Journal of Artificial Intelligence Research 70 (2021) 1517-1555 Submitted 11/2020; published 04/2021

MADRaS : Multi Agent Driving Simulator

Anirban Santara NRBNSNTR@GMAIL.COM
Department of Computer Science and Engineering,
Indian Institute of Technology Kharagpur, Kharagpur, WB, India

Sohan Rudra SOHANRUDRA@IITKGP.AC.IN
Department of Mathematics,
Indian Institute of Technology Kharagpur, Kharagpur, WB, India

Sree Aditya Buridi BURIDIADITYA@IITKGP.AC.IN
Department of Computer Science and Engineering,
Indian Institute of Technology Kharagpur, Kharagpur, WB, India

Meha Kaushik MEHA.KAUSHIK@MICROSOFT.COM
Microsoft, Vancouver, Canada

Abhishek Naik ABHISHEK.NAIK@UALBERTA.CA
Department of Computing Science, University of Alberta,
Alberta, Canada

Bharat Kaul BHARAT.KAUL@INTEL.COM
Parallel Computing Lab, Intel Labs, Intel, Bengaluru, KA, India

Balaraman Ravindran RAVI@CSE.IITM.AC.IN
Robert Bosch Center for Data Science and Artificial Intelligence,
Indian Institute of Technology Madras, Chennai, TN, India

Abstract

Autonomous driving has emerged as one of the most active areas of research as it has the promise of making transportation safer and more efficient than ever before. Most real-world autonomous driving pipelines perform perception, motion planning and action in a loop. In this work we present MADRaS, an open-source multi-agent driving simulator for use in the design and evaluation of motion planning algorithms for autonomous driving. Given a start and a goal state, the task of motion planning is to solve for a sequence of position, orientation and speed values in order to navigate between the states while adhering to safety constraints. These constraints often involve the behaviors of other agents in the environment. MADRaS provides a platform for constructing a wide variety of highway and track driving scenarios where multiple driving agents can be trained for motion planning tasks using reinforcement learning and other machine learning algorithms. MADRaS is built on TORCS, an open-source car-racing simulator. TORCS offers a variety of cars with different dynamic properties and driving tracks with different geometries and surface properties. MADRaS inherits these functionalities from TORCS and introduces support for multi-agent training, inter-vehicular communication, noisy observations, stochastic actions,

©2021 AI Access Foundation. All rights reserved.

and custom traffic cars whose behaviors can be programmed to simulate challenging traffic conditions encountered in the real world. MADRaS can be used to create driving tasks whose complexities can be tuned along eight axes in well defined steps. This makes it particularly suited for curriculum and continual learning. MADRaS is lightweight and it provides a convenient OpenAI Gym interface for independent control of each car. Apart from the primitive steering-acceleration-brake control mode of TORCS, MADRaS offers a hierarchical track-position – speed control mode that can potentially be used to achieve better generalization. MADRaS uses a UDP based client server model where the simulation engine is the server and each client is a driving agent. MADRaS uses multiprocessing to run each agent as a parallel process for efficiency and integrates well with popular reinforcement learning libraries like RLLib. We show experiments on single and multi-agent reinforcement learning with and without curriculum.

1. Introduction

Inefficient driving habits of humans result in accidents, congestion and environmental pollution. These issues can be addressed efficiently if cars are able to operate autonomously. Additionally, humans lose hours of productivity in their cars towards their daily commute. These possibilities have, of late, spurred an unprecedented amount of interest towards self-driving car technology from researchers around the world.

Although realization of fully autonomous driving seems far flung, some specific low level tasks pertaining to driving such as adaptive cruise control, lane keep assistance and parking assistance have already been automated at a production scale in the form of Advanced Driver-Assistance Systems (ADAS) (Dikmen & Burns, 2016; Minster, Haghighat, Chu, & Vogt, 2018). Safe, optimal and fast motion planning in complex, multi-modal, multi-agent, and partially observable environments is the foremost technological challenge towards achieving full autonomy. Achieving these goals tractably using traditional motion planning algorithms – like Model Predictive Control, RRT, A*, and Dijkstra – is only possible under certain simplifying assumptions on the complexity the environment (LaValle, 2006). On the other hand, Machine Learning based approaches including Reinforcement Learning (RL) (Sutton & Barto, 2018) and Learning from Demonstration (LfD) (Argall, Chernova, Veloso, & Browning, 2009) are capable of fast, reactive control under fewer assumptions (Shalev-Shwartz, Shammah, & Shashua, 2016; Bojarski, Yeres, Choromanska, Choromanski, Firner, Jackel, & Muller, 2017; Sharifzadeh, Chiotellis, Triebel, & Cremers, 2016; You, Lu, Filev, & Tsiotras, 2019). However the training phase of these algorithms is often data-hungry (Fayjie, Hossain, Oualid, & Lee, 2018; Talpaert., Sobh., Kiran., Mannion., Yogamani., El-Sallab., & Perez., 2019) especially for those using highly expressive and complex models like deep neural networks. RL based methods also require online interaction with the environment that entails risk (Shalev-Shwartz & Shashua, 2016; Santara, Naik, Ravindran, Das, Mudigere, Avancha, & Kaul, 2018). Driving simulators attempt to address these problems by rendering realistic driving conditions and traffic patterns in which agents can collect training data many times faster than real time. They also provide a sandbox environment where the agent can run into catastrophic situations while learning to drive without causing physical damage in the real world.

