# 

|  |  |
| --- | --- |
|  | |
| 《机器视觉检测技术》  综合实践设计报告 | |
|  | |
|  |  |
| 专题名称： | 数独图像的自动识别 |
| 设计者： | 蒋汉锟 18221197  黎颖 18221007  冯璐 18221338 |
| 指导教师： | 王 尧 |
| 提交日期： | 2020年12月25日 |

目录

[1.设计题目 3](#_Toc59732255)

[2 图像分割 4](#_Toc59732256)

[2.1 自动提取 4](#_Toc59732257)

[2.2 手动提取 13](#_Toc59732258)

[3 图像识别 19](#_Toc59732259)

[3.1 图像预处理 19](#_Toc59732260)

[3.2 图像增强 19](#_Toc59732261)

[3.3 CNN网络 20](#_Toc59732262)

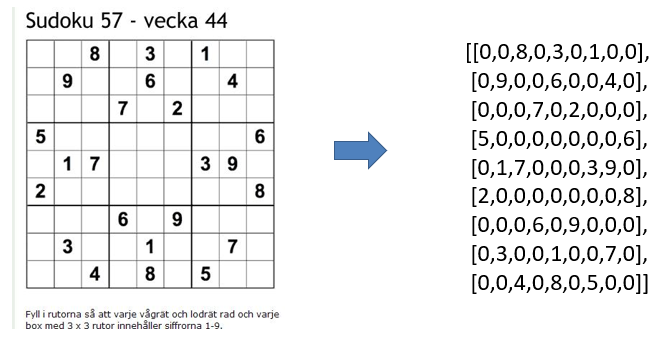
[3.4 模型训练 21](#_Toc59732263)

[4. 数独求解 21](#_Toc59732264)

# 1. 设计题目

基本要求：

* + 从图像中找到表格，提取每个小格
  + 识别小格中的数字，空白的用0表示
  + 形成识别后的9×9矩阵



* + 对图像进行适当的预处理（去掉阴影，提高对比度）提高图像的质量
  + 提高数字的识别准确率
  + 自动从变形的图像中找到数独表格，并变换到合适的角度
  + 自动实现数独的求解



2 图像分割

首先对需要处理的图片进行分析，发现图像中的数独框角度、大小、清晰程度均有差异，因此对需要提取的数独框，采用手动提取和自动提取两种方案。对于自动提取效果差的图像，采取手动提取。

读入图像——变换图像大小——中值滤波消除椒盐噪声——将图片转换为灰度图——获取数独框顶点——对图像进行第一次正畸，若效果较好，则运行自动提取程序；若效果不理想，则采用手动选点方式进行纠正。

2.1 自动提取

由于边缘识别、中值滤波、高斯滤波等算法中的参数需要根据图像大小进行调整，因此我们首先对读入的图像做缩放，至2560×1920大小

*# 选择读入的图像的编号*my\_img\_number = int(input(**'请输入1-24，读取相应编号的图像：**\n**'**))  
*# 读入图像*original\_img = cv2.imread(**'../images/sudokus/sudoku{}.jpg'**.format(my\_img\_number))  
*# 变换图像为指定大小*original\_img = cv2.resize(original\_img, (2560, 1920))

由于彩色图像通道数为3，不利于后续处理，因此，将图像转换为灰度图像

*# 将图片数据类型转换为灰度图*gray\_img = cv2.cvtColor(median\_img, cv2.COLOR\_RGB2GRAY)

对图像进行预处理，由于中值滤波对椒盐噪声有很好的去噪效果，且能够保留边缘，因此，我们对图像进行中值滤波

*# 中值滤波器，消除椒盐噪声*median\_img = cv2.medianBlur(original\_img, 5)

对读入的图像进行观察，如果有明显阴影的话，则需要进行阴影去除，否则会对后续图像二值化的阈值确定、边缘识别产生影响

对于阴影去除，由于阴影的颜色较有效图像的颜色较深，因此我们采取对图像执行一次最小滤波、一次最大滤波后，再归一化使用背景减法得到最终的去除阴影的图像。

最大滤波：对于有一定大小的图像I，逐个遍历I的像素，并且对于每个像素（x，y），找到该像素周围的邻域（大小为N×N的窗口）中的最大灰度值，并进行写入A中相应像素位置（x，y）的最大灰度值。所得图像A称为输入图像I的最大滤波图像。

*# 最大滤波* def max\_filtering(N, I\_temp):  
 wall = np.full((I\_temp.shape[0] + (N // 2) \* 2, I\_temp.shape[1] + (N // 2) \* 2), -1)  
 wall[(N // 2):wall.shape[0] - (N // 2), (N // 2):wall.shape[1] - (N // 2)] = I\_temp.copy()  
 temp = np.full((I\_temp.shape[0] + (N // 2) \* 2, I\_temp.shape[1] + (N // 2) \* 2), -1)  
 for y in range(0, wall.shape[0]):  
 for x in range(0, wall.shape[1]):  
 if wall[y, x] != -1:  
 window = wall[y - (N // 2):y + (N // 2) + 1, x - (N // 2):x + (N // 2) + 1]  
 num = np.amax(window)  
 temp[y, x] = num  
 A = temp[(N // 2):wall.shape[0] - (N // 2), (N // 2):wall.shape[1] - (N // 2)].copy()  
 return A

最小滤波：对于有一定大小的图像I，逐个遍历I的像素，并且对于每个像素（x，y），找到该像素周围的邻域（大小为N×N的窗口）中的最小灰度值，并进行写入B中相应像素位置（x，y）的最小灰度值。所得图像B称为输入图像I的最小滤波图像。

