Abstract

传统上，多对象跟踪和对象检测是使用单独的系统执行的，大多数以前的工作专门针对这些方面中的一个。 跟踪系统显然受益于获得准确的检测结果，但是，文献中有大量证据表明检测器可以从跟踪中受益，例如，可以帮助随着时间推移平滑预测。 在本文中，我们专注于自动驾驶的“检测跟踪”范例，其中两个任务都是关键任务。 我们提出了一种概念上简单有效的联合检测和跟踪模型，称为 RetinaTrack，该模型修改了流行的单阶段 RetinaNet 方法，以便可以进行实例级嵌入训练。 通过对 Waymo Open Dataset 的评估，我们显示出，在所需的计算量显着减少的情况下，我们优于最新的跟踪算法状态。 我们认为，我们简单有效的方法可以为该领域的未来工作提供强大的基准。

1.Introduction

如今，“按检测跟踪”范例已成为多对象跟踪（MOT）的主要方法，其工作原理是独立地检测每个帧的正弦信号，然后跨视频帧执行数据关联。 近年来，由于采用了深度学习，这种方法的两个方面（检测和数据关联）都在重要的技术上取得了进步。

尽管事实上这两个任务经常并存，并且事实很容易使模型适合多任务训练，但直到今天，将这两个方面分开还是比在一个模型中联合训练要普遍得多，大多数论文都将重点放在检测指标或跟踪指标上， 很少两者兼有。 这种任务分离导致了更多复杂的模型和效率较低的方法。这表明该领域的基准测试基准（MOTChallenge [42]）假设模型将使用公开可用的检测，并且论文继续声称使用实时跟踪器，而没有测量执行检测所需的时间。

在本文中，我们主要对自动驾驶领域感兴趣，在该领域中，对象检测和多对象跟踪是关键技术。 如果我们无法检测和跟踪，我们将无法预测车辆和行人的行进路线（以及行进速度），因此，我们将例如不知道是否屈服于行人拐角处，或者即使汽车从对面的车道驶向街道也无法全速行驶。

我们特别关注 RGB 输入，虽然通常不是现代自动驾驶汽车中使用的唯一传感模式，但 RGB 输入起着重要的作用。 RGB 摄像头没有与 LIDAR 相同的范围约束，在检测较小物体方面具有相当大的优势，这对于高速行驶使其对远处的车辆或行人做出反应非常重要的高速公路驾驶尤其重要。

在自动驾驶的设置中，速度和准确性都是必不可少的，因此架构的选择至关重要，因为不能简单地采用最重/性能最高的模型或最轻量级的模型，而不是精确的模型。 我们基于 RetinaNet 检测器[36]建立模型，该检测器是实时的，同时可以达到最先进的精度，并且经过专门设计，可以很好地检测小物体。 在此基础检测器中，我们出于数据关联的目的添加了实例级嵌入。 但是，**原始的 RetinaNet 体系结构不适用于这些实例的嵌入-我们建议对 RetinaNet 的后 FPN 预测子网进行简单但有效的修改以解决这些问题**。我们通过消融显示，我们复制的模型 RetinaTrack 是从跟踪器和检测器的联合训练中受益的。 与基本的 RetinaNet 相比，它的计算开销较小，因此速度很快-由于其简单性，还易于通过 Google TPU 进行训练。

Figure 1: Example vehicle tracking results on the Waymo Open Dataset — tracks are color coded and for clarity we highlight two tracks in each sequence with arrows. Challenges in this dataset include small objects, frequent occlusions due to other traffic or pedestrians, changing scales and low illumination.

总而言之，我们的主要贡献如下：

•我们提出了一个联合训练的检测和跟踪模型-我们的方法简单，有效，可以在自动驾驶汽车中可行地部署。

•我们建议对单发检测架构进行简单的修改，以允许提取实例级别的功能； 我们使用这些功能进行跟踪，但它们也可能用于其他目的。

•我们为 Waymo Open 数据集[2]（图 1）上的 2d 图像建立了检测和跟踪的初始强基线，并表明我们的方法达到了最先进的性能。

我们希望使用简单的模型作为统一的基准，并简化联合检测和跟踪方面的未来研究。

2.1.Object Detectionin Images and Video

近年来，对象检测领域的技术进步呈爆炸式增长，这主要是由社区基准（如 COCO 挑战[37]和开放图像[31]）驱动的。 在检测专用模型体系结构方面也有许多进步，包括基于锚的模型，包括单阶段（例如 SSD [39]，RetinaNet [36]，Yolo 变体[44,45]）和两阶段检测器（例如 ，快速/快速 R-CNN [19,24,47]，R-FCN [13]）以及无新手的模型（例如 CornerNet [32,33]，CenterNet [65]，FCOS [55]）。

