Bridging the Gap Between Modular and End-to-end Autonomous Driving Systems

Eric Leong

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2022-79
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-79.html

May 12, 2022

Bridging the Gap Between Modular and End-to-end Autonomous Driving Systems

by Eric Leong

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:

Joseph E. Gonzalez

Professor Joseph Gonzalez
Research Advisor

5/11/22

(Date)

* * * * * * *

Ion Stoica

Professor Ion Stoica
Second Reader

5/12/22

(Date)

Bridging the Gap Between Modular and End-to-end Autonomous Driving Systems

by

Eric Leong

A thesis submitted in partial satisfaction of the

requirements for the degree of

Masters of Science

in

Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Joseph Gonzalez, Chair
Professor Ion Stoica

Spring 2022

Abstract

Bridging the Gap Between Modular and End-to-end Autonomous Driving Systems

by

Eric Leong

Masters of Science in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Joseph Gonzalez, Chair

We aim to bridge the gap between end-to-end learning and traditional pipeline-based approaches for autonomous vehicles (AVs). In this work, we replace the traditional planning and control algorithms of modular approaches with an end-to-end learned policy, developing a hybrid of the two approaches. Our learned policy takes a bird's eye view representation of the world as input, and produces control actions such as braking, steering, and acceleration. To support the development of this learned policy, we introduce caRLot, a novel OpenAI gym environment that builds atop the open-source Pylot AV platform to provide configurable abstractions in addition to an interface with the CARLA simulator. We use caRLot to learn a model-free reinforcement learning policy that replaces planning and control, and compare its performance and runtime against several state-of-the-art approaches. We find that our hybrid approach has a notable improvement in runtime over a modular driving system, while having a significant advantage in interpretability over end-to-end systems.

To my family.

Contents

List of Figures

List of Tables

Acknowledgments

This thesis would not have been possible without those on the ERDOS team in the RISELab. I will always be grateful to my advisor, Professor Joseph Gonzalez, for the opportunity to work in the ERDOS group and advising me for the past two years. He has provided an abundance of guidance and resources for not only research but also for my academic career. I am also grateful to Professor Ion Stoica for acting as the second reader for this thesis. I'd like to thank Peter Schafhalter for laying the groundwork for this project and providing phenomenal direction throughout the course of it. Furthermore, I am incredibly grateful to him and Sukrit Kalra for mentoring me when I first joined the group and had no clue what AV research involved. I'd like to thank them for their patience, knowledge and support as I slowly integrated into the group and carried out my first projects and their continued guidance from then on. I'd also like to thank Eyal Sela for taking me under his wing as a mentor: patiently supporting my efforts to contribute to projects and always making time to meet whenever anything came up for further discussion. Thank you also to Ionel Gog and Justin Wong for the wonderful collaborations.

I would like to thank all the people who have made me feel eminently supported and fulfilled throughout my five years at Berkeley. I am extremely grateful to my fellow Fifth year MS peers, Zaynah Javed and Ashwin Rastogi, for enduring this final year with me. It cannot be understated how awesome it was to have others to share this unique experience with. I'd also like to thank those in the organization, SEKZ A. Finally, I'd like to thank my mom, dad, and sister. My extended journey at Berkeley would not have been possible without their unconditional love and support.

Chapter 1

Introduction

Traditional approaches to autonomous vehicles (AVs) rely on multi-stage, human-engineered pipelines [25, 24, 57, 3, 4]. Such pipelines provide several important benefits, such as the ability to test, verify, and develop AV software on a per-component basis. As AVs become an impending reality, and governments seek to ensure the safety of such vehicles on the road, this modular approach provides further benefits in the ability to audit and explain an AV's decisions, which becomes especially critical in the case of crashes [57].

Despite the benefits of the engineered approach, advances in deep learning have sparked interest in new approaches that blur the lines between components by propagating gradients and sharing neural network architectures across several tasks. For example, Tesla's Full Self-Driving architecture uses neural networks with shared backbones to complete a variety of perception tasks [36], and Wayve applies a pure machine-learning driven approach to AVs [74].

Such approaches place faith in machine learning to provide better performance in AVs for the following reasons: (*i*) machine learning can discover features that enable safer, more comfortable, and more generalizable autonomous driving than the hand-engineered features used by traditional components in traditional pipelines, (*ii*) learned approaches to driving can scale and improve with data, as opposed to pipeline-based approaches which may require engineers to hand-encode traffic rules and vehicle behaviors for different countries, and (*iii*) a move towards end-to-end learning in AVs may result in better performance and smaller, less complex systems because machine learning can be used to optimize overall system performance [7].

Although the benefits seem promising, the large scale of data required and the difficulty of ensuring reliable behavior have frustrated learned approaches to AVs. In this work, we seek to bridge the gap between modular approaches and end-to-end approaches by developing caRLot, a hybrid between the two approaches. caRLot builds a novel OpenAI gym environment [8] atop the open-source Pylot modular AV platform [27], providing a wide variety of configurable abstractions over both sensors and AV components. We replace two key components of modular pipelines, planning, and control, with an end-to-end solution. Specifically, we use deep reinforcement learning to train a model-free policy that takes an image-like representation of the vehicle's surroundings and directly produce control outputs (e.g., braking, steering, and acceleration). We compare our policy to existing approaches involving end-to-end learning and a traditional pipeline by evaluating on NoCrash benchmark [19] which is run using the CARLA simulator [22], and by comparing overall runtimes.

The rest of this thesis is organized as follows. Chapter 2 provides an overview of autonomous driving systems, discussing both modular and end-to-end approaches in detail while examining relevant works. In Chapter 3, we introduce our approach caRLot, describing important details about our integration with the Pylot AV platform [27] and outlining our environment setup, reward design, and policy. Chapter 4 discusses our experiments related to training and runtime analysis. Finally, Chapter 5 discusses the future directions of research for our work.

Chapter 2

Overview of Autonomous Driving Systems

There are currently two primary paradigms that autonomous driving systems typically follow: end-to-end and modular driving [79]. An end-to-end system aims to learn a direct mapping from the raw sensory input to the vehicle control signals. In contrast, a modular system decomposes the driving task into separate modules or sub-tasks that ultimately link the sensor input to the control signal. There are typically five core modules: sensing, perception, prediction, planning, and control.

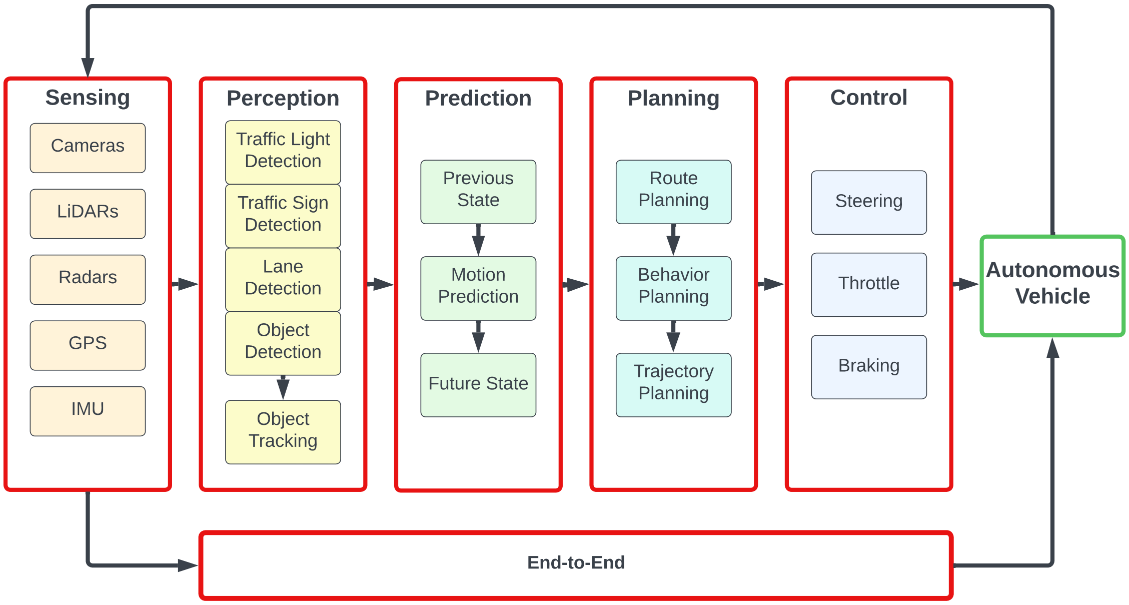


Figure 2.1: ADS Pipeline

In this section, we will first discuss the modular design paradigm, discussing each of the modules in detail. We will then provide an overview of approaches to end-to-end learning. Going over the two approaches in detail will be crucial for understanding our approach to bridging the gap between end-to-end and modular systems.

2.1 Modular Approach

By developing individual modules separately, modular systems divide the challenging task of autonomous driving into an easier-to-solve set of problems [79]. Furthermore, each module is commonly divided into smaller components. Dividing the autonomous task into modules and components with clearly defined intermediate representations and interdependencies contributes to interpretability as one can easily determine the source of error in case of unexpected behavior [70]. Dividing into smaller tasks also allows for ease of transferring knowledge from corresponding literature in computer vision, robotics, and other related fields [34]. Another major advantage of modular systems is that algorithms can be integrated and built upon each other in a modular design, such that changes in a module would not require a revamp in the implementation of other modules [27].

However, modular systems bear several disadvantages. Namely, modular systems face major challenges with generalizing to different scenarios to achieve robustness [70, 34]. The modules are human-engineered with predefined input and output representations and rely on heuristics or intuitions that may not be accurate for all scenarios. For instance, the object detection component in perception outputs just a bounding box and class type for a detected object. This removes a lot of context from the scene that would've been useful for downstream modules for decision-making. Other major challenges in modular systems include over-complexity, dependence on supervised data, and error propagation [79, 34]. Modular systems are often overly-complex, having many models implemented for each module and using many intermediate representations and outputs. This has a growing impact on the overall run time of the system. Furthermore, a lot of these models, especially those in perception, are dependent on supervised data for training, creating a human-defined information bottleneck [79].

We discuss each of the five main modules and the techniques used and challenges faced in their components.

Sensing

In the sensing module, the ADS employs a variety of sensor technologies, including cameras, LiDARs, Radars, GPS, and IMUs. These sensors provide the ego-vehicle with high-resolution information about the color, texture, and depth of objects in their surroundings, as well as important information necessary for localization and mapping [79]. The vast amount of data from the sensors is utilized by downstream operators to perceive the driving environment precisely, allowing the ego-vehicle to plan and make critical decisions accordingly.

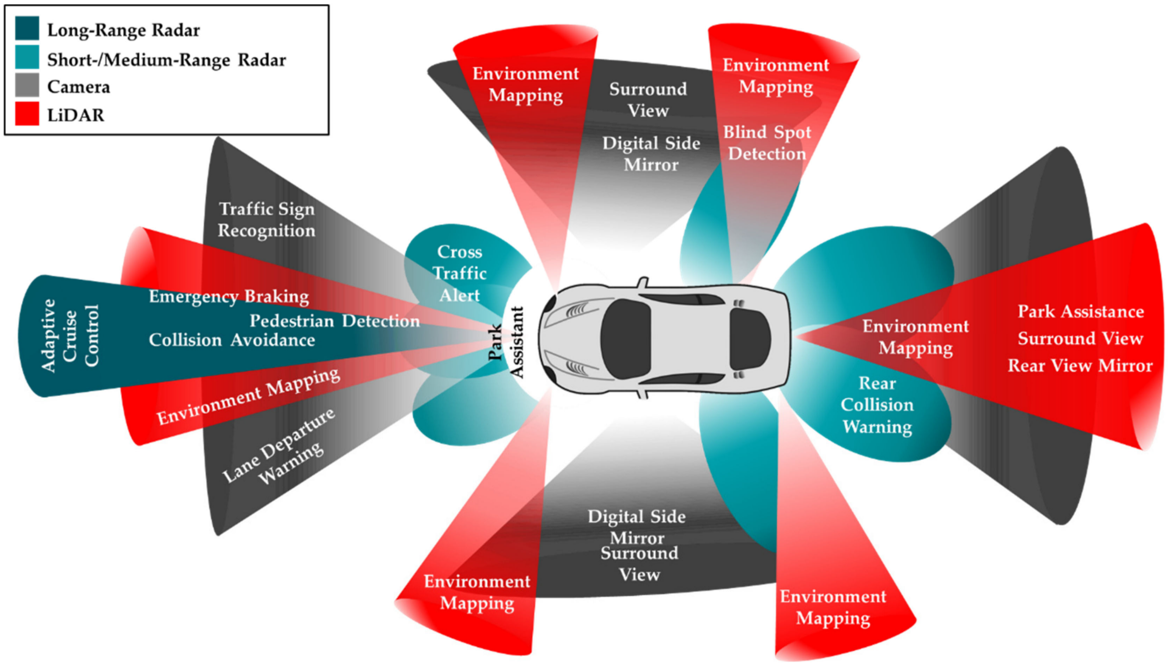


Figure 2.2: Example Sensor Configuration in an autonomous vehicle. Adapted from Sensor and Sensor Fusion Technology in Autonomous Vehicles: A Review [9].

Since each sensor type has different strengths and weaknesses, autonomous vehicles typically employ multiples of each type of sensor to improve the efficiency and reliability of the ADS. For more details, we refer the reader to [9] which provides a detailed overview of sensor technologies and sensor fusion techniques to integrate data from multiple sensing modalities.

Perception

The perception module processes data from various sensors to form a meaningful representation of the driving environment, often using computer vision techniques. With the advent of deep learning and its growing application to computer vision tasks such as object detection and segmentation, perception in autonomous vehicles has become one of the most actively researched AI fields [33, 34]. Though cameras are the primary sensors used for perception as it builds upon decades of computer vision research, 3D vision using the depth data provided by LiDARs and Radars has proven to be a powerful supplement to perception [29, 62, 77]. However, in this work, we will focus on perception restricted to just camera images. The perception module typically includes object detection, object tracking, and lane detection as components [79]. We will briefly cover these components and discuss some of the main challenges in perception.

- **Object detection** refers to the task of identifying and locating objects of interest, given a single camera image. Objects of interest include static objects, such as traffic lights and traffic signs, as well as dynamic objects, which includes other vehicles and pedestrians. Object detection is one of the most critical components of the general autonomous driving pipeline due to its potential impact on safety. For instance, a failed vehicle detection will propagate to downstream modules and will essentially be unknown to the planning and control components. The planning and control components will make decisions without accounting for the vehicle, potentially leading to a crash.

Current state-of-the-art object detection models typically fall within two types: one-stage detectors and two-stage detectors [68]. Two-stage detectors, such as Faster-RCNN [64] and Mask R-CNN [31], first utilize a region proposal network (RPN) to generate regions of interest before sending them to the classification and bounding-box regression stage. In contrast, one-stage detectors, including [63, 47, 45, 71], lack a region proposal stage, performing regression directly over a regular, dense sampling of locations in the image. By virtue of not using an RPN, single-stage detectors trade-off accuracy for much lower computational costs. This introduces the notion of the accuracy-latency trade-off in the perception module, which we will later discuss.

