DeepSORT改进

1、概述

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

2. 方法

2.1 状态估计

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。

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。

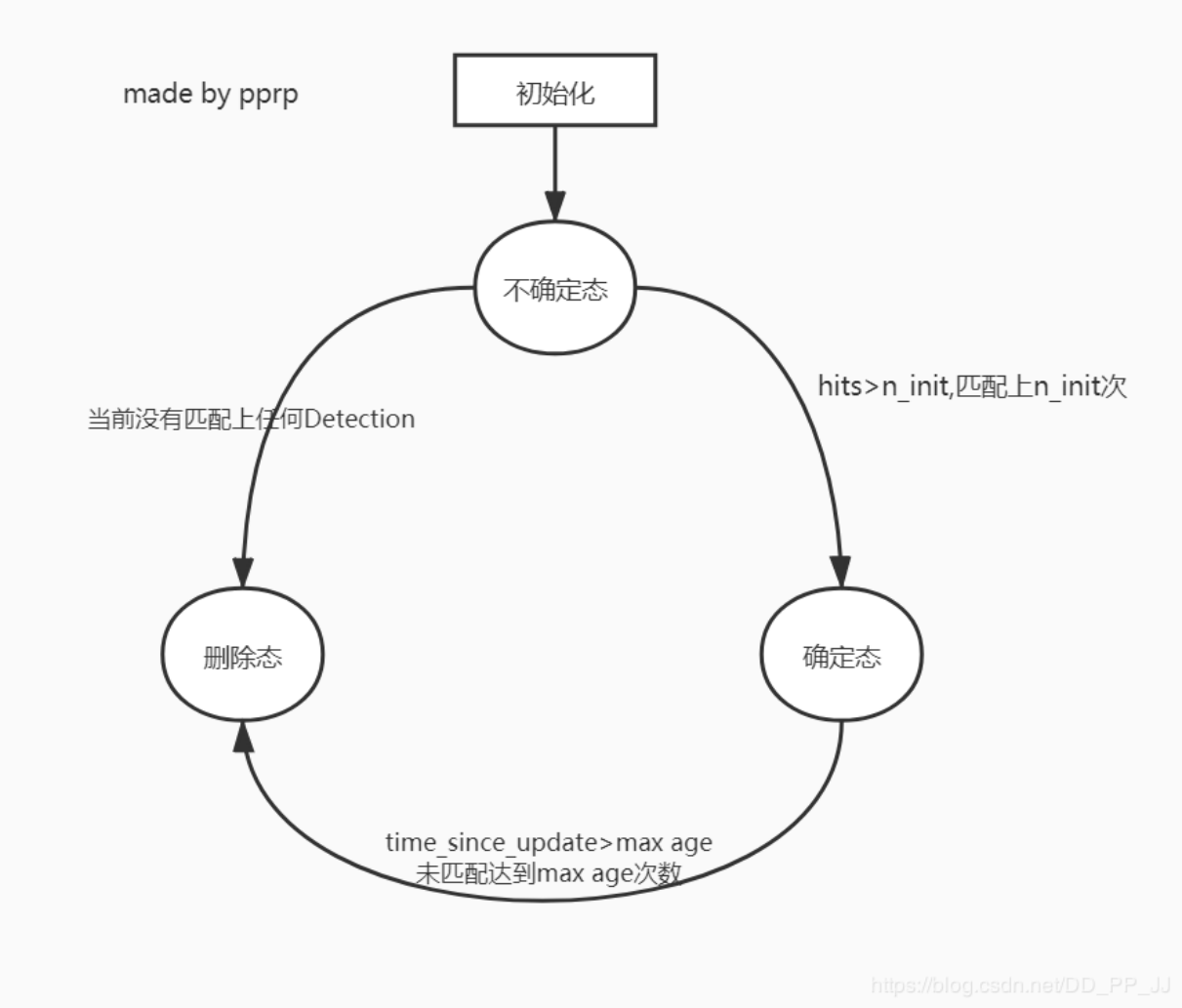