在这些单一的框架架构上构建的方法结合了时间上下文，可以更好地在视频中进行检测（特别是与运动模糊，遮挡，物体的摆放等）。 方法包括使用 3d 卷积（例如 I3D，S3D）[8,41,62]或循环网络[29,38]来提取更好的时间特征。 也有许多作品使用某种形式的类似跟踪的概念进行汇总，但它们的主要重点在于检测而不是跟踪。 例如，有些作品利用流量（或类似流量的量）来聚合特征[6，66–68]。 最近，也有一些论文提出了基于对象级别的基于注意力的聚合方法[14,51,59,60]，该方法可以有效地查看沿轨迹的高级别聚合方法。 在许多这些情况下，简单的启发式方法会随着时间的推移“平滑”预测，包括细管平滑[20]或 SeqNMS [22]。

2.2.Tracking

传统上，跟踪器会播放几个不同的条目。 在上述情况下，跟踪器的作用是提高视频中的检测精度（例如，通过随时间推移平滑预测）。 在其他情况下，跟踪器也已经使用了拖累度（传统上要低得多）的检测器，从而可以基于间歇性检测器更新来进行实时更新（例如[3,7]）。

最后，在诸如自动驾驶和运动分析之类的应用中，赛道输出本身就是独立利益。 例如，典型的行为预测模块将对象轨迹作为输入，以便预测特定对象（例如汽车或行人）的未来行为（并对此做出反应）[9,54,56,63,64]。 在这种情况下，按检测跟踪范式已成为多对象跟踪的主要方法，在该方法中，首先在输入序列的每个帧上进行检测，然后将结果跨帧链接（此第二步称为数据关联）。

在前深度学习时代，按检测跟踪的方法[11,21,61]趋向于集中于使用任何可用的视觉功

能，并找到一种方法来应对各种图优化问题的组合爆炸[12,17，46]通过公式确定最佳轨迹。近年来，使用简单匹配算法（例如匈牙利匹配[43]）和集中学习更有利于数据关联的功能（例如，通过深层几何清除）的作者已经扭转了这一趋势[4,34,48,50,52,53,58]。 例如，[58]提出了 Deep Sort，它是一种简单而强大的基线，采用了精细检测（由 Faster RCNN 生产），并使用精细训练的深度 ReID 模型和 Kalman 滤波器运动模型将它们链接起来。 在这种情况下，与深度排序相比，我们的工作可以看作是简化的管道，它依赖于更轻量级的检测网络，该网络与负责执行 ReID 的子网统一。

2.3.Detection meets Tracking

强大的检测能力对于强力跟踪至关重要。 可以通过常用的 CLEAR MOT 度量标准（MOTA，多目标跟踪精度）[42]看到这一点，该度量标准会惩罚误报，误报和身份切换（前两项与检测相关）。 最近的 Tracktor 论文[4]仅使用单帧 Faster R-CNN 检测模型就将这种观察推到了极限，从而获得了很好的结果。 跟踪本身是通过利用 Faster RCNN 第二阶段的行为来完成的，该行为允许将不精确指定的建议（例如，从前一帧检测到）“捕捉”到图像中最接近的对象。通过一些小的修改（包括经过正式培训的 ReID 组件），Tracktor 目前是 MOT17 挑战赛的最新技术，我们在实验中将其与强大的基线进行了比较。

为了解决检测可能对跟踪指标产生巨大影响的问题，MOT Challenge 等基准测试试图通过让多种方法使用完全相同的现成提供的检测来使事情“公平”。 但是，此限制不必要地将双手联系在一起，因为假定两个人分开分开，因此可以排除联合训练的模型，例如我们自己的模型。 有人怀疑联合检测/跟踪文献的匮乏是否部分归因于对黑匣子检测的重视。

在我们工作之前，每次都要尝试训练联合跟踪/检测模型。 Feichtenhofer 等。 [16]运行 R-FCN（[13]）基础检测架构，并在连续帧的高级特征图之间同时计算相关图，然后将其传递到辅助预测塔，以预测帧间实例运动。 像[16]一样，我们共同训练这两个任务。 但是，在自动驾驶需求驱动下，他们只专注于 Imagenet Vid 的检测指标的情况下，我们同时评估跟踪和检测指标。 基于更强大的单级检测器，我们的架构也更加简单，快捷。

还有许多工作可以通过使用 2d 锚定网格直接使用 3d 输入来预测 3d 细管[18、26]，这些锚定网格允许“通过空间偏移的预测时间序列在时间上摇摆”。 但是，这些方法通常较重，并且需要一种机制来相互关联小管，这通常是结合单帧得分和 IOU 重叠的简单启发式方法。我们直接学习关联检测（并证明这很有用）。

最后，**与我们的方法最相关的工作是 Wang 等。 [57]还结合了基于 FPN 的模型（使用 YOLO v3）和附加的嵌入层。 相比之下，我们使用的 RetinaNet 修改具有更强的检测性能，并且我们证明，如果不修改 FPN，性能会受到影响**。