- **Object Tracking** and multi-object tracking (MOT) tracks multiple objects as they move around frames in a sequence or video. For each object, the object tracker maintains a consistent identifier paired with the bounding box for each frame throughout a sequence. Using the identifier, the output of the MOT can be accumulated over time to represent the trajectory of each object and be used for trajectory prediction.

  Traditional approaches and deep learning approaches are both commonly used. SORT is a classical object tracker that uses traditional algorithms such as the Kalman Filter and Hungarian algorithm to respectively maintain the bounding boxes and IDs [6]. Deep learning approaches such as [48, 60] achieve state-of-the-art performances on many tracking benchmarks [50, 78, 69, 20]. Although deep learning methods generally have significantly better performance than traditional approaches, they also tend to have much higher latency [34].

- **Lane Detection** is the task of detecting lanes and lane segments on the road, given a camera image. Lane detection is crucial for understanding the road semantics to properly navigate the road. Given the large variety of road topology, lane marking types, and complex semantics such as lane direction, lane detection continues to be a very challenging task in perception [79].

  Similar to object tracking, lane detection methods can be classified into two categories: traditional methods and deep learning-based methods. Traditional methods often used primitive information such as color features, texture, and gradients to perform geometric modeling i.e. line detection and fitting, or to employ various energy minimization algorithms [82]. A common issue with traditional approaches is that they are not robust to road scene variations [46]. Deep learning-based models include segmentation approaches, encoder-decoder models, and CNN-RNN hybrid models. For a comprehensive overview of deep learning approaches to lane detection, we refer the reader to [81, 46].

A common characteristic of the perception components is that their state-of-the-art models focus on having either low latency or high accuracy. For instance, the SORT tracker is still often used despite its poor performance compared to deep learning approaches because of its very low latency. This dilemma introduces the phenomenon of the accuracy-latency trade-off, in which complex, highly parameterized deep learning models typically have higher accuracy and worse latency than more traditional, lightweight models. Although it is pivotal for the ego-vehicle's perception system to have high accuracy, having low latency in time-critical settings like self-driving is equally as important.

The accuracy-latency trade-off in perception can be interpreted as weighing the costs between having a missed detection and having a slower reaction time, both of which will have an impact on the overall performance of the autonomous driving system. Errors in the perception module will be propagated to downstream components and can result in suboptimal behaviors that may lead to collisions. Similarly, a slow reaction time detracts from the goal of having real-time control. Self-driving has critical deadlines for making decisions, and if not consistently met, can be extremely costly depending on the driving context. For instance, according to [37], if the ego-vehicle is driving at 40km per hour in an urban area and needs the control signal ready every 1 meter, then the entire pipeline's desired response time should be less than 90ms. Although a more extreme case, this example illustrates how strict runtime requirements limit the types of perception models that could be used in realistic scenarios.

Prediction

The prediction module uses the environment representation from the perception module to evaluate the behaviors of surrounding vehicles and pedestrians and assess the risks of the driving scene [79]. To assess risk, the prediction module tries to predict the future state of the surrounding drivers and pedestrians in a process called motion trajectory prediction. Object tracking plays a pivotal role in prediction by keeping track of each agent's previous states for a certain horizon, which can be used to estimate the velocity over time and lane localization. In addition to the information provided by previous tracks, scene information such as from 3D point clouds from LiDAR sensors and a bird's eye view of the environment are used as input [52]. Using the previous `prediction_num_past_steps` states of the agent, the prediction module will predict its next `prediction_num_future_steps` states.

The linear regression model serves as a baseline for prediction as it assumes that each agent travels at a constant velocity, ignoring the scene context and driving behaviors. Similar to the SORT tracker, linear prediction has poor performance compared to deep learning models but is much more efficient. For a detailed overview of deep learning approaches, we refer the reader to [52].

Prediction is a challenging but critical task in self-driving, as it provides essential information for the ego-vehicle to act proactively, such as to change lanes or drive through intersections. However, the task becomes especially difficult when there are multiple dynamic agents in the scene that must be monitored. The parameters `prediction_num_past_steps` and `prediction_num_future_steps` have even more impact on the model latency, as increasing them will exponentially increase the amount of data tracked or predicted.

Planning

The planning module produces a trajectory of waypoints that the ego-vehicle should follow, using the predicted trajectories of other agents from the prediction module along with information from the localization and mapping component from the perception module. It is typically decomposed into three sequential components: route planning, behavior planning, and motion planning.

Route planning is first in the sequence. Given a starting and goal location, the route planner treats the road network as a directed graph and produces a high-level route to the destination. The road network can be interpreted as a graph, for example, by representing the intersections as vertices and the roads as edges with weights based on distance. Given the representation of road networks as graphs, graph traversal and path searching algorithms such as A* search [56] and Dijkstra's algorithm [21] are commonly used to find the shortest path to the goal location [79].

Behavior planning is the second in the sequence. Using the route plan, the behavior planner outputs high-level behavior that the ego-vehicle should follow. Commonly defined high-level behavior includes staying in the same lane, preparing for a lane change, or following waypoints. Because of the uncertainty of the driving scene, approaches to behavior planning commonly use probabilistic frameworks such as Markov decision processes (MDP) and partially-observable MDPs (POMDPs) [58].

Finally, motion planning, also referred to as trajectory planning, uses the selected high-level behavior to produce a safe, comfortable, and dynamically feasible trajectory towards the goal configuration. Popular approaches include graph search methods [56] and incremental search methods [35]. Graph search methods construct a graphical construction of the ego-vehicle's configuration space and search for the shortest path using graph search methods, while incremental search approaches incrementally construct a tree of reachable states from the initial state of the vehicle and then select the best branch of such a tree. [58] provides a comprehensive overview of motion planning methods.

Control

The control module is responsible for accurately tracking the trajectory generated by the planning module, doing so by adjusting the steering, throttle, and braking of the ego-vehicle. It plays a crucial role in the overall ADS as it is responsible for actually generating the commands that drive the vehicle.

Traditional control methods are the most widely used control strategy and include the PID (proportional-integral-derivative) controller [1] and MPC (model-predictive-control) [39]. Though such traditional approaches are very effective, they often require frequent re-tuning since they do not generalize well to varying operating conditions [66]. Implementation and retuning of these controllers also require system-level knowledge. Furthermore, advanced traditional control methods, such as MPC, are often computationally expensive.

Because of these issues, one may opt for learning-based control methods, such as end-to-end systems, or hybrid control approaches. We will discuss end-to-end systems in more depth in the next section. We refer the reader to [66] for a more detailed overview of traditional control mechanisms and hybrid strategies.

2.2 End-to-End Approach

We now focus on end-to-end approaches to control in autonomous driving systems. Rather than decomposing the driving task into modules, end-to-end strategies find a direct mapping from the sensory input to control commands. End-to-end systems are much less complex than modular systems, usually having simpler models and fewer intermediate representations [70]. However, this is problematic for the model's interpretability, as it becomes more difficult to ascertain why the model behaves abnormally. This is in stark contrast to modular systems where it is simple to determine which module in the pipeline erred.

End-to-end approaches can be broadly classified into two categories: imitation learning and reinforcement learning. We will cover each category in more detail next.

Imitation Learning

Imitation learning is the process of learning and developing new skills by observing these skills performed by another agent. This is often performed through behavioral cloning, a supervised learning strategy in which an agent is trained using labeled data gathered from an expert agent, essentially aiming to "clone" their behavior. In the context of self-driving, an agent learns from recorded expert trajectories containing sensory data and control commands.

End-to-end learning through imitation learning was one of the earliest approaches to self-driving. It was first pioneered in the 1990s by the ALVINN system [61]. ALVINN used a simple 3-layer network and demonstrated success in the road following task. It served as a precursor to DAVE [53] and subsequently DAVE-2 [7], which used simple CNN architectures to map visual input to steering commands. Through the DAVE-2 system, Bojarski et al. demonstrated that CNNs can learn the entire task of lane and road following without decomposition into road or lane marking detection, semantic abstraction, path planning, and control [7].

Though imitation learning poses advantages such as ease of implementation and relatively quick training processes due to supervised learning, it struggles to achieve the reliability and robustness necessary for real-world driving [18]. The amount of expert data that is necessary to achieve reliability is too costly and time-consuming to collect. Furthermore, it is difficult to gather data from demonstrations in dangerous scenarios because of safety concerns. As a result, the collection of states that the expert encounters does not usually cover all the potential states that the trained agent may encounter during testing. Imitative agents will often struggle from covariate shifts when the online behavior begins to deviate from the offline training experience. For example, the imitative agent will struggle to learn driving skills to handle dangerous situations, due to a lack of sample experiences. One approach to alleviating covariate shifts is Data Aggregation (DAgger) [65]. DAgger continuously updates the state distribution by collecting trajectories from the current policy, relabelling them using the expert policy, and aggregating them to the original dataset. However, this demonstrates that an agent would require frequent human intervention at test-time when encountering complex scenarios, such as crossing an intersection or reacting to multiple dynamic objects.

More recent works focus on designing robust policies that can drive in complex urban scenarios. Rather than imitating a human driver, Pan et al. [59] imitated a model predictive controller to perform high-speed driving. Bansal et al. [5] present a mid-level input representation of the environment to reduce sample complexity and use this input to predict a driving trajectory which is then converted into control commands. Codevilla et al. [17] use conditional branching based on high-level commands, similar to the commands in behavior planning, extending imitative models for urban driving. By conditioning on commands, the policy could handle sensorimotor coordination while responding to navigational commands. Zeng et al. [80] use imitation learning to train a cost volume predictor that determines the quality of possible locations on the planning horizon. Chen et al. [11] train an imitative policy using learned affordances to improve driving scene understanding.

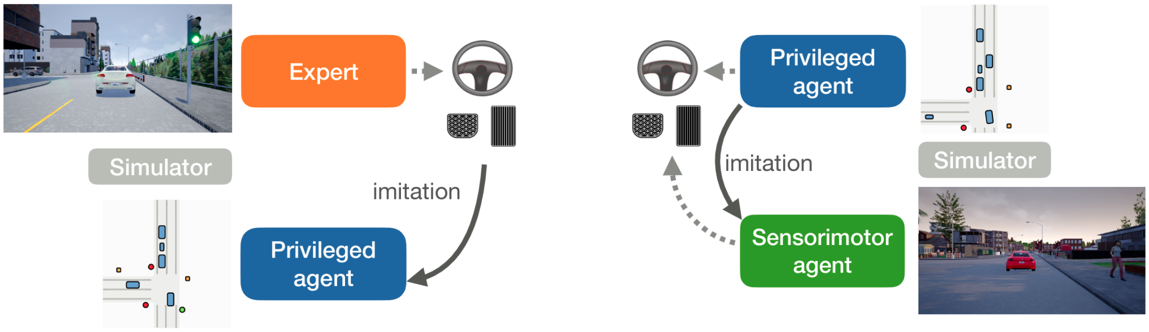


Figure 2.3: Overview of the two-step process in Learning by Cheating [14]. (Left) A privileged agent with direct access to the environment state learns a robust policy by imitating the expert. (Right) A sensorimotor agent without access to privileged information learns the task by imitating the trained privileged agent.

Learning by Cheating uses a two-step process to achieve state-of-the-art performance on the CARLA benchmark [14]. Chen et al. first train a privileged agent to learn to predict future waypoints for the vehicle to steer towards, using ground truth information from the simulator. It then uses the privileged agent to teach a purely vision-based sensorimotor agent in offline settings to accomplish the same task. For both agents, a lightweight keypoint detection architecture is used to predict waypoints. Additional on-policy training is performed to control for distribution shifts and provide stronger imitative supervision signals. Finally, to translate waypoints into control commands, a low-level PID controller is used.

Reinforcement Learning

In reinforcement learning (RL), an autonomous agent learns to perform optimal actions for an assigned task by repeatedly interacting with the environment, essentially learning through trial and error. Unlike imitation learning, no expert is telling the RL agent how to act. Instead of gaining insights about the policy's performance in a supervised manner by comparing to an expert, performance in RL is dictated by a reward function prescribed by the environment.

RL problems are formally described through Markov decision processes (MDPs) [76]. The standard discounted, finite-horizon MDP consists of a set of states S, a set of actions A, transition function $P(s_{t+1}|a_t, s_t)$, reward function $R(s_t, a_t)$, discount factor γ, and horizon T. At each step t, the agent will use information about the current environment state s_t to produce action a_t, which will be enacted on the environment. After taking the action, the agent will receive a reward $r_t = R(s_t, a_t)$ from the environment along with the updated state space s_{t+1}. The reward r_t informs the agent on the quality of the previous state-action pair (s_t, a_t), playing a critical role in learning the optimal task. The objective of the RL agent is to learn a policy that achieves the maximum expected sum of rewards over trajectories in the state space:

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\pi}[\sum_{t=0}^{T} \gamma^t r_t] \tag{2.1}$$

In the context of real-world applications such as autonomous driving, an agent cannot observe all features of the environment's state. Thus, the RL problem for our task will be formulated as a partially-observed MDP (POMDP), introducing the notion of an observation space O and emission probability $P(o_{t+1}|s_t + 1, a_t)$. The agent uses observations o_t from the environment, instead of the state s_t, to compute an action a_t.

The main techniques for solving RL problems include value function based methods, policy search, and hybrid methods. Kiran et al. [2] provide a detailed overview of various techniques, such as Q-learning (value-based), the REINFORCE algorithm (policy search), and the actor-critic algorithm (hybrid).

Similar to other areas, the advent of deep learning has accelerated progress in reinforcement learning. The use of deep neural networks in RL to develop optimal policies defines the field of deep reinforcement learning (DRL). By using deep neural networks, DRL can take advantage of the important properties of deep neural networks. Properties such as the ability to scale to settings with high-dimensional data through representation learning and the ability to act as universal function approximators are most pertinent to RL [2, 32]. DRL has made a number of breakthroughs in recent years. For instance, DRL agents achieved superhuman level performance in games such as Atari [51] and Go [67] which notably have high-dimensional sensory inputs or complex action spaces. From these successes, DRL algorithms have been applied to a wide range of problems, such as robotics, video games, and systems capable of learning to adapt to the real world. DRL for autonomous driving has recently become an emergent field. For a detailed overview of DRL algorithms and research challenges in the field, see [76, 2].

Deep RL for Autonomous Driving

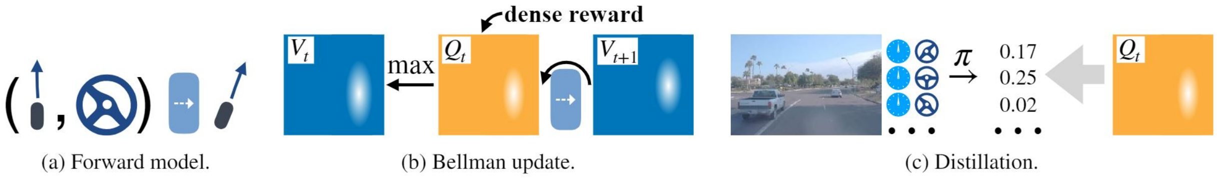
As previously mentioned, imitative models struggle with robustness and reliability due to their dependency on large amounts of expert data. We are motivated to use RL over imitative learning for learning driving policies because unsupervised learning removes the dependency on expert data. We can also learn in a simulated driving environment and learn through trial and error [22]. This is especially important for learning to drive in a complex urban driving environment as we can simulate a large variety of scenarios, including dangerous ones. Thus, the RL agent is more robust than imitative models by nature of having more exposure to many different driving scenarios [16]. Given the high-dimensional environment and complex action space in autonomous driving, we are also motivated to use DRL over traditional RL for learning complex driving policies.