Figure 1: 状态转换

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。

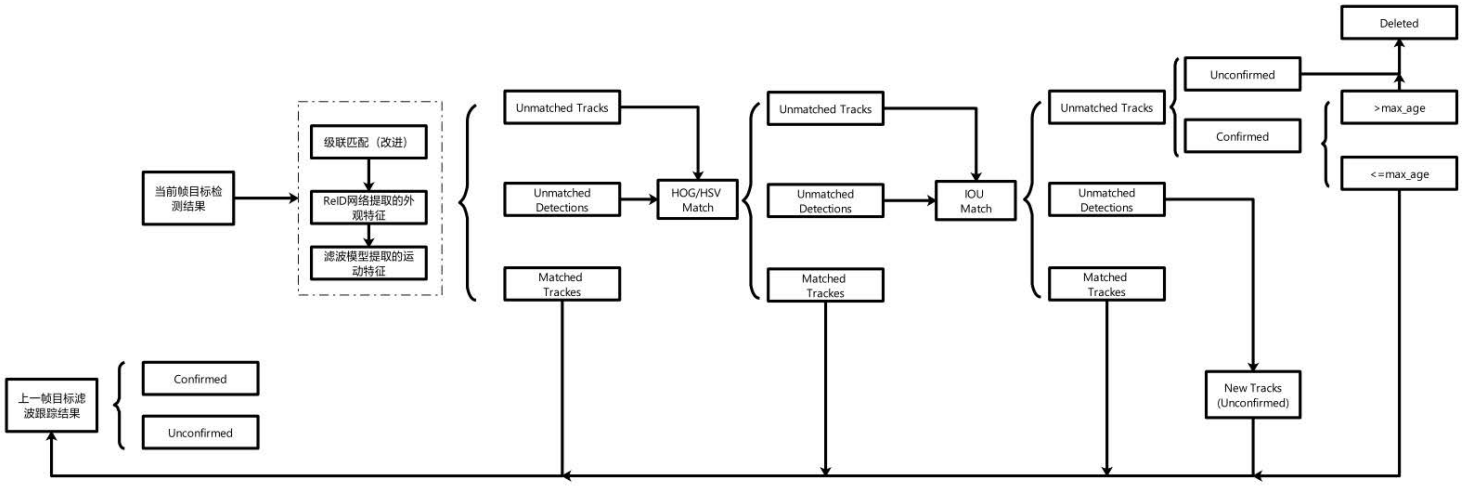

Figure1 : 改进DeepSORT算法

- 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、 H O G 与 H S V 匹 配 以 及 IOU匹配)
- 卡尔曼滤波更新。

