计算机视觉作业5: 背景建模

唐麒 21120299
qitang@bjtu.edu.cn

摘要—在运动目标检测提取中，背景目标对于目标识别和跟踪至关重要。图像变化的原因一般分成相机运动和相机固定。前者一般用光流法，后者一般用背景建模法。基于高斯混合模型的背景建模适合于在相机固定的情况下，从图像序列中分离出背景和前景。其在物体具有重复性运动的情况下鲁棒性较好，比如微风树叶抖动。本文将对基于高斯混合模型的背景建模方法进行介绍，其效果及实时性分析在实验部分给出。

关键词—计算机视觉、背景建模、高斯混合模型

1 简介

基于视频的运动目标检测是将视频序列图像中的变化区域从场景中分割出来，其中，视频序列图像中变化的区域被称为前景，而其他区域被称为背景。运动目标检测是对运动目标进行行为理解和分析的前提和基础，在智能监控、用户交互等领域有着广泛的应用前景。

传统的运动目标检测方法主要包括帧间差分法、背景差分法和光流法。其中，因受计算复杂、抗噪性能差且不利于实现等条件的制约，光流法一般不适用于视频序列图像中的运动目标检测。而基于差分的方法，帧间差分法比较容易收到运动目标的速度影响，无法对运动速度过快或过慢的目标进行较好的检测。而在检测较大面积的运动物体时，易出现目标“空洞”等问题，不利于下游任务。与帧间差分法相比，背景差分法获得的运动目标信息更完整，位置更准确，且实现简单。

背景差分法是通过判断当前帧和背景帧的对应像素点的变化程度来判断当前像素是否属于运动目标。如果对应的像素点变化值小于设定的阈值，则该像素背标记为背景像素，否则为前景像素。从理论上来看，只要能有合适的背景和适当的图像处理，运动目标就可以被较快且较为准确的检测并分割出来。

但在实际应用中，由于背景差分法对背景的依赖非常大。一个朴素的背景建模方法是对在一段时间内的图像进行平均，创建与当前静态场景相似的背景近似值。虽然这在对象连续移动并且背景在大部分时间可见的情况下是有效的，但是对于具有许多运动目标的场景，尤其是当它们移动缓慢时，这种背景建模方法是不鲁棒的。

从统计的角度看，图像中每个像素点的值在短时间内都是围绕某一中心值一定距离的分布。通常，中心值可以用均值来代替，距离则可以用方差来代替。如果数据点足够多的话，是可以说这些点呈高斯分布。由此，若像素点的值偏离均值较远（超出一定方差范围），那么该像素属于前景，否则认为该点属于背景。

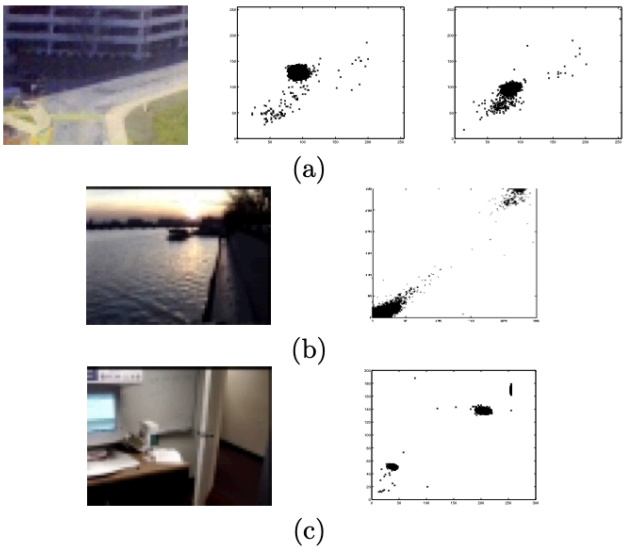

图 1: 视频序列中某一帧图像和图像中单个像素的值随时间变化的散点图：(a) 来自相同像素相隔 2 分钟的散点图；(b) 由水面的镜面反射造成的像素值的两个分布；(c) 监视器闪烁导致的像素值的两个分布.

