
多目标跟踪快速入门教程

GiantPandaCV 公众号出品

GiantPandaCV-pprp

2020-05-07

Contents

零、序言	3
一、多目标跟踪 MOT16 数据集介绍及评价指标	4
1. 多目标跟踪	4
2. MOT16 数据集	4
3. MOT 中的评价指标	8
4. 参考资料	11
二、 MOT 数据标注工具 DarkLabel	11
1. 官方说明	12
2. 主要功能和特点	13
3. 主要用法	14
3.1 鼠标/键盘界面 (Shift / Ctrl = Shift 或 Ctrl)	14
3.2 指定标签和 ID	14
3.3 追踪功能	14
3.4 插值功能	15
3.5 导入视频/视频并在帧之间移动	15
3.6 保存并调出作业数据	15
3.7 数据格式 (语法)	16
4. 举栗子	16
5. ffmpeg 切割视频	18
6. 总结	18
三、 DarkLabel 配套代码	18
1. DarkLabel 转 Detection	19
2. DarkLabel 转 ReID 数据集	22
3. DarkLabel 转 MOT16 格式	28
四、 DeepSORT 论文解析	30
1. 简介	30
2. 方法	30
2.1 状态估计	30
2.2 匹配问题	32
2.3 级联匹配	33
2.4 表观特征	36
3. 实验	37
4. 总结	38

5. 参考	38
五、DeepSORT 核心代码解析	39
1. MOT 主要步骤	39
2. SORT	40
3. Deep SORT	41
4. Deep SORT 代码解析	44
4.1 类图	44
4.2 核心模块	46
5. 流程解析	64
6. 总结	78
7. 参考	78

零、序言

版权声明：此份电子书整理自公众号「GiantPandaCV」, 版权所有 GiantPandaCV, 禁止任何形式的转载, 禁止传播、商用, 违者必究! GiantPandaCV 公众号由专注于技术的一群 95 后创建, 专注于机器学习、深度学习、计算机视觉、图像处理等领域。每天更新一到两篇相关推文, 希望在传播知识、分享知识的同时能够启发你。欢迎关注我们的公众号 GiantPandaCV:



Figure 1: 扫码关注

目前为止，我们已经推出了《从零开始学习 YOLOv3》和《从零开始学习 SSD》两个系列教程，方便各位传阅，但是请注意版权归 GiantPandaCV 公众号所有，不可用于牟利，否则我们将保留起诉的权利。

一、多目标跟踪 MOT16 数据集介绍及评价指标

1. 多目标跟踪

多目标跟踪处理的对象是视频，从视频的第一帧到最后一帧，里边有多个目标在不断运动。多目标跟踪的目的就是将每个目标和其他目标进行区分开来，具体方法是给每个目标分配一个 **ID**，并记录他们的轨迹。

刚开始接触，可能觉得直接将目标检测的算法应用在视频的每一帧就可以完成这个任务了。实际上，目标检测的效果是很不稳定的，其实最大的区别在于，仅仅使用目标检测无法给对象分配 ID，并且跟踪能够优化整个跟踪过程，能让目标框更加稳定。

多目标跟踪中一个比较经典的和深度学习结合比较好的方法就是 **Detection Based Tracking**，对前后两帧进行目标检测检测，然后根据得到的前后两帧的所有目标进行匹配，从而维持 ID。初学者接触比较多的就是 SORT 和 Deep SORT 算法了。

2. MOT16 数据集

MOT16 数据集是在 2016 年提出来的用于衡量多目标跟踪检测和跟踪方法标准的数据集，专门用于行人跟踪。官网地址是：<https://motchallenge.net/>

从官网下载的数据是按照以下的文件结构进行组织的：

```
- MOT16
  - train
    - MOT16-02
      - det
        - det.txt
      - gt
        - gt.txt
      - img1
      - seqinfo.ini
    - MOT16-04
    - MOT16-05
    - MOT16-09
    - MOT16-10
    - MOT16-11
```

```
- MOT16-13
- test
  - MOT16-01
    - det
      - det.txt
    - img1
    - seqinfo.ini
  - MOT16-03
  - MOT16-06
  - MOT16-07
  - MOT16-08
  - MOT16-12
  - MOT16-14
```

在 MOT16 数据集中，是包含了检测得到的框的，这样是可以免去目标检测这个部分，提供统一的目标检测框以后，然后可以比较目标跟踪更关注的部分，而不用在花费精力在目标检测上。

1. seqinfo.ini

在每个子文件夹中都有这个，主要用于说明这个文件的一些信息，比如长度，帧率，图片的长和宽，图片的后缀名。

```
[Sequence]
name=MOT16-09
imDir=img1
frameRate=30
seqLength=525
imWidth=1920
imHeight=1080
imExt=.jpg
```

2. det.txt

这个文件中存储了图片的检测框的信息 (检测得到的信息文件)，部分内容展示如下：

```
1,-1,1359.1,413.27,120.26,362.77,2.3092,-1,-1,-1
1,-1,571.03,402.13,104.56,315.68,1.5028,-1,-1,-1
1,-1,650.8,455.86,63.98,193.94,0.33276,-1,-1,-1
1,-1,721.23,446.86,41.871,127.61,0.27401,-1,-1,-1
2,-1,460.48,442.1,90.896,274.69,0.26749,-1,-1,-1
2,-1,643.66,461.78,59.629,180.89,0.18023,-1,-1,-1
2,-1,1494.4,408.29,112.14,338.41,0.16075,-1,-1,-1
2,-1,572.83,364.89,128.96,388.88,-0.011851,-1,-1,-1
```

```
3,-1,1359.1,413.27,120.26,362.77,2.3387,-1,-1,-1
3,-1,571.03,402.13,104.56,315.68,0.79923,-1,-1,-1
3,-1,1482.5,390.88,128.96,388.88,0.35271,-1,-1,-1
...
600,-1,708.37,476.87,20.435,63.306,-0.1483,-1,-1,-1
600,-1,421.14,446.86,41.871,127.61,-0.19971,-1,-1,-1
600,-1,613.25,412.69,51.78,157.34,-0.38627,-1,-1,-1
600,-1,375.27,454.06,48.246,146.74,-0.42444,-1,-1,-1
```

从左到右分别代表：

- **frame**: 第几帧图片
- **id**: 这个检测框分配的 **id**，在这里都是-1 代表没有 **id** 信息
- **bbox**(四位): 分别是左上角坐标和长宽
- **conf**: 这个 **bbox** 包含物体的置信度，可以看到并不是传统意义的 0-1，分数越高代表置信度越高
- **MOT3D(x,y,z)**: 是在 MOT3D 中使用到的内容，这里关心的是 MOT2D，所以都设置为-1

可以看出以上内容主要提供的和目标检测的信息没有区别，所以也在一定程度上可以用于检测器的训练。

3. gt.txt

这个文件只有 **train** 的子文件夹中有，**test** 中没有，其中内容的格式和 **det.txt** 有一些类似，部分内容如下：

```
1,1,912,484,97,109,0,7,1
2,1,912,484,97,109,0,7,1
3,1,912,484,97,109,0,7,1
...
136,1,912,484,97,109,0,7,0.93878
137,1,912,484,97,109,0,7,0.86735
138,1,912,484,97,109,0,7,0.79592
139,1,912,484,97,109,0,7,0.72449
140,1,912,484,97,109,0,7,0.65306
...
338,1,912,484,97,109,0,7,0
339,1,912,484,97,109,0,7,0
340,1,912,484,97,109,0,7,0
341,1,912,484,97,109,0,7,0
```

```
...  
599,1,912,484,97,109,0,7,1  
600,1,912,484,97,109,0,7,1  
  
1,2,1338,418,167,379,1,1,1  
2,2,1342,417,168,380,1,1,1  
3,2,1346,417,170,380,1,1,1  
...
```

从左到右分别是：

- **frame**: 第几帧图片
- **ID**: 也就是轨迹的 ID，可以看出 **gt** 里边是按照轨迹的 ID 号进行排序的
- **bbox**: 分别是左上角坐标和长宽
- 是否忽略：0 代表忽略
- **classes**: 目标的类别个数（这里是驾驶场景包括 12 个类别），7 代表的是静止的人。

第 8 个类代表错检，9-11 代表被遮挡的类别

Label	ID
Pedestrian	1
Person on vehicle	2
Car	3
Bicycle	4
Motorbike	5
Non motorized vehicle	6
Static person	7
Distractor	8
Occluder	9
Occluder on the ground	10
Occluder full	11
Reflection	12

- 最后一个代表目标运动时被其他目标包含、覆盖、边缘裁剪的情况。

总结：

- **train** 中含有的标注信息主要来自 **det.txt** 和 **gt.txt**。**test** 中只含有 **det.txt**。

- det.txt 含有的有用信息有: frame, bbox, conf
- gt.txt 含有的有用信息有: frame, bbox, conf, id, class
- output.txt(使用 deepsort 得到的文件)中含有的有用信息有: frame, bbox, id

3. MOT 中的评价指标

```
10:28:55 INFO - mergeOverall: 5.368 seconds.
```

	IDF1	IDP	IDR	Rc11	Prcn	GT	MT	PT	ML	FP	FN	IDs	FM	MOTA	MOTP	IDt	IDa	IDm
MOT16-02	23.8%	62.3%	14.7%	19.8%	83.6%	54	5	13	36	691	14307	40	129	15.7%	0.236	13	26	3
MOT16-04	42.5%	64.4%	31.8%	41.8%	84.7%	83	7	44	32	3582	27682	154	625	33.9%	0.207	59	77	8
MOT16-05	43.6%	76.7%	30.5%	34.9%	87.7%	125	8	54	63	332	4440	31	93	29.6%	0.230	24	20	17
MOT16-09	48.2%	56.9%	41.8%	57.2%	77.7%	25	5	16	4	861	2250	50	118	39.9%	0.253	23	28	3
MOT16-10	38.7%	64.7%	27.6%	37.5%	87.7%	54	6	19	29	645	7701	35	200	32.0%	0.242	6	28	4
MOT16-11	51.6%	68.3%	41.5%	53.5%	88.1%	69	10	24	35	661	4266	21	73	46.1%	0.205	6	18	3
MOT16-13	26.1%	85.0%	15.4%	16.6%	91.8%	107	5	26	76	170	9546	22	115	15.0%	0.256	7	16	3
OVERALL	39.3%	65.7%	28.1%	36.4%	85.3%	517	46	196	275	6942	70192	353	1353	29.8%	0.221	138	213	41

```
10:28:55 INFO - Completed
```

评价出发点:

- 所有出现的目标都要及时能够找到;
- 目标位置要尽可能与真实目标位置一致;
- 每个目标都应该被分配一个独一无二的 ID, 并且该目标分配的这个 ID 在整个序列中保持不变。

评价指标数学模型: 评价过程的步骤:

1. 建立目标与假设最优间的最优一一对应关系, 称为 **correspondence**
2. 对所有的 **correspondence**, 计算位置偏移误差
3. 累积结构误差 a. 计算漏检数 b. 计算虚警数 (不存在目标却判断为目标) c. 跟踪目标发生跳变的次数

Evaluation Measures

Lower is better. Higher is better.

Measure	Better	Perfect	Description
Avg Rank	lower	1	This is the rank of each tracker averaged over all present evaluation measures.
MOTA	higher	100 %	Multiple Object Tracking Accuracy [1]. This measure combines three error sources: false positives, missed targets and identity switches.
MOTP	higher	100 %	Multiple Object Tracking Precision [1]. The misalignment between the annotated and the predicted bounding boxes.
IDF1	higher	100 %	ID F1 Score [2]. The ratio of correctly identified detections over the average number of ground-truth and computed detections.
FAF	lower	0	The average number of false alarms per frame.
MT	higher	100 %	Mostly tracked targets. The ratio of ground-truth trajectories that are covered by a track hypothesis for at least 80% of their respective life span.
ML	lower	0 %	Mostly lost targets. The ratio of ground-truth trajectories that are covered by a track hypothesis for at most 20% of their respective life span.
FP	lower	0	The total number of false positives.
FN	lower	0	The total number of false negatives (missed targets).
ID Sw.	lower	0	The total number of identity switches. Please note that we follow the stricter definition of identity switches as described in [3].
Frag	lower	0	The total number of times a trajectory is fragmented (i.e. interrupted during tracking).
Hz	higher	Inf.	Processing speed (in frames per second excluding the detector) on the benchmark.

https://blog.csdn.net/DD_PP_JJ

1. MOTA(Multiple Object Tracking Accuracy)

$$MOTA = 1 - \frac{\sum(FN + FP + IDSW)}{\sum GT} \in (-\infty, 1]$$

FN 为 False Negative, FP 为 False Positive, IDSW 为 ID Switch, GT 是 Ground Truth 物体的数量。

MOTA 主要考虑的是 tracking 中所有对象匹配错误, 主要是 FP, FN, IDs. MOTA 给出的是非常直观的衡量跟踪其在检测物体和保持轨迹时的性能, 与目标检测精度无关。

MOTA 取值小于 100, 但是当跟踪器产生的错误超过了场景中的物体, MOTA 可以变为负数。

ps: MOTA&MOTP 是计算所有帧相关指标后再进行平均的, 不是计算每帧的 rate 然后进行 rate 平

均。

2. MOTP(Multiple Object Tracking Precision)

$$MOTP = \frac{\sum_{t,i} d_{t,i}}{\sum_t c_t}$$

d 为检测目标 i 和给它分配的 **ground truth** 之间在所有帧中的平均度量距离，在这里是使用 **bonding box** 的 **overlap rate** 来进行度量（在这里 MOTP 是越大越好，但对于使用欧氏距离进行度量的就是 MOTP 越小越好，这主要取决于度量距离 d 的定义方式）；而 c 为在当前帧匹配成功的数目。MOTP 主要量化检测器的定位精度，几乎不包含与跟踪器实际性能相关的信息。

3. MT(Mostly Tracked)

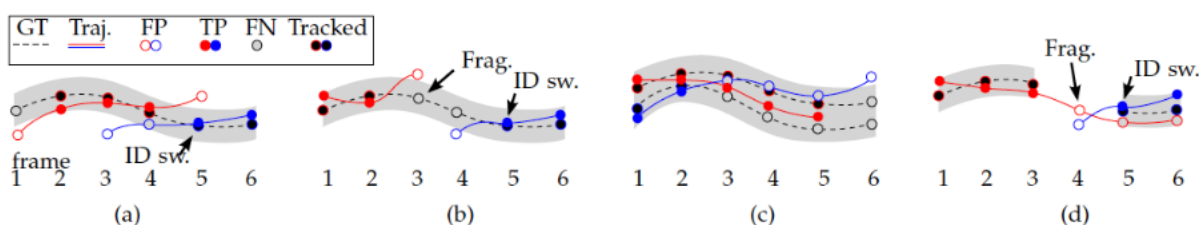
满足 **Ground Truth** 至少在 80% 的时间内都匹配成功的 **track**，在所有追踪目标中所占的比例。注意这里的 MT 和 ML 与当前 **track** 的 ID 是否发生变化无关，只要 **Ground Truth** 与目标匹配上即可。

4. ML (Mostly Lost)

满足 **Ground Truth** 在小于 20% 的时间内匹配成功的 **track**，在所有追踪目标中所占的比例。

5. ID Switch

Ground Truth 所分配的 ID 发生变化的次数，如图 1 中 (a) 所示。



6. FM (Fragmentation)

FM 计算的是跟踪有多少次被打断（既 **Ground Truth** 的 **track** 没有被匹配上），换句话说每当轨迹将其状态从跟踪状态改变为未跟踪状态，并且在稍后的时间点跟踪相同的轨迹时，就会对 FM 进行计数。此处需要注意的是，FM 计数时要求 **ground truth** 的状态需要满足：tracked->untracked->tracked，如图 1 中 (b) 所示，而 (c) 中的不算 FM。需要注意的是，FM 与 ID 是否发生变化无关。

7. FP (False Positive)

当前帧预测的 **track** 和 **detection** 没有匹配上，将错误预测的 **track** 点称为 FP，如图 1 所示。是否匹配成功与匹配时所设置的阈值有关。

8. FN (False Negative)

当前帧预测的 **track** 和 **detection** 没有匹配上，将未被匹配的 **ground truth** 点称为 **FN**（也可以称为 **Miss**）

9. ID scores

MOTA 的主要问题是仅仅考虑跟踪器出错的次数，但是有一些场景（比如航空场景）更加关注一个跟踪器是否尽可能长的跟踪一个目标。这个问题通过构建二分图来解决，主要计算对象是 **IDTP**、**IDFP**、**IDFN**。

- **IDP**:

$$IDP = \frac{IDTP}{IDTP + IDFP}$$

- **IDR**:

$$IDR = \frac{IDTP}{IDTP + IDFN}$$

- **IDF1**: 正确识别的检测与真实数和计算检测的平均数之比

$$IDF1 = \frac{2IDTP}{2IDTP + IDFP + IDFN}$$

4. 参考资料

Evaluating Multiple Object Tracking Performance: The CLEAR MOT Metrics.