DRL for self-driving was first introduced in Learning to Drive in a Day [38]. When using RL for the autonomous driving task, the MDP formulation of the state space S, observation space O, action space A, and reward function R can be freely set. Kendall et al. used monocular camera images to define the observation space O, steering angle continuous between [-1, 1] to define the action space, and a reward function based on minimizing the distance from the center of the lane. They took a model-free RL approach, using the Deep Deterministic Policy Gradients (DDPG) algorithm. Though the results demonstrated the first successful application of DRL for self-driving, their model only handled steering angle for the lane keeping task which does not generalize to the urban driving scenario.

One of the major limitations in model-free RL is sample inefficiency: requiring an enormous amount of data to learn expected behaviors and training is very expensive [49, 28]. Even relatively simple tasks can require millions of training steps. Combined with the high sample complexity in vision-based autonomous driving, urban driving is a daunting task to learn using model-free RL, especially if the original sensor inputs are used. To reduce sample complexity, the sensor inputs can be encoded in a more compact and semantically rich representation that the DRL network will be trained on.

Chen et al. [16] follow this idea and, similar to the previous approach, provide a model-free RL approach to the autonomous driving task. However, their method was able to learn a driving policy robust enough for an urban driving task: driving through a roundabout with many vehicles and pedestrians. Instead of directly using front-view camera images during training, they designed a new representation for the observations, using a bird's-eye-view representation combining multiple perception information from the scene. This served to reduce the sample complexity of observations substantially, especially when compared to raw sensor observations. To further reduce sample complexity, they learned a low-dimensional latent representation from the bird's-eye-view images. Finally, the latent encoding is then used as observations for the RL model. The paper uses off-policy, model-free RL algorithms, such as DDQN [51, 30] and SAC [28]. Off-policy learning, compared to on-policy learning, does not require new samples to be collected for each gradient step and instead aims to reuse past experiences [28]. As a result, off-policy algorithms are more sample-efficient. Though the work demonstrated some success in a specific urban scenario, it struggled to handle crash scenarios and ignored traffic lights.

Toromanoff et al. [72] also develop a latent representation, introducing Implicit Affordances. In contrast to the previous method, the representation provided a sufficient signal to the RL agent for traffic light detection in urban driving. In their approach, they used an end-to-end pipeline composed of two subsystems trained successively. First, a convolutional encoder is trained on auxiliary tasks, such as semantic segmentation and traffic light state detection. The encoder is subsequently frozen and the DRL subsystem is trained using the encoder's latent space. A novel reward design was also introduced, which we will discuss later. Overall, their approach was highly successful and achieved strong performance on the CARLA leaderboard.



(a) Forward model. (b) Bellman update. (c) Distillation.

Figure 2.4: Overview of World on Rails [12]. (a) A forward model is learned using pre-recorded logs. (b) Under the learned forward model, the Bellman equations are used to compute the Q-function. (c) The Q-function supervises the visuomotor driving policy through policy distillation.

More recently, Chen et al. [12] developed a vision-based driving policy for the urban driving task that achieves state-of-the-art performance in difficult benchmarks and the CARLA leaderboard. Unlike the previously discussed approaches, they followed a model-based RL approach, generating a model of the world from pre-recorded logs, and learning a policy that acts upon this model. By learning a model for the environment dynamics, model-based approaches reduce the number of costly interactions required with the real environment, improving sample efficiency [40]. The work makes a further simplifying assumption that neither the agent nor its actions influence the environment. Though this assumption clearly does not hold in the simulator and the real world, it simplified the learning process significantly, allowing for a simplified tabular RL setup. Once the forward model is learned, the action-value function (Q-function) is estimated using dynamic programming and backward induction over a simplified form of the Bellman equations. Finally, we use the tabular approximation of the Q-function to supervise a visuomotor policy through policy distillation.

Reward Design for RL in Autonomous Driving

The design of the reward function is crucial to the performance of any RL policy: RL agents seek to maximize the return from the reward function, therefore the optimal policy is defined for the reward function. However, real-world applications such as self-driving often have sparse and/or delayed rewards. This is problematic because RL agents learn behaviors through the reward signal. For instance, the reward function for the traditional self-driving task of following a route and reaching the goal destination safely is typically defined by a large reward when reaching the goal destination and a penalty when crashing [40]. Both of these states are extremely sparse, and they provide barely any signal to the agent on proper driving behavior. Additional rewards can be provided to the agent, through reward shaping. Reward shaping allows a reward function to be engineered in a way to provide more frequent feedback signals on appropriate behaviors [41]. Examples of reward shaping in autonomous driving include having a penalty for deviating from the center of the lane, for deviating from steering straight, or for getting close to other vehicles.

Even with reward shaping, designing a proper reward function in autonomous driving is challenging. Knox et al. [41] provide a detailed overview of challenges in reward design. An important mention is that reward design is challenging because driving is a multi-attributed problem. Attributes such as progress to the destination, time spent driving, collisions, obeying the law, fuel consumption, etc. all contribute to the quality of driving. It is difficult to enumerate and reward all types of behavior, and also consider their external impacts. Knox et al. also provide a survey of different reward designs for the self-driving task and sanity checks for reward shaping. They mentioned that rewarding behavior that is correlated with performance can backfire. Strongly optimizing such reward can result in policies that prioritize the shaped reward over other performance-related outcomes, which can drive down overall performance.

When designing our reward function in 3.5, we take inspiration from the reward designs in World On Rails [12] and End-to-End Model-Free Reinforcement Learning for Urban Driving using Implicit Affordances [72]. In World On Rails, the agent receives a +1 reward for staying in the target lane at the desired position, orientation, and speed, with a smooth penalty down to a value of 0 for deviations. When at a “zero-speed” region, such as a red light, stop sign, or nearby other agents, the agent is rewarded for having zero velocity or braking. There is no explicit penalty for collisions. Toromanoff et al., on the other hand, define a reward function based on three main components: desired speed, desired position, and desired rotation. The agent is rewarded +1 when at the desired speed, linearly going down to 0 if the agent’s speed is above or below. The desired speed is adaptive to the situation, for example, going to 0 in red-light zones or nearby another agent. It is computed

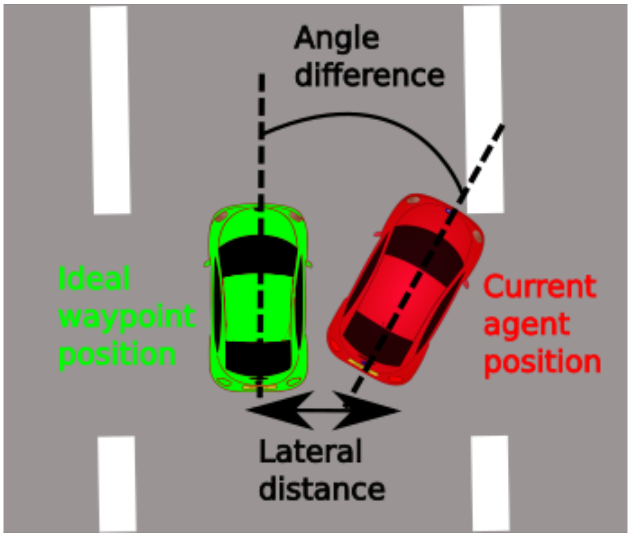


Figure 2.5: Lateral distance and angle distance computation for the reward function proposed by Toromanoff et al. [72].

using "privileged" information from the simulator. The desired position is based on lateral distance from the center of the lane. Finally, the desired rotation is inversely proportional to the angle difference between the agent and the nearest waypoint from the route. Figure 2.5 illustrates how lateral distance and difference in angle are computed.

Chapter 3

caRLot

We aim to build an autonomous driving system that bridges the gap between modular systems and end-to-end systems, addressing the major challenges that each archetype faces, while retaining some of its key features. As discussed in Section 2.1, modular approaches struggle with high-complexity and robustness but are highly interpretable and components easily interchangeable. End-to-end approaches, as discussed in Section 2.2, sacrifice interpretability for simplicity. Furthermore, reinforcement learning, unlike modular approaches and imitation learning, does not have a human-defined information bottleneck as it can learn unsupervised in a simulator through trial-and-error. By training in a simulator, the driving model is also more robust as the agent can experience dangerous situations in simulation. However, as we discussed in Section 2.2, a major challenge in RL, specifically DRL, is poor sample efficiency. When combined with the high sample complexity in autonomous driving, training becomes very expensive.

In this section, we will first discuss Pylot [27], a modular AV platform interfacing with the CARLA simulator [22] that forms the foundations of our approach. We will then build upon what we've explored in previous sections and discuss our proposed autonomous driving system, which we call caRLot.

3.1 Pylot

Pylot is an open-source autonomous driving platform for developing and testing AV components, providing a modular ADS implementation. Modularity in Pylot is achieved by building on top of ERDOS [26], a high-throughput, low latency dataflow system. Under a dataflow system, the AV pipeline is structured as a directed graph where AV components are represented as vertices (ERDOS operators) connected through edges (ERDOS streams). Using the streams, operators can communicate with each other by either reading or writing timestamped messages. Since operators can read from or write to multiple data streams, ERDOS uses a system of watermarks to ensure that data across multiple streams is processed synchronously at a particular timestamp.

Figure 3.1 summarizes the modules in Pylot's AV pipeline, the components within each module, and how they are all interconnected. Through a modular structure, Pylot achieves the benefits of modular systems that we describe in Section 2.1, such as interpretability and ease of integration. Thus, we can use Pylot to study the effects of changing individual components on end-to-end driving behavior, without having to worry about runtime variabilities or interactions between components. We can also easily visualize the output of individual components, using the visualization tools that Pylot offers. To support the development process, Pylot provides reference implementations for components. Moreover, Pylot interfaces with CARLA [22], an urban driving simulator that provides privileged information about the driving environment. When interfacing with CARLA, Pylot offers "perfect" implementations for certain components, using information from the simulator to produce the ground truth for a particular component. This enables us to more accurately evaluate components in isolation and determine the individual impact on end-to-end performance. To take advantage of this feature, we develop our approach and run experiments over the CARLA simulator.

CARLA

Because our approach will be developed over Pylot's interface with the CARLA simulator, it is important to discuss some of CARLA's abstractions when simulating the real-world driving environment. As previously discussed, CARLA offers an abundance of privileged information about the simulation environment, sufficient for producing ground-truth implementations of most of the perception module. Though privileged information is useful for testing purposes, over-relying on it may make it to transfer our system to the real world in the future [40]. We describe some of the privileged information relevant to our approach while considering the "simulator-reality gap".

While driving in the real world, the vehicle's pose is necessary to determine how to follow a specific route towards the destination; the simulator is no different. In Pylot, the vehicle's pose is defined by its position, orientation, forward speed, and velocity. Using the CARLA simulator, Pylot operators can access the ground truth pose of the vehicle at each instance. This is in stark contrast to the real world, where vehicles would use GPS and IMU sensors for localization components to estimate pose. Though Pylot supports localization, we decided to use the simulator's computed pose to eliminate a potential source of error.

To navigate from a start location to an end location, CARLA uses a high-level route planner to generate high-level navigational commands: "turn left", "turn right", "go straight", "follow lane", "change lane left", and "change lane right". These commands are similar to those provided by navigation tools that guide human drivers along a route. We use these commands in our approach, following previous works that use the high-level command to guide the vehicle along reproducible routes and reduce ambiguity at intersections [17, 14, 12].

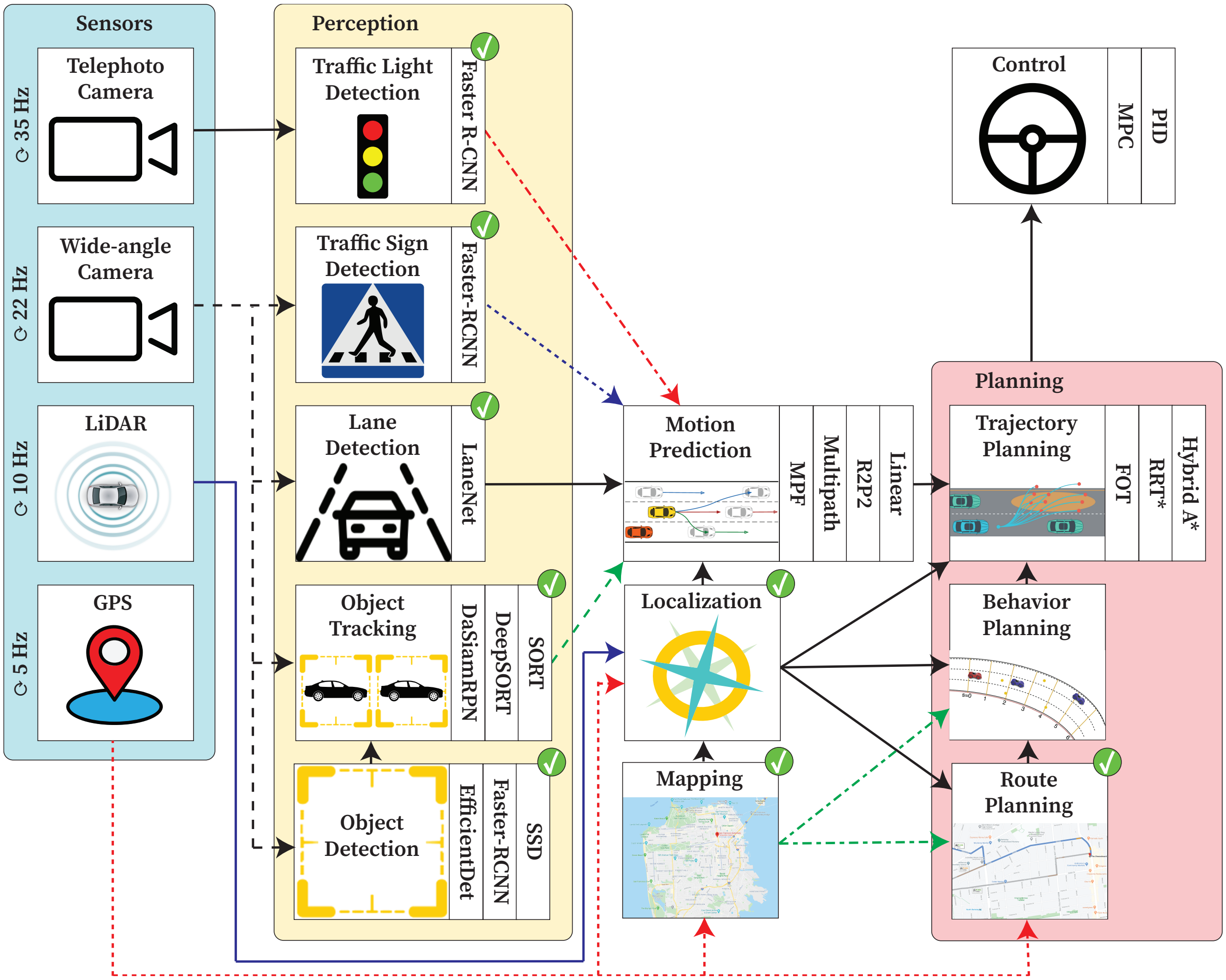


Figure 3.1: Modules and components in the Pylot AV pipeline [27]. Reference implementations are listed next to the component while perfect implementations are provided for components with a green check mark.