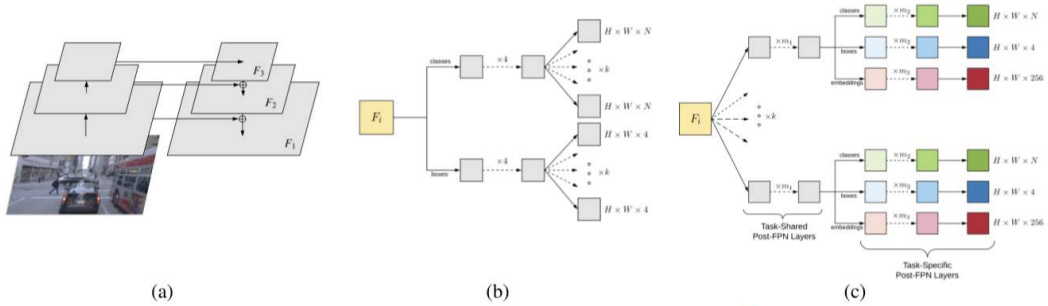
3.The RetinaTrack Architecture

在本节中，我们将介绍 RetinaNet 变体的设计，该设计使我们能够提取每个实例级别的特征。 像其他基于 anchor 的检测器一样，RetinaNet 产生的每个检测都与一个 anchor 相关联。 为了将检测链接到另一帧，我们将希望识别与其对应 anchor 关联的特征向量，并将其传递到嵌入网络，该网络将以度量学习损失进行训练。

3.1.RetinaNet

首先，我们回顾流行的 RetinaNet 架构[36]并解释为什么 vanilla 模型对实例级嵌入是不适合的。现代的卷积目标检测器从沿着图像上方规则网格排列的滑动窗口位置提取特征图。在诸如 RetinaNet 之类的基于 anchor 的方法中，我们在每个网格点的顶部放置 K 个不同形状（纵横比和大小各异）的 anchor 框$\{A_1, \ldots, A_K\}$，并要求模型对于这些 anchors 进行预测

（例如分类对数，方盒回归偏移量）。



(a) (b) (c)

Figure 2: **Architecture diagrams**. (a) Feature Pyramid Network (FPN) and Post-FPN layers of (vanilla) (b) RetinaNet and (c) RetinaTrack . In order to capture instance level features RetinaTrack splits the computational pathways among different anchor shapes at an earlier stage in the Post-FPN subnetwork of RetinaNet. Yellow boxes F_i represent feature maps produced by the FPN. In both models we share convolutional parameters across all FPN layers. At level of a single FPN layer, gray boxes represent convolutional layers that are unshared while colored boxes represent sharing relationships (boxes with the same color share parameters).

在 RetinaNet 的方案中，我们使用基于 FPN 的（特征金字塔网络）特征提取器[35]，它会生成具有不同空间分辨率$W_i \times H_i$的多层特征图F_i（图 2a）。然后将每个特征图F_i传递到两个特定任务的 post-FPN 子网络来预测K个张量（每个都是可能的 anchor 形状）$\{Y_{i,k}^{cls}\}_{k=1:K}$

每个形状$W_i \times H_i \times N$代表N维分类对数，以及形状为$W_i \times H_i \times 4$的 K 张量$\{Y_{i,k}^{loc}\}_{k=1:K}$，表示框回归偏移。（图 2b）。请注意，通常情况下，论文将这些输出折叠为单个组合张量，而不是针对每个 anchor 形状将 K 张量折叠为一个张量，但是，出于我们的目的，为了清楚起见，我们将这些预测分开（最终结果是等效的）。

更正式地说，我们可以将 RetinaNet 分类和位置预测张量写作是每个特征图F_i的函数，如下所示：

$$Y_{i,k}^{cls}(F_i) \equiv \text{Sigmoid}(\text{Conv}(\text{Conv}^{(4)}(F_i;\theta^{cls});\phi_k^{cls})), \quad (1)$$

$$Y_{i,k}^{loc}(F_i) \equiv \text{Conv}(\text{Conv}^{(4)}(F_i;\theta^{loc});\phi_k^{loc}), \quad (2)$$

其中$k \in \{1,\dots K\}$是 K 个 anchor 的索引。 我们使用$Conv^{(4)}$来指 4 个中间3×3卷积层（除非另有说明，否则包括 BN 和 ReLu 层）。FPN 之后的模型参数为θ^{cls}，$\{\phi_k^{cls}\}_{k=1}^{K}$，$\theta^{loc}$和$\{\phi_k^{loc}\}_{k=1}^{K}$。 重要的是，尽管分类和框回归子网络对于给定的 FPN 层具有不同的参数，但这些参数在 FPN 层之间共享，这使我们可以将从不同层提取的特征向量视为属于兼容的嵌入空间。

