2022.07

计算机视觉期末作业

视频目标检测、跟踪、分割

黄元通 20195063

目录

视频目标检测、跟踪、分割

姓名：黄元通 学号：20195063

1 概要

1.1 编程环境

Python 3.9；
安装 opencv-python、opencv-contrib-python、numpy、scikit-image 库。

1.2 概要

程序实现了单目标检测、跟踪、分割，主要包括“目标分割”、“目标跟踪”、“分割与跟踪的结合”三大部分。

目标分割部分实现了利用背景模型进行前景分割、计算轮廓并转换为链码表示，实验并测试了高斯混合模型分离算法（MOG2）、基于 K 近邻的背景分割算法（KNN）两种背景模型。

目标跟踪部分实现利用追踪器跟踪目标、绘制运动轨迹，测试了最小平方误差跟踪器（MOSSE）、通道和空间可靠性跟踪器（CSRT）两种模型，并根据其各自优劣势在不同场景中进行使用。

分割与跟踪的结合部分探究了根据跟踪结果定位分割目标、跟踪器失效时的定位方案等方面问题。

2 具体内容

2.1 目标分割

2.1.1 利用背景模型进行前景分割

1. 背景模型

OpenCV 4.4 中共提高两种背景模型：高斯混合模型分离算法（MOG2）、基于 K 近邻的背景分割算法（KNN）。

均包含以下三个参数：

(1) history：用于建模的历史帧数。越大则使用越多的帧数来计算判断某像素当前帧是否属于背景，同时静止的物体运动后原处判断结果恢复为背景的所需时间也越长；

(2) varThreshold / dist2Threshold：判断某像素当前帧是否属于背景时使用的阈值；其他条件相同时，越小则越容易判断为前景，越大越容易判断为背景；

(3) detectShadows：为 True 时表示进行阴影检测。阴影区使用的像素值为 127（背景为 0，前景为 255），会降低模型的运行速度。

其他条件相同时（history=200, detectShadows=False），阈值均为二值默认值，KNN 和 MOG2 背景模型的分割效果如图 1.1 所示。

图 1.1 KNN（左）与 MOG2（右）分割效果对比

左图中 KNN 的分割到的前景更加完整（召回率更高），且被错认为是前景的背景也更少（假阳性率更低）。因此最终使用 KNN 背景模型，三个参数值分别使用：200、400（默认值）、False。

2. 分割模块流程

分割过程为 Source.py 文件中的 Segment() 函数，参数 _frame、_background_model 分别为当前帧原始图片、背景模型。主要进行：更新背景模型，处理并返回分割出的前景图片。

程序流程如下：

(1) 计算视频的前景。先将_frame 经过 GaussianBlur，以降低噪声的影响；然后调用背景模型的 apply 函数，即可获得当前帧的分割结果（因为未开启阴影检测，所以结果为二值图像）；

(2) 滤波去噪、二值化。对分割结果使用中值滤波 medianBlur 以降低噪声，而且经实验对比，medianBlur 比 GaussianBlur 可更好地去除孤立点；然后使用 threshold，手动设定分割阈值为 200（当不进行阴影检测且只使用中值滤波时图像已是二值图像该步骤无影响）；

(3) 进行形态学改变调整。主要包括腐蚀（若指定范围内有像素为 0，则该像素置为 0）、膨胀（若指定范围内有像素为 255，则该像素置为 255）、去除孤立点（面积过小连通域）、填补面积过小的孔等操作；morphologyEx 进行开运算时（MORPH_OPEN）表示先腐蚀后膨胀，进行闭运算时（MORPH_CLOSE）表示先膨胀后腐蚀；去除孤立点和填补孔使用的是 skimage. morphology 库中的 remove_small_objects、remove_small_holes 函数。

