

计算机视觉作业 -4: 目标跟踪

王天锐 20120318

2021 年 1 月 5 日

摘要

本次实验作业主要是根据老师上课所讲知识,利用 Car_Data 文件夹中的视频序列实现基于 MeanShift(均值漂移) 目标跟踪。待跟踪的目标为场景中的车辆,初始目标位置标定需手工标定,后续帧中的目标位置需通过均值漂移方法得到。

1 方法原理及实现

1.1 Mean Shift

均值漂移算法是属于核密度估计法,它不需要任何先验知识而完全依赖特征空间中样本点的分布。首先是利用直方图法把数据的值域分成若干相等的区间,数据按区间分成若干组,每组数据的个数的占比为每个单元的概率密度估计值。

给定 d 维空间 R^d 中的 n 个样本点 $x_i, i = 1, \dots, n$ 在 x 点的 MeanShift 向量的基本形式定义为:

$$M_h(x) = \frac{1}{k} \sum_{x_i \in S_h} (x_i - x)$$

其中, S_h 是一个半径为 h 的高维球区域, 满足以下关系的 y 点的集合:

$$S_h(x) = \{y : (y - x)^T(y - x) \leq h^2\}$$

k 表示在这 n 个样本点 x_i 中, 有 k 个点落入 S_h 区域中。

可以看到 $(x_i - x)$ 是样本点 x_i 相对于点 x 的偏移向量。根据 MeanShift 向量的定义 $M_h(x)$ 就是对落入区域 S_h 中的 k 个样本点相对于点 x 的偏移向量求和然后再平均。

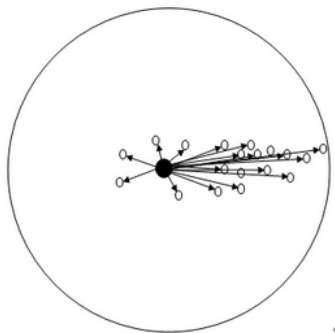


图 1: Mean shift 示意图

如上图所示，大圆圈所圈定的范围就是 S_h ，小圆圈代表落入 S_h 区域内的样本点 $x_i \in S_h$ ，黑点就是 MeanShift 的基准点 x ，箭头表示样本点相对于基准点 x 的偏移向量。可以看出，平均的偏移向量 $M_h(x)$ 会指向样本分量最多的区域。也就是概率密度函数的梯度方向。

1.2 核函数

除开 Meanshift 在本实验还使用了高斯核函数来对窗口进行平滑。核函数的定义为: X 代表一个 d 维的欧式空间， x 是该空间中的一个点，用一列向量表示。 x 的模 $\|x\|^2 = x^T x$ 。 R 表示实数域。如果一个函数 $K : X \rightarrow R$ 存在一个剖面函数 $k : [0, \infty] \rightarrow R$ ，即：

$$K(x) = k(\|x\|^2)$$

并且满足 (1) k 非负。(2) k 是非增。(3) k 是分段连续的，并且 $\int_0^\infty k(r)dr < \infty$ ，那么函数 $K(x)$ 就被称为核函数。在本实验采用的是均值为 0，方差为 1 的高斯核函数，其表达式如下：

$$K(x) = \frac{1}{\sqrt{2\pi}} \exp\left\{-\frac{x^2}{2}\right\}$$

1.3 基于 MeanShift 的目标跟踪法

基于均值漂移的目标跟踪算法通过分别计算目标区域和候选区域内像素的特征值概率得到关于目标模型和候选模型的描述，然后利用相似函数度量初始帧目标模型和当前帧的候选模版的相似性，选择使相似函数最大的候选模型并得到关于目标模型的 Meanshift 向量，这个向量正是目标由初始位置向正确位置移动的向量。由于均值漂移算法的快速收敛性，通过不断迭代计算 Meanshift 向量，算法最终将收敛到目标的真实位置，达到跟踪的目的。如下图所示，目标跟踪开始于点 x_i^0 。箭头表示样本点相对于核函数中心点的漂移向量，平均的漂移向量会指向样本点最密集的方向。通过反复迭代搜索特征空间中样本点最密集的区域，搜索点会沿着样本点密度增加的方向移动，最终到达目的点。

