Object Tracking for Autonomous Driving Systems

Aman Dhar

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2020-81
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-81.html

May 28, 2020

Object Tracking for Autonomous Driving Systems

by Aman Dhar

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Joseph E. Gonzalez
Research Advisor

May 28, 2020
(Date)

Text

* * * * * * *

Professor Ion Stoica
Second Reader

May 28, 2020
(Date)

Object Tracking for Autonomous Driving Systems

Abstract

Object Tracking for Autonomous Driving Systems

by

Aman Dhar

Master of Science in Electrical Engineering and Computer Science

University of California, Berkeley

,

Autonomous driving systems have benefited from recent progress in computer vision and deep learning. In particular, recent object tracking algorithms have used deep learning in various ways to improve tracking performance. However, several challenges related to autonomous driving systems and object tracking remain, and popular evaluation criteria do not always incentivize the development of object trackers that perform well in autonomous driving systems.

This work first identifies several challenges related to object tracking in autonomous driving systems. Next, three different types of object trackers are evaluated in a series of experiments involving an autonomous vehicle in a simulated environment. As part of this evaluation, a framework is described and provided for others to be able to reproduce the results of the experiments and integrate, deploy, and evaluate alternative object trackers. An analysis of the results demonstrates that no single type of object tracker performs best in all driving scenarios. Finally, future research directions regarding object tracking and autonomous driving systems are proposed.

Contents

List of Figures

List of Tables

Acknowledgments

I first thank my advisor, Professor Joseph E. Gonzalez, who provided me with several opportunities, resources, and insights related to this work. In addition, I thank Ionel Gog and Sukrit Kalra for their leadership and assistance in guiding my efforts. Next, I thank the rest of the ERDOS team and all who helped me when I was stuck, including: Peter Schafhalter, Professor Ion Stoica, Edward Fang, Alvin Kao, Mong H. Ng, Rowan McAllister, and Matthew Wright. And finally, I thank my family.

Chapter 1

Introduction

Autonomous driving has gained much interest due to rapid development regarding deep learning, systems, and robotics research. Today, modern vehicles are often already equipped to perform a variety of tasks autonomously, including lane keeping, adaptive cruise control, and self-parking. However, full autonomy has not yet been achieved in consumer vehicles. In order to attain fully autonomous vehicles that drive more safely than human drivers, several components of modern autonomous driving systems must be improved. This thesis focuses specifically on the object tracking component of autonomous driving systems. Beyond the discussion of recent and future directions of relevant research, it includes the evaluation of three object tracking algorithms in a variety of simulated driving scenarios.

The remainder of this thesis is organized into several chapters. Chapter 2 provides background information and common vocabulary regarding both autonomous driving systems and object tracking. Chapter 3 examines related work in depth and introduces additional information concerning modern approaches to the development and evaluation of object tracking algorithms. Chapter 4 outlines perception-related and tracking-related challenges in autonomous driving systems. Chapter 5 describes tracking-related components of Pylot [15], an autonomous vehicle platform, along with CARLA [13], an open-source simulator for autonomous driving research. Chapter 6 details three object tracking algorithms currently implemented in Pylot. Chapter 7 analyzes several experiments involving the three algorithms under different conditions in a variety of driving scenarios. Chapter 8 discusses future directions for research pertaining to object tracking for autonomous driving systems.

Chapter 2

Background

2.1 Autonomous Driving Systems as Dataflow Graphs

Autonomous driving systems are generally composed of five modules: sensing, perception, prediction, planning, and control. The sensing module of a modern autonomous driving system generally collects vast amounts of data from a variety of sources. For example, this data may include images recorded by cameras mounted on the vehicle, LiDAR point clouds, IMU measurements, GPS data, etc. The data is processed by components in the perception module, which generally perform tasks like object detection, tracking, and localization. For example, raw camera frames are sent to an object detection model to detect objects like cars, pedestrians, and traffic signs and lights. Selected outputs from the perception module are accumulated and fed as input to the prediction module, which predicts trajectories for detected moving objects. These trajectories inform the planning module, which determines routes or waypoints for the vehicle to follow while avoiding collisions. Finally, the planned route is translated by the control module into control inputs for the car, which determine values like the acceleration or steering angle the car must execute to successfully follow the path.

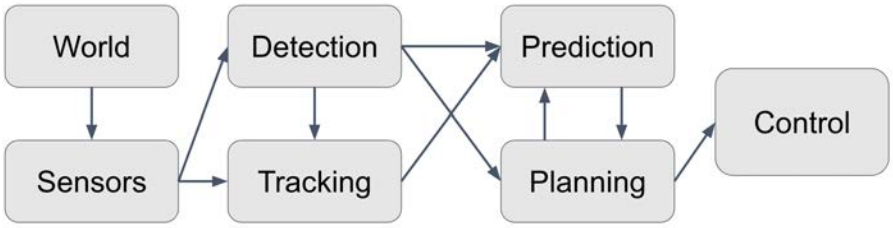


Figure 2.1: An example of how different components of an autonomous driving system can be represented as a dataflow graph. Components like Detection and Tracking can be grouped to form a Perception module.

Common Vocabulary

- ADS(s): Acronym for Autonomous Driving System(s).
- Ego Vehicle: The autonomous vehicle controlled by the system.

2.2 Multiple Object Tracking

Multiple object tracking (MOT) is a computer vision task that aims to record the location of moving objects, commonly referred to as "targets," over time. Beyond autonomous driving, MOT is used in several applications including surveillance and security software, autonomous drone navigation, and medical imaging. Multiple object trackers generally receive camera frames as input in order to update their estimates of the target's location, though other data sources such as LiDAR data may be used as well. Typically, multiple object trackers are designed to track several objects simultaneously; however, a naive multiple object tracker may be constructed by instantiating a single object tracker for each object detected. Both methods are explored in later sections of this work.

Modern multiple object trackers follow the tracking-by-detection paradigm, as they use detection information such as bounding boxes and class labels from an object detection model to begin tracking objects. This is contrasted with detection-free tracking, in which the tracker is manually initialized to follow a fixed number of objects over time. For an ADS, the number and locations of visible objects are rarely known a priori, and objects are expected to enter and exit the view of the camera over time. These factors make tracking-by-detection algorithms much more suitable for autonomous navigation applications.

It may not be feasible to run an object detection model for camera frames as they are captured in real-time. In this case, a tracker will estimate target locations for each frame between detection updates. Once detection output is received, the tracker can "correct" each track using the detected locations. Additionally, a multiple object tracker may use detections to maintain and update a velocity model for each target, which can then be used to generate quick estimates of target locations between detection updates. As more detections are received for a particular track, the tracker will ideally be able to provide more reliable estimations of object locations for intermediate frames between detections.

Beyond tracking objects like cars and pedestrians between detections, tracking algorithms in ADSs must also maintain unique identifiers for each target across detections. This way, the output of a multiple object tracker can be accumulated over several frames to construct trajectories of targets that can be fed as input to the prediction module of the ADS. However, matching new detections *without* unique IDs to the correct trajectory estimates *with* dedicated IDs (tracklet) is nontrivial.

Simpler trackers may represent the detections and existing tracklets as nodes in a weighted bipartite graph. For each pair of detections and tracklets, an edge is constructed and weighted according to a distance metric computed between features of the detection and tracklet. Finding an optimal matching from the graph may fail to produce the correct matches, es-

pecially if trajectory estimates are uncertain and have drifted too far from the targets' true locations. Therefore, multiple object trackers must aim to not only provide accurate estimates of each target's bounding box, but also minimize the number of incorrect matches (ID switches) between detections and tracklets.

Common Vocabulary

- MOT & SOT: Acronyms for Multiple Object Tracking & Single Object Tracking.
- Target: An object tracked by an object tracker.
- Tracklet: The current state and partial trajectory pertaining to a single target that is maintained by the tracker. This may include bounding box features, world or image coordinates, and the ID of the target.
- ID Switch: An event that occurs when two tracklets incorrectly receive detection updates from the other's target and effectively swap targets.

Chapter 3

Related Work

3.1 MOT Metrics

Classical Metrics

Three sets of metrics are commonly used to evaluate and compare multiple object trackers. The first set of metrics provided by Wu and Nevatia [36] consists of five measures that focus on trajectory-related errors. For example, the Mostly Tracked (MT) and Mostly Lost (ML) measures count ground-truth trajectories that are correctly tracked in over 80% and under 20% of the frames respectively. ID switches are also included in this set of metrics.

CLEAR MOT Metrics

The second set of metrics is the CLEAR MOT [3] metrics provided by Bernardin and Stiefelhagen. The first of this set is Multiple Object Tracking Accuracy (MOTA), which serves as a general error rate for trackers. The counts of misses, false positives, and mismatches are summed and divided by the total number of ground-truth objects across all frames to construct a total error rate, which is then subtracted from 1 to compute the MOTA. The maximum MOTA achievable is 1, which indicates no tracking errors were made. No minimum value of MOTA exists as the number of errors made can grow without bound, resulting in a MOTA that approaches negative infinity. The second metric included in this set is Multiple Object Tracking Precision (MOTP), which evaluates the precision of tracked bounding

Classical Metrics [36]	**CLEAR MOT [3]**	**ID Metrics [31]**
ID switches	MOTA	IDP
MT	MOTP	IDR
ML		IDF1

Table 3.1: Common MOT metric groups.

boxes. For each frame, a distance metric between each track hypothesis and the ground truth bounding box is computed and divided by the number of matched objects to compute an average precision. Typically, the distance metric used to evaluate bounding box accuracy is the intersection-over-union (IoU) metric. These values for each frame are then summed over all frames in a sequence to compute the MOTP. While MOTP summarizes bounding box accuracy over time, it only factors tracked and matched objects into its calculation. MOTA summarizes tracking errors over time including tracks that go unmatched. In tracking challenges like the MOTChallenge benchmarks [20, 26, 9, 10], MOTA is generally emphasized over MOTP.

ID Metrics

The third and most recent set of metrics is the ID metrics [31] proposed by Ristani et. al. These metrics complement the CLEAR MOT metrics and focus solely on errors related to ID matching between tracks and detections. The most widely reported metric of the set is the IDF1 metric, which is computed as the harmonic mean of Identification Precision (IDP) and Identification Recall (IDR). IDP is calculated as the ratio of true positive IDs to the number of all IDs reported by a tracker, while IDR is calculated as the ratio of true positive IDs to the total number of ground-truth IDs. While the CLEAR MOT metrics are computed at each frame, the ID metrics are computed from a global one-to-one ID mapping generated over the entire sequence. Each tracklet is mapped to the ground-truth trajectory that it is most consistent with in terms of ID labeling. If a tracklet is associated with a target with ID:1 for the majority of a sequence but occasionally switches to different targets, it is still matched to the ground truth trajectory with ID:1 in the global mapping and is not penalized for the switches. This way, the ID metrics reward trackers for better overall long-term tracking in scenarios like security and surveillance where this may be favored [8, p. 6].

Tracking Metrics for Autonomous Driving

In autonomous navigation applications, it may be more suitable to optimize tracker selection toward the classical and CLEAR MOT metrics over ID metrics, since for a given sequence, the MOTA metric penalizes each ID switch while the ID metrics do not. ID switches can be extremely costly to an ADS, as these events may result in interrupted or fragmented trajectories that weaken prediction and planning capabilities. In theory, an online tracker could be implemented to correct trajectories by modifying past ID labels. However, the search space of potential trajectory corrections can grow very quickly, corrections can require too much additional runtime, and trajectories that are modified very frequently may harm the performance of the prediction module. For these reasons, correcting the past is generally only a feature of batch tracking algorithms that can process an entire sequence offline.

3.2 MOTChallenge: The Multiple Object Tracking Benchmark

MOT research is partially guided by the MOTChallenge benchmarks [20, 26, 9, 10], which each aim to provide a standard method of comparing the performance of various trackers on a benchmark dataset. The challenge has been offered in various formats since 2015, with tasks related to 2D tracking, 3D tracking, combined tracking and segmentation, etc. For the general 2D object tracking challenges like the 2D MOT 2015 Challenge [20], the dataset contains video clips of crowded scenes of pedestrians. A text file is provided with each clip, containing detection and bounding box information for all objects present at each frame. Multiple text files may be included for a single video clip, with each text file's detections coming from a different object detection model.

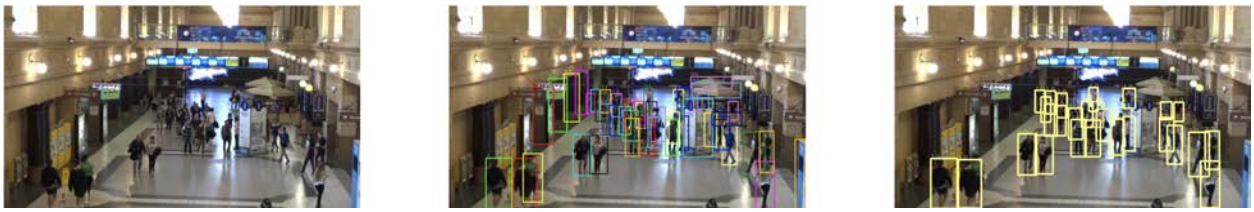


Figure 3.1: A sample sequence from the MOT20 dataset. Left: original video frame; middle: frame with labeled ground truth bounding boxes, right: Faster R-CNN detections. Image adapted from MOT20 paper [10].