3.2.Modifying task-prediction subnetworks to have anchor-level features

根据等式 1,2，在所有 K 个 anchors 之间共享 RetinaNet 的所有卷积参数（直到最终进行分类和回归子网的卷积为止）。 因此，没有明确的方法来提取每个实例的特征，因为如果两个检测匹配到具有不同形状的相同位置的 anchor，则网络中唯一可以区分它们的点就是最终分类和框回归预测。 当对象更可能与共享相同位置的 anchor 对应时，通过遮挡进行跟踪时，这可能会特别有问题（图 3）。

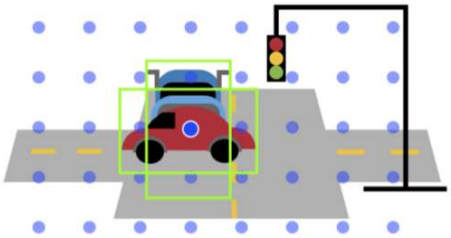


Figure 3: In order to track successfully through occlusions, we need to be able to model that objects that share the same anchor grid center have distinct tracking features. Here, green boxes represent two anchors centered at the same location which match the cars in the scene. Blue dots represent centers of the anchor grid.

我们的解决方案是强制在 post-FPN 预测层之前更早地进行 anchor 之间的拆分，从而使我们能够访问与 anchor 唯一关联的中间层特征（并因此进行最终检测）。 我们提出的修改很简单-我们最终获得了与 RetinaNet 类似的架构，但权重与原始架构相比却有所不同。在我们的 RetinaTrack 模型中，我们通过以下参数化进行预测（参见方程式 1,2）：

$$F_{i,k} = \text{Conv}^{(m_1)}(F_i;\, \theta_k), \tag{3}$$

$$Y^{cls}_{i,k} \equiv \text{Sigmoid}(\text{Conv}(\text{Conv}^{(m_2)}(F_{i,k};\, \theta^{cls});\, \phi^{cls})), \tag{4}$$

$$Y^{loc}_{i,k} \equiv \text{Conv}(\text{Conv}^{(m_2)}(F_{i,k};\, \theta^{loc});\, \phi^{loc}). \tag{5}$$

因此，对于每个 post-FPN 层F_i，我们首先并行应用 K 个卷积序列（具有m_1层）以预测$F_{i,k}$张量，我们将其视为每个 anchor 实例级特征，因为从那时起，将存在与 RetinaNet 架构产生的每一个检测相关联的唯一$F_{i,k}$（图 2c）。我们将模型的第一部分称为任务共享的 post-FPN 层，该层为每一个 K 个 anchor 形状使用单独的参数θ_k，但在 FPN 层之间共享θ_k（分类和定位这两个任务也是如此）。

$F_{i,k}$不是任务特定的特征，而是将两个并行的任务特定的 post-FPN 层序列分别应用于每个$F_{i,k}$之后。 每个序列由m_2 3×3卷积组成，然后是3×3最终卷积，在分类对数（其中 N 是类别数）的情况下具有 N 个输出通道，在盒回归偏移的情况下具有 4 个输出通道。 对于我们的两个任务特定子网，我们在 K 个 anchor 形状以及所有 FPN 层之间共享参数θ^{cls}，φ^{cls}，θ^{loc}和φ^{loc}，因此在任务共享层之后，所有特征都可以视为属于兼容空间。

3.3.Embedding architecture

现在拥有实例级别特征$F_{i,k}$，我们另外应用了由m_3 1×1卷积层组成的任务特定层的第三序列，这些实例层将实例层特征存储在最终轨迹嵌入空间中，每个卷积层映射到 256 个输出通道：

$$Y^{emb}_{i,k} \equiv \text{Conv}^{(m_3)}(F_{i,k};\, \theta^{emb}). \tag{6}$$

除最后的嵌入层外，每次卷积后我们都使用 batchnorm [28]和 ReLU 非线性，并在所有 FPN 层和所有 K anchors 形状上使用相同的共享参数（再次参见图 2c）。

总而言之，RetinaTrack 可以预测每个 anchor 实例级别的特征$F_{i,k}$。 给定检测 d，有一个唯一的 anchor 生成 d，其特征映射$F_{i,k}$是一个与 d 相关的唯一特征向量。 如果 RetinaNet 模型为两个特定于任务的子网络中的每一个运行 4 个卷积层，则在 RetinaTrack 中，每个输出张量都等于$m_1 + m_2 + 1$（在 track embeddings 为$m_1 + m_3$）卷积层，其中m_1，m_2和m_3是结构超参数。 我们将在第 4 节中进一步讨论这些设置的消融。

3.4.Training details

在训练时，我们最小化了两个标准 RetinaNet 损失（用于分类的 Sigmoid Focal Loss，以及用于框回归的 Huber Loss）的未加权总和，以及一个额外的嵌入损失，这鼓励了对应于同