其中上图中的级联匹配展开如下：

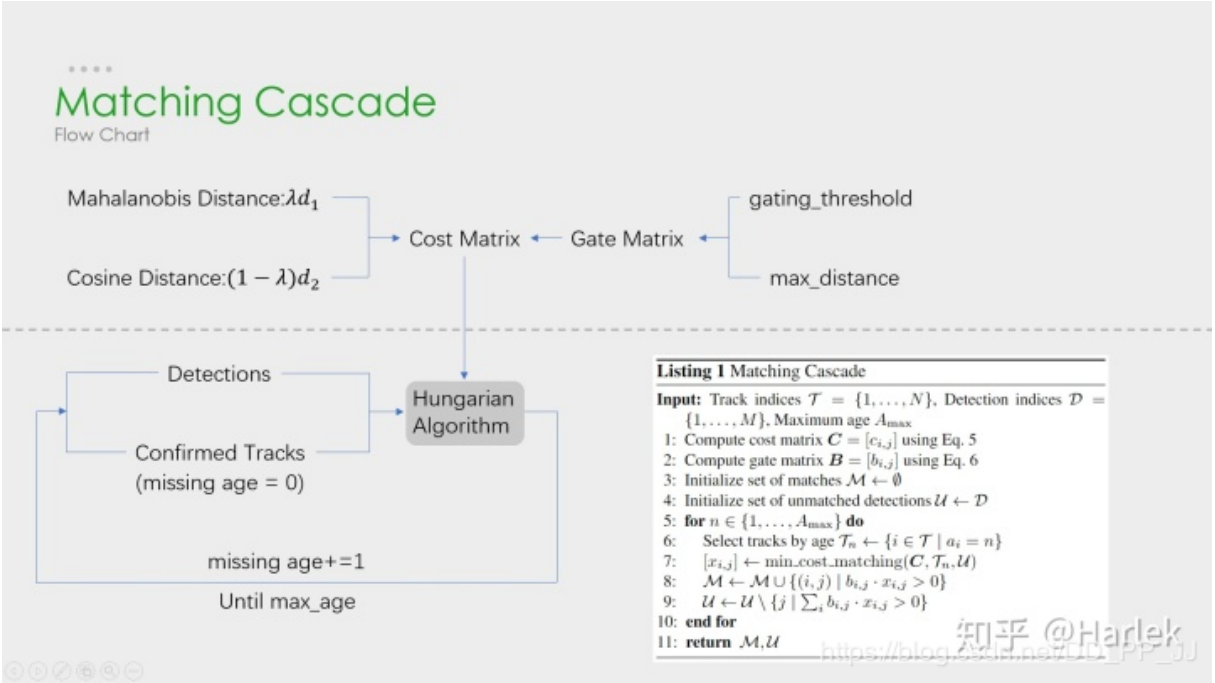

Figure 2: 级联匹配

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. 改进 Deep SORT 代码重新梳理

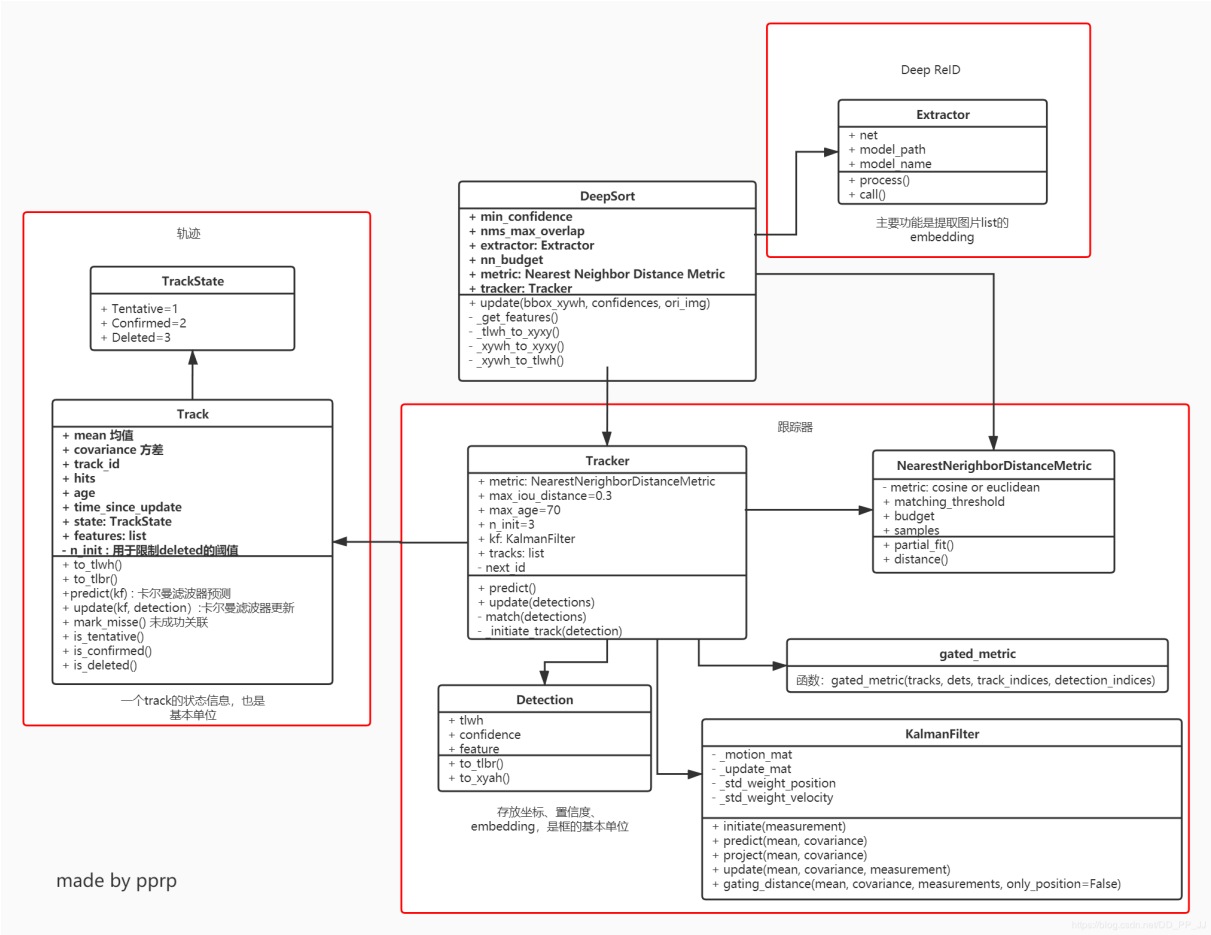

Figure 12: Deep Sort 类图

