Distributed Reinforcement Learning for Autonomous Driving

Zhe Huang

CMU-RI-TR-22-09

April 28, 2022

The Robotics Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA

Thesis Committee:
Jeff Schneider, *advisor*
David Held
Adam Villaflor

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Robotics.

To those who have inspired me,
especially my parents.

Abstract

Due to the complex and safety-critical nature of autonomous driving, recent works typically test their ideas on simulators designed for the very purpose of advancing self-driving research. Despite the convenience of modeling autonomous driving as a trajectory optimization problem, few of these methods resort to online reinforcement learning (RL) to address challenging driving scenarios. This is mainly because classic online RL algorithms are originally designed for toy problems such as Atari games, which are solvable within hours. In contrast, it may take weeks or months to get satisfactory results on self-driving tasks using these online RL methods as a consequence of the time-consuming simulation and the difficulty of the problem itself. Thus, a promising online RL pipeline for autonomous driving should be efficiency driven.

In this thesis, we investigate the inefficiency of directly applying generic single-agent or distributed RL algorithms to CARLA self-driving pipelines due to the expensive simulation cost. We propose two asynchronous distributed RL methods, **Multi-Parallel SAC** (off-policy) and **Multi-Parallel PPO** (on-policy), dedicated to accelerating the online RL training on the CARLA simulator via a specialized distributed framework that establishes both inter-process and intra-process parallelization. We demonstrate that our distributed multi-agent RL algorithms achieve state-of-the-art performances on various CARLA self-driving tasks in much shorter and reasonable time.

Acknowledgments

I would like to express my wholehearted gratitude to my advisor and committee chair, Prof. Jeff Schneider, for his invaluable guidance and extensive support to me and my research throughout my two-year study here at the Robotics Institute, CMU. Without his supervision and his Auton Lab, the whole project would not have been made possible.

Speaking of the computational resources from the Auton Lab, I especially would like to thank Dr. Predrag Punosevac for his excellent operations and maintenance on the lab infrastructure.

I would like to extend my gratitude to Prof. David Held as well. Being my committee member, he has spent his valuable time on reviewing my work and sharing his constructive opinions during my thesis talk.

I have to mention that I am not alone when doing my research. I am fortunate to collaborate with many awesome labmates. I truly appreciate Hitesh Arora and Tanmay Agarwal, our lab alumni, for providing substantial help on understanding the existing codebase as well as sharing their pilot experience on the CARLA self-driving problem. Moreover, I am greatly thankful to Adam Villaflor, Brian Yang, Swapnil Pande and Yeeho Song, for their significant assistance, effort and feedback that make my project drastically better. I honestly believe that our sincere conversations and discussions are extremely helpful and indispensable. All names mentioned in this paragraph are listed in alphabetical order.

Along my journey of my life, I cannot stress this enough that I am deeply indebted to my entire family for their love, care and commitment. I am also grateful to Prof. Yin Li, who has inspired me during my undergraduate years in UW-Madison. Lastly, I simply will not forget all my dear friends, worldwide, who contribute a lot to my happiness and well-being.

At this very moment, I would like to take this opportunity to say “thank you very much indeed” to every individual mentioned above that I cherish. I sincerely wish everyone all the best in life.

Funding

We thank the CMU Argo AI Center for Autonomous Vehicle Research[1] for supporting the work presented in this thesis.

[1] https://labs.ri.cmu.edu/argo-ai-center/

Contents

1 Introduction **1**

2 Background **5**
2.1 Autonomous Driving 5
2.1.1 Modular Pipelines 5
2.1.2 Imitation Learning 6
2.1.3 RL for Self-driving 7
2.2 Reinforcement Learning 9
2.2.1 Preliminaries 9
2.2.2 Off-policy vs. On-policy RL 11
2.2.3 Model-free vs. Model-based RL 11
2.2.4 Online vs. Offline RL 12
2.2.5 Parallel & Distributed RL 12

3 Simulation Environment **15**
3.1 CARLA Simulator 15
3.1.1 Introduction 15
3.1.2 Sensor Affordances 16
3.1.3 Groundtruth Affordances 17
3.1.4 Planners 18
3.2 CARLA Benchmarks 19
3.2.1 CoRL2017 Benchmark 20
3.2.2 NoCrash Benchmark 21
3.2.3 Leaderboard Benchmark 22

4 Multi-Parallel SAC **25**
4.1 Overview 25
4.2 Problem Setup 26
4.3 Method 27
4.3.1 Distributed Framework 27
4.3.2 Multi-agent SAC 32
4.4 Experiments 34
4.4.1 Implementation Details 34
4.4.2 NoCrash Baselines 35

4.4.3 NoCrash Results 37
4.4.4 Sensitivity Analysis 40

5 Multi-Parallel PPO **43**
5.1 Overview 43
5.2 Problem Setup 44
5.3 Method 45
5.3.1 Distributed Framework 45
5.3.2 Multi-agent PPO 47
5.4 Experiments 51
5.4.1 Implementation Details 51
5.4.2 NoCrash Baselines 52
5.4.3 NoCrash Results 52
5.4.4 Leaderboard Baselines 56
5.4.5 Leaderboard Results 57

6 Conclusions **61**

A Supplementary Materials **63**
A.1 Hyperparameters 63
A.2 Adaptive Update Interval 63
A.3 Synchronization Between Servers 65
A.4 Instability in Leaderboard 65

B Limitations of this Study **67**
B.1 Inaccurate Training Speed Profiling 67
B.2 Insufficient Leaderboard Experiments 67
B.3 Lack of Qualitative Analysis 68

C Future Work **69**
C.1 Massive Parallelization 69
C.2 Asynchronous Servers 70
C.3 Accelerating Future Research 70

D Extensions to Other Projects **71**
D.1 Online RL with Vision-based Features 71
D.2 Offline RL with Parallelized Self-play 72

Bibliography **73**

When this thesis is viewed as a PDF, the page header is a link to this Table of Contents. This thesis template is retrieved from https://github.com/felixduvallet/ri-thesis-template.

List of Figures

2.1 A common workflow of modular pipeline for autonomous driving. Different methods may have slightly different workflows. 6

2.2 A common workflow of imitation learning pipeline for self-driving at test time. This figure is only for illustrative purposes. Different methods may have completely different workflows. 8

2.3 The workflow of reinforcement learning pipeline for autonomous driving at test time. This figure is for illustrative purpose only. Different methods may have different inputs and workflows. 9

2.4 An overview of a typical single-agent reinforcement learning (RL) pipeline. RL often involves an agent and an environment. An RL agent consists of an actor and a learner. The actor takes an action to interact with the environment. The environment returns new observations and rewards as feedback. The learner learns a better policy from such feedback data. 10

4.1 The hierarchy of the proposed distributed RL framework. It modifies and extends the standard parameter server architecture [55] with features specifically designed for the CARLA self-driving. There are also two levels of parallelizations, the inter-process level and the intra-process level, both of which run asynchronously. 28

4.2 An overview of the worker-level intra-process parallelization. Each worker is a single process. At worker level, it hosts multiple agents. Each agent has its own local policy copy that can be different from others This helps to maintain asynchronicity at this level. To alleviate communication overhead, agents does not sync with servers. Instead, they fetch local policy updates from workers. Their state transitions will also be gathered at worker's temporary buffer storage. This is similar to having a parameter server at worker-level. 28

4.3 An overview of the server-level inter-process parallelization. In order to share resources constantly, the server process utilizes multithreading and Each server is a thread. All communications are point-to-point. This is good for scalability and fault-tolerance. Servers handle different requests from workers as well as updating the global policy using the global buffer with value/Q function. 29
4.4 The simulation per-step time against number of agents in the environment. With more agents added into the environment, while the time for CARLA to render a frame is increasing, the average step time (i.e. interacting once) for each agent is decreasing until it rebounds. It is therefore concluded that the optimal number of agents on a single CARLA simulation environment is around 7 to 9. 31

5.1 The obstacle sensor array for retrieving obstacle affordances from all angles around of the ego-vehicle. Note that this is purely for illustrative purposes. Neither the sensing radii & ranges nor the sensor placements are drawn exactly. 45
5.2 The workflow of single-agent Proximal Policy Optimization (PPO) [79] algorithm, it features an actor-critic pipeline for policy and value function updates. This figure is only for illustrative purposes. Many components, such as trajectory buffer and importance sampling, are not shown for simplicity. 46
5.3 Different choices of update schemes for distributed on-policy RL. With the driving policy improving, on average, training episodes typically become longer. Option A chooses to update the global policy with a fixed short interval. This might be good initially but will harm the optimization in the long term by breaking up long episodes into several trajectories as well as making agents out-of-sync from the global policy. Option B chooses to update the global policy with a fixed, relatively long interval. This might be good in the long run but will slow down the initial training due to less frequent updates. Option C (Ours) chooses to update adaptively based on the length of training episodes, addressing above-mentioned problems. 47

List of Tables

3.1 Quantitative results on CoRL2017 [23] benchmark under different tasks. Numbers represent the average success rate (in percentage) out of all testing episodes. Refer to 4.4.2 for details about these methods. 20

3.2 Three traffic conditions defined in NoCrash [20] benchmark. 21

3.3 An overview of the CARLA Leaderboard driving route splits. Testing routes are held-out exclusively on the Leaderboard Challenge servers and their details are not publicly available. 22

4.1 Quantitative results on NoCrash [20] benchmark under empty, regular, dense traffic conditions (defined in 3.2.2). Numbers represent the average success rate (in percentage) out of 25 testing episodes. Details of all baseline methods are in 4.4.2. MPSAC stands for our method, Multi-Parallel SAC. MPSAC(1x) refers to our implementation of single-agent SAC [33]. MPSAC(std64x) refers to the 64-agent MPSAC using standard parameter server setup where locally computed gradients are used for the global update. 38

4.2 Approximate simulation timesteps per hour under different settings. MPSAC stands for our method, Multi-Parallel SAC. "Std. param. server" indicates whether the standard parameter server workflow is used (i.e. agents computing gradients w.r.t. local policies and sending them to servers) As is mentioned in 4.3.1, agents refer to individual actor and each worker can have multiple agents. The total number of agents in one training system is equal to the number of workers multiplied by the number of agents per worker. Due to the stochastic nature of reinforcement learning, the number of timesteps needed for one method to reach its peak performance is highly unpredictable. Even for the same method, total training time (defined as the time needed for a method to reach its peak performance) of multiple runs varies a lot. Thus, we report how many timesteps a given method can complete in an hour instead. Nonetheless, the reported profiling results are heavily dependent on cluster status. We acknowledge that our results are indicative and not ideal for quantitative comparison at a fine-grained level. 39

4.3 Sensitivity test of entropy regularization term α on NoCrash [20] benchmark under empty, regular, dense traffic conditions (defined in 3.2.2). Numbers represent the average success rate (in percentage) out of 25 testing episodes. All experiments use Multi-Parallel SAC with the same hyperparameter setup except for the α. 41

5.1 Quantitative results on NoCrash [20] benchmark under empty, regular, dense traffic conditions (defined in 3.2.2). Numbers represent the average success rate (in percentage) out of 25 testing episodes. Baseline methods are detailed in 4.4.2. MPSAC stands for Multi-Parallel SAC, whereas MPPPO stands for Multi-Parallel PPO. In addition, MPPPO(std64x) refers to the 64-agent MPPPO using standard parameter server setup where locally computed gradients are used for the global update. MPPPO(1K64x) refers to the 64-agent MPPPO with a fixed global update interval at 1,000 timesteps, whereas MPPPO(100K64x) refers to the 64-agent MPPPO with a fixed global update interval at 100,000 timesteps. 54

5.2 The impact on the training progress using different rollout lengths and update intervals. The experiment with the adaptive update interval does not use a fixed rollout length as previously described in 5.3.2. Comparing the average reward at the initial of the training (i.e. at 1K timesteps) with the average reward after 1M timesteps, the experiment with the adaptive update interval improves the most during this period of training, showing the effectiveness of our design over a fixed update interval with a fixed rollout length. 54

5.3 Approximate simulation timesteps per hour under different settings. MPSAC stands for Multi-Parallel SAC, whereas MPPPO stands for Multi-Parallel PPO. "Std. param. server" indicates whether the standard parameter server workfow is used (i.e. agents computing gradients w.r.t. local policies and sending them to servers). As is mentioned in 4.3.1, agents refer to individual actor and each worker can have multiple agents. The total number of agents in one training system is equal to the number of workers multiplied by the number of agents per worker. Due to the stochastic nature of reinforcement learning, the number of timesteps needed for one method to reach its peak performance is highly unpredictable. Even for the same method, total training time (defined as the time needed for a method to reach its peak performance) of multiple runs varies a lot. Thus, we report how many timesteps a given method can complete in an hour instead. Nonetheless, the reported profiling results are heavily dependent on cluster status. We acknowledge that our result are indicative and not ideal for quantitative comparison at a fine-grained level. 56
5.4 Leaderboard evaluation results. MPSAC stands for our proposed method, Multi-Parallel SAC, which in this case is trained with the 15-dim state space described in 5.2. MPPPO stands for our proposed method, Multi-Parallel PPO. The "Routes" column indicates how different methods are evaluated. Except for TransFuser AT and MPPPO, other methods are tested on held-out test routes. Hence, our method is only directly comparable with TransFuser AT. Descriptions of these metrics can be found in the leaderboard of the CARLA Leaderboard Challenge, the link to which can be found in 5.4.4. "DS" stands for driving scores. "RC" stands for route completion (in %) For convenience, we merge some official metrics into a single category. "Collis." combines all types of collisions in original leaderboard. "Viol." combines red light and stop sign violation, as well as off-road infraction. Infractions in collisions or violations indicate control or perception issues. "Dev." stands for route deviations, infractions in this category indicate planning issues lane departure issues. "Timeouts" combines the original timeout metric as well as situations when the agent gets blocked. The unit for all infraction-related metrics is infractions per kilometer. 58

A.1 General hyperparameters for all experiments. Note that for obstacle affordance in NoCrash experiments, only a front obstacle sensor is used. 64
A.2 Hyperparameters for Multi-Parallel SAC. 64
A.3 Hyperparameters for Multi-Parallel PPO. 64

Chapter 1

Introduction

Many advances in autonomous driving focus on a modular approach, where the whole task is divided into multiple subtasks such as perception, planning and control [12, 46, 54, 61, 63, 94]. While this paradigm has been performing well for typical traffic scenarios, it struggles to tackle out-of-distribution driving situations without heavily handcrafted special procedures for edge cases. To cope with this issue, reinforcement learning (RL) has come under the spotlight since autonomous driving can be naturally viewed as a trajectory optimization problem, where we need to find an optimal control over the course of driving. Empirical evidence suggests that RL methods are able to achieve this goal in an highly automated manner, without the trouble of manually handling challenging long-tail and rare cases. Their success has already been demonstrated in numerous decision-making tasks, such as playing strategy games or manipulating robots [8, 60, 74, 78, 79, 81, 88].

In spite of its huge potential, RL is not extensively studied in autonomous driving, mainly due to the safety and legal concerns. RL algorithms learn an optimal policy in a trial-and-error fashion, which is simply not allowed in the real world. Fortunately, with the emergence of simulators built for facilitating autonomous driving research [23, 27], utilizing RL methods to improve self-driving capabilities is finally made possible.

Among all categories of RL methods, online RL methods seem to be the most promising considering the help of the simulation environment. An online RL method explores & exploits the learning environment and receives feedbacks directly from the groundtruth dynamics. This allows an online RL method to create its own dataset on the fly, without the worry of biased data distribution on pre-collected datasets. Moreover, training online RL methods often involves random trial-and-error behaviors, which helps them explore rare and long-tail events that are not able to observe otherwise. Recently, several online RL methods have been proposed to handle the decision making process in an automated fashion for autonomous driving [1, 23, 85]. They build their pipelines on the CARLA simulator [23], one of the most widely used simulators in the self-driving community. The built-in complex traffic scenarios, along with the photorealistic rendering, make it ideal for training and testing novel self-driving algorithms. This is also our simulator of choice in this study. A detailed description of the CARLA simulator is in Chapter 3.

However, even with the help of the CARLA simulator, online RL methods for autonomous driving are still few due to its inefficiency. First, autonomous driving by itself is not a simple problem as opposed to the toy control problems (e.g. cartpole [6] or bipedal walker[1]) which are often used to benchmark RL algorithms. The continuous action space with wide ranges, the complex traffic scenarios and the presence of other dynamic traffic actors in the environment result in combinatorial number of total feasible states. This complicates both the exploration and the exploitation process for RL algorithms, meaning that a significant number of interactions with the simulator are inevitable for policy optimization. Moreover, to simulate high quality graphics and realistic environment physics, the CARLA simulator is computationally intensive and is noticeably more time-consuming than common environments used for benchmarks such as OpenAI Gym [10] or Mujoco [84]. Therefore, it is understandable that the time needed to train a self-driving agent on CARLA is orders of magnitude longer than to solve a simple benchmark problem. This makes training online RL algorithms on many CARLA tasks take weeks to months to finish. This discourages the use of online RL methods on autonomous driving.

[1]https://gym.openai.com/envs/BipedalWalker-v2/

One key observation is that those online RL algorithms [1, 23, 85] used for autonomous driving are straightforward adaptations of classic and generic RL methods. Though they work surprisingly well with properly designed state space for perceiving the surrounding environment dynamics, they poorly scale up to harder decision-making problems such as self-driving. To deal with the above-mentioned issue, it is necessary to accelerate the training of RL agents in simulation environments by introducing large-scale parallelization for online RL. To date, there are many out-of-box parallel or distributed RL algorithms [25, 26, 37, 41, 60, 65, 74, 83, 91, 95]. However, they are mainly designed for problems where the utilization of computational resources, the skewed distribution of workloads and the heterogeneity among agents participating in the training are not taken into consideration.

When it comes to parallelizing online RL methods on the CARLA simulator, following issues must be addressed. First, CARLA is resource intense, consuming ∼2GB GPU memory per simulator instance, not to mention the CPU resources it uses for simulation dynamics. This means parallelized methods for the CARLA self-driving should be capable of running across multiple machines in a distributed fashion. Second, current single-agent CARLA RL methods largely under-utilize a single simulator instance. Because single-agent algorithms only have one active ego-vehicle in the environment, at each simulation timestep it only interacts with its near surroundings while the vast majority of the environment is unseen and unused. To prevent the waste of computation, online RL algorithms for self-driving should be devised to parallelize multiple agents within the same environment instance. Third, self-driving as a goal-directed navigation problem has a high variability. The task durations and difficulty levels alternate throughout the training. We simply cannot design a multi-agent algorithm that waits for other agents before it can move to the next step. Thus, asynchronism is indispensable for multi-agent systems in this context.