但场景中光照的变化会给这种背景建模方法带来挑战。如图 1 所示，如果图像中的每个像素值都是在特定光照条件下的特定表面产生的，则在考虑图像采集噪声的同时，单个高斯模型将足以对该像素值建模描述。如果只是光照随时间变化，每个像素用单个自适应高斯建模也足够了。然而实际上，图像某一位置的像素值可能来自于多个表面（如树叶的抖动），并且光照条件也会发生改变。因此，单高斯模型只能描述背景的单一模式，当背景表现为树叶晃动或水面反射的光照变化等多模态形式时就容易出错。这就需要使用自适应高斯混合模型来对背景进行建模。

本文在视频帧序列实现基于高斯混合模型的背景建模，分

别采用了手动实现和调用 OpenCV 的函数两种方式。本文剩余部分将对高斯混合模型的背景建模方法的原理、实现及实验进行介绍，所有实验涉及的方法、函数实现均基于 Python 语言。

2 方法

用高斯混合分布对背景建模的基本思想是把每一个像素点所呈现的像素值用多个高斯分布的叠加来表示。高斯混合分布之所以能够将前景和背景分开是基于如下事实，在长期观测的场景中，背景占大多数时间，更多的数据是支持背景分布的。

视频序列图像的每个像素都由多个单高斯模型描述:

$$\begin{aligned} Pr(x_t) &= \sum_{i=1}^{K} w_{i,t} \cdot N\left(X_t; \mu_{i,t}, \sum_{i,t}\right) \\ \sum_{i,t} &= \sigma_k^2 I \end{aligned} \tag{1}$$

K 表示高斯混合模型中单模型的数量，一般在 3 到 5 之间; $N\left(X_t; \mu_{i,t}, \sum_{i,t}\right)$ 即为单高斯模型，$w_{i,t}$ 表示每个模型的权重。

代码实现如下:

```
# 图像GMM模型定义
class GmmModel:
    def __init__(self, sample_image):
        # 像素点个数
        self.img_size = sample_image.shape[0] *
            sample_image.shape[1]
        # 各像素点的模型个数 （初始化为0）
        self.model_count = np.zeros([1, self.
            img_size], int)
        #GMM高斯模型的个数 K （这里是固定的，有些
            方法可以对每个像素进行自适应模型个数K
            的选取）
        self.k = 4  # 可调整的参数
        # 学习率 Alpha
        self.alpha = 0.005  # 可调整的参数
        # SumOfWeightThreshold T
        self.t = 0.75  # 可调整的参数
        # 各个模型的权重系数（初始化为0）
        self.w = np.zeros([self.k, self.img_size])
        # 各高斯模型的均值（初始化为0）
        self.u = np.zeros([self.k, self.img_size])
        # 各高斯模型的标准差（初始化为默认值）
        self.sigma = np.full([self.k, self.
            img_size], SIGMA)
```

main.py

首先初始化预先定义的 K 个高斯模型，对高斯模型中的参数进行初始化，并求出之后要用到的参数。利用高斯混合模型进行运动目标检测，并对模型的参数进行更新，步骤如下:

1) 第一步: 如果新读入的图像序列中的图片在 (x, y) 处的像素值对于 $i = 1, 2, \ldots, K$ 满足 $|I(x, y, t) - \mu_i(x, y, t)| \leq \lambda\sigma_i(x, y, t)$，$\lambda$ 一般取 2.5, 则新像素点与该单模型匹配。如果存在与新像素点匹配的单模型，则判断为背景，并进入第二步; 如果不存在则判断为前景，并进入第三步。
2) 第二步: 修正与新像素匹配的单模型的权值，新的权值表示如下:

$$w_{i,t} = (1 - \alpha)w_{i,t} + \alpha \tag{2}$$

并更新所匹配到的单模型均值和方差，更新方法如下:

$$\begin{aligned} \mu_k &= (1 - \rho)\mu_{k-1} + \rho X_t \\ \sigma_k^2 &= (1 - \rho)\sigma_{k-1}^2 + \rho (X_t - \mu_k)^T (X_t - \mu_k) \\ \rho &= \alpha N (X_t; \mu_k, \sigma_k) \end{aligned} \tag{3}$$