Although the MOTChallenge series is intended for pedestrian tracking, it does not provide sequences similar enough to those recorded by cameras on autonomous vehicles. The MOT16 dataset [26] involves moving cameras, with a sequence recorded from a bus turning at a busy intersection. However, in this clip and in most others with camera motion, the motion is typically very gradual and does not attain speeds or experience swift changes in direction that autonomous vehicles regularly do. Recent datasets like the one provided in the MOT19 [9] and MOT20 challenges [10] have focused less on camera motion and more on extremely crowded pedestrian scenes.

Additionally, the challenge leaderboards typically emphasize accuracy over runtime, as they feature several of the metrics outlined in the previous section and are sorted by MOTA by default. While the frequency of frames processed per second is provided for each tracker in hertz, this metric is provided through the submission to the leaderboard and is not measured on standardized hardware. The leaderboard includes a marker for each submission to indicate if the tracking method is online, but no quantifiable measure appears to determine if this indicator is assigned or not. Since both trackers are not evaluated on data involving moving cameras and tracker runtimes are not standardized, past and recent iterations of the MOTChallenge are not ideal benchmarks for evaluating and selecting trackers for autonomous navigation tasks. The experiments in this paper attempt to evaluate different

multiple object trackers on the same hardware in different driving scenarios, but further development regarding tracking benchmarks and metrics is still needed.

3.3 Deep Learning in Video Multi-Object Tracking: A Survey

Summary and Contributions

Ciaparrone et. al. [8] evaluate the performance of several multiple object trackers on various MOTChallenge datasets. The authors also discuss several recent approaches to object tracking that involve deep learning to improve performance. The paper does not specifically focus on tracking for autonomous driving or navigation tasks and does not directly study tracking with moving cameras. However, the authors do investigate differences in performance between online and batch tracking algorithms and address critical issues related to tracker runtimes in a variety of settings.

A key contribution presented in this work is the identification of four main stages that most modern MOT algorithms follow. The authors then discuss various approaches to each of the four stages to identify trends that have shown success in improving tracker performance along with trends that have not proven to be necessary for state-of-the-art tracking.

The Four-Stage Framework of Modern MOT

As identified by Ciaparrone et. al., most modern MOT algorithms can be separated into four general stages: detection, feature extraction, affinity, and association. The following section provides detailed descriptions of the four stages, supplemented by additional related work pertaining to each stage.

Stage 1: Detection

The detection stage is typically carried out by a separate object detection model, which processes a camera frame and outputs bounding boxes and class labels of certain objects in the frame. As this output is used to instantiate and update the tracker over time, the accuracy of an object tracker is highly dependent on the accuracy of the bounding boxes received from the object detector. Bounding box accuracy is typically measured by calculating the IoUs between a detected bounding box and the object's corresponding ground-truth bounding box. IoU values above a certain threshold are counted as true positive detections, while those below the same threshold are counted as false positives. The precision for a set of detected bounding boxes for a single class is calculated as the ratio of true positives to (true positives + false positives). Finally, the mean average precision (mAP) is calculated by averaging the precision values over all classes.

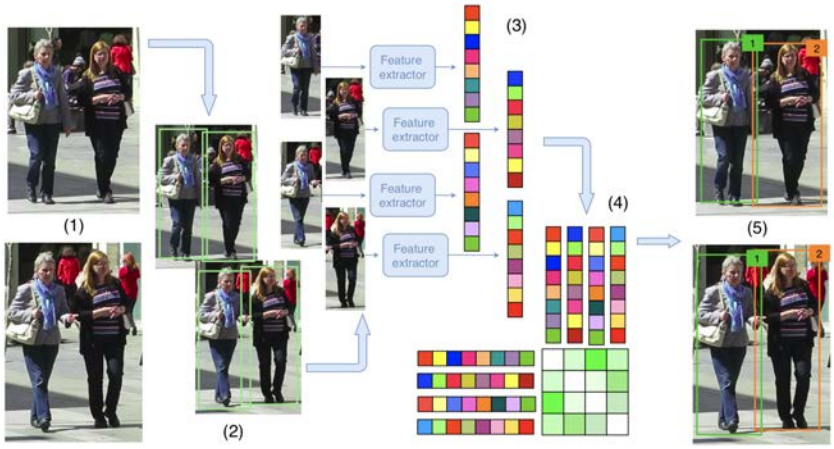


Figure 3.2: The four stages of modern MOT algorithms. Raw camera frames (1) are passed to an object detector that outputs bounding boxes (2). Bounding box image crops are passed to a feature extraction model (3). A cost matrix is computed between each pair of detection and tracklet features in the affinity stage (4). Finally, an optimal association between detections and tracklets is computed (5). Image adapted from Deep Learning in Video Multi-Object Tracking: A Survey [8].

While high-accuracy detection models such as Faster R-CNN [30] are often used to provide input to the tracker, faster but less accurate detectors can be used as well. Faster R-CNN splits the detection process into a region proposal stage and a classification stage, making it a two-shot detector. Object detectors like the Single Shot Multibox Detector (SSD) [24] and YOLOv3 [29] generate detections in one stage, which allows for much lower runtimes. However, single-shot detectors also tend to be less accurate than two-shot detectors, often having trouble with detecting smaller objects [22].

Stage 2: Feature Extraction

After the detection stage, each bounding box is then processed into features during the feature extraction stage. In simpler tracking algorithms, these features may be relatively easy-to-obtain characteristics of the bounding box, such as its dimensions or coordinates within the camera frame. However, several recent trackers utilize neural networks to process bounding box crops from the camera frame into feature vectors. This way, the features extracted from the bounding box of a target encapsulate information about the *appearance* of a target. Feature extraction networks are typically convolutional or Siamese networks that are trained offline on large datasets of bounding box image crops [4]. While neural feature extraction generally improves tracking performance, this comes at the cost of increased runtime. In a forward pass through a tracking algorithm, the feature extraction process is applied to each detected bounding box; additionally, the process may be applied to each

tracklet bounding box estimated by the tracker to generate tracklet features. Otherwise, the most recent detection features for the target are used instead.

Stage 3: Affinity

In the affinity stage, each pair of detection and tracklet features are scored according to a similarity or distance metric. For simpler trackers, the intersection over union metric (IoU) is computed for each pair at this stage using bounding box coordinate features from the previous stage. Advanced trackers with neural feature extraction may compute different distance metrics at this stage, such as the cosine similarity or Euclidean distance between feature vectors, to compute a cost for each detection-tracklet pair. Given m detections and n tracklets, $m \times n$ scores are computed to form an $\mathbb{R}^{m \times n}$ cost matrix in this stage. While Ciaparrone et. al. examine approaches to the affinity stage involving deep learning, they conclude that deep learning in this stage "has not yet been proven to be essential for a good MOT algorithm" [8].

Stage 4: Association

Finally, the association stage uses the cost matrix to handle matching detections to tracklets, discarding outdated tracklets, and instantiating new tracklets for any unmatched detections. The matching step is typically solved using the Hungarian algorithm [19] to find an optimal matching. In some cases, if the cost computed between a detection and tracklet is above a certain threshold, it may be overwritten with a large number to represent an infinite cost. This helps invalidate certain pairs of detections and tracklets from being matched. While the Hungarian algorithm runs in cubic time, an optimal matching between approximately fifty detections to fifty tracklets can be computed in under a few milliseconds using open-source solving programs as shown in Figure 3.3 [16].

Once a detection is matched to a tracklet, the coordinates of the tracklet bounding box may be updated with the coordinates of the detection's bounding box. Values from the detection's bounding box may also be used to update or edit other internal attributes of the tracker (i.e. an estimated velocity model for a target). Recent work has explored the use of deep learning models or reinforcement learning agents to aid classical matching algorithms at this stage, but the Hungarian algorithm is still the most common approach for generating an optimal assignment in milliseconds.

To discard outdated tracklets, tracking algorithms may keep count of the number of frames a tracklet has existed for without a detection match. Then, if the tracklet is not matched to a detection after a certain number of frames, it is forgotten. If a tracker forgets tracklets too quickly or is unable to construct a trajectory, the resulting driving behavior can be unsafe. In the 2018 Uber ATG crash in Tempe, AZ, the tracker was unable to build a trajectory for the pedestrian as class labels from the detector alternated between 'vehicle' and 'other' very frequently, resulting in a fatality [1]. In order to allow trackers to track targets through occlusions or mislabeled detections, the association stage must allow

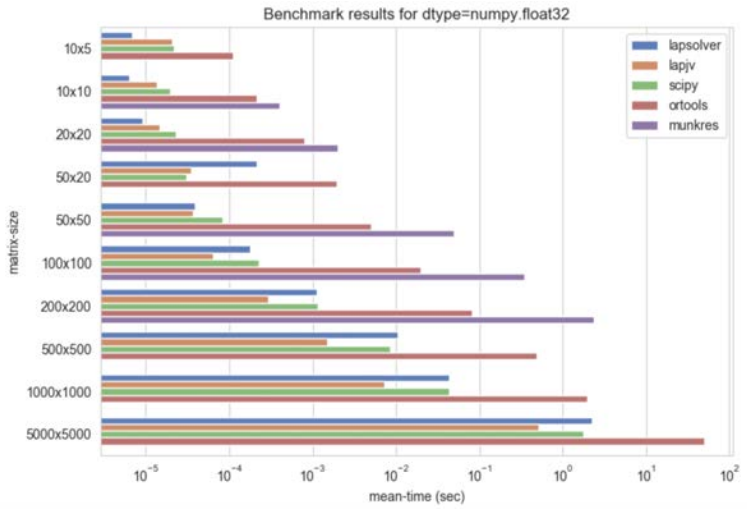


Figure 3.3: A comparison of runtimes between different popular Hungarian algorithm solving programs. Image adapted from py-lapsolver GitHub repository [16].

matches regardless of class label. In addition, the maximum time a tracklet can survive without a matched detection must be set fairly high in order to prevent the tracker from forgetting certain targets if they are occluded. This way, if a target is occluded during a detection update and goes unmatched, the tracker will continue to estimate the location of the target, preventing it from being forgotten early.

Finally, a new tracklet is instantiated for each unmatched detection. If detection quality is poor or prone to providing false positives, tracklets may be instantiated in a tentative state and confirmed only after a certain number of detections have been successfully matched.

Chapter 4

Challenges and Goals

4.1 ADS Perception Challenges

Tight Deadlines

For an autonomous driving system to be practical in the real world, it must react to changes in the environment at least as quickly as a human driver. This implies that the system must be able to run end-to-end, processing sensor input into control signals for the vehicle, in under 700 milliseconds to beat the minimal reaction time of younger drivers as found in [33]. The end-to-end deadline further indicates that each module of the pipeline must take no longer than a few hundred milliseconds to process a single input.

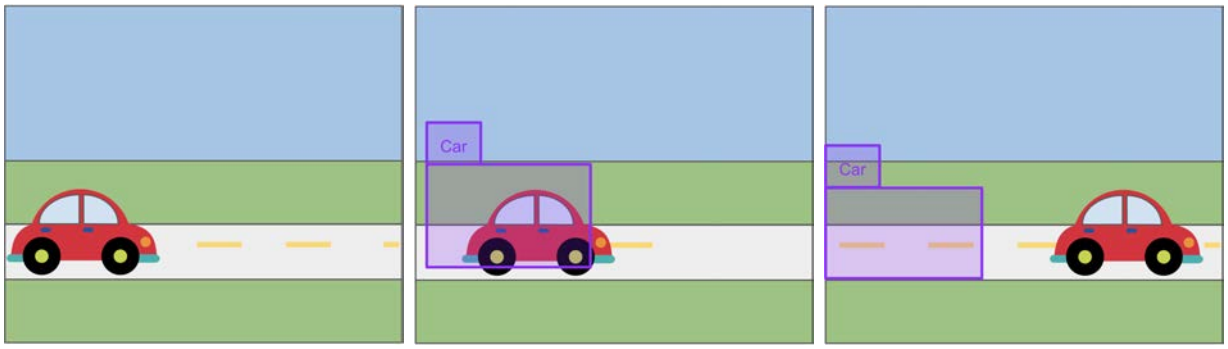
At the module level, the runtime constraint is particularly difficult for the perception module to satisfy, especially without making sacrifices with respect to accuracy. First, in a typical autonomous driving system, the perception module receives multiple gigabytes of rich data from an array of several sensors every second. At a minimum, this data includes high-definition camera images from multiple cameras attached to the vehicle. However, the data may additionally include large LiDAR point clouds and radar sensor data. To generate output for a specific point in time, the perception module must first be able to synchronize and fuse large sensor data inputs before efficiently processing them. Given the diversity of sensor data, the perception module may contain several components for a variety of tasks, including object detection, object tracking, localization, semantic segmentation, depth estimation, and ego to bird's-eye-view transformations [21]. However, certain components of a perception module are not easily parallelizable; for example, an object tracking component requires input from an object detector, so both components must run in sequence for the perception module to finish processing an input. State-of-the-art approaches to object detection and tracking tasks require neural networks to process inputs. Recent trends in deep learning for object detection have indicated that larger models with more parameters achieve higher accuracy. Nevertheless, as the size of a model grows, it becomes increasingly unlikely that it will be able to satisfy the runtime requirements of an autonomous driving system, creating a limitation on achievable accuracy.

Strict Accuracy Requirements

The accuracies of different perception components are extremely critical to the overall safety and performance of the autonomous driving system. A missed detection will not be accounted for by the prediction, planning, and control modules and may result in a collision. On the other hand, sensitive object detectors may result in uncomfortable rides with high jerk as the car quickly decelerates and accelerates in response to false-positive detections. Even if an object is accurately detected, any error related to inaccurate tracking can propagate to downstream components of the system and cause suboptimal behavior.

