深 圳 大 学 实 验 报 告

课程名称：智能识别系统设计

实验项目名称：图像处理与分析实践

学院：计算机与软件学院

专业：软件工程（腾班）

指导教师：沈琳琳、文嘉俊

报告人：洪子敬 学号：2022155033 班级：腾班

实验时间：2024 年 9 月 2 日至 9 月 28 日

实验报告提交时间：2024 年 9 月 28 日

教务处制

一、实验目的与要求：

实验目的：

(1) 掌握图像边缘检测算法原理和实现方法；
(2) 掌握动态阈值分割方法（大津法与迭代二值分割法）的算法原理和实现方法；
(3) 熟悉图像处理从输入到输出的整个流程；
(4) 掌握图像预处理技巧；
(5) 采用统计方法对图像进行分析，并提取有效特征实现分割目的。

实验要求：

(1) 熟悉 OpenCV 变量类型和函数调用；
(2) 熟悉 C++或 Python 语言编程；
(3) 熟悉 OpenCV 与 C++或 Python 语言结合的图像处理编程环境。

二、实验内容与方法：

对于给定的测试数据，包括 10 张车图像和 5 张车牌图像，完成以下三选一任务：

任务一：采用大津法（类内最小、类间最大）算法对车牌图像进行二值分割，并与迭代二分法进行比较，分析两种自适应阈值分割法的性能，包括二值分割质量和分割效率的比较。

任务二：结合统计方法对车图像进行处理，设计图像处理算法从车图像中对车牌进行定位。

（提示：1. 考虑提取车牌中像素剧烈变化的特征；2. 对车图像自下往上进行分析；3.对车图像中的车牌先水平分割，再垂直分割。）

任务三：在任务二的基础上，对所提取的车牌进行倾斜矫正。

（提示：把分割后的车牌区域考虑成积分面积，并考虑不同角度对积分面积特征的影响。）

三、实验步骤与过程：

1. 任务一

（1）使用大津法（类内最小、类间最大）算法对车牌图像进行二值分割

思路分析：大津法的主要思想是实现类内方差最小、类间方差最大；

给定阈值 t，通过此阈值把图像像素分为C_0和C_1两类，其中C_0和C_1出现的概率和均值分别为：w_0, w_1, μ_0, μ_1；而C_0和C_1的方差分别为：$\sigma_0^2(t), \sigma_1^2(t)$，因此我们可以得到类内方差为：

$$\sigma_w^2(t) = w_0\sigma_0^2 + w_1\sigma_1^2$$

类间方差为：

$$\sigma_B^2(t) = w_0(\mu_0 - t)^2 + w_1(\mu_1 - t)^2$$

其中:

$$\sigma^2(t) = \sum_{i=1}^{n}(x_i - \mu)^2 * p_i,\ \ \mathrm{p_i}\text{为像素点出现的概率}$$

最后我们可以得到优化的最终函数为：

$$t^* = \underset{t\in[0,L-1]}{\operatorname{argmax}}\,\sigma_B^2(t)/\,\sigma_w^2(t)$$

最优的阈值 t 通过最优化上述裁决标准得到。

代码实现：首先定义函数 ostu_hand_thresholding，并通过直方图均衡化得到每个像素的概率值，并利用此概率计算全局的平均灰度值，为后续做准备。

```
def otsu_hand_thresholding(image):
    # 计算直方图
    hist, _ = np.histogram(image.flatten(), bins=256, range=[0, 256])
    total_pixels = image.size
    # 归一化直方图
    prob = hist / total_pixels
    # 初始化变量
    max_variance = 0
    best_threshold = 0
    # 全局平均灰度值
    total_mean = np.sum(np.arange(256) * prob)
    weight_background = 0
    mean_background = 0
```

