.grad_w * self.grad_w
        self.w -= eta / np.sqrt(self.h_w) * self.grad_w

        self.h_b += self.grad_b * self.grad_b
        self.b -= eta / np.sqrt(self.h_b) * self.grad_b
```

图 30 优化算法 adaGRad

本次实验小数据集的获取以及显示代码如下：

```

%matplotlib inline

import matplotlib.pyplot as plt
from sklearn import datasets

digits = datasets.load_digits()
print(digits.data.shape)

plt.imshow(digits.data[0].reshape(8, 8), cmap="gray")
plt.show()

print(digits.target.shape)
print(digits.target[:50])

fig = plt.figure()
for i in range(10):
    ax = fig.add_subplot(2, 5, i+1)
    ax.tick_params(labelbottom="off", bottom="off")
    ax.tick_params(labelleft="off", left="off")
    plt.imshow(digits.data[i].reshape(8, 8), cmap="gray")
    plt.title(digits.target[i])

plt.show()

```

本次实验大数据集的获取以及显示代码如下：

1) 导入库

```

import tensorflow as tf
import pylab
from tensorflow.python.framework import graph_util

```

2) 获取数据集 MNIST

```

from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)

```

补全代码，实现手写体识别。并通过图表显示出训练误差与测试误差随着训练轮次的变化情况，使用全部的训练数据和测试数据分别对结果的正确率进行评估。