*# 最小滤波* def min\_filtering(N, A):  
 wall\_min = np.full((A.shape[0] + (N // 2) \* 2, A.shape[1] + (N // 2) \* 2), 300)  
 wall\_min[(N // 2):wall\_min.shape[0] - (N // 2), (N // 2):wall\_min.shape[1] - (N // 2)] = A.copy()  
 temp\_min = np.full((A.shape[0] + (N // 2) \* 2, A.shape[1] + (N // 2) \* 2), 300)  
 for y in range(0, wall\_min.shape[0]):  
 for x in range(0, wall\_min.shape[1]):  
 if wall\_min[y, x] != 300:  
 window\_min = wall\_min[y - (N // 2):y + (N // 2) + 1, x - (N // 2):x + (N // 2) + 1]  
 num\_min = np.amin(window\_min)  
 temp\_min[y, x] = num\_min  
 B = temp\_min[(N // 2):wall\_min.shape[0] - (N // 2), (N // 2):wall\_min.shape[1] - (N // 2)].copy()  
 return B

归一化的背景减法

*# 标准化* def background\_subtraction(I, B):  
 O = I - B  
 norm\_img = cv2.normalize(O, None, 0, 255, norm\_type=cv2.NORM\_MINMAX)  
 return norm\_img

然后对图像进行第一次正畸，由于后续算法中，我们采用Sobel算子对图像X方向、Y方向的边缘分别进行提取，于是需要首先将图像摆正。我们采用边缘识别，认定其中最小外接四边形面积最大的边缘为数独框，获取到此最小外接四边形的4个顶点后，采用仿射变换，对图像进行第一次正畸

def min\_square\_contour(img):  
 *"""  
 提取出数独框所在的最小外接四边形，对图像进行第一次正畸  
 :param img: 获取图像  
 :return: affine\_trans\_img: 仿射变换后的图像  
 """  
 # 提取图像边缘* edges = cv2.Canny(img, 20, 100, apertureSize=3)  
 *# 腐蚀* kernel = cv2.getStructuringElement(cv2.MORPH\_RECT, (15, 15)) *# 腐蚀矩阵* corrosion\_img = cv2.morphologyEx(edges, cv2.MORPH\_DILATE, kernel) *# 腐蚀运算* img\_contour, contours, hierarchy = cv2.findContours(corrosion\_img, cv2.RETR\_TREE, cv2.CHAIN\_APPROX\_SIMPLE)  
 *# 创建显示最小四边形的容器* img\_tmp = np.ones(img.shape, dtype=np.uint8)  
 *# 创建最小外接四边形顶点的容器* boxes = []  
 *# 获取每个轮廓的最小外接四边形* for c in contours:  
 *# 找面积最小的矩形* rect = cv2.minAreaRect(c)  
 *# 得到最小矩形的坐标* tmp\_box = cv2.boxPoints(rect)  
 *# 标准化坐标到整数* tmp\_box = np.int0(tmp\_box)  
 boxes.append(tmp\_box)  
 *# 根据轮廓面积从大到小排序* boxes = sorted(boxes, key=cv2.contourArea, reverse=True)  
 *# 获取面积最大的最小外界四边形（认为此为数独框轮廓的最小外接四边形）* cv2.drawContours(img\_tmp, [boxes[0]], 0, (255, 255, 255), 10)  
 print(**'最小外接四边形的顶点为：**\n**'**, boxes[0])  
 print(**'如果输出的图像不正确，请首先确保顶点顺序依次应为左下角、左上角、右上角、右下角，否则需要对此函数获取仿射变换变换矩阵的部分进行修改！'**)  
 *# 下面进行仿射变换，进行对图像的第一次正畸  
 # 获取原图的宽高* h, w = img.shape[:2]  
 dst = np.array([[0, h], [0, 0], [w, 0]], np.float32)  
 src = np.array([boxes[0][0], boxes[0][1], boxes[0][2]], np.float32)  
 *# 获取变换矩阵* trans = cv2.getAffineTransform(src, dst)  
 *# 仿射变换* affine\_trans\_img = cv2.warpAffine(img, trans, (w, h))  
 figure()  
 title(**'数独框的第一次正畸'**)  
 imshow(affine\_trans\_img, **'gray'**)  
 print(**'关闭所有图像窗口后程序继续运行'**)  
 show()  
 return affine\_trans\_img

得到视角变换后的图像，进行第一次检查，如果效果好，则继续进行下一步的数独框提取操作，否则进行手动提取

对得到的图像进行边缘检测，并得到最大矩形轮廓，认定此轮廓为数独框轮廓，采用Sobel算子分别对x方向、y方向进行边缘提取并对提取出的边缘做闭操作，增强边缘。再对得到的x方向、y方向的边缘取与的位运算，就能够得到数独框的4个角点，再次对这4个角点做轮廓检测，就能够得到这4个角点的位置坐标。再利用透射变换就能够得到数独框。

