二值图像的形态学变换

F1603703 蔡一凡 516030910375

一、 概述

本次实验对二值图像形态学变换（主要指腐蚀、膨胀操作及其衍生的开操作、闭操作）进行了实现。

本次实验采用了 python 语言，并主要使用了如下的库：

- tkinter：用于图形化界面（GUI）的展示；
- matplotlib：用于图像的读取操作；
- PIL：用于将图像从二维矩阵形式转换为 tkinter 能识别的形式；
- numpy：用于基本的矩阵存储及运算；
- threading：用于实现多线程运算与展示

除了上述提到的内容，实验的主要内容均为手动实现，如算法等均为使用库中提供的接口。

二、 主要功能及其实现

本次实验在图形化界面的基础上实现了图像的读取、展示、膨胀、腐蚀操作，并且支持手动输入可变的结构元。

1. 程序界面

程序入口为 main.py 文件，运行后如下图所示，程序界面主要由三个按钮、两个输入框和一个展示画布组成。

Path 对应的输入框支持图片路径的指定，可以输入 img 目录下的图像文件名或相对路径，亦可输入绝对路径来对计算机中的图像进行读取；Kernel 对应的输入框为结构元的设定，每个数值的范围为 0-255，同一行以英文逗号“,”隔开。

Load image 按钮为读取图像，如果图像较大，则可能需要一定的时间；erosion 和 dilation

分别对应腐蚀和膨胀操作。

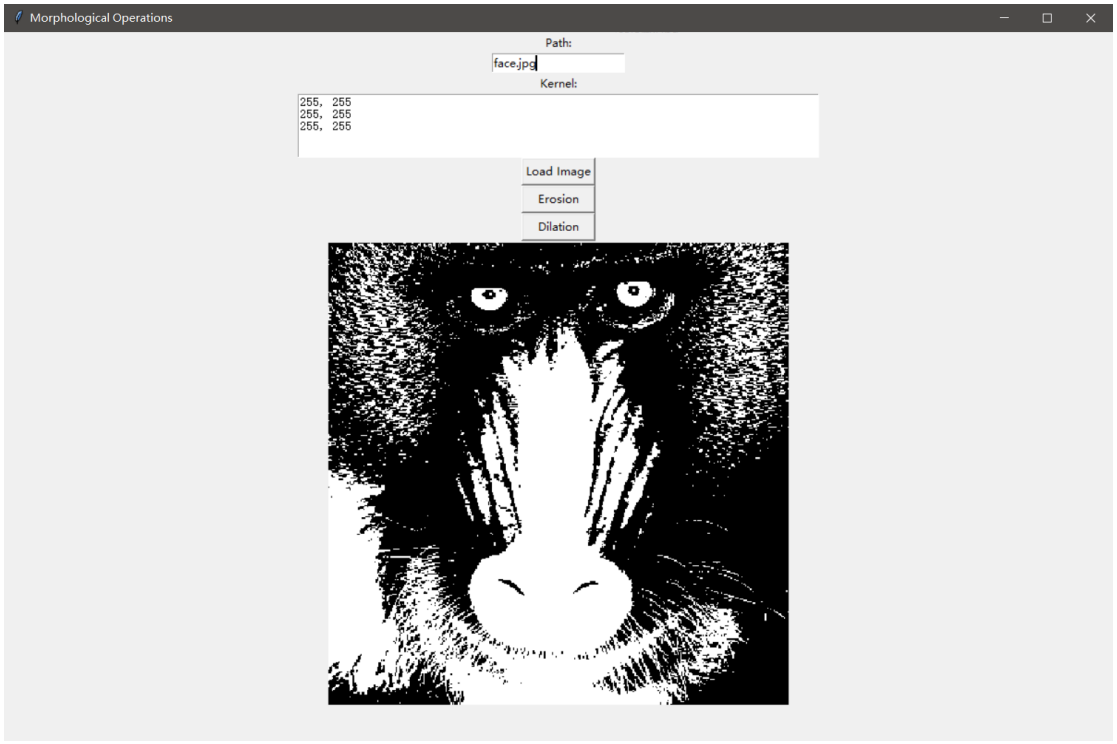

图 1 程序界面

2. 核心算法实现

本次实验采用 numpy.array 作为图像的保存形式。首先在进行膨胀和腐蚀操作之前，先对图像的尺寸做微小的调整，以满足结构元扫描的需要。为了保证先被扫描到的像素运算后的结果不会影响到之后被扫描的点，我们另开辟了一个二维 array 来储存处理后的结果。

主要过程是从左到右，从上到下依次扫描与结构元尺寸对应个数的像素点。

在进行腐蚀操作时，将结构元矩阵与原图的对应部分矩阵相减，并求出结构矩阵种元素的最大值。若为 0，则代表结构元中每个白色的像素点均对应了图中的白色像素，因而结果中该点为白色，否则为黑色。

在进行膨胀操作时，将结构元矩阵与原图的对应部分矩阵相加，并求出结构矩阵中元素的最大值。若为 510（用灰度表示，若用二值表示则为 2），则代表结构元对应的白色像素有原图中的白色与之对应，因此输出图像的该点为白色，否则为黑色。