Real world driving scenarios have a high degree of variability and require the driver to optimize for multiple – often conflicting – objectives depending on the situation they are in. Curriculum learning (Bengio, Louradour, Collobert, & Weston, 2009) and continual learning (Parisi, Kemker, Part, Kanan, & Wermter, 2019) are two families of machine learning algorithms that are relevant in this case. Curriculum learning provides a way of learning complex skills efficiently by breaking up the problem into a hierarchy of sub-tasks and learning to accomplish them in the order of increasing complexity. Continual learning on the other hand deals with learning to accomplish new tasks without forgetting previously acquired skills. A simulator for curriculum and continual learning of autonomous driving agents should be able to create a large variety of driving scenarios with fine-grained control on their complexities.

Since the early days of autonomous driving research, simulators have been used in the development of different parts of the perceive-plan-act pipeline (Sulkowski, Bugiel, & Izydorczyk, 2018). Most of these simulators cater to the task of perception. Back in 1989, the creator of ALVINN, Dean A. Pomerleau, had used a simulator to generate training images for road detection (Pomerleau, 1989). Thanks to the recent advances in computer graphics, modern driving simulators and games like Grand Theft Auto – V [1] can render photo-realistic driving scenes with accurate depiction of illumination, weather and other physical phenomena. They also simulate real-life sensors that can be used to collect synthetic data from these scenes to augment real-world driving datasets. Recent works (Chen, Seff, Kornhauser, & Xiao, 2015; Richter, Vineet, Roth, & Koltun, 2016; Richter, Hayder, & Koltun, 2017; Ros, Sellart, Materzynska, Vazquez, & Lopez, 2016) have demonstrated that training perception algorithms on these augmented datasets result in better generalization in the real world that is crucial for safe and reliable autonomous driving. Most notable open-source driving simulators in this category are CARLA (Dosovitskiy, Ros, Codevilla, Lopez, & Koltun, 2017), Microsoft AirSim (Shah, Dey, Lovett, & Kapoor, 2018), DeepDrive.io and Udacity's Self Driving Car Simulator (Brown et al., 2018). These simulators can, in principle, be also used for planning tasks. However, an agent learning to face real world driving scenarios must learn to be invariant to road geometries, traffic patterns and vehicular dynamics. These simulators do not offer enough variability along these dimensions that is necessary to learn the invariances. In a typical driving scene, multiple entities (cars, buses, bikes, and pedestrians) try to achieve their objectives of getting from one place to another fast, yet safely and reliably. A simulator for such an environment should provide an easy way to create arbitrary traffic configurations. The task of negotiating in traffic is akin to finding the winning strategy in a multi-agent game (Dresner & Stone, 2008). Hence, an autonomous driving simulator should be able to simulate different varieties of traffic and support multiple agents learning to negotiate and drive through cooperation and competition. Among the aforementioned simulators, AirSim, DeepDrive.io and Udacity provide some preset driving conditions mostly without traffic. They do not provide any straightforward way to create custom traffic or train multiple agents. CARLA does provide an API for

[1] Available online at `https://www.rockstargames.com/V/`

independent control of cars that can be used for multi-agent training and creating custom traffic cars. However, most of the variability presented by CARLA is in the perceived inputs and not in the behavioral dynamics of the ego-vehicle or the traffic agents. This motivated us to develop a dedicated simulator for learning to plan in autonomous driving with a focus on learning invariances to road geometries, traffic patterns and vehicular dynamics in both single and multi-agent learning settings.

In this paper we present MADRaS, a *M*ulti-*A*gent *DR*iving *S*imulator for motion planning in autonomous driving and demonstrate its ability to create driving scenarios with high degrees of variability. We present results of training reinforcement learning agents to accomplish challenging tasks like driving vehicles with drastically different dynamics, maneuvering through a variety of track geometries, navigating through a narrow road avoiding collisions with both moving and parked traffic cars and making two cars learn to cooperate and pass through a traffic bottleneck. We also demonstrate how curriculum learning can help in reducing the sample complexity of some of these tasks. Built on top of the TORCS platform (Wymann, Espié, Guionneau, Dimitrakakis, Coulom, & Sumner, 2000), MADRaS uses simplified physics simulation and representative graphics to reduce the computational overhead for perception and action. It allows for the addition of learning and non-learning cars to a driving scene to create custom traffic configurations and train multiple agents simultaneously. Each driving agent gets a high-level object-oriented representation of the world as observation and an OpenAI gym (Brockman, Cheung, Pettersson, Schneider, Schulman, Tang, & Zaremba, 2016) interface for independent control. MADRaS is open source[2] and aims to contribute to the democratization of artificial intelligence.

The rest of the paper is organised as follows. Section 2 introduces the theoretical concepts that guide the organization of MADRaS. Section 3 describes our contributions in this project in detail. Section 4 presents six experimental studies that highlight the ability of MADRaS to simulate driving tasks of high variance. Finally, Section 5 concludes the paper with scopes of future work.

2. Background

In this section, we introduce the concepts of Markov Decision Process (MDP), Markov Game (MG), Reinforcement Learning (RL) and Episodic Learning that comprise the foundation of MADRaS.