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

```python
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。

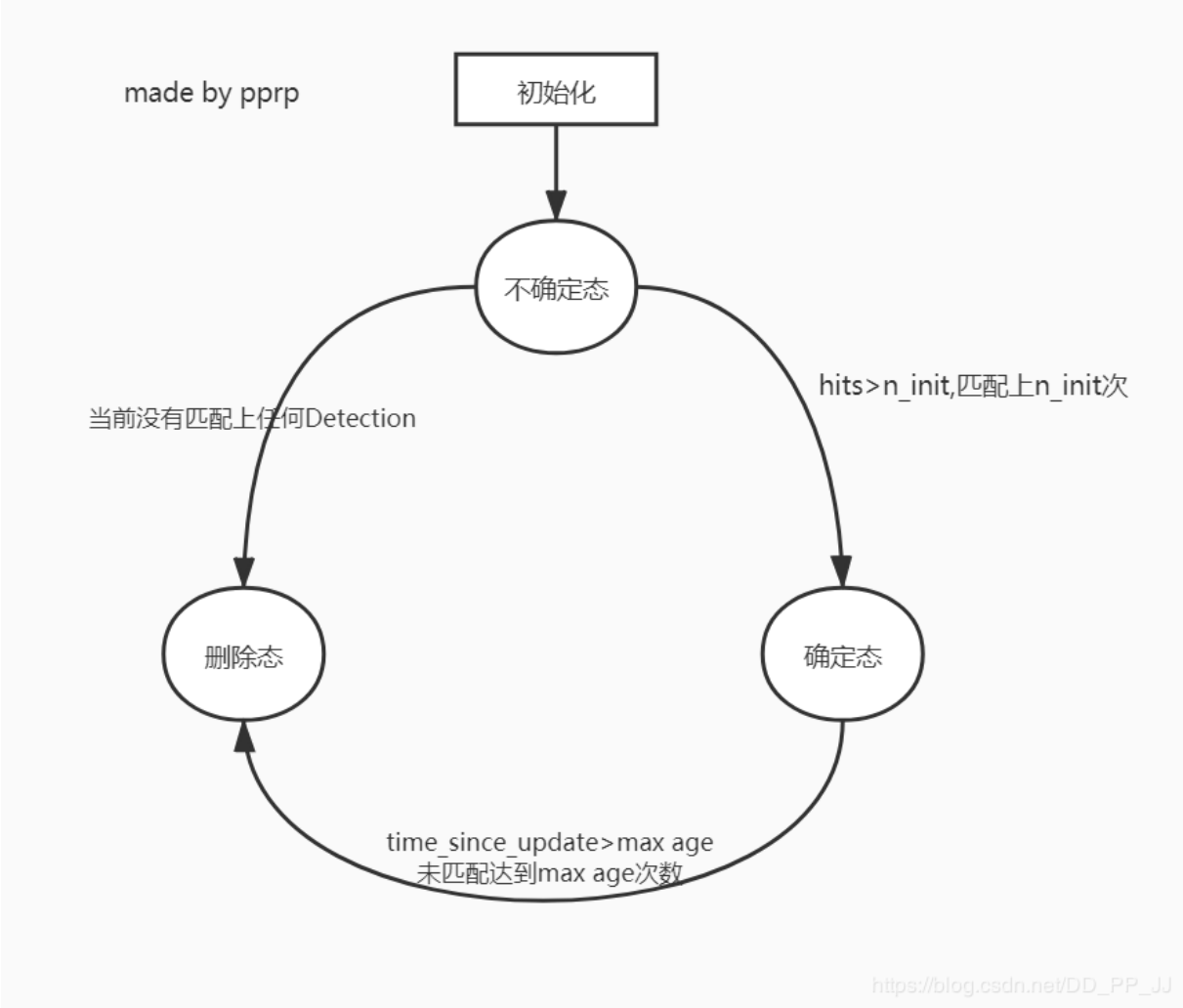

Figure 13: 状态转换图

max_age 代表一个 Track 存活期限，他需要和 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
```