Perception Stack

We denote the sensing, perception, and prediction modules as the perception stack as they are responsible for perceiving the surrounding environment and extracting the information necessary for the planning and control modules to ensure safe navigation.

For the purposes of this paper, we focus on the following subset of sensor technologies supported in Pylot: cameras and GPS. In Pylot, cameras can easily be integrated into the vehicle by defining a `CameraSetup` for each camera. The `CameraSetup` is defined by the camera's image width, image height, and transform with respect to the vehicle, field of view, and camera type. CARLA supports RGB cameras, depth cameras, and segmentation cameras. To attach a camera to the ego-vehicle, the CarlaCameraDriverOperator is created with the defined CameraSetup. The operator will receive camera frames from the CARLA simulator and send them on an output stream as a FrameMessage. For visualization purposes, the camera frame can be rendered as a 2D RGB image. Though the transform configuration in CameraSetup is generally used to transform the camera to a location and orientation relative to the vehicle, Pylot supports bird's eye view transforms to transform the camera's view to a top-down perspective.

Similar to the camera, a GPS can be attached at a specific location and rotation relative to the vehicle. As we've discussed, the GPS sensor plays an important role in mapping and localization. Since we will be using the privileged pose information from the simulator, localization with GPS (and IMU) will not be necessary. However, the GPS is still necessary for determining which high-level command to follow, while the ego-vehicle is progressing through the computed route.

In Pylot, nearly all components in the perception module are dependent on just camera sensors, which contributes to our decision to limit sensor types to cameras and GPS. For our pipeline, we will utilize all of the perception components because each plays a critical role in achieving urban driving. The perception components, traffic light detection, lane detection, object detection, and object tracking have individual Pylot operators for reference implementation and "perfect" implementation. Traffic sign detection is integrated into object detection: traffic signs and stop signs will be detected as obstacles and labeled appropriately for downstream operators. Perception operators will process camera data from the input camera stream to compute their results, which they communicate through their corresponding message type. For instance, object detection operators transmit detections through an ObstacleMessage, containing the information for a particular obstacle's ID, bounding box, and class label. The traffic light detection operator provides nearly the same information as the object detector in a TrafficLightsMessage but provides a traffic light state (green, yellow, red, off) instead of a class label. Lane detection operators send a LaneMessage, containing the detected lanes which are each represented as a sequence of lane markings for both sides of the lane. Object tracking operators utilize the detections from the ObstacleMessage and send their results through an ObstacleTrajectoryMessage to the prediction module.

From the ObstacleTrajectoryMessage, the prediction module observes the bounding box and identifier of each obstacle at every time instance to predict future behavior, generating an ObstaclePredictionMessage. The message contains the past and predicted trajectory of each obstacle. Unlike the perception components, the prediction module lacks a "perfect" implementation, so the linear predictor reference implementation is used as a baseline.

Once all of the perception messages are received, Pylot's internal representation of the world, the World class, can be updated with the perception outputs. Although Pylot uses World as the representation of the planning world for the planning module, it serves as a convenient representation of the environment's current state. The World class stores information about the driving environment, such as the ego-vehicle's current state and its trajectory, the route to be followed, detected agents and their trajectories, detected lanes,

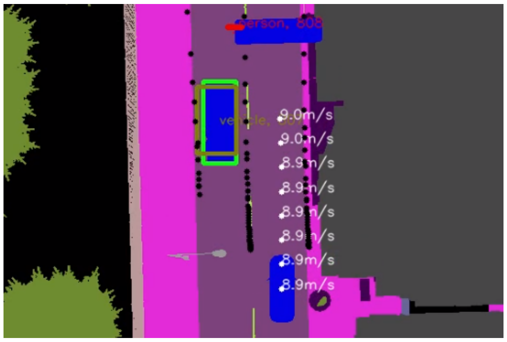


Figure 3.2: A bird's eye view representation of an AV's surroundings produced by the Pylot AV pipeline [27]. The representation is annotated with driveable regions, nearby vehicles, pedestrians, and the predicted paths of nearby agents.

and detected traffic lights and signs.

3.2 Approach

Our approach, caRLot, builds an OpenAI gym environment [8] atop the Pylot modular AV platform, which will interface with the CARLA simulator. By integrating with Pylot and CARLA, we gain access to previously discussed features, namely a framework for a modular ADS, a simulator that can be used for training and evaluating RL policies, and reference and "perfect" implementations of AV components. As shown in Figure 3.3, caRLot will use just the perception stack of Pylot, replacing the planning and control modules with a separately learned DRL policy. Essentially, our design bridges the gap between modular and end-to-end approaches by developing a hybrid system.

Our hybrid approach consists of the modules in the perception stack (sensing, perception, prediction) and an end-to-end RL model. In designing caRLot, we consider the challenges of achieving robustness, efficiency, and interpretability. We specifically replaced the planning and control module with an end-to-end RL policy to improve robustness and efficiency. Replacing these modules with a single learned policy reduces the overall complexity of the system, eliminating the need for the predefined output structure between planning and control components. Explicitly replacing the control module with a learned policy also improves our ability to generalize across a range of scenarios and lowers the computational complexity. For our end-to-end model, we use deep reinforcement learning over imitation learning because RL models can take advantage of the CARLA's simulated driving environment and learn from a large variety of scenarios to develop a robust, reliable policy. We make a further design choice to opt for model-free RL approaches over model-based ones as our end-to-end model. Though model-based policies have improved sample efficiency, they rely heavily on the simulator's environment dynamics which may not be indicative of the real-world driving environment. Furthermore, using model-free RL enables us to easily interchange and experiment with different algorithms as long as they are compatible with our environment. We will discuss this in further detail in Section 3.6

These added benefits of using end-to-end learning in the ADS pipeline questions the need for a hybrid approach, rather than a full end-to-end system. Retaining the perception and prediction modules will improve the interpretability of our overall system and the sample efficiency of the downstream DRL policy. Since we maintain the perception stack of the modular approach, we can still observe the intermediate outputs between modules for debugging purposes. We've discussed in 2.2 that DRL models struggle with sample inefficiency, which is only exacerbated by the high sample complexity of raw sensor data in self-driving. As demonstrated by Chen et al. [16], we can use the output of the perception stack instead of raw sensor data for our DRL model to reduce the sample complexity of the observation space substantially. However, instead of passing in the perception stack output directly, we design a bird's eye view representation to further promote learning by improving sample efficiency. Finally, we follow prior approaches to reducing ambiguity at intersections by incorporating routing information into our observations, using high-level navigational commands [44, 14, 12]. We will now discuss more specifics about the implementation of our approach in the subsequent sections.

3.3 Environment Setup

Since we realize our caRLot through an OpenAI gym environment built atop Pylot, we first discuss our gym environment setup and the Pylot integrations.

Pylot Setup

When initializing or resetting the gym environment, all of the Pylot operators must be initialized. Since we will be interfacing with CARLA, we use Pylot's `CarlaOperator` which will act as a bridge to the CARLA simulator. The `CarlaOperator` sets the required town and weather, initializes the required number of actors, and initializes the ego-vehicle. It also provides access to privileged information, such as the current ego-vehicle pose and the ground truth information necessary to create perfect components.

With the ego-vehicle constructed in the simulator, we can begin adding the necessary operators that form our perception stack. We configure caRLot to use cameras with a defined resolution of `camera_width` × `camera_height`. Though cameras of different placements and orientations can be used (left, right, and rear cameras), we use only center cameras: cameras positioned at the center of the vehicle with a forward-facing orientation. Following the center camera setup, we add RGB, depth, and segmented `CameraDriverOperators` to the ego-vehicle, each of which provides camera frames of their corresponding type through an ERDOS stream. In addition to the camera sensors, a GNSS (GPS) sensor is added to our vehicle, used specifically for routing purposes. Since we will be using the ground truth pose

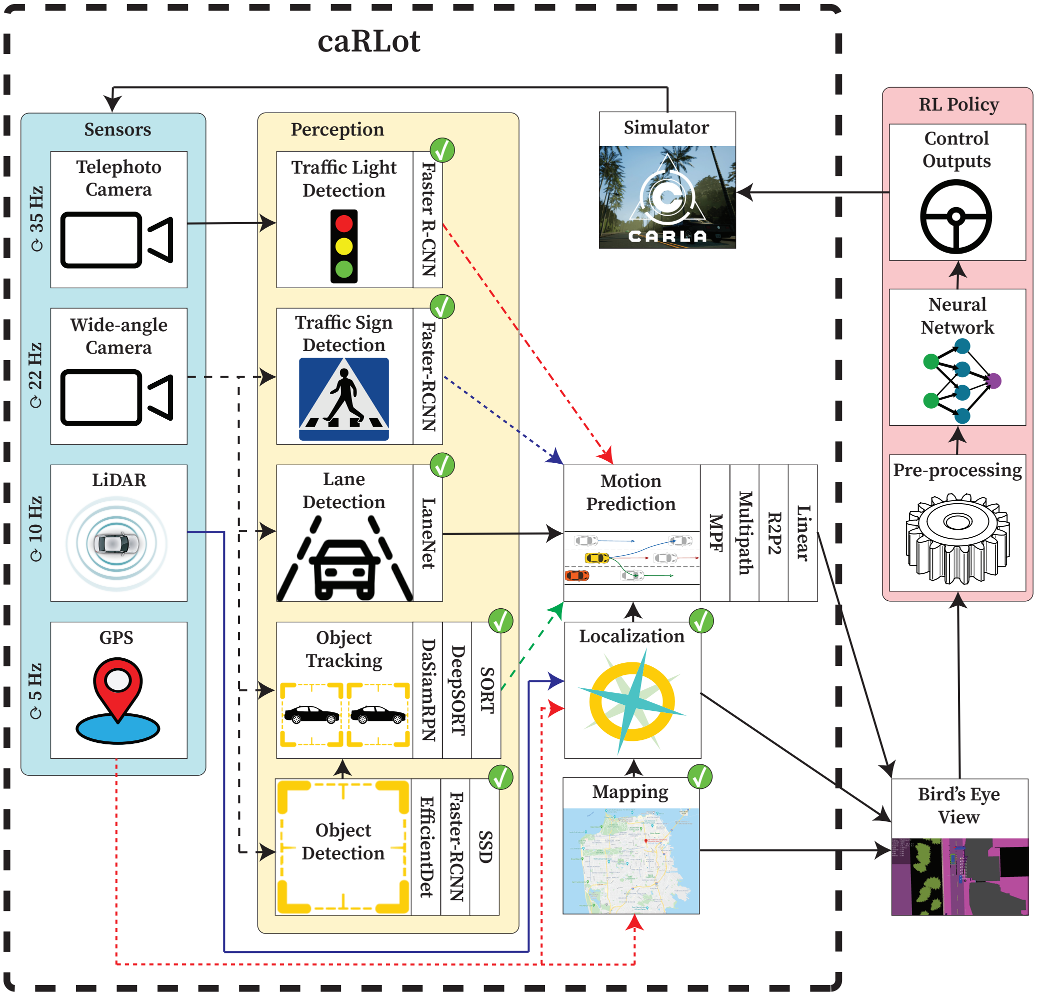


Figure 3.3: The caRLot gym environment integrates with Pylot [27] and CARLA [22] to provide customizable configurations of interconnected sensors and components to produce a bird's eye view (BEV) representation of the AV's surroundings. Thus, caRLot enables the development of fully learned policies that produce control outputs directly from BEV, and take the place of traditional AV trajectory planning and control algorithms (e.g. RRT Planner [35] or model predictive control [39]). caRLot can evaluate an policy's robustness by taking advantage of Pylot's variety of component implementations (including "perfect" implementations that use ground-truth information from the simulator, marked with a green check mark) to introduce pertubations in the BEV, mirroring the introduction of new weights and different model architectures in the development cycle of real world AVs.

instead of estimating it through localization, an IMU sensor will be unnecessary. Finally, a collision sensor is added to the ego-vehicle to notify downstream operators of collision events with other agents.

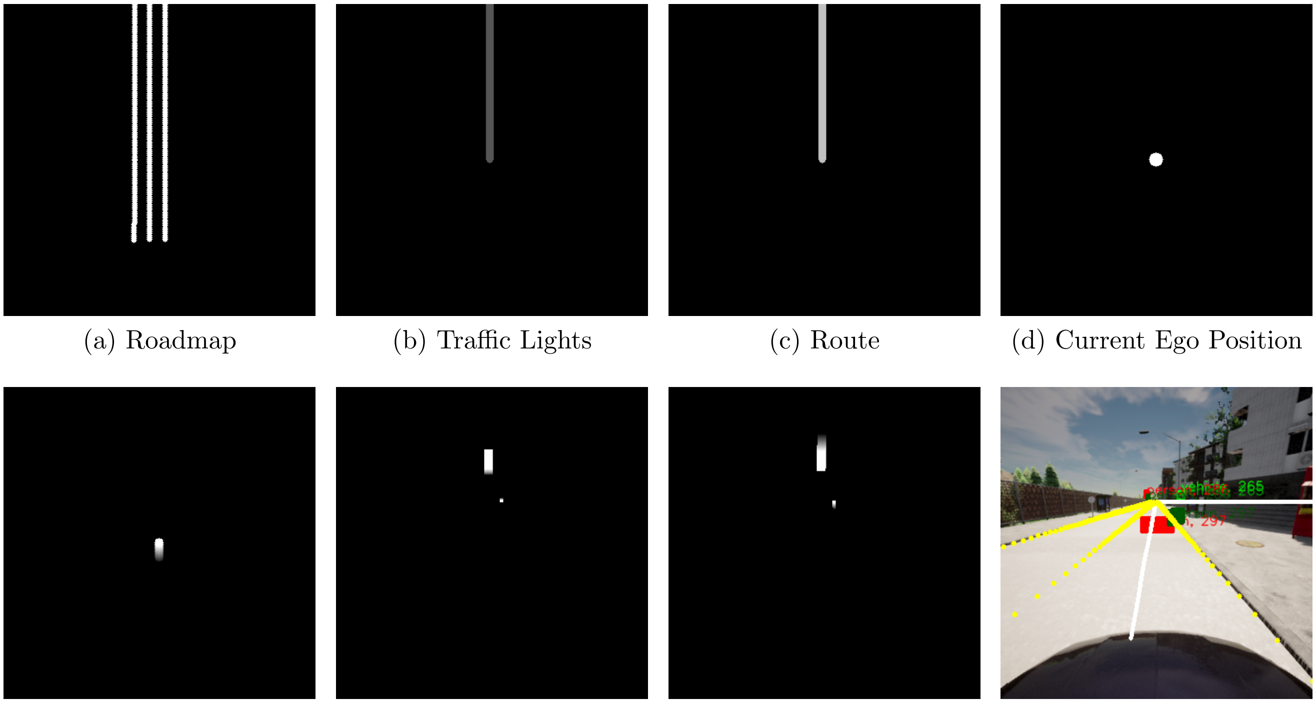
Once the camera sensors are set up, we can configure the remainder of the perception stack, the perception and prediction components. We set up caRLot to use Pylot's implementations for obstacle detection, obstacle tracking, lane detection, traffic light detection, and trajectory prediction (see 3.1 for more details). Each of these components is customizable; caRLot can select from a variety of models as well as "perfect" components that use privileged information from the simulator to provide ground truth predictions. Once all of the operators in the perception stack are configured, we initialize Pylot's planning *World*, which will represent the state of the driving environment perceived by the ego-vehicle.