Classic Accuracy vs. Real-world Accuracy

The relationship between runtime and accuracy for perception module components is further complicated by the environment and driving behavior of the ego vehicle. A busy environment with several pedestrians and vehicles usually increases the runtimes of object detectors and trackers. This reduces the feasible set of object detectors and trackers for the autonomous driving system, eliminating slower but potentially more accurate models.



(a) $t = 0$ (b) $t = 1$ (c) $t = 5$

Figure 4.1: An example of how a faster but less accurate detector may allow for better driving performance than a slower but more accurate detector. Let Model #1 be a faster but less accurate object detector, while Model #2 is a slower but more accurate object detector. Frame (a) is passed to both Model #1 and Model #2 at time $t = 0$. Frame (b) is recorded at $t = 1$ after Model #1 finishes processing Frame (a). The bounding box produced by Model #1 on Frame (a) is slightly too large and outdated with respect to the conditions at Frame (b). Frame (c) is recorded at $t = 5$ after Model #2 finishes processing Frame (a). The bounding box produced by Model #2 is perfect with respect to Frame (a) but is very outdated when it becomes available at $t = 5$.

Strictly speaking, perception models produce outputs that are outdated with respect to the state of the real world, as a nonzero amount of time is required to process an input. Consider the simplified scenario in Figure 4.1 that provides an example of this phenomenon.

If an object detector requires time to run and receives camera frames with negligible delay, the bounding boxes it outputs are no longer accurate when compared to the actual locations of the detected objects. The discrepancy between accuracy with respect to the environment (real-world accuracy) and accuracy with respect to the input (classic accuracy) is a function of not only tracker runtime, but also the velocity of the detected object relative to the ego vehicle. For a stationary object and a vehicle driving at speeds of 25 mph, 45 mph, or 65 mph, detections that are perfectly accurate with respect to the input image may be inaccurate with respect to the real world by 1.12 m, 2.01 m, or 2.91 m respectively. If the detected object is not stationary and is instead another car traveling towards the ego vehicle at the same speeds, the worst-case distance discrepancies are doubled. This way, it may be more advantageous to the ADS to select models with relatively lower runtimes and lower measures of classic accuracy, as these models may provide output with higher real-world accuracy than slower models.

The complexity, runtime requirements, and accuracy demands of the perception module introduce a rich tradeoff space between the models used for each perception component. Different tracking models may perform more accurately depending on characteristics of the environment. Additionally, models that reduce the overall runtime of the perception module may not only help an ADS meet its deadlines, but also provide more accurate output with respect to the environment.

4.2 Object Tracking Challenges

Detection Accuracy

The accuracy of trackers under the tracking-by-detection paradigm relies greatly on the accuracy and runtime of the detector providing input detections. In an ADS, highly accurate object detectors may require more runtime to process a camera frame, leaving less time for a tracker to execute. However, faster object detectors may be less accurate, providing false positive detections or missing obstacles altogether. Even if a greater runtime budget is allocated to the tracking component in this case, it is very unlikely that the tracker will somehow compensate for errors made by the object detector. On the other hand, simpler trackers may see MOTA and MOTP improvements as more high-quality detection updates are received. Therefore, it is typically best to be cautious and aim to use the highest accuracy detector as much as possible.

Classic Accuracy vs. Real-world Accuracy

Due to the time constraints of an ADS and importance of high-quality detections, tracking components require online multiple object trackers that can process several frames per second. This presents a challenge in selecting trackers, as more-accurate trackers may require more complicated feature extraction and affinity stages that require more time to run. At

high driving speeds, a slower but "more-accurate" tracker may provide tracks that are outdated by hundreds of milliseconds to the prediction module, resulting in worse performance in subsequent components. On the other hand, a faster but "less-accurate" tracker with simpler feature extraction and affinity computations may provide trajectories that are only outdated by tens of milliseconds, providing the prediction module with inputs that are more accurate with respect to the real world at that instant.

Camera Motion

Multiple object trackers in an ADS must also adjust to the motion of the camera as the ego vehicle drives in order to most-accurately track targets. However, most popular tracking algorithms, datasets, and challenges either focus primarily on tracking from a stationary camera or assume simple velocity models for the targets. This causes the majority of multiple object trackers to provide inaccurate estimates of target bounding boxes during periods of accelerating or turning camera motion. Another approach to tracking involves the application of neural feature extraction for each camera frame. This way, for intermediate frames between detections, the tracker can search a region close to the most recent detection coordinates to estimate the new location of the target. However, in cases involving substantial camera motion, this search space may no longer fully include the target and make tracking less accurate. Therefore, a state-of-the-art online tracker may require modifications that account for camera motion to be implemented most effectively in an ADS.

Occlusions

Additionally, trackers must be able to track objects through occlusions. For example, in the case where a tracked pedestrian is hidden behind a passing car for a few frames, the pedestrian's trajectory should ideally be maintained throughout the occlusion to better estimate its future location. This is a common problem in crowded scenes when pedestrians cross paths frequently. In autonomous vehicles, the occlusion problem is exacerbated due to the added motion through the outdoor environments. Pedestrians may be occluded several times throughout a driving sequence in several ways due to other pedestrians, passing cars, garbage cans, traffic signs and lights, trees, etc. Multiple object trackers with complex neural feature extraction stages are generally designed to target this occlusion problem. If a pedestrian occludes another for a few frames before moving away, a simple affinity stage using IoU scores may result in an ID switch between both pedestrians. However, a tracker with a neural feature extraction stage will use features of each pedestrian's appearance in the affinity stage in an attempt to avoid this problem.

4.3 Goals

Accurate and fast object tracking can address many of the challenges presented thus far and improve autonomous driving systems overall. While much of MOT research focuses on tracking humans, not much is known about how general measures of MOT performance translate to autonomous driving systems. However, further research at the intersection of tracking and autonomous driving has great potential to improve the overall safety and performance of modern autonomous driving systems. Therefore, the primary goal of the system design and experiments in the following chapters is to evaluate different types of object tracking algorithms in a variety of simulated driving scenarios. In doing so, this work makes the following contributions:

- Provides object tracking experimentation and evaluation capabilities within an ADS. MOT-related components of Pylot [15], an ADS integrated with the CARLA simulator [13], are described in Chapter 5. Chapter 6 includes details specific to the implementation of three MOT algorithms in Pylot. This work allows researchers to experiment with different object trackers beyond the ones evaluated in this thesis.

- Demonstrates that no single type of tracker performs best across all driving scenarios. This implies that a robust ADS must either intelligently switch between existing trackers depending on the environment, or that the development of new trackers better-suited for autonomous driving is needed.

- Shows that popular tracking metrics may not always respond in a predictable manner to changes in driving scenarios. Several MOT metrics must be considered together in order to best understand and evaluate the performance of modern object trackers in autonomous driving contexts.

- Outlines research directions for future work related to object tracking for ADSs. The challenges listed in this chapter coupled with the results discussed in Chapters 7 and 8 explain the most pressing needs of object trackers in ADSs.

Chapter 5

System & Simulator

5.1 Pylot

Pylot [15] is an autonomous driving platform designed to address some of the aforementioned challenges related to ADSs. For experiments in Chapter 7, Pylot specifically interfaces with CARLA [13], an open-source simulator for autonomous driving research. However, Pylot can also operate as an ADS for real-world cars and has operated on a Lincoln MKZ on a closed test track.

The underlying execution engine of Pylot is ERDOS [14], an open-source execution engine for autonomous driving and robotics applications. ERDOS is a streaming dataflow system designed for tasks requiring low latency and high throughput. Pylot components for ADS tasks like object tracking or planning are implemented as ERDOS operators. Operators are connected by data streams to form a dataflow graph, and an operator can both subscribe and publish data to multiple data streams. To provide determinism, ERDOS uses a system of watermarks. This allows components of Pylot to process data that is synchronized for a particular time interval from multiple sources.

Tracking-Related Operators, Message Types, and Setup

This section explains how different MOT algorithms are integrated into Pylot and provides further details on tracking-related operators and message types. Implementation details specific to particular trackers are addressed in Chapter 6. Figure 5.1 shows a high-level diagram of the tracking-related portion of the Pylot dataflow graph. This example is relevant to the configuration of Pylot in later experiments involving CARLA, which use ground-truth detection input from the PerfectDetectorOperator.

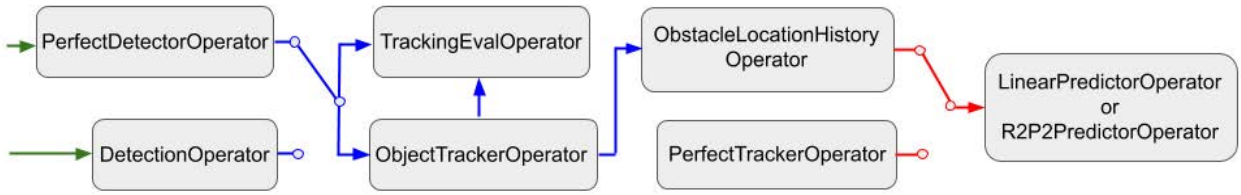


Figure 5.1: A diagram of the tracking-related portion of the Pylot pipeline. Green edges represent FrameMessages, blue edges represent ObstaclesMessages, and red edges represent ObstacleTrajectoriesMessages. Small circles are used to indicate a switch that can flip between two inputs. Only a subset of all operators and connected data streams are depicted.

Operators

An overview of ERDOS operators related to tracking is provided below:

- The **DetectionOperator** wraps an object detection model. The operator receives camera frames as input and outputs detection information including object bounding boxes, class labels, detection confidence scores, etc.
- The **PerfectDetectorOperator** is implemented to output ground-truth bounding boxes of objects obtained from CARLA for each frame recorded by the ego vehicle camera.
- The **ObjectTrackerOperator** wraps a tracker instance. The operator receives camera frames and outputs tracker bounding box information at each time step. The operator also receives detection data as input, though this is not required at each time step.
- The **ObstacleLocationHistoryOperator** accumulates tracker bounding boxes to output trajectories of object locations to the prediction module. This task involves several input data streams, collecting data for ego vehicle location and speed, depth frames, and tracked bounding boxes. This is required as ObjectTrackerOperator output is recorded with respect to the coordinate system of the camera frame; however, the locations must be converted into world coordinates to form usable trajectories for the prediction module.
- The **PerfectTrackerOperator** is implemented to output ground-truth trajectories of objects obtained from CARLA for each time step.
- The **TrackingEvalOperator** requires input from a detection operator and a tracking operator to calculate and log tracking-related metrics. Metrics are computed via the motmetrics package [17].

Messages

Pylot operators transmit data via ERDOS messages that are sent on and read from data streams. Pylot implements several extensions of the base ERDOS message to send different types of data between operators.

- The **FrameMessage** contains a single camera frame for a given timestep. It sends camera frames from the environment to operators for object detection and tracking.
- The **ObstaclesMessage** is used to transmit detection information from object detection to object tracking components. Detection information is encoded as a list of DetectedObstacle objects, which store information for a particular obstacle's ID, bounding box, and class label.
- The **ObstacleTrajectoriesMessage** is used to send obstacle trajectories from the perception module to prediction components. A message contains a list of Trajectory objects, which store the most recent bounding box and past locations of an object up to a predefined limit.

Tracker Implementation

Listing 5.1: Example code for a tracker class.

```
class MultiObjectTracker(object):
    def __init__(self):
        self.tracklets = []

    def reinitialize(self,frame,obstacles):
        det_features = [feature(frame, ob) for ob in obstacles]
        track_features = [feature(frame, tr) for tr in self.tracklets]
        cost_matrix = compute_costs(det_features, track_features)
        # Run Hungarian algorithm to compute associations
        # Modify self.tracklets to contain updated tracklets
        self.tracklets = updated_tracklets

    def track(self,frame):
        tracker_estimates = []
        for tracklet in self.tracklets:
            tracklet.update(frame)
            tracklet.age += 1
            if tracklet.age <= LIMIT:
                tracker_estimates.append(tracklet)
        self.tracklets = tracker_estimates
        return self.tracklets
```

As of this writing, Pylot includes several features that allow for new trackers to be implemented within the tracking component. To integrate a multiple object tracker into Pylot, a user must write a tracker class that extends Pylot's MultiObjectTracker. This subclass must provide two functions, `reinitialize` and `track`.

The `reinitialize` function receives a camera frame and a list of detected obstacles as input, and is responsible for carrying out the feature extraction, affinity, and association stages of the tracking process. The `track` function receives the current camera frame as input and is responsible for progressing each tracklet one time step forward. Although the `track` function receives a camera frame as input, this frame may be unused depending on the tracker. A tracker using a simple velocity model to update tracks may not require the image as input, while a tracker updating tracks using image features extracted via neural networks may depend entirely on the image.

Data Collection

Pylot also includes a set of scripts that collectively allow a user to gather data for training and evaluating a feature extraction model. Data gathering scripts and operators collect bounding box logs and camera frames from CARLA, and a separate script collects text logs of detected bounding boxes in the MOTChallenge format. Another script is included to extract logged bounding boxes from camera frames and organize the crops into separate training and test datasets organized by object class.

As the CARLA simulator does not provide ground truth 2D bounding boxes from a camera view, a variety of heuristics and some manual data cleaning are necessary to obtain reasonable data for training perception models. To clean a dataset of bounding box image crops for training tracker feature extraction models, we remove bounding boxes with areas below 1500 pixels, as this indicates that the object is too far away from the ego vehicle. Crops that are too small do not provide enough appearance information to the trackers, and at inference time, the trackers are not likely to receive very small bounding boxes from most object detectors. Finally, manual data cleaning is required in certain instances to remove low-quality bounding boxes, including those with occluded targets.