3) 若该像素不与任何一个单模型匹配，则:1) 如果当前单模型的数目已经达到允许的最大数目，则去除当前单模型集合中重要性最小的单模型; 2) 增加一个新的单模型，新的单模型的权重为一个较小的值，均值为新像素的值，方差为给定的较大的值，并对权重进行归一化。

$$w_{i,t} = \frac{w_{i,t}}{\sum_{j=1}^{K} w_{i,t}}, (i = 1, 2, \cdots, K) \tag{4}$$

由高斯混合模型进行背景建模的基本思想可知，背景模型具有以下特点:(1) 权重大（先验大）: 背景出现的频率高;(2) 方差小: 像素值变化不大。因此，可以根据

$$\frac{w}{\sigma} = \frac{w_{i,t}}{\sum_{i,t}} \tag{5}$$

作为重要性排序的依据。排序及删减过程如下: (1) 计算每个单模型的重要性值 $\frac{w}{\sigma}$。(2) 对于各个单模型按照重要性的大小进行排序，重要性大的排在前面。(3) 若前 N 个单模型的权重满足 $\sum_{i=1}^{N} w_{i,t} > T$，则仅用这 N 个单模型作为背景模型，并删除其他的单模型。

代码实现如下:

```
# 训练模型参数
def gmm_model_train(gmm_model, single_frame):
    # start_time = time.time()
    img_rows = single_frame.shape[0]
    img_cols = single_frame.shape[1]
    for m in range(img_rows):
        for n in range(img_cols):
```

```
            # 用于标识是否存在与像素点(m,n)匹配的分布模型
            matched = False
            for k in range(gmm_model.model_count[0, m * img_cols + n]):
                # 计算像素点与高斯分布均值的差值
                difference = abs(single_frame[m, n] - gmm_model.u[k, m * img_cols + n])
                distance = difference * difference
                # 当前像素点满足当前高斯分布
                if difference <= 2.5 * gmm_model.sigma[k, m * img_cols + n]:
                    matched = True
                    # 更新第k个高斯分布模型的参数
                    # 计算第k个高斯分布在该像素点上的概率密度
                    prob = (1 / (2 * np.pi * gmm_model.sigma[k, m * img_cols + n] ** 2) ** 0.5) * np.exp(
                        -(single_frame[m, n] - gmm_model.u[k, m * img_cols + n]) ** 2 / (
                            2 * (gmm_model.sigma[k, m * img_cols + n] ** 2)))
                    p = gmm_model.alpha * prob
                    # update weight
                    gmm_model.w[k, m * img_cols + n] = (1 - gmm_model.alpha) * gmm_model.w[
                        k, m * img_cols + n] + gmm_model.alpha
                    # update mean
                    gmm_model.u[k, m * img_cols + n] = (1 - p) * gmm_model.u[k, m * img_cols + n] + p * single_frame[
                        m, n]
                    # update standard deviation
                    if gmm_model.sigma[k, m * img_cols + n] < SIGMA / 2:
                        gmm_model.sigma[k, m * img_cols + n] = SIGMA / 2
                    else:
                        gmm_model.sigma[k, m * img_cols + n] = ((1 - p) * gmm_model.sigma[
                            k, m * img_cols + n] ** 2 + p * distance) ** 0.5
                    break
                else:
                    # 当前像素点不满足当前高斯分布
                    # 只更新weight
                    gmm_model.w[k, m * img_cols + n] = (1 - gmm_model.alpha) * gmm_model.w[k, m * img_cols + n]
                # 对k个gmm_model进行排序，便于后面高斯分布模型的替换和更新
                gmm_model_sort(gmm_model, m, n, img_cols)
            '''
            # 当前像素点未匹配到任何一个高斯分布，则需要新建一个高斯分布
            # 这里需要考虑两种情况
            # 1. 存在未初始化的高斯分布：此时可新建一个高斯分布
            # 2. k个高斯分布均已被初始化：此时替换order_weight最低的分布模型
            '''
            if not matched:
                # print('(', m, ',', n, ')', 'no matching distribution')
                # condition 1
                model_count = gmm_model.model_count[0, m * img_cols + n]
                if model_count < gmm_model.k:
                    # 初始化weight
                    gmm_model.w[model_count, m * img_cols + n] = WEIGHT
                    # 初始化mean
                    gmm_model.u[model_count, m * img_cols + n] = single_frame[m, n]
                    # 初始化standard deviation
                    gmm_model.sigma[model_count, m * img_cols + n] = SIGMA
                    gmm_model.model_count[0, m * img_cols + n] = model_count + 1
                # condition 2
                else:
                    # update weight
                    gmm_model.w[gmm_model.k - 1, m * img_cols + n] = WEIGHT
                    # update mean
                    gmm_model.u[gmm_model.k - 1, m * img_cols + n] = single_frame[m, n]
                    # update standard deviation
                    gmm_model.sigma[gmm_model.k - 1, m * img_cols + n] = SIGMA
            # weight归一化
            # 加上此判断条件，运行速度快了很多
            if sum(gmm_model.w[:, m * img_cols + n]) != 0:
                gmm_model.w[:, m * img_cols + n] = gmm_model.w[:, m * img_cols + n] / sum(
                    gmm_model.w[:, m * img_cols + n])
    # end_time = time.time()
```

```
    # print(end_time - start_time)


# 对指定像素点的k个gmm_model进行排序(依据: w/sigma)
def gmm_model_sort(gmm_model, m, n, img_cols):
    # 构造排序依据
    order_weight = gmm_model.w[:, m * img_cols + n] / gmm_model.sigma[:, m * img_cols + n]
    # 封装order_weight和权重
    zip_ow_weight = zip(order_weight, gmm_model.w[:, m * img_cols + n])
    # 封装order_weight和均值
    zip_ow_mean = zip(order_weight, gmm_model.u[:, m * img_cols + n])
    # 封装order_weight和标准差
    zip_ow_standard_deviation = zip(order_weight, gmm_model.sigma[:, m * img_cols + n])
    zip_ow_weight = sorted(zip_ow_weight, reverse=True)
    zip_ow_mean = sorted(zip_ow_mean, reverse=True)
    zip_ow_standard_deviation = sorted(zip_ow_standard_deviation, reverse=True)
    temp, gmm_model.w[:, m * img_cols + n] = zip(*zip_ow_weight)
    temp, gmm_model.u[:, m * img_cols + n] = zip(*zip_ow_mean)
    temp, gmm_model.sigma[:, m * img_cols + n] = zip(*zip_ow_standard_deviation)
```