洪子敬
2022155033

接着是核心部分，我们通过迭代每一个像素值，计算前景和后景各自的权重和概率以及方差，按照之前的公式计算类间方差和类内方差，最后通过保留最大的类间方差与类内方差比对应的 t 值，从而得到最优的阈值。

```
# 迭代每个可能的阈值
for t in range(256):
    weight_background += prob[t]
    if weight_background == 0:
        continue

    weight_foreground = 1 - weight_background
    if weight_foreground == 0:
        break
    # 计算前景与后景的均值
    mean_background += t * prob[t]
    mean_foreground = (total_mean - mean_background) / weight_foreground
    # 计算前景和背景的方差
    variance_background = np.sum(((np.arange(t + 1) - mean_background) ** 2) * prob[:t + 1]) / weight_background
    variance_foreground = np.sum(((np.arange(t + 1, *args: 256) - mean_foreground) ** 2) * prob[t + 1:]) / weight_foreground

    # 计算类间方差
    variance_between = weight_background * (mean_background - t) ** 2 + weight_foreground * (mean_foreground - t) ** 2
    # 计算类内方差
    variance_within = weight_background * variance_background * variance_background + weight_foreground * variance_foreground * variance_foreground
    # 更新最大类间方差和最佳阈值
    if variance_within != 0:
        if variance_between / variance_within > max_variance:
            max_variance = variance_between / variance_within
            best_threshold = t
print("手动大津法最佳阈值为：", best_threshold)
```

洪子敬
2022155033

最后是利用此最佳阈值对图像进行二值分割，返回处理图像即可。

```
# _, thresholded_image = cv2.threshold(image, best_threshold, 255, cv2.THRESH_BINARY)
thresholded_image = np.where(image >= best_threshold, x: 255, y: 0).astype(np.uint8)
return thresholded_image
```

（2）使用迭代二分算法对车牌图像进行二值分割

思路分析：迭代二分法首先初始化阈值为最大最小灰度值的平均值，然后利用此阈值划分目标和背景，最后取目标和背景的平均灰度的平均值作为新阈值，以此往复，直到达到最大迭代步数或者前后阈值变化小于某一特定值结束。

代码实现：实现方式与思路基本一致，首先建立函数 iterative_bisection，初始化 low（背景）和 high（目标）分别为 0 和 255，通过最大迭代次数的 for 循环进行迭代，循环体内，每次通过目标背景的均值取平均计算出新的阈值，并通过此阈值计算出新的目标背景均值并更新，结束时判断前后阈值变化是否达到设置条件，是则退出循环，否则继续。最后也是利用得到的阈值对图像进行二值分割，并将得到的新图像返回即可。

```python
def iterative_bisection(image, max_iter=100, epsilon=0.01):
    # image = cv2.equalizeHist(image)
    # 迭代二分法进行阈值分割
    low, high = 0, 255
    previous_mid = -100
    for _ in range(max_iter):
        mid = (low + high) // 2
        lower_mean = image[image <= mid].mean() if np.any(image <= mid) else 0
        upper_mean = image[image > mid].mean() if np.any(image > mid) else 0

        if abs(previous_mid - mid) < epsilon:
            break
        previous_mid = mid

        low = lower_mean
        high = upper_mean

    # _, thresholded_image = cv2.threshold(image, mid, 255, cv2.THRESH_BINARY)
    thresholded_image = np.where(image >= mid, x: 255, y: 0).astype(np.uint8)
    return thresholded_image
```