Tracker Experimentation and Evaluation

To compare different object trackers as fairly as possible, Pylot provides a few flags that allow the user to specify values for common tracker parameters:

- `min_matching_iou`: If the IoU distance metric is used in the affinity stage, the threshold specified by this flag is used to mark certain pairs of detections and tracks as invalid matches. Note that IoU *distance* is calculated as $1 - \text{IoU}(\text{bbox}_1, \text{bbox}_2)$, so `min_matching_iou` actually specifies a maximum IoU distance of 1 - `min_matching_iou` between detections and tracks. The default value is 0.5.

- `obstacle_tracking_max_age`: This flag is used in the association stages of the trackers to determine when a tracklet is outdated and ready to be removed from the tracker. For all trackers, the age is incremented by one whenever a tracklet fails to be matched during a detection update. The age is also incremented by one for every frame received by the tracker without corresponding detections. In either case, immediately after the age is incremented, it is compared to the limit set by `obstacle_tracking_max_age`. A tracklet is removed if its age is strictly greater than `obstacle_tracking_max_age`. Once a successful match is made, the age of the matched tracklet is reset to zero. The default value is 3.

In Chapter 7, different combinations of these parameters are tested in various driving scenarios in CARLA.

5.2 CARLA

The CARLA simulator [13] provides various types of sensor data for simulated vehicles, including cameras, LiDAR, etc. Additionally, CARLA provides support for building and evaluating driving in predetermined scenarios. The following section details different scenarios implemented in CARLA that are used for tracking experiments.

Scenarios

(a) TrackPedestrians (b) ManyPedestrians

Figure 5.2: Screenshots recorded from the TrackPedestrians and ManyPedestrians scenarios. Both scenarios are constructed in CARLA Town 01, but TrackPedestrians has an average of 4.5 pedestrians tracked within 50m of the ego vehicle. ManyPedestrians increases this average to 6.7 pedestrians.

TrackPedestrians

The TrackPedestrians scenario places the ego vehicle at an intersection in front of a green light. The road is empty and there are approximately three to five pedestrians on either sidewalk moving forward away from the vehicle. The pedestrians all walk at a constant speed of 1.4 m/s (approximately 3 mph), making them easy to track well with simpler trackers. This way, the TrackPedestrians scenario serves as an easy testing ground for trackers, and experiments that run in this scenario should provide baseline runtime and accuracy metrics for each tracker.

ManyPedestrians

The ManyPedestrians scenario follows the same setup as the TrackPedestrians scenario, except the number of pedestrians visible in a given frame is doubled. The pedestrians still walk at a speed of 1.4 m/s making their motion predictable over time, but the increased number of them is meant to create more occlusions. Experiments that run in this scenario are compared to similar experiments in the TrackPedestrians scenario to understand how tracker runtimes and accuracies are affected as the number of targets increases.

ManyPedestriansWithTurn

The ManyPedestriansWithTurn scenario follows the same configuration of pedestrians as the ManyPedestrians scenario. However, the ego vehicle starts further back on the road around a corner and aims to maintain a target speed of 10 m/s (approximately 22 mph). The vehicle drives approximately five meters, makes a right turn along the road, and drives for an additional five meters before reaching the ego vehicle start point from the ManyPedestrians scenario. Experiments in this scenario are used to determine how the performance of the trackers is affected by turning camera motion, as the other scenarios involve straight-line paths.

Graphics

CARLA not only allows users to create and organize a variety of driving scenarios in different maps, but also provides several options for users to configure the appearance of the scenarios as recorded by simulated cameras. As of release 0.9.8, users can select daytime or nighttime modes to change lighting conditions [12]. Different weather options are also available, allowing users to specify the sun altitude angle, cloudiness, wind, and precipitation. Users can control several camera parameters including focal length, motion blur, and color temperature. Beyond these options for scenario customization, CARLA has promised additional options to be released in the future. For example, the team hopes to expand the available options for pedestrian clothing, hairstyles, and skin [11].

While CARLA continues to expand the number of options for scenario customization, the appearance of the simulator is still far from the appearance of the real world. Simulation

Figure 5.3: Four different weather settings at the same location in CARLA Town02. Top left: clear day, top right: daytime rain, bottom left: clear sunset, bottom right: daytime shortly after rain. Image adapted from CARLA paper [13].

graphics are limited, making perception in simulation a less challenging task than perception in the real world. This domain gap between the CARLA simulator and the real world affects the performance of perception components that rely on camera frames for input. If simulator conditions are not varied enough, models trained on data from the simulator may easily overfit to the simulator conditions and perform terribly on real-world data. If the same experiments are performed in real-world translations of the CARLA scenarios, exact MOT metrics are unlikely to hold. However, the conditions of the CARLA simulator are complex enough that the general conclusions from the experiments in this work are useful for real-world autonomous driving.

Hardware

All experiments in Chapter 7 are performed on a machine featuring a Nvidia Titan X Pascal GPU with CUDA version 10.2, driver version 440.33.01, and 12GB of memory. The machine runs Ubuntu 18.04.3 LTS. Multiple instances of the CARLA simulator and Pylot are able to run simultaneously with this configuration of hardware.

Chapter 6

Multiple Object Trackers in Pylot

Three different object tracking algorithms from the following works are implemented in Pylot: Simple Online Realtime Tracking [6] (SORT), DeepSORT [35], and Distractor-Aware Siamese Region Proposal Networks [38] (DaSiamRPN). Both SORT and DeepSORT are multiple object trackers, while DaSiamRPN — a single object tracker — is incorporated into Pylot's tracking framework to create a multiple object tracker. The following section contains further implementation details for each of the three trackers in Pylot.

6.1 Simple Online and Realtime Tracking (SORT)

SORT is a lightweight, open-source multiple object tracker that relies on the Kalman filter and Hungarian algorithm. Object positions are estimated from frame to frame with a linear constant velocity model. Each tracklet maintains a state vector containing the target's estimated position and velocity. During the affinity stage, a cost matrix is constructed by computing the IoU distance between each detection and tracklet. Certain assignments are marked impossible if the IoU distance for the assignment is greater than a certain threshold. Optimal assignments are generated from the cost matrix using the Hungarian algorithm. For each matched (detection, track) pair, the detected bounding box is used to update the position components of the target state, and the velocity components are estimated via a Kalman filter. This simple framework does not involve neural networks beyond the detection stage, making SORT an extremely fast multiple object tracker. However, this simplicity also hinders the accuracy of SORT, especially with respect to the aforementioned general tracking issues such as occlusions or camera motion.

Pylot Implementation

Minor modifications are made to the implementation of SORT in Pylot. These modifications allow the user to modify parameters to the SORT tracker more easily, such as matching IoU

thresholds, maximum tracker ages, and individual tracklet parameters like target IDs and class labels. No major changes to the algorithm are made.

6.2 SORT with a Deep Association Metric (DeepSORT)

DeepSORT extends SORT by incorporating appearance information for each tracked object into the affinity stage. The underlying Kalman filter and target state definition are unchanged, but a convolutional feature extraction model is used to process bounding box crops of targets into feature vectors. During the affinity stage, a cost between each pair of detections and tracks is computed via a weighted sum of two separate distance metrics. The first is a squared Mahalanobis distance metric computed using the bounding box from a detection and Kalman state vector of a track. The second is a cosine distance computed between each pair of detection and tracklet feature vectors. The weighted sum of these distances essentially allows DeepSORT to account for both a position cost and appearance cost between each detection and tracklet.

During the association stage, the weighted costs pass through a Mahalanobis gating step, which marks certain pairs of detections and tracks invalid for matching. Interestingly, the original paper mentions that the squared Mahalanobis distance term should be ignored in cases involving substantial camera motion. By default, the open-source implementation provided by the Wojke et. al. does not include the squared Mahalanobis distance in the cost matrix calculations as of this writing [34]. However, this Mahalanobis gating step in the association stage is still included.

On the test set from the MOT16 challenge [26], DeepSORT matches detections to tracks more accurately than SORT, avoiding lower counts of ID switches and tracking through occlusions more effectively. In addition, DeepSORT achieves higher MOTA, MT, and ML metrics. However, DeepSORT achieves a slightly lower MOTP than SORT, and also produces more false positive tracks.

Pylot Implementation

Several modifications are made to the original implementation of DeepSORT in order to best integrate the tracker into Pylot. First, the original feature extraction model is trained on the MARS dataset [37], which contains videos of pedestrians walking around a university. In order to best track both pedestrians *and* vehicles, a separate feature extraction model is trained on a dataset generated from CARLA. To obtain the dataset, camera frames and 2D object bounding boxes were collected from a series of random drives through the Town01 and Town02 maps in CARLA. For each drive, the towns were populated with 50-100 vehicles spanning 23 classes as well as 100-200 pedestrians spanning 11 classes. The number of instances of vehicles and pedestrians is much greater than the number of classes in order to collect images of each class from several different angles during a drive. Furthermore, each

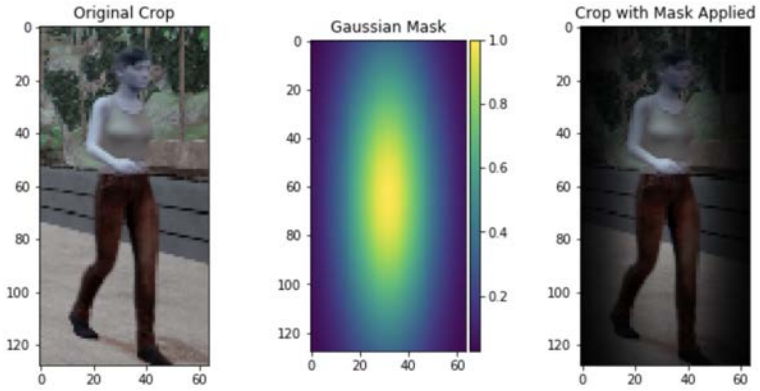


Figure 6.1: An example showing how the Gaussian mask is applied to crops of pedestrians before being passed to the DeepSORT feature extraction model. The specific values of the covariance matrix were generally chosen to preserve pedestrian pixels, so the mask extends vertically further than it does horizontally.

random drive began at a randomly chosen start point in the map, and weather conditions were modified over the drives in each town in an effort to collect a diverse set of crops of vehicles and pedestrians against several different backgrounds. This way, the dataset promotes the learning of relevant features of vehicles and pedestrians and helps models ignore background elements. However, due to time and data collection constraints, the final dataset is smaller than the MARS dataset and contains 18,194 crops in total. To train the network, the dataset is split into a training set of 16,386 crops while the test set contains the remaining 1,807 crops. The dataset can be accessed at this link.

Before image crops are passed to the model during training, a Gaussian mask with covariance matrix given by $\text{diag}(0.35^2, 0.22^2)$ is applied to each crop [25]. This assumes the object of interest is centered within the bounding box, as the application of the Gaussian mask lowers pixel values around the edges of the box (see Figure 6.1). This modification is intended to guide the model to ignore background elements and learn more salient features of the pedestrian or vehicle.

Next, the notion of tentative tracks is removed from the original implementation of DeepSORT. By default, the original implementation initializes tracks in a tentative state, requiring at least three consecutive detection updates before the tracklet is confirmed. If a tentative tracklet goes unmatched during a detection update, it is deleted. Additionally, the example use of the tracker provided by the authors only outputs confirmed tracklets. While this technique may be useful in limiting false positives, it may also result in unsafe driving in an ADS as each valid tracklet must wait up to several hundred milliseconds before it is confirmed and sent to the prediction component. Moreover, this technique will delete every proposed tracklet if detections are not provided for every frame. Therefore, the implementation of DeepSORT in Pylot confirms each after one detection match.

Name	Kernel Size/Stride	Output Size
Conv 1	3×3/2	32×63×63
Conv 2	3×3/2	64×31×31
Conv 3	3×3/2	128×15×15
Conv 4	1×1/2	256×8×8
Conv 5	1×1/2	256×4×4
Conv 6	3×3/2	512
Conv 7	1×1/1	1024

Table 6.1: Architecture of modified DeepSORT feature extraction model [25] with 1,902,080 parameters. Batch normalization and ReLU activations follow each Conv layer. Layers Conv 4 and Conv 5 act as pooling layers.

Name	Kernel Size/Stride	Output Size
Conv 1	3×3/1	32×128×64
Conv 2	3×3/1	32×128×64
Max Pool 3	3×3/2	32×64×32
Residual 4	3×3/1	32×64×32
Residual 5	3×3/1	32×64×32
Residual 6	3×3/2	64×32×16
Residual 7	3×3/1	64×32×16
Residual 8	3×3/2	128×16×8
Residual 9	3×3/1	128×16×8
Dense 10		128

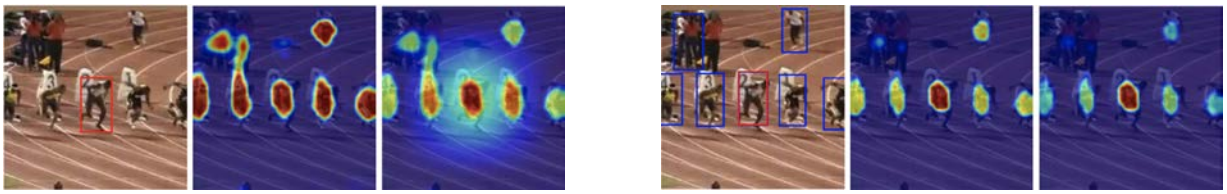
Table 6.2: Architecture of original DeepSORT feature extraction model [35] with 2,800,864 parameters. Batch normalization and ELU activations follow each Conv, Residual, and Dense layer.