<https://motchallenge.net/>

<https://blog.csdn.net/u012477435/article/details/104158573>

二、MOT 数据标注工具 DarkLabel

DarkLabel 是一个轻量的视频标注软件，相比于 **ViTBAT** 等软件而言，不需要安装就可以使用，本文将介绍 **darklabel** 软件的使用指南。

笔者最终从公开的软件中选择了 **DarkLabel**。**DarkLabel** 体积非常小，开箱即用，不需要配置环境 (**Vatic** 需要在 **linux** 下配置相关环境)，对 **window** 用户很友好。不过该软件使用说明实际上不多，本文总结了大部分的用法，实际运用还需要读者研究。

DarkLabel 导出的格式可以通过脚本转化，变成标准的目标检测数据集格式、**ReID** 数据集格式、**MOT** 数据集格式。

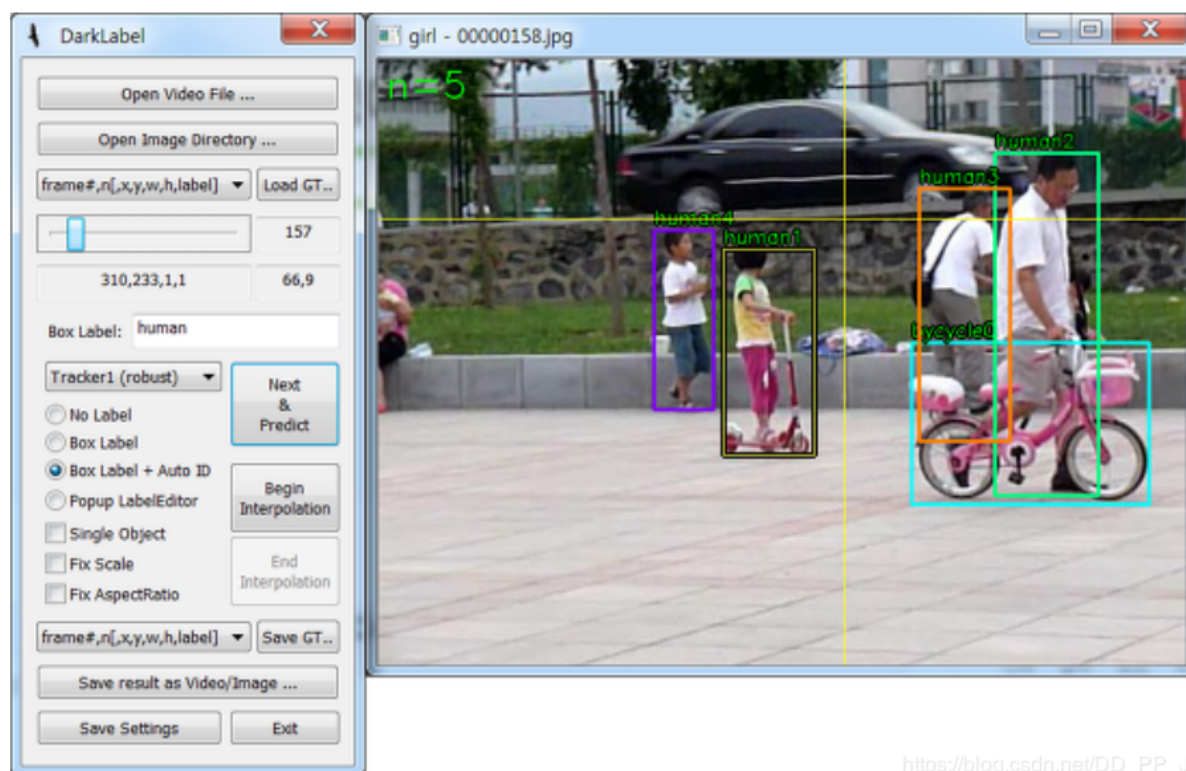
之后会在这个视频标注的基础上进行一些脚本的编写，可以批量构建 ReID 数据集、目标检测数据集和 MOT 数据集。

1. 官方说明

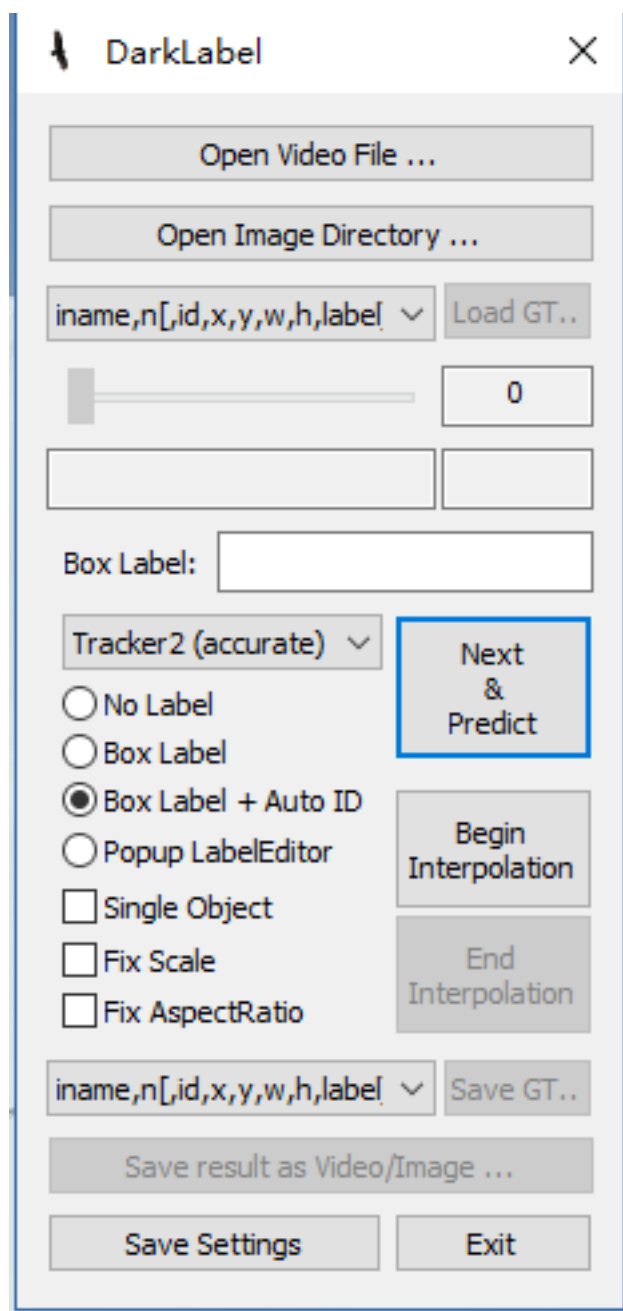
它是一个实用程序，可以沿着视频（avi，mpg）或图像列表中对象的矩形边界框以各种格式标记和保存。该程序可用于创建用于对象识别或图像跟踪目的的数据库。最大的功能是快速响应，便捷的界面以及减少工作量的便捷功能（自动跟踪，使用插值进行标记，自动 ID 标记）。任何人都可以将其用于非商业目的，如果您有任何问题或建议，请在评论中让我知道。最初是为我自己创建的，最近我花了些时间来改进该程序（ver1.3）。我们已经改进了难以看清的细微之处，但是改善了程序的质量，执行的稳定性和未知性。

– Dark Programmer

软件示意：



工具栏在左侧：



2. 主要功能和特点

- 支持各种格式的视频（avi，mpg 等）和图像列表（jpg，bmp，png 等）
- 多框设置和标签设置支持
- 支持对象识别和图像跟踪中使用的各种数据格式

- 使用图像跟踪器自动标记（通过跟踪标记）
- 支持使用插值功能的间隔标签
- 自动标记功能，可按类别自动为每个对象分配唯一的 ID

3. 主要用法

3.1 鼠标/键盘界面（**Shift / Ctrl = Shift** 或 **Ctrl**）

- 鼠标拖动：创建一个框
- **Shift / Ctrl** + 拖动：编辑框
- 双击：选择/取消相同 ID 对象的轨迹
- 右键单击：删除所有选定的对象轨迹（删除部分）
- 右键单击：删除最近创建的框（如果未选择任何轨迹）
- **Shift / Ctrl** + 右键单击（特定框）：仅删除所选框
- **Shift / Ctrl** + 右键单击（空）：删除当前屏幕上的所有框
- **Shift / Ctrl** + 双击（特定框）：修改所选框的标签
- **Shift / Ctrl** + 双击（轨迹）：在所选轨迹上批量更改标签
- 箭头键 / PgUp / PgDn / Home / End：移动视频帧（图像）
- **Enter** 键：使用图像跟踪功能自动生成框（通过跟踪进行标记）

3.2 指定标签和 ID

- 无标签：创建未标签的框
- 框标签：用户指定的标签（例如，人类）
- **box** 标签 + 自动编号：自动编号自定义标签（例如 **human0**, **human1** 等）
- 如果指定了 **id**，则可以选择/编辑轨迹单位对象
- **popuplabeleditor**：注册标签列表窗口的弹出窗口（已在 **labels.txt** 文件中注册）
- 如果在弹出窗口中按快捷键（1 - 9），则会自动输入标签。
- **Label + id** 显示在屏幕上，但在内部，标签和 ID 分开。
- 当另存为 **gt** 数据时，选择仅标签格式以保存可见标签（标签 + **id**）
- 另存为 **gt** 数据时，如果选择了标签和 ID 分类格式，则标签和 ID 将分开保存。

3.3 追踪功能

这是这个软件比较好的功能之一，可以用传统方法（KCF 类似的算法）跟踪目标，只需要对不准确的目标进行人工调整即可，大大减少了工作量。

- 通过使用图像跟踪功能设置下一帧的框（分配相同的 ID / 标签）
- 多达 100 个同时跟踪
- **tracker1**（稳健）算法：长时间跟踪目标
- **tracker2**（准确）算法：准确跟踪目标（例如汽车）
- 输入键/下一步和预测按钮
- 注意！使用跟踪时，下一帧上的原始框消失

tracker1 和 tracker2 在不同场景下各有利弊，可以都试试。

3.4 插值功能

- 跟踪功能方便，但问题不准确
- 在视频部分按对象标记时使用
 - 开始插补按钮：开始插补功能
 - 在目标对象的轨迹的一半处绘制一个方框（航路点的种类）
 - 航路点框为紫色，插值框为黑色。
 - 更正插值错误的部分（**Shift / Ctrl + 拖动**），添加任意数量的航路点（不考虑顺序）/删除
 - 结束插补按钮：将工作结束和工作轨迹注册为数据

3.5 导入视频/视频并在帧之间移动

- 打开视频文件：打开视频文件（avi, mpg, mp4, wmv, mov, ...）
- 打开图像目录：打开文件夹中的所有图像（jpg, bmp, png 等）
- 在视频帧之间移动：键盘 **→**, **←**, **PgUp**, **PgDn**, **Home**, **End**, 滑块控制

3.6 保存并调出作业数据

- 加载 GT：以所选格式加载地面真相文件。
- 保存 GT：以所选数据格式保存到目前为止已获得的结果。
- 导入数据时，需要选择与实际数据文件匹配的格式，但是在保存数据时，可以将其保存为所需的任何格式。
- 在图像列表中工作时，使用帧号（**frame #**）格式，按文件名排序时的图像顺序将变为帧号（对于诸如 00000.jpg, 00002.jpg 等的列表很有用）
- 保存设置：保存当前选择的数据格式和选项（运行程序时自动还原）

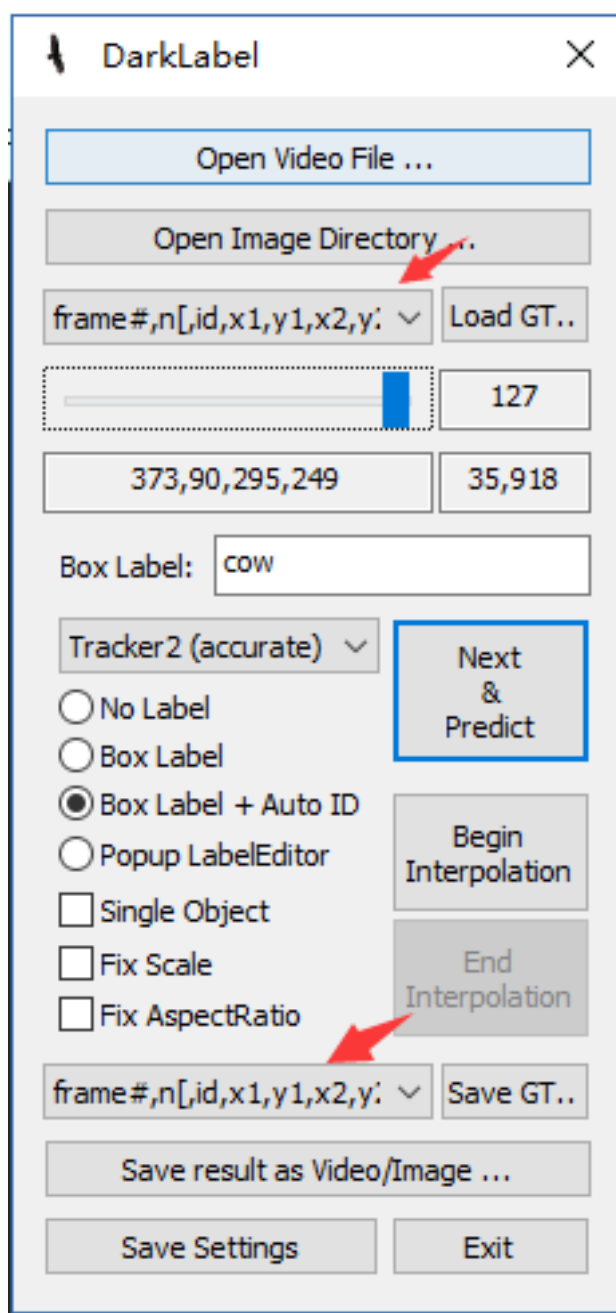
3.7 数据格式（语法）

- |：换行
- []：重复短语
- **frame #**：帧号（视频的帧号，图像列表中的图像顺序）
- **iname**：图像文件名（仅在使用图像列表时有效）
- **标签**：标签
- **id**：对象的唯一 ID
- **n**：在图像上设置的边界矩形的数量
- **x, y**：边界矩形的左侧和顶部位置
- **w, h**：边界矩形的宽度和高度
- **cx, cy**：边界矩形的中心坐标
- **x1, y1, x2, y2**：边界矩形的左上，右下位置

4. 举栗子

视频演示：<https://www.youtube.com/watch?v=vbydG78Al8s&t=11s>

1. 选择 **open video file**，选择一个视频打开，最好不要太长
2. 左右拖动一下滑块，看一下准备标注的对象
3. 如果标注视频选择左侧工具栏中第三行，下拉找到 **frame** 开头的内容比如：**frame#, n, [id, x1,y1,x2,y2,label]**，意思是左上角坐标和右下角坐标。



4. 然后右侧框中进行画框，然后可以采用以下几种方法继续标注

- 画框以后，长按 Enter 键 (Enter 键是 Next&Predict 的快捷键)，就会采用 Tracker2 中的模式进行预测
- 调整框：键盘长按 ctrl 键的同时，用鼠标拖动已经标注的目标框。

5. 将这段视频标注完成后，点击 Save GT，保存为 txt 文件

5. ffmpeg 切割视频

```
ffmpeg -i C:/plutopr.mp4 -acodec copy
      -vf scale=1280:720
      -ss 00:00:10 -t 15 C:/cutout1.mp4 -y
```

1. -ss time_off set the start time offset 设置从视频的哪个时间点开始截取，上文从视频的第 10s 开始截取
2. -to 截到视频的哪个时间点结束。上文到视频的第 15s 结束。截出的视频共 5s. 如果用-t 表示截取多长的时间如上文-to 换位-t 则是截取从视频的第 10s 开始，截取 15s 时长的视频。即截出来的视频共 15s.
3. -vcodec copy 表示使用跟原视频一样的视频编解码器。
4. -acodec copy 表示使用跟原视频一样的音频编解码器。
5. -i 表示源视频文件
6. -y 表示如果输出文件已存在则覆盖。

6. 总结

这个软件是笔者自己进行项目的时候用到的一款标注软件，大部分视频标注软件要不就是太大 (ViTBAT 软件)，要不就是需要 Linux 环境，所以在 Window 上标注的话很不方便，经过了很长时间探索，最终找到这款软件。此外，这款软件源码没有公开，开发者声明可以用于非商业目的。

DarkLabel 软件的获取可以在 GiantPandaCV 公众号后台回复“darklabel”，即可得到该软件的下载链接。

三、DarkLabel 配套代码

先附上脚本地址：<https://github.com/pprp/SimpleCVReproduction/tree/master/DarkLabel>

先来了解一下为何 DarkLabel 能生成这么多格式的数据集，来看看 DarkLabel 的格式：

```
frame(从 0 开始计), 数量, id(从 0 开始), box(x1,y1,x2,y2), class=null
0,4,0,450,194,558,276,null,1,408,147,469,206,null,2,374,199,435,307,null,3,153,213,218,3
1,4,0,450,194,558,276,null,1,408,147,469,206,null,2,374,199,435,307,null,3,153,213,218,3
2,4,0,450,194,558,276,null,1,408,147,469,206,null,2,374,199,435,307,null,3,153,213,218,3
```

每一帧，每张图片上的目标都可以提取到，并且每个目标有 bbox、分配了一个 ID、class

这些信息都可以满足目标检测、ReID、跟踪数据集。

ps: 说明一下, 以下脚本都是笔者自己写的, 专用于单类的检测、跟踪、重识别的代码, 如果有需要多类的, 还需要自己修改多类部分的代码。另外以下只针对 Darklabel 中 **frame#,n,[id,x1,y1,x2,y2,label]** 格式。

1. DarkLabel 转 Detection

这里笔者写了一个脚本转成 VOC2007 中的 xml 格式的标注, 代码如下:

```
import cv2
import os
import shutil
import tqdm
import sys

root_path = r"I:\Dataset\VideoAnnotation"

def print_flush(str):
    print(str, end='\r')
    sys.stdout.flush()

def genXML(xml_dir, outname, bboxes, width, height):
    xml_file = open((xml_dir + '/' + outname + '.xml'), 'w')
    xml_file.write('<annotation>\n')
    xml_file.write('    <folder>VOC2007</folder>\n')
    xml_file.write('    <filename>' + outname + '.jpg' + '</filename>\n')
    xml_file.write('    <size>\n')
    xml_file.write('        <width>' + str(width) + '</width>\n')
    xml_file.write('        <height>' + str(height) + '</height>\n')
    xml_file.write('        <depth>3</depth>\n')
    xml_file.write('    </size>\n')

    for bbox in bboxes:
        x1, y1, x2, y2 = bbox
        xml_file.write('    <object>\n')
        xml_file.write('        <name>' + 'cow' + '</name>\n')
        xml_file.write('        <pose>Unspecified</pose>\n')
        xml_file.write('        <truncated>0</truncated>\n')
        xml_file.write('        <difficult>0</difficult>\n')
        xml_file.write('        <bndbox>\n')
        xml_file.write('            <xmin>' + str(x1) + '</xmin>\n')
        xml_file.write('            <ymin>' + str(y1) + '</ymin>\n')
        xml_file.write('            <xmax>' + str(x2) + '</xmax>\n')
```

```

        xml_file.write('                <ymax>' + str(y2) + '</ymax>\n')
        xml_file.write('            </bndbox>\n')
        xml_file.write('    </object>\n')

xml_file.write('</annotation>')

def gen_empty_xml(xml_dir, outname, width, height):
    xml_file = open((xml_dir + '/' + outname + '.xml'), 'w')
    xml_file.write('<annotation>\n')
    xml_file.write('    <folder>VOC2007</folder>\n')
    xml_file.write('    <filename>' + outname + '.png' + '</filename>\n')
    xml_file.write('    <size>\n')
    xml_file.write('        <width>' + str(width) + '</width>\n')
    xml_file.write('        <height>' + str(height) + '</height>\n')
    xml_file.write('        <depth>3</depth>\n')
    xml_file.write('    </size>\n')
    xml_file.write('</annotation>')

def getJPG(src_video_file, tmp_video_frame_save_dir):
    # gen jpg from video
    cap = cv2.VideoCapture(src_video_file)
    if not os.path.exists(tmp_video_frame_save_dir):
        os.makedirs(tmp_video_frame_save_dir)

    frame_cnt = 0
    isrun, frame = cap.read()
    width, height = frame.shape[1], frame.shape[0]

    while (isrun):
        save_name = append_name + "_" + str(frame_cnt) + ".jpg"
        cv2.imwrite(os.path.join(tmp_video_frame_save_dir, save_name), frame)
        frame_cnt += 1
        print_flush("Extracting frame :%d" % frame_cnt)
        isrun, frame = cap.read()
    return width, height

def delTmpFrame(tmp_video_frame_save_dir):
    if os.path.exists(tmp_video_frame_save_dir):
        shutil.rmtree(tmp_video_frame_save_dir)
    print('delete %s success!' % tmp_video_frame_save_dir)

```

```

def assign_jpgAndAnnot(src_annot_file, dst_annot_dir, dst_jpg_dir,
    ↪ tmp_video_frame_save_dir, width, height):
    # get coords from annotations files
    txt_file = open(src_annot_file, "r")

    content = txt_file.readlines()

    for line in content:
        item = line[:-1]
        items = item.split(',')
        frame_id, num_of_cow = items[0], items[1]
        print_flush("Assign jpg and annotation : %s" % frame_id)

        bboxes = []

        for i in range(int(num_of_cow)):
            obj_id = items[1 + i * 6 + 1]
            obj_x1, obj_y1 = int(items[1 + i * 6 + 2]), int(items[1 + i * 6 +
    ↪ 3])
            obj_x2, obj_y2 = int(items[1 + i * 6 + 4]), int(items[1 + i * 6 +
    ↪ 5])

            # preprocess the coords
            obj_x1 = max(1, obj_x1)
            obj_y1 = max(1, obj_y1)
            obj_x2 = min(width, obj_x2)
            obj_y2 = min(height, obj_y2)
            bboxes.append([obj_x1, obj_y1, obj_x2, obj_y2])

        genXML(dst_annot_dir, append_name + "_" + str(frame_id), bboxes,
    ↪ width,
            height)
        shutil.copy(
            os.path.join(tmp_video_frame_save_dir,
                append_name + "_" + str(frame_id) + ".jpg"),
            os.path.join(dst_jpg_dir, append_name + "_" + str(frame_id) +
    ↪ ".jpg"))

        txt_file.close()

if __name__ == "__main__":
    append_names = ["cutout%d" % i for i in range(19, 66)]