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

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)
```

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

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 budge = 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
        # budge 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。

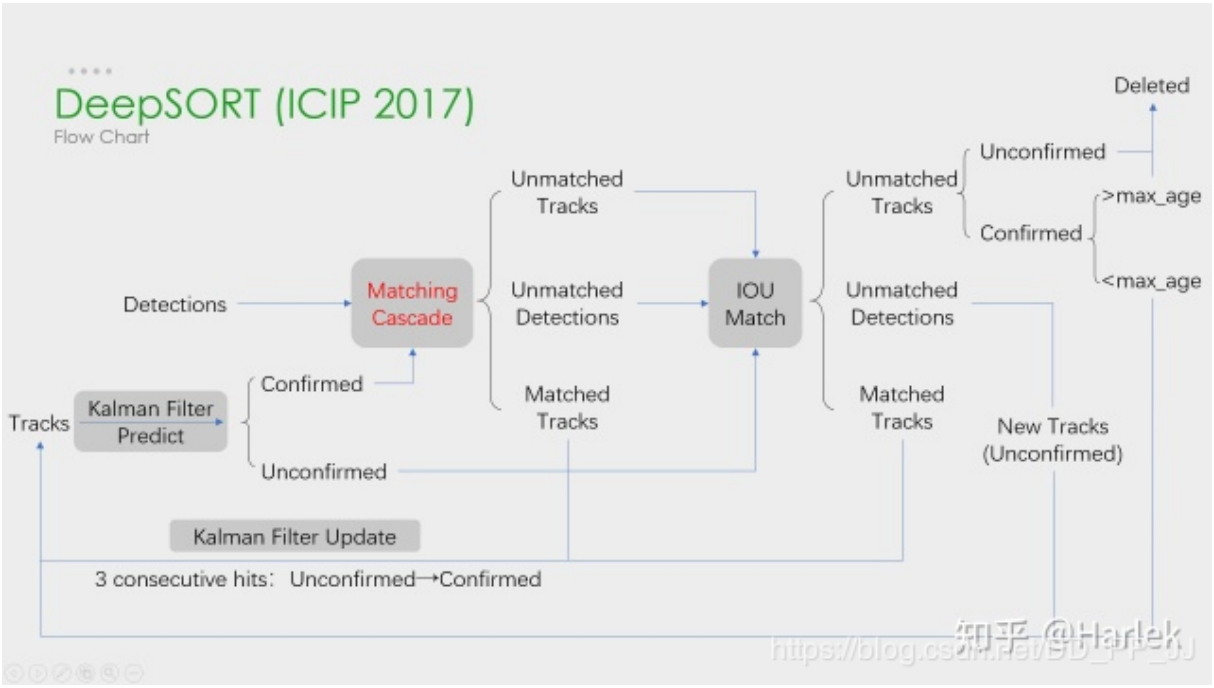

Figure 16: 图片来自知乎 Harlek

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 $\mathcal{T} = \{1, \ldots, N\}$, Detection indices $\mathcal{D} = \{1, \ldots, M\}$, Maximum age $A_{\max}$

Compute cost matrix $\boldsymbol{C} = [c_{i,j}]$ using Eq. 5
Compute gate matrix $\boldsymbol{B} = [b_{i,j}]$ using Eq. 6
Initialize set of matches $\mathcal{M} \leftarrow \emptyset$
Initialize set of unmatched detections $\mathcal{U} \leftarrow \mathcal{D}$
for $n \in \{1, \ldots, A_{\max}\}$ **do**
    Select tracks by age $\mathcal{T}_n \leftarrow \{i \in \mathcal{T} \mid a_i = n\}$