Finally, the architecture and training procedure for the feature extraction model is modified to aid in learning features optimized toward the cosine similarity cost used in the affinity stage. The network is adapted from the version implemented by Maiya [25] and is integrated with the original DeepSORT code [34], allowing users to easily train their own feature extraction models. A Siamese network implemented in PyTorch [28] with multiple convolutional layers is trained to minimize a triplet loss function. Given an anchor image A, a "positive" image P of the same class as the anchor, and a "negative" image N of a different class, the loss function attempts to minimize the squared cosine distance between A and P feature vectors while maximizing the squared cosine distance between A and N feature vectors. Therefore, Siamese networks trained with triplet loss functions are not guided toward learning features that best represent a class, but features that best distinguish a class from other object classes.

The architecture of Siamese network has several differences compared to the original feature extraction model in DeepSORT. However, these models were implemented to serve different purposes; the DeepSORT model is trained for pedestrian tracking tasks involving real-world pedestrians, while the Siamese network is originally trained in [25] to track 184 classes of vehicles from the Nvidia AI City Challenge dataset [27]. The current implementation of DeepSORT in Pylot trains the Siamese network on a dataset of both vehicles and pedestrians recorded from the CARLA simulator [13]. See Tables 6.1 and 6.2 for a side-by-side comparison of the architectures of this new Siamese network and the original network provided by the DeepSORT authors.

6.3 Distractor-Aware Siamese Region Proposal Networks (DaSiamRPN)

DaSiamRPN is a single object tracker that combines a Siamese feature extraction network with a novel training strategy to learn distractor-aware features. A velocity model for each target is not estimated, as the tracker relies entirely on image features to track an object. At each time step, a region proposal network proposes a search space for the target using the image and most recent detection. The tracker employs bounding box regression within the proposed search region to locate the target and update the track. As tracking without new detections still requires an image frame to pass through a neural net, this step is slower when compared to SORT and DeepSORT. However, this also means that tracker estimations between detections are based on image features rather than estimated velocities. This generally allows Siamese networks to provide more accurate estimations between detections.



(a) General Siamese tracker

(b) Distractor-aware Siamese tracker

Figure 6.2: Comparison of general Siamese tracker and DaSiamRPN training processes and resulting feature maps. DaSiamRPN is provided data with labeled distractors, allowing it to track the object of interest while assigning much less weight to targets that have similar image features. Image adapted from DaSiamRPN paper [38].

Pylot Implementation

In Pylot, DaSiamRPN is implemented as a multiple object tracker. A tracker object is instantiated for each detected obstacle, and detections are matched to tracks using the IoU distance as the cost for the assignment problem. The original region proposal network is trained using a variety of techniques to learn to account for distractors in the image. Additionally, the original distractor-aware training method requires special annotations (see Figure 6.2) and several data augmentation techniques to be applied to the dataset.

This strategy was difficult to reproduce on the CARLA dataset collected for training the DeepSORT feature extractor due to time and data-collection constraints. Therefore, the current implementation of DaSiamRPN in Pylot does not use a network trained on data collected from CARLA. Instead, pretrained weights from the DaSiamRPN submission to the VOT-18 challenge [18] are used, as the challenge required trackers to track a variety of objects including humans and vehicles. These weights are learned by applying distractor-aware

training to the ImageNet [32] and COCO [23] datasets, which capture a greater diversity of lighting conditions, backgrounds, and objects than the CARLA dataset. In theory, this should allow the VOT-18 weights for DaSiamRPN to track CARLA objects over multiple frames.

As is similarly implemented in the affinity stages of SORT and DeepSORT, the cost matrix is modified so that distant detections and tracklets are marked as invalid pairs. In the current implementation of DaSiamRPN in Pylot, this means that values in the cost matrix are overwritten with `NaN` if they are greater than the value specified by `min_matching_iou`. This invalidates certain pairings of detections and tracks to the Hungarian algorithm program [16] in the association stage. New tracks are instantiated for unmatched detections, and unmatched tracks are only discarded if they have not been updated for a number of frames greater than the value specified by `obstacle_track_max_age`.

Chapter 7

Experiments

In the following sections, SORT, DeepSORT, and DaSiamRPN are evaluated using several different parameter configurations in a variety of CARLA scenarios. For each tracking parameter configuration, data is collected over three separate trials. Metrics like MOTA and MOTP for each configuration are averaged across the trials in an effort to provide more accurate estimates of the metrics and reduce the influence of rare events. While the simulated scenarios are meant to be deterministic, we notice that small variable-length delays between the CARLA simulator, scenario runner, and Pylot can affect the start time of Pylot's object tracking component by a few seconds. To fairly assess accuracy metrics for each tracker, ground truth detections from the PerfectDetectionOperator are used as detection input. Unless stated otherwise, the PerfectDetectionOperator is configured in most experiments to output detections for objects that are within 50 m of the ego vehicle. Route and motion plans are provided by the simulator, and a PID controller is used to execute these plans. The planner and controller aim to stay on the road and avoid other cars and pedestrians who may be crossing the path of the ego vehicle. All code, data, and figures related to the experiments can be accessed at this link.

7.1 Tracking Performance Baselines

This section evaluates each tracker in the TrackPedestrians scenario using a camera mounted on a stationary vehicle in order to establish baseline runtimes and accuracy metrics. This allows for the analysis of tracker performance in the simplest scenario a typical autonomous vehicle may experience. For each trial, the trackers run in the TrackPedestrians scenario for a thirty second period and receive ground truth detections for each frame. This way, the tracking bounding box accuracy is not affected by any noise or uncertainty from an object detection model, and corresponding metrics like MOTP are only affected by incorrect matches.

Table 7.1: Runtime Statistics in TrackPedestrians Scenario

	SORT	**DeepSORT**	**DaSiamRPN**
Average	8.37	41.46	91.04
Min	2.72	14.46	35.91
Max	87.59	209.60	271.61
Std. Dev.	6.84	19.33	30.32
1st Percentile	3.28	17.35	40.20
10th Percentile	3.90	22.19	55.24
25th Percentile	4.52	27.30	68.97
50th Percentile	5.57	36.80	86.76
75th Percentile	10.13	50.25	109.82
90th Percentile	15.80	67.17	130.16
99th Percentile	36.33	107.33	176.45

Analysis of Runtimes

As seen in Table 7.1, the SORT tracker has the lowest average runtime of the three at 8.37ms, as there is no neural feature extraction required in the algorithm. Although both DeepSORT and DaSiamRPN use neural networks during the feature extraction stage, the average runtime of DeepSORT (41.46ms) is much lower than the average runtime of DaSiamRPN (91.04ms). This is likely due to the fact that DeepSORT is designed as a multiple object tracker, so only one forward pass containing a batch of n bounding boxes is required to generate n feature vectors. On the other hand, DaSiamRPN is a single object tracker; the DaSiamRPN implementation in Pylot requires n instances of the DaSiamRPN single-object tracker, so n forward passes are needed to generate n feature vectors. Further work in this implementation could batch the feature extraction step to require only one forward pass, which would greatly reduce DaSiamRPN runtimes in Pylot. However, even with this slowdown, all three trackers have relatively low average runtimes that do not eliminate them from performing in an ADS.

Analysis of MOT Metrics

Several MOT metrics are analyzed as the IoU threshold (specified by the `min_matching_iou`) is varied between values 0, 0.1, 0.25, 0.5. These values are used to eliminate certain detections from being matched to tracks. The value of 0 indicates that no pairs of detections and tracks are marked invalid during the association step; every pair is considered as a possible match. On the other end, a value of 0.5 expresses the most strict requirement, as pairs of detections and tracks that do not have an IoU of at least 0.5 are unable to be matched. 0.5 is chosen as it is a common IoU distance used in object detection literature (between detected and ground truth bounding boxes) for separating true positive detections from false positive detections

[23, 30]. However, 0.5 is also generally higher than the thresholds chosen in object trackers [5, 34]. Experiments are also conducted with IoU thresholds of 0.1 and 0.25 in order to provide additional granularity.

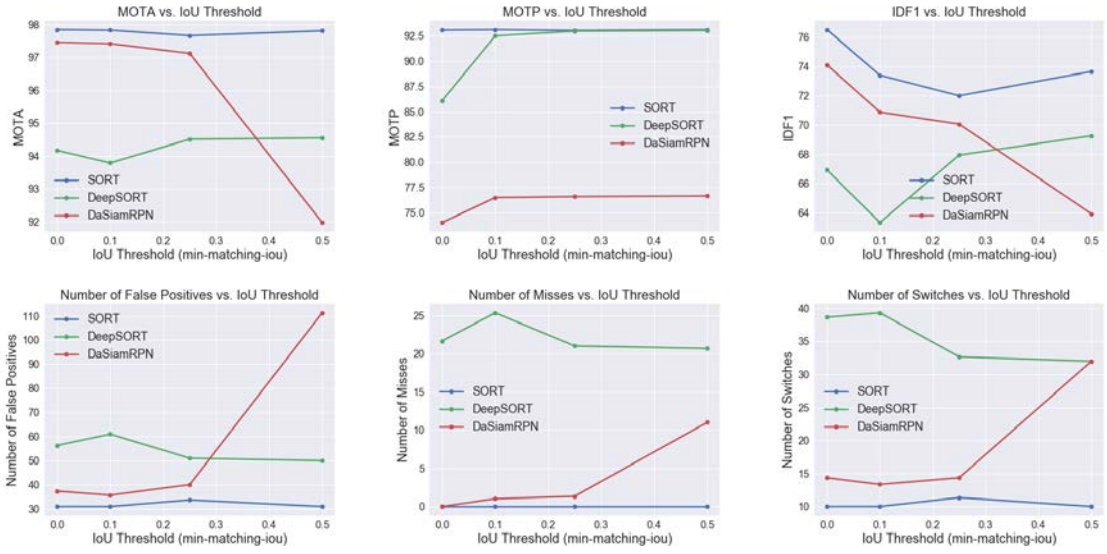
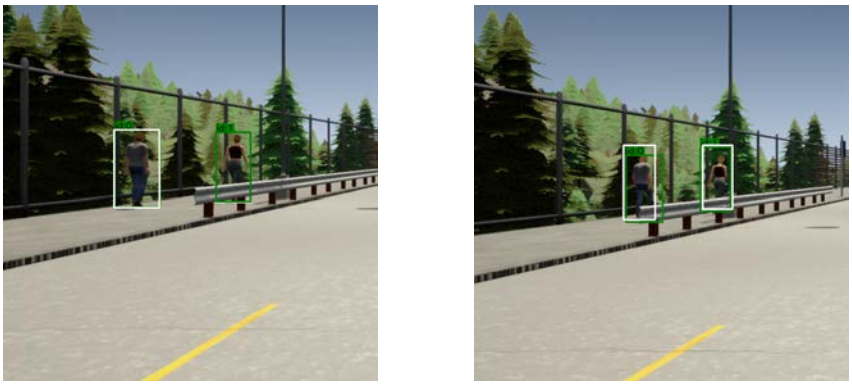


Figure 7.1: CDFs of tracker runtimes as detection frequency decreases from once per frame to once every four frames.

Overall, the plots in Figure 7.1 indicate that SORT performs best in the TrackPedestrians scenario. Across the four chosen IoU thresholds, it consistently achieves higher MOTA, MOTP, and IDF1 metrics while also achieving lower numbers of ID switches, misses, and false positives compared to DeepSORT and DaSiamRPN. The scenario is well-suited for SORT, as pedestrians move at predictable, constant speeds with respect to the (stationary) ego vehicle, and there are minimal occlusions.

DaSiamRPN also performs very well in this scenario, though it experiences difficulty when the IoU threshold is too strict at 0.5. In addition, DaSiamRPN achieves much lower MOTP than SORT and DeepSORT. This indicates that the IoU metrics calculated between DaSiamRPN tracklets and detections is often lower than 0.5, which may explain why DaSiamRPN then has to instantiate new tracks and sees higher values of ID switches, misses, and false positives.

Examining this further, it appears that DaSiamRPN is penalized because it resizes each bounding box after the feature extraction step to fit the visible portions of the target. This results in a mismatch between the ground-truth bounding box and estimated target bounding box, which lowers the MOTP of DaSiamRPN (see Figure 7.2). Expectations for the MOTP of DaSiamRPN in later experiments must adjust accordingly to this. However, qualitatively, the bounding boxes estimated by DaSiamRPN still appear to fit the targets quite well. For the purposes of an ADS, it still appears that DaSiamRPN could produce accurate trajectories



(a) SORT (b) DaSiamRPN

Figure 7.2: Comparison of tracklets (green) to ground truth bounding boxes (white) between SORT and DaSiamRPN. DaSiamRPN resizes detected bounding boxes using image features while SORT uses the ground truth bounding box dimensions directly. This may be responsible for the lower MOTP achieved by DaSiamRPN relative to the MOTP achieved by the other trackers.

in similar scenarios, even if it does not compare favorably with other trackers in terms of MOTP.

Finally, DeepSORT tends to provide higher numbers of tracking errors than the other trackers across most IoU thresholds. While these errors contribute to lower MOTA values, DeepSORT still achieves very high MOTA in the TrackPedestrians scenario overall. Moreover, higher IoU thresholds greatly help DeepSORT eliminate incorrect associations, causing MOTA and MOTP to trend upwards while tracking errors trend downwards.

Overall, SORT has the best performance of the three trackers in this experiment for two main reasons. The first is that the scenario is simple; there are few occlusions, the car is stationary, and the pedestrians move at constant velocities. The second is that SORT does not make errors based on appearance features of the targets, while DeepSORT and DaSiamRPN may occasionally make errors as they rely on appearance features for matching or tracking.