```

```
for append_name in append_names:
    print("processing",append_name)
    src_video_file = os.path.join(root_path, append_name + ".mp4")

    if not os.path.exists(src_video_file):
        continue

    src_annot_file = os.path.join(root_path, append_name + "_gt.txt")

    dst_annot_dir = os.path.join(root_path, "Annotations")
    dst_jpg_dir = os.path.join(root_path, "JPEGImages")

    tmp_video_frame_save_dir = os.path.join(root_path, append_name)

    width, height = getJPG(src_video_file, tmp_video_frame_save_dir)

    assign_jpgAndAnnot(src_annot_file, dst_annot_dir, dst_jpg_dir,
        ↪ tmp_video_frame_save_dir, width, height)

    delTmpFrame(tmp_video_frame_save_dir)
```

如果想转成 U 版 yolo 需要的格式可以点击 https://github.com/pprp/voc2007_for_yolo_torch 使用这里的脚本。

2. DarkLabel 转 ReID 数据集

ReID 数据集其实与分类数据集很相似，最出名的是 Market1501 数据集，对这个数据集不熟悉的可以先百度一下。简单来说 ReID 数据集只比分类中多了 query, gallery 的概念，也很简单。转换代码如下：

```
import os
import shutil
import cv2
import numpy as np
import glob
import sys
import random
"""[summary]
根据视频和 darklabel 得到的标注文件
"""

def preprocessVideo(video_path):
```

```
'''
预处理，将视频变为一帧一帧的图片
'''
if not os.path.exists(video_frame_save_path):
    os.mkdir(video_frame_save_path)

vidcap = cv2.VideoCapture(video_path)
(cap, frame) = vidcap.read()

height = frame.shape[0]
width = frame.shape[1]

cnt_frame = 0

while (cap):
    cv2.imwrite(
        os.path.join(video_frame_save_path, "frame_%d.jpg" %
        ↪ (cnt_frame)),
        frame)
    cnt_frame += 1
    print(cnt_frame, end="\r")
    sys.stdout.flush()
    (cap, frame) = vidcap.read()
vidcap.release()
return width, height

def postprocess(video_frame_save_path):
    '''
    后处理，删除无用的文件夹
    '''
    if os.path.exists(video_frame_save_path):
        shutil.rmtree(video_frame_save_path)

def extractVideoImgs(frame, video_frame_save_path, coords):
    '''
    抠图
    '''
    x1, y1, x2, y2 = coords
    # get image from save path
    img = cv2.imread(
        os.path.join(video_frame_save_path, "frame_%d.jpg" % (frame)))
```

```
if img is None:
    return None
# crop
save_img = img[y1:y2, x1:x2]
return save_img

def bbox_iious(box1, box2):
    b1_x1, b1_y1, b1_x2, b1_y2 = box1[0], box1[1], box1[2], box1[3]
    b2_x1, b2_y1, b2_x2, b2_y2 = box2[0], box2[1], box2[2], box2[3]

    # Intersection area
    inter_area = (min(b1_x2, b2_x2) - max(b1_x1, b2_x1)) * \
        (min(b1_y2, b2_y2) - max(b1_y1, b2_y1))

    # Union Area
    w1, h1 = b1_x2 - b1_x1, b1_y2 - b1_y1
    w2, h2 = b2_x2 - b2_x1, b2_y2 - b2_y1
    union_area = (w1 * h1 + 1e-16) + w2 * h2 - inter_area

    return inter_area / union_area

def bbox_iou(box1, box2):
    # format box1: x1,y1,x2,y2
    # format box2: a1,b1,a2,b2
    x1, y1, x2, y2 = box1
    a1, b1, a2, b2 = box2

    i_left_top_x = max(a1, x1)
    i_left_top_y = max(b1, y1)

    i_bottom_right_x = min(a2, x2)
    i_bottom_right_y = min(b2, y2)

    intersection = (i_bottom_right_x - i_left_top_x) * (i_bottom_right_y -
        i_left_top_y)

    area_two_box = (x2 - x1) * (y2 - y1) + (a2 - a1) * (b2 - b1)

    return intersection * 1.0 / (area_two_box - intersection)
```



```
def restrictCoords(width, height, x, y):
    x = max(1, x)
    y = max(1, y)
    x = min(x, width)
    y = min(y, height)
    return x, y

if __name__ == "__main__":

    total_cow_num = 0

    root_dir = "./data/videoAndLabel"
    reid_dst_path = "./data/reid"
    done_dir = "./data/done"

    txt_list = glob.glob(os.path.join(root_dir, "*.txt"))
    video_list = glob.glob(os.path.join(root_dir, "*.mp4"))

    for i in range(len(txt_list)):
        txt_path = txt_list[i]
        video_path = video_list[i]

        print("processing:", video_path)

        if not os.path.exists(txt_path):
            continue

        video_name = os.path.basename(video_path).split('.')[0]
        video_frame_save_path = os.path.join(os.path.dirname(video_path),
                                              video_name)

        f_txt = open(txt_path, "r")

        width, height = preprocessVideo(video_path)

        print("done")

        # video_cow_id = video_name + str(total_cow_num)

        for line in f_txt.readlines():
            bboxes = line.split(',')
            total_cow_num += 1
```

```
ids = []
frame_id = int(bboxes[0])

box_list = []

if frame_id % 30 != 0:
    continue

num_object = int(bboxes[1])
for num_obj in range(num_object):
    # obj = 0, 1, 2
    obj_id = bboxes[1 + (num_obj) * 6 + 1]
    obj_x1 = int(bboxes[1 + (num_obj) * 6 + 2])
    obj_y1 = int(bboxes[1 + (num_obj) * 6 + 3])
    obj_x2 = int(bboxes[1 + (num_obj) * 6 + 4])
    obj_y2 = int(bboxes[1 + (num_obj) * 6 + 5])

    box_list.append([obj_x1, obj_y1, obj_x2, obj_y2])
    # process coord
    obj_x1, obj_y1 = restrictCoords(width, height, obj_x1, obj_y1)
    obj_x2, obj_y2 = restrictCoords(width, height, obj_x2, obj_y2)

    specific_object_name = video_name + "_" + obj_id

    # mkdir for reid dataset
    id_dir = os.path.join(reid_dst_path, specific_object_name)

    if not os.path.exists(id_dir):
        os.makedirs(id_dir)

    # save pic
    img = extractVideoImgs(frame_id, video_frame_save_path,
                           (obj_x1, obj_y1, obj_x2, obj_y2))
    print(type(img))

    if img is None or img.shape[0] == 0 or img.shape[1] == 0:
        print(specific_object_name + " is empty")
        continue

    # print(frame_id)
    img = cv2.resize(img, (256, 256))

    normalizedImg = np.zeros((256, 256))
```

```
img = cv2.normalize(img, normalizedImg, 0, 255,
                    cv2.NORM_MINMAX)

cv2.imwrite(
    os.path.join(id_dir, "%s_%d.jpg") %
    (specific_object_name, frame_id), img)

max_w = width - 256
max_h = height - 256

# 随机选取左上角坐标
select_x = random.randint(1, max_w)
select_y = random.randint(1, max_h)
rand_box = [select_x, select_y, select_x + 256, select_y + 256]

# 背景图保存位置
bg_dir = os.path.join(reid_dst_path, "bg")
if not os.path.exists(bg_dir):
    os.makedirs(bg_dir)

iou_list = []

for idx in range(len(box_list)):
    cow_box = box_list[idx]
    iou = bbox_iou(cow_box, rand_box)
    iou_list.append(iou)

# print("iou list:" , iou_list)

if np.array(iou_list).all() < 0:
    img = extractVideoImgs(frame_id, video_frame_save_path,
                           rand_box)

    if img is None:
        print(specific_object_name + "is empty")
        continue
    normalizedImg = np.zeros((256, 256))
    img = cv2.normalize(img, normalizedImg, 0, 255,
                        cv2.NORM_MINMAX)
    cv2.imwrite(
        os.path.join(bg_dir, "bg_%s_%d.jpg") %
        (video_name, frame_id), img)

f_txt.close()
```

```

postprocess(video_frame_save_path)
shutil.move(video_path, done_dir)
shutil.move(txt_path, done_dir)

```

数据集配套代码在: https://github.com/pprp/reid_for_deepsort

3. DarkLabel 转 MOT16 格式

其实 DarkLabel 标注得到信息和 MOT16 是几乎一致的, 只不过需要转化一下, 脚本如下:

```

import os
'''
gt.txt:
-----
frame(从 1 开始计), id, box(left top w, h), ignore=1(不忽略), class=1(从 1 开始),
↪ 覆盖 =1),
1,1,1363,569,103,241,1,1,0.86014
2,1,1362,568,103,241,1,1,0.86173
3,1,1362,568,103,241,1,1,0.86173
4,1,1362,568,103,241,1,1,0.86173

cutout24_gt.txt
---
frame(从 0 开始计), 数量, id(从 0 开始), box(x1,y1,x2,y2), class=null
0,4,0,450,194,558,276,null,1,408,147,469,206,null,2,374,199,435,307,null,3,153,213,218,3
1,4,0,450,194,558,276,null,1,408,147,469,206,null,2,374,199,435,307,null,3,153,213,218,3
2,4,0,450,194,558,276,null,1,408,147,469,206,null,2,374,199,435,307,null,3,153,213,218,3
'''

def xyxy2xywh(x):
    # Convert bounding box format from [x1, y1, x2, y2] to [x, y, w, h]
    # y = torch.zeros_like(x) if isinstance(x,
    #                                     torch.Tensor) else np.zeros_like(x)
    y = [0, 0, 0, 0]

    y[0] = (x[0] + x[2]) / 2
    y[1] = (x[1] + x[3]) / 2
    y[2] = x[2] - x[0]
    y[3] = x[3] - x[1]
    return y

def process_darklabel(video_label_path, mot_label_path):

```

```
f = open(video_label_path, "r")
f_o = open(mot_label_path, "w")

contents = f.readlines()

for line in contents:
    line = line[:-1]
    num_list = [num for num in line.split(',')]

    frame_id = int(num_list[0]) + 1
    total_num = int(num_list[1])

    base = 2

    for i in range(total_num):

        print(base, base + i * 6, base + i * 6 + 4)

        _id = int(num_list[base + i * 6]) + 1
        _box_x1 = int(num_list[base + i * 6 + 1])
        _box_y1 = int(num_list[base + i * 6 + 2])
        _box_x2 = int(num_list[base + i * 6 + 3])
        _box_y2 = int(num_list[base + i * 6 + 4])

        y = xyxy2xywh([_box_x1, _box_y1, _box_x2, _box_y2])

        write_line = "%d,%d,%d,%d,%d,%d,1,1,1\n" % (frame_id, _id, y[0],
                                                    y[1], y[2], y[3])

        f_o.write(write_line)

f.close()
f_o.close()

if __name__ == "__main__":
    root_dir = "./data/videosample"

    for item in os.listdir(root_dir):
        full_path = os.path.join(root_dir, item)

        video_path = os.path.join(full_path, item+".mp4")
        video_label_path = os.path.join(full_path, item + "_gt.txt")
        mot_label_path = os.path.join(full_path, "gt.txt")
```

```
process_darklabel(video_label_path, mot_label_path)
```

DarkLabel 软件可以到公众号 GiantPandaCV 后台回复“darklabel”获取。

以上就是 DarkLabel 转各种数据集格式脚本文本了，DarkLabel 还是非常方便的，可以快速构建自己的数据集。通常两分钟的视频可以生成 2880 张之多的图片，但是在目标检测中并不推荐将所有的图片都作为训练集，因为前后帧之间差距太小了，几乎是一模一样的。这种数据会导致训练速度很慢、泛化能力变差。

有两种解决方案：

- 可以选择隔几帧选取一帧作为数据集，比如每隔 10 帧作为数据集。具体选择多少作为间隔还是具体问题具体分析，如果视频中变化目标变化较快，可以适当缩短间隔；如果视频中大部分都是静止对象，可以适当增大间隔。
- 还有一种更好的方案是：对原视频用 ffmpeg 提取关键帧，将关键帧的内容作为数据集。关键帧和关键帧之间的差距比较大，适合作为目标检测数据集。

四、DeepSORT 论文解析

Deep SORT 论文核心内容，包括状态估计、匹配方法、级联匹配、表观模型等核心内容。

1. 简介

Simple Online and Realtime Tracking(SORT) 是一个非常简单、有效、实用的多目标跟踪算法。在 SORT 中，仅仅通过 IOU 来进行匹配虽然速度非常快，但是 ID switch 依然非常大。

本文提出了 Deep SORT 算法，相比 SORT，通过集成表观信息来提升 SORT 的表现。通过这个扩展，模型能够更好地处理目标被长时间遮挡的情况，将 ID switch 指标降低了 45%。表观信息也就是目标对应的特征，论文中通过在大型行人重识别数据集上训练得到的深度关联度量来提取表观特征 (借用了 ReID 领域的模型)。

2. 方法

2.1 状态估计

延续 SORT 算法使用 8 维的状态空间 $(u, v, r, h, \dot{x}, \dot{y}, \dot{r}, \dot{h})$ ，其中 (u, v) 代表 bbox 的中心点，宽高比 r ，高 h 以及对应的在图像坐标上的相对速度。

论文使用具有等速运动和线性观测模型的标准卡尔曼滤波器，将以上 8 维状态作为物体状态的直接观测模型。

每一个轨迹，都计算当前帧距上次匹配成功帧的差值，代码中对应 `time_since_update` 变量。该变量在卡尔曼滤波器 `predict` 的时候递增，在轨迹和 `detection` 关联的时候重置为 0。

超过最大年龄 A_{max} 的轨迹被认为离开图片区域，将从轨迹集合中删除，被设置为删除状态。代码中最大年龄默认值为 70，是级联匹配中的循环次数。

如果 `detection` 没有和现有 `track` 匹配上的，那么将对这个 `detection` 进行初始化，转变为新的 `Track`。新的 `Track` 初始化的时候的状态是未确定态，只有满足连续三帧都成功匹配，才能将未确定态转化为确定态。

如果处于未确定态的 `Track` 没有在 `n_init` 帧中匹配上 `detection`，将变为删除态，从轨迹集合中删除。

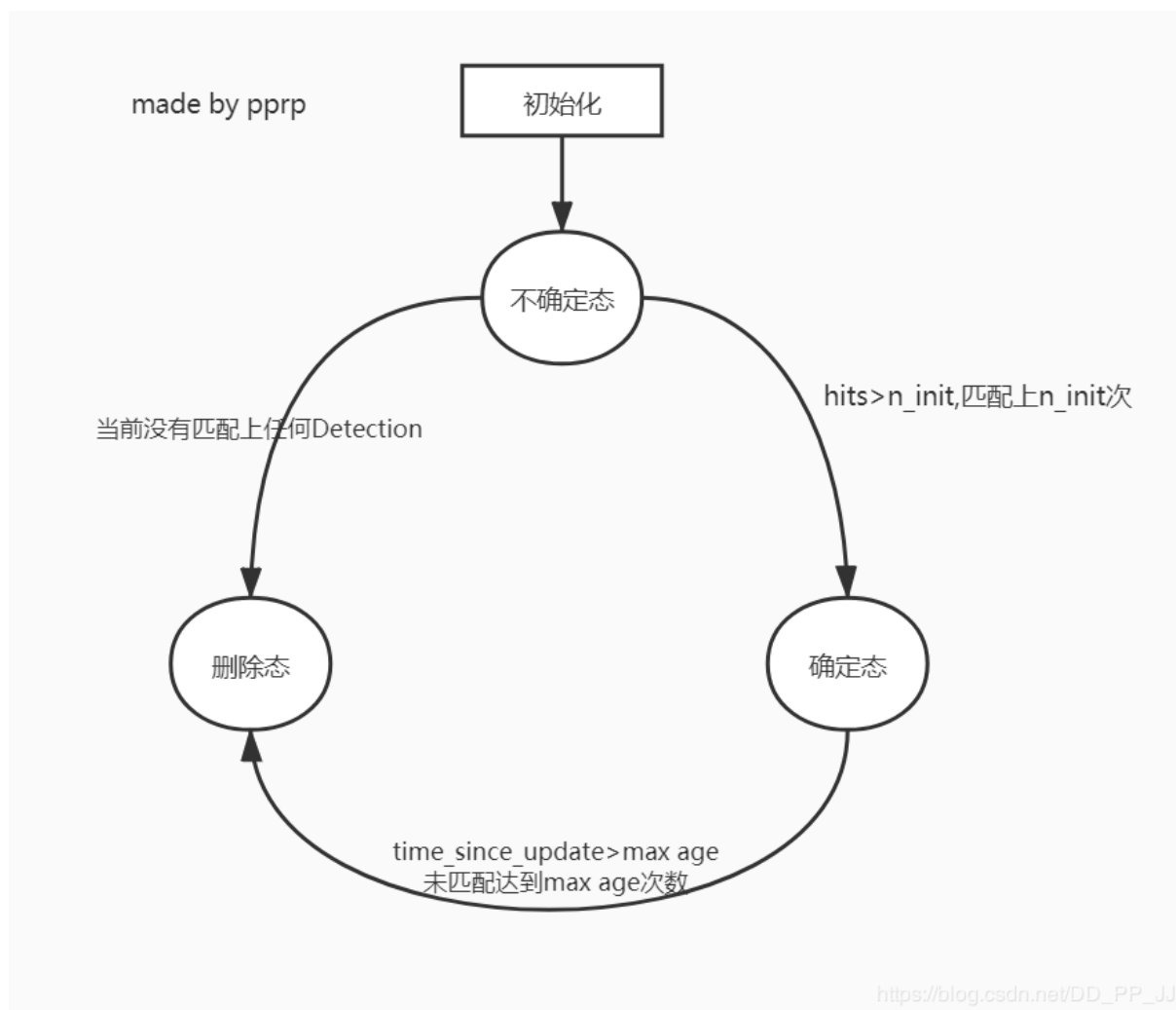


Figure 2: 状态转换

2.2 匹配问题

Assignment Problem 指派或者匹配问题，在这里主要是匹配轨迹 **Track** 和观测结果 **Detection**。这种匹配问题经常是使用匈牙利算法 (或者 **KM** 算法) 来解决，该算法求解对象是一个代价矩阵，所以首先讨论一下如何求代价矩阵：