In this thesis, we propose two distributed multi-agent RL methods, namely, Multi-Parallel SAC and Multi-Parallel PPO. They are designed with the philosophy of addressing all parallelization issues mentioned above. First, the parameter server infrastructure, which is at the heart of both methods, manages all communications via TCP sockets, allowing our training system to be distributed across different

GPUs & CPUs on different machines. Second, to make full utilization of every simulation step, our methods are designed with both inter-process and intra-process parallelization. Inter-process parallelization allows multiple CARLA environments to participate in the distributed training system, whereas intra-process parallelization allows multiple ego-vehicles to run concurrently in each CARLA instance. Third, our methods support asynchronicity across different processes. There are no collective synchronization throughout the training, hence no delays. This also contributes to the great scalability and fault tolerance of our proposed methods.

Although the backbones of our methods are two completely different generic RL algorithms (i.e. off-policy vs. on-policy), they share the same parameter server [55] infrastructure. We achieve this versatility by accommodating the needs of both types of RL algorithms when designing our distributed RL framework. On the one hand, off-policy RL requires the collected data to be shared across learners as well as a high throughput design that handles frequent policy updates. On the other hand, on-policy RL assumes improving a policy directly using data collected from the same policy, which requires that all training participants host similar policies at all times. Therefore, for off-policy RL, the focus is to increase concurrency and reduce communication overhead due to its frequent update interval. This is addressed by having worker-level parameter servers from the multi-level framework. For on-policy RL, the focus is to reduce the delay between local policies and global policies. We also devise an adaptive update interval mechanism to mitigate this issue.

To sum up, in this thesis, we introduce a distributed RL framework designed for the CARLA self-driving problem. We instantiate it with two representative RL methods, denoted as Multi-Parallel SAC and Multi-Parallel PPO. We would like to demonstrate that our parallelized methods have the capability of accelerating the online RL training on the CARLA simulator as well as achieving state-of-the-art performances on challenging CARLA self-driving tasks in the meantime.

Chapter 2

Background

2.1 Autonomous Driving

2.1.1 Modular Pipelines

Modular pipelines are one of the most popular autonomous driving methods. As is illustrated in Figure 2.1, these methods aim at dividing the autonomous driving task into multiple subtasks (i.e. submodules) that separately handle planning, control, perception, mapping and localization problems. [12, 46, 54, 61, 63, 94]

Modular pipelines typically start with sensor inputs and routing information, followed by a mapping and localization submodule which helps the ego-vehicle estimate its location and pose. [5, 9, 94]. After this stage, a dedicated perception system, also using a suite of sensor inputs, is used to perceive the surrounding environment, resulting in predictions such as lane detection, traffic sign recognition, obstacle detection or pedestrian behavior forcast [3, 5, 54, 94]. These perception predictions are subsequently used for motion planning. In modular systems, motion planners

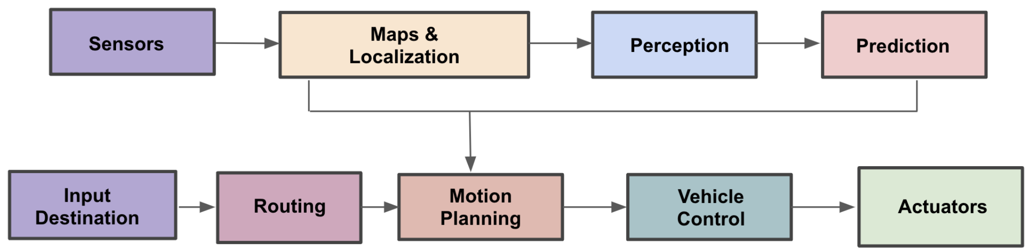

Figure 2.1: A common workflow of modular pipeline for autonomous driving. Different methods may have slightly different workflows.

are often used to predict optimal future trajectories and then invoke the control module to manipulate the actuators [22, 53, 54, 62]. Many recent works are also focused on improving individual subtasks. For example, Djuric et al. [54] propose an uncertainty-aware motion planning method via rasterized bird's-eye-view images. Gwon et al. [31] design an efficient high-definition road-map generation system with refined 3D road geometry data.

One obvious advantage of these modular systems is that they are intuitively designed in an interpretable way. On the other hand, being strictly modularized, it requires huge engineering efforts to implement and maintain system functionalities. Also, as different submodules are highly interdependent, the flexibility of changing or upgrading some of the submodules is doubtful. Furthermore, it seems the whole system works with lots of configurable parameters from each submodule. This combinatorial complexity could make it hard to tune.

2.1.2 Imitation Learning

Imitation learning methods attempt to learn a policy from collected expert data which maps input data to (pseudo-)groundtruth actions. As is illustrated in Fig 2.2, the input data typically consists of high-dimensional sensor inputs and additional low-dimensional inputs such as destination locations or directional commands. During the training stage the objective is to mimic an expert's behaviors based on the same input data used by the expert. Thus, for imitation learning methods, it is key to get a wide variety of expert demonstrations which covers most scenarios.

Recently, many imitation learning methods have been proposed for self-driving. Dosovitskiy et al. [23] directly learn a controller via imitation learning on perception features and speed measurements. Codevilla et al. [19] propose a conditional imitation learning algorithm that utilizes vision-based inputs as well as high-level command inputs as constraints, which makes imitation learning agent capable of responding to navigational commands at test time. Chen et al. [14] train a pure vision-based agent out of a privileged agent learnt from groundtruth data, both with imitation learning. There is also another family of imitation learning pipelines where imitation learning is combined with reinforcement learning (RL). Liang et al. [56] utilize visual and odometry inputs to train an imitation learning model for preliminary self-driving and then use online RL to finetune the model. Chen et al. [15] first train an offline RL agent using a dataset containing privileged information collected on the CARLA [23] simulator. They then treat the learnt agent as an expert driver and use knowledge distillation [38] with imitation learning to produce a sensorimotor agent.

One of the biggest challenges for imitation learning methods, not only on the self-driving problem, but on many others as well, is their poor generalizability. They often fail at situations that are unseen during the training [42]. Moreover, the learnt imitation learning policies may be biased since the distribution of an expert dataset is imbalanced with only having few or no negative examples at all [20, 42]. This leads to an undesired scenario where imitation learning agents cannot recover from unseen errors. Those small errors, accumulated over time, result in completely deviated trajectories [68]. Also, the performance of imitation learning, if not provided with additional help, is often capped by the expert's ability [11].

2.1.3 RL for Self-driving

Reinforcement learning (RL) methods view the autonomous driving problem as a long-trajectory decision-making problem which is solvable using various RL methods.

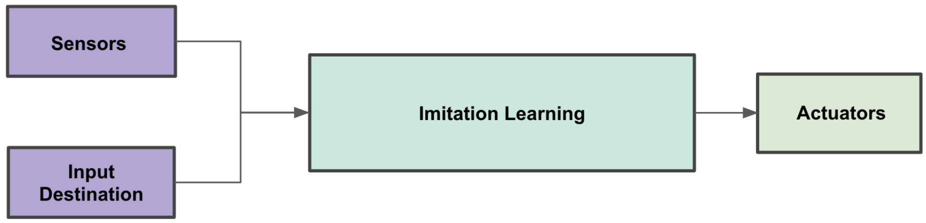

Figure 2.2: A common workflow of imitation learning pipeline for self-driving at test time. This figure is only for illustrative purposes. Different methods may have completely different workflows.

Different RL methods on self-driving may have different structures and training procedures. As is shown in Figure 2.3, the input used in RL methods typically involves information regarding current state of the environment, such as perception feedbacks, and goal-directing features such as short-term route planning information (e.g. the next waypoint or orientation).

Recently, more and more RL methods have been proposed to tackle self-driving problem. Mnih et al. [60] learn an A3C agent on TORCS [27] simulator using only RGB images as input. Dosovitskiy et al. [23], stack two image frames together with low-level measurements to learn their agent on CARLA. Kendall et al. [47] use VAE [50] to decode monocular RGB camera input along with speed and steering inputs, they train a DDPG [57] agent and demonstrate its capability on simple driving tasks. Toromanoff et al. [85] use a ResNet [36] that encodes vision inputs to implicit affordances. This results in a low-dimensional input for RL agent, thus making it easier for the agent to optimize on urban self-driving tasks. Chen et al. [15] explore offline RL on self-driving. They first collect an offline dataset on CARLA. Assuming a non-reactive world model and a low-dimensional and compact forward model of the ego-vehicle, a tabular Q-learning is performed using dynamic programming. They then use knowledge distillation [38] to train a visuomotor policy to avoid the use of privilege information. Agarwal et al. [1] choose a set of CARLA affordances to simplify the state space in their RL setup. They demonstrate its effectiveness with different online RL methods on CARLA.

As is empirically shown in aforementioned studies, formulating autonomous driving

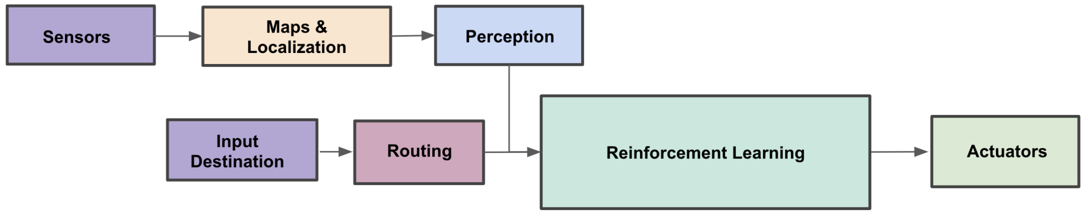

Figure 2.3: The workflow of reinforcement learning pipeline for autonomous driving at test time. This figure is for illustrative purpose only. Different methods may have different inputs and workflows.

as an RL problem has the following benefits.

- Self-driving is naturally a sequential decision-making problem. RL is specialized to solve this kind of trajectory optimization problems.
- Dense rewards and modest time horizons enable RL to make drastic changes and quickly respond to driving situations.
- RL has the potential to explore long-tail events and learn from such experiences.

Also, when compared with modular approaches, which requires handcrafting every rule for every possible edge case, RL being highly automated makes it easier to train and deploy as a fully functional self-driving system. In addition, when compared with imitation learning, RL often showcases better generalizability and its performance is not upper-bounded by a demonstrator's performance.

2.2 Reinforcement Learning

2.2.1 Preliminaries

Reinforcement learning (RL) is widely used to solve trajectory optimization problems. Generally, RL algorithms involve generating samples by running a policy, estimating the return of that policy and improving the policy accordingly. An anatomy of general

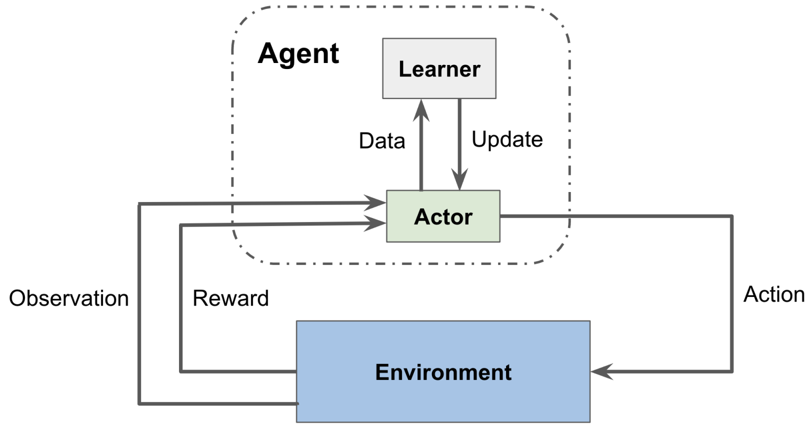

Figure 2.4: An overview of a typical single-agent reinforcement learning (RL) pipeline. RL often involves an agent and an environment. An RL agent consists of an actor and a learner. The actor takes an action to interact with the environment. The environment returns new observations and rewards as feedback. The learner learns a better policy from such feedback data.

RL algorithms is illustrated in Figure 2.4. RL algorithms are typically made up of an agent taking actions in an environment, which provides observations and rewards as the feedback in response to such actions.

The RL environment is generally treated as a Markov decision process (MDP) [7], which can be represented as $(\mathcal{S}, \mathcal{A}, p, \mathcal{R}, \gamma)$, where $\mathcal{S}$ denotes the state space, $\mathcal{A}$ denotes the action space. $p(s'|s, a)$ represents the transition probability from an individual state s to s' given action a, where $s, s' \in \mathcal{S}$ and $a \in \mathcal{A}$. $\mathcal{R}$ is the reward function and $\gamma \in [0, 1]$ is the discount factor.

Denote a sequence of rewards along a trajectory τ as

$$r_\tau = \{r_0, r_1, \ldots, r_T\}. \tag{2.1}$$

The discounted reward at step t can be given as

$$R_t = \sum_{i=t}^{T} \gamma^{t-i}\, r_i. \tag{2.2}$$

The goal of RL is to find a policy that maximizes the expectation over the total discounted cumulative reward,

$$\mathbb{E}\left[R_0\right] = \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t \, r_t\right]. \tag{2.3}$$

2.2.2 Off-policy vs. On-policy RL

RL algorithms can be categorized into off-policy methods and on-policy methods. Off-policy algorithms evaluate and improve a target policy that is different from the one they used to explore the environment and generate experiences. Popular methods in this category are most of Q-learning family (DQN [59], Dueling DQN [89], Double DQN [86]), off-policy actor-critic methods such as SAC [33, 34], and off-policy policy gradients methods (DPG [80], DDPG [57], TD3 [29]). These methods often collect a large pool of state transitions (i.e. experiences) formulated as (s, a, s', r, d) where d denotes the termination state indicator. At the time of update they randomly sample a batch of transitions to improve their policy. These methods often have explicit exploration strategies such as ϵ-greedy [59].

On the other hand, on-policy methods directly improve a target policy on top of the current policy. Popular methods in this category include on-policy temporal-difference learning variants such as SARSA [75], on-policy actor-critic methods (AC [51], A2C & A3C [60]) and on-policy policy gradients methods (REINFORCE [92], TRPO [78], PPO [79]) Many of them make use of long-term cumulative rewards instead of learning on the temporal difference. This may be helpful for long trajectory optimizations.

2.2.3 Model-free vs. Model-based RL

Most online RL methods are model-free by design, meaning that they do not assume access to latent environment dynamics and thus cannot know the impact of their actions beforehand. This gives them greatly flexibilities on problems in different domains. Model-based RL algorithms (World Models [32], I2A [90], MBMF [64], MBVE [28], AlphaZero [81]), however, taking advantage of the groundtruth model behind the environment, have the capability of predicting the next states and rewards in the environment and planning their actions accordingly.

2.2.4 Online vs. Offline RL

Most common RL methods are online RL methods. They gather their experiences by constantly exploring and exploiting in an interactive environment from which they get live feedbacks on their actions and information regarding their status in the environment. These feedbacks and updates are exact as they come from the environment dynamics. However, in offline RL, we lose such direct access to the groundtruth environment. Instead, a fixed dataset is used in offline RL and consists of trajectories collected by some behavioral policy in the environment. Offline algorithms subsequently learn a policy based on the limited data with no interactions with the environment they will be deployed upon. There is a wide variety of offline RL algorithms, both model-free (e.g. CQL [52], Decision Transformer [16]) and model-based (e.g. MOReL [48], MOPO [93], Trajectory Transformer [44]).

2.2.5 Parallel & Distributed RL

Large-scale parallel and distributed RL methods [8, 25, 26, 37, 40, 41, 60, 65, 81, 83, 88, 91, 95] are essential for solving challenging trajectory optimization problems where the complexity of those environments is orders of magnitude higher than toy problem environments typically used by RL community to benchmark RL algorithm performance. These methods aim for extending the existing single-agent RL optimization process to a multi-agent training system with a pool of processes working on the same optimization objective.

One of the major branching points of those methods is about the way they handle the synchronization. Synchronous RL methods sync their policy or local progress, fully or partially, across participating processes at all times (or most of times) in the course of the training. It is usually done by collective synchronization operations such as all-reduce or broadcast. Such synchronous methods suffer greatly in terms of the scalability and the stability of such systems. With the number of processes in a training system growing, the communication cost for synchronization also grows linearly. The fault tolerant ability of such system is also questionable as one failed process may block the synchronization, thus blocking the whole system.

Asynchronous RL [40], on the other hand, refers to methods that do not require collective synchronizations to maintain the uniformity or synchronicity across processes. Hence, they do not have the above-mentioned issues. In order to implement this idea, they often have an asymmetrical design where the training procedure is divided into two tasks, namely, data collecting and learning. Data collectors (i.e. agents) are only responsible for collecting rollouts in the environment whereas a learner, such as a parameter server [55], is used to update the policy using collected data. All communications and data transfers are point-to-point, which are good for scalability and overall fault tolerance of the training system.

Off-policy methods, since they learn from other policies by design, it is relatively intuitive to parallelize them via asynchronism. The majority of the existing asynchronous multi-agent RL algorithms are from the off-policy family. Nair et al. [65] propose a distributed DQN framework named GORILA which uses parameter servers to coordinate actor-learner synchronization. To improve the reliability, servers are able to deny and drop outdate gradient updates from learners. Horgan et al. [41] design a distributed DQN algorithm called Ape-X DQN using a distributed prioritize replay buffer [77] to improve the training efficiency.

Unfortunately, most on-policy methods achieve massive parallelization with synchronism, or they tend to break the on-policy learning practice by either optimizing on trajectories from other policies or truncated trajectories with compromise cumulative rewards. Zhang et al. [95] design a synchronous PPO framework named MDPPO by mixing up trajectories from different policies, which essentially converts PPO algorithm to off-policy methods. Wijmans et al. [91] propose a decentralized distributed PPO algorithm that asynchronously collects data while synchronously updating the policy. To reduce synchronization wait time, it forcibly terminates some longer-than-other episodes. This potentially makes some scenarios never get learnt. Stooke et al. [83] propose both synchronous and asynchronous methods for PPO policy updates. However, the asynchronous method, namely APPO, requires frequent breakup of trajectories (e.g. every 64 timesteps), resulting in inaccurate cumulative rewards which weaken the performance. Moreover, Mnih et al. [60] propose an on-policy actor-critic method called Asynchronous Advantage Actor-Critic (A3C). However, the model is put in shared memory and is updated via Hogwild [72], thus remaining synchronized throughout the training. Due to this, it can only be deployed on a single machine. Whether this can be counted as a bona fide asynchronous RL algorithm remains a question. Hence, Espeholt et al. [25] try to parallelize the A3C method via V-trace correction during updates, allowing training across multiple machines.