2.1 Reinforcement Learning in Markov Games

Markov Decision Process (MDP) is a mathematical construct that is commonly used in the artificial intelligence literature to describe an environment in which agents learn to act (Sutton & Barto, 2018). In a single-agent learning set-up, an MDP can be expressed as a 4-tuple: $\mathcal{M} = \langle S, A, P, R \rangle$. It consists of a state space S, an action space A, a transition

[2] Code available at `https://github.com/madras-simulator/MADRaS`

dynamics function $P : S \times A \times S \rightarrow [0, 1]$ that gives the probability distribution over next states for each action taken in a given state and a reward function $R : S \times A \rightarrow \mathbb{R}$ that qualifies the task at hand. An agent learning to act in this environment receives observations about the current state and samples actions from its policy $\pi : S \times A \rightarrow [0, 1]$ which is a conditional distribution over A given a state in S. The reward function R gives scalar feedback about these actions that indicate the agent's progress towards the goal. The agent optimizes the parameters of its policy to maximize the cumulative reward received from the environment. This form of learning through trial and error with feedback from the environment is known as Reinforcement Learning (RL).

In a multi-agent reinforcement learning set up, the environment is described as a Markov Game (MG) which is a generalization of MDP to capture the interplay of multiple agents (Littman, 1994; Bu, Babu, De Schutter, et al., 2008; Bowling & Veloso, 2000; Da Silva & Costa, 2019; Lin, Beling, & Cogill, 2017; Yu, Song, & Ermon, 2019; Lin, Adams, & Beling, 2018). An MG is a tuple $\langle S, \{\alpha_i\}_{i=1}^{n}, \{A_i\}_{i=1}^{n}, P, \{R_i\}_{i=1}^{n} \rangle$. Here, $\{\alpha_i\}_{i=1}^{n}$ denotes a set of n agents that simultaneously learn to act in an environment with state space S and transition dynamics function P. A_i and R_i denote the set of actions and reward function for agent α_i.

2.2 Episodic Learning

In episodic learning (Seel, 2011), an agent's experience happens in the form of episodes. Each episode begins with the agent in one of the initial states of the environment. The state of the environment changes in response to the agent's actions and the episode ends when the environment sends a *done* signal to the agent. In a general multi-agent learning setting, the environment may send a done signal to each agent separately at different time steps resulting in different episode lengths for each agent. When the episodes of all the agents end, the environment resets itself to one of its initial states and starts new episodes for each agent.

3. MADRaS Simulator

In this section we describe the structure and organization of the MADRaS simulator which constitutes the main contribution of this paper. The current version of MADRaS is focused on track driving. Track driving is traditionally used in the automotive world to benchmark driver skill and car agility. We first present a brief overview of the TORCS simulator and associated prior works that MADRaS builds upon. Then we describe the new features that we develop in this project and present a thorough empirical analysis of their relevance in the context of planning in autonomous driving.

3.1 TORCS Simulator

MADRaS is based on TORCS which stands for The Open Racing Car Simulator (Wymann et al., 2000). It is capable of simulating the essential elements of vehicular dynamics such as mass, rotational inertia, collision, mechanics of suspensions, links and differentials, friction and aerodynamics. Physics simulation is simplified and is carried out through Euler

Table 1: Comparison of Gym TORCS (Yoshida, 2016) with MADRaS

Feature	**Gym TORCS**	**MADRaS**
scr-server architecture	✓	✓
observation noise	×	✓
stochastic outcomes of actions	×	✓
parallel rollout support	×	✓
multi-agent training	×	✓
inter-vehicular communication	×	✓
custom traffic cars	×	✓
domain randomization	×	✓
centralized configuration	×	✓
modular reward and done functions	×	✓
hierarchical action space	×	✓

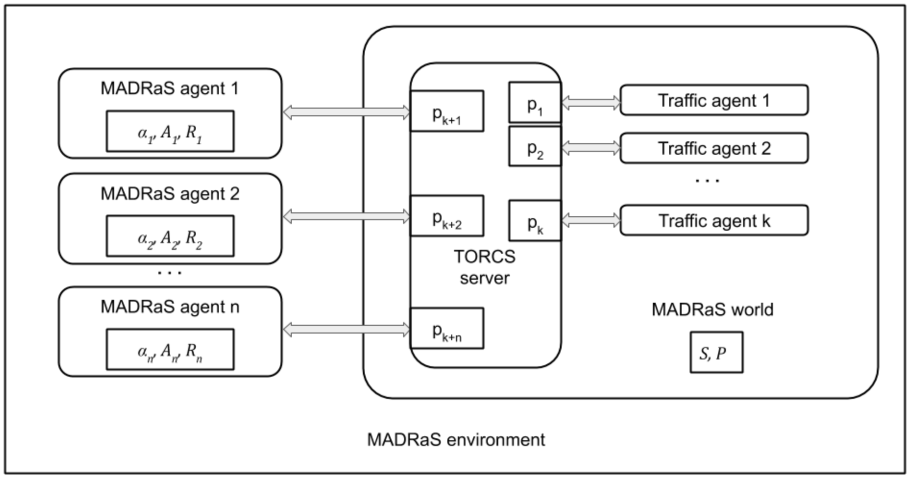


Figure 1: Architecture of the MADRaS simulation environment. Each double headed arrow indicates one UDP communication channel between the TORCS server and one of the clients (traffic or MADRaS agents). The server listens to the i^{th} client through a dedicated port denoted by p_i in the figure. MADRaS assigns these ports in order, first to the traffic agents and then to the learning agents. The Markov Game terms are also marked in their respective places of definition in the figure.