- 使用平方马氏距离来度量 **Track** 和 **Detection** 之间的距离，由于两者使用的是高斯分布来进行表示的，很适合使用马氏距离来度量两个分布之间的距离。马氏距离又称为协方差距离，是一种有效计算两个未知样本集相似度的方法，所以在这里度量 **Track** 和 **Detection** 的匹配程度。

$$d^{(1)}(i, j) = (d_j - y_i)^T S_i^{-1} (d_j - y_i)$$

$$b_{i,j}^{(1)} = 1[d^{(1)}(i,j) \leq t^{(1)}]$$

d_j 代表第 j 个 **detection**, y_i 代表第 i 个 **track**, S_i^{-1} 代表 d 和 y 的协方差。

第二个公式是一个指示器, 比较的是马氏距离和卡方分布的阈值, $t^{(1)}=9.4877$, 如果马氏距离小于该阈值, 代表成功匹配。

- 使用 **cosine** 距离来度量表观特征之间的距离, **reid** 模型抽出得到一个 128 维的向量, 使用余弦距离来进行比对:

$$d^{(2)}(i,j) = \min\{1 - r_j^T r_k^{(i)} | r_k^{(i)} \in R_i\}$$

$r_j^T r_k^{(i)}$ 计算的是余弦相似度, 而余弦距离 = 1 - 余弦相似度, 通过 **cosine** 距离来度量 **track** 的表观特征和 **detection** 对应的表观特征, 来更加准确地预测 ID。SORT 中仅仅用运动信息进行匹配会导致 ID Switch 比较严重, 引入外观模型 + 级联匹配可以缓解这个问题。

$$b_{i,j}^{(2)} = 1[d^{(2)}(i,j) \leq t^{(2)}]$$

同上, 余弦距离这部分也使用了一个指示器, 如果余弦距离小于 $t^{(2)}$, 则认为匹配上。这个阈值在代码中被设置为 0.2 (由参数 **max_dist** 控制), 这个属于超参数, 在人脸识别中一般设置为 0.6。

- 综合匹配度是通过运动模型和外观模型的加权得到的

$$c_{i,j} = \lambda d^{(1)}(i,j) + (1 - \lambda) d^{(2)}(i,j)$$

其中 λ 是一个超参数, 在代码中默认为 0。作者认为在摄像头有实质性移动的时候这样设置比较合适, 也就是在关联矩阵中只使用外观模型进行计算。但并不是说马氏距离在 Deep SORT 中毫无用处, 马氏距离会对外观模型得到的距离矩阵进行限制, 忽视掉明显不可行的分配。

$$b_{i,j} = \prod_{m=1}^2 b_{i,j}^{(m)}$$

$b_{i,j}$ 也是指示器, 只有 $b_{i,j} = 1$ 的时候才会被人为初步匹配上。

2.3 级联匹配

级联匹配是 Deep SORT 区别于 SORT 的一个核心算法, 致力于解决目标被长时间遮挡的情况。为了让当前 **Detection** 匹配上当前时刻较近的 **Track**, 匹配的时候 **Detection** 优先匹配消失时间较短的

Track。

当目标被长时间遮挡，之后卡尔曼滤波预测结果将增加非常大的不确定性 (因为在被遮挡这段时间没有观测对象来调整，所以不确定性会增加)，状态空间内的可观察性就会大大降低。

在两个 Track 竞争同一个 Detection 的时候，消失时间更长的 Track 往往匹配得到的马氏距离更小，使得 Detection 更可能和遮挡时间较长的 Track 相关联，这种情况会破坏一个 Track 的持续性，这也就是 SORT 中 ID Switch 太高的原因之一。

所以论文提出级联匹配：

Listing 1 Matching Cascade

Input: Track indices $\mathcal{T} = \{1, \dots, N\}$, Detection indices $\mathcal{D} = \{1, \dots, M\}$, Maximum age A_{\max}

- 1: Compute cost matrix $C = [c_{i,j}]$ using Eq. 5
- 2: Compute gate matrix $B = [b_{i,j}]$ using Eq. 6
- 3: Initialize set of matches $\mathcal{M} \leftarrow \emptyset$
- 4: Initialize set of unmatched detections $\mathcal{U} \leftarrow \mathcal{D}$
- 5: **for** $n \in \{1, \dots, A_{\max}\}$ **do**
- 6: Select tracks by age $\mathcal{T}_n \leftarrow \{i \in \mathcal{T} \mid a_i = n\}$
- 7: $[x_{i,j}] \leftarrow \text{min_cost_matching}(C, \mathcal{T}_n, \mathcal{U})$
- 8: $\mathcal{M} \leftarrow \mathcal{M} \cup \{(i, j) \mid b_{i,j} \cdot x_{i,j} > 0\}$
- 9: $\mathcal{U} \leftarrow \mathcal{U} \setminus \{j \mid \sum_i b_{i,j} \cdot x_{i,j} > 0\}$
- 10: **end for**
- 11: **return** \mathcal{M}, \mathcal{U}

https://blog.csdn.net/DD_PP_JJ

Figure 3: 级联匹配伪代码

伪代码中需要注意的是匹配顺序，优先匹配 age 比较小的轨迹，对应实现如下：

```
# 1. 分配 track_indices 和 detection_indices
if track_indices is None:
    track_indices = list(range(len(tracks)))

if detection_indices is None:
    detection_indices = list(range(len(detections)))
```

```
unmatched_detections = detection_indices

matches = []
# cascade_depth = max_age 默认为 70
for level in range(cascade_depth):
    if len(unmatched_detections) == 0: # No detections left
        break

    track_indices_l = [
        k for k in track_indices
        if tracks[k].time_since_update == 1 + level
    ]
    if len(track_indices_l) == 0: # Nothing to match at this level
        continue

    # 2. 级联匹配核心内容就是这个函数
    matches_l, _, unmatched_detections = \
        min_cost_matching( # max_distance=0.2
            distance_metric, max_distance, tracks, detections,
            track_indices_l, unmatched_detections)
    matches += matches_l
unmatched_tracks = list(set(track_indices) - set(k for k, _ in matches))
return matches, unmatched_tracks, unmatched_detections
```

在匹配的最后阶段还对 `unconfirmed` 和 `age=1` 的未匹配轨迹进行基于 IOU 的匹配 (和 SORT 一致)。这可以缓解因为表观突变或者部分遮挡导致的较大变化。

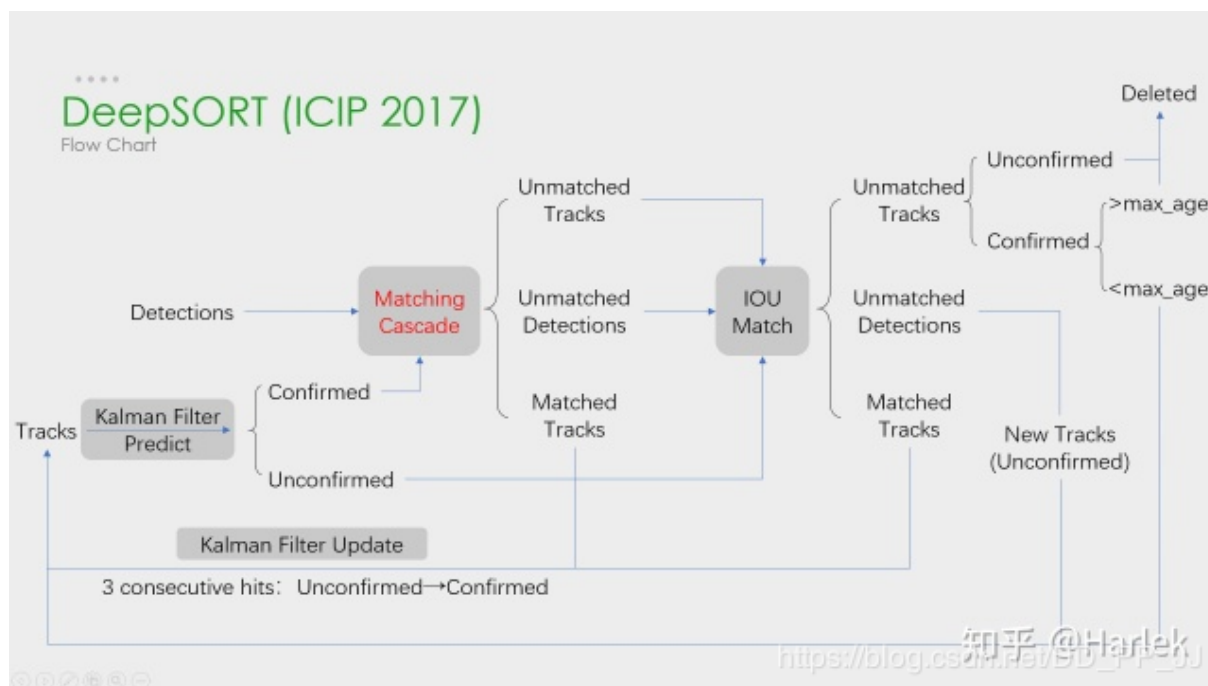


Figure 4: 知乎猫弟文章中的流程图

2.4 表观特征

表观特征这部分借用了行人重识别领域的网络模型，这部分的网络是需要提前离线学习好，其功能是提取出具有区分度的特征。

论文中用的是 wide residual network, 具体结构如下图所示:

Name	Patch Size/Stride	Output Size
Conv 1	$3 \times 3/1$	$32 \times 128 \times 64$
Conv 2	$3 \times 3/1$	$32 \times 128 \times 64$
Max Pool 3	$3 \times 3/2$	$32 \times 64 \times 32$
Residual 4	$3 \times 3/1$	$32 \times 64 \times 32$
Residual 5	$3 \times 3/1$	$32 \times 64 \times 32$
Residual 6	$3 \times 3/2$	$64 \times 32 \times 16$
Residual 7	$3 \times 3/1$	$64 \times 32 \times 16$
Residual 8	$3 \times 3/2$	$128 \times 16 \times 8$
Residual 9	$3 \times 3/1$	$128 \times 16 \times 8$
Dense 10		128
Batch and ℓ_2 normalization		128

Figure 5: CNN 网络结构

网络最后的输出是一个 128 维的向量用于代表该部分表观特征 (一般维度越高区分度越高带来的计算量越大)。最后使用了 L2 归一化来将特征映射到单位超球面上, 以便进一步使用余弦表观来度量相似度。

3. 实验

选用 MOTA、MOTP、MT、ML、FN、ID swiches、FM 等指标进行评估模型。

相比 SORT, Deep SORT 的 ID Switch 指标下降了 45%, 达到了当时的 SOTA。

经过实验, 发现 Deep SORT 的 MOTA、MOTP、MT、ML、FN 指标对于之前都有提升。

FP 很多, 主要是由于 Detection 和 Max age 过大导致的。

速度达到了 20Hz, 其中一半时间都花费在表观特征提取。

		MOTA ↑	MOTP ↑	MT ↑	ML ↓	ID ↓	FM ↓	FP ↓	FN ↓	Runtime ↑
KDNT [16]*	BATCH	68.2	79.4	41.0%	19.0%	933	1093	11479	45605	0.7 Hz
LMP_p [17]*	BATCH	71.0	80.2	46.9%	21.9%	434	587	7880	44564	0.5 Hz
MCMOT_HDM [18]	BATCH	62.4	78.3	31.5%	24.2%	1394	1318	9855	57257	35 Hz
NOMTwSDP16 [19]	BATCH	62.2	79.6	32.5%	31.1%	406	642	5119	63352	3 Hz
EAMTT [20]	ONLINE	52.5	78.8	19.0%	34.9%	910	1321	4407	81223	12 Hz
POI [16]*	ONLINE	66.1	79.5	34.0%	20.8%	805	3093	5061	55914	10 Hz
SORT [12]*	ONLINE	59.8	79.6	25.4%	22.7%	1423	1835	8698	63245	60 Hz
Deep SORT (Ours)*	ONLINE	61.4	79.1	32.8%	18.2%	781	2008	12852	56668	40 Hz

Figure 6: 实验对比

4. 总结

Deep SORT 可以看成三部分:

- 检测: 目标检测的效果对结果影响非常非常大, 并且 Recall 和 Precision 都应该很高才可以满足要求. 据笔者测试, 如果使用 yolov3 作为目标检测器, 目标跟踪过程中大概 60% 的时间都花费在 yolov3 上, 并且场景中的目标越多, 这部分耗时也越多 (NMS 花费的时间).
- 表观特征: 也就是 reid 模型, 原论文中用的是 wide residual network, 含有的参数量比较大, 可以考虑用新的、性能更好、参数量更低的 ReID 模型来完成这部分工作。笔者看到好多人推荐使用 OSNet, 但是实际使用的效果并不是特别好。
- 关联: 包括卡尔曼滤波算法和匈牙利算法。

改进空间:

最近非常多优秀的工作的思路是认为 reid 这部分特征提取和目标检测网络无法特征重用, 所以想将这两部分融合到一块。

JDE=YOLOv3 和 reid 融合

FairMOT=CenterNet 和 reid 融合

最近看了 CenterNet, 感觉这种无需 anchor 来匹配的方式非常优雅, 所以非常推荐 FairMOT, 效果非常出色, 适合作为研究的 baseline。

5. 参考

距离: https://blog.csdn.net/Kevin_cc98/article/details/73742037

论文地址: <https://arxiv.org/pdf/1703.07402.pdf>

代码地址: https://github.com/nwojke/deep_SORT

FairMOT: <https://github.com/ifzhang/FairMOT>

博客: <https://www.cnblogs.com/YiXiaoZhou/p/7074037.html>

五、DeepSORT 核心代码解析

Deep SORT 是多目标跟踪 (Multi-Object Tracking) 中常用到的一种算法，是一个 Detection Based Tracking 的方法。这个算法工业界关注度非常高，在知乎上有很多文章都是使用了 Deep SORT 进行工程部署。笔者将参考前辈的博客，结合自己的实践 (理论 & 代码) 对 Deep SORT 算法进行代码层面的解析。

在之前笔者写的一篇Deep SORT 论文阅读总结中，总结了 DeepSORT 论文中提到的核心观点，如果对 Deep SORT 不是很熟悉，可以先理解一下，然后再来看解读代码的部分。

1. MOT 主要步骤

在《DEEP LEARNING IN VIDEO MULTI-OBJECT TRACKING: A SURVEY》这篇基于深度学习的多目标跟踪的综述中，描述了 MOT 问题中四个主要步骤：

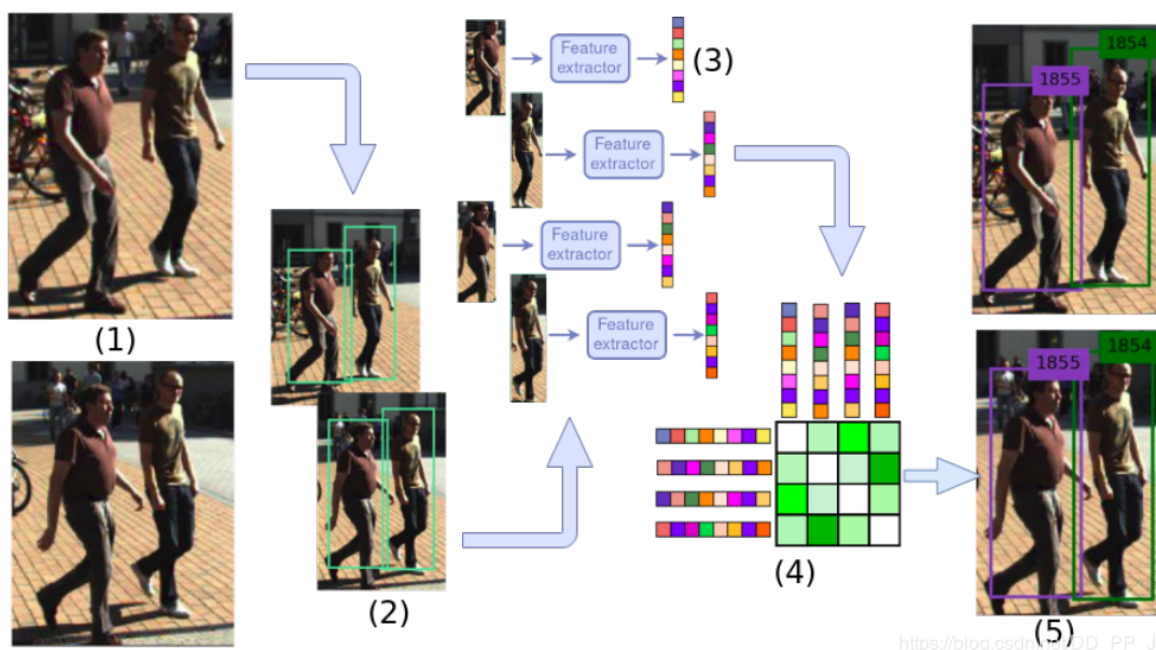


Figure 7: 多目标跟踪众多的主要步骤

- 给定视频原始帧。
- 运行目标检测器如 Faster R-CNN、YOLOv3、SSD 等进行检测，获取目标检测框。

- 将所有目标框中对应的目标抠出来，进行特征提取（包括表观特征或者运动特征）。
- 进行相似度计算，计算前后两帧目标之间的匹配程度（前后属于同一个目标的之间的距离比较小，不同目标的距离比较大）
- 数据关联，为每个对象分配目标的 ID。

以上就是四个核心步骤，其中核心是检测，SORT 论文的摘要中提到，仅仅换一个更好的检测器，就可以将目标跟踪表现提升 18.9%。

2. SORT

Deep SORT 算法的前身是 SORT, 全称是 Simple Online and Realtime Tracking。简单介绍一下，SORT 最大特点是基于 Faster R-CNN 的目标检测方法，并利用卡尔曼滤波算法 + 匈牙利算法，极大提高了多目标跟踪的速度，同时达到了 SOTA 的准确率。

这个算法确实是在实际应用中使用较为广泛的一个算法，核心就是两个算法：卡尔曼滤波和匈牙利算法。

卡尔曼滤波算法分为两个过程，预测和更新。该算法将目标的运动状态定义为 8 个正态分布的向量。

预测：当目标经过移动，通过上一帧的目标框和速度等参数，预测出当前帧的目标框位置和速度等参数。

更新：预测值和观测值，两个正态分布的状态进行线性加权，得到目前系统预测的状态。

匈牙利算法：解决的是一个分配问题，在 MOT 主要步骤中的计算相似度的，得到了前后两帧的相似度矩阵。匈牙利算法就是通过求解这个相似度矩阵，从而解决前后两帧真正匹配的目标。这部分 sklearn 库有对应的函数 `linear_assignment` 来进行求解。

SORT 算法中是通过前后两帧 IOU 来构建相似度矩阵，所以 SORT 计算速度非常快。

下图是一张 SORT 核心算法流程图：

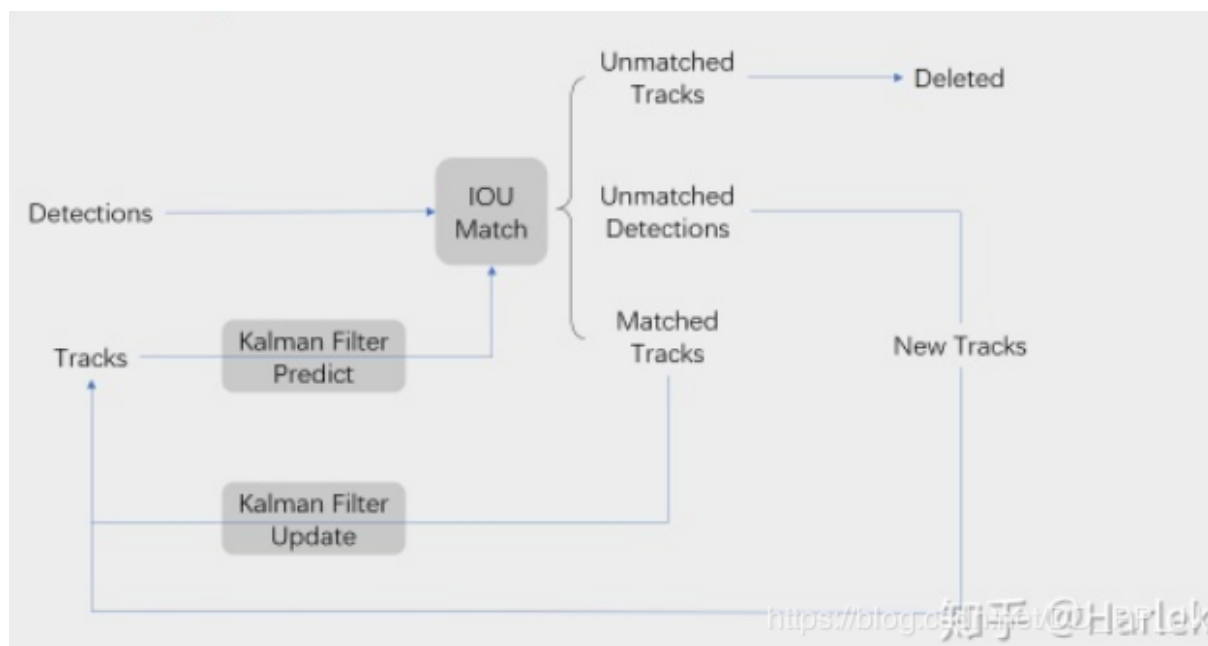


Figure 8: Harlek 提供的 SORT 解析图

Detections 是通过目标检测器得到的目标框，**Tracks** 是一段轨迹。核心是匹配的过程与卡尔曼滤波的预测和更新过程。

流程如下：目标检测器得到目标框 **Detections**，同时卡尔曼滤波器预测当前的帧的 **Tracks**，然后将 **Detections** 和 **Tracks** 进行 IOU 匹配，最终得到的结果分为：