Recently, there is also a trend among large-scale RL methods that they often come with a task-specific design to fully consider the characteristics of the task environment. Silver [81] propose a board game agent, namely AlphaZero, which utilizes Monte Carlo tree search (MCTS) with different search spaces for chess, shogi as well as Go. Vinyals et al. [88] train an RL agent called AlphaStar on the real-time strategy game Starcraft II [1]. To deal with the complexity of this game and its adversarial nature, an RL pipeline including self-imitation learning and game theory is specifically designed for solving this problem. Berner et al. [8] train an RL agent to play Dota 2 [2] with competitive self-play and hero-specific observation spaces and features.

[1] https://starcraft2.com/en-us/
[2] https://www.dota2.com/home

Chapter 3

Simulation Environment

3.1 CARLA Simulator

3.1.1 Introduction

For online Reinforcement Learning (RL) algorithms, learning to drive requires a large number of samples, which are only possible in simulators. In this study, we choose to use one of the most popular urban driving simulators, namely the CARLA simulator [23]. CARLA is open-source and is built upon Unreal Engine [24] which features photorealistic graphics, as well as PhysX[1] which performs realistic physics simulation. To facilitate autonomous driving research, CARLA Python-API has a full suite of tools, including pre-defined urban driving maps, common perception sensors and configurable environment dynamics. CARLA is also used as a performance evaluator for autonomous driving, hosting a wide variety of self-driving benchmarks, including CoRL2017 [23], NoCrash [20] and CARLA Leaderboard[2]. In

[1]https://developer.nvidia.com/physx-sdk
[2]https://leaderboard.carla.org/

conclusion, CARLA provides autonomous driving researchers an all-in-one platform for developing, training and evaluating their self-driving systems and comparing them in a standardized simulation environment.

3.1.2 Sensor Affordances

One of the benefits of using the CARLA simulator is its comprehensive sensor library, such as IMU, GNSS, LIDAR and various semantic or RGB cameras. In self-driving systems, those sensors provide measurements of ego-vehicle's current status as well as building up inputs for ego-vehicle's perception system which are crucial to most autonomous driving solutions. Like other methods, our parallel RL methods also take advantage of these sensors in the observation space of our agent. Here are the details of relevant sensors used in our methods.

IMU sensor. The inertial measurement unit (IMU) in CARLA measures linear acceleration (in m/s^2), angular velocity (in rad/sec) and orientation (in radians, where north is $[0.0, -1.0, 0.0]$) of the ego-vehicle. This sensor is ticked per simulation step. After receiving IMU readings, we are able to keep track of the ego-vehicle odometry as well as its current velocity.

GNSS sensor. The Global Navigation Satellite System (GNSS) sensor in CARLA provides the raw latitude, longitude and altitude information of the ego-vehicle every timestep. After getting the raw geolocation coordinates, we can compute local projected coordinates which are used by many CARLA APIs following the data conversion rules in World Geodetic System 1984 (WGS 84) [66]. For this step, we use `pyproj`[3] package to do the projection.

Lane invasion detector. This sensor in CARLA detects the crossing of various lane markings, such as solid yellow lines, broken white lines or crosswalk curbs. This helps us determine whether the ego-vehicle has made lane changes during an episode. Based

[3]https://github.com/pyproj4/pyproj

on this information, we can detect lane invasion infractions such as deviating from the designated lane or driving onto sidewalks. In all our experiments, we terminate the current episode once a lane invasion event is detected.

Collision detector. This sensor in CARLA detects collisions between ego-vehicle and other objects in the driving environment, including static objects such as buildings, bushes, poles as well as dynamic objects such as pedestrians and other vehicles. Note that several collisions may be detected at a single timestep. During training, we use the collision sensor to determine whether the ego-vehicle has hit something and if so we will terminate the training episode. We terminate an episode and count it as a failure at test time if any collision infraction is detected.

Obstacle detector. This sensor in CARLA detects any obstacle that is in its capsular-shape sensing range (sphere tracing in Unreal Engine[24]). It is capable of tracing both static and dynamic objects. The radius and maximum sensing proximity is configurable. Once some actors are within the specified threshold, it will report the obstacle types as well as the distances between them and the ego-vehicle. This is of a great assistance to carry out self-driving experiments without the trouble of setting up a vision-based perception system.

3.1.3 Groundtruth Affordances

Besides using sensors, we can directly query useful groundtruths pertaining to the world dynamics of the CARLA simulator and incorporate it into the feature space of our agent. These groundtruths are often referred to as privilege information. The only reason that they are directly accessible is that the CARLA simulator has such features. In real-world self-driving applications, we are unable to have privilege information at hand without the use of sensors and corresponding perception algorithms.

Since we are focused on solving the decision-making problem for self-driving (i.e. controlling the steering and the throttle of the ego-vehicle), we would like to access useful affordances to simplify the self-driving problem in simulation settings. This is a reasonable act as all the affordances we use in our methods are well-studied and solved problems in the real world. They are just not the focus of our study.

Traffic light affordance. We assume full access to traffic light information including knowing which traffic light is currently affecting the ego-vehicle and its state (i.e. red, yellow or green). In real-world driving environment where direct access to this information is infeasible, traffic light detection problem has been tackled using object detection framework with high definition maps [21, 39].

Stop sign affordance. Similar to the traffic light affordance, we assume full access to which stop sign is currently affecting the ego-vehicle. To further simplify this problem, in our implementation we directly convert a stop sign to a timed red light, absorbing it into the traffic light affordance in our feature space. In real-world driving environment where direct access to this information is infeasible, this is actually a special case of the traffic sign recognition problem and it has been well addressed by modern vision-based recognition framework [30, 45].

Dense waypoints. A waypoint is a CARLA 3D vector object consisting of its location on the map as well as its orientation. The ego-vehicle is supposed to drive along waypoints and follow the direction its nearest waypoint specifies. CARLA provides users with a groundtruth waypoint system that covers every lane on every road in the simulation environment. They are critical for navigating the ego-vehicle from one location to another on the map. In real-world self-driving solutions, the high definition map often contains similar features for navigation [31, 58].

3.1.4 Planners

There are two types of route information inputs in the CARLA simulator, waypoints or waypoint-command pairs, which contain high-level navigational commands, such as “go straight”, “turn left” or “move one lane to the right”. Thus, we mainly have two types of planner for solving the route planning problem (i.e. finding a path from the origin to the destination at each episode).

Waypoint planner. This planner takes two waypoints as input, representing the starting location and the destination location, correspondingly. The waypoint planner then uses a heuristic-based A^* search [35] algorithm that determines the optimal trajectory between them at a configurable resolution. The planner returns a detailed waypoint list and navigates the ego-vehicle via nearest waypoint. This planner is the default planner provided by CARLA [23].

Waypoint-command planner. Waypoint-commands are introduced by CARLA Leaderboard Benchmark (see details in 3.2.3). They take the format of waypoint-command pairs where each waypoint is followed by a verbal command indicating the desired moving direction of the ego-vehicle (e.g. left turn, right turn, straight, etc.). This planner takes multiple waypoint-commands as input. The waypoint-command planner will break them into multiple straight driving sections and concatenate these sections based on high-level commands. Within each section the high-resolution waypoints are generated using the waypoint planner mentioned above. In the end, the waypoint-command planner returns a concatenation waypoint list with proper order. Since we include dense waypoint affordance in the state space of our methods, in all of our Leaderboard experiments we do not use waypoint-command planner. Instead, we query dense waypoint affordance from Leaderboard and use the waypoint planner instead.

3.2 CARLA Benchmarks

Like many other algorithms, the performance of imitation learning and reinforcement learning (RL) algorithms is highly dependent on trajectories (i.e. datasets) that those methods are optimized on. In order to standardize the training and evaluation of imitation learning or RL self-driving methods, many benchmarks are created on the CARLA simulator. These benchmarks act like “datasets”. They set up training and

	CIL	CIRL	CAL	CILRS	LBC	IA	ARL
Straight	97	100	93	96	100	100	100
One Turn	59	71	82	84	100	100	100
Navigation	40	53	70	69	98	100	100
Dynamic Navi.	38	41	64	66	98	98	100

Table 3.1: Quantitative results on CoRL2017 [23] benchmark under different tasks. Numbers represent the average success rate (in percentage) out of all testing episodes. Refer to 4.4.2 for details about these methods.

testing trajectories with pre-defined routes at pre-defined traffic densities.

However, unlike datasets of many other learning-based tasks, in an RL task, data samples extracted from the same trajectory may still be different at each run. This is due to the stochasticity in both the environment that hosts this task as well as the RL exploration process itself. Therefore, in the CARLA simulator, different runs on the same trajectory may give different results due to aforementioned stochastic factors. During training time, this is actually helpful as it naturally encourages data augmentation and creates more traffic scenarios. At test time, however, this introduces instabilities and it is suggested that we run the evaluation multiple times to reduce the variance.

3.2.1 CoRL2017 Benchmark

The CoRL2017 benchmark [23], also known as the Original CARLA Benchmark, proposes four evaluation tasks: *Straight*, *One Turn*, *Navigation*, *Dynamic Navigation*. Testing routes of each task are defined using dense waypoints from source locations used to spawn the ego-vehicle to destinations. As is suggested in their names, *Straight* task consists of routes that only needs driving straight, whereas in *One Turn* task every destination is one turn away from the source location. To make it more difficult, each route in *Navigation* task has multiple intersections and is longer than routes of previous tasks. *Dynamic Navigation* task further adds dynamic actors (i.e. other vehicles) into the testing environment.

Task	# of vehicles at training	# of vehicles at testing
NoCrash Empty	0	0
NoCrash Regular	20	15
NoCrash Dense	100	70

Table 3.2: Three traffic conditions defined in NoCrash [20] benchmark.

However, this benchmark only examines whether the ego-vehicle can successfully reach the destination. It does not consider infractions as failure cases. The naive success criteria make this benchmark a little unrealistic and too simple. Due to such setup, this benchmark is considered solved. As is shown in Table 3.1, LBC [14], IA [85] and ARL [1] are able to achieve near-full or full success in all testing tasks. Therefore, in this study, we would like to focus on improving our RL methods on other benchmarks which are more challenging.

3.2.2 NoCrash Benchmark

Since the CoRL2017 benchmark (described in 3.2.1) fails to address infractions, which include traffic light violations, lane invasions or collisions, it is not capable of accurately measuring the driving performance of self-driving algorithms realistically. It also over-simplifies the self-driving problem because of the lack of dynamic actors in most of its tasks. Thus, the NoCrash benchmark [20] is proposed to fix this problem by identifying collisions as failures. This benchmark sets up training in CARLA *Town01* and evaluates the agent's driving behaviors on 25 goal-directed navigation routes in *Town02*. It also configures three traffic density levels, from roads being empty to dense congestions during test time by adjusting the number of dynamic actors. Specific settings are shown in Table 3.2.

If the ego-vehicle can navigate itself from the source to the destination without hitting other objects, including both dynamic actors and static environmental objects, the success criteria are met. However, infractions other than collisions, such as traffic light violations or lane invasions, are still not addressed in the NoCrash benchmark.

	Open-access	# of routes	Maps
Devtest	Yes	4	*Town01*, *Town03*, *Town04*, *Town06*
Training	Yes	50	*Town01*, *Town03*, *Town04*, *Town06*
Evaluation	Yes	26	*Town02*, *Town04*, *Town05*
Test	No	Unknown	Unknown

Table 3.3: An overview of the CARLA Leaderboard driving route splits. Testing routes are held-out exclusively on the Leaderboard Challenge servers and their details are not publicly available.

3.2.3 Leaderboard Benchmark

To further benchmark driving proficiency, a more realistic and complex benchmark called CARLA Leaderboard [4] is introduced. This benchmark consists of goal-oriented urban driving dynamics across multiple CARLA towns, with both highway driving (i.e. driving on controlled access roads) and city driving scenarios. Moreover, it adds ten NHTSA[5] pre-crash scenarios that reproduce real-world driving risks. Those scenarios can be summarized as below.

- recovery from control loss
- emergency brake for objects
- lane changing
- lane merging onto highway
- negotiations at traffic intersections
- handling traffic lights and signs

Many of them (e.g. lane change, lane merge, coping with pedestrians, etc.) require dealing with dynamic objects in the environment, which significantly increases the level of difficulty. When it comes to the evaluation, instead of only indicating the success or failure of a testing episode, the Leaderboard benchmark calculates a driving score for each episode from multiple aspects. These include infractions (collisions, traffic light or sign infractions, lane invasions and out-of-road infractions), route completion, scenario-specific penalties and overall driving time. The final driving score is also more comprehensive compared with the NoCrash benchmark. Instead of just focusing on success rate at test time, the leaderboard score is evaluated from

[4] https://leaderboard.carla.org/
[5] https://www.nhtsa.gov/

multiple aspects, including the number of collisions, the number of infractions, overall route completion rate, etc.

As is shown in Table 3.3, CARLA Leaderboard provides publicly available devtest, training and evaluation routes. However, the testing routes are held out privately and can only be accessed by participating in the CARLA Leaderboard Challenge[6].

[6]https://leaderboard.carla.org/challenge/

Chapter 4

Multi-Parallel SAC

4.1 Overview

In this chapter, we propose a multi-agent RL algorithm, namely, Multi-Parallel SAC, to efficiently handle the complex urban autonomous driving problem. We set up our RL problem formulation following [1] and achieve the state-of-the-art performance on CARLA NoCrash[20] benchmark in a timely manner. Multi-Parallel SAC is a distributed off-policy reinforcement learning (RL) method that is meant to significantly accelerate the training of online RL on the CARLA simulator. It consists of two levels of parallelization. For inter-process parallelization, it utilizes the parameter server architecture [55] to split the workload and assign different tasks on different processes. The two main components of this multiprocessing architecture are servers and workers. To avoid single-point failure, workers run asynchronously during the entire training. We use the PyTorch[1] [69] built-in Gloo[2] backend for the inter-process point-to-point communication. To further improve the utilization of a single CARLA instance, a multi-agent subsystem is used to achieve intra-process

[1]https://pytorch.org/docs/stable/distributed.html
[2]https://github.com/facebookincubator/gloo

parallelization.

4.2 Problem Setup

RL algorithms generally prefer low-dimensional features and actions to high-dimensional ones as low-dimensional inputs and outputs require less exploration to cover most of the feasible states. Hence, we choose to follow [1] by using navigational sensor inputs (IMU & GNSS), affordances (i.e. privilege information) in the CARLA simulator (discussed in 3.1) to formulate low-dimensional vector representations for this RL problem. We also follow their design of reward function with minor adaptations.

State space. The vectorized state space, $\mathcal{S}$, can be expended into a 7-dim vector, denoted as

$$\mathcal{S} = \begin{bmatrix} \tilde{w}_\theta & o_v^{\mathrm{f}} & o_\Delta^{\mathrm{f}} & e_\theta & e_v & \tilde{e}_\delta & \tilde{r} \end{bmatrix}^T, \tag{4.1}$$

where terms with a tilde indicate they come from affordances whereas terms without a tilde are from sensors. Specifically, $\tilde{w}_\theta$ is the average orientation (in radians) of next 5 waypoints, o_v^{f} and o_Δ^{f} are front obstacle speed and distance w.r.t. the ego-vehicle, respectively. e_θ and e_v refers to the current steering angle and current speed of the ego-vehicle, which can be inferred from the IMU sensor. $\tilde{e}_\delta$ stands for the lane center deviation (in meters) of the ego-vehicle from waypoints. $\tilde{r}$ indicates the status of restrictions (i.e. traffic light affordance and stop sign affordance) imminent ahead. We do not include the distance to the goal described in [1] in our state space as in preliminary experiments we do not find it helpful.

Action space. To simply the vehicle control problem, we refer to [1] to define a 2-dim vectorized action space, $\mathcal{A}$, consisting of the target speed of the ego-vehicle, $\hat{e}_v$, as well as the predicted steering angle of the ego-vehicle, $\hat{e}_\theta$. In our settings, it is possible that our policy network outputs a negative $\hat{e}_\theta$. We interpret such negative $\hat{e}_\theta$ as the full braking of the ego-vehicle since the ego-vehicle does not reverse in our settings. In all experiments, we clip $\hat{e}_\theta$ to a range of $[-0.5, 0.5]$ and clip $\hat{e}_v$ to $[0, 25]$

km/h. We then use a longitudinal PID controller [4, 73] to infer the adjustment needed on the actuator, subsequently configuring the throttle and the brake in order to reach the predicted target speed $\hat{e}_v$ given e_v.

Reward function. For the reward function, $\mathcal{R}$, we imitate [1] and formulate it as

$$\mathcal{R} = \alpha_v R_v + \alpha_\theta R_\theta + \alpha_\delta R_\delta + \alpha_I R_I, \tag{4.2}$$

where α terms are positive coefficients. R_v is the reward for the speed, which can be both positive or negative depending on situations. R_θ, R_δ and R_I are negative penalties for steering angles, lane center deviation and infractions, respectively. We do not clip the reward during the training.

4.3 Method

Overall, Multi-Parallel SAC can be disintegrated into two parts, the distributed training framework via the parameter server structure [55] and the Soft Actor Critic (SAC) [33] algorithm that powers each agent in such training framework.