这样，就完成了前景分割及处理，对比效果如图 1.2 所示。

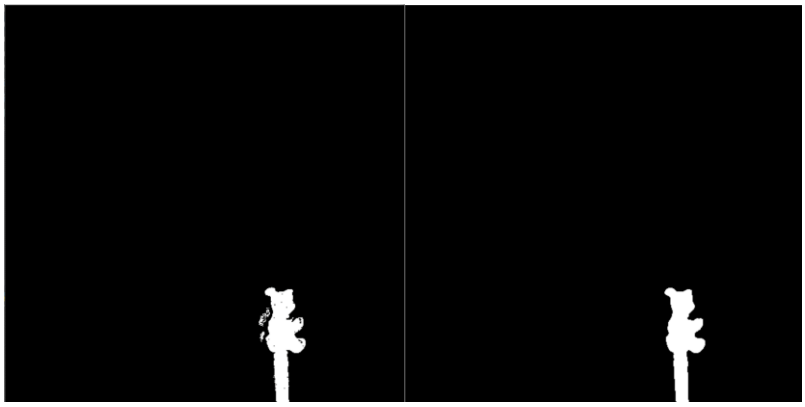


图 1.2 前景处理前（左）、处理后（右）对比

处理后前景区域内的孔被成功去除，且较小的误分割区域也被去除，而某些情况下（如边缘形状较复杂），morphologyEx 开运算会导致边缘很细微的轮廓丢失（向内丢失）、闭运算会导致假阳性区域增加（向外增加），这是腐蚀和膨胀不可避免的权衡选择。

2.1.2 计算轮廓并转换为链码

1. 计算轮廓

该过程为 Source.py 文件中的 Contour_inChainCode ()函数，参数_frame、_segmented 分别为当前帧原始图片、经 Segement()函数得到的二值化（前景）分割图。主要进行：根据分割图片计算轮廓，并转换为链码形式返回。

由分割图计算轮廓使用的是 OpenCV 中的 findContours 函数，主要参数有 mode、method，其含义分别为：

(1) mode：计算轮廓时使用的寻找模式。

1. RETR_LIST：寻找所有轮廓，且互相之间不建立等级关系；
2. RETR_EXTERNAL：只找外边界；
3. RETR_CCOMP：提取所有轮廓，并将轮廓组织成双层结构，顶层为连通域的外围边界，次层位内层边界；
4. RETR_TREE：提取所有轮廓，轮廓间建立树状关系；

(2) method：轮廓的表示方式。

1. CHAIN_APPROX_NONE：储存所有点（绝对坐标）；
2. CHAIN_APPROX_SIMPLE：压缩水平方向，垂直方向，对角线方向的元素，只保留转折点（绝对坐标）。

因为分割图已经失去了前景的内部信息，只含有边缘信息，所以不同寻找模式下找到的结果都相同，而又需要转换为链码所以需要所有点的信息。因此 mode、method 取值分别为 RETR_EXTERNAL、CHAIN_APPROX_NONE。

2. 转为链码表示

按照课程规定，在正常坐标系中，链码应该按照逆时针方向编码，即图 1.3 所示。

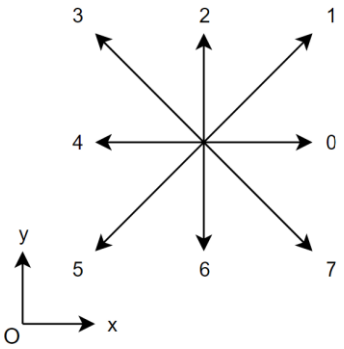


图 1.3 正常坐标系下链码的编码方向，逆时针

而 findContours 得到的是轮廓上所有点在 OpenCV 坐标系下（坐标原点在左上角，x 轴正方向向右，y 轴正方向向下）的绝对坐标，所以编码时(1, 0), (1, 1), …, (1, -1)等各方向应依次表示为 0, 1, …, 7，即图 1.4 所示顺时针方向，这样得到的链码在正常坐标系下便是逆时针方向了。

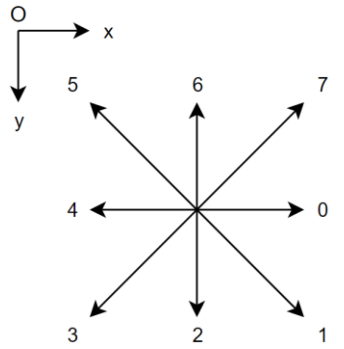


图 1.4 OpenCV 坐标系下链码的编码方向，顺时针

使用 numpy.diff 函数计算得到轮廓上第 i 与 i+1 个元素的差值得到 difference，然后使用 dic.get 遍历 difference 便可将差值表示转化为链码表示，值得一提的是遍历使用 map 表达式的效率最高，高于列表迭代、循环等的效率（循环的效率最低），所以在遇到遍历的情况时可以尽量使用 map 函数以提高运行效率。