- **Unmatched Tracks**，这部分被认为是失配，**Detection** 和 **Track** 无法匹配，如果失配持续了 T_{lost} 次，该目标 ID 将从图片中删除。
- **Unmatched Detections**，这部分说明没有任意一个 **Track** 能匹配 **Detection**，所以要为这个 **detection** 分配一个新的 **track**。
- **Matched Track**，这部分说明得到了匹配。

卡尔曼滤波可以根据 **Tracks** 状态预测下一帧的目标框状态。

卡尔曼滤波更新是对观测值 (匹配上的 **Track**) 和估计值更新所有 **track** 的状态。

3. Deep SORT

DeepSort 中最大的特点是加入外观信息，借用了 ReID 领域模型来提取特征，减少了 ID switch 的次数。整体流程图如下：

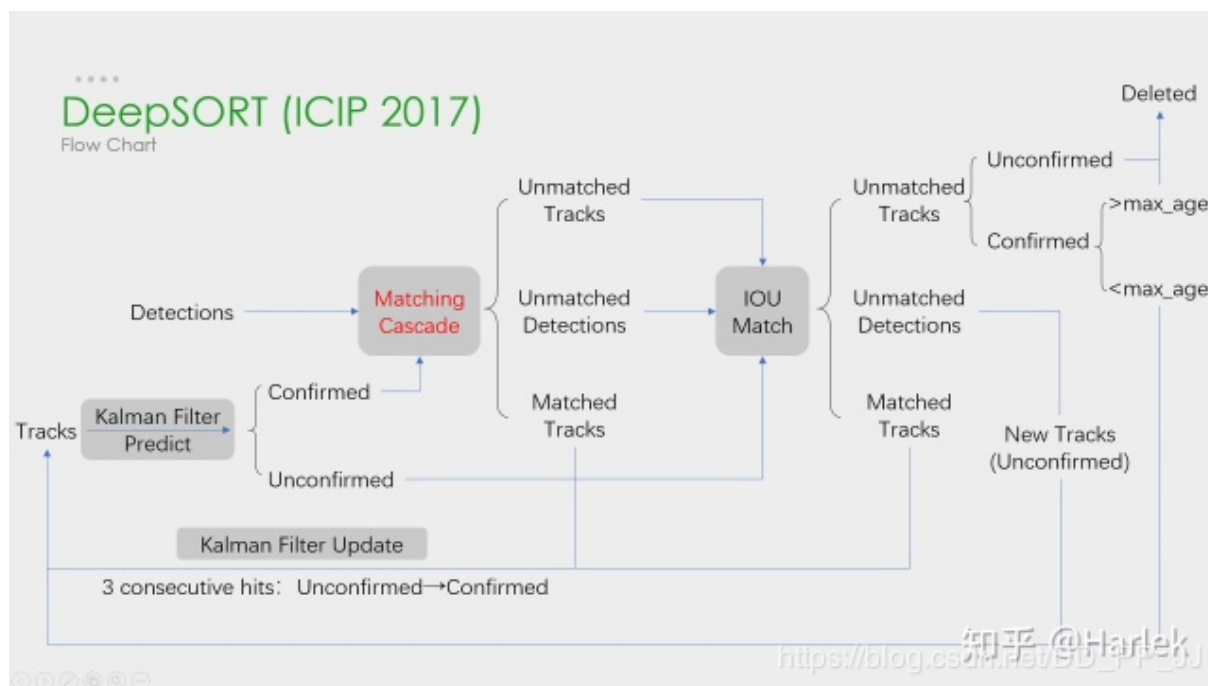


Figure 9: 图片来自知乎 Harlek

可以看出，Deep SORT 算法在 SORT 算法的基础上增加了级联匹配 (Matching Cascade)+ 新轨迹的确认 (confirmed)。总体流程就是：

- 卡尔曼滤波器预测轨迹 Tracks
- 使用匈牙利算法将预测得到的轨迹 Tracks 和当前帧中的 detections 进行匹配 (级联匹配和 IOU 匹配)
- 卡尔曼滤波更新。

其中上图中的级联匹配展开如下：

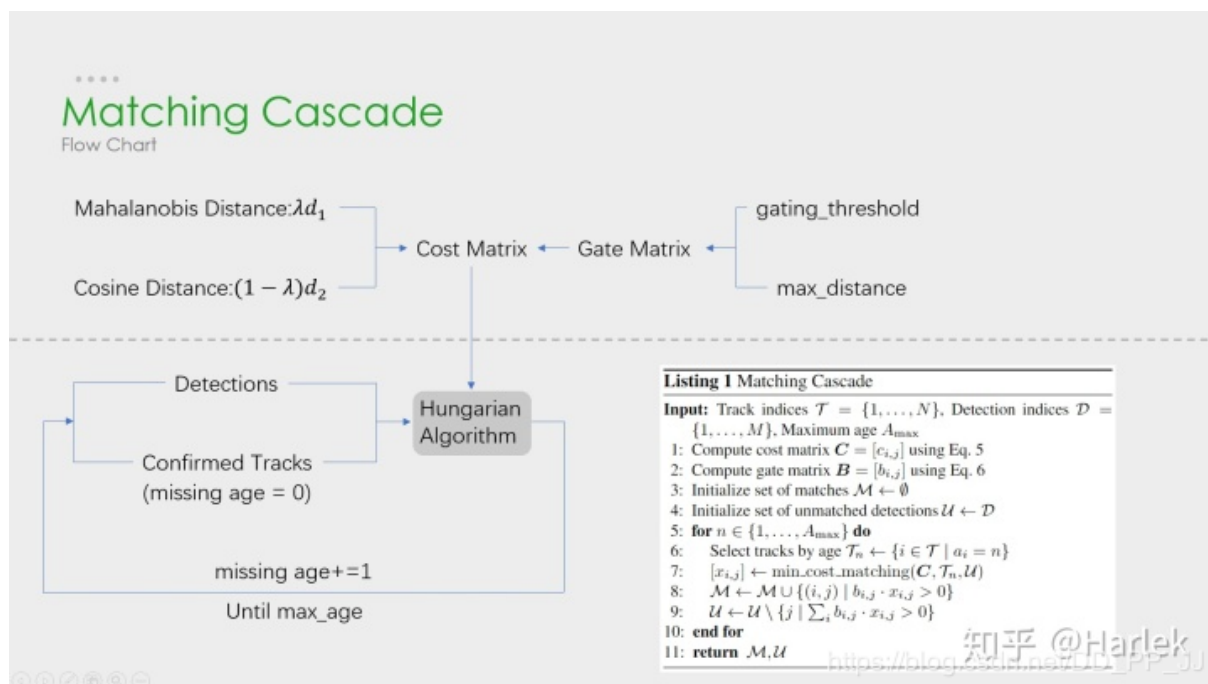


Figure 10: 图片来自知乎 Harlek

上图非常清晰地解释了如何进行级联匹配，上图由虚线划分为两部分：

上半部分中计算相似度矩阵的方法使用到了外观模型 (ReID) 和运动模型 (马氏距离) 来计算相似度，得到代价矩阵，另外一个则是门控矩阵，用于限制代价矩阵中过大的值。

下半部分中是级联匹配的数据关联步骤，匹配过程是一个循环 (max age 个迭代，默认为 70)，也就是从 missing age=0 到 missing age=70 的轨迹和 Detections 进行匹配，没有丢失过的轨迹优先匹配，丢失较为久远的就靠后匹配。通过这部分处理，可以重新将被遮挡目标找回，降低被遮挡然后再出现的目标发生的 ID Switch 次数。

将 Detection 和 Track 进行匹配，所以出现几种情况

1. Detection 和 Track 匹配，也就是 **Matched Tracks**。普通连续跟踪的目标都属于这种情况，前后两帧都有目标，能够匹配上。
2. Detection 没有找到匹配的 Track，也就是 **Unmatched Detections**。图像中突然出现新的目标的时候，Detection 无法在之前的 Track 找到匹配的目标。
3. Track 没有找到匹配的 Detection，也就是 **Unmatched Tracks**。连续追踪的目标超出图像区域，Track 无法与当前任意一个 Detection 匹配。
4. 以上没有涉及一种特殊的情况，就是两个目标遮挡的情况。刚刚被遮挡的目标的 Track 也无法匹配 Detection，目标暂时从图像中消失。之后被遮挡目标再次出现的时候，应该尽量让被遮挡目标分配的 ID 不发生变动，减少 ID Switch 出现的次数，这就需要用到级联匹配了。

4. Deep SORT 代码解析

论文中提供的代码是如下地址: https://github.com/nwojke/deep_sort

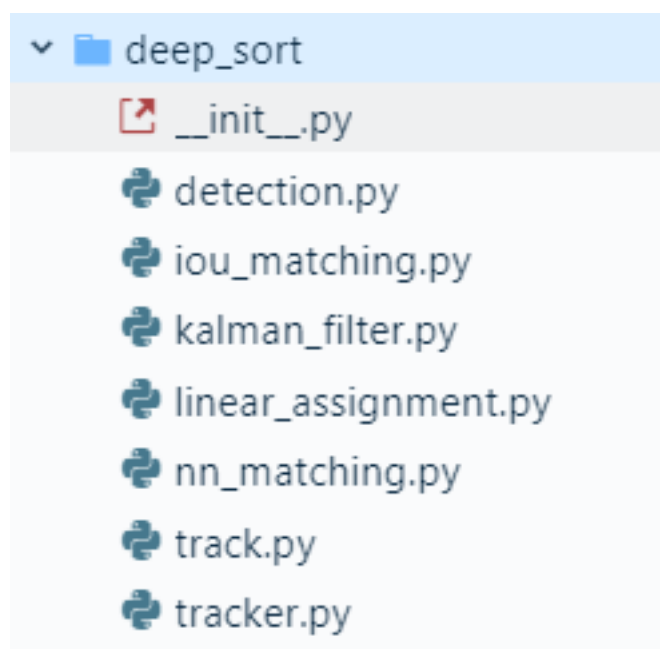


Figure 11: Github 库中 Deep_sort 文件结构

上图是 Github 库中有关 Deep SORT 的核心代码，不包括 Faster R-CNN 检测部分，所以主要将讲解这部分的几个文件，笔者也对其中核心代码进行了部分注释，地址在: https://github.com/pprp/deep_sort_yolov3_pytorch，将其中的目标检测器换成了 U 版的 yolov3，将 deep_sort 文件中的核心进行了调用。

4.1 类图

下图是笔者总结的这几个类调用的类图 (不是特别严谨，但是能大概展示各个模块的关系):

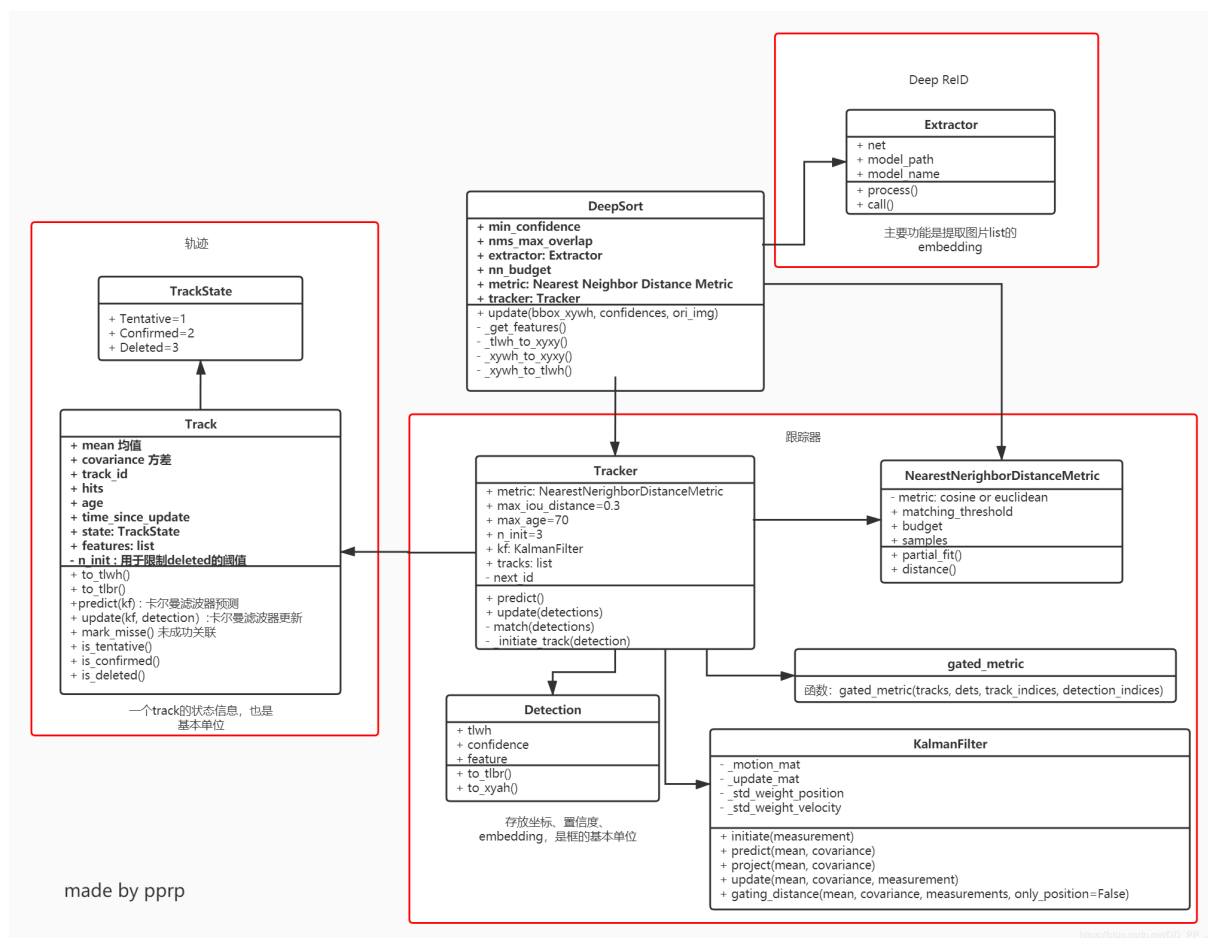


Figure 12: Deep Sort 类图

DeepSort 是核心类，调用其他模块，大体上可以分为三个模块：

- ReID 模块，用于提取表观特征，原论文中是生成了 128 维的 embedding。
- Track 模块，轨迹类，用于保存一个 Track 的状态信息，是一个基本单位。
- Tracker 模块，Tracker 模块掌握最核心的算法，卡尔曼滤波和匈牙利算法都是通过调用这个模块来完成的。

DeepSort 类对外接口非常简单：

```
self.deepsort = DeepSort(args.deepsort_checkpoint)# 实例化
outputs = self.deepsort.update(bbox_xcycwh, cls_conf, im)# 通过接收目标检测结果
↪ 进行更新
```

在外部调用的时候只需要以上两步即可，非常简单。

通过类图，对整体模块有了框架上理解，下面深入理解一下这些模块。

4.2 核心模块

```
Detection 类 python class Detection(object):          """          This class
represents a bounding box detection in a single image.          """
def __init__(self, tlwh, confidence, feature):          self.tlwh
= np.asarray(tlwh, dtype=np.float)          self.confidence =
float(confidence)          self.feature = np.asarray(feature,
dtype=np.float32)          def to_tlbr(self):          """Convert bounding
box to format `(min x, min y, max x, max y)`, i.e.,          `(top
left, bottom right)`.          """          ret = self.tlwh.copy()
ret[2:] += ret[:2]          return ret          def to_xyah(self):
"""Convert bounding box to format `(center x, center y, aspect ratio,
height)`, where the aspect ratio is `width / height`.          """
ret = self.tlwh.copy()          ret[:2] += ret[2:] / 2          ret[2]
/= ret[3]          return ret
```

Detection 类用于保存通过目标检测器得到的一个检测框，包含 **top left** 坐标 + 框的宽和高，以及该 **bbox** 的置信度还有通过 **reid** 获取得到的对应的 **embedding**。除此以外提供了不同 **bbox** 位置格式的转换方法：

- **tlwh**: 代表左上角坐标 + 宽高
- **tlbr**: 代表左上角坐标 + 右下角坐标
- **xyah**: 代表中心坐标 + 宽高比 + 高

Track 类

```
class Track:
    # 一个轨迹的信息，包含 (x,y,a,h) & v
    """
    A single target track with state space `(x, y, a, h)` and associated
    velocities, where `(x, y)` is the center of the bounding box, `a` is the
    aspect ratio and `h` is the height.
    """

    def __init__(self, mean, covariance, track_id, n_init, max_age,
                 feature=None):
        # max age 是一个存活期限，默认为 70 帧，在
        self.mean = mean
        self.covariance = covariance
        self.track_id = track_id
        self.hits = 1
```

```
# hits 和 n_init 进行比较
# hits 每次 update 的时候进行一次更新（只有 match 的时候才进行 update）
# hits 代表匹配上了多少次，匹配次数超过 n_init 就会设置为 confirmed 状态
self.age = 1 # 没有用到，和 time_since_update 功能重复
self.time_since_update = 0
# 每次调用 predict 函数的时候就会 +1
# 每次调用 update 函数的时候就会设置为 0

self.state = TrackState.Tentative
self.features = []
# 每个 track 对应多个 features，每次更新都将最新的 feature 添加到列表中
if feature is not None:
    self.features.append(feature)

self._n_init = n_init # 如果连续 n_init 帧都没有出现匹配，设置为 deleted 状态
↪
self._max_age = max_age # 上限
```

Track 类主要存储的是轨迹信息，mean 和 covariance 是保存的框的位置和速度信息，track_id 代表分配给这个轨迹的 ID。state 代表框的状态，有三种：

- **Tentative:** 不确定态，这种状态会在初始化一个 Track 的时候分配，并且只有在连续匹配上 n_init 帧才会转变为确定态。如果在处于不确定态的情况下没有匹配上任何 detection，那将转变为删除态。
- **Confirmed:** 确定态，代表该 Track 确实处于匹配状态。如果当前 Track 属于确定态，但是失配连续达到 max age 次数的时候，就会被转变为删除态。
- **Deleted:** 删除态，说明该 Track 已经失效。