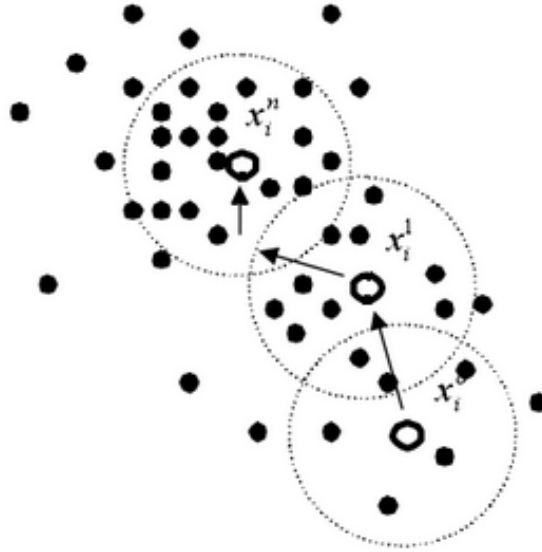


图 2: Mean shift 过程示意图

整体代码实现如下所示:

```
1 def mean_shift(car_frame, top_y, top_x, n_rows, n_cols):
2     nbin = 32
3     h = (n_cols // 2) * 2 + (n_rows // 2) * 2 # 带宽
4     center = [n_rows // 2, n_cols // 2] # 中心点位置
5     dis_k = np.zeros((n_rows, n_cols)) # 距离权重矩阵k
6     dis_g = np.zeros((n_rows, n_cols)) # 距离权重g
7     C = 0 # 权值归一化系数
8     points_top = np.zeros((101, 2)) # 每个图片目标对应的左上角的坐标
9     points_top[0] += np.array([top_x, top_y]) # 将第一帧我们画的放进去
10    # 核函数
11    for i in range(n_rows):
12        for j in range(n_cols):
13            dist = ((i - center[0]) ** 2 + (j - center[1]) ** 2) / h # 计算该点离目标框
14                                中心的距离
15            dis_k[i, j] = gauss(dist)
16            dis_g[i, j] = dis_k[i, j] * dist # 高斯平滑核
17            C = C + dis_k[i, j]
18    frame_hist = np.zeros(nbin)
19    frame_bin = np.zeros((n_rows, n_cols))
20    error = 0
21    # 目标frame分布统计
22    for i in range(n_rows):
23        for j in range(n_cols):
24            frame_bin[i, j] = (car_frame[i, j]) // nbin
25            frame_hist[ int(frame_bin[i, j])] += dis_k[i, j]
26            frame_hist = frame_hist / C
27
28    # 迭代处理图
29    for pic_i in range(2, 101):
30        if pic_i < 10:
31            pic_name = "car00%d.bmp" % pic_i
32        elif 100 > pic_i >= 10:
33            pic_name = "car0%d.bmp" % pic_i
34        else:
35            pic_name = "car%d.bmp" % pic_i
36        path = os.path.join(r"./data/tracking/Car_Data/", pic_name)
37        im_i = cv2.imread(path)
38        im_i = cv2.cvtColor(im_i, cv2.COLOR_BGR2GRAY)
39        # 设置阈值
40        time = 0
41        while time < 10:
42            time += 1
43            # 拿出上一次画出的框
44            car_frame = im_i[top_y:top_y + n_rows + 1, top_x:top_x + n_cols + 1]
45            # 待测frame分布统计
```

```

45     frame_bin_now = np.zeros((n_rows, n_cols))
46     frame_hist_now = np.zeros(nbin)
47     for i in range(n_rows):
48         for j in range(n_cols):
49             frame_bin_now[i, j] = (car_frame[i, j]) // nbin
50             frame_hist_now[ int(frame_bin_now[i, j])] += dis_k[i, j]
51             frame_hist_now = frame_hist_now / C
52     w = np.zeros(nbin)
53     # 相似性计算
54     for i in range(nbin):
55         if frame_hist_now[i] != 0:
56             w[i] = np.sqrt(frame_hist[i] / frame_hist_now[i])
57         else:
58             w[i] = 0
59     w = w / 2
60     coeff = 0
61     wx = [0, 0]
62     # 计算更新方向
63     for i in range(n_rows):
64         for j in range(n_cols):
65             coeff = coeff + dis_g[i, j] * w[ int(frame_bin_now[i, j])]
66             wx = wx + dis_g[i, j] * w[ int(frame_bin_now[i, j])] * np.
                array([i - center[0], j - center[1]])
67     delta = wx / coeff
68     top_x = int(top_x + delta[1])
69     top_y = int(top_y + delta[0])
70     # 更新
71     if delta[0] ** 2 + delta[1] ** 2 == error:
72         break
73     else:
74         error = delta[0] ** 2 + delta[1] ** 2
75     points_top[pic_i - 1] += np.array([top_x, top_y])