integration of differential equations at a temporal discretization level of 0.002 seconds. The rendering pipeline is lightweight and based on OpenGL (Neider, Davis, & Woo, 1993) that can be turned off for faster training. TORCS offers a large variety of tracks and cars as free assets that we discuss later in this section. It also provides a number of programmed robot cars with different levels of performance that can be used to benchmark the performance of human players and software driving agents. TORCS was built with the goal of developing Artificial Intelligence for vehicular control and has been used extensively by the machine learning community ever since its inception (Li, Song, & Ermon, 2017; Lillicrap, Hunt, Pritzel, Heess, Erez, Tassa, Silver, & Wierstra, 2019; Loiacono, Prete, Lanzi, & Cardamone, 2010b; Koutník, Cuccu, Schmidhuber, & Gomez, 2013; Koutník, Schmidhuber, & Gomez, 2014; Onieva, Cardamone, Loiacono, & Lanzi, 2010).

3.2 SCR Server-Client Architecture

The Simulated Car Racing (SCR) Championship (Loiacono, Lanzi, Togelius, Onieva, Pelta, Butz, Lonneker, Cardamone, Perez, Sáez, et al., 2010a) is an annual car-racing competition where participants submit controllers for racing in the TORCS environment. It provides a software patch for TORCS known as `scr_server` (Loiacono, Cardamone, & Lanzi, 2013). It sets up a UDP based client-server architecture in which the competing cars can operate independent of one another. The server runs the TORCS simulator. Each client represents a car that runs as a separate process and communicates with the server through a dedicated UDP port. The patch also provides a layer of abstraction over TORCS in which each car has access to an egocentric view of the environment and not the entire game state. The server polls actions from the clients and updates the game-state every 0.02 seconds of simulated time. The official build of TORCS supports up to 10 SCR clients at a time but with modifications like in (Kaushik, Prasad, Krishna, & Ravindran, 2018) the number of clients can be increased arbitrarily.

3.3 GymTORCS Environment

GymTORCS (Yoshida, 2016) is an OpenAI Gym (Brockman et al., 2016) wrapper for SCR cars built for use in Reinforcement Learning experiments. It uses a custom library called *Snake Oil* to create a client for communicating with the TORCS server through the `scr_server` interface. Snake Oil also provides plug-ins for automatic-transmission, traction control and throttle control which can be used to provide different control modes to the driving agent. GymTORCS is popular in the reinforcement learning community for experiments on driving tasks (Kaushik et al., 2018; Liu, Siravuru, Prabhakar, Veloso, & Kantor, 2017; de Bruin, Kober, Tuyls, & Babuška, 2018; Dossa, Lian, Nomoto, Matsubara, & Uehara, 2019). MADRaS builds on GymTORCS by increasing its stability and ease of use and adding features like multi-agent training and custom traffic cars.

3.4 MADRaS: Multi-Agent Driving Simulator

Having described TORCS and associated prior works that form the foundation of MADRaS, we now present our contributions in this project. As GymTORCS is pre-dominantly designed for single-agent training, the environment is inherently structured as an MDP. This restricts its usage for multi-agent training. MADRaS is GymTORCS restructured as an MG with some added functionalities (see Table 1). Figure 1 describes the architecture of MADRaS. *MADRaS Environment* consists of a *MADRaS World* and a given number of *MADRaS Agents* ($\{\alpha_i\}_i$). *MADRaS World* consists of a TORCS server and a given number of traffic agents each of which executes an independently configured behavior. The state space (S) and the transition dynamics (P) of the MG are defined by the *MADRaS World*. Each *MADRaS Agent* α_i runs as an SCR Client with a modified Snake Oil interface that has its own action space A_i and reward function R_i which are independent of the action spaces and reward functions of the other agents. Unlike GymTORCS, *MADRaS Agents* can not reset the TORCS server. This allows for multiple agents to complete their episodes independently. *MADRaS Environment* resets its *MADRaS World* and in turn its TORCS server when all the agents have terminated their episodes. MADRaS also provides a number of ways to configure the initial state of the environment for the task at hand. The initial distance from the start line and position with respect to the track edges can be specified individually for both the learning cars as well as the traffic agents. Thus MADRaS harnesses the full potential of the SCR server-client architecture and enables multi-agent training. We describe the salient features of MADRaS in the remaining part of this section.

3.4.1 Traffic Agents

MADRaS introduces support for adding non-learning traffic agents in the environment that execute a pre-defined behaviour. These are different from the robot cars that come bundled with TORCS for benchmarking racing agents. MADRaS provides a base class that can be used as template to create traffic cars with interesting behavioral patterns and some sample traffic classes as free assets (see Table 2). The base class also comes equipped with methods to prevent collision and going out of track. Each traffic agent runs as a parallel process independent of the learning agent and has an SCR client that talks to the TORCS server through a dedicated port. MADRaS takes care of the configuration and assignment of a requisite number of server ports for connecting all the learning and traffic agents properly at the start of each episode. The number and behavior of traffic agents can be varied between episodes.