一轨迹的检测具有相似的嵌入。 具体来说，我们使用 BatchHard 策略对 triplets 进行采样[25]，从而训练了 triplet loss[10,49]。

$$\mathcal{L}_{BH}(\theta;\, X) = \sum_{j=1}^{A} \text{SoftPlus}\left(m + \max_{\substack{p=1\ldots A \\ t_j = t_p}} D_{jp} - \min_{\substack{\ell=1\ldots A \\ t_j \neq t_\ell}} D_{j\ell} \right), \tag{7}$$

其中 A 是与 GT 框匹配的 anchor 数量，t_y是分配给 anchor y的轨迹标识，D_{ab}是 anchor a 和 anchor b 的嵌入之间的非平方欧几里得距离，m 是边距（设置为 m = 0.1（在实验中）。因此，通过为每个锚定找到一个硬正样例和一个硬负样例来产生三元组。 实际上，我们采样 64 个三元组进行计算损失。

对于检测损失，我们遵循与[36]中所述类似的目标分配约定。 具体而言，anchor 与 GT 框（IOU）重叠为 0.5 或更高，否则为背景。 此外，对于每个 GT 框，即使该 IOU 小于阈值，我们也会强制最接近的 anchor（相对于 IOU）匹配。 对于 triplet losses，我们遵循类似的约定将轨道标识分配给 anchor，对正匹配使用更严格的 IOU =0.7 或更高的标准-发现此更严格的标准可改善跟踪结果。 仅使用与跟踪身份匹配的 anchor 来生成三元组。 而且，三元组总是从一样的 clip 内产生的。

我们使用 Momentum SGD 在 Google TPU（v3）[30]上进行训练，重量衰减为 0.0004，动量为 0.9。 我们使用 128 个 clips 构建每个批次，为每个 clip 绘制两个帧，每个 clip 间隔 8 帧（Waymo 序列以 10Hz 运行，因此对应于 0.8 秒的时间跨度）。 将批次放置在 32 个 TPU 内核上，将来自同一 clip 的帧并置放置，得出每核 4 帧对的批量大小。 除非另外指定，否则图像将被调整为 1024×1024 分辨率，并且为了在 TPU 内存中适应该分辨率，我们在所有训练过程中都将混合精度训练与 bfloat16 类型一起使用[1]。

我们使用在 COCO 数据集上预训练的 RetinaTrack 模型（删除嵌入投影）初始化模型。接下来（除非另有说明），我们对前 1000 个步骤使用线性学习速率预热进行训练，增加到基本学习速率 0.001，然后对 9K 步骤使用余弦退火学习速率[40]。 根据 RetinaNet，我们使用随机水平偏移和随机作物数据扩充。 我们还允许所有批处理规范层在训练期间独立更新，并且即使相邻卷积层是共享的，也不要强行捆绑它们。

3.5.Inference and Tracking Logic

我们在基于贪婪二部匹配的简单单一假设跟踪系统中使用 embeddings。 在推论时，我们构建了一个跟踪存储，其中存储了有状态的跟踪信息。 对于每个轨道，我们保存以前的检测（包括边界框，类预测和得分），embedding 向量和“轨道状态”，以指示轨道是活着的还是死的（为简单起见，我们不认为轨道曾经处于“暂定”状态） 状态，请参见[58]）。 我们将轨道存储初始化为空，然后对于 cilp 中的每一帧，我们从 RetinaTrack 中选取得分最高的 100 个检测对应的 embedding 向量。

通过分数阈值过滤这些检测，然后我们通过一些指定的相似度函数 S 将剩余的嵌入向量与轨道存储中的向量进行比较，并运行贪婪的二分匹配（不允许在余弦距离超过阈值$1-\epsilon$的情况下进行匹配）。 基于此贪婪匹配，然后将检测添加到轨道存储中的现有轨道，或使用它来初始化新轨道。 在我们的实验中，我们的相似度函数 S 始终是 IOU 重叠的统一加权和（使用 0.4 的截断阈值）和 embeddings 之间的余弦距离。

对于轨道存储中的每个存活轨道，我们最多可以保存其最新的（检测，嵌入矢量，状态）三元组的 H 个，从而允许新的检测与所有轨道的所有 H 个最新观测值相匹配。 轨道可以保持多达 40 帧的存活状态，以便重新识别。 相反，如果未在 40 帧以上重新识别，则将轨道标记为“已死”。

Architecture	Share task weights	m_1	m_2	K	mAP	Inference time (ms per frame)
RetinaNet	No	-	-	6	36.17	45
RetinaNet	Yes	-	-	6	35.35	40
RetinaNet	No	-	-	1	31.45	37
RetinaNet	Yes	-	-	1	30.71	30
RetinaTrack	-	1	3	6	35.11	83
RetinaTrack	-	2	2	6	35.55	75
RetinaTrack	-	3	1	6	35.74	74

Figure 4: **COCO17 ablations.** Performance of vanilla RetinaNet and RetinaTrack (without tracking embedding layers) in terms of single image object detection performance on COCO17. m_1 denotes the number of task-shared post-FPN layers and m_2 denotes the number of task-specific post-FPN layers.