洪子敬
2022155033

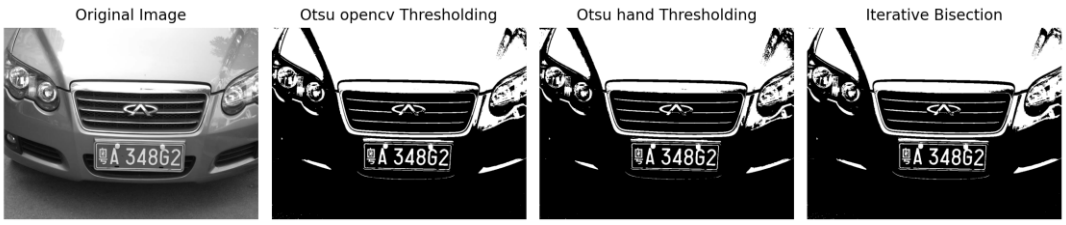
（3）两种方法的实验结果比较

由于 opencv 中自带实现了大津法的函数，所以我也调用了 opencv 中的方法与我们自己手动实现的方法进行了比较。大津法的 opencv 实现如下：

```python
def otsu_thresholding(image):
    # 使用大津法进行阈值分割
    t, thresholded_image = cv2.threshold(image, thresh: 0, maxval: 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
    print("opencv自带大津法最佳阈值为：", t)
    return thresholded_image
```

HZJ

随机选择一张车图像，使用三种方法得到结果。原始图像、大津法 opencv 实现、手动大津法实现与迭代二分法实现的结果如下所示：



HZJ

迭代二分法此处设置的最大迭代步数为 100，epsilon 设置为 0.01，当然可以设置的更加精确，但耗时也随之变长，这里只选择测试效果较好的一组参数。

通过观察上述结果我们可以看到整体二值化的效果均较为接近。直观上看不出明显差别，所以我们通过比较时间以及各自设置的阈值，如下所示：

```
opencv自带大津法最佳阈值为： 142.0
Otsu opencv Thresholding uses:  0.011564493179321289
手动大津法最佳阈值为： 150
Otsu hand Thresholding uses:  0.02086162567138672
手动迭代二分法最佳阈值为： 142.0
Iterative Bisection uses: 0.018062353134155273
```

HZJ

通过观察实验结果发现，迭代二分法阈值与 opencv 自带函数选择的阈值一样，说明参数选择较为不错，而手动大津法则会有所偏差，应该是具体实现细节上的差异导致的。在运行时间上，opencv 自带函数毋庸置疑是最快的，而手动实现的大津法相比手动迭代的二分法也要慢一些，猜测 opencv 大津法实现应该是使用某种优化算法进行加速，总体来看如果我们无法明确阈值的情况下，大津法是最优的；如果阈值范围确定，迭代二分法效果应该更好。

2. 任务二

结合统计方法对车图像进行处理，设计图像处理算法从车图像中对车牌进行定位。

思路分析：综合网上资料和个人思考，最终采用形态学方法对图像进行车牌轮廓定位，并通过 opencv 自带函数将符合预期的可能轮廓提取出来，最后通过颜色识别找到最像车牌的轮廓，将其对应位置的图像圈出，剪切出该图像并展示。

图像预处理：在使用形态学方法划分轮廓之前，我们先对图像进行了一些操作，使得后续的形态学处理更加方便和有效。

（1）图像尺寸调整

由于每张图片的像素都不一样，为了图片展示方便，我将图像最大像素限制在 400 像素。

洪子敬
2022155033

```python
# 调整尺寸
def resize_img(img):
    height,width = img.shape[:-1]
    # 限制最大像素为400
    scale = 400/max(height, width)
    img = cv2.resize(img, dsize: None, fx=scale, fy=scale,interpolation=cv2.INTER_CUBIC)
    return img
```

（2）将图像转为灰度图

```python
# 转灰度图
img_gray = cv2.cvtColor(image_resize, cv2.COLOR_BGR2GRAY)
```

（3）使用高斯模糊对图像进行降噪

```python
# 降噪
image_blured = cv2.GaussianBlur(img_gray, ksize: (5,5), sigmaX: 0)
```

（4）将图像进行灰度拉伸

对图像的像素值进行重新映射，映射到更加广泛的范围（即投影到 0-255 范围内），从而增强图像的对比度，使得图像中亮暗会更加明显。投影公式如下所示：

$$output(i,j) = \frac{255}{\max - min} * (input(i,j) - min)$$

洪子敬
2022155033

```python
# 灰度拉伸
def stretch(img):
    maxi = img.max()
    mini = img.min()
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            img[i,j] = 255/(maxi-mini)*img[i,j] - (255*mini)/(maxi-mini)
    return img
```