    $[x_{i,j}] \leftarrow \text{min\_cost\_matching}(\boldsymbol{C}, \mathcal{T}_n, \mathcal{U})$
    $\mathcal{M} \leftarrow \mathcal{M} \cup \{(i,j) \mid b_{i,j} \cdot x_{i,j} > 0\}$
    $\mathcal{U} \leftarrow \mathcal{U} \setminus \{j \mid \sum_i b_{i,j} \cdot x_{i,j} > 0\}$
end for
return $\mathcal{M}, \mathcal{U}$

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 8x8 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：

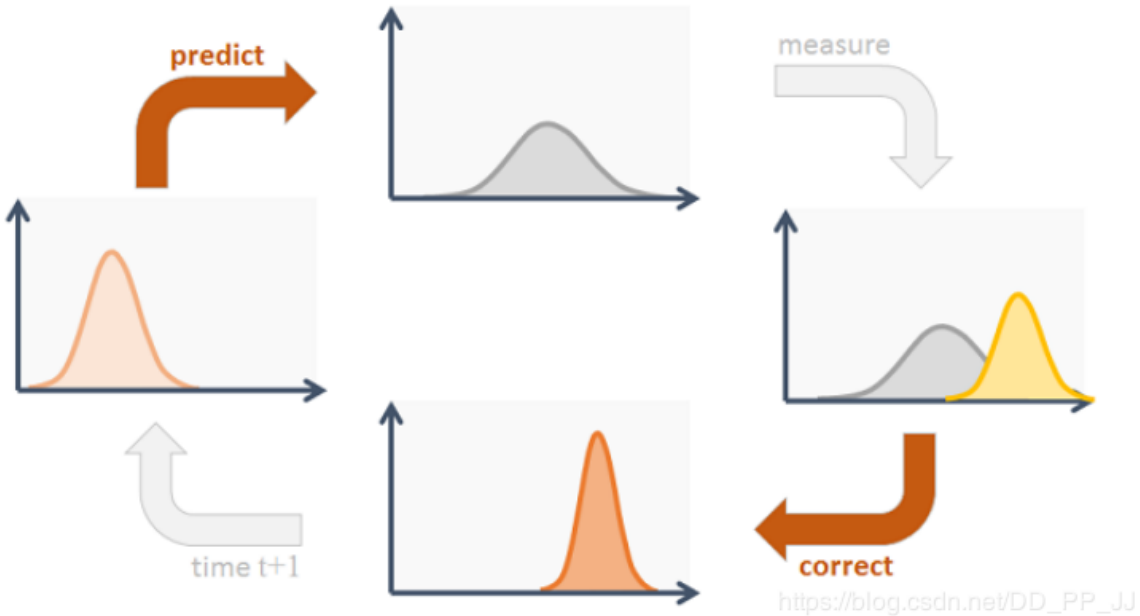

Figure 18: DeepSORT: Deep Learning to Track Custom Objects in a Video

1. 卡尔曼滤波首先根据当前帧 (time=t) 的状态进行预测，得到预测下一帧的状态 (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：

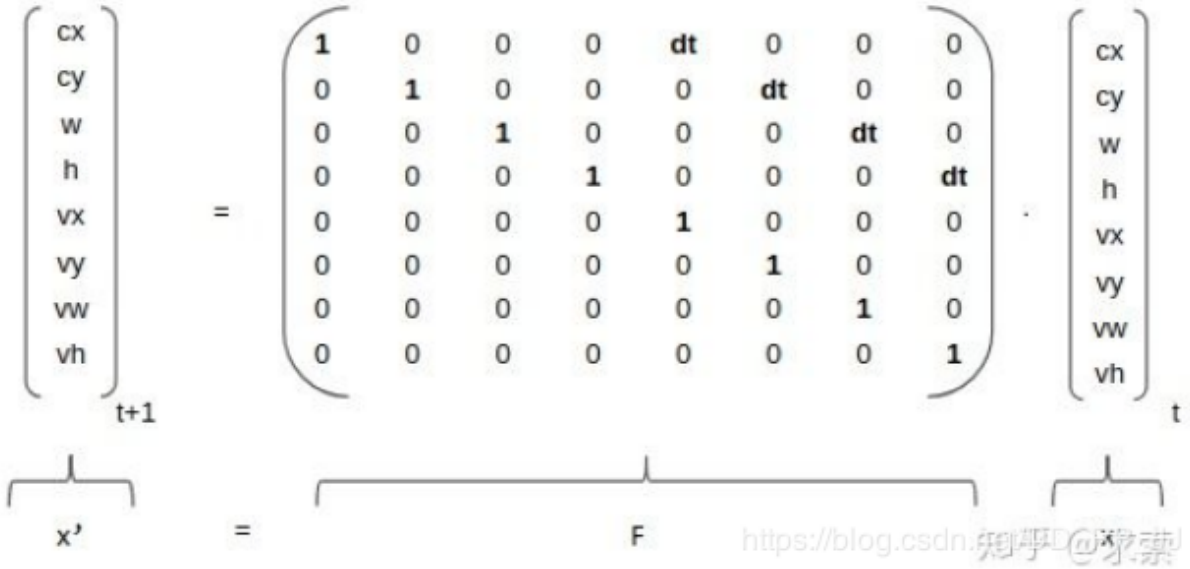

Figure 19:

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

$$y = z - Hx'$$

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

$$K = P'H^T S^{-1}$$

$$x = x' + Ky$$

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