4.3.1 Distributed Framework

The overview of the asynchronous distributed RL framework used in Multi-Parallel SAC is illustrated in Figure 4.1. It is mainly built upon a parameter server framework. At the inter-process level of parallelization, shown in Figure 4.3, two sets of processes are assigned with two different tasks. They are called servers and workers, correspondingly. Servers refer to processes which maintain a shared copy of latest policy. On the other hand, workers refer to processes which have their local copies of policies that are periodically sync-ed with servers by requesting policy parameter updates. This makes sure that no synchronization happens among workers, reducing

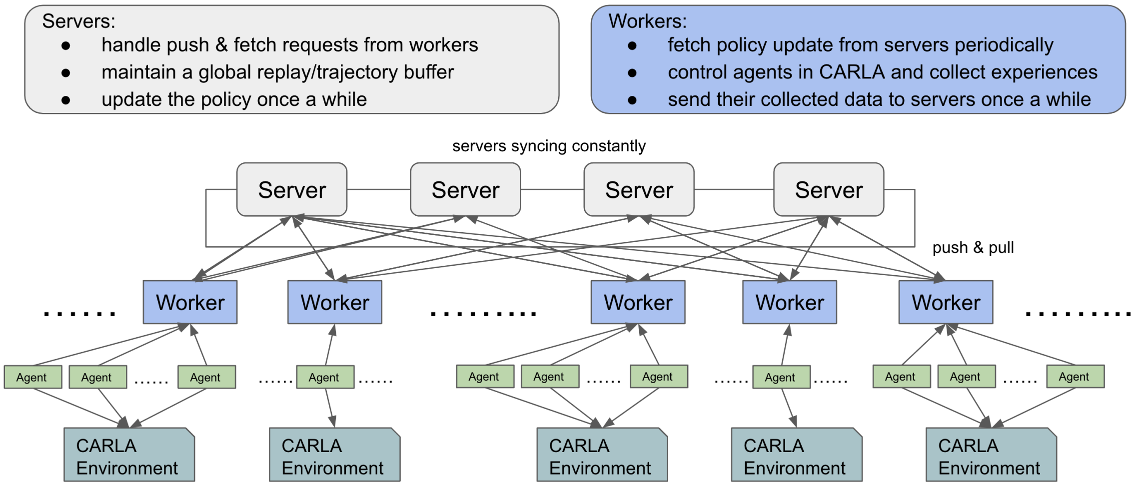

Figure 4.1: The hierarchy of the proposed distributed RL framework. It modifies and extends the standard parameter server architecture [55] with features specifically designed for the CARLA self-driving. There are also two levels of parallelizations, the inter-process level and the intra-process level, both of which run asynchronously.

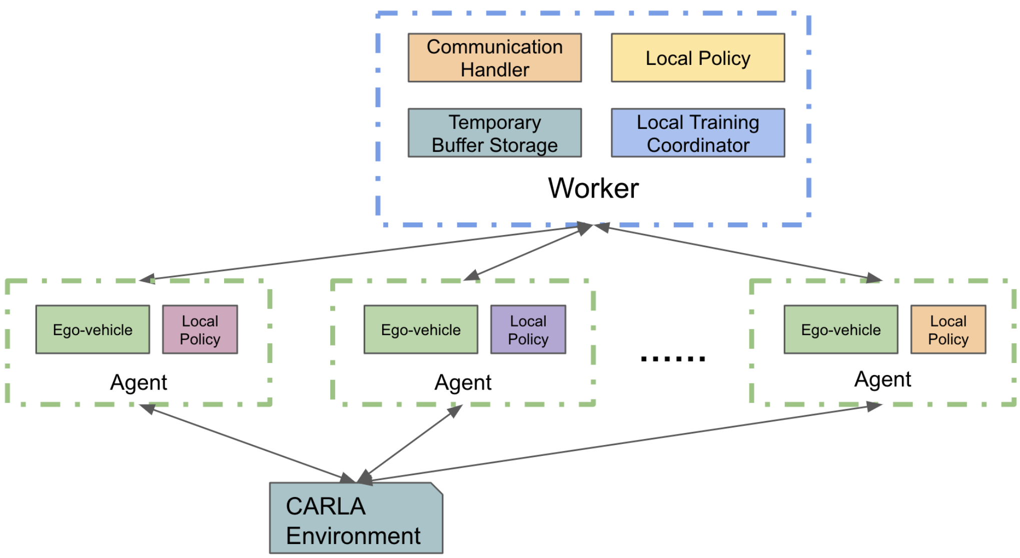

Figure 4.2: An overview of the worker-level intra-process parallelization. Each worker is a single process. At worker level, it hosts multiple agents. Each agent has its own local policy copy that can be different from others This helps to maintain asynchronicity at this level. To alleviate communication overhead, agents does not sync with servers. Instead, they fetch local policy updates from workers. Their state transitions will also be gathered at worker's temporary buffer storage. This is similar to having a parameter server at worker-level.

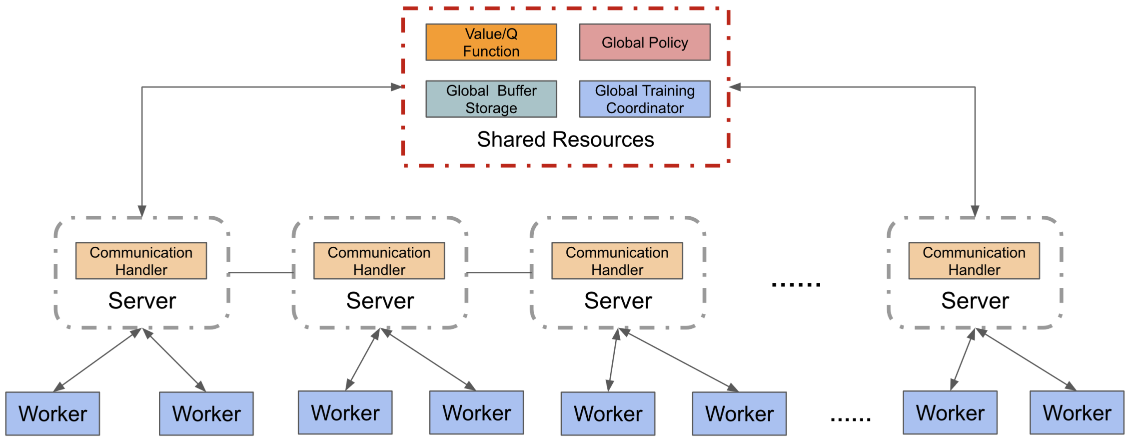

Figure 4.3: An overview of the server-level inter-process parallelization. In order to share resources constantly, the server process utilizes multithreading and Each server is a thread. All communications are point-to-point. This is good for scalability and fault-tolerance. Servers handle different requests from workers as well as updating the global policy using the global buffer with value/Q function.

the possibility of single-point failure when the number of workers is significant in distributed RL applications. In our setup, each worker has its own CARLA instance as the RL environment. Servers do not interact with the environment themselves. Instead, they interact with workers to handle push and fetch requests from workers.

In order to minimize the computation and communication cost, we substantially modify the original parameter server framework in [55] for this CARLA self-driving problem via the following approaches. First, in the original parameter server framework, local policy gradients are computed on the worker side and workers are supposed to push the gradients to servers, where servers apply them to update the global policy. There are many issues raised by following this practice on this specific RL problem. 1) As we setup our problem using low-dimensional state space and action space, the size of state transitions are significantly smaller than the size of gradients. It is relatively inefficient for workers to send out gradients to servers, unnecessarily increasing the overall communication cost. 2) Collecting locally computed gradients to update a global policy does not make sense in RL as local data pools are non-i.i.d. since they are generated from unique experiences of individual agents. This results in gradients being highly biased. 3) Due to the nature of trial-and-error process, RL methods generally suffer high instability and generate noisy gradients, especially when it comes to a single local agent exploring complex problems such as urban self-driving. This directly impacts the training progress as some of the out-of-distribution local gradients may poison the global policy. 4) One of the bottlenecks of training an RL agent on the CARLA simulator is that the simulator needs noticeable amount of time to render each timestep. In our framework it is the workers that interact with simulators, which keeps them highly occupied throughout the training while servers are not busy at all. In this case, the additional computation for workers to generate gradients via backpropagation will make this workload distribution further asymmetrical. Thus, in Multi-Parallel SAC, we decide to let servers maintain a global experience replay buffer and draw batches of state transitions to carry out policy updates throughout the training following the rules defined in single-agent SAC [33] algorithm. Workers, consequently, only need to collect state transitions by exploring their simulation environments and periodically push them to the centralized buffer hosted on servers, which reduce the computation burden on workers.

Another unique problem we pay attention to in Multi-Parallel SAC is how to max out the utilization of resources on every single CARLA simulator instance. We have already created an inter-process level parallelization by previously mentioned parameter server framework. However, we have not addressed the intra-process efficiency of each worker. Without the parallelization at intra-process level, we can only have a single ego-vehicle (i.e. a single RL agent) in each CARLA instance. At each timestep the ego-vehicle only interacts with its near surroundings while other places rendered by the simulator are simply ignored. This wastes a lot of computations and GPU resources. As is illustrated in Figure 4.4, it takes around $0.08s$ for CARLA to render one step (i.e. for all agents in the same environment to interact with the environment once) when there is only one agent in the simulator. Compared with having 8 agents in a single CARLA environment, although the per-step time is increased to around $0.20s$, the total number of interactions made during that CARLA step is 8. On average, it takes only around $0.025s$ to interact with the environment once. This significantly improves the utilization rate of each CARLA instance. Based on the diminishing return shown in Figure 4.4, having roughly 8 agents in one worker will make full use of this intra-process parallelization. As is depicted in Figure 4.2, to

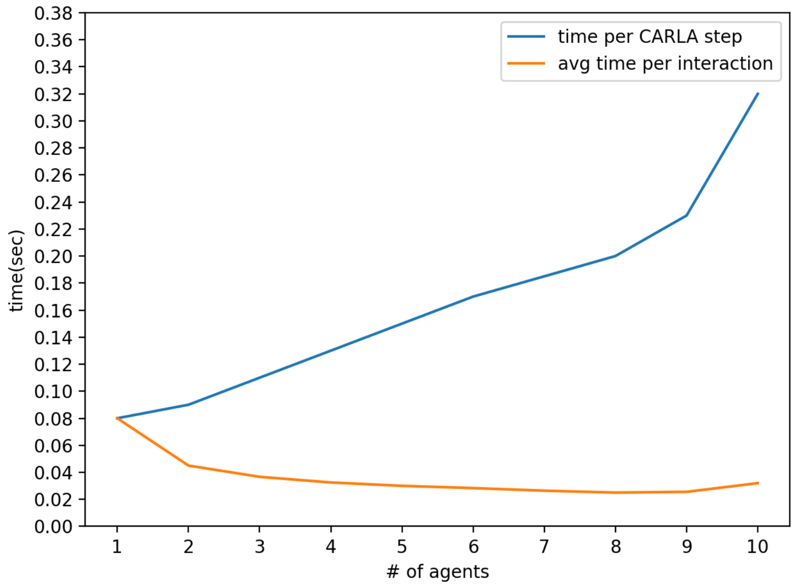

Figure 4.4: The simulation per-step time against number of agents in the environment. With more agents added into the environment, while the time for CARLA to render a frame is increasing, the average step time (i.e. interacting once) for each agent is decreasing until it rebounds. It is therefore concluded that the optimal number of agents on a single CARLA simulation environment is around 7 to 9.

avoid intra-process level agent-server or agent-worker synchronization, workers now maintain a separate policy for each agent and will make fetch & push requests once an agent demands. Since CARLA renders every simulation synchronously, at this level, all agents run sequentially but use individual policies, which is still considered to be (partially) asynchronous in terms of how the RL algorithm executes.

4.3.2 Multi-agent SAC

At the core of this self-driving trajectory optimization pipeline is the SAC [33] algorithm. While this is not the focus of this study, for completeness, we will briefly demonstrate how SAC works in Multi-Parallel SAC hierarchy. The main difference is that because workers only periodically (i.e. not every timestep) fetch the up-to-date global policy with servers and there is no synchronization among workers, asynchronicity is thus introduced into the system. Presumably, there will be multiple SAC policies coexisting throughout the training.

To represent this multi-policy system, we define a totally ordered set of policies as

$$(\pi_\Theta, \leq) = \{\pi_\theta^{(0)}, \pi_\theta^{(1)}, \pi_\theta^{(2)}, \dots\} \tag{4.3}$$

which contains all policies generated during the training process. The superscripts are ordinal timestamps for each policy, representing a strict old-to-new order. At initialization, this policy set only has one element, $\pi_\theta^{(0)}$, referring to the initial policy. As is discussed in 4.3.1, severs always maintain an up-to-date global policy which can be inferred as $\pi_\theta^{(|\pi_\Theta|-1)}$. After each server policy update, a new policy, $\pi_\theta^{(|\pi_\Theta|)}$, is created and is appended to π_Θ. Thus, the cardinality of π_Θ will keep incrementing as the training goes.

For a set of n agents, $W = \{w_1, w_2, w_3, \dots, w_n\}$, their role in Multi-Parallel SAC is to provide the global experience buffer, $\mathcal{D}$, with state transitions $(s_i, a_i^{(j)}, r_i, s_i', d_i)$ consisting of current state s_i, policy-dependent action $a^{(j)}$, reward r_i, the next state s_i', and terminal state indicator d_i, where $a^{(j)} \sim \pi_\theta^{(j)}(\cdot|s)$, $0 \leq j < |\pi_\Theta|$ and the subscript

i, where $i \in W$, denotes which agent this transition comes from.

Thus, we can represent the global buffer $\mathcal{D}$ as

$$\mathcal{D} = \{(s_i, a_i^{(j)}, r_i, s_i', d_i) : i \in W,\ 0 \le j < |\pi_\Theta|\}. \tag{4.4}$$

Since $\pi_\theta^{(j)} \le \pi_\theta^{(|\pi_\Theta|-1)}$ and $\pi_\theta^{(j)}, \pi_\theta^{(|\pi_\Theta|-1)} \in \pi_\Theta$, this indicates that transitions in $\mathcal{D}$ from agents using different old policies are equivalent to past experiences of different global policies as if they were generated by servers at different timestamps. Thus, we can simplify this problem to resemble single-agent SAC where $\mathcal{D} = (s, a, r, s', d)$.

We then would like to show that the off-policy updates keep making progress. Following SAC update routine, we sample a batch of transitions, B, from $\mathcal{D}$. We can compute the gradient of Q-functions by

$$\nabla_i \frac{1}{|B|} \sum_{(s,a,r,s',d)\in B} \left(Q_i^{(|\pi_\Theta|-1)}(s,a) - y(r,s',d) \right)^2 \quad \text{for } \ i = 1, 2, \tag{4.5}$$

where $Q_i^{(|\pi_\Theta|-1)}$ represents Q-functions associated with the latest policy, $\pi_\theta^{(|\pi_\Theta|-1)}$. Given discount factor γ and entropy coefficient α, Q-function targets $y(r, s', d)$ can be expended as

$$y(r, s', d) = r + \gamma(1-d) \left(\min_{i=1,2} Q_i^{(|\pi_\Theta|-1)}(s', \bar{a}') - \alpha \log \pi_\theta^{(|\pi_\Theta|-1)}(\bar{a}'|s') \right), \tag{4.6}$$

where $\bar{a}' \sim \pi_\theta^{(|\pi_\Theta|-1)}(\cdot|s')$. The policy gradient can be updated by using the gradient

$$\nabla \frac{1}{|B|} \sum_{s\in B} \left(\min_{i=1,2} Q_i^{(|\pi_\Theta|-1)}(s, \tilde{a}) - \alpha \log \pi_\theta^{(|\pi_\Theta|-1)}(\tilde{a}|s) \right), \tag{4.7}$$

where $\tilde{a} \sim \pi_\theta^{(|\pi_\Theta|-1)}(\cdot|s)$ and is differentiable w.r.t. the policy parameters.

As is shown in Equation 4.5, 4.6 and 4.7, the optimization only happens on the latest policy and its Q-functions. This simply implies that we keep generating the next policy $\pi_\theta^{(|\pi_\Theta|)}$ on top of the current latest one. Thus, it theoretically guarantees that the trajectory optimization is always moving forward. We can only achieve this by shifting the policy update from the worker end to the server end, since adding up gradients computed on old policies from workers following the standard parameter server procedure does not necessarily optimize the global policy.

4.4 Experiments

In this section, we will discuss the experiment results of the proposed Multi-Parallel SAC on the NoCrash benchmark [20]. We will discuss both the performance of our method as well as the training speed-up compared with single-agent methods.

4.4.1 Implementation Details

Following SAC [33], the actor network (i.e. policy network) is a two-layer MLP with ReLU and Tanh activations, whereas critic networks (i.e. Q-networks) are three-layer MLPs with ReLU activations. The target Q-networks are obtained by Polyak averaging [70] the Q-network parameters over the course of training. We instantiate both policy network and two sets of Q-networks for clipped double-Q design on servers, which are updated every 25 global timesteps. Since Multi-Parallel SAC separates actors and learners, workers do not need access to Q-networks. Only policy networks are instantiated on workers, which are sync-ed with servers at every 25 local timesteps. To add more flexibility, each worker will host multiple instances of policy networks and each instance is assigned to a different agent of that worker. Those copies do not share weights and they sync with servers individually via workers. During the training, this makes agents running in the same environment act differently. We fix the entropy coefficient, α since we find using the learnable entropy temperature proposed in [34] makes the training more unstable. For additional hyperparameter settings and implementation details, see Appendix A.

We use CARLA 0.9.10.1[3] for all experiments. We train our Multi-Parallel SAC on *Town01* for at most 16 million timesteps, including $10,000$ initial timesteps of random exploration. However, we find that test time performance has already peaked after 2∼3 million timesteps of training. During the training, we terminate an episode when we meet any of the following criteria.

- The agent crashes into other objects.
- The agent is out of the designated target lane.
- The agent reaches time limit ($10,000$ steps for each episode).
- The agent successfully reaches within 10m of the goal.

Hardware-wise, we run all experiments on a 4-GPU (GeForce RTX 2080Ti [4]) node on our cluster with 192GB RAM and 40 CPU cores available.

During test time, we only use one agent to perform driving tasks on *Town02*. To mitigate the stochasticity in the CARLA environment, for each evaluation task, we run 3 times using different seeds. The final result is the average success rate on NoCrash [20] tasks with empty, regular or dense traffic.

4.4.2 NoCrash Baselines

Following [1] , we exhaustively compare our method against all three popular self-driving approaches, namely, the modular approach, the imitation learning and reinforcement learning (RL). For RL methods, we compare our method with both online RL methods and offline RL methods. Details of other methods are as follows. Note that the description of CIRL [56] is for 3.2.1.

Conditional Imitation Learning (CIL) [19]. This work proposes a conditional imitation learning algorithm that utilizes vision-based inputs as well as high-level

[3]https://carla.org/2020/09/25/release-0.9.10/
[4]https://www.nvidia.com/en-me/geforce/graphics-cards/rtx-2080-ti/

command inputs as constraints, which makes imitation learning agent capable of responding to navigational commands at test time.