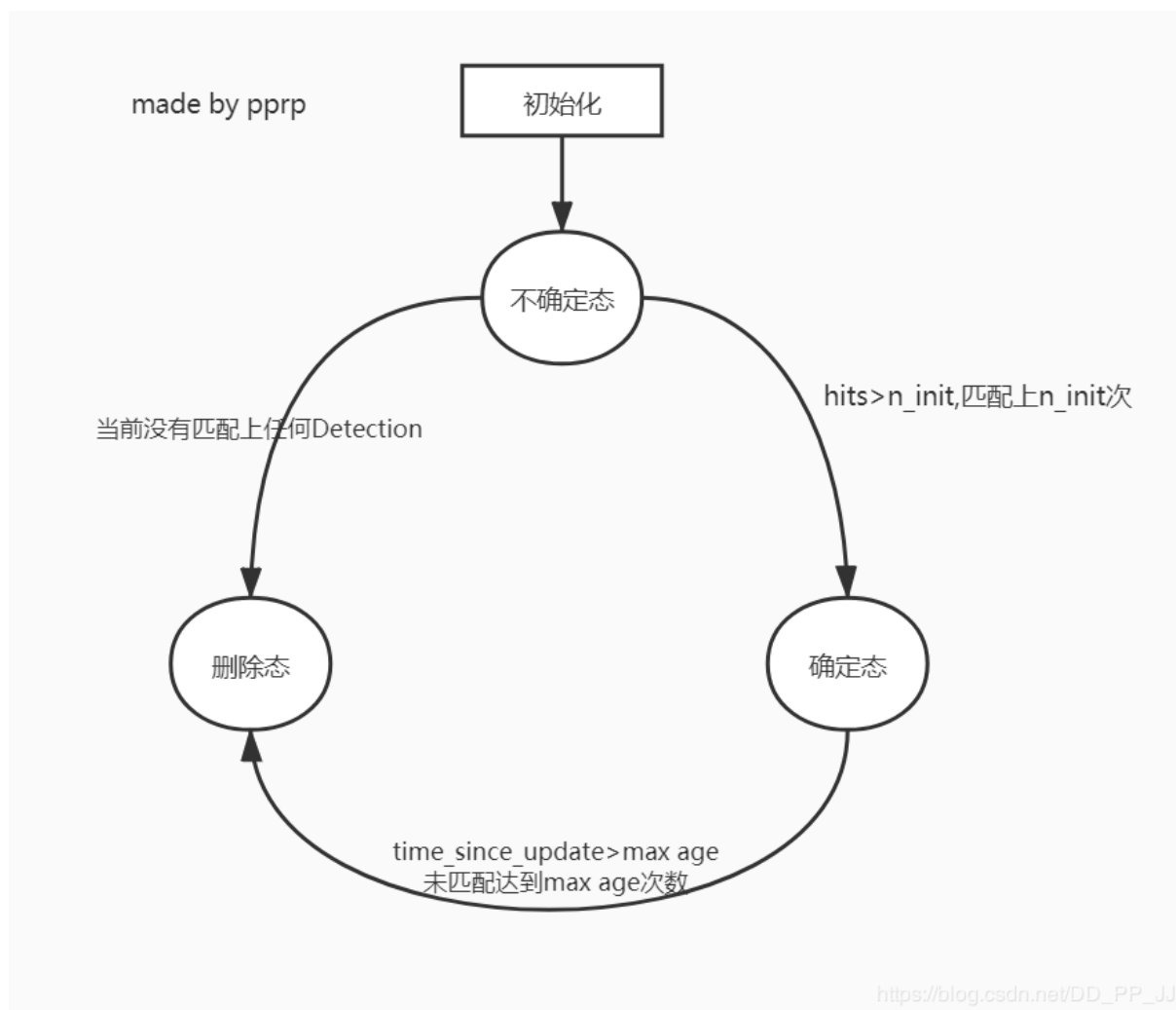


Figure 13: 状态转换图

max_age 代表一个 Track 存活期限，他和 **time_since_update** 变量进行比对。**time_since_update** 是每次轨迹调用 **update** 函数的时候就会 +1，每次调用 **predict** 的时候就会重置为 0，也就是说如果一个轨迹长时间没有 **update**(没有匹配上) 的时候，就会不断增加，直到 **time_since_update** 超过 **max age**(默认 70)，将这个 Track 从 Tracker 中的列表删除。

hits 代表连续确认多少次，用在从不确定态转为确定态的时候。每次 Track 进行 **update** 的时候，**hits** 就会 +1，如果 **hits > n_init**(默认为 3)，也就是连续三帧的该轨迹都得到了匹配，这时候才将不确定态转为确定态。

需要说明的是每个轨迹还有一个重要的变量，**features** 列表，存储该轨迹在不同帧对应位置通过 ReID 提取到的特征。为何要保存这个列表，而不是将其更新为当前最新的特征呢？这是为了解决目标被遮挡后再次出现的问题，需要从以往帧对应的特征进行匹配。另外，如果特征过多会严重拖慢计

算速度，所以有一个参数 **budget** 用来控制特征列表的长度，取最新的 **budget** 个 features, 将旧的删除掉。

ReID 特征提取部分 ReID 网络是独立于目标检测和跟踪器的模块，功能是提取对应 bounding box 中的 feature, 得到一个固定维度的 embedding 作为该 bbox 的代表，供计算相似度时使用。

```
class Extractor(object):
    def __init__(self, model_name, model_path, use_cuda=True):
        self.net = build_model(name=model_name,
                                num_classes=96)
        self.device = "cuda" if torch.cuda.is_available(
        ) and use_cuda else "cpu"
        state_dict = torch.load(model_path)['net_dict']
        self.net.load_state_dict(state_dict)
        print("Loading weights from {}... Done!".format(model_path))
        self.net.to(self.device)
        self.size = (128,128)
        self.norm = transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize([0.3568, 0.3141, 0.2781],
                                  [0.1752, 0.1857, 0.1879])
        ])

    def _preprocess(self, im_crops):
        """
        TODO:
            1. to float with scale from 0 to 1
            2. resize to (64, 128) as Market1501 dataset did
            3. concatenate to a numpy array
            3. to torch Tensor
            4. normalize
        """
        def _resize(im, size):
            return cv2.resize(im.astype(np.float32) / 255., size)

        im_batch = torch.cat([
            self.norm(_resize(im, self.size)).unsqueeze(0) for im in im_crops
        ],dim=0).float()
        return im_batch

    def __call__(self, im_crops):
        im_batch = self._preprocess(im_crops)
```

```

with torch.no_grad():
    im_batch = im_batch.to(self.device)
    features = self.net(im_batch)
    return features.cpu().numpy()

```

模型训练是按照传统 ReID 的方法进行，使用 Extractor 类的时候输入为一个 list 的图片，得到图片对应的特征。

NearestNeighborDistanceMetric 类 这个类中用到了两个计算距离的函数：

1. 计算欧氏距离

```

def _pdist(a, b):
    # 用于计算成对的平方距离
    # a NxM 代表 N 个对象，每个对象有 M 个数值作为 embedding 进行比较
    # b LxM 代表 L 个对象，每个对象有 M 个数值作为 embedding 进行比较
    # 返回的是 NxL 的矩阵，比如 dist[i][j] 代表 a[i] 和 b[j] 之间的平方和距离
    # 实现见: https://blog.csdn.net/frankzd/article/details/80251042
    a, b = np.asarray(a), np.asarray(b) # 拷贝一份数据
    if len(a) == 0 or len(b) == 0:
        return np.zeros((len(a), len(b)))
    a2, b2 = np.square(a).sum(axis=1), np.square(
        b).sum(axis=1) # 求每个 embedding 的平方和
    # sum(N) + sum(L) - 2 x [NxM]x[MxL] = [NxL]
    r2 = -2. * np.dot(a, b.T) + a2[:, None] + b2[None, :]
    r2 = np.clip(r2, 0., float(np.inf))
    return r2

```

$$\begin{aligned}
 & \text{dist} \\
 = & \sqrt{\begin{pmatrix} \|P_1\|^2 & \|P_1\|^2 & \cdots & \|P_1\|^2 \\ \|P_2\|^2 & \|P_2\|^2 & \cdots & \|P_2\|^2 \\ \vdots & \vdots & \ddots & \vdots \\ \|P_M\|^2 & \|P_M\|^2 & \cdots & \|P_M\|^2 \end{pmatrix} + \begin{pmatrix} \|C_1\|^2 & \|C_2\|^2 & \cdots & \|C_N\|^2 \\ \|C_1\|^2 & \|C_2\|^2 & \cdots & \|C_N\|^2 \\ \vdots & \vdots & \ddots & \vdots \\ \|C_1\|^2 & \|C_2\|^2 & \cdots & \|C_N\|^2 \end{pmatrix}} - 2 \times PC^T
 \end{aligned}$$

Figure 14: 图源自 csdn 博客

2. 计算余弦距离

```

def _cosine_distance(a, b, data_is_normalized=False):
    # a 和 b 之间的余弦距离
    # a : [NxM] b : [LxM]

```

```

# 余弦距离 = 1 - 余弦相似度
# https://blog.csdn.net/u013749540/article/details/51813922
if not data_is_normalized:
    # 需要将余弦相似度转化成类似欧氏距离的余弦距离。
    a = np.asarray(a) / np.linalg.norm(a, axis=1, keepdims=True)
    # np.linalg.norm 操作是求向量的范式，默认是 L2 范式，等同于求向量的欧式距离。
    b = np.asarray(b) / np.linalg.norm(b, axis=1, keepdims=True)
return 1. - np.dot(a, b.T)

```

$$\cos \theta = \frac{\sum_{i=1}^n (A_i \times B_i)}{\sqrt{\sum_{i=1}^n (A_i)^2} \times \sqrt{\sum_{i=1}^n (B_i)^2}}$$

$$= \frac{A^T \cdot B}{\|A\| \times \|B\|}$$

https://blog.csdn.net/DD_PP_JJ

Figure 15: 图源 csdn 博客

以上代码对应公式，注意余弦距离 = 1 - 余弦相似度。

最近邻距离度量类

```

class NearestNeighborDistanceMetric(object):
    # 对于每个目标，返回一个最近的距离
    def __init__(self, metric, matching_threshold, budget=None):
        # 默认 matching_threshold = 0.2 budget = 100
        if metric == "euclidean":
            # 使用最近邻欧氏距离
            self._metric = _nn_euclidean_distance
        elif metric == "cosine":
            # 使用最近邻余弦距离
            self._metric = _nn_cosine_distance
        else:
            raise ValueError("Invalid metric; must be either 'euclidean' or
                               'cosine'")

```

```

self.matching_threshold = matching_threshold
# 在级联匹配的函数中调用
self.budget = budget
# budget 预算, 控制 feature 的多少
self.samples = {}
# samples 是一个字典 {id->feature list}

def partial_fit(self, features, targets, active_targets):
    # 作用: 部分拟合, 用新的数据更新测量距离
    # 调用: 在特征集更新模块部分调用, tracker.update() 中
    for feature, target in zip(features, targets):
        self.samples.setdefault(target, []).append(feature)
        # 对应目标下添加新的 feature, 更新 feature 集合
        # 目标 id : feature list
        if self.budget is not None:
            self.samples[target] = self.samples[target][-self.budget:]
        # 设置预算, 每个类最多多少个目标, 超过直接忽略

    # 筛选激活的目标
    self.samples = {k: self.samples[k] for k in active_targets}

def distance(self, features, targets):
    # 作用: 比较 feature 和 targets 之间的距离, 返回一个代价矩阵
    # 调用: 在匹配阶段, 将 distance 封装为 gated_metric,
    #       进行外观信息 (reid 得到的深度特征)+
    #       运动信息 (马氏距离用于度量两个分布相似程度)
    cost_matrix = np.zeros((len(targets), len(features)))
    for i, target in enumerate(targets):
        cost_matrix[i, :] = self._metric(self.samples[target], features)
    return cost_matrix

```

Tracker 类 Tracker 类是最核心的类, Tracker 中保存了所有的轨迹信息, 负责初始化第一帧的轨迹、卡尔曼滤波的预测和更新、负责级联匹配、IOU 匹配等等核心工作。

```

class Tracker:
    # 是一个多目标 tracker, 保存了很多个 track 轨迹
    # 负责调用卡尔曼滤波来预测 track 的新状态 + 进行匹配工作 + 初始化第一帧
    # Tracker 调用 update 或 predict 的时候, 其中的每个 track 也会各自调用自己的
    #   ↪ update 或 predict
    """
    This is the multi-target tracker.

```

```

"""

def __init__(self, metric, max_iou_distance=0.7, max_age=70, n_init=3):
    # 调用的时候，后边的参数全部是默认的
    self.metric = metric
    # metric 是一个类，用于计算距离（余弦距离或马氏距离）
    self.max_iou_distance = max_iou_distance
    # 最大 iou, iou 匹配的时候使用
    self.max_age = max_age
    # 直接指定级联匹配的 cascade_depth 参数
    self.n_init = n_init
    # n_init 代表需要 n_init 次数的 update 才会将 track 状态设置为 confirmed

    self.kf = kalman_filter.KalmanFilter()# 卡尔曼滤波器
    self.tracks = [] # 保存一系列轨迹
    self._next_id = 1 # 下一个分配的轨迹 id
def predict(self):
    # 遍历每个 track 都进行一次预测
    """Propagate track state distributions one time step forward.

    This function should be called once every time step, before `update`.
    """
    for track in self.tracks:
        track.predict(self.kf)

```

然后来看最核心的 update 函数和 match 函数，可以对照下面的流程图一起看：

update 函数

```

def update(self, detections):
    # 进行测量的更新和轨迹管理
    """Perform measurement update and track management.

    Parameters
    -----
    detections : List[deep_sort.detection.Detection]
        A list of detections at the current time step.

    """
    # Run matching cascade.
    matches, unmatched_tracks, unmatched_detections = \
        self._match(detections)

    # Update track set.

```

```

# 1. 针对匹配上的结果
for track_idx, detection_idx in matches:
    # track 更新对应的 detection
    self.tracks[track_idx].update(self.kf, detections[detection_idx])

# 2. 针对未匹配的 tracker, 调用 mark_missed 标记
# track 失配, 若待定则删除, 若 update 时间很久也删除
# max age 是一个存活期限, 默认为 70 帧
for track_idx in unmatched_tracks:
    self.tracks[track_idx].mark_missed()

# 3. 针对未匹配的 detection, detection 失配, 进行初始化
for detection_idx in unmatched_detections:
    self._initiate_track(detections[detection_idx])

# 得到最新的 tracks 列表, 保存的是标记为 confirmed 和 Tentative 的 track
self.tracks = [t for t in self.tracks if not t.is_deleted()]

# Update distance metric.
active_targets = [t.track_id for t in self.tracks if t.is_confirmed()]
# 获取所有 confirmed 状态的 track id
features, targets = [], []
for track in self.tracks:
    if not track.is_confirmed():
        continue
    features += track.features # 将 tracks 列表拼接到 features 列表
    # 获取每个 feature 对应的 track id
    targets += [track.track_id for _ in track.features]
    track.features = []

# 距离度量中的 特征集更新
self.metric.partial_fit(np.asarray(features), np.asarray(targets),
                        active_targets)

```

match 函数:

```

def _match(self, detections):
    # 主要功能是进行匹配, 找到匹配的, 未匹配的部分
    def gated_metric(tracks, dets, track_indices, detection_indices):
        # 功能: 用于计算 track 和 detection 之间的距离, 代价函数
        # 需要使用在 KM 算法之前
        # 调用:
        # cost_matrix = distance_metric(tracks, detections,
        #                                track_indices, detection_indices)

```

```

features = np.array([dets[i].feature for i in detection_indices])
targets = np.array([tracks[i].track_id for i in track_indices])

# 1. 通过最近邻计算出代价矩阵 cosine distance
cost_matrix = self.metric.distance(features, targets)
# 2. 计算马氏距离, 得到新的状态矩阵
cost_matrix = linear_assignment.gate_cost_matrix(
    self.kf, cost_matrix, tracks, dets, track_indices,
    detection_indices)
return cost_matrix

# Split track set into confirmed and unconfirmed tracks.
# 划分不同轨迹的状态
confirmed_tracks = [
    i for i, t in enumerate(self.tracks) if t.is_confirmed()
]
unconfirmed_tracks = [
    i for i, t in enumerate(self.tracks) if not t.is_confirmed()
]

# 进行级联匹配, 得到匹配的 track、不匹配的 track、不匹配的 detection
'''
!!!!!!!!!!!!!!
级联匹配
!!!!!!!!!!!!!!
'''
# gated_metric->cosine distance
# 仅仅对确定态的轨迹进行级联匹配
matches_a, unmatched_tracks_a, unmatched_detections = \
    linear_assignment.matching_cascade(
        gated_metric,
        self.metric.matching_threshold,
        self.max_age,
        self.tracks,
        detections,
        confirmed_tracks)

# 将所有状态为未确定态的轨迹和刚刚没有匹配上的轨迹组合为 iou_track_candidates,
# 进行 IoU 的匹配
iou_track_candidates = unconfirmed_tracks + [
    k for k in unmatched_tracks_a
    if self.tracks[k].time_since_update == 1 # 刚刚没有匹配上
]

```

```

# 未匹配
unmatched_tracks_a = [
    k for k in unmatched_tracks_a
    if self.tracks[k].time_since_update != 1 # 已经很久没有匹配上
]

'''
!!!!!!!!!!!!
IOU 匹配
对级联匹配中还没有匹配成功的目标再进行 IoU 匹配
!!!!!!!!!!!!
'''

# 虽然和级联匹配中使用的都是 min_cost_matching 作为核心,
# 这里使用的 metric 是 iou cost 和以上不同
matches_b, unmatched_tracks_b, unmatched_detections = \
    linear_assignment.min_cost_matching(
        iou_matching.iou_cost,
        self.max_iou_distance,
        self.tracks,
        detections,
        iou_track_candidates,
        unmatched_detections)

matches = matches_a + matches_b # 组合两部分 match 得到的结果

unmatched_tracks = list(set(unmatched_tracks_a + unmatched_tracks_b))
return matches, unmatched_tracks, unmatched_detections