3.4.2 Tracks

One of the major advantages of TORCS as the platform of choice for building MADRaS is the availability of a large number of tracks with different geometric (see Figure 2) and surface properties. At the time of writing this paper, TORCS offers 9 oval, 21 road, and 8 dirt tracks. It also offers a software package[3] that can be used to create different variants

[3] Official track-editor package of TORCS: `http://www.berniw.org/trb/download/trackeditor-0.6.2c.tar.bz2`

Table 2: Sample traffic agents in MADRaS.

Name	Behaviour
`ConstVelTrafficAgent`	Drives at a given speed at a given track-position.
`SinusoidalSpeedAgent`	Varies the speed sinusoidally while driving at a given track-position.
`RandomLaneSwitchAgent`	Agent switches lanes randomly while driving.
`DriveAndParkAgent`	Agent drives to a given distance and track-position and parks itself.
`ParkedAgent`	Agent remains parked at a given distance and track-position throughout.
`RandomStoppingAgent`	Agent halts randomly while driving.

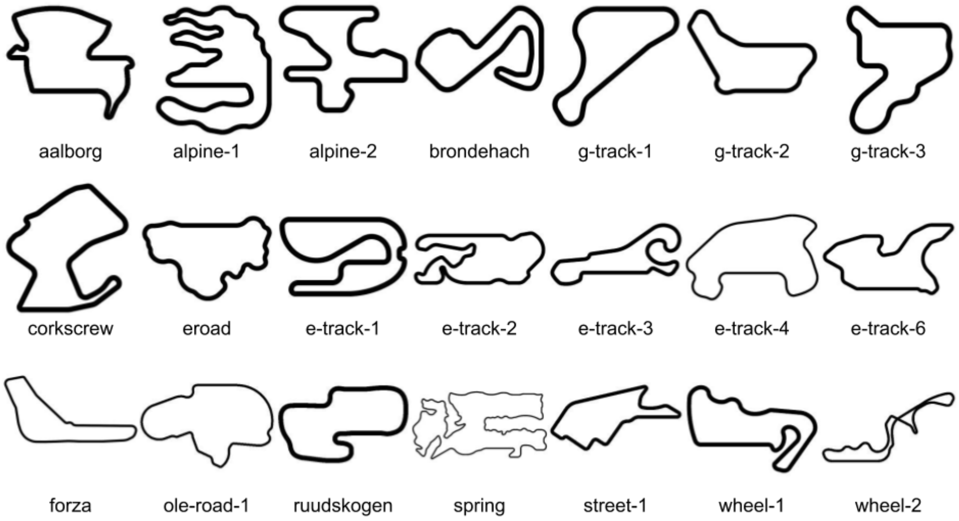


Figure 2: Schematic diagrams of road tracks in TORCS (Wymann et al., 2000).

of these tracks. MADRaS inherits these free assets from the TORCS project. A limitation of GymTORCS is that a track chosen at the beginning of a training experiment remains fixed throughout. This often causes the agent to memorize the track resulting in poor generalization. MADRaS ameliorates this by introducing an option to select a track at the beginning of each episode. Thus the agent can be exposed to multiple tracks during training.

3.4.3 Car Models

TORCS provides 42 car models with a wide range of dynamic properties. However, GymTORCS only supports a single default car type named `car1-trb1`. MADRaS is capable of changing cars at the beginning of each training episode. Thus it makes it possible to train an agent to drive cars with drastically different dynamic properties. Also, the learning and traffic agents can be assigned different car types for visual distinction.

3.4.4 Modular Configuration

As Reinforcement Learning (RL) is one of the most powerful and actively researched approaches for robot motion planning, MADRaS has some features tailor-made for that purpose. The exercise of tuning an RL algorithm for a given task usually involves tweaking the reward function and episode termination ("done") criteria. It is important to keep accurate track of these parameters across experiments to be able to arrive at the optimal training configuration. GymTORCS has particularly poor configurability as it requires the user to make changes in the Python source code which are difficult to keep track of. The entire MADRaS environment including the initial state and the reward and done functions are configurable through a single file named `madras_config.yml`. A copy of this configuration file can be saved in the training directory for effortless tracking across experiments. Please refer to Appendix A for a detailed discussion on commonly used configuration variables in `madras_config.yml` and their functions. Appendix C explains how the initial state can be configured for each episode.

The reward and done functions are usually composed of multiple parts that try to capture events like arrival at the goal state, crashes and damages. Modularity of these definitions in code is essential for fast iteration. MADRaS provides `MadrasReward` and `MadrasDone` base classes as templates for defining the components of the reward and done functions. Specifying a reward or done function in MADRaS is as simple as listing the names of their components in the configuration file. Each MADRaS Agent comes with a `reward_handler` and a `done_handler` that organize the listed components and set up the corresponding functions. This modular architecture makes it easy to define new reward and done functions and plug them in and out of experiments easily.