```
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 4x4 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。

$$K = P'H^T S^{-1}$$

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。

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

5. 流程解析

流程部分主要按照以下流程图来走一遍：

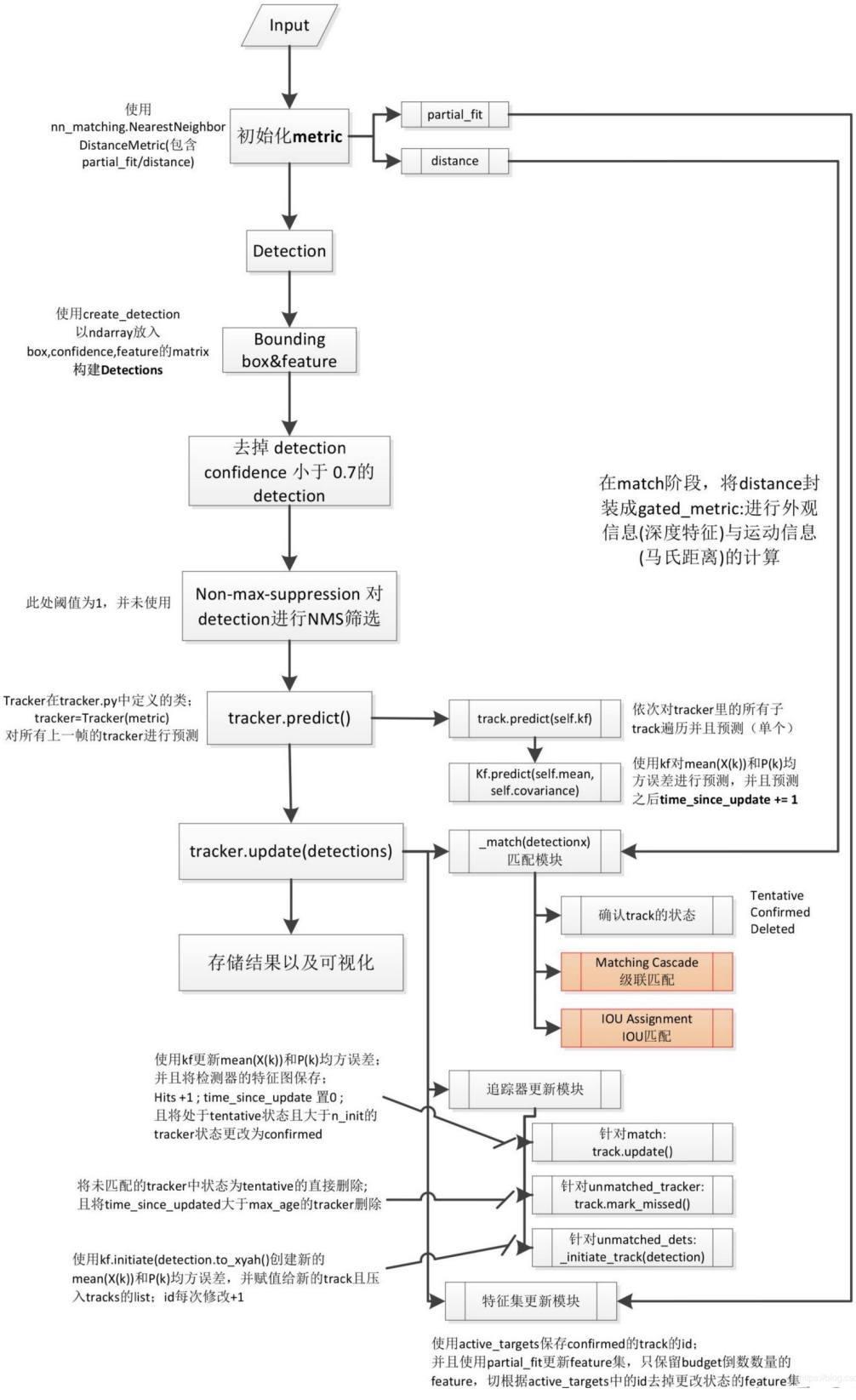

Figure 20: 改进DeepSORT详细流程图

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，NearestNeighborDistanceMetric 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：