```

2 实验结果及分析

2.1 序列帧展示

下图是图片序列的第一张图片，我们的目标是全程跟踪里面的小车。而且由于是车在路上跑，所以整个图片的像素分布理论上不会出现非常大的变化。



图 3: 第一帧图片

2.2 目标选取

这里利用 opencv 的方法为用户提供图片选取的操作，如下所示。



图 4: 目标选择

选取好目标后，则利用 MeanShift 算法进行跟踪。如下展示了部分过程的结果。

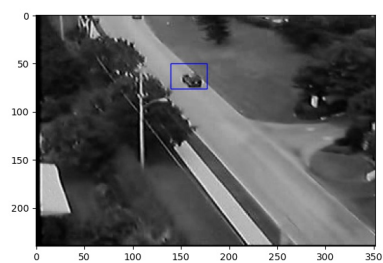


图 5: 准确跟踪

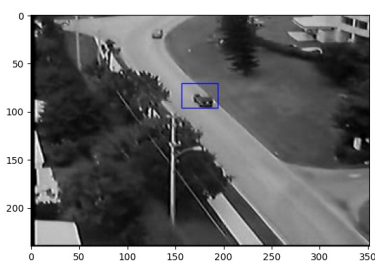


图 6: 准确跟踪

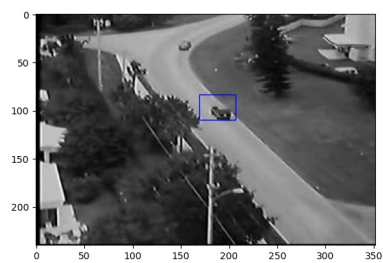


图 7: 准确跟踪

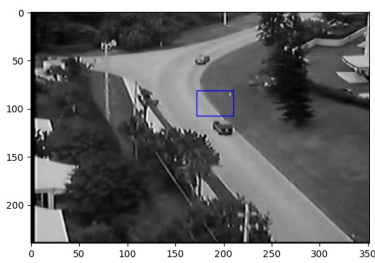


图 8: 开始偏移 (驶入弯道)

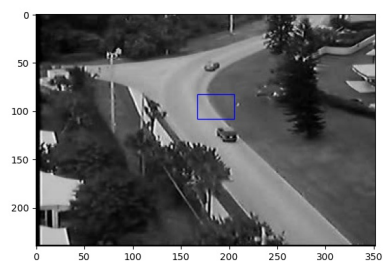


图 9: 完全偏移



图 10: 完全偏移

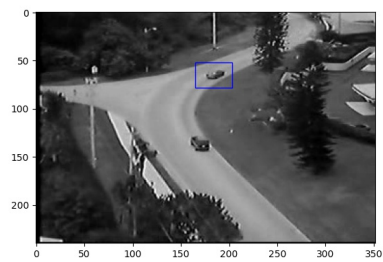


图 11: 跟踪到第二辆车



图 12: 跟踪到第二辆车

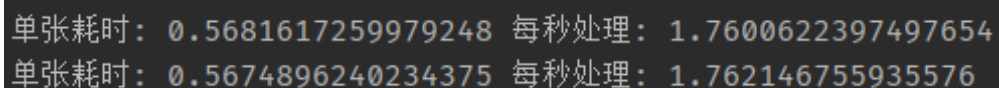
从结果来看，刚开始在直线路段上跟踪效果还比较理想。但是一进弯道后发现跟踪的目标框开始

超前我们的目标小车，最后跟踪到前面那辆小车上。这里分析原因是因为，在直线路段上一个 frame 里的像素值分布不会发生过大的变化，并且均值漂移向量方向比较单一。但是到了弯道上，每个 frame 上像素点的分布变化开始变大，使得均值漂移向量出现问题。导致跟丢，并且往拐弯的方向（右上）开始跑，发现第二辆车后，其正好进入了那段直线路段，所以就跟上第二辆车了。

总结发现，MeanShift 算法由于其使用的是直方图特征统计，所以会严重受影响于图像的像素分布。如果 frame 环境发生较大变化会导致跟踪效果并不理想。缺少了一定的空间信息。并且再加上后一帧是依赖于前一帧的处理结果进行进一步判断，导致一步错步步错的情况出现。

2.3 实时性分析

下图是实验算法实时性测试。



```
单张耗时: 0.5681617259979248 每秒处理: 1.7600622397497654
单张耗时: 0.5674896240234375 每秒处理: 1.762146755935576
```

图 13: Meanshift 耗时情况

可以看到在商务笔记本算力情况下，该算法的处理速度是稍微有点慢的，每帧图片需要耗时长半秒钟。远远小于人眼的处理速度。这里分析原因是因为本身算法的循环较多，图片维度如果比较大的话，会导致时间复杂度指数级增长，导致算法耗时过多。

3 总结

经过这次实验作业，我更加深刻地理解了图像跟踪的相关知识。对 MeanShift 算法进行了复现和实验。通过实验将上课所学的知识应用到了实践中去。通过实验加深了一些课堂上理论的理解。还让我发现了 MeanShift 的一些问题所在。也开发了我的自主探索的意识。非常感谢朱老师给我这次实验的机会。