3. 算法输出

采用的输入为如左下的图像，进行二值化处理后如下右图所示。

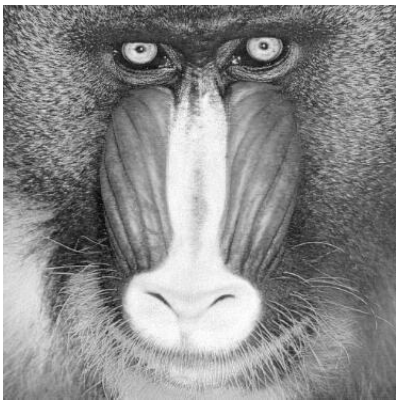

图 2 原图

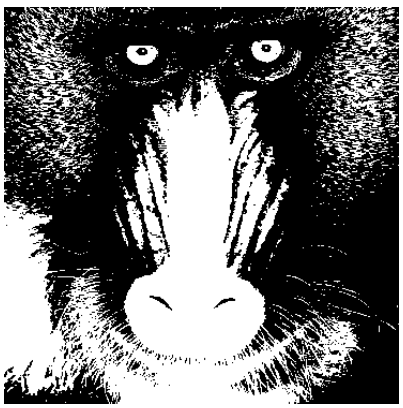

图 3 二值化后的图像

1）用 3×3 的全填充结构元进行操作

可以观察到腐蚀以及开操作后，图像亮部减少；膨胀以及闭操作后，图像亮部增多，符合理论预期。

图 4　一次腐蚀操作后的图像

图 5　一次膨胀之后的操作

图 6 一次闭操作之后的图像

图 7 一次开操作之后的图像

2） 用 1×5 的全填充结构元进行操作

可以观察到使用 1×5 的结构元进行操作后，输出图像中可以明显观察到横条纹理，与理论预期相符。

图 8 一次腐蚀操作之后的图像

图 9 一次膨胀操作之后的图像

图 10 一次腐蚀操作之后的图像

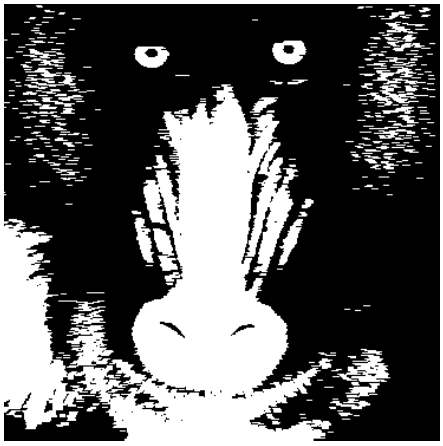

图 11 一次膨胀操作之后的图像

三、 额外功能的实现与探索

1. 从彩色图像到二值图像的实现

从彩色图像到灰度图像，采用了通用的算法，即将 R, G, B 分别乘以 0.299, 0.587, 0.114 并相加，得到灰度图像。

从灰度图像到二值图像，常用的阈值选择方式有取平均值、OTSU、Kittle 等算法。本程序中，我采用并手动实现了了 OTSU 算法[1]。OTSU 的中心思想是阈值 T 应使目标与背景两类的类间方差最大。算法通过计算每一种可能的阈值并取其最大类间方差对应的值为最终阈值。其实现在 image.py 的 to_binary 函数中。

2. 算法的性能优化

该程序在 Windows 10 家庭版，Intel® Core™ i5-6300HQ CPU 2.30Hz，8GB RAM 环境下进行性能测试。使用 3×3 的全白结构元，对大小为 1920×1080 大小的图像进行腐蚀、膨胀操作。在默认单线程实现下，程序平均需要 14.40s 完成一次操作。考虑到速度较慢，我进行了多线程的实现。具体方法是将图像分割为四个部分，并启动四个线程进行腐蚀、膨胀的操作，并将最终结果合并在一起。

启用了多线程后，程序的性能并未得到提升，反而有所降低，平均需要 20.94s 完成一次操作。首先，我猜测可能是各线程访问了同一个矩阵变量（用于储存原图）；因此，我试图将原图进行拷贝，并让各线程访问了不同的地址。然而结果依旧是 20s 左右。

最终，我查阅了相关资料。根据相关文档[2][3]，python 在调用 C 代码时，会使用到 python 全局的 GIL(Global Interpreter Lock)。尽管在实际进行计算 np.max 的过程中，锁会被释放，然而程序中有大量的 for 循环语句来实现结构元的移动，而 np.max 的计算量相比之下显得微不足道。因此，有大量时间消耗在了获取锁和释放锁的过程中，使得多线程最终未能取得理想的效果。

四、 参考资料

[1] OTSU 算法：https://en.wikipedia.org/wiki/Otsu%27s_method

[2] GIL：https://wiki.python.org/moin/GlobalInterpreterLock

[3] GIL 与 numpy：https://scipy-cookbook.readthedocs.io/items/ParallelProgramming.html