7.2 Tracking Performance vs. Detection Frequency

In this experiment, detection updates are sent to the tracker operator every 2nd, 3rd, or 4th frame, leaving the trackers to generate and output estimated bounding boxes for each frame between detections. For trials in which detection updates are received every 3rd or 4th frame, the limit set by `obstacle_tracking_max_age` is increased from 5 to 10. This allows the trackers to continue to track through an occlusion that results in a missed detection update, even as detections are received less frequently.

Analysis of Runtimes

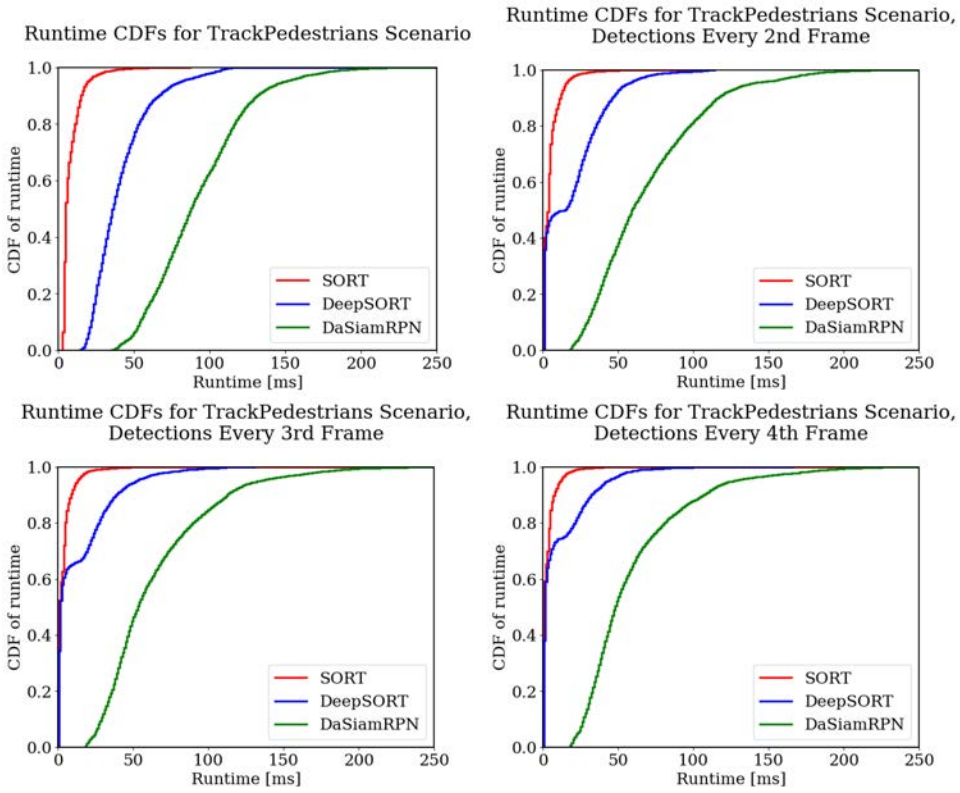


Figure 7.3: CDFs of tracker runtimes as detection frequency decreases from once per frame to once every four frames.

As the number of frames between detections increases, the average runtimes of SORT and DeepSORT decrease while the average runtime of DaSiamRPN remains fairly consistent. When no detections are received for a given timestep, both SORT and DeepSORT skip their feature extraction, affinity, and association stages. Instead, both trackers update each track according to the estimated velocity of the target. This allows some speedup for SORT,

though it is relatively small given that SORT typically requires very little time to run. The runtime savings are much more impactful for DeepSORT, as the neural feature extraction stage requires much more runtime in DeepSORT than in SORT.

The reduced runtimes for intermediate frames between detections are reflected in the runtime plots. For example, when detections are received every third frame, the inflection point in the DeepSORT runtime CDF curve is near 66%, indicating that DeepSORT and SORT behave similarly for frames between detection updates as one would expect. The remaining frames require neural feature extraction, so the rightmost values of each CDF curve approach those of the DeepSORT runtime CDF curve from the previous section.

Unlike SORT and DeepSORT, DaSiamRPN relies on image features to estimate target positions between detections rather than a constant velocity model. Therefore, it performs a similar amount of work for each frame whether or not the frame is received with bounding boxes from an object detector. MOT algorithms that do not rely on image features to estimate target bounding boxes generally have an advantage in terms of runtime; however, the use of image features should theoretically allow for better estimates of target bounding boxes and perform better in terms of other tracking metrics.

Analysis of MOT Metrics

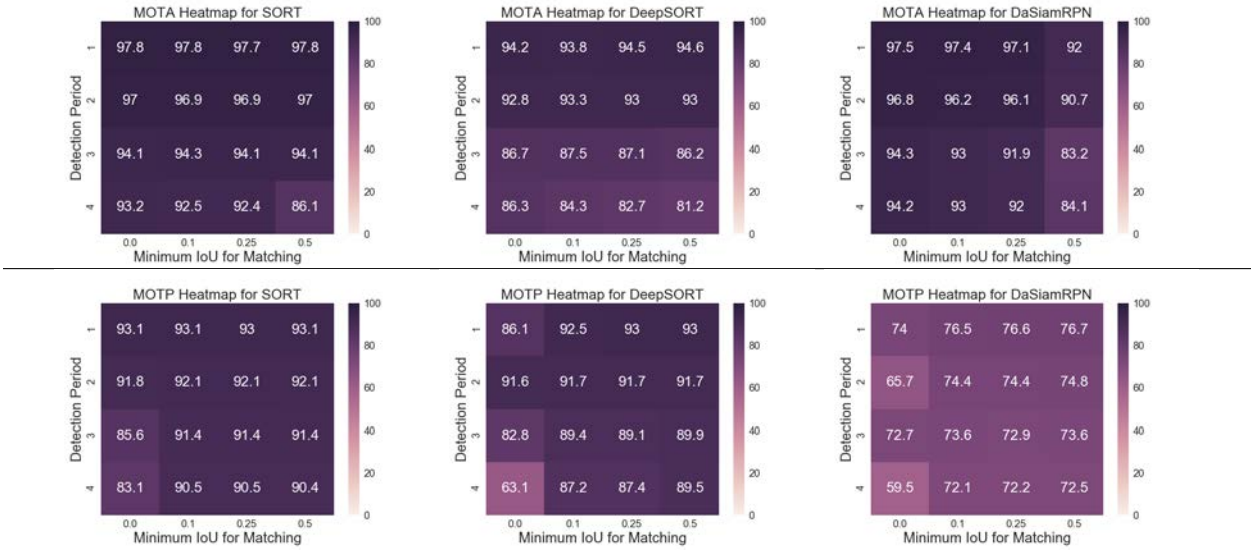


Figure 7.4: Heatmap plots showing how MOTA (first row) and MOTP (second row) are affected by increasing time between detections and stricter matching criteria in the Track-Pedestrians scenario. A detection period of 1 indicates that detections are received every frame. Therefore, the top row of each heatmap corresponds to results for experiments in the previous section.

As expected, Figure 7.4 shows that all of the trackers achieve lower MOTA values as the detection period increases and updates become more sparse. While the MOTA values are fairly close for each configuration in the TrackPedestrians scenario, SORT slightly outperforms the other two trackers in most cases. MOTP for all three trackers does not change much as the IoU threshold increases, and they slightly decrease as the detection periods increase. However, MOTP values are notably lower when there is no minimum IoU threshold, showing that requiring a nonzero overlap between detections and tracklets can significantly help improve tracking by eliminating unlikely pairs. This shows that in predictable driving scenarios, a simple tracker MOT algorithm like SORT is preferable over more complex trackers, as it achieves slightly higher accuracy across several tracker metrics while also achieving much lower runtimes.

7.3 Tracking Performance vs. Number of Targets Tracked

This experiment attempts to determine how the performance of each tracker scales as the number of targets in a scene increases. Data is collected from sequences in both the TrackPedestrians and ManyPedestrians scenarios, and each trial is listed in Table 7.2. To gain enough samples for runtimes when less than three or more than nine targets are visible, the radius of the perfect detector is shifted from its default value of 50m to 20m, 70m, or 75m in different trials.

Config. ID	**Scenario**	**Detection Radius (m)**
1	TrackPedestrians	20
2	TrackPedestrians	50
3	ManyPedestrians	50
4	ManyPedestrians	70
5	ManyPedestrians	75

Table 7.2: Experiments used in the analysis of runtimes. Each row corresponds to a configuration that was run with each of the three trackers (15 total trials recorded). For each tracker, data recorded from the five trials results in at least one hundred runtime values for each number of targets between 1 and 12.

Analysis of Runtimes

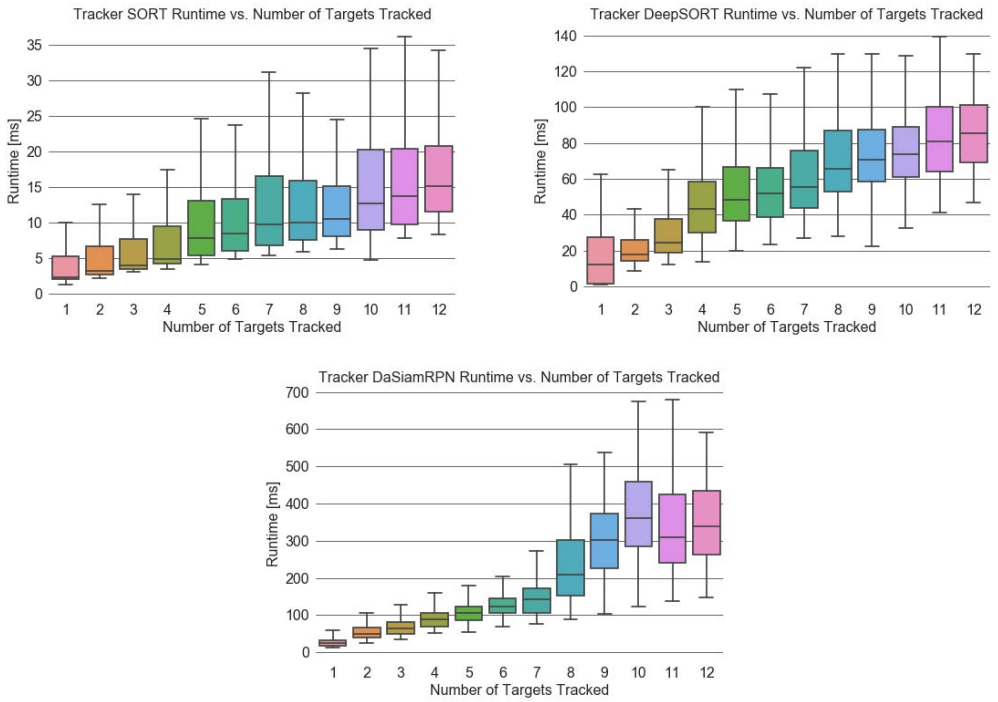


Figure 7.5: Distributions of tracker runtimes as number of targets increases. Note that the y-axes of each plot are different.

A visualization of tracker runtime distributions as the number of targets in a scene increases is provided in Figure 7.5. For each of the trackers, the average runtime generally increases as the number of targets tracked increases. While SORT runtimes increase, tracking twelve objects is still possible in well under 50 ms. Due to this speed, the number of targets tracked generally should not prohibit the use of SORT in an ADS in most driving scenarios. On the other hand, more advanced trackers like DeepSORT and DaSiamRPN may not be able to feasibly run in an ADS if the number of targets grows too much. In particular, the runtime for DaSiamRPN grows very quickly and should only be used to track a few targets at a time. Again, this drastic increase can be avoided if the Pylot implementation of DaSiamRPN is modified to process n bounding box crops in one pass rather than n passes.

Analysis of MOT Metrics

To understand how tracker metrics are affected by a greater number of targets, the experiments in Section 7.1 are reproduced in the ManyPedestrians scenario with results depicted in Figure 7.6. The TrackPedestrians scenario contains an average of 4.5 pedestrians tracked

in each frame, while the ManyPedestrians scenario contains an average of 6.7 pedestrians tracked in each frame. Figure 7.7 is included to compare how MOTA and MOTP change from TrackPedestrians to ManyPedestrians across different tracking parameter configurations.

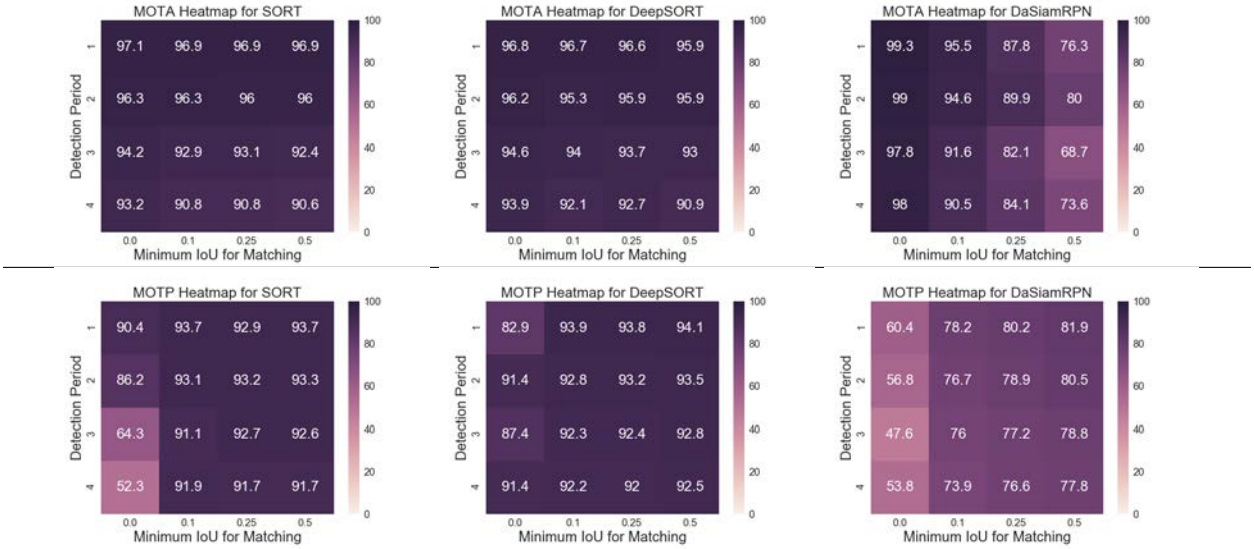


Figure 7.6: Heatmap plots showing MOTA (first row) and MOTP (second row) for different tracking parameter configurations the ManyPedestrians scenario.