Since we will be evaluating performance based on our agent's ability to reach a target destination, we implemented a gym environment wrapper, `RouteWrapper`. `RouteWrapper` receives the spawn and goal point indices which determine where in the current town the ego-vehicle is spawned and will aim to reach. If these indices are not set, a random point will be selected. `RouteWrapper` will convert input point indices into actual locations in the CARLA map and uses the simulator to compute a high-level route from the start location to the goal location. `RouteWrapper` also computes the distance to the goal destination and lets us know when we've reached the destination.

Observation Representation

At each environment step, we feed camera inputs to the AV perception components and receive as outputs raw observations of the vehicle's surroundings from perception messages. This will include all of the data stored in `TrafficLightsMessage`, `ObstacleMessage`, `ObstacleTrajectoryMessage`, `LaneMessage`, `ObstaclePredictionMessage` such as detected traffic lights, detected lane markings, and bounding boxes of nearby agents and their predicted motion. We also read from the GNSS sensor to receive the current latitude, longitude, and altitude of the vehicle. This information is used to fetch the high-level navigation command of the closest waypoint in the computed route. The high-level command is then mapped into a numerical value ranging from $[0, 5]$ in the following order: "turn left", "turn right", "go straight", "follow lane", "change lane left", and "change lane right". Finally, we update the `World` with the raw perception observations to maintain the perceived environment state at each iteration.

In pre-processing steps, we facilitate learning by transforming this abstract representation into a more sample-efficient representation of the AV's surroundings. Using Pylot's visualization tools, we could draw the perception stack outputs onto a camera frame. For instance, the detected objects would be visualized with a bounding box around the object with a class label and detected lanes visualized by plotting the lane markings on the frame. We utilize the camera setup's extrinsic and intrinsic to compute a top-down transform to convert our camera representation into a bird's-eye-view representation. This is done by transforming all detection coordinates into bird's-eye-view coordinates, before drawing onto



(a) Roadmap (b) Traffic Lights (c) Route (d) Current Ego Position

(e) Past Ego Trajectory (f) Past Obstacle Trajectory (g) Predicted Obstacle Trajectory (h) Center Camera View

Figure 3.4: Bird's-eye-view representation of perception stack output (a-g) and original center camera view (h)

a frame. As shown in 3.4, the modified representation comprises of several images of size $W \times H$ rendered into a top-down (bird's-eye-view) coordinate system. Each of the images is grayscale. (a) Roadmap: a binary image $\{0, 255\}$ containing information about the road. Detected lane markings and stop signs are rendered on the road map. (b) Traffic Lights: the intended route that the ego-vehicle should follow, with brightness varying on traffic light state. The brightest level for red lights, an intermediate gray level for yellow lights, and a darker level for green and unknown lights. (c) Route: the intended route that the ego-vehicle should follow, with brightness varying on high-level command. (d) Current ego position: a binary image 0, 255 with the ego-vehicle's current position rendered as a large circle. (e) Past ego trajectory: the ego-vehicle's past t positions are rendered as small circles with reduced brightness meaning earlier timesteps. (f) Past Obstacle trajectory: past bounding boxes of detected agents with reduced brightness meaning earlier timesteps. (g) Predicted obstacle trajectory: predicted bounding boxes of detected agents with reduced brightness meaning later timesteps.

Since the original camera image is of dimension `camera_width` × `camera_height`, we resize each image to $W \times H$. We then concatenate all of the images to form a $W \times H \times 7$

representation. We use this bird's-eye-view representation as observations for our caRLot gym environment.

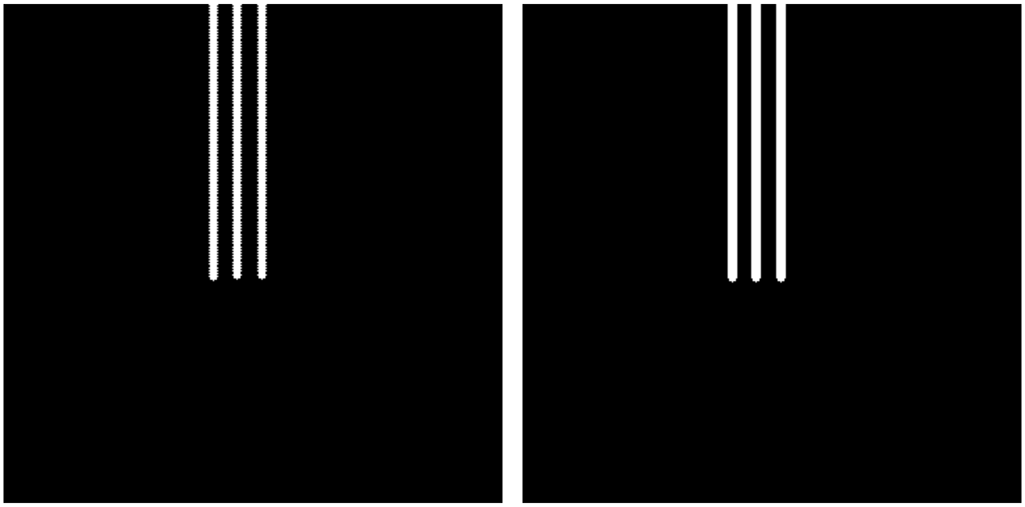
Action Space

The traditional action space A for the ego-vehicle has continuous steering: $s \in [-1, 1]$, continuous throttle: $t \in [0, 1]$, and discrete braking: $b \in \{0, 1\}$. To expedite training, we take inspiration from CARLA's RLlib integration [10] and Toromanoff et al.'s work [72] and discretize the continuous action spaces as follows: restrict $s \in \{0, \pm 0.5, \pm 0.75\}$ and restrict $t \in \{0.0, 0.3, 0.6, 1.0\}$. We can further simplify the action space into just 28 discrete actions using various combinations, ranging in actions from coasting, applying break, and moving straight, left, and right at various speeds and orientations. Discrete actions can be compensated in the future by making the actions more fine-grained and increasing the number of discrete actions [72].

3.4 caRLot Implementation

caRLot is implemented as an OpenAI gym environment that wraps Pylot's APIs to set up a customizable simulated AV pipeline across all components except planning and control. A caRLot environment takes as input a configuration (e.g. a YAML file or a Python dictionary), which it uses to set up Pylot's sensors and components. We encountered several challenges in supporting the `reset()` method of the OpenAI Gym interface. Firstly, Pylot is not designed to frequently reset its internal state, so caRLot shuts down and restarts Pylot upon every reset. Consequently, we needed to add features to caRLot to manage the execution and graceful termination of Pylot processes, to avoid hanging and orphaned processes. Moreover, we wanted to run multiple invocations of Pylot in parallel to support training at scale; however, this caused issues when running multiple Pylot instances on the same machine, as Pylot uses statically-allocated TCP ports to communicate between the processes that encapsulate AV components, resulting in errors when multiple processes attempted to bind to the same ports. To solve this problem, we modified Pylot to dynamically select available TCP ports which prevent different processes from binding to the same port.

Another major challenge we encountered during our initial implementation was that environment steps were extremely slow. We learn from initial experiments, discussed in Chapter4, that environment steps took 10.7 seconds on average. Through further analysis, we determined that a large amount of time was spent converting the raw perception output into our perception representation, specifically, drawing detected lanes onto the Roadmap frame using Pylot's visualization tools. This task was taking a substantial amount of time because the "perfect" lane detection operator represented each lane using on average 10,000 lane markings. Transforming all markings into the bird's eye view representation was very computationally expensive. To alleviate the issue, we sampled from the set of detected lane markings and decided to only transform and plot the lane markings we sampled. This in-



(a) Roadmap with sampling

(b) Roadmap without sampling

Figure 3.5: Comparison between the roadmap perception component with (a) and without (b) sampling. With sampling, the lanes appear more slightly more jagged.

troduced a trade-off between latency and quality of the detected lane representation: using fewer samples would improve the latency but make the lane appear more dotted. We experimented with various sampling approaches and settled on using systematic sampling with a periodic interval of 10. We similarly applied this optimization for other frames that required many transforms. In addition to sampling, we removed some repetition in drawing the lanes. Since each lane is represented by left markings and right markings, the lane markings in the middle of two adjacent lanes will be overlapping and drawn twice. Thus, we made further optimizations by drawing the overlapping markings only once. Through these optimizations, we were able to reduce the duration of each environment step to 800-1000 ms, which is over a 90% reduction. Furthermore, the improved runtime did not have a large impact on the visual quality of the frames. Visual changes to the representation did not appear too drastic, as seen in Figure 3.5. Most of the remaining latency is attributed to the delay in gathering observations from the simulator.

3.5 Reward Design

Initial Design

We experimented with two different reward functions. Our first reward function depended only on the current ego-vehicle's pose. Since we planned to evaluate our model on the NoCrash benchmarks, our initial rewards setup focused on rewarding behavior that would lead to getting to the destination as efficiently as possible. A straightforward way of treating

Listing 3.1: An example rollout using caRLot. The `config` specifies caRLot should set up the underlying AV pipeline. In the example, we use perfect perception components, and use linear prediction. The environment is then created using the config, and wrapped by the `RouteWrapper`, which generates random start and goal locations between which the AV must navigate. The policy then interacts with the environment by producing actions from the provided observation.

```
config = pylot_env.PylotConfig(camera_image_width=360,
                               camera_image_height=360,
                               bird_image_width=96,
                               bird_image_height=96,
                               town=1)
config.add_lane_detection(‘perfect’)
config.add_obstacle_detection(‘perfect’)
config.add_traffic_light_detection(‘perfect’)
config.add_obstacle_tracking(‘perfect’)
config.add_prediction(‘linear’)
config.add_gnss_sensor(‘gnss_sensor’)
env = pylot_env.PylotEnv(config)
env = pylot_env.RouteWrapper(env,
                             spawn_point_index=-1,
                             goal_point_index=-1)

obs = env.reset()
env.render()
policy = ...

done = False
while not done:
    action = policy.predict(obs)
    obs, reward, done, info = env.step(action)
    env.render()
```

this task would be to have a large positive reward when reaching the destination and a large negative reward when crashing into an obstacle. The discount factor would place more emphasis on achieving the goal sooner. However, reaching the goal destination is a very sparse event and can only occur once the policy has learned to navigate through a difficult route. Thus, we added reward shaping to provide more frequent signals on appropriate behaviors during training.

First, to ensure that the ego-vehicle was moving towards the goal destination, we introduced a continuous reward based on the distance traveled towards the destination. We used the `RouteWrapper` to compute the distance between the current ego position and the goal destination. The reward is positive if the distance to the destination decreases (getting closer to the destination), and negative otherwise (moving away from the destination). To ensure that the agent does not remain idle, we penalized the agent (-10) if it remained stagnant in a position for a certain threshold `max_time_idle`. We determined the agent is stagnant by checking if the distance traveled in each step was close to 0. To promote following the route, the agent is penalized (-10) if it makes a wrong turn at an intersection. We assumed an agent made the wrong turn if the agent moves away from the goal destination for `max_wrong_route` consecutive steps.

During training, we prescribe a set of termination conditions that, when met, will send a “done” signal to end the current episode. The primary conditions to terminate occur when the ego-vehicle reaches the goal destination or when it crashes. The collision sensor notifies the rest of the pipeline when our agent crashed. During training, we also terminate when `max_time_idle` or `max_wrong_route` thresholds are met. Lastly, we tracked the number of steps taken in the current episode and terminate the episode when reaching `max_time_step` number of steps. The agent is also provided a reward of +10 when reaching this condition. This serves as an intermediary goal for the agent: instead of just aiming to reach the destination, the agent tries to move towards the goal without crashing.

Revised Design

After running experiments with the initial rewards design, we decided to revise our rewards function, following prior works more closely. Our incentives for redesigning the rewards are discussed in Chapter 4. For our revised reward function, we drew inspiration from the reward designs in World On Rails [12] and End-to-End Model-Free Reinforcement Learning for Urban Driving using Implicit Affordances [72]. Rather than solely basing the reward on distance traveled like our initial approach, we consider the following components: speed reward, stop reward, brake reward, position reward, orientation reward, and steering reward. For convenience, at timestep t, we define the ego-vehicle’s speed as v_t and high-level navigation command to follow as c_t. The current control command is `throttle`, `steer`, `brake`.