Controllable Imitative Reinforcement Learning (CIRL) [56]. This work proposes a two-stage imitation-to-RL pipeline for urban self-driving. It pre-trains the policy using supervised imitation learning and then continues the policy optimization via an off-policy RL method based on DDPG [57].

Conditional Affordance Learning (CAL) [76]. This work proposes the use of a pre-trained VGG-16 [82] as a feature extractor to extract high-level low-dimensional affordances from videos. After getting affordance predictions, a rule-based controller is used to manipulate the ego-vehicle.

Conditional Imitation Learning-based ResNet (CILRS) [20]. This work builds on top of CIL and proposes the use of the ResNet [36] to directly predict the desired steering angle and target speed from vision inputs.

Learning by Cheating (LBC) [14]. This work proposes a two-stage imitation learning pipeline to train a pure vision-based self-driving agent. It first trains a privileged agent with expert demonstrations. After that, the privileged agent will guide the learning of the sensorimotor agent. Both trainings involves the use of imitation learning.

RL with Implicit Affordances (IA) [85]. This work proposes to learn a DQN [59] agent for urban autonomous driving. To extract a high-level observation space, it uses the ResNet [36] that encodes vision inputs to implicit affordances, resulting in low-dimensional features for RL agent.

Affordance-based RL (ARL) [1]. This work proposes a series of online RL methods to solve complex urban driving problem. It explicitly selects helpful affordances from CARLA sensors and groundtruths as RL inputs .

World on Rails (WOR) [15]. This work proposes a vision-based and model-based offline RL method that assumes a non-reactive world model and a low-dimensional and compact forward model of the ego-vehicle. After offline RL training and knowledge distillation [38], a visuomotor policy is eventually learnt to only use raw sensor inputs to carry out autonomous driving.

Since we have not reproduced many of the aforementioned methods, we use numbers reported from those works to present their performances. It is worth noting that not all methods originally run on the same version of the CARLA simulator. This might slightly affects the result for vision-based methods as the rendering engine, texture details and shading & lighting are varying by versions. This, however, does not affect our method because we do not use any high-dimensional visual inputs.

When it comes to the training speed comparison, we only compare our Multi-Parallel SAC with ARL [1] as we share similar state space. Also, since we run our methods on our lab cluster, we only have a rough approximation of the training speed as the workload on the cluster is always fluctuating and beyond our control.

4.4.3 NoCrash Results

We begin with the analysis of the success rate on NoCrash [20] benchmark under different traffic conditions. The overall results for all above-mentioned methods and our proposed Multi-Parallel SAC (denoted as MPSAC) are in Table 4.1. Especially for dense traffic scenario, which is the most challenging task in this benchmark, our method achieves state-of-the-art performance, surpassing all other methods.

We notice that MPSAC(std64x), a 64-agent training system with standard parameter server workflow, fails miserably as it does not learn how to drive at all. This suggests that the local gradients are highly noisy and biased, which are unable to optimize the global policy. Also, since these gradients are computed using local objectives instead of the global objectives shown in Equation 4.5 and 4.7, they lack theoretical guarantee on making progress in the training. Hence, the global policy cannot be

	CIL	CAL	CILRS	LBC	IA	WOR	ARL	MPSAC (1x)	MPSAC (std64x)	MPSAC (64x)
Empty	48	36	51	100	99	94	100	96	0	97
Regular	27	26	44	94	87	89	98	92	0	96
Dense	10	9	38	51	42	74	91	92	0	92

Table 4.1: Quantitative results on NoCrash [20] benchmark under empty, regular, dense traffic conditions (defined in 3.2.2). Numbers represent the average success rate (in percentage) out of 25 testing episodes. Details of all baseline methods are in 4.4.2. MPSAC stands for our method, Multi-Parallel SAC. MPSAC(1x) refers to our implementation of single-agent SAC [33]. MPSAC(std64x) refers to the 64-agent MPSAC using standard parameter server setup where locally computed gradients are used for the global update.

effectively updated using such gradients. This highlights the importance of modifying the original parameter server workflow to support distributed RL methods.

We also notice that the success rate of our agents does not differ by much when changing the number of dynamic actors in the test environment. This is different from CIL [19], CAL [76], CILRS [20], LBC [14], IA [85], WOR [15], whose performances drop sharply when the driving environment becomes more complex and chaotic. We suspect this is because these agents do not properly stop for obstacles or traffic lights, whereas our agents, which use similar state space as ARL [1], do a much better job at obstacle avoidance and traffic rule obedience. In addition, we discover that almost all failure cases of our multi-agent runs come from timeouts where our agent basically gets stuck on the road, being static. We suspect this is because the low-dimensional state space and action space are susceptible to a special deadlock situation, where the agent erroneously predict a zero or effectively zero target speed in certain states. Once the agent stops, our low-dimensional observation is basically frozen unless there are changes in affordances. Consequently, the same observation causes the agent to keep making the same error as SAC is deterministic at test time.

We also report the approximate simulation timesteps per hour for different methods. Results and settings are in Table 4.2. We only compare our method with ARL [1] since we share many similarities in terms of the RL setup. We choose not to report the total training time for each method although our method with 64 agents can finish

	ARL	MPSAC				
# of servers	-	1	1	1	1	4
# of workers	-	1	8	8	8	8
# of agents per worker	-	1	1	8	8	8
Std. param. server	-	×	×	✓	×	×
Approx. steps per hour	15K	15K	110K	30K	340K	345K

Table 4.2: Approximate simulation timesteps per hour under different settings. MPSAC stands for our method, Multi-Parallel SAC. "Std. param. server" indicates whether the standard parameter server workflow is used (i.e. agents computing gradients w.r.t. local policies and sending them to servers) As is mentioned in 4.3.1, agents refer to individual actor and each worker can have multiple agents. The total number of agents in one training system is equal to the number of workers multiplied by the number of agents per worker. Due to the stochastic nature of reinforcement learning, the number of timesteps needed for one method to reach its peak performance is highly unpredictable. Even for the same method, total training time (defined as the time needed for a method to reach its peak performance) of multiple runs varies a lot. Thus, we report how many timesteps a given method can complete in an hour instead. Nonetheless, the reported profiling results are heavily dependent on cluster status. We acknowledge that our results are indicative and not ideal for quantitative comparison at a fine-grained level.

the training under 7 hours, which is days in advance compared with ARL's [1] typical 1∼2 weeks of training time. The reason that we are reluctant to report the total training time is that the RL training process is highly stochastic and unstable [2, 18]. Thus, we do not suggest interpreting the total training time as a sound metric for quantitative analysis. Instead, we will focus on simulation timesteps per hour to demonstrate the effectiveness of distributed training.

Based on our our experiments, our 64-agent Multi-Parallel SAC, MPSAC(64x), accelerates the training significantly by more than 20× compared with our single-agent Multi-Parallel SAC baseline, MPSAC(1x). Combined with their results shown in Table 4.1, it indicates that this acceleration is not at the expense of the performance. We also notice that the number of servers do not affect the training speed even with a large number of agents active in the training system. This may suggest the bottleneck of the training is still at the worker end with the simulator rather than the communication or the computation on servers. This indicates further speeding up by scaling up is still possible. As is also clearly shown in the result, compared with only having a single agent in a worker, having multiple agents in one simulator can make better use of the simulation environment, improving the training efficiency. This also proves the effectiveness of our multi-level parallelization design.

As an ablation study on our modified update scheme, we find that MPSAC(std64x), the 64-agent MPSAC using standard parameter server routine, has a much slower training speed in comparison with MPSAC(64x), which pushes local buffers to servers, avoiding local gradient computations.

4.4.4 Sensitivity Analysis

We notice that ARL [1], which also uses low-dimensional affordances, seems to suffer less from the timeout issue as they are built upon on-policy PPO [79] algorithm. Thus, we suspect that complex trajectory optimization problems like self-driving are not ideal for entropy-regularized RL methods such as SAC because they heavily

$\log(\alpha)$	-2	-1	0	1	2	3
Empty	0	0	92	97	100	99
Regular	0	0	93	96	92	56
Dense	0	0	93	92	56	44

Table 4.3: Sensitivity test of entropy regularization term α on NoCrash [20] benchmark under empty, regular, dense traffic conditions (defined in 3.2.2). Numbers represent the average success rate (in percentage) out of 25 testing episodes. All experiments use Multi-Parallel SAC with the same hyperparameter setup except for the α.

rely on a fragile balance between exploration and exploitation, which is hard to find in such problems. This might result in SAC converging suboptimally. To demonstrate this problem, we investigate the importance of the entropy regularization coefficient, α, used in Equation 4.6 and 4.7. It strongly determines the trade-off between incentivizing more entropy (i.e. exploration) versus maximizing the total return (i.e. exploitation). Thus, we conduct different experiments on the NoCrash benchmark using different αs while keeping all other hyperparameters the same.

As is shown in the Table 4.3, we search over different αs linearly on logarithmic scale. These results clearly demonstrate that SAC algorithm is sensitive on the exploration-exploitation trade-off as small tweaks in α can lead to drastic performance gap. Generally, we find that the larger the entropy regularization term α is, the more SAC encourages stochastic behaviors. When faced with simple driving scenarios such as no other vehicles nearby (i.e. an empty town), a high entropy level helps the ego-vehicle avoid overfitting, thus improving the generalization during similar scenarios at test time. However, this high entropy adversely affects the ego-vehicle's ability to solve challenging scenarios such as driving in a busy traffic where the margin of error is much more constrained. Suboptimal behaviors in such scenarios will certainly cause accidents or infractions. On the other hand, having little to no entropy (i.e. $\log(\alpha) = -2, -1$) in the environment will prevent the agent from exploring possible solutions to the problem, thus not learning correctly.

Our best result comes from using $\log(\alpha) = 1$ (i.e. $\alpha = e$), which seems to find a good balance between exploration and exploitation for NoCrash Benchmark. However, this does not necessarily work for other benchmarks. To this end, we report the MPSAC agent performance on the Leaderboard benchmark also using $\log(\alpha) = 1$, which is detailed in Table 5.4. The multi-agent system is trained with the same 15-dim enhanced state space described in 5.2. The overall performance is very poor with lots of timeout issues, which are the same symptom demonstrated by MPSAC agents on NoCrash benchmark with little to no entropy (i.e. $\log(\alpha) = -2, -1$). Hence, this indicates that the entropy level needed for different tasks differs a lot, which is not ideal for complex and uncertain environments.

These experiments confirm our speculation that off-policy RL methods, often with explicit entropy regularization, are only suitable to solve problems where subproblems have the same or similar level of difficulty with each other. With diverse driving tasks and intractable variability, it is hard to maintain the balance between exploration and exploitation for self-driving tasks. Therefore, we would like to neutralize this problem using on-policy RL methods.

Chapter 5

Multi-Parallel PPO

5.1 Overview

Our preliminary parallel RL method, Multi-Parallel SAC, has achieved great training time speed-up as well as maintaining a good performance at test time. However, during the development we also realize the following problems. We notice that the SAC [33, 34], being an off-policy reinforcement learning (RL) method, heavily relies on an accurate tuning of the exploration-exploitation trade-off, which requires certain prior knowledge to the environment dynamics. This alone is unrealistic for autonomous driving because of the variability presented in the environment. For example, the amount of uncertainty varies a lot when comparing driving straight with no other actors nearby versus making a left turn in heavy traffic. Another issue revealed in our previous method is that off-policy methods tend to update their policies and replay buffers at a high frequency (e.g. per timestep, every 5 timesteps or every 25 timesteps). This results in frequent worker-server communication for buffer and parameter transfer. Although we have shown in 4.4.3 that serves are able to handle such workload, this inevitably interrupts the training routine on workers as they have to frequently upload buffer contents and reload network parameters. In conclusion, our concerns are as follows.

- It is hard for off-policy RL methods to balance between exploration and exploitation due to the complexity of self-driving problem.
- The pattern of off-policy RL methods frequently updating parameters create heavy communication workload for workers.

On the other hand, on-policy RL methods [60, 78, 79, 92] are born with solutions to these problem. First, on-policy learners do not have a dilemma where they must weigh between exploration and exploitation. They optimize directly on their past trajectories. Also, on-policy methods optimize on long-term cumulative rewards, allowing them to collect longer trajectories between each policy update. In our distributed asynchronous training setup, this effectively reduces worker-server communication.

In this chapter, we propose yet another multi-agent RL algorithm called Multi-Parallel PPO which inherits the same parallelization framework from Multi-Parallel SAC while effectively addressing above-mentioned issues. Furthermore, besides showing near-perfect performance on NoCrash [20] benchmark, we would like to demonstrate that Multi-Parallel PPO achieves competitive result on CARLA Leaderboard (described in 3.2.3), whose difficulty is significant and unmatched by any other benchmarks so far in the CARLA simulation environment.

5.2 Problem Setup

As a continuation of our previous research, our RL problem setup is basically the same as we describe in 4.2. For tackling NoCrash benchmark, we use the same state space $\mathcal{S}$, action space $\mathcal{A}$ and reward function $\mathcal{R}$ as we previously described. However, for CARLA Leaderboard, since it contains scenarios such as lane change and lane merging, only accessing front obstacle information, o_v^{f} and o_Δ^{f}, is not enough. These scenarios are intractable without obstacle affordances from other directions.

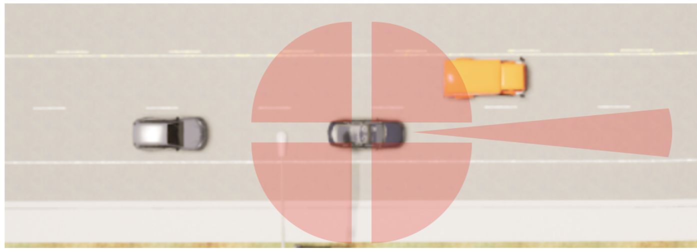

Figure 5.1: The obstacle sensor array for retrieving obstacle affordances from all angles around of the ego-vehicle. Note that this is purely for illustrative purposes. Neither the sensing radii & ranges nor the sensor placements are drawn exactly.

As a result, we tweak our state space $\mathcal{S}$ for the CARLA Leaderboard experiments while still keeping $\mathcal{A}$ and $\mathcal{R}$ the same. As is illustrated in Figure 5.1, we intuitively add four more obstacle sensors on the ego-vehicle, facing front-left, rear-left, rear-right and front-right, respectively. All non-front obstacle sensors are configured to have a wider and shorter sensing range to reduce irrelevant detections. Hence, our new 15-dim state space can be represented as

$$\mathcal{S} = \begin{bmatrix} \tilde{w}_\theta & \mathbf{O}_v & \mathbf{O}_\Delta & e_\theta & e_v & \tilde{e}_\delta & \tilde{r} \end{bmatrix}^T, \tag{5.1}$$

where

$$\mathbf{O}_v = \begin{bmatrix} o_v^{\text{f}} & o_v^{\text{fl}} & o_v^{\text{rl}} & o_v^{\text{rr}} & o_v^{\text{fr}} \end{bmatrix}^T, \quad \mathbf{O}_\Delta = \begin{bmatrix} o_\Delta^{\text{f}} & o_\Delta^{\text{fl}} & o_\Delta^{\text{rl}} & o_\Delta^{\text{rr}} & o_\Delta^{\text{fr}} \end{bmatrix}^T. \tag{5.2}$$

5.3 Method

5.3.1 Distributed Framework

As our distributed framework is designed to be compatible with both off-policy and on-policy methods, we continue to use our two-layer parallelization hierarchy illustrated in Figure 4.1. For asynchronous communication, we continue to use our modified parameter server architecture described in 4.3.1 with some necessary changes

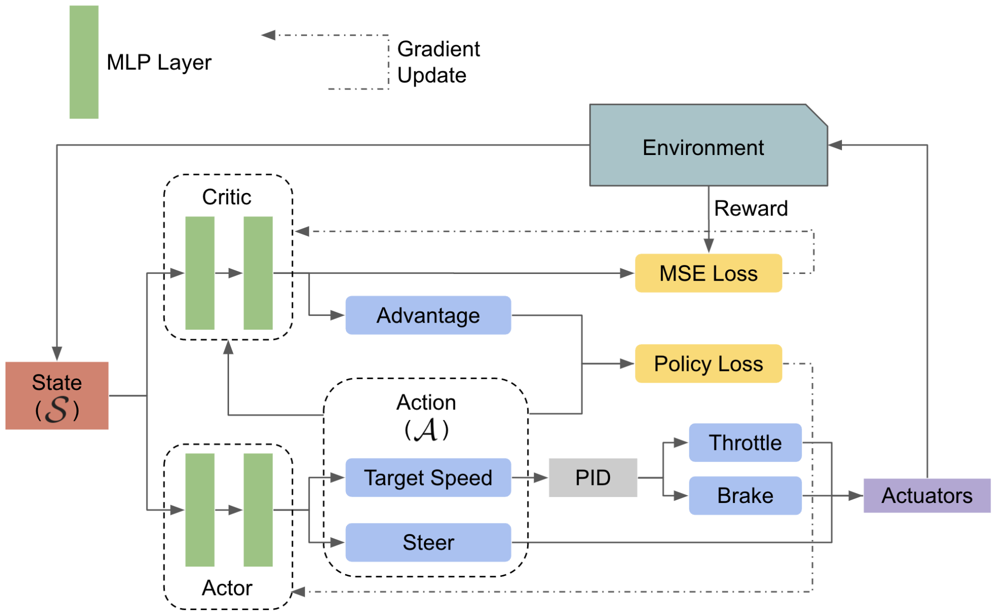

Figure 5.2: The workflow of single-agent Proximal Policy Optimization (PPO) [79] algorithm, it features an actor-critic pipeline for policy and value function updates. This figure is only for illustrative purposes. Many components, such as trajectory buffer and importance sampling, are not shown for simplicity.

for on-policy optimization. For servers, they now do not maintain a global experience replay buffer. Instead, they now only maintain a trajectory buffer which only saves trajectories sent by workers. This buffer empties itself after each update as each trajectory are used only once. For agents, they now only make push request after collecting a compete trajectory (i.e. finishing an episode) using their local policies. To make sure all trajectories are consistent, agents do not request for policy synchronization with servers in the middle of an episode. They now only fetch global policy once an episode is terminated. These changes together make our distributed parameter server architecture coordinate the training in an on-policy fashion. Since trajectories in self-driving problem typically last for hundreds or thousands of simulation timesteps. This reduces worker-server communication frequency significantly.