```

以上两部分结合注释和以下流程图可以更容易理解。

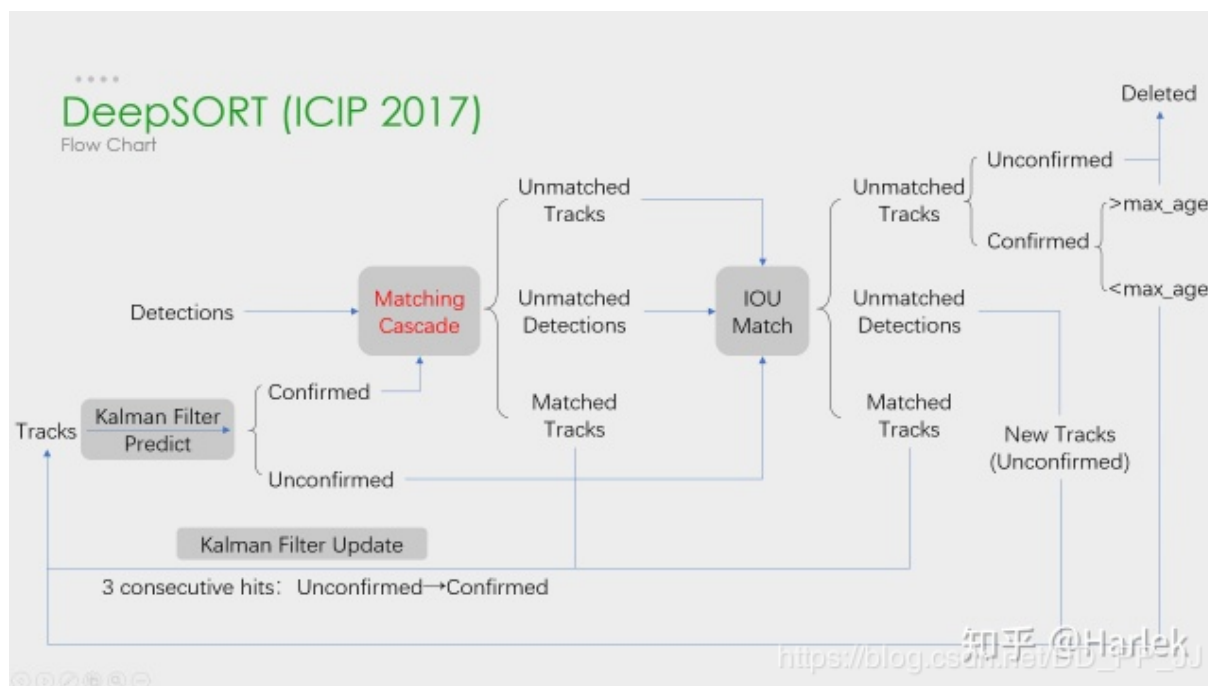


Figure 16: 图片来自知乎 Harlek

级联匹配 下边是论文中给出的级联匹配的伪代码:

Listing 1 Matching Cascade

Input: Track indices $\mathcal{T} = \{1, \dots, N\}$, Detection indices $\mathcal{D} = \{1, \dots, M\}$, Maximum age A_{\max}

- 1: Compute cost matrix $C = [c_{i,j}]$ using Eq. 5
- 2: Compute gate matrix $B = [b_{i,j}]$ using Eq. 6
- 3: Initialize set of matches $\mathcal{M} \leftarrow \emptyset$
- 4: Initialize set of unmatched detections $\mathcal{U} \leftarrow \mathcal{D}$
- 5: **for** $n \in \{1, \dots, A_{\max}\}$ **do**
- 6: Select tracks by age $\mathcal{T}_n \leftarrow \{i \in \mathcal{T} \mid a_i = n\}$
- 7: $[x_{i,j}] \leftarrow \text{min_cost_matching}(C, \mathcal{T}_n, \mathcal{U})$
- 8: $\mathcal{M} \leftarrow \mathcal{M} \cup \{(i, j) \mid b_{i,j} \cdot x_{i,j} > 0\}$
- 9: $\mathcal{U} \leftarrow \mathcal{U} \setminus \{j \mid \sum_i b_{i,j} \cdot x_{i,j} > 0\}$
- 10: **end for**
- 11: **return** \mathcal{M}, \mathcal{U}

https://blog.csdn.net/DD_PP_U

Figure 17: 论文中的级联匹配的伪代码

以下代码是伪代码对应的实现

```
# 1. 分配 track_indices 和 detection_indices
if track_indices is None:
    track_indices = list(range(len(tracks)))

if detection_indices is None:
    detection_indices = list(range(len(detections)))

unmatched_detections = detection_indices

matches = []
# cascade_depth = max age 默认为 70
for level in range(cascade_depth):
    if len(unmatched_detections) == 0: # No detections left
        break

    track_indices_l = [
```

```

        k for k in track_indices
        if tracks[k].time_since_update == 1 + level
    ]
    if len(track_indices_l) == 0: # Nothing to match at this level
        continue

    # 2. 级联匹配核心内容就是这个函数
    matches_l, _, unmatched_detections = \
        min_cost_matching( # max_distance=0.2
            distance_metric, max_distance, tracks, detections,
            track_indices_l, unmatched_detections)
    matches += matches_l
    unmatched_tracks = list(set(track_indices) - set(k for k, _ in matches))

```

门控矩阵 门控矩阵的作用就是通过计算卡尔曼滤波的状态分布和测量值之间的距离对代价矩阵进行限制。

代价矩阵中的距离是 Track 和 Detection 之间的表观相似度，假如一个轨迹要去匹配两个表观特征非常相似的 Detection，这样就很容易出错，但是这个时候分别让两个 Detection 计算与这个轨迹的马氏距离，并使用一个阈值 `gating_threshold` 进行限制，所以就可以将马氏距离较远的那个 Detection 区分开，可以降低错误的匹配。

```

def gate_cost_matrix(
    kf, cost_matrix, tracks, detections, track_indices,
    ↪ detection_indices,
    gated_cost=INFTY_COST, only_position=False):
    # 根据通过卡尔曼滤波获得的状态分布，使成本矩阵中的不可行条目无效。
    gating_dim = 2 if only_position else 4
    gating_threshold = kalman_filter.chi2inv95[gating_dim] # 9.4877

    measurements = np.asarray([detections[i].to_xyah()
                                for i in detection_indices])
    for row, track_idx in enumerate(track_indices):
        track = tracks[track_idx]
        gating_distance = kf.gating_distance(
            track.mean, track.covariance, measurements, only_position)
        cost_matrix[row, gating_distance >
                    gating_threshold] = gated_cost # 设置为 inf
    return cost_matrix

```

卡尔曼滤波器 在 Deep SORT 中，需要估计 Track 的以下状态：

- 均值：用 8 维向量 $(x, y, a, h, vx, vy, va, vh)$ 表示。 (x, y) 是框的中心坐标，宽高比是 a , 高度 h 以及对应的速度，所有的速度都将初始化为 0。
- 协方差：表示目标位置信息的不确定程度，用 8×8 的对角矩阵来表示，矩阵对应的值越大，代表不确定程度越高。

下图代表卡尔曼滤波器主要过程：

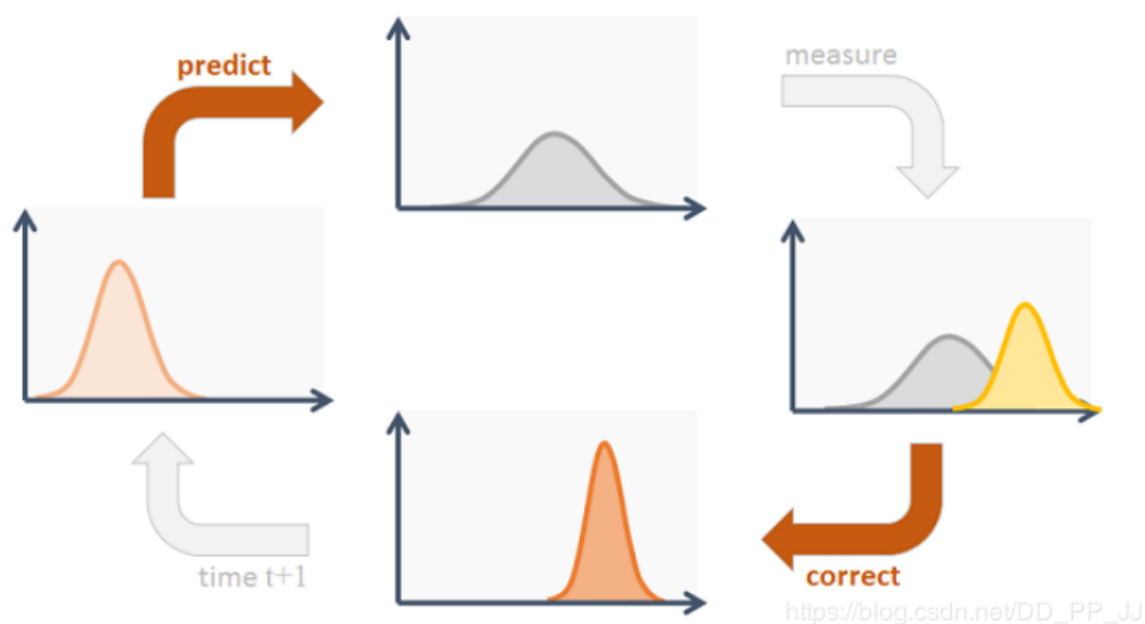


Figure 18: DeepSORT: Deep Learning to Track Custom Objects in a Video

1. 卡尔曼滤波首先根据当前帧 ($\text{time}=t$) 的状态进行预测，得到预测下一帧的状态 ($\text{time}=t+1$)
2. 得到测量结果，在 Deep SORT 中对应的测量就是 Detection，即目标检测器提供的检测框。
3. 将预测结果和测量结果进行更新。

下面这部分主要参考：<https://zhuanlan.zhihu.com/p/90835266>

如果对卡尔曼滤波算法有较为深入的了解，可以结合卡尔曼滤波算法和代码进行理解。

预测分两个公式：

第一个公式：

$$x' = Fx$$

其中 F 是状态转移矩阵，如下图：

$$\begin{pmatrix} cx \\ cy \\ w \\ h \\ vx \\ vy \\ vw \\ vh \end{pmatrix}_{t+1} = \begin{pmatrix} 1 & 0 & 0 & 0 & dt & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & dt & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & dt & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & dt \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} cx \\ cy \\ w \\ h \\ vx \\ vy \\ vw \\ vh \end{pmatrix}_t$$

$\underbrace{\quad}_{x'} = \underbrace{\quad}_{F} \cdot \underbrace{\quad}_{x_t}$

https://blog.csdn.net/qq_37614340

Figure 19: 图源知乎 @ 求索

第二个公式:

$$P' = FPF^T + Q$$

P 是当前帧 (time=t) 的协方差, Q 是卡尔曼滤波器的运动估计误差, 代表不确定程度。

```
def predict(self, mean, covariance):
    # 相当于得到 t 时刻估计值
    # Q 预测过程中噪声协方差
    std_pos = [
        self._std_weight_position * mean[3],
        self._std_weight_position * mean[3],
        1e-2,
        self._std_weight_position * mean[3]]

    std_vel = [
        self._std_weight_velocity * mean[3],
        self._std_weight_velocity * mean[3],
        1e-5,
        self._std_weight_velocity * mean[3]]

    # np.r_ 按列连接两个矩阵
    # 初始化噪声矩阵 Q
```

```

motion_cov = np.diag(np.square(np.r_[std_pos, std_vel]))

#  $x' = Fx$ 
mean = np.dot(self._motion_mat, mean)

#  $P' = FPF^T + Q$ 
covariance = np.linalg.multi_dot((
    self._motion_mat, covariance, self._motion_mat.T)) + motion_cov

return mean, covariance

```

更新的公式

$$y = z - Hx'$$

$$S = HP'H^T + R$$

$$K = P'H^TS^{-1}$$

$$x = x' + Ky$$

$$P = (I - KH)P'$$

```

def project(self, mean, covariance):
    # R 测量过程中噪声的协方差
    std = [
        self._std_weight_position * mean[3],
        self._std_weight_position * mean[3],
        1e-1,
        self._std_weight_position * mean[3]]

    # 初始化噪声矩阵 R
    innovation_cov = np.diag(np.square(std))

    # 将均值向量映射到检测空间, 即  $Hx'$ 
    mean = np.dot(self._update_mat, mean)

    # 将协方差矩阵映射到检测空间, 即  $HP'H^T$ 
    covariance = np.linalg.multi_dot((
        self._update_mat, covariance, self._update_mat.T))

```

```

    return mean, covariance + innovation_cov

def update(self, mean, covariance, measurement):
    # 通过估计值和观测值估计最新结果

    # 将均值和协方差映射到检测空间, 得到  $Hx'$  和  $S$ 
    projected_mean, projected_cov = self.project(mean, covariance)

    # 矩阵分解
    chol_factor, lower = scipy.linalg.cho_factor(
        projected_cov, lower=True, check_finite=False)

    # 计算卡尔曼增益  $K$ 
    kalman_gain = scipy.linalg.cho_solve(
        (chol_factor, lower), np.dot(covariance, self._update_mat.T).T,
        check_finite=False).T

    #  $z - Hx'$ 
    innovation = measurement - projected_mean

    #  $x = x' + Ky$ 
    new_mean = mean + np.dot(innovation, kalman_gain.T)

    #  $P = (I - KH)P'$ 
    new_covariance = covariance - np.linalg.multi_dot((
        kalman_gain, projected_cov, kalman_gain.T))
    return new_mean, new_covariance

```

$$y = z - Hx'$$

这个公式中, z 是 Detection 的 mean, 不包含变化值, 状态为 $[cx, cy, a, h]$ 。H 是测量矩阵, 将 Track 的均值向量 x' 映射到检测空间。计算的 y 是 Detection 和 Track 的均值误差。

$$S = HP'H^T + R$$

R 是目标检测器的噪声矩阵, 是一个 4×4 的对角矩阵。对角线上的值分别为中心点两个坐标以及宽高的噪声。

$$K = P'H^TS^{-1}$$

计算的是卡尔曼增益，是作用于衡量估计误差的权重。

$$x = x' + Ky$$

更新后的均值向量 x 。

$$P = (I - KH)P'$$

更新后的协方差矩阵。

卡尔曼滤波笔者理解也不是很深入，没有推导过公式，对这部分感兴趣的推荐几个博客：

1. 卡尔曼滤波+python 写的 demo: https://zhuanlan.zhihu.com/p/113685503?utm_source=wechat_session&utm_medium=web
2. 详解 + 推导: <https://blog.csdn.net/honyniu/article/details/88697520>

5. 流程解析

流程部分主要按照以下流程图来走一遍：

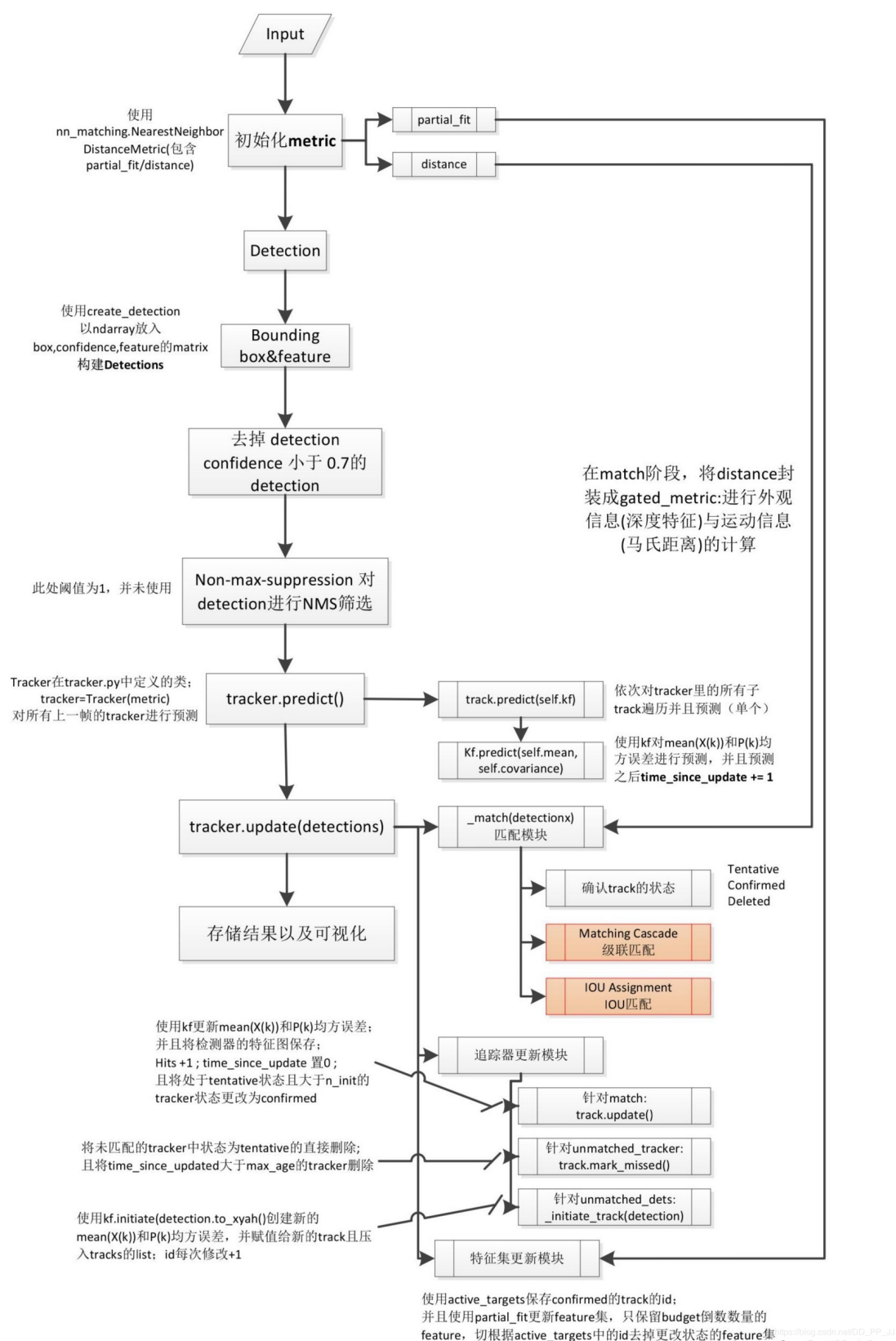


Figure 20: 知乎 @ 猫弟总结的 deep sort 流程图

感谢知乎 @ 猫弟总结的流程图，讲解非常地清晰，如果单纯看代码，非常容易混淆。比如说代价矩阵的计算这部分，连续套了三个函数，才被真正调用。上图将整体流程总结地非常棒。笔者将参考以上流程结合代码进行梳理：

1. 分析 detector 类中的 Deep SORT 调用：

```
class Detector(object):
    def __init__(self, args):
        self.args = args
        if args.display:
            cv2.namedWindow("test", cv2.WINDOW_NORMAL)
            cv2.resizeWindow("test", args.display_width, args.display_height)

        device = torch.device(
            'cuda') if torch.cuda.is_available() else torch.device('cpu')

        self.vdo = cv2.VideoCapture()
        self.yolo3 = InferYOLOv3(args.yolo_cfg,
                                args.img_size,
                                args.yolo_weights,
                                args.data_cfg,
                                device,
                                conf_thres=args.conf_thresh,
                                nms_thres=args.nms_thresh)

        self.deepsort = DeepSort(args.deepsort_checkpoint)
```

初始化 DeepSORT 对象，更新部分接收目标检测得到的框的位置，置信度和图片：

```
outputs = self.deepsort.update(bbox_xcycwh, cls_conf, im)
```