The speed reward r_{speed} is based on the idea of a target speed, introduced by Toromanoff et al. [72]. We introduce two parameters into our caRLot configuration: target speed v_{turn_tgt} and turn target speed v_{tgt}. These parameters determine the speed that the ego-vehicle should maintain when there are no events at the scene. Moreover, we use the current high-level command value, c_t, to determine whether to use target speed or the turn target speed. To compute the target speed in different traffic scenarios, we multiply the target speed with a "speed factor". The speed factor v_f, within range $[0, 1]$, is computed using information from the perception stack, such as detected obstacles and traffic lights. The speed factor is reduced whenever the ego-vehicle's path is blocked by an obstacle or agent, or if it must stop at a traffic light. When the speed factor is 0, the ego-vehicle is expected to make a full stop. We define the component in the following equation, where the ego-vehicle's current speed is defined as v_t:

$$r_{speed} = \begin{cases} 1 - \frac{|v_f \cdot v_{turn\_tgt} - v_t|}{v_{turn\_tgt}} & \text{if } c_t \in [0, 1] \\ 1 - \frac{|v_f \cdot v_{tgt} - v_t|}{v_{tgt}} & \text{otherwise} \end{cases} \tag{3.1}$$

The stop reward r_{stop} is also based on the agent's current speed, providing a penalty when the agent is driving in a "zero-speed" region with greater than zero speed. We determine if the agent is within the region by checking if the speed factor is 0. The reward ranges from $[0, -1]$, minimum when $v_t = 0$. We compute the stop reward as follows:

$$r_{stop} = \begin{cases} -\tanh v_t & \text{if } v_f = 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.2}$$

The brake reward r_{brake} is similarly provided when the ego-vehicle is within the "zero-speed" region. A greedy reward of +1 is provided when the agent brakes in the region.

$$r_{brake} = \begin{cases} 1 & \text{if } v_f = 0 \cap \text{brake} = 1 \\ 0 & \text{otherwise} \end{cases} \tag{3.3}$$

The position and orientation rewards are computed similarly to how they're computed in Figure 2.5. The position reward r_pos ranges from [0, -1] and is based on the lateral distance from the ego-vehicle to the center of the lane. The lateral distance d_{lat} is computed using the vehicle's current pose and the closest waypoint in the route. The reward is 0 when the vehicle is exactly at the center, and -1 when the vehicle is out of the lane. We define the maximum distance from the lane center as d_{max} for the computation:

$$r_{pos} = -d_{lat}/d_{max} \tag{3.4}$$

The orientation reward r_{orient} is based on the heading error of the ego-vehicle. We define heading error as the angle between the ego-vehicle's forward vector (f_e) and the closest waypoint's forward vector (f_w). The heading error will range from [0, 1], maximized when the forward vectors are perpendicular. We define the orientation reward, ranging from [0, -1]:

$$r_{orient} = 1 - \arccos\left(\frac{f_w \cdot f_e}{||f_w||\ ||f_e||}\right) \tag{3.5}$$

Finally, the steer reward is based on whether the current steer command matches the high-level navigation command. If they do not match, the steer reward is set to -1.

$$r_{steer} = \begin{cases} -1 & (\text{steer} \neq 0 \cap c_t \in [2]) \cup \\ & (\text{steer} > 0 \cap c_t \in [0, 4]) \cup \\ & (\text{steer} < 0 \cap c_t \in [1, 5]) \\ 0 & \text{otherwise} \end{cases} \tag{3.6}$$

All reward components are additive, so the total reward r_t at timestep t is:

$$r_t = r_{speed} + r_{stop} + r_{brake} + r_{pos} + r_{orient} + r_{steer} \tag{3.7}$$

During training, we terminate the episode when the ego-vehicle reaches the goal destination, crashes or goes outside the target lane. Leaving the target is indicated by a $r_{pos} <= -1.0$. Unlike our initial reward design, we do not provide any additional reward or penalty when the episode terminates. Chen et al. found it unnecessary to explicitly penalize collisions when using zero-speed zones and brake rewards [12].

3.6 Policy

Using the caRLot gym environment and defined reward functions, we aim to build an end-to-end driving policy $\pi(I)$ that at each timestep t, maps I, the bird's eye view representation of the perception stack output and high-level command, to a discrete action $a \in A$ mapping to a raw control command. To implement and train our RL policy, we integrated our gym environment with Ray RLlib [43]. RLlib is an open-source library for reinforcement learning that offers a simple, customizable RL training workflow. RLlib also has an existing set of implemented model-free RL algorithms with default configurations, making it straightforward to experiment with various RL algorithms as long as they are compatible with the environment setup. Furthermore, the model architecture used in each algorithm can be customized appropriately. Since we will be training a vision-based policy, we design a CNN-based neural network architecture. Other relevant training configurations that we modify include the batch size, optimizer, learning rate, and exploration method. We will discuss how each parameter is set during training in Chapter 4. For the purposes of this paper, we use RLlib to train our models in an online setting. RLlib supports offline learning, which we will address in Chapter 5.

RL Algorithm

To decide on which model-free RL algorithms to use for training, we consider the following challenges: sample inefficiency and slow environment. We can also only consider algorithms that support discrete actions. Though we addressed the challenge of sample efficiency by introducing a perception representation with more optimal sample complexity, it is still necessary to utilize a sample-efficient RL algorithm during the training process. This is largely in part to our slow simulator environment. Even though we significantly improved the run time of each environment step, the current duration of 800-1000 ms is still a challenge for online training. We decided to narrow the list of available RLlib algorithms to those that support off-policy learning. Off-policy algorithms are significantly more sample-efficient than on-policy algorithms because past experiences can be reused through a replay buffer. Furthermore, using an off-policy RL algorithm is consistent with previous approaches that used model-free RL to learn a driving policy [72, 16]. We experiment with off-policy algorithms IMPALA [23], DQN [51, 30], and SAC [28]. We discuss each algorithm when describing our experiment setup in Chapter 4.

Chapter 4

Experiments

4.1 Experimental Setup

All experiments in this chapter were performed on top of CARLA 0.9.10 on a machine having 2 Xeon Gold 6226 CPUs, 128GB of DDR4 RAM, and 2 Titan-RTX 2080 GPU with CUDA version 11.4.

Benchmark

We will train and evaluate our method on the CARLA simulator, using the caRLot gym environment. We specifically use the CARLA NoCrash benchmark, which is designed to test the ego vehicle's ability to handle complex events caused by changing traffic conditions and dynamic agents (pedestrians, other vehicles) in the scene, similar to an urban environment [19]. The NoCrash benchmark includes 50 predefined routes: 25 in the training town (Town 1) and 25 in the test town (Town 2). Each route is defined by a spawn and goal point index, each of which corresponds to a predefined location in the CARLA town. Agents are evaluated on success rate. A trial is considered successful if the agent can safely navigate from the starting position to the goal position within a time limit. We set the time limit to 10,000 environment steps, which approximates the maximum number of steps necessary to drive each route at 5 km/h. In addition to failing to meet the time limit, a trial is deemed a failure when the collision sensor detects that a collision above a preset threshold has occurred.

The CARLA simulator provides different weather settings, allowing users to specify weather conditions such as cloudiness, precipitation, and solar altitude angle. The NoCrash benchmark is evaluated under six weather presets, split into train and test sets. For instance, the training weathers are “Clear noon”, “Clear noon after rain”, “Heavy raining noon”, and “Clear sunset” while the test weathers are “After rain sunset” and “Soft raining sunset”. However, we reduce the scope of training and evaluation to use only the “ClearNoon” weather preset.

The NoCrash Benchmark also proposes three different tasks: Empty Town, Regular Traffic, and Dense Traffic. We run our experiments on the Regular Traffic condition, randomly distributing 20 vehicles and 40 pedestrians throughout the town. Each vehicle and pedestrian move autonomously throughout the town, simulating a multi-agent urban environment.

Training Setup

Using our integration with RLlib, we train an RL agent on the caRLot gym environment under an online setting. Though we can fundamentally use any supported RL algorithm, we experiment with and train agents using the following sample-efficient, off-policy algorithms: DQN, SAC, and IMPALA. For our gym environment, we use our standard perception stack consisting of the following Pylot components: camera sensors, GPS, object detection, object tracking, lane detection, traffic light detection, and object prediction. Furthermore, we use the "perfect" versions of the object detection, object tracking, lane detection, and traffic light detection operators and use linear trajectory prediction.

Regarding the sensor configuration, we initially set the camera resolution to 96×96. However, we observed that it was very challenging to identify or distinguish certain features in our representation, such as the object trajectories and lane markings. Consequently, we increased the camera resolution and set it to 300×300 to ensure that the perception stack has enough resolution for precise detection in the scene. Once the perception stack representation is constructed with the larger resolution, its resolution can be resized to the bird's-eye-view resolution, which we set to 96×96. Under this setup, the policy $\pi(I)$ will receive a $96 \times 96 \times 7$ dimensional input, so we set $dim = 96$.

On top of the caRLot environment, we add two gym environment wrappers: `RouteWrapper` and `NormalizeObservation`. `NormalizeObservation` uses a running mean and std to normalize our observation representation. The `RouteWrapper` is necessary so that the agent can follow a defined route, computed when given a spawn and goal point index. Through the `RouteWrapper`, we can train each agent using the training routes from the CARLA simulator NoCrash benchmark. At each new episode, the agent is given a random spawn and goal point index from the set of training routes. The `RouteWrapper` will update the base environment with the route so that the correct high-level commands will be used in the perception representation and reward function. Once the environment is finished initializing, the agent will aim to follow the route at a target speed, until a termination condition is triggered. We set the target speed to 6.5 km/s and turn target speed to 6.0 km/s. matching the target speeds set by Chen et al. [12]. Once the episode terminates, the environment (and all Pylot operators) will be reinitialized before the new episode can begin with a newly selected route. This ensures that the simulator and Pylot operators are always up to date and properly synced.

RL Configuration

We train a policy $\pi(I)$, over the caRLot gym environment, to learn how to drive through routes from the NoCrash benchmark using various RL algorithms. We first discuss the common settings for the RL algorithms we experiment with. In each of our experiments, we use the same policy network architecture: a vision-based neural network architecture consisting of 4 convolutional layers followed by 2 fully connected layers. Supposing each convolutional layer is defined by the tuple (filters, kernel size, strides), our convolutional layers consists of [(16, 5, 4), (32, 5, 2), (32, 5, 2), (64, 5, 2)]. The fully connected layers have hidden dimensions of [256, 512] and are followed by a tanh activation. To train all of our networks, we use the Adam optimizer with a preset initial learning rate, dependent on the algorithm. We use the default training configurations for each algorithm and discuss deviations from the default for each algorithm in the following:

1) IMPALA: Importance Weighted Actor-Learner Architecture (IMPALA) is proposed to be data efficient and highly scalable [23]. IMPALA follows an actor-critic setup, using a central learner running SGD in a tight loop while asynchronously pulling sample batches from many actor processes. A novel off-policy learning algorithm called V-Trace is used for off-policy correction.

Training experiments with IMPALA occurred prior to the other experiments. The IMPALA experiment is run using the initial rewards function before the optimizations for environment steps from Section 3.4 were applied. Without any optimizations, we were only able to train the IMPALA agent for 15K timesteps. Under our initial rewards setup, we set `max_time_step = 150`, `max_time_idle = 75`, and `max_wrong_route = 25`. We used the Adam optimizer with initial learning rate 0.0025. The discount factor is set to 0.99. We use stochastic sampling as the exploration algorithm.

2) DQN: The Deep Q-network (DQN) algorithm uses a deep neural network to approximate the Q value $Q_\pi(s, a)$, the expected total reward from taking action a in state s and following policy π afterward. With a target network, DQN supports off-policy learning, using a replay buffer to store transition pairs. A larger replay buffer improves stability but due to memory limitations, we set the replay buffer size to just 1000 transitions.

The standard DQN implementation in RLlib includes dueling DQN [73] and Double DQN (DDQN) [30]. Dueling DQN helps differentiate actions between each other, while DDQN alleviates the issue that the original DQN algorithm has of overestimating Q-values. We retain both of these modifications to improve the training process for the agent.

For the environment, we use our revised reward function. The networks are trained for about 150k training steps, using the Adam optimizer with learning rate 0.0025. We set the train batch size to 64, with a rollout fragment length of 16. The discount factor is set to 0.99. We use RLlib's SoftQ exploration algorithm with temperature set to 1.0 to perform stochastic sampling over the discrete action distribution.

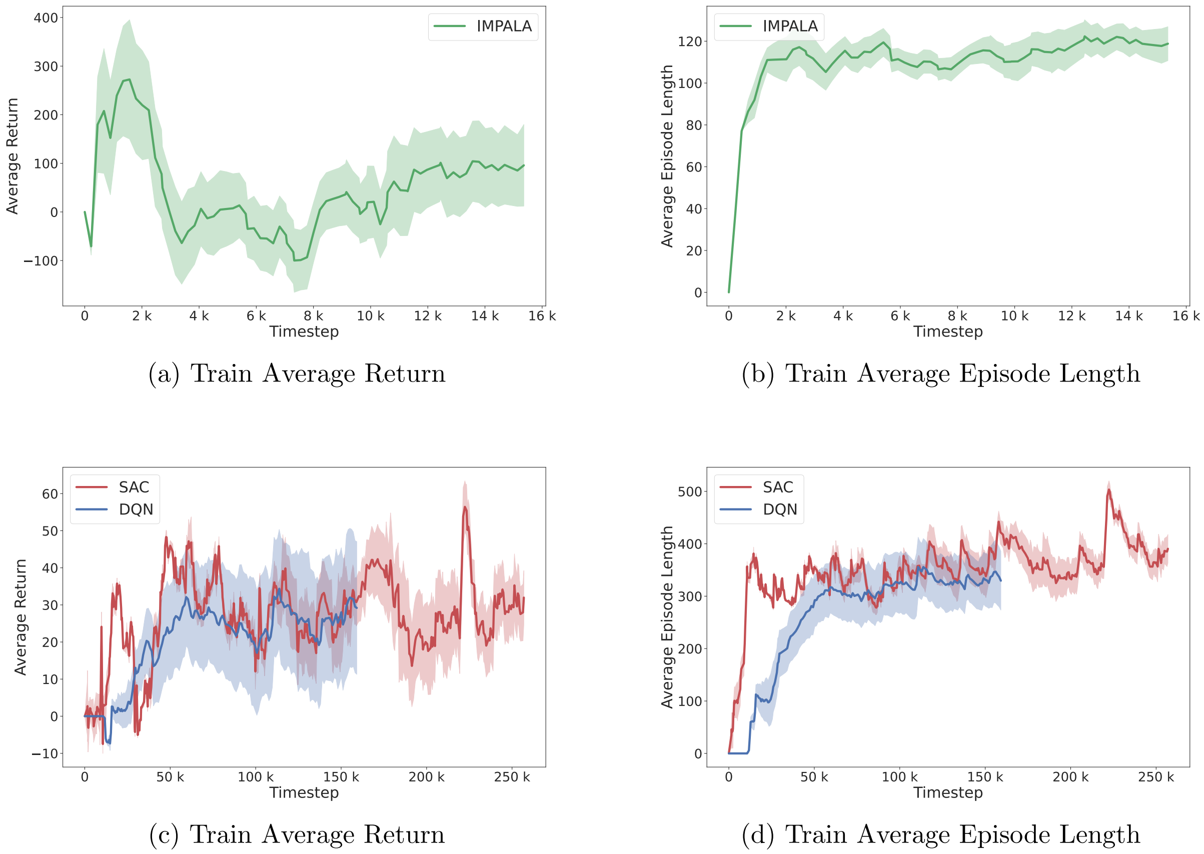
3) SAC: DQN methods often suffer from brittle convergence properties, requiring substantial hyperparameter tuning. The Soft Actor-Critic (SAC) algorithm is proposed to alleviate this issue by using maximum entropy RL. Instead of just maximizing the expected reward, it also maximizes the entropy. SAC defines three networks: a soft Q network, a soft value network, and a policy network and alternates between optimizing the policy evaluation networks (soft Q and soft value) and policy improvement network (policy). We use the same architecture for all the networks. SAC supports using off-policy data from a replay buffer because both value estimators and the policy can be trained entirely on off-policy data. Similar to DQN, we set the buffer size to 1000 due to memory limitations.

Similar to DQN, we use the revised reward function in the environment. The networks are trained for 250k timesteps using the Adam optimizer with learning rate of $3e^{-4}$. The train batch size is set to 64 with a rollout fragment length of 1. The discount factor is set to 0.99. The exploration strategy is included in the SAC policy network, performing stochastic sampling by default.