（5）将图像进行二值化

我们将图像的像素进行二值化（即 0 和 255），使得图像轮廓会更加清楚；实现也比较简单，这里直接利用 opencv 自带的函数中的大津法进行二值化。

HZJ

```python
# 二值化
def binarization(img):
    ret, img_binary = cv2.threshold(img, thresh: 0, maxval: 255, cv2.THRESH_BINARY+cv2.THRESH_OTSU)
    return ret, img_binary
```

（6）对图像进行边缘检测

在二值化图像的基础上，为了方便形态学对轮廓进行处理，我们使用边缘检测的方法对图像边缘进行提取，这里我们直接使用 opencv 库中的常用边缘检测方法，并设置最小阈值为 100，最大阈值为 200 进行图像操作。

```
# 边缘检测
def myCanny(img, low_threshold, high_threshold):
    img_canny = cv2.Canny(img, low_threshold, high_threshold)
    return img_canny
```

HZJ

图像形态学处理：形态学处理操作主要包括开运算和闭运算，每种运算都有腐蚀和膨胀两个操作。下面详细介绍腐蚀和膨胀操作：

腐蚀：通过一个结构元素（通常是一个小的矩形、圆形或其他形状，自定义核）对图像进行处理。它的效果是使图像中的物体边界向内收缩。具体是将结构元素在图像上滑动，当结构元素完全覆盖一个像素区域时，输出该区域的最小值。

膨胀：与腐蚀相反，它通过结构元素使图像中的物体边界向外扩展。具体是将结构元素在图像上滑动，当结构元素覆盖一个像素区域时，输出该区域的最大值。

开运算：先腐蚀后膨胀，使用结构元素对图像进行腐蚀操作，去除小的物体和噪声。腐蚀操作会使得物体的边界向内收缩，接着对腐蚀后的图像进行膨胀操作，恢复物体的大小，填补小的孔洞。**开运算可以去除小的噪声，保留较大且形状完整的物体，适用于去噪和细化形状**。

闭运算：先膨胀后腐蚀，使用结构元素对图像进行膨胀操作，扩大物体的边界，连接较近的物体，接着对膨胀后的图像进行腐蚀操作，恢复物体的原始形状。**闭运算可以填补小的孔洞和空隙，连接相邻的物体，适用于填补图像中的小间隙和增强物体的整体性**。

通过开闭运算的概念我们不能知道，首先要**先通过闭运算**填补二值图像中的细小间隙，形成完整物体，**接着使用开运算**去除小的噪声物体，最终留下我们要的轮廓图像。

具体代码实现上，我们使用的也是 opencv 自带的函数，为了能将物体进行准确分割，我们先使用两个不同核的闭运算对图像中的物体进行分割，接着用了两次开运算进行了噪声的去除。（**实际上我们发现影响最后结果的主要是闭运算的效果**）

```
# 形态学处理 开闭运算
def opening_with_closing(img):
    # 进行闭运算
    kernel = np.ones( shape: (8, 20), np.uint8)  # 定义结构元素
    # kernel = np.ones((2, 10), np.uint8)  # 定义结构元素
    img_closing1 = cv2.morphologyEx(img, cv2.MORPH_CLOSE, kernel)
    cv2.imshow( winname: "Closing image1", img_closing1)
    cv2.waitKey(0)
    kernel = np.ones( shape: (4, 15), np.uint8)  # 定义结构元素
    # kernel = np.ones((2, 15), np.uint8)  # 定义结构元素
    img_closing = cv2.morphologyEx(img_closing1, cv2.MORPH_CLOSE, kernel)
    cv2.imshow( winname: "Closing image2", img_closing)
    cv2.waitKey(0)
    kernel = np.ones( shape: (4, 6), np.uint8)  # 定义结构元素
    # 进行开运算
    img_opening1 = cv2.morphologyEx(img_closing, cv2.MORPH_OPEN, kernel)
    cv2.imshow( winname: "Opening_1 image", img_opening1)
    cv2.waitKey(0)
    # 再次进行开运算
    kernel = np.ones( shape: (8, 20), np.uint8)
    img_opening2 = cv2.morphologyEx(img_opening1, cv2.MORPH_OPEN, kernel)
    cv2.imshow( winname: "Opening_2 image", img_opening2)
    cv2.waitKey(0)
    return img_opening2
```