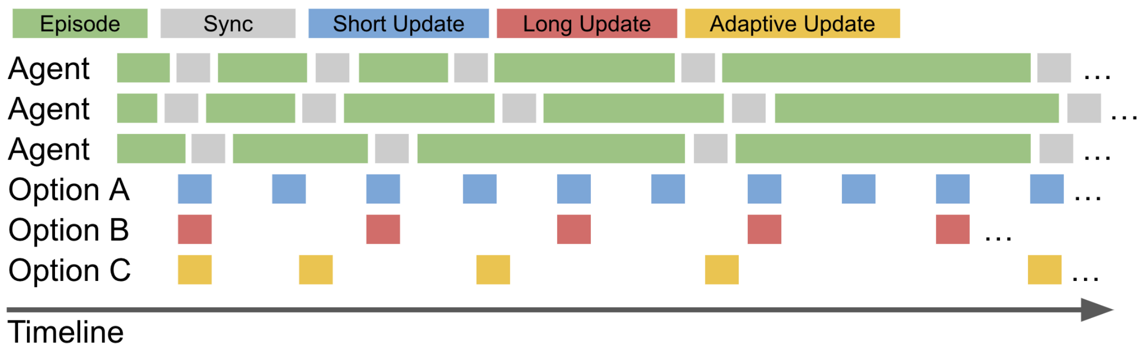

Figure 5.3: Different choices of update schemes for distributed on-policy RL. With the driving policy improving, on average, training episodes typically become longer. Option A chooses to update the global policy with a fixed short interval. This might be good initially but will harm the optimization in the long term by breaking up long episodes into several trajectories as well as making agents out-of-sync from the global policy. Option B chooses to update the global policy with a fixed, relatively long interval. This might be good in the long run but will slow down the initial training due to less frequent updates. Option C (Ours) chooses to update adaptively based on the length of training episodes, addressing above-mentioned problems.

5.3.2 Multi-agent PPO

While it may be relatively trivial and intuitive to parallelize off-policy methods by simply adding more agents as data collectors, the way to parallelize on-policy methods are not that obvious.

As its name suggests, Multi-Parallel PPO is built on top of PPO [79], an online single-agent on-policy RL method. The workflow of PPO is depicted in Figure 5.2. At every update, PPO performs policy optimization using just its own trajectories. However, with our training framework being asynchronous, at any given time there might be multiple policies in the system. Local policies on workers are not necessarily the up-to-date policy and may be out of sync over the course of training due to the lack of communication. If we still try to collectively learn a global policy by either sending local trajectories or local gradients to servers, it will violate the assumption of on-policy methods that they should learn from their own trajectories. Not alleviating this issue might break the PPO algorithm. Hence, we would like to show how we specifically handle this problem and adapt the original single-agent PPO for distributed asynchronous training.

Therefore, our objective right now is to find a way to scale up standard PPO algorithm while keeping the integrity of on-policy optimization. To figure that out, let us dig into the theory behind PPO. We first define what is a trajectory. A trajectory τ is a sequence of states and actions as

$$\tau = \{s_1, a_1, s_2, a_2, \ldots, s_T, a_T\}. \tag{5.3}$$

One of the PPO variants updates the objective using first-order approximation (i.e. PPO-CLIP [79]). This involves two policies, the to-be-calculated new policy, π_θ, and the current policy, $\pi_{\theta_{old}}$, which is used for trajectory collection, parameterized by θ and θ_{old}, respectively. Given advantage estimation $\hat{A}_t$, we can retrieve θ by maximizing the expectation of $\hat{A}_t$. We have the objective as

$$\theta = \underset{\theta}{\operatorname{argmax}} \sum_{t=1}^{T} \mathbb{E}_{(s_t,a_t)\sim p_\theta(s_t,a_t)}[\hat{A}_t], \tag{5.4}$$

where $p_\theta(\cdot)$ is the probability distribution of states and actions under policy π_θ. Thus, we can write down the corresponding loss function as

$$\mathcal{L}(\theta) = \sum_{t=1}^{T} \mathbb{E}_{(s_t,a_t)\sim p_\theta(s_t,a_t)}[\hat{A}_t] = \sum_{t=1}^{T} \mathbb{E}_{s_t\sim p_\theta(s_t)}\left[\mathbb{E}_{a_t\sim\pi_\theta(a_t|s_t)}[\hat{A}_t]\right]. \tag{5.5}$$

As θ right now is unknown and we do not have any samples from π_θ, we can use importance sampling [67] to utilize samples from $\pi_{\theta_{old}}$ to optimize $\mathcal{L}(\theta)$. We can re-write our loss function as

$$\mathcal{L}(\theta) = \sum_{t=1}^{T} \mathbb{E}_{s_t\sim p_{\theta_{old}}(s_t)}\left[\frac{p_\theta(s_t)}{p_{\theta_{old}}(s_t)}\mathbb{E}_{a_t\sim\pi_{\theta_{old}}(a_t|s_t)}\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t\right]\right]. \tag{5.6}$$

The term $\frac{p_\theta(s_t)}{p_{\theta_{old}}(s_t)}$ represents how probability distribution of state s_t shifts from $\pi_{\theta_{old}}$ to π_θ. One of our assumptions here is that $\pi_{\theta_{old}}$ and π_θ are very close since we constrain the change in policy. As a result, we can ignore this term by assuming $\frac{p_\theta(s_t)}{p_{\theta_{old}}(s_t)} \approx 1$. We can then simply approximate the loss as

$$\mathcal{L}(\theta) \approx \sum_{t=1}^{T} \mathbb{E}_{s_t \sim p_{\theta_{old}}(s_t)} \left[\mathbb{E}_{a_t \sim \pi_{\theta_{old}}(a_t|s_t)} \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}_t \right] \right] = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}_t \right], \quad (5.7)$$

where the expectation $\hat{\mathbb{E}}_t[\dots]$ indicates the empirical average over a finite batch of samples. We notice that the final output of Equation 5.7 is exactly the surrogate objective $L^{\mathrm{CPI}}(\theta)$ shown in original PPO [79] paper for PPO-CLIP (Equation 6).

Now that we have connected our deduction to PPO's, we notice that the only assumption we make here is that $\pi_{\theta_{old}}$ and π_θ should be close to each other. In our asynchronous multi-agent PPO setup, we have multiple local policies which are parameterized by different θ_{old}s, denoted as $\theta^{(1)}_{old}, \theta^{(2)}_{old}, \theta^{(3)}_{old}, ..., \theta^{(n)}_{old}$, and an up-to-date global policy which is parameterized by parameter θ_{glb}. Thus, so long as θ_{old}s synchronize with θ_{glb} frequently, it is safe to assume

$$\frac{p_{\theta^{(i)}_{old}}(s_t)}{p_{\theta_{glb}}(s_t)} \approx 1, \qquad i = 1, 2, \dots, n, \quad (5.8)$$

Due to asynchronicity, we need to use samples collected from $\pi_{\theta_{old}}$s to optimize the unknown new policy, π_θ. We can re-formulate the loss function by importance sampling on π_θ as

$$\mathcal{L}(\theta) = \sum_{t=1}^{T} \mathbb{E}_{s_t \sim p_{\theta^{(i)}_{old}}(s_t)} \left[\frac{p_\theta(s_t)}{p_{\theta^{(i)}_{old}}(s_t)} \mathbb{E}_{a_t \sim \pi_{\theta^{(i)}_{old}}(a_t|s_t)} \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta^{(i)}_{old}}(a_t|s_t)} \hat{A}_t \right] \right]. \quad (5.9)$$

Given the assumption in Equation 5.8, we have

$$\frac{p_\theta(s_t)}{p_{\theta^{(i)}_{old}}(s_t)} \approx \frac{p_\theta(s_t)}{p_{\theta^{(i)}_{old}}(s_t)} \cdot \frac{p_{\theta^{(i)}_{old}}(s_t)}{p_{\theta_{glb}}(s_t)} = \frac{p_\theta(s_t)}{p_{\theta_{glb}}(s_t)} \approx 1, \qquad i = 1, 2, \dots, n, \quad (5.10)$$

which means we can proceed with training the global policy with local trajectories iff.

our assumptions in 5.8 hold. This means although asynchronous RL is necessary for practicality, we should bridge the gap between the global policy and local policies as much as possible in order to limit the overall asynchronicity in the training system.

To uphold these assumptions on policy proximity, in Multi-Parallel PPO, the ideal case is that worker-server synchronization always happens more frequently or at least as frequently as the global policy is updated. However, due to the on-policy constraint, we cannot intervene the sync frequency between servers and each agent in workers since we are not able to switch policies in the course of an episode. Also, we cannot control the length of an episode, either.

To save us from this dilemma, we propose to set global policy update interval adaptively according to the average length of current training episodes (i.e. length of episodes). An illustration of different update schedules is in Figure 5.3. Specifically, we do not enforce a fixed rollout length such as 100 or 1K that might break up an episode. In case that an episode is unreasonably long, we truncate the trajectory after it reaches some exceptional length, such as 20,000. Otherwise, we only collect complete trajectories when an episode terminates. Thus, the length of the trajectories is varied during the training, reflecting the average length of the episodes. Our update frequency is therefore adjusted to be equal to the number of workers multiplied by the average length of certain number of past trajectories in a sliding window fashion.

By using our adaptive update interval, we are able to quickly update the global policy at the early stage of the training when the average trajectory length is short. As the training proceeds, the average length of trajectories tend to increase as agents make fewer mistakes and can stick on the road longer. The policy update interval on servers is thus automatically increased to make sure each agent in workers can keep pace with the global policy. This makes policy update less frequent during the later stage of training when each agent performs longer episodes and seldom has the chance to sync up its local policy. While we do not set an upper limit, we impose a minimal interval between updates which ensures at the early stage of the training we get enough meaningful trajectories before computing policy gradients.

5.4 Experiments

In this section, we will discuss the experiment result of the proposed Multi-Parallel PPO on both NoCrash benchmark [20] and CARLA Leaderboard (detailed in 3.2.3). For NoCrash benchmark, We will discuss both the performance of Multi-Parallel PPO as well as the training acceleration compared with other methods, including Multi-Parallel SAC. For Leaderboard, We mainly focus on analyzing the driving performance of our agent.

5.4.1 Implementation Details

Multi-Parallel PPO specifics. Both the policy network and value function network are three-layer MLPs with Tanh activations. Following Multi-Parallel SAC, different agents in the same worker do not share the same local policy. Instead, they individually fetch the global policy from servers through worker-server communication after finishing an episode. Agents only push full trajectories to servers in order to keep cumulative rewards accurate. They do not upload partial trajectories in the middle of an episode. The minimal global policy update interval is 100 timesteps. Servers will adjust the update scheme roughly in proportion to the average length of episodes in recent training. For additional hyperparameter settings and implementation details, especially w.r.t. the adaptive update interval, see Appendix A.

NoCrash experiments. For NoCrash experiments, we follow all previous training and testing setups described in 4.4.1. As this is true for most on-policy methods, we find that Multi-Parallel PPO requires more simulation interactions to converge compared with Multi-Parallel SAC, bringing up the total number of timesteps needed to reach peak performance at test time to around 10 million.

Leaderboard experiments. For CARLA Leaderboard, we train 16 million simulation timesteps on all training routes. Unlike Nocrash benchmark, Leaderboard routes are across multiple towns, including *Town01*, *Town03*, *Town04* and *Town06*. Thus, we randomly switch the map on each environment every 100 episodes during the training. To avoid deterministic behaviors, we turn off pre-defined traffic scenarios, which regulate other actors' behaviors. For Leaderboard evaluation, since we have no access to testing routes used for the Leaderboard Challenge, we evaluate our method on evaluation routes via official released evaluation script [1].

5.4.2 NoCrash Baselines

In consistent with Multi-Parallel SAC, we continue to use those methods described in 4.4.2 as NoCrash benchmark baselines in our comparison. We categorize them into modular pipelines, imitation learning and reinforcement learning (RL), which are listed as follows.

- Modular pipelines: CAL [76].
- Imitation Learning: CIL [19], CILRS [20], LBC [14].
- Reinforcement Learning: IA [85], ARL [1], WOR [15].

5.4.3 NoCrash Results

We begin with the analysis of the success rate on NoCrash [20] benchmark under different traffic conditions. The overall results for all above-mentioned methods and our proposed Multi-Parallel SAC (denoted as MPSAC) and Multi-Parallel PPO (denoted as MPPPO) are in Table 5.1. Especially for dense traffic scenario, which is the most challenging task in this benchmark, our method achieves state-of-the-art performance, surpassing all other methods.

As an ablation study, we report the performance of MPPPO(std64x), which is a

[1] https://github.com/carla-simulator/leaderboard

64-agent training system following standard parameter server routine where the global policy is updated based on local gradients. After 10M timesteps of training, the MPPPO(std64x) seems to learn nothing. We suspect that even with the difference between local policies and the global policy being very small thanks to frequent synchronizations, the local gradients of MPPPO(std64x) agents are still too divergent to collectively optimize the global policy in a stable direction. This is in consistent with what we have observed from MPSAC(std64x). We also report the performance of MPPPO(1K64x) and MPPPO(10K64x), which uses a fixed short update interval and a fixed long update interval respectively (illustrated in Figure 5.3). As expected, when the global update interval is too short, the model truncates its trajectories too often, making long horizon events hard to capture. Thus, it fails to handle traffic lights and obstacles properly, resulting in poor performance in regular and dense traffic scenarios. For MPPPO(100K64x) where the update interval is set longer to circumvent the aforementioned issue, it results in low utilization of shorter episodes in the course of the training, especially those at the early stage. This makes the whole training inefficient and unable to finish using the same number of total interactions with the environment.

We further investigate the effectiveness of the proposed adaptive update interval via comparing it with a series of experiments using fixed update schedules with varied fixed rollout lengths. The result is shown in Table 5.2. The average reward at 1K timesteps indicates the initial status of the training. Due to the stochasticity, these numbers are varied but are generally around -900. After roughly 1M timesteps of training, it is noticeable that the experiment using our adaptive update interval progresses the most in terms of having the largest average reward in comparison with all other experiments. It is also shown that the rollout length may affect the training according to those three experiments using 10K as the update interval but each has a different rollout length. The optimization tends to be less effective with the rollout length being either too short (i.e. 125) or too long (i.e. 1K). Evidence also suggests that using a small update interval may improve the training in the CARLA self-driving problem. However, the minimal interval is lower-bounded by the fixed rollout length. This prevents us from further increasing the update frequency without making the rollout length unreasonably short.

	CIL	CAL	CILRS	LBC	IA	ARL	WOR	MPSAC (64x)	MPPPO (std64x)	MPPPO (1K64x)	MPPPO (100K64x)	MPPPO (64x)
Empty	48	36	51	100	99	100	94	96	0	99	99	99
Regular	27	26	44	94	87	98	89	92	0	68	64	99
Dense	10	9	38	51	42	91	74	92	0	12	51	96

Table 5.1: Quantitative results on NoCrash [20] benchmark under empty, regular, dense traffic conditions (defined in 3.2.2). Numbers represent the average success rate (in percentage) out of 25 testing episodes. Baseline methods are detailed in 4.4.2. MPSAC stands for Multi-Parallel SAC, whereas MPPPO stands for Multi-Parallel PPO. In addition, MPPPO(std64x) refers to the 64-agent MPPPO using standard parameter server setup where locally computed gradients are used for the global update. MPPPO(1K64x) refers to the 64-agent MPPPO with a fixed global update interval at 1,000 timesteps, whereas MPPPO(100K64x) refers to the 64-agent MPPPO with a fixed global update interval at 100,000 timesteps.

	Multi-Parallel PPO						
Rollout length	125	200	200	1K	1K	1.2K	-
Update interval	10K	10K	1K	10K	1K	100K	`Adaptive`
Avg. reward at 1K timesteps	-830	-820	-925	-1,034	-875	-1,073	-1,027
Avg. reward at 1M timesteps	-457	-267	-282	-501	-229	-552	-93

Table 5.2: The impact on the training progress using different rollout lengths and update intervals. The experiment with the adaptive update interval does not use a fixed rollout length as previously described in 5.3.2. Comparing the average reward at the initial of the training (i.e. at 1K timesteps) with the average reward after 1M timesteps, the experiment with the adaptive update interval improves the most during this period of training, showing the effectiveness of our design over a fixed update interval with a fixed rollout length.

Similar to the performance of Multi-Parallel SAC, the success rate of Multi-Parallel PPO is not drastically affected by the number of dynamic actors presented in the environment, indicating a well-learnt controller capable of safety-critical driving.

Compared with Multi-Parallel SAC, Multi-Parallel PPO prevails on all three tasks. Since our methods share the same asynchronous and parallel training framework, the only difference is the underlying RL algorithm itself. This echoes with our speculation that off-policy methods struggle to balance between exploration and exploitation in complex urban autonomous driving situations. Without this dilemma, on-policy methods are more robust to such world dynamics.

We also find that our Multi-Parallel PPO agent outperforms ARL [1] on NoCrash regular and dense environments. This is interesting since ARL uses the same single-agent PPO backbone and similar low-dimensional affordances. We suspect this is either because the increased diversity of trajectories in multi-agent system is helpful for finding a better local optima, or because ARL agent has not fully converged especially on difficult tasks, which definitely take a very long time to do, or both.

When it comes to the training acceleration (shown in Table 5.3, we do notice that our 64-agent Multi-Parallel PPO trains slightly slower than Multi-Parallel SAC. This might be because the single-agent baseline version of Multi-Parallel PPO runs slower than both ARL [1] and Multi-Parallel SAC. Still, we are able to show that our parallelization framework results in a 25× speed-up comparing single-agent version versus 64-agent version and a 50× speed-up comparing single-agent version versus 128-agent version. This linear acceleration in scale also suggests that the training system is still capable of accommodating more agents until it starts to show a diminishing return. We also notice that MPPPO(std64x), the 64-agent MPPPO using standard parameter server routine, consumes much more time by computing gradients and transferring them to servers, resulting in a severe reduction of the training speed.