main.py

利用训练得到的高斯混合模型对当前帧进行运动目标检测的代码实现如下:

```
# 根据训练得到的GMM模型，对输入图像进行背景剪除操作，将处理后的图像返回
def background_subtract(gmm_model, single_frame):
    # 首先对计算得到的高斯模型进行筛选 需要满足条件sum(weight_i)>T
    img_rows = single_frame.shape[0]
    img_cols = single_frame.shape[1]
    for pixel_index in range(img_rows * img_cols):
        weight_sum = 0
        for k in range(gmm_model.model_count[0, pixel_index]):
            weight_sum = weight_sum + gmm_model.w[k, pixel_index]
            # 如果前K个模型已经满足权重阈值，则只选择前K个模型
            if weight_sum > gmm_model.t:
                gmm_model.model_count[0, pixel_index] = k + 1
                break
    # 初始化处理后的图片
    frame_parsed = np.full([img_rows, img_cols], 255, np.uint8)
    for m in range(img_rows):
        for n in range(img_cols):
            hit = False
            for k in range(gmm_model.model_count[0, m * img_cols + n]):
                # 计算当前像素与高斯分布均值的差值
                difference = abs(single_frame[m, n] - gmm_model.u[k, m * img_cols + n])
                if difference <= 2 * gmm_model.sigma[k, m * img_cols + n]:
                    # 背景
                    hit = True
                    break
            if hit:
                # 前景
                frame_parsed[m, n] = 0
    return frame_parsed
```

main.py

在获得运动目标的二值图像后，利用数学形态学的膨胀和腐蚀操作来分别消除目标中的空洞和噪声。

3 实验

这一章节将会利用上一章介绍的高斯混合模型的背景建模方法在视频序列图像上进行实验，并对其实现效果、实时性进行展示和分析。

首先，对手动实现的高斯混合模型、分别在 R、G、B 三个通道手动实现的高斯混合模型及调用的 OpenCV 高斯混合模型函数的实现效果进行对比。对于手动实现的高斯混合模型，分别设置参数 $K = 4$，$\alpha = 0.005$，$T = 0.75$，初始 $\mu = 0$，$\sigma = 30$，$w = 0.1$。对于 OpenCV 高斯混合模型函数，设置输入参数和手动实现的训练图像数量相同，varThreshold = 100。可视化结果（部分）如图 2 所示。