def get\_points\_autoly(img):  
 *"""  
 自动获取数独框的4个顶点  
 :param img: 输入图像  
 :return: points: 得到的数独框4个顶点  
 """  
 # 统计概率霍夫直线变换* def line\_detect\_possible\_demo(image):  
 lines = cv2.HoughLinesP(image, 1, np.pi / 180, threshold=100, minLineLength=0, maxLineGap=100)  
 *# 创建保存霍夫变换直线检测结果的容器* hough\_img = np.ones(image.shape, dtype=np.uint8)  
 for line in lines:  
 x1, y1, x2, y2 = line[0]  
 cv2.line(hough\_img, (x1, y1), (x2, y2), (255, 255, 255), 10)  
 *# 进行概率霍夫直线检测* ret, hough\_img = cv2.threshold(hough\_img, 127, 255, cv2.THRESH\_BINARY)  
 return hough\_img  
 *# 边缘检测* edges = cv2.Canny(img, 10, 60, apertureSize=3)  
 *# 腐蚀* kernel = cv2.getStructuringElement(cv2.MORPH\_RECT, (30, 30)) *# 腐蚀矩阵* corrosion\_img = cv2.morphologyEx(edges, cv2.MORPH\_DILATE, kernel) *# 腐蚀运算  
 # 轮廓检测* img\_contour, contours, hierarchy = cv2.findContours(corrosion\_img, cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE)  
 *# 得到最大矩形轮廓* max\_area = 0  
 biggest\_contour = None  
 for cnt in contours:  
 area = cv2.contourArea(cnt)  
 if area > max\_area:  
 max\_area = area  
 biggest\_contour = cnt  
 *# 创建最大矩形轮廓的容器* tmp\_img = np.ones(img.shape, dtype=np.uint8)  
 *# 画出最大矩形轮廓* tmp\_img = cv2.drawContours(tmp\_img, [biggest\_contour], 0, (255, 255, 255), 30)  
 *# tmp\_img = cv2.cvtColor(tmp\_img, cv2.COLOR\_RGB2GRAY)  
  
 # 通过Sobel算子提取x方向的边缘，并做直线检测后做闭操作* dx = cv2.Sobel(tmp\_img, cv2.CV\_16S, 1, 0)  
 dx = cv2.convertScaleAbs(dx)  
 cv2.normalize(dx, dx, 0, 255, cv2.NORM\_MINMAX)  
 ret, close = cv2.threshold(dx, 0, 255, cv2.THRESH\_BINARY + cv2.THRESH\_OTSU)  
 kernelx = cv2.getStructuringElement(cv2.MORPH\_RECT, (2, 10))  
 close = cv2.morphologyEx(close, cv2.MORPH\_DILATE, kernelx, iterations=1)  
 binary, contour, hierarchy = cv2.findContours(close, cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE)  
 for cnt in contour:  
 x, y, w, h = cv2.boundingRect(cnt)  
 if h / w > 5:  
 cv2.drawContours(close, [cnt], 0, 255, -1)  
 else:  
 cv2.drawContours(close, [cnt], 0, 0, -1)  
  
 close = cv2.morphologyEx(close, cv2.MORPH\_CLOSE, None, iterations=1)  
 closex = close.copy()  
 line\_closex = line\_detect\_possible\_demo(closex)  
 kernel = cv2.getStructuringElement(cv2.MORPH\_RECT, (40, 40)) *# 腐蚀矩阵* fushi\_line\_closex = cv2.morphologyEx(line\_closex, cv2.MORPH\_DILATE, kernel) *# 腐蚀运算  
  
 # 通过Sobel算子提取y方向的边缘，并做直线检测后做闭操作* dy = cv2.Sobel(tmp\_img, cv2.CV\_16S, 0, 1)  
 dy = cv2.convertScaleAbs(dy)  
 cv2.normalize(dy, dy, 0, 255, cv2.NORM\_MINMAX)  
 ret, close = cv2.threshold(dy, 0, 255, cv2.THRESH\_BINARY + cv2.THRESH\_OTSU)  
 kernely = cv2.getStructuringElement(cv2.MORPH\_RECT, (10, 2))  
 close = cv2.morphologyEx(close, cv2.MORPH\_DILATE, kernely, iterations=1)  
 binary, contour, hierarchy = cv2.findContours(close, cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE)  
 for cnt in contour:  
 x, y, w, h = cv2.boundingRect(cnt)  
 if w / h > 5:  
 cv2.drawContours(close, [cnt], 0, 255, -1)  
 else:  
 cv2.drawContours(close, [cnt], 0, 0, -1)  
  
 close = cv2.morphologyEx(close, cv2.MORPH\_CLOSE, None, iterations=1)  
 closey = close.copy()  
 line\_closey = line\_detect\_possible\_demo(closey)  
 kernel = cv2.getStructuringElement(cv2.MORPH\_RECT, (100, 60)) *# 腐蚀矩阵* fushi\_line\_closey = cv2.morphologyEx(line\_closey, cv2.MORPH\_DILATE, kernel) *# 对腐蚀运算  
 # 得到数独框的4个顶点* res = cv2.bitwise\_and(fushi\_line\_closex, fushi\_line\_closey)  
 *# 轮廓检测* binary, contour, hierarchy = cv2.findContours(res, cv2.RETR\_LIST, cv2.CHAIN\_APPROX\_SIMPLE)  
 centroids = []  
 for cnt in contour:  
 if cv2.contourArea(cnt) > 20:  
 mom = cv2.moments(cnt)  
 (x, y) = int(mom[**'m10'**] / mom[**'m00'**]), int(mom[**'m01'**] / mom[**'m00'**])  
 centroids.append((x, y))  
 centroids = np.array(centroids, dtype=np.float32)  
 c = centroids.reshape((4, 2))  
 c2 = c[np.argsort(c[:, 1])]  
 *# 得到角点* b = np.vstack([c2[i \* 10:(i + 1) \* 10][np.argsort(c2[i \* 10:(i + 1) \* 10, 0])] for i in range(10)])  
 bm = b.reshape((2, 2, 2))  
 return bm  
  
  
def perspective\_trans(img, src\_points):  
 *"""  
 透视变换  
 :param img: 需要透视变换的图像  
 :param src\_points: 数独框顶点  
 :return: perspective\_trans\_img: 透视变换后的图像  
 """* h, w = img.shape[:2]  
 dst = np.array([[0, h], [0, 0], [w, 0], [w, h]], np.float32)  
 src = np.array([src\_points[0][0], src\_points[0][1], src\_points[1][0], src\_points[1][1]], np.float32)  
 *# 得到透视变换的矩阵* trans = cv2.getPerspectiveTransform(src, dst)  
 perspective\_trans\_img = cv2.warpPerspective(img, trans, (w, h))  
 return perspective\_trans\_img