4.Experiments

在我们的实验中，我们专注于最近发布的 Waymo Open 数据集[2] v1（简称为 Waymo）。我们还将在 4.4 节中报告较大的 v1.1 版本的结果。 该数据集在 Waymo 车辆中以 10 Hz 的频率收集了 200K 帧的注释，并涵盖了各种地理和天气条件。 帧来自 5 个摄像头位置（正面和侧面）。 出于本文的目的，我们将重点放在二维检测和跟踪上，并且更具体地说，仅针对“车辆”类，因为数据集的主要类不平衡，这不是我们的主要重点。 除了 Waymo，我们还报告了 COCO17 数据集的消融[37]。

最后，我们评估了通过标准均值 AP [15,18,37]（mAP）以及 CLEAR MOT 跟踪指标[5,42]所测量的检测和跟踪指标，特别是使用 COCO AP（IOU 阈值的平均值在 0.5 和 0.95 之间）和 py- motmetrics 库。1 我们还使用 Nvidia V100 GPU 进行基准测试，以每帧毫秒为单位报告推理时间。 对于所有模型，我们仅以“深度学习部分”为基准，而忽略了跟踪器所需的任何簿记逻辑，该逻辑通常非常轻便。

同时评估模型以进行检测和跟踪需要一定注意。 检测 mAP 衡量的是模型在平均精度和召回率之间进行权衡而无需硬操作点的平均能力-因此，最好使用低或零分数阈值进行检测 mAP。 但是，诸如 MOTA 之类的 CLEAR MOT 跟踪指标需要选择一个工作点，因为它们直接引用了正确/错误的正负，并且在实践中对这些超参数选择相当敏感。 通常最好使用较高的分数阈值来报告跟踪指标，以免出现假阳性。 在我们的实验中，我们可以简单地使用单独的阈值进行评估: 我们使用接近零的分数阈值并使用较高的分数阈值作为跟踪器来评估模型。

4.1.Evaluating Retina Track as a detector

作为初步消融（表 4），我们通过对 COCO17 进行评估，研究了视网膜组织的视网膜结构改变对标准单幅图像检测的影响。 在这些实验中，我们删除了 RetinaTrack 的嵌入层，因为 COCO 不是视频数据集。

仅对于这些实验，我们训练的设置与后来的 Waymo 实验相比略有不同。 我们使用 Resnet-50 作为基本特征提取器（初始化了 Imagenet），并以 bfloat16 [1]的混合精度以 896×896 的分辨率进行训练。 我们训练了 64 个大小为 8 个 TPU v3 内核的批次，并执行了每个内核的批次归一化。 我们将前 2K 步使用线性学习率预热，增加到基本学习率 0.004，然后将余弦退火学习率[40]用于 23K 步。 请注意，我们可以使用较重的特征提取器或使用更高的图像分辨率来提高性能，但主要目的是为了消除 RetinaNet 和 RetinaTrack 的 Post-FPN 子网的变化。

回想一下，m1 和 m2 分别是指任务共享和任务特定的后 FPN 子网的卷积数。 我们将 m1 + m2 = 4 设置为与 RetinaNet 相当。 K 是每个位置的锚形状的数量，默认情况下，我们将其设置为 6，但为了表明每个位置具有多个锚形状对于检测很重要，我们还与简化的 RetinaNet（每个位置仅使用 1 个框）进行了比较。 最后，我们尝试使用原始 RetinaNet 版本，在该版本中，特定于任务的子网必须共享其权重（表 4 中的“共享任务权重”列），因为

这更接近于 RetinaTrack 的任务共享的 FPN 后层。

我们首先注意到，每个位置使用 K = 6 个 anchor 对于提高 COCO 的性能非常重要，并且拥有一个单独的任务特定子网比共享它更好，这证实了[36]的观点。 我们还观察到，通过使用 RetinaTrack，我们能够按设计提取每个实例的特征（我们将在接下来的跟踪中使用它，但通常会很有用），同时在 COCO 上实现类似的检测性能。 如果不需要每个实例级别的特征，则仍可以使用 RetinaNet 的原始预测头布局（与 SSD [39]和许多论文所使用的 RPN 相似，例如[23,47]）获得更好的数字。 在 RetinaTrack 的（m1，m2）的 3 个设置中，我们发现在单个任务指定层（m2 = 1）之后使用 3 个任务共享层（m1 = 3），与其他配置相比略有优势。

我们在表 4 中报告了运行时间（平均超过 500 张 COCO 图像）。我们的修改增加了原始 RetinaNet 上的运行时间-这并不奇怪，因为 FPN 后子网的成本现在已乘以 K。在 RetinaTrack 的三种变体中， （m1 = 3，m2 = 1）还是最快的。

4.2.Architectural ablations

对于其余的实验，我们在 Waymo 上进行评估，这次包括具有三重态损失训练的嵌入网络，并使用第 3.5 节中描述的系统另外评估跟踪性能。

# embedding layers	MOTA	mAP
0	38.52	37.93
2	39.19	38.24
4	38.85	38.24

Figure 6: Track embedding subnetwork depth ablation. We train versions of RetinaTrack with $m_3 = 0, 2$, and 4 projection layers.