3.4.5 Observation Space

The Snake Oil library of GymTORCS provides a parser for the state information returned by the TORCS server. These state variables include odometry, range data, obstacle detection, engine statistics and metadata regarding the position of the ego vehicle relative to the other cars on the road. Such a high-level representation of the world is common in practical autonomous driving pipelines (Bansal, Krizhevsky, & Ogale, 2018) as it helps in decoupling the perception and planning modules allowing them to be improved independently and also reduces the sample complexity of machine learning based planning algorithms (Shalev-Shwartz & Shashua, 2016). Raw visual inputs in the form of a stream of images are also available. For a full list of state variables please refer to the Simulated Car Race Championship paper (Loiacono et al., 2013). The observation vector of a MADRaS agent is composed of a selection of these normalized state variables. For modularity and ease of configuration, MADRaS provides an `observation_handler` class that can toggle between different sets of observed variables. The observations can optionally be made noisy to simulate a partially observed driving scenario.

3.4.6 Action Space

The Snake Oil library allows GymTORCS agents to control cars via steering, acceleration and brake commands. MADRaS inherits this primitive control mode and provides a generalised interface that supports both hierarchical and non-hierarchical controllers. We show experiments with both kinds of controllers and compare their relative performances in this paper. The hierarchical controller used in our experiment implements a track-position – speed control mode. In track-position – speed control mode, a MADRaS agent produces its desired position with respect to the left and right edges of the track and its desired speed. A low-level controller takes these non-primitive actions (*desires*) as inputs and calculates a sequence of steering, acceleration and brake commands. The architecture of MADRaS does not restrict the class of low-level controllers. We use a simple PID controller in the experiments presented in this paper and we plan to add more tuned low level controllers to the repository in the future.

The PID controller used in our experiments works in feedback mode over a number of time steps denoted by `PID_latency`. The `PID_latency` controls the relative time scales of the higher and lower level action spaces. The following is the expression of a PID controller for control variable u.

$$u(t) = K_p e(t) + K_i \int_0^t e(t')dt' + K_d \frac{de(t)}{dt} \tag{1}$$

K_p, K_i and K_d are the constants for the proportional, integral and derivative terms respectively. Appendix B gives a detailed account of the implementation and behavior of the PID controller. The track-position – speed action space is inspired by (Shalev-Shwartz et al., 2016), where the authors note that training an RL agent to generate high-level desires while relegating the low-level implementation of the desires to an analytical controller like PID significantly reduces real world risk and increases the explainability of the agent's behavior. High level actions have also been reported to show better generalizability across vehicular platforms (Behere & Törngren, 2016). All actions are normalized between -1 and 1 for ease of optimization of neural network policies. The outcomes of the agent's actions can optionally be made stochastic. MADRaS implements this stochasticity by adding zero-mean Gaussian noise to actions before sending them to the TORCS server.

3.4.7 Inter-vehicular Communication

The most salient feature of MADRaS is its support for multi-agent training. The success of multi-agent learning is contingent on the ability of the agents to communicate among themselves and plan actions taking into account the states and actions of the other agents (Lowe, Wu, Tamar, Harb, Abbeel, & Mordatch, 2017). MADRaS provides a highly flexible framework for inter-vehicular communication through a communication buffer and an agent mapping function. The agent mapping function allows the user to specify a list of variables that the i^{th} agent wants to observe from the j^{th} agent. The communication buffer records these shared variables from the step $t-1$ and makes them a part of the agents' observation

Table 3: Parameters of the PID controller used in our experiments.

	K_p	K_i	K_d
acceleration PID	10.5	0.05	2.8
steering PID	5.1	0.001	0.000001

vectors at step t.

3.4.8 Curriculum Design for Driving Agents

MADRaS has been designed to provide a playground for reinforcement learning agents to learn to drive any car on any track in any kind of traffic within the TORCS environment. In order to construct a driving problem of high variance, MADRaS can present an agent with a different car to drive in a different track with a different number of traffic cars of different behaviors chosen randomly or in a given order in every training episode. MADRaS can also produce additional stochasticity by making the outcome of an action probabilistic. Training deep neural network policies in high variance environments poses a highly non-convex problem that is difficult to optimize. Curriculum learning (Bengio et al., 2009) has been shown to be effective in reducing the sample complexity in such problems. Curriculum learning involves training an agent on a sequence of tasks of increasing complexity. MADRaS is designed with curriculum learning in mind. The complexity of the driving task in MADRaS can be systematically increased in well defined steps along the following eight dimensions:

1. Number of learning agents.
2. Number of cars to be presented to the agent to drive.
3. Number of tracks to be presented to the agent to drive.
4. Number of traffic agents.
5. Level of obstructive behavior from the traffic agents.
6. Target speed of the learning agent(s).
7. Degree of stochasticity to action-outcomes.
8. Presence of noise in observations.

In the following section we present a set of experiments to highlight the key features of MADRaS.

4. Experiments

In this section we present the results of six experiments on single and multi-agent RL for learning to drive in MADRaS. The purpose of these experiments is to highlight the features of MADRaS that were discussed in the previous section as an improvement over GymTORCS.