2.2 目标跟踪

2.2.1 利用追踪器跟踪目标

1. 追踪器

OpenCV 在拓展包 opencv-contrib-python 中共提供 8 个追踪器，较常使用的共三个分别为：MOSSE、KCF、CSRT，三者在视频质量较高、较稳定时跟踪效果依次变好，而跟踪时间也依次增加。实验中发现：

(1) 在有遮挡的情况下：KCF 表现为三者最差，MOSSE 也较容易丢失目标，CSRT 跟踪效果很好；

(2) CSRT 需要非常好的初始目标区域选取：需要将目标物体完全框在内，而同时又不能大太多，而且目标需要在区域中心。否则就会跟踪目标失败，反而一直非常稳定地跟踪在初始位置的背景上；

(3) 每秒处理速度：使用 MOSSE 时 60 帧左右，使用 CSRT 时 25 帧左右。

初始目标区域选取过大时 MOSSE、CSRT 跟踪效果对比如图 2.1 所示，目标不在初始选取区域的中心时 MOSSE、CSRT 跟踪效果对比如图 2.2 所示。

图 2.1 初始目标区域选取过大：MOSSE（左）、CSRT（右）

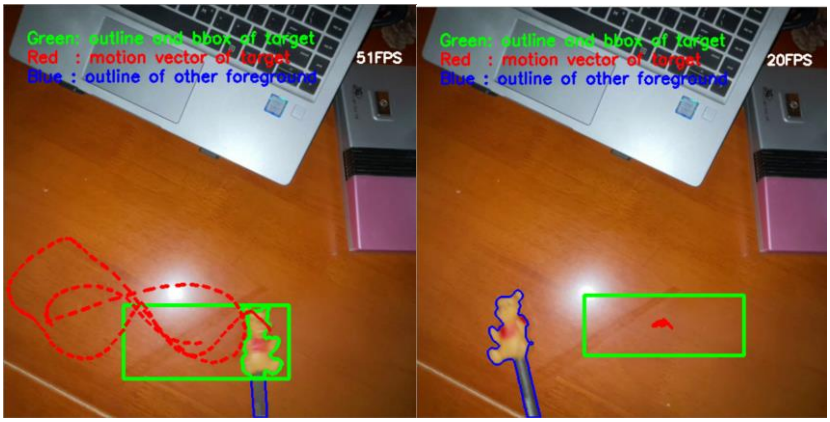


图 2.2 目标不在初始选取区域中心：MOSSE（左）、CSRT（右）

可见 CSRT 仅在初始目标区域选取得非常精准的情况下才能获得比 MOSSE 等跟踪器更好的结果，出现选取过大、目标不在初始选取区域的中心时，MOSSE 甚至能以 3~5 倍的速度获得比 CSRT 准确非常多的结果（某些情况下甚至能完整地全程跟踪上）。

2. 追踪器的选取

根据以上实验结果，设计以下追踪器选取方案：

视频开始时，调用 selectROI：

(1) 如果要跟踪的目标在当前视野内：

1. 则用户框选目标的最小外接矩形，使用该区域作为初始目标区域；
2. 并使用追踪器 CSRT。

(2) 如果不在视野内：

1. 用户按任意键跳过，使用随机一个背景小区域（例如画面中心取 10*10 大小）作为初始目标区域，交由后续程序进行更多调整以找到后期出现的目标；
2. 并使用追踪器 MOSSE。

3. 追踪器的程序实现

程序中，追踪器的实现和使用流程为：

(1) 开始前，调用追踪器的 init(frame, init_location)函数，使用视频第一帧和初始目标区域初始化追踪器；

(2) 开始遍历视频后，调用 tracker.update(frame)即可使用追踪器在画面内进行跟踪，其返回结果为 succeed、bbox 分别表示是否追踪成功、（追踪成功时）目标的定位框；

(3) 如果追踪成功，则将 bbox 中 4 个值取出转为 int，分别表示定位框的左上角的 x 坐标、左上角的 y 坐标、x 方向宽度、y 方向高度；
如果追踪失败返回 None 交由后续程序处理。

2.2.2 绘制运动轨迹

使用追踪器返回的定位框的中心代表目标位置。因为每一帧中由背景模型计算出的目标轮廓可能会由于光线、角度等原因产生变化，而定位框（在追踪器较好地跟上目标的情况下）与目标的相对位置较为固定，因此作者认为追踪器得到的定位框中心比背景模型得到的轮廓中心更接近物体的运动轨迹。