	ARL	MPSAC		MPPPO			
# of servers	-	1	4	1	4	4	4
# of workers	-	1	8	1	8	8	16
# of agents per worker	-	1	8	1	8	8	8
Std. param. server	-	×	×	×	✓	×	×
Approx. steps per hour	15K	15K	345K	12K	50K	300K	600K

Table 5.3: Approximate simulation timesteps per hour under different settings. MPSAC stands for Multi-Parallel SAC, whereas MPPPO stands for Multi-Parallel PPO. "Std. param. server" indicates whether the standard parameter server workfow is used (i.e. agents computing gradients w.r.t. local policies and sending them to servers). As is mentioned in 4.3.1, agents refer to individual actor and each worker can have multiple agents. The total number of agents in one training system is equal to the number of workers multiplied by the number of agents per worker. Due to the stochastic nature of reinforcement learning, the number of timesteps needed for one method to reach its peak performance is highly unpredictable. Even for the same method, total training time (defined as the time needed for a method to reach its peak performance) of multiple runs varies a lot. Thus, we report how many timesteps a given method can complete in an hour instead. Nonetheless, the reported profiling results are heavily dependent on cluster status. We acknowledge that our result are indicative and not ideal for quantitative comparison at a fine-grained level.

5.4.4 Leaderboard Baselines

CARLA Leaderboard is a newly established benchmark in the CARLA community and is under rapid development. We select publicized and non-anonymous sensor track submissions [2] as our baselines. Except for IA [85], LBC [14], CILRS [20] and WOR [15], which are already detailed in 4.4.2, other methods, including TransFuser [71], TransFuser+ [43], NEAT [17] and LAV [13], are briefly introduced as follows.

Multi-Modal Fusion Transformer (TransFuser) [71]. This work proposes an imitation learning pipeline for tackling complex urban driving problem. It designs a privileged expert to collect a multi-modal (images and LiDAR point clouds) dataset for training the agent. It uses transformer [87] for attention-based sensor fusion. In our experiments, we also include this TransFuser autopilot into our Leaderboard

[2]https://leaderboard.carla.org/leaderboard/

baselines, denoted as **TransFuser AT**, accordingly.

Simple and Effective Expert Driver (TransFuser+) [43]. This work proposes an improved imitation learning pipeline for TransFuser [71]. Instead of using original TransFuser autopilot, the agent learns on expert data collected by a better rule-based autopilot which handles traffic rules and collision avoidance more realistically.

Neural Attention Fields (NEAT) [17]. This work proposes a conditional imitation method with multi-veiw images. After feature extraction by the ResNet [36], the model predicts waypoints and semantic segmentation results via an attention mechanism and a continuous implicit function for scene representation.

Learning from All Vehicles (LAV) [13]. This work proposes a new framework on top of WOR [15] by utilizing trajectories from other vehicles presented nearby the ego-vehicle in the course of training. This is made possible by a learnt viewpoint invariant perception system. To actively avoid collisions, the agent also learns a vehicle-aware motion planner.

5.4.5 Leaderboard Results

Because CARLA Leaderboard forbids the direct usage of privilege information in any self-driving method, we cannot run our Multi-Parallel PPO on the privately held-out test routes. The reason for our method relying on privilege affordances is that we would like to focus on solving long-trajectory planning and control problems for self-driving via pure online RL without the participation of imitation learning like all other methods do. We will not be able to bypass these affordances unless we shift our concentration to working on perception and sensor fusion problems, which we are reluctant to do. Still, we would like to show our result in Table 5.4 on Leaderboard evaluation routes using official evaluator, both of which are open to public access.

We acknowledge that our results are not comparable with leaderboard submissions

	Routes	DS	RC	Collis.	Viol.	Dev.	Timeouts
CILRS	Test	5.37	14.40	6.52	6.17	4.14	4.28
LBC	Test	8.94	17.54	1.56	2.23	0.03	4.69
TransFuser	Test	16.93	51.82	2.19	1.83	0	1.97
NEAT	Test	21.83	41.71	1.40	3.38	0	5.22
IA	Test	24.98	46.97	4.80	2.37	1.44	1.73
WOR	Test	31.37	57.65	2.98	1.75	1.69	0.47
TransFuser+	Test	34.58	69.84	0.77	0.93	0	2.41
LAV	Test	61.85	94.46	0.76	0.42	0	0.14
TransFuser AT	Eval.	83.95	99.63	0.04	0.08	0	0.01
MPSAC	Eval.	8.29	10.52	0.19	2.04	0.41	11.68
MPPPO	Eval.	80.64	99.27	0.06	0.05	0	0.03

Table 5.4: Leaderboard evaluation results. MPSAC stands for our proposed method, Multi-Parallel SAC, which in this case is trained with the 15-dim state space described in 5.2. MPPPO stands for our proposed method, Multi-Parallel PPO. The "Routes" column indicates how different methods are evaluated. Except for TransFuser AT and MPPPO, other methods are tested on held-out test routes. Hence, our method is only directly comparable with TransFuser AT. Descriptions of these metrics can be found in the leaderboard of the CARLA Leaderboard Challenge, the link to which can be found in 5.4.4. "DS" stands for driving scores. "RC" stands for route completion (in %) For convenience, we merge some official metrics into a single category. "Collis." combines all types of collisions in original leaderboard. "Viol." combines red light and stop sign violation, as well as off-road infraction. Infractions in collisions or violations indicate control or perception issues. "Dev." stands for route deviations, infractions in this category indicate planning issues lane departure issues. "Timeouts" combines the original timeout metric as well as situations when the agent gets blocked. The unit for all infraction-related metrics is infractions per kilometer.

due to discrepancies in our settings. However, we demonstrate that our method is capable of achieving a high driving score with few collisions, traffic rule violations and timeouts. This suggests that our Multi-Parallel PPO, or generally online RL methods, are capable of navigating through complex and even adversarial driving scenarios assuming a perfect perception system.

We also demonstrate that the performance of TransFuser autopilot [71] (denoted as TransFuser AT) on evaluation routes. TransFuser AT is the privileged expert used in TransFuser which also makes use of privilege information. However, unlike our agent, TransFuser AT is a sophisticated rule-based agent strictly calculating on the environment dynamics for its every movement. Surprisingly, as is shown in Table 5.4, the performance of our method across multiple metrics is on par with the TransFuser AT. This strongly suggests that our 15-dim state space with simple affordances can effectively deal with challenging scenarios.

Chapter 6

Conclusions

In this thesis, we have proposed two distributed asynchronous multi-agent reinforcement learning (RL) algorithms, namely, Multi-Parallel SAC and Multi-Parallel PPO. We have addressed various problems regarding the online training of RL agents on the CARLA simulator, including the low utilization of computational resources, the skewed distribution of workloads and the heterogeneity among agents participating in the training. We have demonstrated that our methods can significantly accelerate the online RL training on the CARLA simulator and achieve state-of-the-art performances across different CARLA self-driving benchmarks in significantly less time.

Appendix A

Supplementary Materials

A.1 Hyperparameters

In all experiments on both methods, hyperparameters we used in general are in Table A.1 during the training. Notions are in consistent with that have been used in the main body of this thesis. To produce our best models for both NoCrash and the Leaderboard benchmarks, hyperparameters specifically related to Multi-Parallel SAC and Multi-Parallel PPO are listed in Table A.2 and Table A.3, respectively. We use Adam optimizer [49] with default parameters among all experiments.

A.2 Adaptive Update Interval

In 5.3.2, we mention that our global update interval is adaptive compared with fixed update intervals typically used in other distributed RL methods. We are aware that many asynchronous distributed RL methods do not enforce a precise global update

Notation	Description	Value
R_v	Default speed reward.	1
α_v	Coefficient for speed reward.	1
R_θ	Default steering penalty.	-1
α_θ	Coefficient for steering penalty.	0
R_δ	Default lane deviation penalty.	-1
α_δ	Coefficient for lane deviation penalty.	1
R_I	Default infraction penalty.	-1
α_I	Coefficient for infraction penalty.	250
$\max o_v^{\mathrm{f}}$	Front obstacle detection range.	$10m$
$\max o_\Delta^{\mathrm{f}}$	Front obstacle detection radius.	0.5
$\max \mathbf{O}_v \setminus o_v^{\mathrm{f}}$	Other obstacle detection ranges.	$5m$
$\max \mathbf{O}_\Delta \setminus o_\Delta^{\mathrm{f}}$	Other obstacle detection radii.	0.7854

Table A.1: General hyperparameters for all experiments. Note that for obstacle affordance in NoCrash experiments, only a front obstacle sensor is used.

Notation	Description	Value
lr_π	Learning rate of the global policy.	4e-4
lr_q	Learning rate of Q-networks.	4e-4
lr_τ	Step size of Polyak averaging.	0.01
α	Fixed entropy temperature.	e
buffer_size	The size of global buffer pool.	1,000,000
batch_size	Batch size of each gradient update.	512
freq_q	Q-network update frequency.	25
freq_t	Target Q-network update frequency.	25

Table A.2: Hyperparameters for Multi-Parallel SAC.

Notation	Description	Value
lr_π	Learning rate of the global policy.	4e-4
ϵ	Clipping range for policy updates.	0.2
∇_{clip}	Gradient clipping parameter.	0.5
min_steps	Minimum interval between updates.	100
win_size	Sliding window size for rollout length.	100
n_steps	Maximum rollout length.	20,000
n_epochs	Number of epochs in each update.	10

Table A.3: Hyperparameters for Multi-Parallel PPO.

schedule due to the nature of asynchronicity. Their policy update mechanism typically involves agents terminating local trajectories at some fixed rollout length. Upon receiving local trajectories, learners will either directly make a gradient update, or will wait until they collect a certain number of trajectories from different agents to ensure the stability of the training. Either way, their update schedules are performed roughly around a pre-determined frequency which does not reflect the current progress of the training, unlike our adaptive update interval, which is dynamically adjusted depending on the current status of the training.

A.3 Synchronization Between Servers

Though we do not need synchronizations between workers since they are designed to run asynchronously, we do have to constantly keep all servers synced to maintain a uniformity of the global policy. We do so by simply using multi-threading and put all servers on a single machine in all our experiments. In theory, servers are able to live on different machines with an efficient low-level communication mechanism.

A.4 Instability in Leaderboard

During both the training and the testing on Leaderboard, we sometimes observe a weird error pertaining to loading pedestrians on the Leaderboard's end, which often causes pedestrian objects missing from scenarios. Here is one example of such error messages that we have encountered.

```
Skipping scenario ‘Scenario3’ due to setup error:
Error: Unable to spawn vehicle walker.pedestrian.0016 at
Transform(Location(x=252.996002, y=-76.986099, z=2.390000),
Rotation(pitch=0.000000, yaw=541.393188, roll=0.000000))
```

Due to this intermittent issue out of some Leaderboard internal disfunctions, our Leaderboard evaluation results on Multi-Parallel PPO and TransFuser autopilot [71] may not be accurate.

Appendix B

Limitations of this Study

B.1 Inaccurate Training Speed Profiling

We acknowledge the fact that the training speed fluctuates a lot even during a single training run, not to mention the variance across multiple runs and multiple machines on the lab cluster. Since running these large-scale experiments requires multiple GPUs and CPU cores, we simply cannot find a reliable local workstation isolated us from other cluster users to profile our methods. Hence, in this thesis we give our best approximations by averaging out the instability over a long period of training.

B.2 Insufficient Leaderboard Experiments

We acknowledge the fact that we cannot fairly compare our models with publicized Leaderboard Challenge results since 1) we access the CARLA groundtruth affordances at test time and 2) we do not run our models on the same testing routes as those routes are undisclosed to the public. Since our fundamental research question is centered on efficiently solving complex self-driving scenarios by pure online reinforcement learning (RL). In order to better focus on our objective, we assume full access to our simple and effective selection of necessary affordances and leave the problem on how we can get these affordances in real-world situation to others (see 3.1.3).

B.3 Lack of Qualitative Analysis

We acknowledge the fact that we do not include qualitative analysis (e.g. images and videos) on the driving behavior of our agents. This is not because we do not have qualitative materials at hand, but because it is hard to demonstrate them in this medium. In fact, we would like to claim that our models can drive stably, turn smoothly and stay at the center of lane steadily. We would also like to claim that our models constantly obey traffic lights & stop signs and yield to all types of obstacles. However, we cannot find an effective way to show these properties without showing videos of those driving scenarios, which is implausible to include in the thesis. Instead, We select eight representative driving videos featuring different benchmarks and scenarios. We upload them on the YouTube [1] for public access. Please refer to those video clips as qualitative evidence on the performance of our trained agents.

[1]https://www.youtube.com/playlist?list=PL6MavIKKfWVR8I8jxJ6PaUNJN5hW-EoF9

Appendix C

Future Work

C.1 Massive Parallelization

Currently, due to the resource constraint, we are only able to run our distributed RL framework at a relatively small scale (e.g. 64 agents or 128 agents). We rely on our lab cluster to perform all the experiments. Without inter-node InfiniBand connection, the maximum number of GPUs we can utilize right now during a single experiment is limited to 4 GPUs on a single node. Since each CARLA simulator instance requires lots of GPU memory ($\sim$2GB), currently the lack of GPUs is our bottleneck.

With new 8-GPU nodes consisting of massive memory graphics cards recently added to our lab cluster, we are hopefully able to run massive parallelization with hundreds or thousands of agents collaborating in the training system. We would thus want to try if we can solve the CARLA self-driving benchmarks in hours or even minutes.

C.2 Asynchronous Servers

Currently, because of using shared resources on the server end, our servers must be put on the same machine, which is not good for the scalability as well as the fault tolerance of the distributed system. At present, we have to do this since servers need constant synchronization. In the future, we would like to remove this synchronicity from our distributed RL framework, which will make our training system enjoy the full benefit of being asynchronous. This might be achieved through adapting asynchronous gradient update methods to our needs.

C.3 Accelerating Future Research

In this thesis, we have shown that RL is promising for self-driving tasks. In order to develop novel RL ideas to further improve the self-driving, we need to extend our proposed distributed RL framework to accelerate future RL research on the CARLA simulator. In fact, we have already begun using our distributed RL framework in some of our ongoing CARLA self-driving research works with some proper adaptation. Details of those projects are in Appendix D.

Appendix D

Extensions to Other Projects

One of the very purposes of this study is to build a generic distributed multi-agent framework to accelerate future reinforcement learning (RL) research on the CARLA simulator [23]. We achieve this goal by successfully presenting a parameter server-based distributed training system for both off-policy and on-policy methods with two-layer parallelization specifically designed for the CARLA self-driving. To demonstrate that our framework is versatile to the need of many other RL tasks, we briefly describe two concurrent RL projects in our lab that take advantage of our framework.

D.1 Online RL with Vision-based Features

One of the projects currently in our lab involves training an online RL agent with bird's-eye view RGB camera as perception input. To infuse image observations into our state space, latent features are extracted out of stacked RGB images via an autoencoder pre-trained with auxiliary tasks.

As is discussed in the paper, online RL training on CARLA takes very long time to converge, let alone the added burden of the high-frequency sensor stream. Therefore, we adapt our current distributed framework to accept the new state space. In order to lower the GPU memory usage, we instantiate a single autoencoder per worker that is shared with all the agents in that worker. It functions perfectly and helps them develop their novel method in a timely manner.

D.2 Offline RL with Parallelized Self-play

Another ongoing project in our lab resorts to offline RL to learn an effective self-driving policy with the goal that it will eventually be deployed to learn from pre-collected real-world driving logs. Inspired by LAV [13], they find it helpful to simultaneously learn from other actors' experiences because they may encounter interesting scenarios while the ego-vehicle does not. Borrowing the idea of self-play from AlphaStar [88], which is deemed useful for RL training in adversarial environment, they initialize and adapt driving policies used by non-ego-vehicles over time in the training with old policies from the learning ego-vehicle or rule-based autopilot.

To achieve the goal of learning from multiple self-play actors, we apply our distributed asynchronous framework in this project by docking it with an multi-agent offline self-driving environment and adapting our worker-server communication protocol to periodically distribute old policies to non-ego-vehicles for self-play purposes.

Bibliography

[1] Tanmay Agarwal, Hitesh Arora, and Jeff Schneider. Affordance-based reinforcement learning for urban driving. *arXiv preprint arXiv:2101.05970*, 2021. 1, 2.1.3, 3.2.1, 4.1, 4.2, 4.2, 4.4.2, 4.4.3, 4.4.3, 4.4.4, 5.4.2, 5.4.3, 5.4.3

[2] Marcin Andrychowicz, Anton Raichuk, Piotr Stańczyk, Manu Orsini, Sertan Girgin, Raphael Marinier, Léonard Hussenot, Matthieu Geist, Olivier Pietquin, Marcin Michalski, et al. What matters in on-policy reinforcement learning? a large-scale empirical study. *arXiv preprint arXiv:2006.05990*, 2020. 4.4.3

[3] Eduardo Arnold, Omar Y. Al-Jarrah, Mehrdad Dianati, Saber Fallah, David Oxtoby, and Alex Mouzakitis. A survey on 3d object detection methods for autonomous driving applications. *IEEE Transactions on Intelligent Transportation Systems*, 20(10):3782–3795, 2019. doi: 10.1109/TITS.2019.2892405. 2.1.1

[4] Karl Johan Åström and Tore Hägglund. *PID Controllers: Theory, Design, and Tuning*. ISA - The Instrumentation, Systems and Automation Society, 1995. ISBN 1-55617-516-7. 4.2

[5] Claudine Badue, Rânik Guidolini, Raphael Vivacqua Carneiro, Pedro Azevedo, Vinicius B Cardoso, Avelino Forechi, Luan Jesus, Rodrigo Berriel, Thiago M Paixao, Filipe Mutz, et al. Self-driving cars: A survey. *Expert Systems with Applications*, 165:113816, 2021. 2.1.1

[6] Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5):834–846, 1983. doi: 10.1109/TSMC.1983.6313077. 1

[7] Richard Bellmen. A markovian decision process. *Journal of Mathematics and Mechanics*, 6(5):679–684, 1957. ISSN 00959057, 19435274. URL `http://www.jstor.org/stable/24900506`. 2.2.1

[8] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Christo-

pher Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub W. Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning. *ArXiv*, abs/1912.06680, 2019. 1, 2.2.5

[9] Guillaume Bresson, Zayed Alsayed, Li Yu, and Sébastien Glaser. Simultaneous Localization And Mapping: A Survey of Current Trends in Autonomous Driving. *IEEE Transactions on Intelligent Vehicles*, XX:1, 2017. doi: 10.1109/TIV.2017.2749181. URL https://hal.archives-ouvertes.fr/hal-01615897. 2.1.1

[10] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016. 1

[11] Daniel S Brown, Wonjoon Goo, and Scott Niekum. Better-than-demonstrator imitation learning via automatically-ranked demonstrations. In *Conference on robot learning*, pages 330–359. PMLR, 2020. 2.1.2