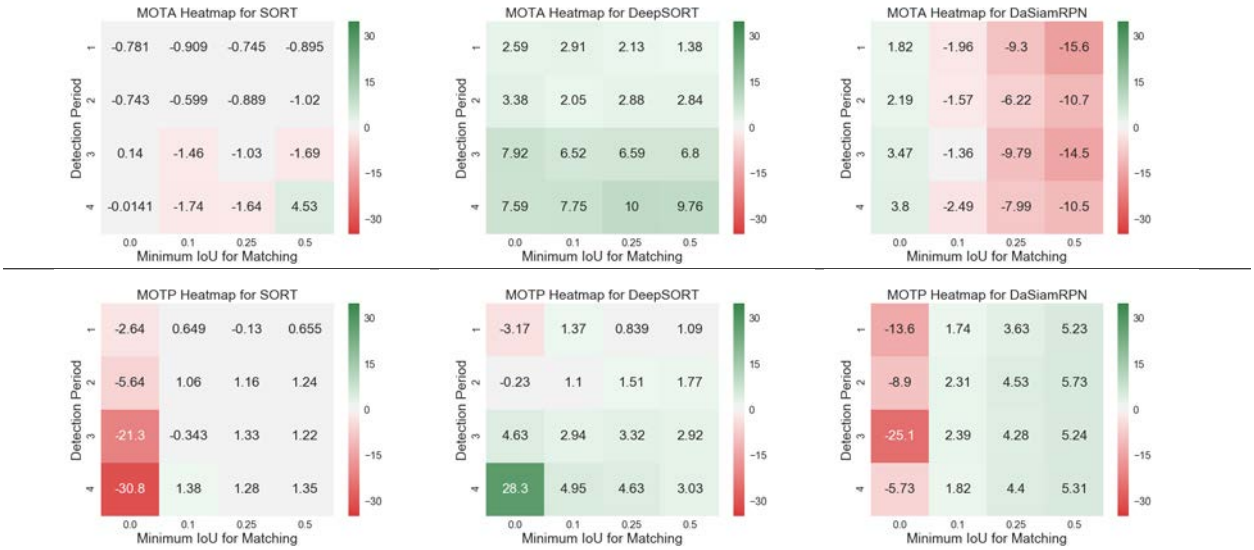


Figure 7.7: Difference in MOTA and MOTP going from TrackPedestrians scenario to ManyPedestrians scenario (essentially Figure 7.4 subtracted from Figure 7.6).

Each tracker responds differently to the increase in the number of targets. SORT performs similarly in both scenarios, though it achieves much lower MOTP when there is no minimum IoU threshold to invalidate distant associations. As the time between detections increases, the MOTP decreases even further. On average, SORT produces more misses, ID switches, and false positives in the ManyPedestrians scenario as well. However, the overall performance of SORT is still very strong in this scenario, as the camera is still stationary and pedestrians move throughout the scene at constant velocities.

	SORT	**DeepSORT**	**DaSiamRPN**
Average Δ Misses	+15.31	+41.79	+74.29
Average Δ False Positives	+132.64	-28.12	+228.38
Average Δ ID Switches	+23.10	-7.48	+27.60

Table 7.3: Average change in different types of tracking errors from TrackPedestrians to ManyPedestrians. These errors affect the MOTA for each tracker.

DeepSORT tends to achieve higher MOTA in the ManyPedestrians scenario across most configurations of IoU thresholds and detection frequencies. This does not imply that DeepSORT makes fewer tracking errors in the ManyPedestrians scenario than in the TrackPedestrians scenario; instead, it shows that DeepSORT makes errors at lower rates. In the ManyPedestrians scenario, DeepSORT produces more misses on average, but also surprisingly produces fewer false positives and ID switches as seen in Table 7.3. As the frequency of detections decreases, DeepSORT outperforms SORT in terms of MOTA and MOTP.

Finally, DaSiamRPN generally performs worse with more visible pedestrians. It achieves the highest average MOTA of the three trackers at 99.3, which occurs when a detection is received for every frame and there is no minimum IoU required for matching. However, the MOTP falls to 60.4 in this configuration, down 13.6 points from the same configuration in TrackPedestrians. On average, DaSiamRPN creates more misses and ID switches when there are more targets. In addition, DaSiamRPN proposes many more false positives.

Qualitatively, it appears that DaSiamRPN and the other trackers see improvements in MOTP as they are able to track a greater number of pedestrians in the distance. The pedestrians that are farther away move slowly and have smaller bounding boxes, so trackers do not have to shift bounding box estimates very far for each time step. This appears to help increase the MOTP of each tracker. Overall, the trackers still make more errors on average when there are more pedestrians in a scene, but the errors per pedestrian decrease and can lead to increased MOTA and MOTP in some cases.

7.4 Tracking While Driving Through a Turn

This experiment evaluates the trackers in the ManyPedestriansWithTurn scenario and compares the results to those from the ManyPedestrians scenario in the previous section. Sequences in the ManyPedestriansWithTurn scenario are only recorded for twenty seconds in simulation time instead of thirty seconds; this way, one can study how turning and severe camera motion affects tracking metrics before the ego vehicle in the scenario stops turning and drives forward for too long. However, the scenario still includes several meters of predictable straight-line driving at the beginning and end. SORT and DeepSORT rely on learning parameters for a constant velocity model for each target, and are therefore not well-equipped to handle severe turning motion immediately followed by straight-line motion. DaSiamRPN applies neural feature extraction to each frame to track each target, but the large number of targets still makes this a difficult task.

Analysis of MOT Metrics

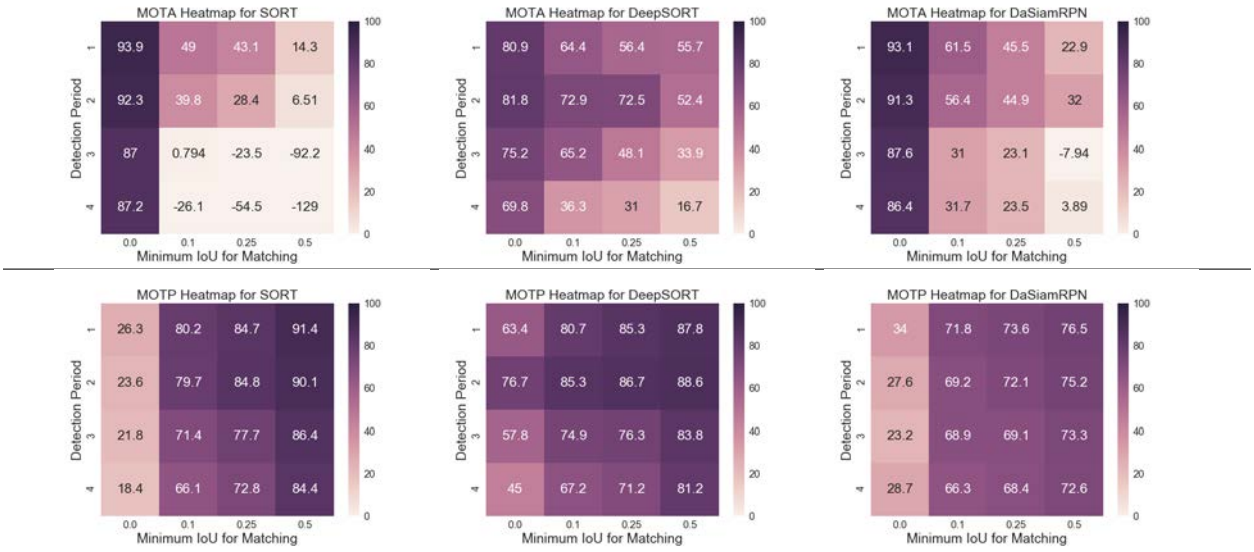


Figure 7.8: Heatmap plots showing how MOTA (first row) and MOTP (second row) are affected by increasing time between detections and stricter matching criteria in the ManyPedestriansWithTurn scenario. Note that in the MOTA plots, cells holding negative values are filled with the same color no matter the magnitude of the value.

With camera motion added, SORT struggles to track pedestrians, especially if a minimum IoU threshold is nonzero. The lack of a minimum IoU threshold prevents SORT from discarding tracklets, forcing it to maintain every tracklet initialized. As seen in Figure 7.8,

	SORT	DeepSORT	DaSiamRPN
Average Δ Misses	+68.08	+83.98	-14.71
Average Δ False Positives	+296.42	-2.44	-9.88
Average Δ ID Switches	+93.62	-0.08	+22.50

Table 7.4: Average change in different types of tracking errors from ManyPedestrians to ManyPedestriansWithTurn. These errors affect the MOTA for each tracker.

	SORT	DeepSORT	DaSiamRPN
Δ Misses (min, max)	(-41.67, +307.67)	(-152.33, +489.67)	(-225.33, +181.00)
Δ False Positives (min, max)	(-285.33, +1264.33)	(-99.33, +102.67)	(-633.67, +526.00)
Δ ID Switches (min, max)	(-19.33, +242.33)	(-30.67, +25.33)	(-77.33, +115.33)

Table 7.5: Minimum and maximum changes in different types of tracking errors from ManyPedestrians to ManyPedestriansWithTurn, computed across all 16 pairs of detection frequencies and IoU thresholds.

this leads to high MOTA in the scenario, though the estimated bounding boxes SORT produces stray very far from the targets and result in extremely low MOTP. As the detection frequency decreases and the IoU threshold increases, SORT tends to achieve negative MOTA. When viewed in simulation, trials with nonzero IoU thresholds show that SORT initiates and deletes tracks for most frames during the turn, because the camera motion prevents matches between most pairs of detections and poorly estimated tracklets.

DeepSORT responds much more favorably to the scenario across most parameter configurations, though it still experiences increased difficulty as the detection frequency decreases and the IoU threshold increases. Table 7.4 indicates that DeepSORT experiences an increased number of misses. This increase is not the only factor in reducing MOTA, as certain configurations may result in different numbers of errors that average close to 0. Table 7.5 provides minimum and maximum changes in tracking errors across all sixteen parameter configurations. While the number of misses decreases by 152.33 for one configuration between the ManyPedestrians and ManyPedestriansWithTurn scenarios, overall MOTA typically decreases for most configurations.

DaSiamRPN performs slightly better than SORT across most parameter configurations. The MOT implementation of DaSiamRPN computes IoU distances as the cost metric in the affinity stage, just as the SORT tracker does. This explains some of the similar patterns observed in the MOTA and MOTP heatmaps. However, DaSiamRPN has the advantage of neural feature extraction to help track targets between detections, while SORT relies on

a simple constant velocity model. Therefore, DaSiamRPN is able to track targets between detection updates more accurately and achieve lower error rates.

Overall, all three of the trackers struggle to accurately track targets when camera motion is added. While this is partially expected, the way the struggles are reflected across different tracking metrics is not as predictable. Further experimentation along with qualitative observations are needed to fully explain tracking behavior and understand performance when object tracking is studied in an ADS.

MOTA and MOTP Over Time

To further investigate how tracking performance is influenced by camera motion, it may be useful to not only examine final MOTA and MOTP for a sequence, but also intermediate MOTA and MOTP as the sequence occurs. To study this, we first select trials where we detect every frame and use an IoU threshold of 0.1. For each tracker, three trials are conducted with this parameter configuration. The recorded MOTA and MOTP at each frame are averaged across the three sequences to create an average sequence. Figure 7.9 then depicts the 10-frame moving averages of the MOTA and MOTP values over an average sequence.

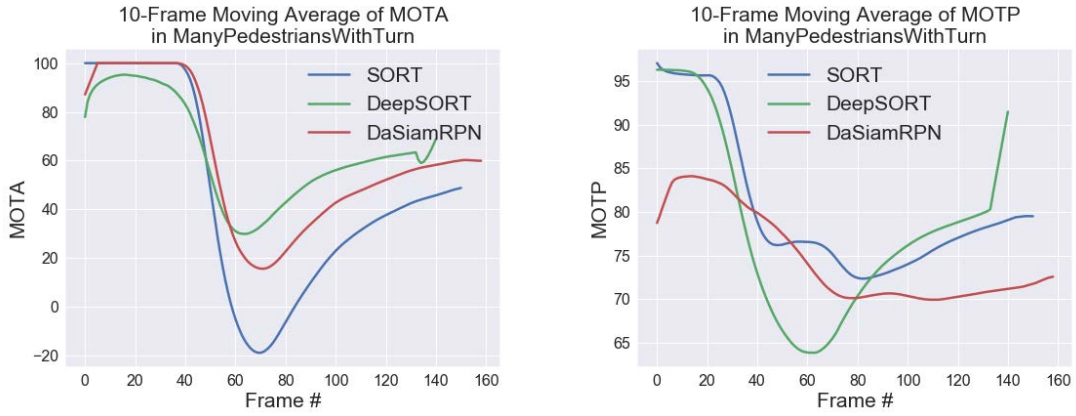


Figure 7.9: Moving averages of MOTA and MOTP in the ManyPedestriansWithTurn scenario. Data for these plots is collected from trials with detections every frame and IoU thresholds of 0.1.