4.1 Experimental Setup

We demonstrate how MADRaS can be used to create a wide variety of driving tasks that can be addressed by RL. Table 4 presents a brief outline of our experiments and their individual motivations. We use the Proximal Policy Optimization (PPO) algorithm (Schulman, Wolski, Dhariwal, Radford, & Klimov, 2017) for RL in all our experiments. PPO is a trust-region based local policy optimization algorithm that has been shown to be very effective in learning policies for continuous control tasks (Andrychowicz, Baker, Chociej, Jozefowicz, McGrew, Pachocki, Petron, Plappert, Powell, Ray, et al., 2020). We save the comparison of different RL algorithms on MADRaS tasks for a future paper in the interest of brevity. All the performance statistics presented in this section are estimated over at least 100 episodes. All experiments with the track-position – speed action space have a `PID_latency` of 5 time steps. The reward functions of the RL agents are defined as weighted sums of reward (r) and penalty (p) components with weights w_r and w_p, respectively:

$$agent\_reward = \sum_{r \in rewards} w_r r - \sum_{p \in penalties} w_p p \tag{2}$$

Some general purpose reward and penalty components that are used in all the experiments are as follows:

Progress Reward: Progress Reward rewards the agent for making a finite progress at every time step. We calculate progress relative to a target speed. We reward the agent proportional to its speed until it reaches the target speed. If the speed goes beyond the target speed, we do not give the agent any extra reward. This way we prevent the agent from maximizing its cumulative rewards by running fast and crashing rather than finishing the race. Let $d(t)$ be the distance (in meters) covered by the agent in the t^{th} time step and s_{target} denote the target speed in meters per step. Progress reward is given by:

$$progress\_reward(t) = \min\left(1, \frac{d(t)}{s_{target}}\right) \tag{3}$$

Average Speed Reward: Average Speed Reward rewards the agent for maintaining a high average speed only if it manages to complete a full lap of the track. Suppose the average speed of the agent for a lap is s_{avg}. Average Speed Reward is calculated as:

$$average\_reward = \frac{s_{avg}}{s_{target}} \tag{4}$$

The Average Speed Reward is also scaled (but not capped) relative to the target speed s_{target} of the agent.

Angular Acceleration Penalty: This penalty is meant to discourage the agent from making frequent unnecessary side-wise movements while running down a track. We calculate a numerical approximation of angular acceleration from the the past 3 recorded values of the *angle* between the car's direction and the direction of the track axis. We scale the penalty with respect to a reference $\alpha_{reference}$. Let a_{t-2}, a_{t-1}, a_t be three consecutive angles of the agent. We calculate Angular Acceleration Penalty as:

$$angular\_accleration\_penalty(t) = \frac{|a_t + a_{t-2} - 2a_{t-1}|}{\alpha_{reference}} \tag{5}$$

We set $\alpha_{reference}$ to 2.0 in all our experiments.

Turn Backward Penalty: A fixed penalty of -1 if the car turns backwards.

Collision Penalty: A fixed penalty of -1 if the car collides with obstacles or other cars and incurs a damage.

Apart from these we also use task specific rewards that we define separately in each experiment.

We terminate an episode if one of the following events happen:

- car turns backwards,
- car goes out of track,
- car collides with an obstacle,
- agent fails to complete its task within the maximum allowable duration of an episode,
- agent successfully completes the task at hand.

Unless otherwise stated, we set the learning rate to 5×10^{-5}. The policy and value functions are modelled using fully connected neural networks with 2 hidden layers and 256 *tanh*–units in each layer. We use the PPO implementation of RLLib (Liang, Liaw, Moritz, Nishihara, Fox, Goldberg, Gonzalez, Jordan, & Stoica, 2018) for all our experiments for its stability and support for multi-agent training. The PID parameters used for track-position – speed control are given in Table 3. Although ideally these parameters must be tuned for each car and for each speed range, we use the same set of parameters (originally tuned for medium-low speeds of `car1-trb1`) everywhere to check if it is possible to teach RL agents to be robust to imperfections in the low level controller.

The remaining part of this section is dedicated to a detailed discussion of our experiments and major observations[4] that can be made from them.

Experiment 1: Generalization across tracks with higher level actions

In our first experiment, we compare two RL agents, one having the high-level track-position – speed (T-S) control mode and the other having the low-level steer – acceleration – brake (S-A-B) control mode, on their ability to generalize across multiple driving tracks in MADRaS. We train the agents to drive `car1-stock1` in the `Alpine-1` track and evaluate them on the

[4] Accompanying video: https://youtu.be/io5mP0HUytY

Table 4: Outline of the experiments presented in this paper.

Exp No.	**Exp Name**	**Motivation**
1	Generalization across tracks with higher level actions	Comparison of primitive and high-level control modes offered by MADRaS in terms of generalization and handling.
2	Generalization across vehicular dynamics through random car selection	Demonstration of how one of the task-randomization modes of MADRaS can be leveraged to train a single agent to drive a wide range of cars with different vehicular dynamics by RL.
3	Curriculum learning for driving in the `Spring` track	Showcasing how the complexity of a driving task in MADRaS can be tuned in well defined steps for designing curricula for learning agents.
4	Learning under partial observability and stochastic outcomes of actions	Learning robust driving policies using the ability of MADRaS to simulate noisy sensor and imprecise control scenarios.
5	Learning to drive in traffic	Example of how MADRaS's library of driving agents with pre-defined behaviors can be used to simulate a variety of real-world scenarios for learning to negotiate complex traffic situations.
6	Learning to navigate safely through a traffic bottleneck by multi-agent cooperation and RL	Demonstration of the inter-vehicular communication architecture and the multi-agent training infrastructure of MADRaS.