4.2 Training Results

The training results are summarized in Figure 4.1. From the learning curves of our initial experiment (plots a-b), we see that the average return for the IMPALA agent had minimal improvement. Though further training may eventually lead to more noticeable improvement, we identified issues with the initial reward design that might have contributed to unsatisfactory learning of the task. For instance, it was unnecessary to include a penalty for wrong turns, using `max_wrong_route`, as this is already accounted for by the distance to goal reward. Furthermore, high-level commands were not incorporated into the rewards, so there wasn't a reward signal that guided the agent towards correct steering. Our initial training efforts with IMPALA were drastically hindered by the slow environment step time since the experiment occurred before we introduced the optimizations discussed in Section 3.4. The average environment wait time during initial training was 10.7 seconds, and training just 15K timesteps took around 39 hours. From the initial training experiment, we were able to pinpoint the issue and apply the optimizations we discussed in Section 3.4.

After modifying the rewards and optimizing environment step time, we ran our training experiments for the DQN and SAC agents for about 150K training steps. Comparing the learning curves of SAC and DQN in plots (c-d), we see no significant difference in either the average return or average episode length curves. The learning curves demonstrate that the policies are slowly improving as training progresses. However, the shaded regions demonstrate that DQN is substantially more unstable than SAC, having a very large range of returns and episode lengths. In contrast, the shaded regions for the SAC agent are very minimal throughout most of the curve. This concurs with our previous discussion that DQN struggles with instability and poor convergence, issues that SAC tries to mitigate [28]. Fol-



(a) Train Average Return

(b) Train Average Episode Length

(c) Train Average Return

(d) Train Average Episode Length

Figure 4.1: Learning curves of different RL approaches on the training routes of the NoCrash Benchmark. (a-b) For our initial experiment, the IMPALA agent used our initial reward function. (c-d) The DQN and SAC agents were trained in a following experiment using our revised reward design, hence the difference in reward scale. The statistics are computed from episode rollouts throughout training. The shaded region represents half a standard deviation about the mean at each timestep.

lowing this analysis, we trained just our SAC policy for an additional 100K timesteps, due to limited computing resources. We use the SAC agent for analysis in the subsequent sections.

It is important to note that our online training efforts were still bottlenecked by the slow environment step time, even after optimizing the environment steps. The wait time was still on average 1 second. Thus, training our SAC policy for 350K timesteps took approximately 108 hours. This had a considerable impact on our agent's performance in evaluation, which we discuss next.

4.3 Evaluation

Table 4.1: Success Rate (%) Comparison

Town	LBC	Rails	**caRLot**
train	88	96	**0**
test	80	92	**0**

We attempted to compare our approach to two state-of-the-art approaches: Learning by Cheating [14] and World on Rails [12]. To evaluate our baselines, we ported each of these approaches onto our caRLot-based infrastructure. We utilized the provided pre-trained models on the NoCrash routes from the papers and were able to replicate the claimed results without retraining or modifying the models, demonstrating the versatility of the caRLot software artifact. Table 4.1 list the success rate of the baselines on the CARLA NoCrash benchmark.

Unfortunately, we found that our RL policy was unable to complete a single NoCrash route. We attribute our model's poor performance to insufficient training, as we believe 250k training steps are inadequate for a complex policy to converge for a difficult benchmark. For reference, Chen et al. trained their model-free RL models for about 200 epochs before they exhibited signs of convergence for a simpler task [16]. As each epoch trained using 50k frames, about 10M training steps in total were needed before their model-free RL policy began to reach its goal state. For the Carla leaderboard task [22], Toromanoff et al. trained their model-free policy for 20M steps, taking 23 days of training [72].

For a more direct comparison within our task, we compare with the training process Chen et al. used in World On Rails [12] for the NoCrash benchmark. Instead of online learning, they use offline learning on collected data to train the Rails model to convergence. To gather data, a privileged autopilot drives through the training routes, collecting 270K total frames. The pre-trained Rails model we evaluated was trained for 16 epochs through each of the samples, which equates to over 4 million training steps. Provided that Rails uses a model-based policy and, as a result, is more sample efficient than our model-free strategy, it is not unreasonable to assume that 250k training steps are still an order of magnitude less than what is necessary to sufficiently train our model for the NoCrash driving task. We address potential solutions to improving training on caRLot in Chapter 5.

4.4 Runtime

We compare the runtime of the end-to-end approaches, modular approach (Pylot), and our approach when running through a route in the NoCrash benchmark. We specifically run route town 1, start index 75, and goal index 225 for 1000 timesteps and compute the statistics. We compute the end-to-end runtime from first receiving the sensor data to generating the control commands. This is straightforward for LBC and Rails, as only the runtime of the policy needs to be measured. To compute the end-to-end runtime of caRLot and Pylot, we compute the runtime of the individual Pylot operators using Pylot's logging tools. The end-to-end runtime for Pylot is simply the accumulated runtime of all the operators, excluding the ones for interfacing with the simulator. For caRLot, we sum the runtimes of the perception and prediction operators and the runtime of the RL policy.

However, the overall runtime of Pylot and caRLot will vary depending on the specific operator used for a component. For instance, using the SORT tracker operator will have a much lower runtime than the other tracking operators. To ensure a fair comparison, we keep the operators for the perception and prediction components in caRLot and Pylot consistent. We use a Faster-RCNN model for object detection and traffic light detection, LaneNet [55] for lane detection, the SORT object tracker [6], and linear trajectory prediction. For the planning and control components in Pylot, we use the Frenet Optimal Trajectory planner [75] and the model predictive controller [39].

Table 4.2: Overall Runtime Statistics (ms)

	LBC	Rails	caRLot	Pylot
mean	9.27	9.22	481.84	641.46
std	0.88	0.72	56.45	60.22
min	8.54	8.68	415.91	515.09
25%	8.79	9.00	465.89	612.85
50%	8.85	9.21	477.01	637.45
75%	10.20	9.32	490.54	655.92
max	30.83	27.53	4591.71	5029.03

The overall runtime results are summarized in Table 4.2. Clearly, the end-to-end models are on average substantially faster (9.27 and 9.22 ms) than both caRLot (481.84 ms) and Pylot (641.46 ms). This concurs with our previous discussion about the higher complexity of modular systems compared to end-to-end systems. The components in the perception

Table 4.3: Component Runtime Statistics (ms)

	Perception + Prediction	caRLot Policy	Planning + Control
mean	478.17	3.67	163.29
std	56.44	0.81	21.00
min	412.95	2.96	102.15
25%	462.52	3.36	150.33
50%	473.52	3.49	163.93
75%	486.87	3.67	169.04
max	4571.46	20.24	457.57

modules are especially inefficient, contributing to most of the runtime of caRLot and Pylot as seen in Table 4.3. However, caRLot has lower mean runtime (481.84 ms) than Pylot (641.46 ms), which is about a 25% improvement. Though they share the same perception components, Pylot's planning and control components are quite expensive, having a significantly higher average runtime (163.29 ms) than caRLot's RL policy (3.67 ms). Though not as substantial of a difference as from end-to-end systems, caRLot's lower runtime than the modular approach provides additional breathing room for meeting the critical deadlines in real-world driving. This breathing room can either be preserved to provide better guarantees that deadlines are met or used for more complex, accurate perception models with regard to the accuracy-latency trade-off, which we discussed in Chapter 2.

Chapter 5

Future work

There are many directions in which our work in caRLot can be improved and further extended. We discuss a few areas in the following sections.

5.1 Policy Learning

Though we developed an infrastructure for training driving policies using the caRLot environment, it currently only supports online learning. Online learning in our setting has major drawbacks, which we will address solutions to in this section. Furthermore, we discuss offline learning as a possible alternative.

Improving Online Learning

Online training using caRLot is currently bottlenecked by the simulator delay during each environment step. Since each environment step takes on average 900 ms after applying some optimizations, it'd still take about 250 hours of continuous training to run 1 million training steps. Not only is this impractical, but a million training steps also is likely not enough for a complex policy to converge, even when using sample-efficient RL methods such as model-based approaches. Due to these drawbacks, this work was unable to adequately train a driving model for the NoCrash benchmark, This would've been necessary to evaluate the effectiveness of our proposed perception representation for improving sample efficiency.

To make online learning more practical, future work in reducing the latency of environment steps is necessary. Since the majority of the remaining delay comes from the delay in gathering observations from the simulator, future work should focus on reducing this delay. From further investigation, we discover that the "perfect" lane detection and "perfect" traffic light detection operators, take about 200 ms each. Therefore, an effective next step is to optimize how these operators gather privileged data from the simulator and how they compute the ground truth from the simulator data.

Integrating Offline Learning

Offline reinforcement learning serves as a great alternative to online learning, using previously collected datasets in the environment to learn a policy without having to actively interact with the environment. This would enable us to circumvent the bottleneck created by the latency of environment steps. Integrating offline training is a straightforward task, provided that RLlib has offline training APIs for its existing set of RL algorithms. Furthermore, the RLlib algorithm library includes some algorithms specific to offline RL, such as Conservative Q-Learning (CQL) [42]. These algorithms address major challenges in offline RL, such as the overestimation of Q-values. However, distribution shifts and function approximation are still major issues when using offline RL, and must be addressed when extending offline RL for this work. For instance, policies can be trained from offline data and finetuned through online learning using AWAC [54].

To support offline learning, a data collection tool must be integrated with the caRLot gym environment. For an offline policy to be successful, the offline dataset must cover a large, robust set of trajectories. Fortunately, it is easier to gather a robust dataset from the simulator, than it is from the real world to use for behavioral cloning. A data collection approach similar to previous works that use a privileged autopilot to traverse the route can be integrated [14, 12, 13].

5.2 Interpretability

Compared to traditional end-to-end models, caRLot has an advantage in terms of interpretability: it constructs an intermediate representation of the agent's understanding of the environment using the perception stack. Our use of an intermediate bird's-eye-view representation is quite similar to Chen et al.'s application for improving interpretability [15]. The intermediate representation provides us a better understanding of failure cases, such as when trying to diagnose that a crash was caused by a failed detection of another vehicle.

However, caRLot is still quite limited in interpretability compared to modular systems. Since our driving policy is model-free, it behaves quite similarly to a black box: the policy network $\pi(I)$ provides no intuition on how the bird's-eye-view images are used to generate control commands. In contrast, model-based approaches provide more intuition through their learned understanding of environment dynamics. Future work can explore how model-based algorithms can be integrated with the caRLot environment to improve interpretability. Model-based methods are also substantially more sample-efficient, which can mean fewer environment steps in our expensive environment. However, as we've discussed, using a model-based approach may introduce difficulties with integrating the trained model for use in the real world. Thus, other approaches to improving interpretability include analyzing the visual saliency of the policy network, predicting waypoints instead of control commands, and having auxiliary tasks [70, 12, 13].

Chapter 6

Conclusion

We introduce caRLot, a novel OpenAI Gym environment that enables reinforcement learning with realistic AV observations. We successfully port state-of-the-art models to caRLot and evaluate them on a widely-used benchmark. In addition, we train a model-free reinforcement learning policy that replaces the planning and control components of a typical AV pipeline. Unfortunately, this policy does not outperform solutions that use end-to-end learning; we believe that our policy's ability to learn is bottlenecked by slow simulation time. Thus, despite using a sample-efficient, off-policy reinforcement learning algorithm such as SAC, training would take very long and under the time constraints of the project, we were unable to adequately train our policies. This, of course, had a significant impact on the agent's performance in the NoCrash benchmark as the agent was unable to complete a route.

Despite our policy's poor performance due to such issues, we still demonstrated the major benefits of our hybrid approach. The addition of an intermediate perception representation provides a significant advantage over end-to-end models in terms of interpretability. On the other hand, replacing the planning and control stages of modular systems with a simpler learned policy led to a 25% runtime improvement. Though our work is preliminary, it illustrates the potential of hybrid systems that can achieve "the best of both worlds".

Bibliography

[1] Kiam Heong Ang, G. Chong, and Yun Li. “PID control system analysis, design, and technology”. In: *IEEE Transactions on Control Systems Technology* 13.4 (2005), pp. 559–576. DOI: 10.1109/TCST.2005.847331.

[2] Kai Arulkumaran et al. “A Brief Survey of Deep Reinforcement Learning”. In: *CoRR* abs/1708.05866 (2017). arXiv: 1708.05866. URL: http://arxiv.org/abs/1708.05866.

[3] Autoware. *Autoware User’s Manual - Document Version 1.1*. https://github.com/CPFL/Autoware-Manuals/blob/master/en/Autoware_UsersManual_v1.1.md. 2017.

[4] Baidu. *Apollo 3.0 Software Architecture*. https://github.com/ApolloAuto/apollo/blob/master/docs/specs/Apollo_3.0_Software_Architecture.md. 2018.

[5] Mayank Bansal, Alex Krizhevsky, and Abhijit S. Ogale. “ChauffeurNet: Learning to Drive by Imitating the Best and Synthesizing the Worst”. In: *CoRR* abs/1812.03079 (2018). arXiv: 1812.03079. URL: http://arxiv.org/abs/1812.03079.

[6] Alex Bewley et al. “Simple Online and Realtime Tracking”. In: *Proceedings of the 23th IEEE International Conference on Image Processing (ICIP)*. 2016, pp. 3464–3468. DOI: 10.1109/ICIP.2016.7533003.

[7] Mariusz Bojarski et al. “End to End Learning for Self-Driving Cars”. In: *CoRR* abs/1604.07316 (2016). arXiv: 1604.07316. URL: http://arxiv.org/abs/1604.07316.

[8] Greg Brockman et al. “Openai gym”. In: *arXiv preprint arXiv:1606.01540* (2016).

[9] Sean Campbell et al. “Sensor Technology in Autonomous Vehicles : A review”. In: *2018 29th Irish Signals and Systems Conference (ISSC)*. 2018, pp. 1–4. DOI: 10.1109/ISSC.2018.8585340.

[10] *CARLA and RLlib integration*. https://github.com/carla-simulator/rllib-integration.

[11] Chenyi Chen et al. “DeepDriving: Learning Affordance for Direct Perception in Autonomous Driving”. In: *2015 IEEE International Conference on Computer Vision (ICCV)*. 2015, pp. 2722–2730. DOI: 10.1109/ICCV.2015.312.

[12] Dian Chen, Vladlen Koltun, and Philipp Krähenbühl. "Learning to drive from a world on rails". In: *CoRR* abs/2105.00636 (2021). arXiv: 2105.00636. URL: https://arxiv.org/abs/2105.00636.

[13] Dian Chen and Philipp Krähenbühl. *Learning from All Vehicles*. 2022. DOI: 10.48550/ARXIV.2203.11934. URL: https://arxiv.org/abs/2203.11934.

[14] Dian Chen et al. "Learning by Cheating". In: *CoRR* abs/1912.12294 (2019). arXiv: 1912.12294. URL: http://arxiv.org/abs/1912.12294.