In both plots in Figure 7.9, any of the three trackers may lead the other trackers in the corresponding metric at a certain point. At the beginning of the scenario, MOTA is high for all three trackers as there are only a few pedestrians to predict and the ego vehicle drives forward. Once the ego vehicle begins to turn, all three trackers experience difficulty and produce more errors. SORT is the most affected, followed by DaSiamRPN and then DeepSORT. Once the ego vehicle completes the turn and the scenario begins to

look exactly like the ManyPedestrians scenario, MOTA begins to recover toward values seen in the ManyPedestrians scenario. MOTP also decreases for all three trackers during the turn and begins to recover once the turn is completed. However, the degree of recovery or steepness in the curves is not as easily explainable.

These patterns indicate that for the ManyPedestriansWithTurn scenario, optimal tracking from a fixed set of three trackers cannot be achieved via the selection of only one tracker. Instead, each tracker may perform better than another at various points in the sequence. Therefore, switching between trackers depending on the conditions of the environment may be able to yield better tracking performance from the fixed set of trackers over the entire sequence. However, further experimentation in more scenarios and with more tracking parameter configurations is needed to further develop this idea.

Chapter 8

Future Work

There are several areas in which this work can be extended in order to understand more about object tracking in autonomous driving systems. A few of these areas are outlined in the following sections.

Broader Evaluation

First, the findings of this paper are limited in that only three types of object tracking algorithms are evaluated. Trackers that have achieved state of the art performance across recent MOTChallenge benchmarks like Tracktor [2] and MPNTrack [7] may perform better than the slightly older approaches evaluated in this work. In addition, a larger dataset of CARLA bounding box crops should be collected for training feature extraction models. Ideally, the magnitude of this dataset should be closer to those of standard object tracking datasets, such as the MARS dataset [37] for pedestrians or the Nvidia AI City Challenge dataset [27] for vehicles. Finally, the experiments in this work involve varying scenarios, IoU thresholds, and detection frequency. However, other values for tracking parameters like maximum tracker age should also be tested, and more complicated driving scenarios should be used to evaluate different object trackers as well.

SOT to MOT

While this work included the modification of an SOT algorithm in DaSiamRPN to track multiple objects, further refinements to this approach could be made. First, the implementation of DaSiamRPN should be modified to synchronize updates for each object together so that only one neural net forward pass is required rather than one pass per target. This modification should not change the positive correlation between tracker runtime and the number of targets tracked, but it should greatly reduce the runtime of DaSiamRPN overall. This way, DaSiamRPN will be feasible in a greater number of driving scenarios and more competitive with the other trackers in terms of runtime. In addition, different cost functions besides IoU distance should be used in the affinity stage; functions computed on extracted feature vectors may help improve the association stage and reduce ID switching.

In general tracking research, further experimentation applying SOT algorithms to the MOT task may be helpful in developing stronger MOT algorithms. While the implementation of DaSiamRPN in Pylot contains code to extend DaSiamRPN for MOT tasks, this type of code should be generalized into a framework for other SOT algorithms to easily be extended for MOT purposes. This would allow for experimentation in the feature extraction, affinity, and association stages, allowing researchers to find the best-performing MOT version of a single object tracker. Ideally, developers of SOT algorithms would be able to easily submit results to MOTChallenge benchmarks given a proper SOT-to-MOT framework.

Explicitly Accounting for Camera Motion

As seen in the results of Section 7.4, further research toward tracking multiple objects when camera motion is present is needed for a variety of applications involving autonomous navigation. While current tracking approaches can occasionally compensate for straight-line motion, turning motion greatly affects tracking performance. A tracker that additionally receives camera speed and direction as input should allow for more accurate estimates of target locations between detections.

Tracking and Driving Safety

Finally, while the experiments presented in this work study MOT algorithms in isolation, further research is needed to understand how different types of tracking algorithms affect the entire driving pipeline. The experiments covered in the previous chapter observe how common tracking metrics are affected by different trackers and parameters; however, the outcome of driving behavior in response to these changes is not studied. This raises several questions regarding tracking metrics and outcomes in an ADS:

- How do different tracking accuracy metrics and runtimes affect the performance of intermediate ADS components for prediction, planning, and control?
- How do different tracking accuracy metrics and runtimes affect the number of collisions or near-misses in different driving scenarios?
- Can a rule-based system for choosing the optimal object tracker be devised if one knows the environment dynamics, ego vehicle speed, and properties of each available tracker?
- Can an ADS dynamically choose the optimal object tracker in different driving scenarios in order to drive more safely?

All of these questions require experiments involving the full end-to-end pipeline of Pylot to run across multiple scenarios. Further experimentation involving dynamic deadlines for each component may also be helpful in understanding how faster MOT algorithms help provide additional time to other components. Ultimately, the work presented in this report is preliminary regarding object tracking and ADSs, and there are several paths toward advancing knowledge in this complex domain.

Chapter 9

Conclusion

This work examines object tracking in ADSs. First, previous work and challenges related to perception in ADSs and multiple object tracking are reviewed. Next, experiments are conducted using three different MOT algorithms implemented in Pylot, an autonomous driving platform. The experiments are performed in the CARLA simulator [13] in three different driving scenarios in an effort to understand how the three MOT algorithms behave under different environmental conditions. For most of the experiments, MOT performance is analyzed from both runtime and accuracy perspectives, with several popular MOT metrics supporting discussions of the latter. The analysis provides insight into the current state of MOT and shows that MOT in autonomous driving contexts still has several areas for improvement. Overall, while this work covers only a portion of object tracking and autonomous driving research in simulation, it provides key insights relevant to tracking and autonomous driving in the real world. Additional experimentation related to object tracking for ADSs is crucial to improving the performance of not only the tracking component, but also downstream prediction, planning, and control components. Ultimately, progress at the tracking stage of an ADS will help drive the development of safer, more reliable autonomous vehicles forward.

Bibliography

[1] Ensar Becic. *VEHICLE AUTOMATION REPORT*. Tech. rep. HWY18MH010. Washington: National Transportation Safety Board, 2019. URL: https://dms.ntsb.gov/public/62500-62999/62978/629713.pdf.

[2] Philipp Bergmann, Tim Meinhardt, and Laura Leal-Taixé. "Tracking without bells and whistles". In: *CoRR* abs/1903.05625 (2019). arXiv: 1903.05625. URL: http://arxiv.org/abs/1903.05625.

[3] Keni Bernardin and Rainer Stiefelhagen. "Evaluating Multiple Object Tracking Performance: The CLEAR MOT Metrics". In: *J. Image Video Process.* 2008 (Jan. 2008). ISSN: 1687-5176. DOI: 10.1155/2008/246309. URL: https://doi.org/10.1155/2008/246309.

[4] Luca Bertinetto et al. *Fully-Convolutional Siamese Networks for Object Tracking*. 2016. arXiv: 1606.09549 [cs.CV].

[5] Alex Bewley. *sort*. https://github.com/abewley/sort. 2020.

[6] Alex Bewley et al. "Simple online and realtime tracking". In: *2016 IEEE International Conference on Image Processing (ICIP)*. 2016, pp. 3464–3468. DOI: 10.1109/ICIP.2016.7533003.

[7] Guillem Brasó and Laura Leal-Taixé. *Learning a Neural Solver for Multiple Object Tracking*. 2019. arXiv: 1912.07515 [cs.CV].

[8] Gioele Ciaparrone et al. "Deep learning in video multi-object tracking: A survey". In: *Neurocomputing* 381 (Mar. 2020), pp. 61–88. ISSN: 0925-2312. DOI: 10.1016/j.neucom.2019.11.023. URL: http://dx.doi.org/10.1016/j.neucom.2019.11.023.

[9] P. Dendorfer et al. "CVPR19 Tracking and Detection Challenge: How crowded can it get?" In: *arXiv:1906.04567 [cs]* (June 2019). arXiv: 1906.04567. URL: http://arxiv.org/abs/1906.04567.

[10] P. Dendorfer et al. "MOT20: A benchmark for multi object tracking in crowded scenes". In: *arXiv:2003.09003[cs]* (Mar. 2020). arXiv: 2003.09003. URL: http://arxiv.org/abs/1906.04567.

[11] Francesc Domene. *CARLA 0.9.6 release*. 2019. URL: http://carla.org/2019/07/12/release-0.9.6/.

[12] Francesc Domene. *CARLA 0.9.8 release*. 2020. URL: http://carla.org/2020/03/09/release-0.9.8/.

[13] Alexey Dosovitskiy et al. "CARLA: An Open Urban Driving Simulator". In: *Proceedings of the 1st Annual Conference on Robot Learning*. 2017, pp. 1–16.

[14] Ionel Gog et al. *erdos*. https://github.com/erdos-project/erdos. 2020.

[15] Ionel Gog et al. *pylot*. https://github.com/erdos-project/pylot. 2020.

[16] Christoph Heindl. *py-lapsolver*. https://github.com/cheind/py-lapsolver. 2020.

[17] Christoph Heindl. *py-motmetrics*. https://github.com/cheind/py-motmetrics. 2020.

[18] Matej Kristan et al. *The sixth Visual Object Tracking VOT2018 challenge results*. 2018.

[19] Harold W. Kuhn. "The Hungarian Method for the assignment problems". In: *Naval Research Logistics Quarterly* 2 (1955), pp. 83–97.

[20] L. Leal-Taixé et al. "MOTChallenge 2015: Towards a Benchmark for Multi-Target Tracking". In: *arXiv:1504.01942 [cs]* (Apr. 2015). arXiv: 1504.01942. URL: http://arxiv.org/abs/1504.01942.

[21] Ming Liang et al. "Deep Continuous Fusion for Multi-sensor 3D Object Detection". In: *Computer Vision – ECCV 2018*. Ed. by Vittorio Ferrari et al. Cham: Springer International Publishing, 2018, pp. 663–678. ISBN: 978-3-030-01270-0.

[22] Tsung-Yi Lin et al. "Focal Loss for Dense Object Detection". In: *CoRR* abs/1708.02002 (2017). arXiv: 1708.02002. URL: http://arxiv.org/abs/1708.02002.

[23] Tsung-Yi Lin et al. "Microsoft COCO: Common Objects in Context". In: *CoRR* abs/1405.0312 (2014). arXiv: 1405.0312. URL: http://arxiv.org/abs/1405.0312.

[24] Wei Liu et al. "SSD: Single Shot MultiBox Detector". In: *CoRR* abs/1512.02325 (2015). arXiv: 1512.02325. URL: http://arxiv.org/abs/1512.02325.

[25] Shishira R Maiya. *DeepSORT: Deep Learning to Track Custom Objects in a Video*. 2019. URL: https://nanonets.com/blog/object-tracking-deepsort/.

[26] Anton Milan et al. "MOT16: A Benchmark for Multi-Object Tracking". In: *CoRR* abs/1603.00831 (2016). arXiv: 1603.00831. URL: http://arxiv.org/abs/1603.00831.

[27] Milind Naphade et al. *The 4th AI City Challenge*. 2020. arXiv: 2004.14619 [cs.CV].

[28] Adam Paszke et al. "Automatic differentiation in PyTorch". In: (2017).

[29] Joseph Redmon and Ali Farhadi. "YOLOv3: An Incremental Improvement". In: *CoRR* abs/1804.02767 (2018). arXiv: 1804.02767. URL: http://arxiv.org/abs/1804.02767.

[30] Shaoqing Ren et al. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks". In: *CoRR* abs/1506.01497 (2015). arXiv: 1506.01497. URL: http://arxiv.org/abs/1506.01497.

[31] Ergys Ristani et al. "Performance Measures and a Data Set for Multi-Target, Multi-Camera Tracking". In: *CoRR* abs/1609.01775 (2016). arXiv: 1609.01775. URL: http://arxiv.org/abs/1609.01775.

[32] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: 10.1007/s11263-015-0816-y.

[33] Heikki Summala. "Brake Reaction Times and Driver Behavior Analysis". In: *Transportation Human Factors* 2 (Sept. 2000), pp. 217–226. DOI: 10.1207/STHF0203_2.

[34] Nicolai Wojke. *deep_sort*. https://github.com/nwojke/deep_sort. 2020.

[35] Nicolai Wojke, Alex Bewley, and Dietrich Paulus. "Simple Online and Realtime Tracking with a Deep Association Metric". In: *2017 IEEE International Conference on Image Processing (ICIP)*. IEEE. 2017, pp. 3645–3649. DOI: 10.1109/ICIP.2017.8296962.

[36] Bo Wu and Ram Nevatia. "Detection and Tracking of Multiple, Partially Occluded Humans by Bayesian Combination of Edgelet based Part Detectors". In: *International Journal of Computer Vision* 75.2 (2007), pp. 247–266. DOI: 10.1007/s11263-006-0027-7. URL: https://doi.org/10.1007/s11263-006-0027-7.

[37] Liang Zheng et al. "MARS: A Video Benchmark for Large-Scale Person Re-Identification". In: *Computer Vision – ECCV 2016*. Ed. by Bastian Leibe et al. Cham: Springer International Publishing, 2016, pp. 868–884. ISBN: 978-3-319-46466-4.

[38] Zheng Zhu et al. "Distractor-aware Siamese Networks for Visual Object Tracking". In: *European Conference on Computer Vision*. 2018.