我们首先消融嵌入网络的深度（请参见表 6），在该网络中我们使用 m3 = 0、2 和 4 个投影层（在上面的 COCO 消融中显示 m1 = 3 和 m2 = 1 来训练模型）训练模型，从而获得最佳性能 具有 2 层检测和跟踪功能。

为嵌入子网设置 m3 = 2 层，我们通过表 3.4 中介绍的方法在表 5 的 Waymo 数据集上展示消融。

为了演示 RetinaTrack 锚点级特征在跟踪中的价值，我们评估了两种原始版本的 RetinaNet 架构-（1）使用 K = 1 锚点形状的一种，因为在这种情况下可以提取每个实例的特征向量 ，以及（2）标准 K = 6 设置，其中在跟踪过程中，我们只需将“碰撞”在同一空间中心的锚的嵌入强制为相同（我们将该基线称为 RetinaNet *）。

与 COCO 消融一样，我们看到使用多个（K = 6）锚形状对于检测和跟踪指标都很重要。因此，毫无疑问，RetinaTrack 明显优于基于 RetinaNet 的（K = 1）跟踪基线，主要是因为它是一个更强大的检测器。 但是，与未加星标的对应部分相比，RetinaNet *的行均显示出较低的 MOTA 和 mAP 结果，这表明“滥用”vanilla RetinaNet 忽略碰撞的锚点进行跟踪对于检测和跟踪都是有害的，因此强调了 RetinaTrack 的按锚嵌入的重要性。

我们最好的 RetinaTrack 配置达到 39.12 MOTA，mAP 为 38.24。 与 vanilla RetinaNet 在 RetinaTrack 上保持轻微边缘的 COCO 消融相反，在这里我们看到 RetinaTrack 的性能优于 RetinaNet 作为检测器，这表明通过包含跟踪损耗，可提高检测性能。

Architecture	Share task weights	m_1	m_2	m_3	K	MOTA	mAP	Inference time (ms per frame)
RetinaNet	No	-	-	-	6	-	38.19	34
RetinaNet*	No	-	-	-	6	38.02	37.43	44
RetinaNet	Yes	-	-	-	6	-	37.95	30
RetinaNet*	Yes	-	-	-	6	37.63	36.75	40
RetinaNet	No	-	-	2	1	30.94	35.20	33
RetinaNet	Yes	-	-	2	1	31.20	35.08	29
RetinaTrack	-	1	3	2	6	38.71	37.96	88
RetinaTrack	-	2	2	2	6	39.08	38.14	81
RetinaTrack	-	3	1	2	6	39.12	38.24	70

Figure 5: **Waymo ablations.** Performance of vanilla RetinaNet and RetinaTrack (including tracking embedding layers) in terms of detection mAP and tracking MOTA on the Waymo Open Dataset. m_1 denotes the number of task-shared post-FPN layers, m_2 denotes the number of task-specific post-FPN layers, and m_3 denotes the number of embedding layers. RetinaNet* is a vanilla RetinaNet model (with $K = 6$) trained with tracking losses where instance embedding vectors are shared among "colliding anchors".

最终，每帧运行时间为 70ms，我们注意到 RetinaTrack 的推理速度比 Waymo 数据集中的传感器帧速率（10 Hz）快。 与 COCO 设置相比，RetinaTrack 必须运行附加的卷积层以进行嵌入，但是由于 COCO 具有 80 个类，这使网络顶部稍重，因此 Waymo 设置中的最终运行时间略短。

4.3.Joint vs Independent training

为了证明联合训练具有检测和跟踪任务的好处，我们现在将 RetinaTrack 与使用与 RetinaTrack 相同的跟踪系统的三个自然基线进行比较，但会更改基础数据关联相似性函数（表 8）：

Model	MOTA	mAP	Inference time (ms)
IOU baseline	35.36	38.53	70
RetinaTrack w/o triplet loss	37.92	38.58	70
RetinaTrack , w/R-50 ReID	37.39	38.58	80
RetinaTrack	39.19	38.24	70

Figure 8: Comparison of joint training (RetinaTrack) with alternatives: (1) IOU based similarity tracker, (2) RetinaTrack w/o triplet loss, (3) RetinaTrack w/R-50 ReID,