2. 顺着 DeepSORT 类的 update 函数看

```
class DeepSort(object):
    def __init__(self, model_path, max_dist=0.2):
        self.min_confidence = 0.3
        # yolov3 中检测结果置信度阈值，筛选置信度小于 0.3 的 detection。

        self.nms_max_overlap = 1.0
        # 非极大抑制阈值，设置为 1 代表不进行抑制

        # 用于提取图片的 embedding，返回的是一个 batch 图片对应的特征
        self.extractor = Extractor("resnet18",
                                   model_path,
                                   use_cuda=True)
```

```

max_cosine_distance = max_dist
# 用在级联匹配的地方，如果大于改阈值，就直接忽略
nn_budget = 100
# 预算，每个类别最多的样本个数，如果超过，删除旧的

# 第一个参数可选 'cosine' or 'euclidean'
metric = NearestNeighborDistanceMetric("cosine",
                                       max_cosine_distance,
                                       nn_budget)

self.tracker = Tracker(metric)

def update(self, bbox_xywh, confidences, ori_img):
    self.height, self.width = ori_img.shape[:2]
    # generate detections
    features = self._get_features(bbox_xywh, ori_img)
    # 从原图中 crop bbox 对应图片并计算得到 embedding
    bbox_tlwh = self._xywh_to_tlwh(bbox_xywh)

    detections = [
        Detection(bbox_tlwh[i], conf, features[i])
        for i, conf in enumerate(confidences) if conf >
            self.min_confidence
    ] # 筛选小于 min_confidence 的目标，并构造一个 Detection 对象构成的列表
    # Detection 是一个存储图中一个 bbox 结果
    # 需要: 1. bbox(tlwh 形式) 2. 对应置信度 3. 对应 embedding

    # run on non-maximum supression
    boxes = np.array([d.tlwh for d in detections])
    scores = np.array([d.confidence for d in detections])

    # 使用非极大抑制
    # 默认 nms_thres=1 的时候开启也没有用，实际上并没有进行非极大抑制
    indices = non_max_suppression(boxes, self.nms_max_overlap, scores)
    detections = [detections[i] for i in indices]

    # update tracker
    # tracker 给出一个预测结果，然后将 detection 传入，进行卡尔曼滤波操作
    self.tracker.predict()
    self.tracker.update(detections)

    # output bbox identities
    # 存储结果以及可视化
    outputs = []

```

```

    for track in self.tracker.tracks:
        if not track.is_confirmed() or track.time_since_update > 1:
            continue
        box = track.to_tlwh()
        x1, y1, x2, y2 = self._tlwh_to_xyxy(box)
        track_id = track.track_id
        outputs.append(np.array([x1, y1, x2, y2, track_id],
↪ dtype=np.int))

    if len(outputs) > 0:
        outputs = np.stack(outputs, axis=0)
    return np.array(outputs)

```

从这里开始对照以上流程图会更加清晰。在 Deep SORT 初始化的过程中有一个核心 metric，Nearest-NeighborDistanceMetric 类会在匹配和特征集更新的时候用到。

梳理 DeepSORT 的 update 流程：

- 根据传入的参数（bbox_xywh, conf, img）使用 ReID 模型提取对应 bbox 的表观特征。
- 构建 detections 的列表，列表中的内容就是 Detection 类, 在此处限制了 bbox 的最小置信度。
- 使用非极大抑制算法，由于默认 nms_thres=1，实际上并没有用。
- Tracker 类进行一次预测，然后将 detections 传入，进行更新。
- 最后将 Tracker 中保存的轨迹中状态属于确认态的轨迹返回。

以上核心在 Tracker 的 predict 和 update 函数，接着梳理。

3. Tracker 的 predict 函数

Tracker 是一个多目标跟踪器，保存了很多个 track 轨迹，负责调用卡尔曼滤波来预测 track 的新状态 + 进行匹配工作 + 初始化第一帧。Tracker 调用 update 或 predict 的时候，其中的每个 track 也会各自调用自己的 update 或 predict

```

class Tracker:
    def __init__(self, metric, max_iou_distance=0.7, max_age=70, n_init=3):
        # 调用的时候，后边的参数全部是默认的
        self.metric = metric
        self.max_iou_distance = max_iou_distance
        # 最大 iou, iou 匹配的时候使用
        self.max_age = max_age
        # 直接指定级联匹配的 cascade_depth 参数
        self.n_init = n_init
        # n_init 代表需要 n_init 次数的 update 才会将 track 状态设置为 confirmed

```

```

self.kf = kalman_filter.KalmanFilter() # 卡尔曼滤波器
self.tracks = [] # 保存一系列轨迹
self._next_id = 1 # 下一个分配的轨迹 id

def predict(self):
    # 遍历每个 track 都进行一次预测
    """Propagate track state distributions one time step forward.
    This function should be called once every time step, before `update`.
    """

    for track in self.tracks:
        track.predict(self.kf)

```

predict 主要是对轨迹列表中所有的轨迹使用卡尔曼滤波算法进行状态的预测。

4. Tracker 的更新

Tracker 的更新属于最核心的部分。

```

def update(self, detections):
    # 进行测量的更新和轨迹管理
    """Perform measurement update and track management.

    Parameters
    -----
    detections : List[deep_sort.detection.Detection]
        A list of detections at the current time step.

    """

    # Run matching cascade.
    matches, unmatched_tracks, unmatched_detections = \
        self._match(detections)

    # Update track set.
    # 1. 针对匹配上的结果
    for track_idx, detection_idx in matches:
        # track 更新对应的 detection
        self.tracks[track_idx].update(self.kf, detections[detection_idx])

    # 2. 针对未匹配的 tracker, 调用 mark_missed 标记
    # track 失配, 若待定则删除, 若 update 时间很久也删除
    # max age 是一个存活期限, 默认为 70 帧
    for track_idx in unmatched_tracks:
        self.tracks[track_idx].mark_missed()

```

```

# 3. 针对未匹配的 detection, detection 失配, 进行初始化
for detection_idx in unmatched_detections:
    self._initiate_track(detections[detection_idx])

# 得到最新的 tracks 列表, 保存的是标记为 confirmed 和 Tentative 的 track
self.tracks = [t for t in self.tracks if not t.is_deleted()]

# Update distance metric.
active_targets = [t.track_id for t in self.tracks if
↪ t.is_confirmed()]
# 获取所有 confirmed 状态的 track id
features, targets = [], []
for track in self.tracks:
    if not track.is_confirmed():
        continue
    features += track.features # 将 tracks 列表拼接到 features 列表
    # 获取每个 feature 对应的 track id
    targets += [track.track_id for _ in track.features]
    track.features = []

# 距离度量中的 特征集更新
self.metric.partial_fit(np.asarray(features),
↪ np.asarray(targets), active_targets)

```

这部分注释已经很详细了, 主要是一些后处理代码, 需要关注的是对匹配上的, 未匹配的 *Detection*, 未匹配的 *Track* 三者进行的处理以及最后进行特征集更新部分, 可以对照流程图梳理。

Tracker 的 *update* 函数的核心函数是 *match* 函数, 描述如何进行匹配的流程:

```

def _match(self, detections):
    # 主要功能是进行匹配, 找到匹配的, 未匹配的部分
    def gated_metric(tracks, dets, track_indices, detection_indices):
        # 功能: 用于计算 track 和 detection 之间的距离, 代价函数
        # 需要使用在 KM 算法之前
        # 调用:
        # cost_matrix = distance_metric(tracks, detections,
        #                               track_indices, detection_indices)
        features = np.array([dets[i].feature for i in detection_indices])
        targets = np.array([tracks[i].track_id for i in track_indices])

        # 1. 通过最近邻计算出代价矩阵 cosine distance
        cost_matrix = self.metric.distance(features, targets)

```

```

# 2. 计算马氏距离, 得到新的状态矩阵
cost_matrix = linear_assignment.gate_cost_matrix(
    self.kf, cost_matrix, tracks, dets, track_indices,
    detection_indices)
return cost_matrix

# Split track set into confirmed and unconfirmed tracks.
# 划分不同轨迹的状态
confirmed_tracks = [
    i for i, t in enumerate(self.tracks) if t.is_confirmed()
]
unconfirmed_tracks = [
    i for i, t in enumerate(self.tracks) if not t.is_confirmed()
]

# 进行级联匹配, 得到匹配的 track、不匹配的 track、不匹配的 detection
'''
!!!!!!!!!!!!
级联匹配
!!!!!!!!!!!!
'''
# gated_metric->cosine distance
# 仅仅对确定态的轨迹进行级联匹配
matches_a, unmatched_tracks_a, unmatched_detections = \
    linear_assignment.matching_cascade(
        gated_metric,
        self.metric.matching_threshold,
        self.max_age,
        self.tracks,
        detections,
        confirmed_tracks)

# 将所有状态为未确定态的轨迹和刚刚没有匹配上的轨迹组合为 iou_track_candidates,
# 进行 IoU 的匹配
iou_track_candidates = unconfirmed_tracks + [
    k for k in unmatched_tracks_a
    if self.tracks[k].time_since_update == 1 # 刚刚没有匹配上
]
# 未匹配
unmatched_tracks_a = [
    k for k in unmatched_tracks_a
    if self.tracks[k].time_since_update != 1 # 已经很久没有匹配上
]

```

```

'''
!!!!!!!!!!!!
IOU 匹配
对级联匹配中还没有匹配成功的目标再进行 IoU 匹配
!!!!!!!!!!!!
'''
# 虽然和级联匹配中使用的都是 min_cost_matching 作为核心,
# 这里使用的 metric 是 iou cost 和以上不同
matches_b, unmatched_tracks_b, unmatched_detections = \
    linear_assignment.min_cost_matching(
        iou_matching.iou_cost,
        self.max_iou_distance,
        self.tracks,
        detections,
        iou_track_candidates,
        unmatched_detections)

matches = matches_a + matches_b # 组合两部分 match 得到的结果

unmatched_tracks = list(set(unmatched_tracks_a + unmatched_tracks_b))
return matches, unmatched_tracks, unmatched_detections

```

对照下图来看会顺畅很多:


```

if detection_indices is None:
    detection_indices = list(range(len(detections)))

unmatched_detections = detection_indices

matches = []
# cascade_depth = max_age 默认为 70
for level in range(cascade_depth):
    if len(unmatched_detections) == 0: # No detections left
        break

    track_indices_l = [
        k for k in track_indices
        if tracks[k].time_since_update == 1 + level
    ]
    if len(track_indices_l) == 0: # Nothing to match at this level
        continue

    # 2. 级联匹配核心内容就是这个函数
    matches_l, _, unmatched_detections = \
        min_cost_matching( # max_distance=0.2
            distance_metric, max_distance, tracks, detections,
            track_indices_l, unmatched_detections)
    matches += matches_l
    unmatched_tracks = list(set(track_indices) - set(k for k, _ in matches))
return matches, unmatched_tracks, unmatched_detections

```

可以看到和伪代码是一致的，文章上半部分也有提到这部分代码。这部分代码中还有一个核心的函数 `min_cost_matching`，这个函数可以接收不同的 `distance_metric`，在级联匹配和 IoU 匹配中都有用到。

`min_cost_matching` 函数：

```

def min_cost_matching(
    distance_metric, max_distance, tracks, detections,
    ↪ track_indices=None,
    detection_indices=None):

    if track_indices is None:
        track_indices = np.arange(len(tracks))
    if detection_indices is None:
        detection_indices = np.arange(len(detections))

```

```

if len(detection_indices) == 0 or len(track_indices) == 0:
    return [], track_indices, detection_indices # Nothing to match.
# -----
# Gated_distance——>
#     1. cosine distance
#     2. 马氏距离
# 得到代价矩阵
# -----
# iou_cost——>
#     仅仅计算 track 和 detection 之间的 iou 距离
# -----
cost_matrix = distance_metric(
    tracks, detections, track_indices, detection_indices)
# -----
# gated_distance 中设置距离中最高上限，
# 这里最远距离实际是在 deep sort 类中的 max_dist 参数设置的
# 默认 max_dist=0.2， 距离越小越好
# -----
# iou_cost 情况下，max_distance 的设置对应 tracker 中的 max_iou_distance，
# 默认值为 max_iou_distance=0.7
# 注意结果是 1-iou，所以越小越好
# -----
cost_matrix[cost_matrix > max_distance] = max_distance + 1e-5

# 匈牙利算法或者 KM 算法
row_indices, col_indices = linear_assignment(cost_matrix)

matches, unmatched_tracks, unmatched_detections = [], [], []

# 这几个 for 循环用于对匹配结果进行筛选，得到匹配和未匹配的结果
for col, detection_idx in enumerate(detection_indices):
    if col not in col_indices:
        unmatched_detections.append(detection_idx)

for row, track_idx in enumerate(track_indices):
    if row not in row_indices:
        unmatched_tracks.append(track_idx)

for row, col in zip(row_indices, col_indices):
    track_idx = track_indices[row]
    detection_idx = detection_indices[col]
    if cost_matrix[row, col] > max_distance:
        unmatched_tracks.append(track_idx)

```

```

        unmatched_detections.append(detection_idx)
    else:
        matches.append((track_idx, detection_idx))
# 得到匹配, 未匹配轨迹, 未匹配检测
    return matches, unmatched_tracks, unmatched_detections

```

注释中提到 `distance_metric` 是有两个的:

- 第一个是级联匹配中传入的 `distance_metric` 是 `gated_metric`, 其内部核心是计算的表观特征的级联匹配。

```

def gated_metric(tracks, dets, track_indices, detection_indices):
    # 功能: 用于计算 track 和 detection 之间的距离, 代价函数
    #         需要使用在 KM 算法之前
    # 调用:
    # cost_matrix = distance_metric(tracks, detections,
    #                               track_indices, detection_indices)
    features = np.array([dets[i].feature for i in detection_indices])
    targets = np.array([tracks[i].track_id for i in track_indices])

    # 1. 通过最近邻计算出代价矩阵 cosine distance
    cost_matrix = self.metric.distance(features, targets)

    # 2. 计算马氏距离, 得到新的状态矩阵
    cost_matrix = linear_assignment.gate_cost_matrix(
        self.kf, cost_matrix, tracks, dets, track_indices,
        detection_indices)
    return cost_matrix

```

对应下图进行理解 (下图上半部分就是对应的 `gated_metric` 函数):

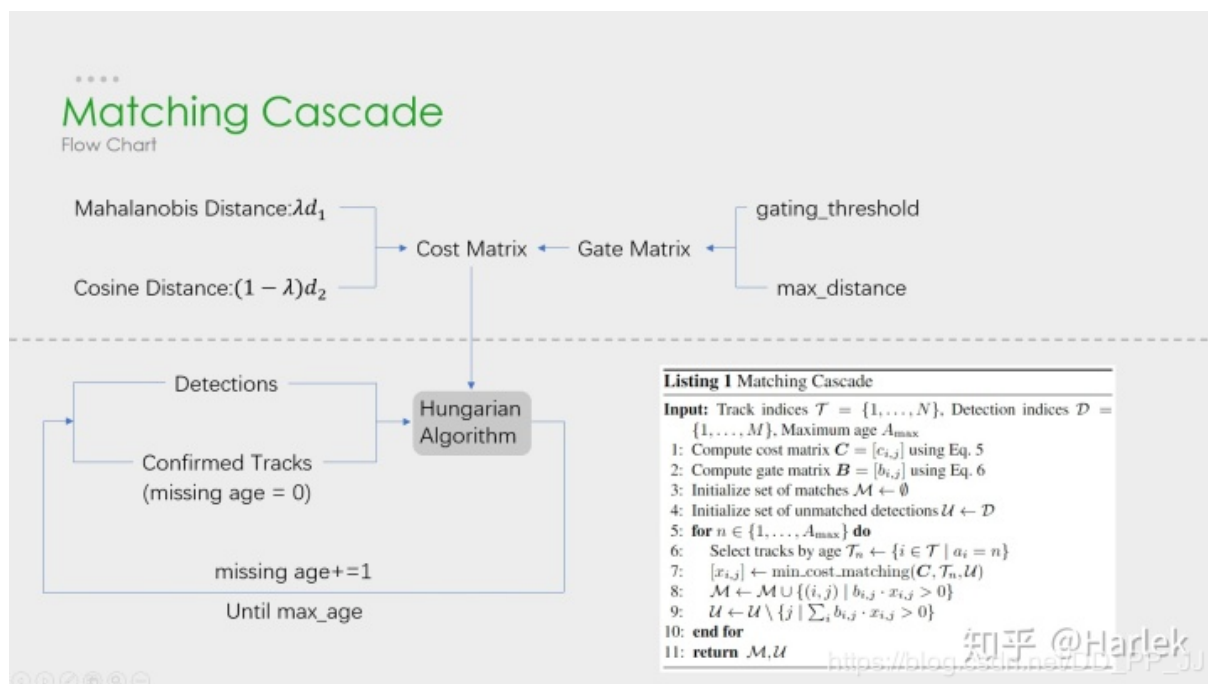


Figure 22: 图片来自知乎 Harlek

- 第二个是 IOU 匹配中的 `iou_matching.iou_cost`:

```
# 虽然和级联匹配中使用的都是 min_cost_matching 作为核心,
# 这里使用的 metric 是 iou cost 和以上不同
matches_b, unmatched_tracks_b, unmatched_detections = \
    linear_assignment.min_cost_matching(
        iou_matching.iou_cost,
        self.max_iou_distance,
        self.tracks,
        detections,
        iou_track_candidates,
        unmatched_detections)
```

`iou_cost` 代价很容易理解,用于计算 Track 和 Detection 之间的 IOU 距离矩阵。

```
def iou_cost(tracks, detections, track_indices=None,
             detection_indices=None):
    # 计算 track 和 detection 之间的 iou 距离矩阵

    if track_indices is None:
        track_indices = np.arange(len(tracks))
    if detection_indices is None:
        detection_indices = np.arange(len(detections))
```

```
cost_matrix = np.zeros((len(track_indices), len(detection_indices)))
for row, track_idx in enumerate(track_indices):
    if tracks[track_idx].time_since_update > 1:
        cost_matrix[row, :] = linear_assignment.INFTY_COST
        continue

    bbox = tracks[track_idx].to_tlwh()
    candidates = np.asarray(
        [detections[i].tlwh for i in detection_indices])
    cost_matrix[row, :] = 1. - iou(bbox, candidates)
return cost_matrix
```

6. 总结

以上就是 Deep SORT 算法代码部分的解析，核心在于类图和流程图，理解 Deep SORT 实现的过程。

如果第一次接触到多目标跟踪算法领域的，可以到知乎上看这篇文章以及其系列，对新手非常友好：

<https://zhuanlan.zhihu.com/p/62827974>

笔者也收集了一些多目标跟踪领域中认可度比较高、常见的库，在这里分享给大家：

- SORT 官方代码: <https://github.com/abewley/sort>
- DeepSORT 官方代码: https://github.com/nwojke/deep_sort
- 奇点大佬 keras 实现 DeepSORT: https://github.com/Qidian213/deep_sort_yolov3
- CenterNet 作检测器的 DeepSORT: <https://github.com/xingyizhou/CenterTrack> 和 <https://github.com/kimyoon-young/centerNet-deep-sort>
- JDE Github 地址: <https://github.com/Zhongdao/Towards-Realtime-MOT>
- FairMOT Github 地址: <https://github.com/ifzhang/FairMOT>
- 笔者修改的代码: https://github.com/pprp/deep_sort_yolov3_pytorch

笔者也是最近一段时间接触目标跟踪领域，数学水平非常有限 (卡尔曼滤波只能肤浅了解大概过程，但是还不会推导)。本文目标就是帮助新入门多目标跟踪的新人快速了解 Deep SORT 流程，由于自身水平有限，也欢迎大佬对文中不足之处进行指点一二。

7. 参考

<https://arxiv.org/abs/1703.07402>

https://github.com/pprp/deep_sort_yolov3_pytorch

<https://www.cnblogs.com/yanwei-li/p/8643446.html>

<https://zhuanlan.zhihu.com/p/97449724>

<https://zhuanlan.zhihu.com/p/80764724>

<https://zhuanlan.zhihu.com/p/90835266>

<https://zhuanlan.zhihu.com/p/113685503>

以上讲解对应的带注释代码在:

https://github.com/pprp/deep_sort_yolov3_pytorch

官方无注释版本代码在:

https://github.com/pprp/SimpleCVReproduction/tree/master/deep_sort-master

DarkLabel 软件配套代码在:

<https://github.com/pprp/SimpleCVReproduction/tree/master/DarkLabel>

感谢阅读