再次对所得图像进行判断，如果得到数独框图像，就继续下一步的图像分割，否则采用手动提取

对于图像分割，由于图像中数独框没有较大的弯曲现象，因此我们直接等份数独框即可

def img\_splite(img, img\_number):  
 *"""  
 将数独图像分割为81个36\*36的小图像  
 :param img:  
 :param img\_number:  
 :return: None  
 """* print(**'正在进行图像分割，请等待...'**)  
 h, w = img.shape[:2]  
 h\_slice = int(h / 9)  
 w\_slice = int(w / 9)  
 for i in range(0, 9):  
 for j in range(0, 9):  
 *# 要被切割的开始的像素的高度值* beH = h\_slice \* i  
 *# 要被切割的结束的像素的高度值* hEnd = h\_slice \* (i + 1)  
 *# 要被切割的开始的像素的宽度值* beW = w\_slice \* j  
 *# 要被切割的结束的像素的宽度值* wLen = w\_slice \* (j + 1)  
 dst\_img = img[beH:hEnd, beW:wLen]  
 *# 存储图像* dst\_img = cv2.resize(dst\_img, (36, 36))  
 cv2.imwrite(**"../images/sudokus/sudoku{}/{}\_{}.png"**.format(img\_number, i + 1, j + 1), dst\_img)  
 *# 提示运行结束* print(**'已保存分割图像至文件夹sudoku{}，请至相应查验'**.format(img\_number))

如果没有建立保存分割图像的文件夹，则需要首先运行下列代码，建立对应文件夹

*# # 创建24个文件夹，分别用以装24个数独图像的分割图  
# file\_path=os.path.abspath(r"../images/sudokus")  
# for i in range(1,25):  
# a="sudoku{}".format(i)  
# file\_name=file\_path + "\\"+a  
# os.makedirs(file\_name)*

2.2 手动提取

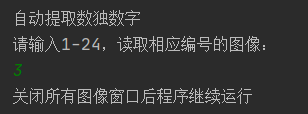
对于手动提取，我们需要手动获取数独框的4个角点，然后进行一次透射变换即可。图像的预处理和后续图像分割操作和自动提取相同

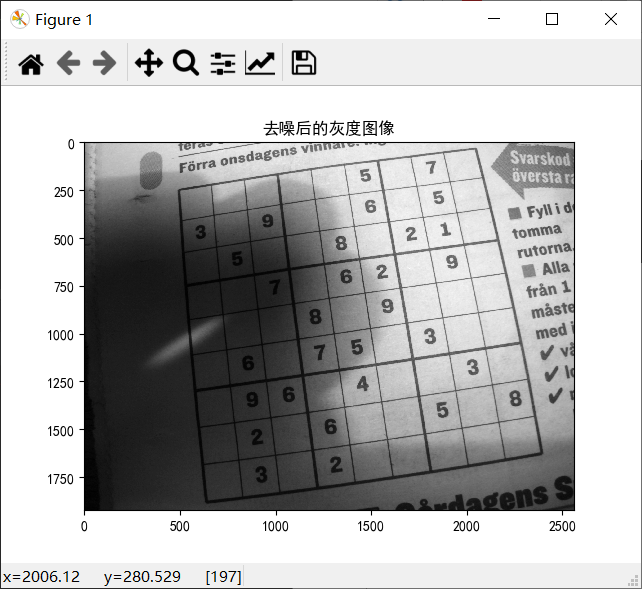
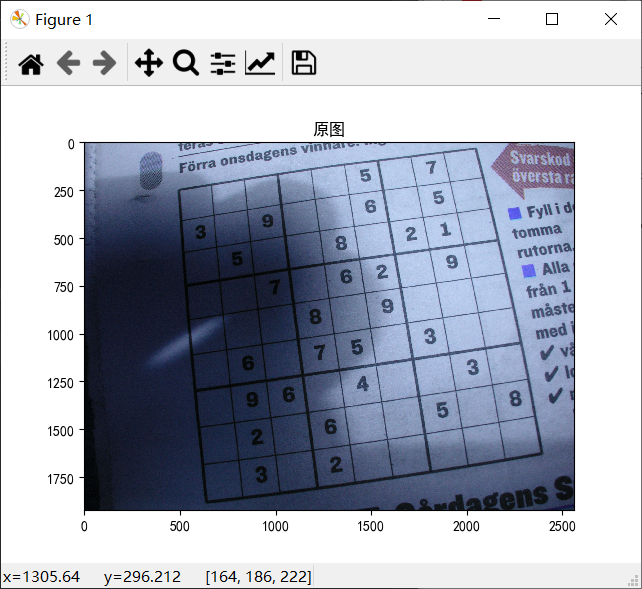
def get\_points\_manually(img):  
 *"""  
 手动获取数独框的4个顶点  
 :param img: 输入图像  
 :return: points: 得到的数独框4个顶点  
 """  
 # 改变这个参数调整对应点的对数* point\_pair = 4  
 *# 显示图像* figure()  
 title(**'选取数独框'**)  
 imshow(img, **'gray'**)  
 *# 取点* print(**'请点击{}个数独框顶点，依次应为左下角、左上角、右上角、右下角，请尽量准确选取'**.format(point\_pair))  
 *# 创建点的容器* points = []  
 for i in range(point\_pair):  
 x = ginput(1)  
 print(**'第{}个:'**.format(i + 1), x)  
 points.append(x)  
 return points