[12] Mark Campbell, Magnus Egerstedt, Jonathan P. How, and Richard M. Murray. Autonomous driving in urban environments: approaches, lessons and challenges. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 368(1928):4649–4672, 2010. doi: 10.1098/rsta.2010.0110. URL https://royalsocietypublishing.org/doi/abs/10.1098/rsta.2010.0110. 1, 2.1.1

[13] Dian Chen and Philipp Krähenbühl. Learning from all vehicles. *arXiv preprint arXiv:2203.11934*, 2022. 5.4.4, D.2

[14] Dian Chen, Brady Zhou, Vladlen Koltun, and Philipp Krähenbühl. Learning by cheating. In *Conference on Robot Learning*, pages 66–75. PMLR, 2020. 2.1.2, 3.2.1, 4.4.2, 4.4.3, 5.4.2, 5.4.4

[15] Dian Chen, Vladlen Koltun, and Philipp Krähenbühl. Learning to drive from a world on rails. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 15590–15599, 2021. 2.1.2, 2.1.3, 4.4.2, 4.4.3, 5.4.2, 5.4.4

[16] Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Misha Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. Decision transformer: Reinforcement learning via sequence modeling. *Advances in neural information processing systems*, 34, 2021. 2.2.4

[17] Kashyap Chitta, Aditya Prakash, and Andreas Geiger. Neat: Neural attention fields for end-to-end autonomous driving. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 15793–15803, 2021. 5.4.4

[18] Kaleigh Clary, Emma Tosch, John Foley, and David Jensen. Let's play again:

Variability of deep reinforcement learning agents in atari environments. *arXiv preprint arXiv:1904.06312*, 2019. 4.4.3

[19] Felipe Codevilla, Matthias Müller, Antonio López, Vladlen Koltun, and Alexey Dosovitskiy. End-to-end driving via conditional imitation learning. In *2018 IEEE international conference on robotics and automation (ICRA)*, pages 4693–4700. IEEE, 2018. 2.1.2, 4.4.2, 4.4.3, 5.4.2

[20] Felipe Codevilla, Eder Santana, Antonio M. López, and Adrien Gaidon. Exploring the limitations of behavior cloning for autonomous driving. *CoRR*, abs/1904.08980, 2019. URL `http://arxiv.org/abs/1904.08980`. (document), 2.1.2, 3.1.1, 3.2, 3.2.2, 4.1, 4.4, 4.4.1, 4.4.2, 4.4.3, 4.1, 4.4.3, 4.3, 5.1, 5.4, 5.4.2, 5.4.3, 5.1, 5.4.4

[21] Moises Diaz, Pietro Cerri, Giuseppe Pirlo, Miguel A. Ferrer, and Donato Impedovo. A survey on traffic light detection. In Vittorio Murino, Enrico Puppo, Diego Sona, Marco Cristani, and Carlo Sansone, editors, *New Trends in Image Analysis and Processing – ICIAP 2015 Workshops*, pages 201–208, Cham, 2015. Springer International Publishing. ISBN 978-3-319-23222-5. 3.1.3

[22] Nemanja Djuric, Vladan Radosavljevic, Henggang Cui, Thi Nguyen, Fang-Chieh Chou, Tsung-Han Lin, Nitin Singh, and Jeff Schneider. Uncertainty-aware short-term motion prediction of traffic actors for autonomous driving. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pages 2095–2104, 2020. 2.1.1

[23] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An open urban driving simulator. In Sergey Levine, Vincent Vanhoucke, and Ken Goldberg, editors, *Proceedings of the 1st Annual Conference on Robot Learning*, volume 78 of *Proceedings of Machine Learning Research*, pages 1–16. PMLR, 13–15 Nov 2017. URL `https://proceedings.mlr.press/v78/dosovitskiy17a.html`. (document), 1, 2.1.2, 2.1.3, 3.1.1, 3.1.4, 3.1, 3.2.1, D

[24] Epic Games. Unreal engine. URL `https://www.unrealengine.com`. 3.1.1, 3.1.2

[25] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Vlad Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *International Conference on Machine Learning*, pages 1407–1416. PMLR, 2018. 1, 2.2.5

[26] Lasse Espeholt, Raphaël Marinier, Piotr Stanczyk, Ke Wang, and Marcin Michalski. Seed rl: Scalable and efficient deep-rl with accelerated central inference. *arXiv preprint arXiv:1910.06591*, 2019. 1, 2.2.5

[27] Eric Espié, Christophe Guionneau, Bernhard Wymann, Christos Dimitrakakis,

Rémi Coulom, and Andrew Sumner. Torcs, the open racing car simulator. 2005. 1, 2.1.3

[28] Vladimir Feinberg, Alvin Wan, Ion Stoica, Michael I Jordan, Joseph E Gonzalez, and Sergey Levine. Model-based value estimation for efficient model-free reinforcement learning. *arXiv preprint arXiv:1803.00101*, 2018. 2.2.3

[29] Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International conference on machine learning*, pages 1587–1596. PMLR, 2018. 2.2.2

[30] O. K. Golovnin and R. V. Yarmov. Universal convolutional neural network for recognition of traffic lights and road signs in video frames. In Denis B. Solovev, Grigorios L. Kyriakopoulos, and Terziev Venelin, editors, *SMART Automatics and Energy*, pages 459–468, Singapore, 2022. Springer Singapore. ISBN 978-981-16-8759-4. 3.1.3

[31] Gi-Poong Gwon, Woo-Sol Hur, Seong-Woo Kim, and Seung-Woo Seo. Generation of a precise and efficient lane-level road map for intelligent vehicle systems. *IEEE Transactions on Vehicular Technology*, 66(6):4517–4533, 2016. 2.1.1, 3.1.3

[32] David Ha and Jürgen Schmidhuber. Recurrent world models facilitate policy evolution. *Advances in neural information processing systems*, 31, 2018. 2.2.3

[33] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018. (document), 2.2.2, 4.3, 4.3.1, 4.3.2, 4.4.1, 4.1, 5.1

[34] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, et al. Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905*, 2018. 2.2.2, 4.4.1, 5.1

[35] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968. doi: 10.1109/TSSC.1968.300136. 3.1.4

[36] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016. 2.1.3, 4.4.2, 5.4.4

[37] Nicolas Heess, Dhruva TB, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, Tom Erez, Ziyu Wang, SM Eslami, et al. Emergence of locomotion behaviours in rich environments. *arXiv preprint arXiv:1707.02286*, 2017. 1, 2.2.5

[38] Geoffrey Hinton, Oriol Vinyals, Jeff Dean, et al. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2(7), 2015. 2.1.2, 2.1.3, 4.4.2

[39] Manato Hirabayashi, Adi Sujiwo, Abraham Monrroy, Shinpei Kato, and Masato Edahiro. Traffic light recognition using high-definition map features. *Robotics and Autonomous Systems*, 111:62–72, 2019. ISSN 0921-8890. doi: https://doi.org/10.1016/j.robot.2018.10.004. URL https://www.sciencedirect.com/science/article/pii/S0921889018301234. 3.1.3

[40] Matt Hoffman, Bobak Shahriari, John Aslanides, Gabriel Barth-Maron, Feryal Behbahani, Tamara Norman, Abbas Abdolmaleki, Albin Cassirer, Fan Yang, Kate Baumli, et al. Acme: A research framework for distributed reinforcement learning. *arXiv preprint arXiv:2006.00979*, 2020. 2.2.5

[41] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado Van Hasselt, and David Silver. Distributed prioritized experience replay. *arXiv preprint arXiv:1803.00933*, 2018. 1, 2.2.5

[42] Ahmed Hussein, Mohamed Medhat Gaber, Eyad Elyan, and Chrisina Jayne. Imitation learning: A survey of learning methods. 50(2), apr 2017. ISSN 0360-0300. doi: 10.1145/3054912. URL https://doi.org/10.1145/3054912. 2.1.2

[43] Bernhard Jaeger. Expert drivers for autonomous driving. Master's thesis, University of Tübingen, 2021. 5.4.4

[44] Michael Janner, Qiyang Li, and Sergey Levine. Offline reinforcement learning as one big sequence modeling problem. In *Advances in Neural Information Processing Systems*, 2021. 2.2.4

[45] Ajay Jose, Harish Thodupunoori, and Binoy B Nair. A novel traffic sign recognition system combining viola–jones framework and deep learning. In *Soft Computing and Signal Processing*, pages 507–517. Springer, 2019. 3.1.3

[46] Takeo Kanade, Chuck Thorpe, and William (Red) L. Whittaker. Autonomous land vehicle project at cmu. In *Proceedings of ACM 14th Annual Conference on Computer Science (CSC '86)*, pages 71 – 80, February 1986. 1, 2.1.1

[47] Alex Kendall, Jeffrey Hawke, David Janz, Przemyslaw Mazur, Daniele Reda, John-Mark Allen, Vinh-Dieu Lam, Alex Bewley, and Amar Shah. Learning to drive in a day. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 8248–8254. IEEE, 2019. 2.1.3

[48] Rahul Kidambi, Aravind Rajeswaran, Praneeth Netrapalli, and Thorsten Joachims. Morel: Model-based offline reinforcement learning. *Advances in neural information processing systems*, 33:21810–21823, 2020. 2.2.4

[49] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization.

CoRR, abs/1412.6980, 2015. A.1

[50] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013. 2.1.3

[51] Vijay Konda and John Tsitsiklis. Actor-critic algorithms. *Society for Industrial and Applied Mathematics*, 42, 04 2001. 2.2.2

[52] Aviral Kumar, Aurick Zhou, George Tucker, and Sergey Levine. Conservative q-learning for offline reinforcement learning. *Advances in Neural Information Processing Systems*, 33:1179–1191, 2020. 2.2.4

[53] Stephanie Lefevre, Dizan Vasquez, and Christian Laugier. A survey on motion prediction and risk assessment for intelligent vehicles. *Robomech Journal*, 1, 07 2014. doi: 10.1186/s40648-014-0001-z. 2.1.1

[54] Jesse Levinson, Jake Askeland, Jan Becker, Jennifer Dolson, David Held, Soeren Kammel, J. Zico Kolter, Dirk Langer, Oliver Pink, Vaughan Pratt, Michael Sokolsky, Ganymed Stanek, David Stavens, Alex Teichman, Moritz Werling, and Sebastian Thrun. Towards fully autonomous driving: Systems and algorithms. In *2011 IEEE Intelligent Vehicles Symposium (IV)*, pages 163–168, 2011. doi: 10.1109/IVS.2011.5940562. 1, 2.1.1

[55] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 583–598, USA, 2014. USENIX Association. ISBN 9781931971164. (document), 1, 2.2.5, 4.1, 4.3, 4.1, 4.3.1

[56] Xiaodan Liang, Tairui Wang, Luona Yang, and Eric Xing. Cirl: Controllable imitative reinforcement learning for vision-based self-driving. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 584–599, 2018. 2.1.2, 4.4.2

[57] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015. 2.1.3, 2.2.2, 4.4.2

[58] Rong Liu, Jinling Wang, and Bingqi Zhang. High definition map for automated driving: Overview and analysis. *Journal of Navigation*, 73(2):324–341, 2020. doi: 10.1017/S0373463319000638. 3.1.3

[59] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013. 2.2.2, 4.4.2

[60] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1928–1937, New York, New York, USA, 20–22 Jun 2016. PMLR. URL https://proceedings.mlr.press/v48/mniha16.html. 1, 2.1.3, 2.2.2, 2.2.5, 5.1

[61] Michael Montemerlo, Jan Becker, Suhrid Bhat, Hendrik Dahlkamp, Dmitri Dolgov, Scott Ettinger, Dirk Haehnel, Tim Hilden, Gabe Hoffmann, Burkhard Huhnke, Doug Johnston, Stefan Klumpp, Dirk Langer, Anthony Levandowski, Jesse Levinson, Julien Marcil, David Orenstein, Johannes Paefgen, Isaac Penny, Anna Petrovskaya, Mike Pflueger, Ganymed Stanek, David Stavens, Antone Vogt, and Sebastian Thrun. Junior: The stanford entry in the urban challenge. *Journal of Field Robotics*, 25(9):569–597, 2008. doi: https://doi.org/10.1002/rob.20258. 1, 2.1.1

[62] Sajjad Mozaffari, Omar Y Al-Jarrah, Mehrdad Dianati, Paul Jennings, and Alexandros Mouzakitis. Deep learning-based vehicle behavior prediction for autonomous driving applications: A review. *IEEE Transactions on Intelligent Transportation Systems*, 23(1):33–47, 2020. 2.1.1

[63] Farzeen Munir, Shoaib Azam, Muhammad Hussain, Ahmed Sheri, and Moongu Jeon. Autonomous vehicle: The architecture aspect of self driving car. pages 1–5, 10 2018. ISBN 978-1-4503-6620-5. doi: 10.1145/3290589.3290599. 1, 2.1.1

[64] Anusha Nagabandi, Gregory Kahn, Ronald S Fearing, and Sergey Levine. Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7559–7566. IEEE, 2018. 2.2.3

[65] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, et al. Massively parallel methods for deep reinforcement learning. *arXiv preprint arXiv:1507.04296*, 2015. 1, 2.2.5

[66] National Imagery and Mapping Agency. Department of defense world geodetic system 1984: its definition and relationships with local geodetic systems. Technical Report TR8350.2, National Imagery and Mapping Agency, St. Louis, MO, USA, January 2000. URL http://earth-info.nga.mil/GandG/publications/tr8350.2/tr8350_2.html. 3.1.2

[67] Man-Suk Oh and James O. Berger. Integration of multimodal functions by monte carlo importance sampling. *Journal of the American Statistical Association*, 88(422):450–456, 1993. doi: 10.1080/01621459.1993.10476295. URL https://www.tandfonline.com/doi/abs/10.1080/01621459.1993.10476295. 5.3.2

[68] Takayuki Osa, Joni Pajarinen, Gerhard Neumann, J Andrew Bagnell, Pieter Abbeel, and Jan Peters. An algorithmic perspective on imitation learning. *arXiv preprint arXiv:1811.06711*, 2018. 2.1.2

[69] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`. 4.1

[70] B. T. Polyak and A. B. Juditsky. Acceleration of stochastic approximation by averaging. *SIAM Journal on Control and Optimization*, 30(4):838–855, 1992. doi: 10.1137/0330046. URL `https://doi.org/10.1137/0330046`. 4.4.1

[71] Aditya Prakash, Kashyap Chitta, and Andreas Geiger. Multi-modal fusion transformer for end-to-end autonomous driving. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7077–7087, 2021. 5.4.4, 5.4.5, A.4

[72] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. *Advances in neural information processing systems*, 24, 2011. 2.2.5

[73] Daniel E. Rivera, Manfred Morari, and Sigurd Skogestad. Internal model control: Pid controller design. *Industrial & Engineering Chemistry Process Design and Development*, 25(1):252–265, 1986. doi: 10.1021/i200032a041. URL `https://doi.org/10.1021/i200032a041`. 4.2

[74] Nikita Rudin, David Hoeller, Philipp Reist, and Marco Hutter. Learning to walk in minutes using massively parallel deep reinforcement learning. In *Conference on Robot Learning*, pages 91–100. PMLR, 2022. 1

[75] G. Rummery and Mahesan Niranjan. On-line q-learning using connectionist systems. *Technical Report CUED/F-INFENG/TR 166*, 11 1994. 2.2.2

[76] Axel Sauer, Nikolay Savinov, and Andreas Geiger. Conditional affordance learning for driving in urban environments. In *Conference on Robot Learning*, pages 237–252. PMLR, 2018. 4.4.2, 4.4.3, 5.4.2

[77] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015. 2.2.5

[78] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp

Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015. 1, 2.2.2, 5.1

[79] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017. (document), 1, 2.2.2, 4.4.4, 5.1, 5.2, 5.3.2, 5.3.2, 5.3.2

[80] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, pages 387–395. JMLR.org, 2014. 2.2.2

[81] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018. doi: 10.1126/science.aar6404. 1, 2.2.3, 2.2.5

[82] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. 4.4.2

[83] Adam Stooke and Pieter Abbeel. Accelerated methods for deep reinforcement learning. *arXiv preprint arXiv:1803.02811*, 2018. 1, 2.2.5

[84] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, 2012. doi: 10.1109/IROS.2012.6386109. 1

[85] Marin Toromanoff, Emilie Wirbel, and Fabien Moutarde. End-to-end model-free reinforcement learning for urban driving using implicit affordances. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 7153–7162, 2020. 1, 2.1.3, 3.2.1, 4.4.2, 4.4.3, 5.4.2, 5.4.4

[86] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016. 2.2.2

[87] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL `https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf`. 5.4.4

[88] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja

Huang, L. Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander Sasha Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom Le Paine, Caglar Gulcehre, Ziyun Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy P. Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, pages 1–5, 2019. 1, 2.2.5, D.2

[89] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016. 2.2.2

[90] Théophane Weber, Sébastien Racaniere, David P Reichert, Lars Buesing, Arthur Guez, Danilo Jimenez Rezende, Adria Puigdomenech Badia, Oriol Vinyals, Nicolas Heess, Yujia Li, et al. Imagination-augmented agents for deep reinforcement learning. *arXiv preprint arXiv:1707.06203*, 2017. 2.2.3

[91] Erik Wijmans, Abhishek Kadian, Ari Morcos, Stefan Lee, Irfan Essa, Devi Parikh, Manolis Savva, and Dhruv Batra. Dd-ppo: Learning near-perfect pointgoal navigators from 2.5 billion frames. *arXiv preprint arXiv:1911.00357*, 2019. 1, 2.2.5

[92] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992. 2.2.2, 5.1

[93] Tianhe Yu, Garrett Thomas, Lantao Yu, Stefano Ermon, James Y Zou, Sergey Levine, Chelsea Finn, and Tengyu Ma. Mopo: Model-based offline policy optimization. *Advances in Neural Information Processing Systems*, 33:14129–14142, 2020. 2.2.4

[94] Ekim Yurtsever, Jacob Lambert, Alexander Carballo, and Kazuya Takeda. A survey of autonomous driving: Common practices and emerging technologies. *IEEE access*, 8:58443–58469, 2020. 1, 2.1.1

[95] Zhenyu Zhang, Xiangfeng Luo, Tong Liu, Shaorong Xie, Jianshu Wang, Wei Wang, Yang Li, and Yan Peng. Proximal policy optimization with mixed distributed training. In *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 1452–1456, 2019. doi: 10.1109/ICTAI.2019.00206. 1, 2.2.5