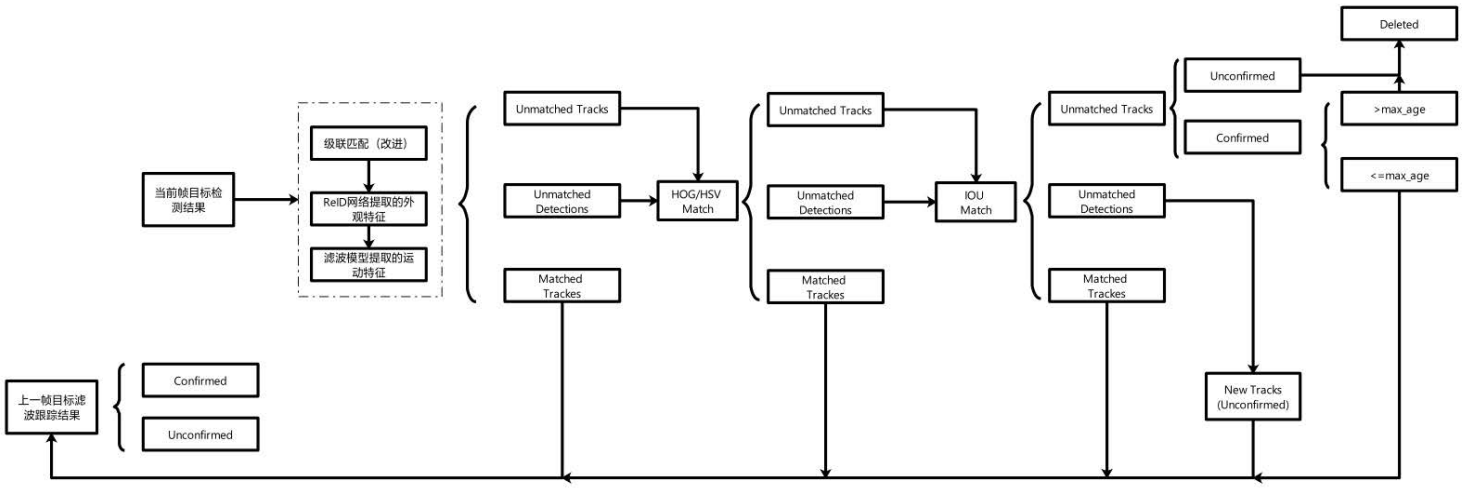

Figure 21: 改进DeepSORT匹配

可以看到，匹配函数的核心是级联匹配 +IOU 匹配，先来看看级联匹配：

调用在这里：

```
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)
```

级联匹配函数展开：

```
def matching_cascade(
        distance_metric, max_distance, cascade_depth, tracks, detections,
        track_indices=None, detection_indices=None):
    # 级联匹配

    # 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
```

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))
```

```python
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)：

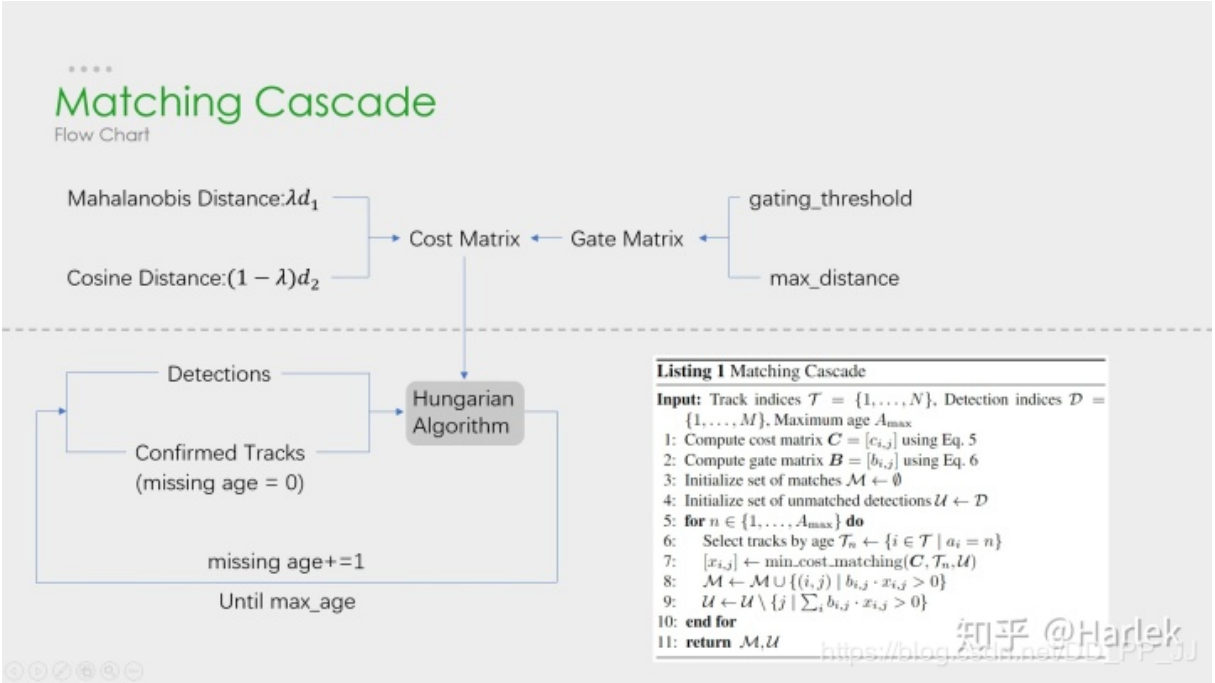

Figure 22: 级联匹配

- 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
```