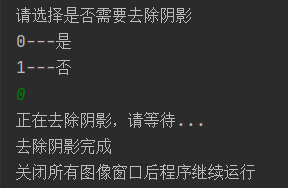
def perspective\_trans(img, src\_points):  
 *"""  
 透视变换  
 :param img: 需要透视变换的图像  
 :param src\_points: 数独框顶点  
 :return: perspective\_trans\_img: 透视变换后的图像  
 """* h, w = img.shape[:2]  
 dst = np.array([[0, h], [0, 0], [w, 0], [w, h]], np.float32)  
 *# 对于实物，未检测出来的图像采取手动取点* src = np.array([src\_points[0], src\_points[1], src\_points[2], src\_points[3]], np.float32)  
 *# 得到透视变换的矩阵* trans = cv2.getPerspectiveTransform(src, dst)  
 perspective\_trans\_img = cv2.warpPerspective(img, trans, (w, h))  
 return perspective\_trans\_img

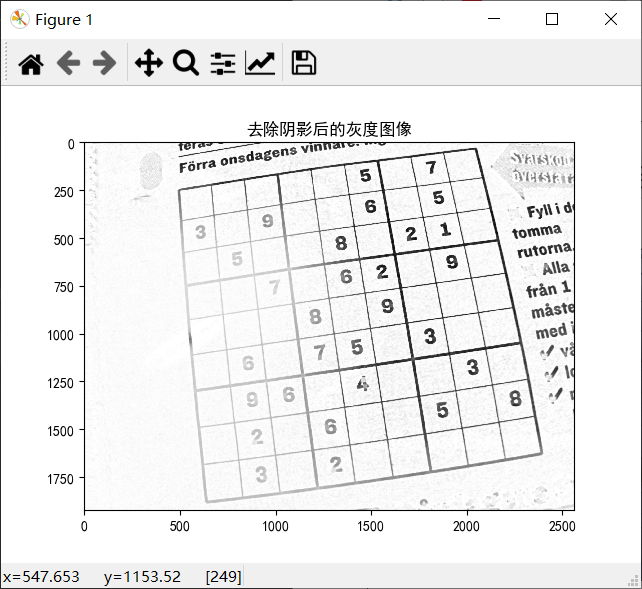
具体运行效果展示如下：（包括输入与输出）

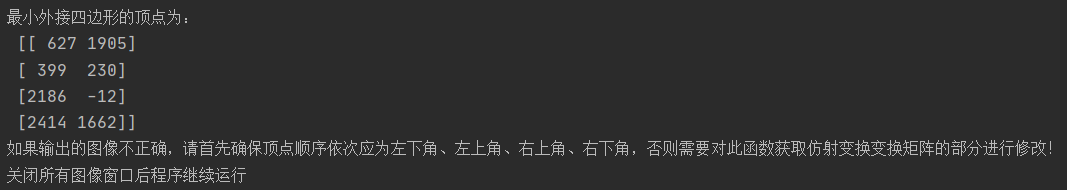
自动提取：

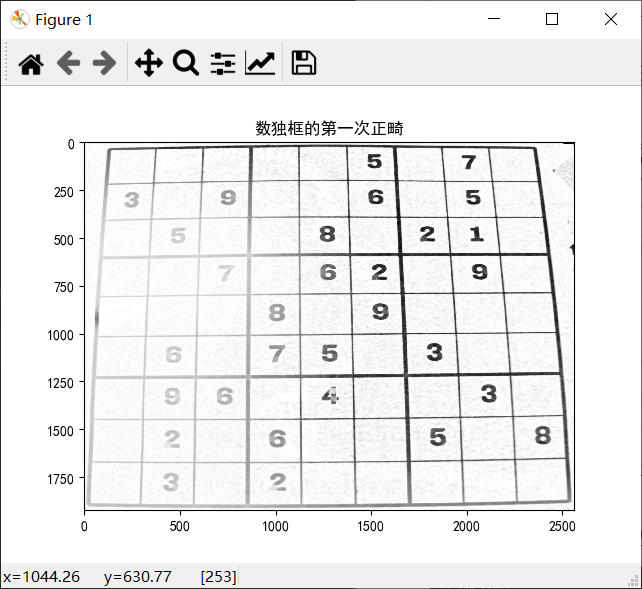


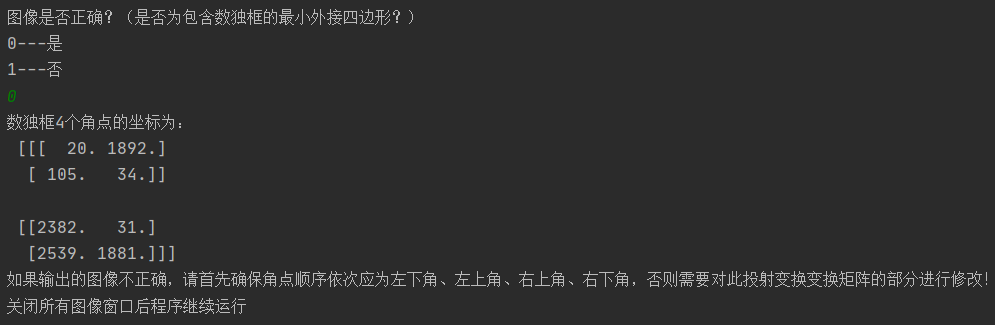


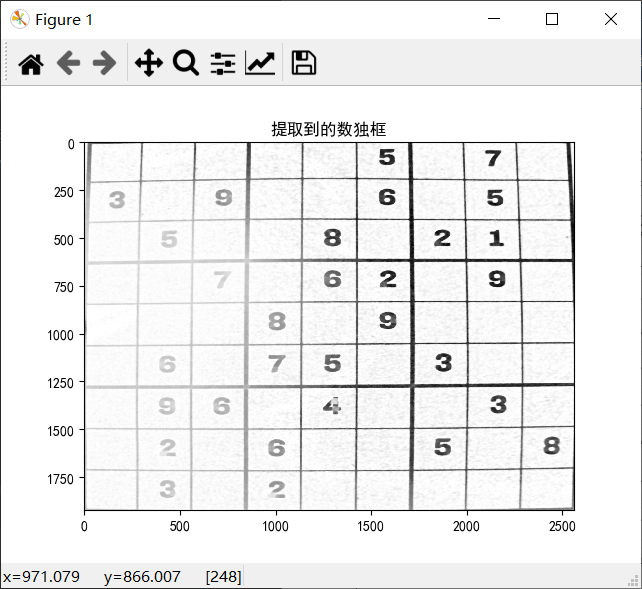


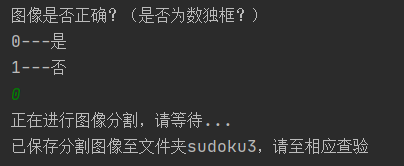


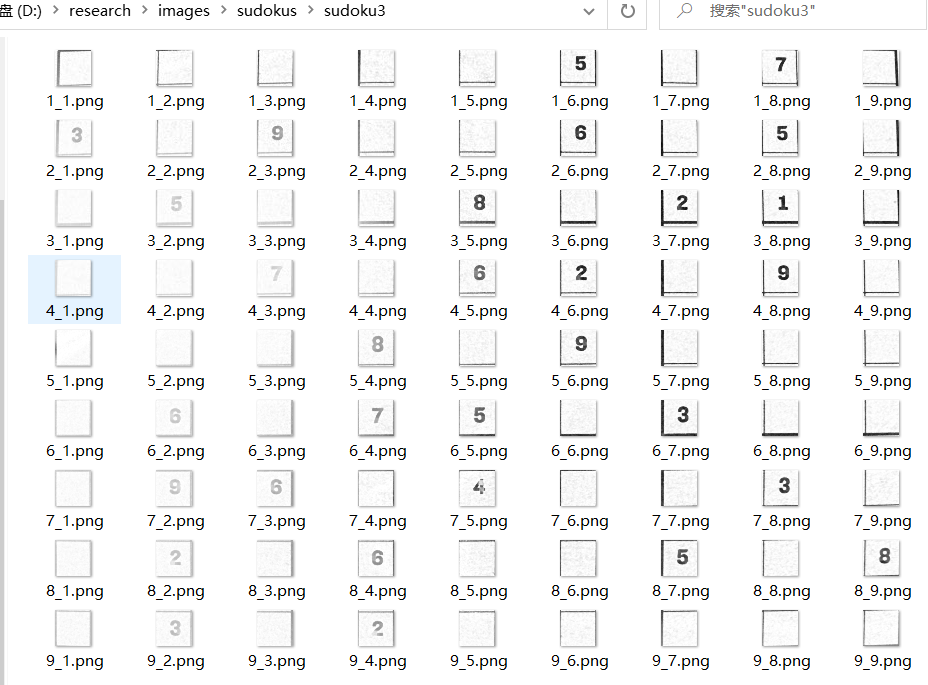




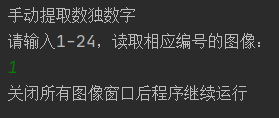


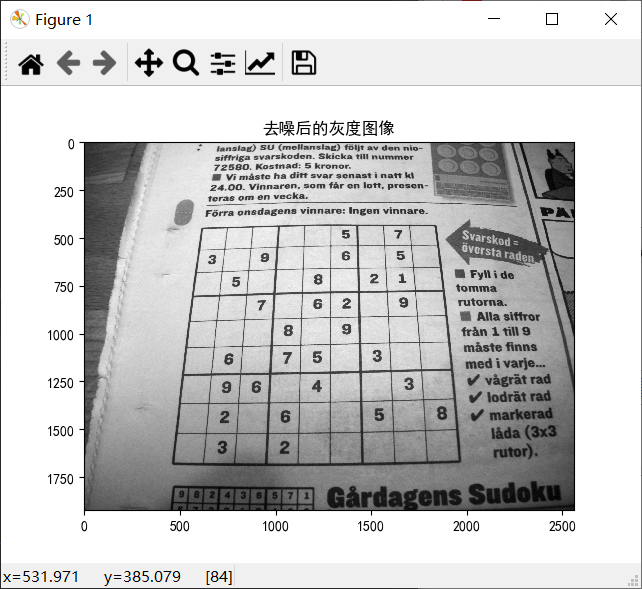
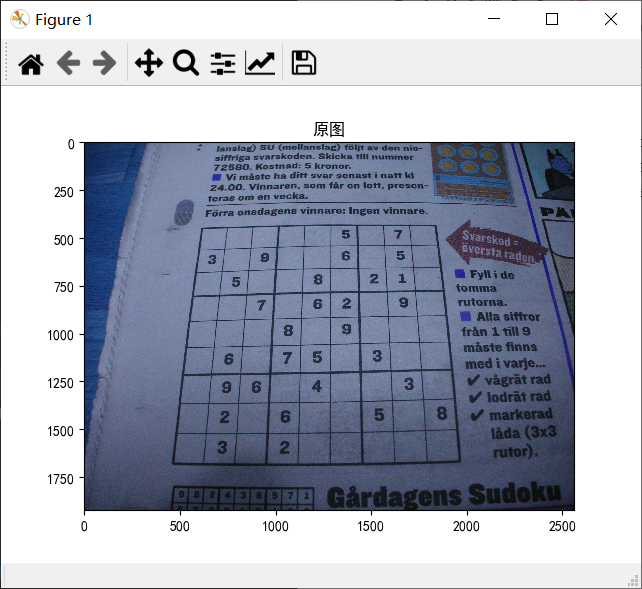