运动轨迹存储及绘制流程为：

(1) 使用 int(x + w / 2)和 int(y + h / 2)分别计算出中心的 xy 坐标得到目

标物体位置 location；

(2) 然后将 location 添加到 location_list 中，储存所有历史时刻物体的位置；

(3) 遍历 location_list，第 i-1 与 i 个元素的连线就表示物体在第 i 时刻物体的运动轨迹；而该运动距离与当时的速率成正比，且方向就是运动方向，所以可以转化为第 i 时刻的速度矢量图（有方向和长度的无箭头线段）。程序具体实现时采用的是使用该线段的 1/2 表示速度矢量，使用 cv2.line 绘制第 i-1 点的位置、到第 i-1 点和 i 点中点位置的连线。

2.3 分割与跟踪的结合

2.3.1 利用跟踪结果定位分割目标

视频画面中难以避免地会出现存在多个前景物体的情况。从分割结果仅能分辨出不连通前景为不同物体，但无法判断各个轮廓分别对应上一帧中的哪一个轮廓。

因此保存到链码记录 chain_code_list 中的链码，只使用位于追踪器定位框中的分割结果来进行计算，即：仅有位于定位框中的前景才算做为目标的轮廓。例如本实验使用的样本视频中，连接目标玩偶的笔杆、作为遮挡物出现的手，虽然都是前景，但均不属于目标。

实现方法为：生成定位框内值为 255、定位框外值为 0 的、与视频长宽相同的蒙板矩阵 mask；然后使用(mask & segmented)传入计算轮廓及链码的 Contour_inChainCode 函数中。

核心代码如下：

```
mask = numpy.zeros_like(segmented)
mask [y:y + h, x:x + w] = 255
chain_code = Contour_inChainCode(frame, mask & segmented)
```

2.3.2 跟踪器失效时的定位方案

1. 恢复定位

上文在“追踪器选取方案”中提到：跟踪目标不在初始画面内时，需要“交由后续程序进行更多调整以找到后期出现的目标”；在“追踪器的程序实现”中：“如果追踪失败返回 None 交由后续程序处理”。即没有目标、或目标跟踪失败时，如何正确定位目标位置。

本程序中的处理方法是，令两种情况均转化为追踪器跟踪失败，然后使用分割图中得到的、与上一时刻目标所在位置最近的、前景轮廓的位置作为此时目标的位置。此方法基于以下两点假设偏置：

(1) 进行单目标跟踪；

(2) 根据实际物理情况，i+1 时刻，与 i 时刻目标所在位置最近的前景就是目标。

程序实现 main.py 函数 Relocate()，作用是：跟踪失败时，使用与上一时刻定位框最近的轮廓作为目标定位。其主要流程与 Contour_inChainCode 大致相同，主要核心区别是从 findContours 算出的多个 contours 中选取出目标 contour 的准则不同。

排序使用 min 函数，准则（参数 key）由 Source.py 中函数_distance()生成，该函数返回一个计算与给定 previous_bbox 的棋盘距离的函数，该过程主要流程为：

(1) _distance 从外部接收上一时刻定位框 previous_bbox，作为局部变量；

(2) 内部定义函数_distance1 参数为从外部接收一个轮廓 contour，使用 boundingRect 获取其外接矩形，然后通过以下格式计算 contour 与局部变量 previous_bbox 间的距离（棋盘距离）：

$$distance = \sum_{i\in\{x,y,w,h\}} \left|i_{contour} - i_{previous-bbox}\right|$$

(3) 由_distance 将该函数_distance1 返回即可。

通过此排序可获得当前所有前景中，与前一时刻定位框最近的前景，将其定义为此刻的目标。

进一步，依据此后是继续采用追踪器定位目标、还是只依靠分割来定位目标，有以下两种具体实现方案。

2. 后续定位方案

方案 1：继续采用追踪器

取出分割图中得到的与上一时刻目标所在位置最近的前景轮廓，计算其外接矩形，然后生成新的追踪器（MOSSE），并使用此 bbox 进行初始化。

但因为最初几帧时没有历史信息，背景模型会将画面绝大部分归为前景、或极小部分归为前景，因此若只在追踪器跟踪失败时初始化追踪器会导致从一开始，追踪器就变成追踪整个画面或一个随机的局部，如下图 3.1 所示。