从图 2 可以看出，对于两种手动实现的方法而言，相比于在灰度图上对每个像素进行高斯混合模型建模，分别在 R、G、B 三个通道对每个像素进行建模获得的结果更加准确，表现在目标更加完整，作为前景的车辆空洞更少。但多通道的高斯混合模型建模方法也带来了更大的计算开销，约为前者的三倍。而调用的 OpenCV 高斯混合模型函数虽然其训练及测试速度较快，但其效果略差，一方面噪声较多且目标中存在大量空洞，另一方面即使传入了对阴影检测的参数，实际结果也仍有部分阴影被分为前景。三种实现方法的实时性如表 1 所示。

表 1: 不同实现方法的实时性比较

	手动实现	多通道手动实现	调用 OpenCV 函数
单张耗时（秒）	1.013	3.249	0.005
处理帧数（fps）	0.0107	0.0033	2.1469

下面将对部分参数设置对背景建模效果的影响进行分析。为了说明方便，我们仅在第一种实现方法将参数对建模的影响进行分析。

(a) (b) (c) (d)

图 2: 在 Scene_Data 视频序列上实现混合高斯模型的背景建模。第一行：原始视频序列图像；第二行：手动实现的高斯混合模型；第三行: 分别在 R、G、B 三个通道手动实现的高斯混合模型；第四行：调用的 OpenCV 高斯混合模型函数。(a) 第一百一十帧, (b) 第一百三十五帧, (c) 第一百五十五帧, (d) 第一百七十五帧, and (e) 第二百帧.

(a) (b) (c) (d) (e)

图 3: 在 Scene_Data 视频序列上实现实现混合高斯模型的背景建模: (a) 第一百五十五帧, (b) $\sigma = 20$, (c) $\sigma = 30$, (d) $\sigma = 50$, and (e) $\sigma = 80$, .

(a) (b) (c) (d)

图 4: 在 Scene_Data 视频序列上实现实现混合高斯模型的背景建模: (a) 第一百五十五帧, (b) $alpha = 0.005$, (c) $\alpha = 0.01$, and (c) $\alpha = 0.1$, .

首先探讨参数 σ 初始值对背景建模效果的影响，固定其他参数的取值，σ 分别取 20、30、50 和 80，可视化结果如图 3 所示。可以看出，随着 σ 的增大，视频序列图像中运动的物体被误分为背景的像素越多，即检测出的运动目标越来越不完整。其原因在于随着 σ 的增大，允许像素值偏离均值的范围越大，从而前景被误分类符合背景像素某一高斯模型的可能性越大，最终导致其被误分为背景。同理，如果单高斯模型的权重初始值设置越小，则背景像素被当作前景的可能性也越大。

其次，讨论参数 α 对背景建模效果的影响，固定其他参数的取值，α 分别取 0.005、0.01 和 0.1，可视化结果如图 4 所示。α 可以看作模型的“学习率”，可以看到模型在多帧训练的基础上，α 的取值对建模效果影响不大，略微在噪声和尺度较小的运动目标检测上有一定的区别。

4 总结

本文主要对基于高斯混合模型的背景建模方法进行了介绍，并对不同实现方法的建模效果及实时性进行了实验和分析。基于高斯混合模型的背景建模方法具有更强的鲁棒性和准确性，通过多个高斯模型加权构建实时背景，更贴近实际场景；同时减少了光线变化、空气流动及相机抖动等复杂因素

的干扰，避免了对新加入后静止不动的目标的误判发生。一定程度上减少了背景差分法对背景的依赖。通过膨胀、腐蚀等处理，可以得到更加清晰和完整的运动目标。在实时性方面，仅在灰度图像上对每个像素建立高斯混合模型，可以取得性能和实时性的较好平衡，而 OpenCV 封装的高斯混合模型函数的实时性最好。