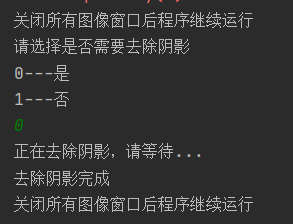


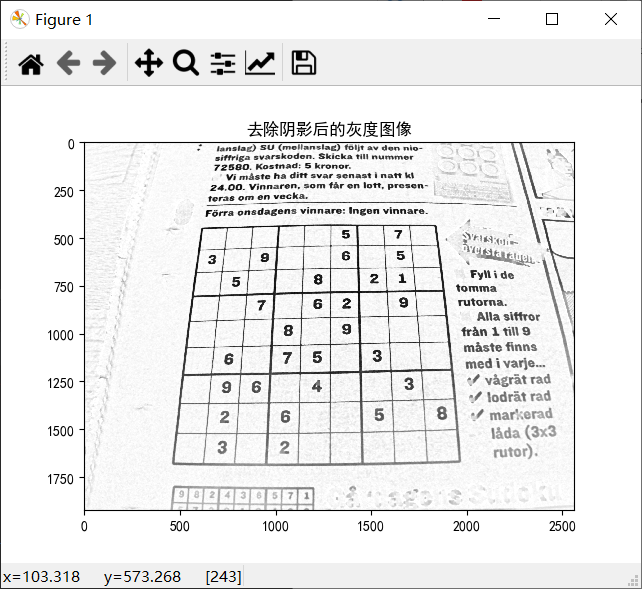


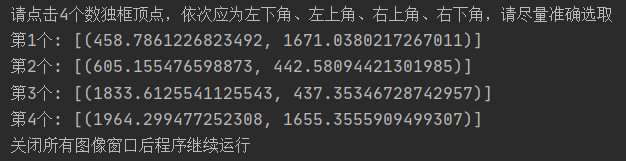
**手动提取：**

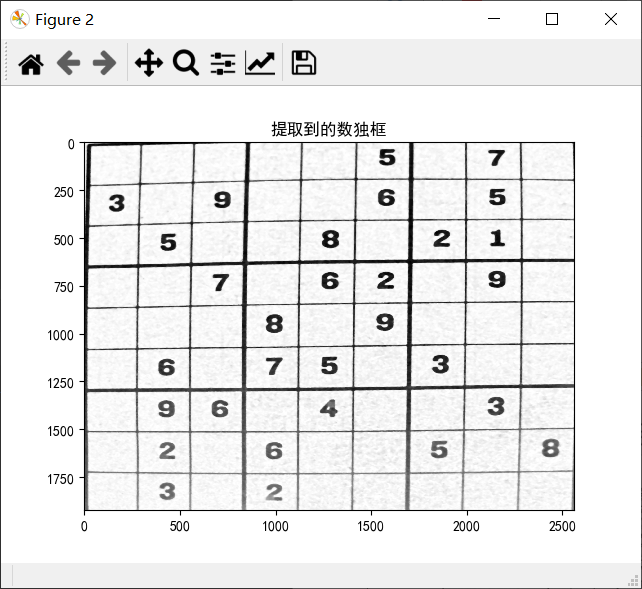
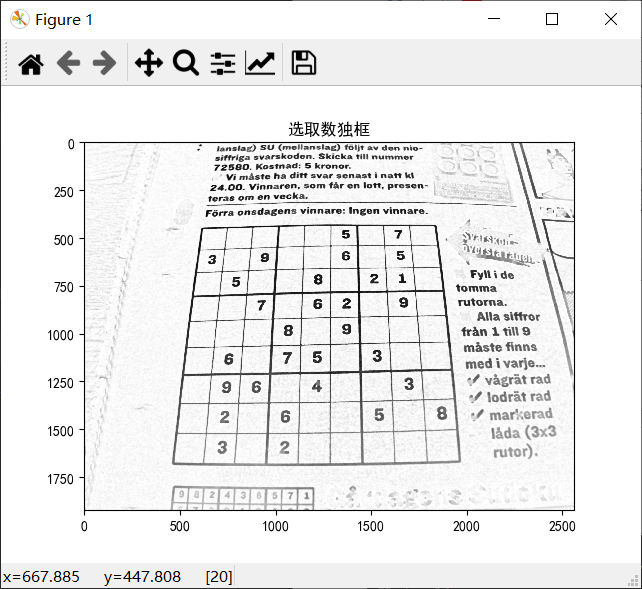


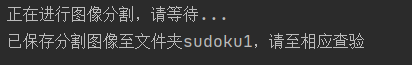














3 图像识别

3.1 图像预处理

原始数据集中的图像具有以下明显的特点：图像存在深浅不一的黑边，背景深度不同，图像对比度相差较大。这些特点会影响CNN网络对于图像特征的提取，容易导致过拟合或欠拟合现象的发生，因此，我们采用如下方法进行图像预处理。

1. 将图像缩放到大小，删除图像边缘的四个像素，进行图像黑边的滤除；
2. 将图像二值化，背景为黑色，数字为白色，对图像进行归一化，消除背景深度和对比度对CNN网络的影响。

具体实现代码如下所示。

img = cv2.imread(imname,cv2.IMREAD\_GRAYSCALE)

*# 去除黑边*

img = cv2.resize(img,(36,36))

img = img[4:-4,4:-4]

*# 参考minst数据集，进行二值处理并黑白反转*

ret, img = cv2.threshold(img,100,255,cv2.THRESH\_BINARY\_INV)

3.2 图像增强

由于数据集图像较少，使用数据增强可以增加数据集容量，优化模型鲁棒性。我们对图像进行随机旋转，水平平移，数值平移，错切变换，放大等操作，将一张原始图像变换得到5张新图像。使得训练集容量破1w张。