图 3.1 追踪整个画（左）、追踪随机局部（右）

很显然，若后续想要继续采用追踪器，仅在追踪失败后才更新追踪器是不可取的：因此增加一条：每隔固定时间（或帧数）就更新一次追踪器（程序中阈值设定为 30 帧）。另外，因为追踪器初始化过后，再调用其 init 函数就不再有效，所以在 Relocate() 函数中，还需要增加：tracker = cv2.TrackerMOSSE_create()

方案 2：只依靠分割

但因为课程要求时刻进行目标分割，所以实验中发现，这种情况下再采用追踪器并不能提高程序运行速度（追踪器为局部搜索，基于分割的跟踪需要全局搜索，所以单独的追踪器比单独的基于分割的跟踪要快）。所以更合理的方案变成了使用基于分割的跟踪方案。

程序有以下几处删减：

(1) 初始目标区域选取时，若用户未选择，则跟踪器不初始化；

(2) Relocate()函数中删去重新生成跟踪器和跟踪器的初始；

(3) 去掉每隔固定时间（或帧数）就更新一次追踪器的判断条件。

在实验条件下，该方案的运行速度为每秒 60 帧左右。

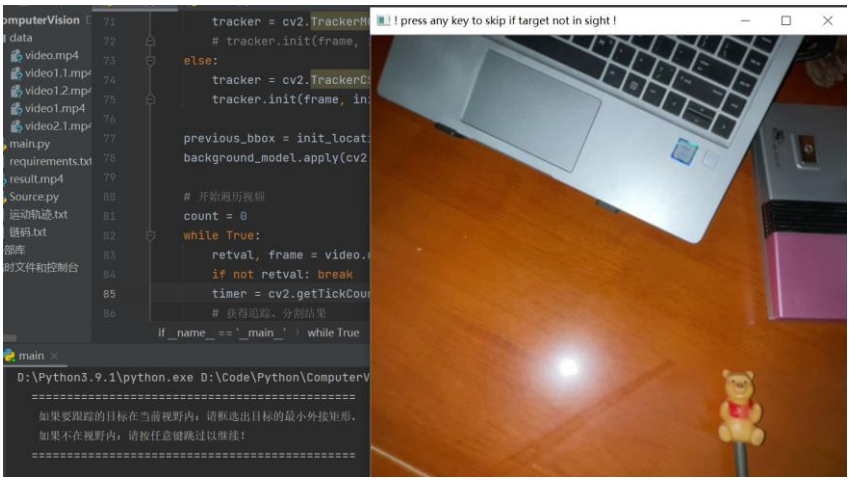
值得注意的是：

若初始目标区域选取时，目标不在视野内，那么无论是方案 1 还是方案 2 都无法以前景的局部作为目标，只能以完整连通域（完整的一个前景物体）为单位进行跟踪；

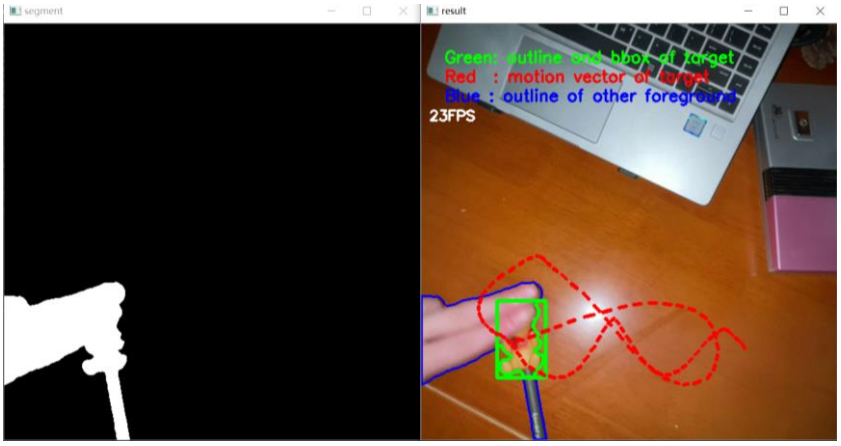
若存在目标物体在视野内形状不断变化的情况（如本实验样例视频中的笔杆出现的长度就是不断变化的），因为目标形状改变会导致追踪器每隔固定时间都出现一次突然的定位框形状改变，所以此情况下方案 2 比方案 1 得到的运动轨迹更加光滑连续。

3 运行结果

初始目标区域选取及提示界面：



运行过程截图。图示为出现遮挡时，仍能精确地追踪：



处理结果以视频形式保存，每一帧中目标边缘链码表示、运动轨迹以文件形式保存：