•**An IOU baseline**，其中检测相似性仅通过 IOU 重叠（无嵌入）来衡量，

• **RetinaTrackw/o triplet loss**，其中我们忽略了三重损失（因此不训练模型专门用于跟踪），并通过每个实例特征向量 Fi，k 和

•**RetinaTrackw / R-50 ReID**，其中我们在训练 RetinaTrack 时再次忽略了三重态损失，并将检测结果提供给精细训练的重新识别（ReID）模型。 对于 ReID 模型，我们训练了基于 Resnet-50 的 TriNet 模型[25]在 Waymo 上执行 ReID。

我们观察到，即使是仅使用 IOU 的跟踪器也可以在 Waymo 上提供相当强的基线，这很可能是由于其具有强大的检测模型-当汽车缓慢行驶时（例如，高速公路驾驶），该跟踪器可能更准确 。 但是，使用视觉嵌入可以使我们在所有情况下都优于该简单基准，并且在接受检测和度量学习损失训练的 RetinaTrack 上，它们的综合性能均优于这些基准。

4.4.Comparison against state of the art

Model	MOTA	TP	FP	ID switches	mAP	Inference time (ms per frame)
Tracktor	35.30	106006	15617	16652	36.17	45
Tracktor++	37.94	112801	15642	10370	36.17	2645
RetinaTrack	39.19	112025	11669	5712	38.24	70

Figure 7: We compare RetinaTrack to Tracktor/Tracktor++ [4] which are currently state of the art on the MOT17 Challenge.

最后，我们将（表 7）与最新的 Tracktor 和 Tracktor ++算法进行比较，这些算法目前是 MOT Challenge 的最新技术。 对于这些实验，我们使用 Tracktor 和 Tracktor ++自己的 Tensorflow 重新实现，其中添加了 ReID 组件和相机运动补偿（CMC）。 我们的实现与原始

论文中描述的细节有所不同，因为它基于 Tensor 流对象检测 API [27]，并且不使用 FPN。我们使用与第 4.3 节中相同的 ReID 模型，该模型与 Tracktor 论文中采用的方法相匹配。 为了验证我们的重新实现具有竞争力，我们将基于 Resnet101 的 Tracktor 模型的结果提交给了官方 MOT Challenge 服务器，该服务器与使用 FPN 的官方提交获得了几乎相同的 MOTA 编号（53.4 与 53.5）。 我们还从基于 Resnet-152 的跟踪器提交了结果，该跟踪器目前在公共排行榜上的性能超过了排行榜（MOTA 为 56.7）。

在 Waymo 上，我们使用以 1024×1024 分辨率运行的基于 Resnet-50 的 Tracktor，以与我们的模型相当。 如果将表 8 中的 Tracktor（没有 CMC or ReID）的 MOTA 得分与 IOU 跟踪性能进行比较，我们会发现这两种方法大致相当。 我们相信基于 IOU 的跟踪可以在这里与 Tracktor 达到同等水平，这是因为（1）首先要进行高度准确的检测，以及（2）严重的摄像机运动会伤害 Tracktor。

Model	MOTA	mAP	Inference time (ms)
IOU baseline	38.25	45.78	70
Tracktor++	42.62	42.41	2645
RetinaTrack	44.92	45.70	70

Figure 9: Evaluations on the Waymo v1.1 dataset (which has a 4× larger training set than the v1 dataset).

实际上，我们发现 Tracktor 需要使用“ ++”才能大大胜过基于 IOU 的跟踪器。 但是，它要慢得多-除了运行 Faster R-CNN 之外，它还必须为 ReID 运行第二个 Resnet-50 模型，然后再运行 CMC（这非常耗时）。

RetinaTrack 在跟踪和检测方面均优于两种变体。通过显着减少误报和 ID 切换的数量，它可以实现这些改进。尽管 RetinaTrack 的速度比 Vanilla Tracktor 慢(其运行时间由 FasterR-CNN 支配)，但其速度却比 Tracktor ++快得多。

Evaluation on the Waymo v1.1 dataset. 作为将来进行比较的基准，我们还重现了对 Waymo v1.1 版本的评估，该版本带有约 800K 框架，用于训练约 1.7M 带注释的车辆。 对于这些评估，我们以 0.004 的基本学习率训练了 100K 步（并固定了所有其他超参数）。 结果显示在表 9 中，在表 9 中，我们再次看到相同的趋势，RetinaTrack 的性能明显优于基于基线的 IOU 轨迹，而 Tracker ++的性能显着提高，运行时间明显加快。

5.Conclusion

在本文中，我们提出了一个简单但有效的模型 RetinaTrack，该模型可以联合进行检测和跟踪任务的训练，并扩展单级检测器以处理实例级属性，这表明我们可能会对跟踪之外的应用程序具有独立的兴趣。

此外，我们已经证明了联合训练在训练独立检测和跟踪模型的主流方法上的有效性。这种方法可以使 RetinaTrack 在多目标跟踪方面超越当前技术水平，同时显着提高速度并能够跟踪长时间消失的对象。 最后，希望我们的工作可以为将来的检测和跟踪研究提供强大的基准。