我们使用keras深度学习工具包对图像的数据增强部分进行具体实现，引入keras.preprocessing.image.ImageDataGenerator并完成参数定义，具体代码如下。

datagen = ImageDataGenerator(

rotation\_range = 5, *# 随机旋转度数*

width\_shift\_range = 0.1, *# 随机水平平移*

height\_shift\_range = 0.1, *# 随机竖直平移*

shear\_range = 5, *# 随机错切变换*

zoom\_range = 0.1, *# 随机放大*

fill\_mode = 'nearest', *# 填充方式*

)

3.3 CNN网络

使用卷积神经网络对数据集进行训练，定义如下网络结构。

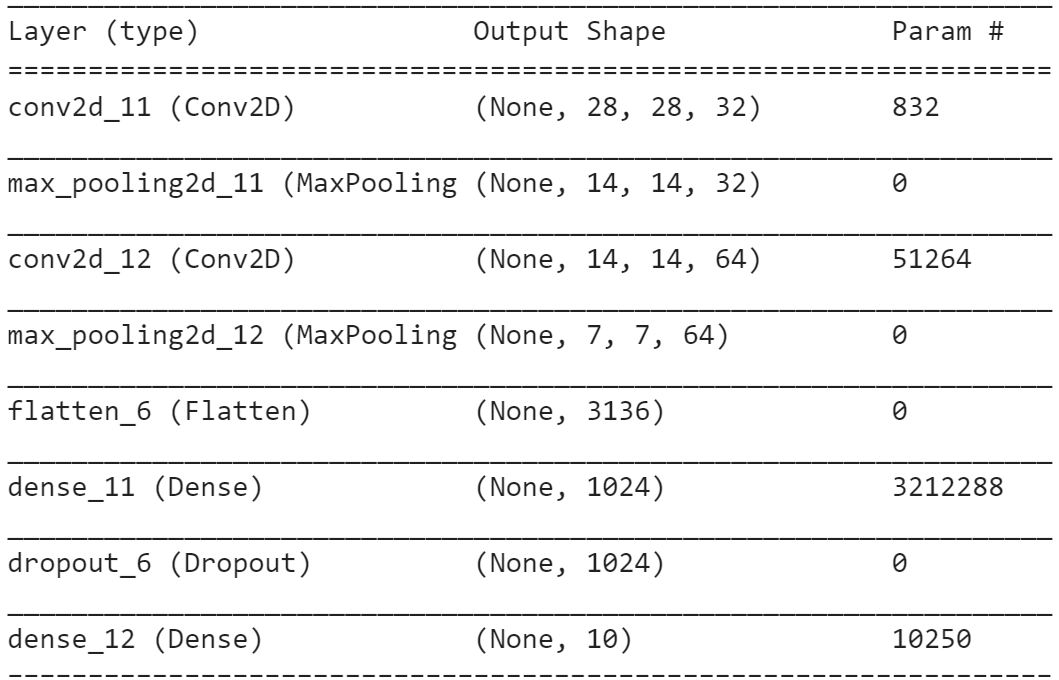


图1.3.1 神经网络结构

第一层网络为卷积层，使用32个的卷积核对特征进行初步提取，使用same padding的方式保证边缘效果，利用relu函数进行激活。

第二层为池化层，池化，步长为2，池化方法为平均池化。

第三层为卷积层，使用64个的卷积核对特征进行进一步提取，此时，卷积核的感受野大于第一层网络的卷积，对特征可以进行更深层次的提取。和第一层网络一样，使用same padding的方式保证边缘效果，利用relu函数进行激活。

第四层为池化层，和第二层作用一样。

第五层为Flatten，将池化结果resize到一维向量。

第六层为全连接，将CNN的结果进行初步提炼。

第七层为dropout，训练时随机中断一些神经元的连接，这种方法可以有效的防止模型过拟合，提高鲁棒性。

第八层为全连接层，利用softmax激活函数输出one-hot编码结果。

在训练过程中，利用adam优化器进行训练，利用categorical\_crossentropy作为loss function。

3.4 模型训练

定义batch\_size=64，对数据集进行10次训练，得到在训练集上准确性99.82%，在测试集上准确性100%。有效的提高了预测精度。

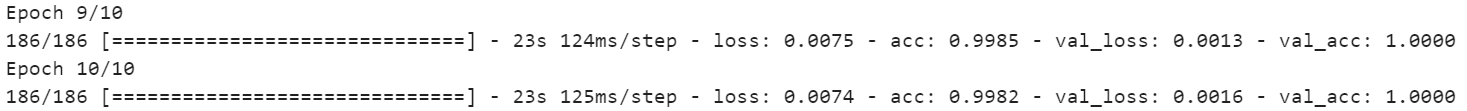


图1.4.1 训练结果

以第一个数独为例，得到如下结果

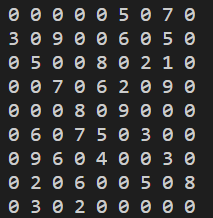


图1.4.1 识别结果

4. 数独求解

通过回溯算法进行求解，具体思路如下：

1. 求出每个数字为0的位置可以填的数，并将其位置和能填的数分别以key和value的方式存储到字典里面；
2. 将字典里的数据按照所能填写的数据的多少进行排序，先在能填的数少的里面选取一个进行填写
3. 将填写的过程记录到列表里面，这个地方暂时想到的就是用列表记录填写过程
4. 更新1、2步，若出现某一步可填写的数据为空，说明之前某一步的选择有问题，回溯回去，更换数值，然后回到步骤1
5. 当所有的数都填完后，退出循环

当图像识别不准的时候，会引起数独无法求解，故对求解过程进行超时判断，当求解时间超过2s时，抛出异常。

以第一个数独为例，有如下求解结果。

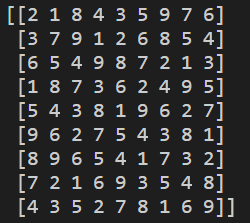


图4.1 数独求解