[15] Jianyu Chen, Shengbo Eben Li, and Masayoshi Tomizuka. "Interpretable End-to-end Urban Autonomous Driving with Latent Deep Reinforcement Learning". In: *CoRR* abs/2001.08726 (2020). arXiv: 2001.08726. URL: https://arxiv.org/abs/2001.08726.

[16] Jianyu Chen, Bodi Yuan, and Masayoshi Tomizuka. "Model-free Deep Reinforcement Learning for Urban Autonomous Driving". In: *CoRR* abs/1904.09503 (2019). arXiv: 1904.09503. URL: http://arxiv.org/abs/1904.09503.

[17] Felipe Codevilla et al. "End-to-end Driving via Conditional Imitation Learning". In: *CoRR* abs/1710.02410 (2017). arXiv: 1710.02410. URL: http://arxiv.org/abs/1710.02410.

[18] Felipe Codevilla et al. "Exploring the Limitations of Behavior Cloning for Autonomous Driving". In: *CoRR* abs/1904.08980 (2019). arXiv: 1904.08980. URL: http://arxiv.org/abs/1904.08980.

[19] Felipe Codevilla et al. "Exploring the limitations of behavior cloning for autonomous driving". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019, pp. 9329–9338.

[20] Achal Dave et al. "TAO: A Large-Scale Benchmark for Tracking Any Object". In: *CoRR* abs/2005.10356 (2020). arXiv: 2005.10356. URL: https://arxiv.org/abs/2005.10356.

[21] Edsger W Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische mathematik* 1.1 (1959), pp. 269–271.

[22] Alexey Dosovitskiy et al. "CARLA: An Open Urban Driving Simulator". In: *CoRR* abs/1711.03938 (2017). arXiv: 1711.03938. URL: http://arxiv.org/abs/1711.03938.

[23] Lasse Espeholt et al. "IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures". In: *CoRR* abs/1802.01561 (2018). arXiv: 1802.01561. URL: http://arxiv.org/abs/1802.01561.

[24] Ford. *A Matter of Trust: Ford's Approach to Developing Self-driving Vehicles*. https://media.ford.com/content/dam/fordmedia/pdf/Ford_AV_LLC_FINAL_HR_2.pdf. 2018.

[25] General Motors. *2018 Self-driving safety report*. https://www.gm.com/content/dam/company/docs/us/en/gmcom/gmsafetyreport.pdf. 2018.

[26] Ionel Gog et al. "D3: A Dynamic Deadline-Driven Approach for Building Autonomous Vehicles". In: *Proceedings of the Seventeenth European Conference on Computer Systems*. EuroSys '22. Rennes, France: Association for Computing Machinery, 2022, pp. 453–471. ISBN: 9781450391627. DOI: 10.1145/3492321.3519576. URL: https://doi.org/10.1145/3492321.3519576.

[27] Ionel Gog et al. "Pylot: A modular platform for exploring latency-accuracy tradeoffs in autonomous vehicles". In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2021, pp. 8806–8813.

[28] Tuomas Haarnoja et al. "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor". In: *CoRR* abs/1801.01290 (2018). arXiv: 1801.01290. URL: http://arxiv.org/abs/1801.01290.

[29] Christian Häne et al. "3D Visual Perception for Self-Driving Cars using a Multi-Camera System: Calibration, Mapping, Localization, and Obstacle Detection". In: *CoRR* abs/1708.09839 (2017). arXiv: 1708.09839. URL: http://arxiv.org/abs/1708.09839.

[30] Hado van Hasselt, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-learning". In: *CoRR* abs/1509.06461 (2015). arXiv: 1509.06461. URL: http://arxiv.org/abs/1509.06461.

[31] Kaiming He et al. "Mask R-CNN". In: *CoRR* abs/1703.06870 (2017). arXiv: 1703.06870. URL: http://arxiv.org/abs/1703.06870.

[32] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5 (1989), pp. 359–366. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(89)90020-8. URL: https://www.sciencedirect.com/science/article/pii/0893608089900208.

[33] Yu Huang and Yue Chen. "Autonomous Driving with Deep Learning: A Survey of State-of-Art Technologies". In: *CoRR* abs/2006.06091 (2020). arXiv: 2006.06091. URL: https://arxiv.org/abs/2006.06091.

[34] Joel Janai et al. "Computer Vision for Autonomous Vehicles: Problems, Datasets and State-of-the-Art". In: *CoRR* abs/1704.05519 (2017). arXiv: 1704.05519. URL: http://arxiv.org/abs/1704.05519.

[35] Sertac Karaman and Emilio Frazzoli. "Sampling-based Algorithms for Optimal Motion Planning". In: *CoRR* abs/1105.1186 (2011). arXiv: 1105.1186. URL: http://arxiv.org/abs/1105.1186.

[36] Andrej Karpathy. *Pytorch at Tesla*. https://youtu.be/oBklltKXtDE.

[37] Shinpei Kato et al. "Autoware on Board: Enabling Autonomous Vehicles with Embedded Systems". In: *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems*. ICCPS '18. Porto, Portugal: IEEE Press, 2018, pp. 287–296. ISBN: 9781538653012. DOI: 10.1109/ICCPS.2018.00035. URL: https://doi.org/10.1109/ICCPS.2018.00035.

[38] Alex Kendall et al. "Learning to Drive in a Day". In: *CoRR* abs/1807.00412 (2018). arXiv: 1807.00412. URL: http://arxiv.org/abs/1807.00412.

[39] E. Kim and Myoungho Sunwoo. "Model predictive control strategy for smooth path tracking of autonomous vehicles with steering actuator dynamics". In: *International Journal of Automotive Technology* 15 (Dec. 2014), pp. 1155–1164. DOI: 10.1007/s12239-014-0120-9.

[40] Bangalore Ravi Kiran et al. "Deep Reinforcement Learning for Autonomous Driving: A Survey". In: *CoRR* abs/2002.00444 (2020). arXiv: 2002.00444. URL: https://arxiv.org/abs/2002.00444.

[41] W. Bradley Knox et al. "Reward (Mis)design for Autonomous Driving". In: *CoRR* abs/2104.13906 (2021). arXiv: 2104.13906. URL: https://arxiv.org/abs/2104.13906.

[42] Aviral Kumar et al. "Conservative Q-Learning for Offline Reinforcement Learning". In: *CoRR* abs/2006.04779 (2020). arXiv: 2006.04779. URL: https://arxiv.org/abs/2006.04779.

[43] Eric Liang et al. "RLlib: Abstractions for distributed reinforcement learning". In: *International Conference on Machine Learning*. PMLR. 2018, pp. 3053–3062.

[44] Xiaodan Liang et al. "CIRL: Controllable Imitative Reinforcement Learning for Vision-based Self-driving". In: *CoRR* abs/1807.03776 (2018). arXiv: 1807.03776. URL: http://arxiv.org/abs/1807.03776.

[45] 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.

[46] Liangkai Liu et al. "Computing Systems for Autonomous Driving: State-of-the-Art and Challenges". In: *CoRR* abs/2009.14349 (2020). arXiv: 2009.14349. URL: https://arxiv.org/abs/2009.14349.

[47] 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.

[48] Zhichao Lu et al. "RetinaTrack: Online Single Stage Joint Detection and Tracking". In: *CoRR* abs/2003.13870 (2020). arXiv: 2003.13870. URL: https://arxiv.org/abs/2003.13870.

[49] Vincent Mai, Kaustubh Mani, and Liam Paull. *Sample Efficient Deep Reinforcement Learning via Uncertainty Estimation*. 2022. DOI: 10.48550/ARXIV.2201.01666. URL: https://arxiv.org/abs/2201.01666.

[50] 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.

[51] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *CoRR* abs/1312.5602 (2013). arXiv: 1312.5602. URL: http://arxiv.org/abs/1312.5602.

[52] Sajjad Mozaffari et al. "Deep Learning-based Vehicle Behaviour Prediction For Autonomous Driving Applications: A Review". In: *CoRR* abs/1912.11676 (2019). arXiv: 1912.11676. URL: http://arxiv.org/abs/1912.11676.

[53] Urs Muller et al. "Off-Road Obstacle Avoidance through End-to-End Learning". In: *Advances in Neural Information Processing Systems*. Ed. by Y. Weiss, B. Schölkopf, and J. Platt. Vol. 18. MIT Press, 2005. URL: https://proceedings.neurips.cc/paper/2005/file/fdf1bc5669e8ff5ba45d02fded729feb-Paper.pdf.

[54] Ashvin Nair et al. "Accelerating Online Reinforcement Learning with Offline Datasets". In: *CoRR* abs/2006.09359 (2020). arXiv: 2006.09359. URL: https://arxiv.org/abs/2006.09359.

[55] Davy Neven et al. "Towards End-to-End Lane Detection: an Instance Segmentation Approach". In: *CoRR* abs/1802.05591 (2018). arXiv: 1802.05591. URL: http://arxiv.org/abs/1802.05591.

[56] Nils J. Nilsson. "A Mobius Automation: An Application of Artificial Intelligence Techniques". In: *Proceedings of the 1st International Joint Conference on Artificial Intelligence*. IJCAI'69. Washington, DC: Morgan Kaufmann Publishers Inc., 1969, pp. 509–520.

[57] *NTSB's Accident Report on the Uber Self-Driving Vehicle Crash*. https://www.ntsb.gov/investigations/AccidentReports/Reports/HAR1903.pdf.

[58] Brian Paden et al. "A Survey of Motion Planning and Control Techniques for Self-driving Urban Vehicles". In: *CoRR* abs/1604.07446 (2016). arXiv: 1604.07446. URL: http://arxiv.org/abs/1604.07446.

[59] Yunpeng Pan et al. "Agile Off-Road Autonomous Driving Using End-to-End Deep Imitation Learning". In: *CoRR* abs/1709.07174 (2017). arXiv: 1709.07174. URL: http://arxiv.org/abs/1709.07174.

[60] Jiangmiao Pang et al. "Quasi-Dense Instance Similarity Learning". In: *CoRR* abs/2006.06664 (2020). arXiv: 2006.06664. URL: https://arxiv.org/abs/2006.06664.

[61] Dean Pomerleau. "ALVINN: An Autonomous Land Vehicle In a Neural Network". In: *Proceedings of (NeurIPS) Neural Information Processing Systems*. Ed. by D.S. Touretzky. Morgan Kaufmann, Dec. 1989, pp. 305–313.

[62] Rui Qian, Xin Lai, and Xirong Li. "3D Object Detection for Autonomous Driving: A Survey". In: *CoRR* abs/2106.10823 (2021). arXiv: 2106.10823. URL: https://arxiv.org/abs/2106.10823.

[63] Joseph Redmon et al. “You Only Look Once: Unified, Real-Time Object Detection”. In: *CoRR* abs/1506.02640 (2015). arXiv: 1506.02640. URL: http://arxiv.org/abs/1506.02640.

[64] 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.

[65] Stephane Ross and Drew Bagnell. “Efficient Reductions for Imitation Learning”. In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, May 2010, pp. 661–668. URL: https://proceedings.mlr.press/v9/ross10a.html.

[66] Chinmay Vilas Samak, Tanmay Vilas Samak, and Sivanathan Kandhasamy. “Control Strategies for Autonomous Vehicles”. In: *CoRR* abs/2011.08729 (2020). arXiv: 2011.08729. URL: https://arxiv.org/abs/2011.08729.

[67] David Silver et al. “Mastering the game of Go without human knowledge”. In: *Nature* 550 (Oct. 2017), pp. 354–. URL: http://dx.doi.org/10.1038/nature24270.

[68] Petru Soviany and Radu Tudor Ionescu. “Optimizing the Trade-off between Single-Stage and Two-Stage Object Detectors using Image Difficulty Prediction”. In: *CoRR* abs/1803.08707 (2018). arXiv: 1803.08707. URL: http://arxiv.org/abs/1803.08707.

[69] Pei Sun et al. “Scalability in Perception for Autonomous Driving: Waymo Open Dataset”. In: *CoRR* abs/1912.04838 (2019). arXiv: 1912.04838. URL: http://arxiv.org/abs/1912.04838.

[70] Ardi Tampuu et al. “A Survey of End-to-End Driving: Architectures and Training Methods”. In: *CoRR* abs/2003.06404 (2020). arXiv: 2003.06404. URL: https://arxiv.org/abs/2003.06404.

[71] Mingxing Tan, Ruoming Pang, and Quoc V. Le. “EfficientDet: Scalable and Efficient Object Detection”. In: *CoRR* abs/1911.09070 (2019). arXiv: 1911.09070. URL: http://arxiv.org/abs/1911.09070.

[72] Marin Toromanoff, Émilie Wirbel, and Fabien Moutarde. “End-to-End Model-Free Reinforcement Learning for Urban Driving using Implicit Affordances”. In: *CoRR* abs/1911.10868 (2019). arXiv: 1911.10868. URL: http://arxiv.org/abs/1911.10868.

[73] Ziyu Wang, Nando de Freitas, and Marc Lanctot. “Dueling Network Architectures for Deep Reinforcement Learning”. In: *CoRR* abs/1511.06581 (2015). arXiv: 1511.06581. URL: http://arxiv.org/abs/1511.06581.

[74] *Wayve.AI*. https://wayve.ai.

[75] Moritz Werling et al. “Optimal trajectory generation for dynamic street scenarios in a Frenét Frame”. In: *2010 IEEE International Conference on Robotics and Automation*. 2010, pp. 987–993. DOI: 10.1109/ROBOT.2010.5509799.

[76] Marco Wiering and Martijn van Otterlo. *Reinforcement Learning: State-of-the-Art*. Springer Publishing Company, Incorporated, 2014. ISBN: 364244685X.

[77] Yi Xiao et al. “Multimodal End-to-End Autonomous Driving”. In: *CoRR* abs/1906.03199 (2019). arXiv: 1906.03199. URL: http://arxiv.org/abs/1906.03199.

[78] Fisher Yu et al. “BDD100K: A Diverse Driving Video Database with Scalable Annotation Tooling”. In: *CoRR* abs/1805.04687 (2018). arXiv: 1805.04687. URL: http://arxiv.org/abs/1805.04687.

[79] Ekim Yurtsever et al. “A Survey of Autonomous Driving: Common Practices and Emerging Technologies”. In: *IEEE Access* 8 (2020), pp. 58443–58469. ISSN: 2169-3536. DOI: 10.1109/access.2020.2983149. URL: http://dx.doi.org/10.1109/ACCESS.2020.2983149.

[80] Wenyuan Zeng et al. “End-to-end Interpretable Neural Motion Planner”. In: *CoRR* abs/2101.06679 (2021). arXiv: 2101.06679. URL: https://arxiv.org/abs/2101.06679.

[81] Youcheng Zhang et al. “Deep Learning in Lane Marking Detection: A Survey”. In: *IEEE Transactions on Intelligent Transportation Systems* (2021), pp. 1–17. DOI: 10.1109/TITS.2021.3070111.

[82] Qin Zou et al. “Robust Lane Detection from Continuous Driving Scenes Using Deep Neural Networks”. In: *CoRR* abs/1903.02193 (2019). arXiv: 1903.02193. URL: http://arxiv.org/abs/1903.02193.