洪子敬
2022155033

差分、轮廓处理和颜色识别：在获得形态学处理后的图像后，我们要找到对于我们有利的部分（白色部分），所以我们用原图像与形态学处理后的结果进行差分，从而得到我们想要的结果。接着我们利用 opencv 自带的函数 findContours 函数对轮廓进行提取。对于提取到的所有轮廓，我们计算每个轮廓的面积和高宽比。接着对每个轮廓进行遍历，选择满足常见车牌形状约束的轮廓（这里设置为高宽比范围是[2,4]，面积范围是[1000,20000]），对于符合条件的轮廓，我们构建颜色掩膜，用来提取符合颜色的轮廓（由观察发现我们使用的车牌数据集中基本都是蓝色，通过查阅资料发现，常见蓝色车牌的 hsv 取值上界为[124,255,255]，取值下界为[100,43,46]。至于使用 hsv 而不是 rgb 是因为 hsv 更适合进行颜色识别）。对于符合条件的轮廓，我们计算其轮廓内像素点的总权重并除以轮廓面积，通过比较平均权重最大的区域从而得到最接近目标的轮廓（轮廓均通过其左上点和右下点构建矩形，最终返回的轮廓也是矩形四顶点的坐标，方便我们后续在原图像上进行划分和切割）。

```python
# 车牌定位
def locate_license(origin_image, img):
    # 提取轮廓
    contours, hierarchy = cv2.findContours(img, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    # 画出轮廓
    img_copy = origin_image.copy()
    img_copy = cv2.drawContours(img_copy, contours, -1, color: (255, 0, 0), thickness: 5)
    cv2.imshow( winname: 'All Contours',img_copy)
    cv2.waitKey(0)
    block = []
    for c in contours:
        # 找出轮廓的左上点和右下点，由此计算它的面积和长度比
        pos = find_rect_pos(c)
        # 计算轮廓面积及高宽比
        a = (pos[2] - pos[0]) * (pos[3] - pos[1])  # 面积
        s = (pos[2] - pos[0]) / (pos[3] - pos[1])  # 长度比
        block.append([pos, a, s])
    # 选出面积最大的五个区域 b[1]是对第二个元素
    block = sorted(block, key=lambda b: b[1])[-5:]
```

洪子敬
2022155033
（轮廓处理代码）

```python
# 根据颜色识别找到最像车牌的区域
maxWeight, maxIndex = 0, -1
for i in range(len(block)):
    # 高宽比、面积进行限制
    if 2 <= block[i][2] <= 4 and 1000 <= block[i][1] <= 20000:
        b = origin_image[block[i][0][1]: block[i][0][3], block[i][0][0]: block[i][0][2]]
        # 颜色空间变换 hsv更适合颜色识别
        hsv = cv2.cvtColor(b, cv2.COLOR_BGR2HSV)
        # 使用指定的 HSV 颜色范围构建掩膜，提取符合条件的颜色部分
        lower = np.array([100, 43, 46])
        upper = np.array([124, 255, 255])
        mask = cv2.inRange(hsv, lower, upper)
        # 计算掩膜中符合颜色条件的像素点的权重
        w1 = 0
        for m in mask:
            w1 += m / 255
        # 统计所有符合条件的像素点的总权重
        w2 = 0
        for n in w1:
            w2 += n
        area = block[i][1]
        averaged_weight = w2 / area if area > 0 else 0

        # 选择权重最大的区域
        if averaged_weight > maxWeight:
            maxIndex = i
            maxWeight = averaged_weight
# 选择权重最大的区域作为车牌区域
rect = block[maxIndex][0]
```

洪子敬
2022155033
（颜色识别代码）

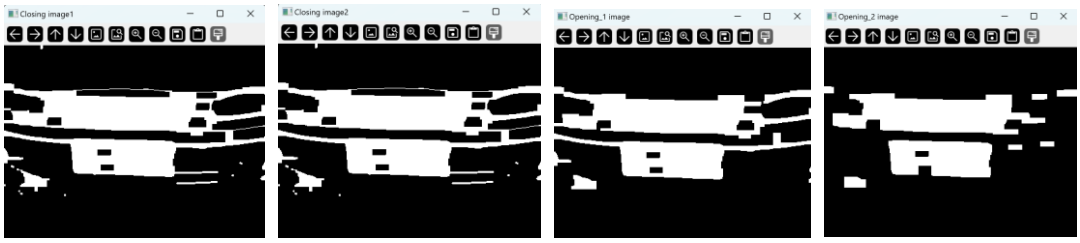
使用时也发现**最难把握的还是结构元素的设计**，对于不同的图片都达不到一致统一的核，所以效果的好坏主要**取决于核的设计能否将车牌与其他元素分隔开**，如果能成功分割，车牌就可以成功识别出来。

结果展示：在获取目标区域的四个顶点后，我们可直接调用 opencv 中 rectangle 函数将车牌画出来，并通过裁剪像素得到车牌图片。下面展示上述过程的结果图：
这里以车图像数据集中的“car5.bmp”图像为例（之前展示的形态学代码中选定的核也是

最适合此图像的核，若要更换其他图片，大概率根据情况是要调换一下核的）。

上面图像从左到右、从上到下分别是：

灰度图—高斯降噪图—灰度拉伸图—差分图—二值图—边缘检测图



上面图像从左往右分别为：

第一次闭运算—第二次闭运算—第一次开运算—第二次开运算

上面左图为所以提取出来的轮廓的展示，右图是最终选择的轮廓，我们可以看到还是很精确的，说明形态学运算十分有效（关键是核选的好），将车牌提取出来如下：

实验结果虽然十分准确，说明了形态学这种统计方法的有效性，但唯一的弊端就是很难找到符合所有车图像的核（结构元素）。实际操作时有些核可以使用多种车图像，有些还不能，这是因为容易识别到车上方的区域，这个区域干扰较大，不仅连成片，而且像素较大还无法直接去除，这是主要的影响因素，其他表现得都不错。

四、实验结论：

本次实验对任务一和任务二进行了实验，成功基于迭代二分法以及大津法的原理编写了了代码并做了效果比较，此外，还对车图像的车牌进行了提取，通过多种图像处理方法和形态学统计方法成功解决了问题，但仍可改进。本次实验圆满结束。

五、实验体会：

本次实验任务一还是比较基础，在懂的原理之后查阅相关资料就可以成功写出代码实现；比较难的还是任务二，题目要求只能使用统计学方法而不能使用深度学习和机器学习的方法，我在网上查阅了很多资料，基于统计的方法只找到了形态学处理方法，实际上还使用了很多图像处理的方法，诸如差分、边缘检测等等对图像进行了预处理，在做任务的同时增强了对 opencv 库的应用以及对图像处理方法的理解。虽然用了很多方法，但初步结果还是不理想，后面才发现主要问题是形态学的闭运算，如果核设计的好，没有把车牌和上面的区域连在一起，则最终结果则会成功显示车牌，但由于车类型的差异以及拍摄角度问题，形态学操作终会受制于核的设计，因此实验还是无法设计出通用的核对于不同的车图像。任务三也是找不到方法来解决，感觉应该是对图像处理接触的太少，算是留一个遗憾，看后续的学习能不能将其解决。
总而言之，本次实验还是学习到很多知识，虽有不足，但在不断进步，仍需继续加油。

指导教师批阅意见：

成绩评定：

指导教师签字：
2024 年 9 月 30 日

备注：