Table 5: RL training criteria for Experiments 1-3. Please refer to (Loiacono et al., 2013) for details on the observed variables.

Reward function	**Reward Function Component**	**Weightage**
	Progress Reward	1.0
	Average Speed Reward	1.0
	Collision Penalty	10.0
	Turn Backward Penalty	10.0
	Angular Acceleration Penalty	5.0
Observed variables	angle, track, trackPos, speedX, speedY, speedZ	
Done criteria	One Lap Completed, Time Out, Collision, Turn Backward, Out of Track	

other road tracks. Table 5 lists the observed variables and the components of the reward and done functions. We set the maximum duration of an episode at 15000 time steps and the target speed at 100 km/hour. We evaluate the agents in terms of the average fraction of lap covered in an episode, average speed and successful lap completion rate.

Table 7 presents the results of this experiment. We see that the agent with high-level track-position – speed (T-S) control generalizes significantly better than the one with low-level steer – acceleration – brake (S-A-B) control as given by higher average scores. The low-level S-A-B control mode gives the agent tighter control of the car that can be exploited to perform maneuvers very specific to the training track in order to navigate the twists and turns while maintaining a high average speed (see the accompanying video). This results in the agent overfitting to the training track and it fails to make any significant progress in some of the test tracks. Implementing a desired track-position and speed may require different sequences of low-level actions in different tracks. Relegating the low-level control to a PID controller gives the T-S agent better generalization to track-geometries than the S-A-B agent.

Experiment 2: Generalization across vehicular dynamics through random car selection

In our second experiment, we leverage the ability of MADRaS to change the agent's car at the beginning of each episode to train a driving policy that generalizes to multiple cars with significantly different vehicular dynamics. Table 6 gives some physical parameters of the cars used in this experiment that characterize their handling and dynamics. Heavier cars with a low centre of gravity e.g. `car1-stock1`, `car3-trb1` and `car1-stock2` are more stable and handle better with less body-roll around tight corners. The variation of torque with the RPM (Rotations Per Minute) of a car's engine plays a crucial role in deciding its dynamics. The torque produced by an engine decides how fast the car can accelerate. Torque is usually a strong function of engine RPM. While running at a given RPM, a car can accelerate faster if its engine can produce higher torque at that RPM. Figure 3 gives the torque-RPM curves for the cars used in this experiment. The cars fall in two broad categories in terms of the overall shape of this curve. Cars with a "∪"-shaped curve e.g. `buggy`, `baja-bug` and `155-DTM` have high torque at low (< 1000) and high (> 10000) RPM and significantly lower values in the middle. The other category of cars e.g. `car1-stock1`, `car3-trb1` and `car1-stock2` have a "hat" (∩)-shaped curve with low torque at low and high RPM and high values in the middle. When the agent needs high torque to accelerate from a standstill, speed up or climb uphill, it needs to take the engine RPM to the high-torque zone with a suitable sequence of accelerator inputs. The high-torque zones of the aforementioned categories of cars are roughly opposite to one another. This makes it challenging for a driving agent to generalize to both kinds of cars.

We choose the `Alpine-1` track for this experiment. The `Alpine-1` track is one of the hardest road tracks of MADRaS with sharp left and right turns and a few stretches of slippery road. We set the maximum duration of an episode to 20000 time steps and the target speed to 100 km/hour. We evaluate the agent in terms of average fraction of the lap covered per episode and average speed.

First, we train two PPO agents to drive `car1-stock1` (∩-shaped torque-rpm curve) and `buggy` (∪-shaped torque-rpm curve) using the S-A-B control mode. We evaluate them on five test cars of different dynamic properties. Table 8 presents the results. We see that an agent trained on a car of one torque-RPM category has difficulty generalizing to the cars of the other category. While the `car1-stock1` agent generalizes to `car3-trb1`, `kc-2000gt` and `car1-stock2` with ∩-shaped torque-rpm curves, it fails to drive `155-DTM` and `baja-bug` that have ∪-shaped torque-rpm curves. The `buggy` agent on the other hand generalizes fairly to `155-DTM` and `baja-bug` but fails to drive the other three test cars due to mismatch in dynamic properties. With a view to aiding in generalization through domain randomization, we leverage the ability of MADRaS to randomly switch cars between episodes and present `car1-stock1` and `buggy` to the same agent with equal probability. We observe that this training strategy brings remarkable generalization across both categories of test vehicles with significant improvement both in terms of average fraction of lap covered in an episode and average speed.

Experiment 3: Curriculum learning for driving in the `Spring` track

In our third experiment, we present a study to demonstrate how the ability of MADRaS to control the complexity of a driving task in well defined steps can be used to design curricula for an RL agent to accomplish complex tasks in a sample efficient way. We attempt to train a PPO agent to drive `car1-stock1` on `Spring` track using the primitive S-A-B action space. With a length of 22.1 km, `Spring` is the longest track in TORCS. It has the largest number of turns with different grades of sharpness, both in the left and right directions. It also has ramps and declines. The surface texture varies from place to place. These make it the toughest road track to drive in TORCS. We set the target speed to 100 Km/hr and maximum episode length to 40000 steps. Figure 4 and Table 9 show the results of this study. We see that training from scratch on `Spring` fails to complete one lap of the track

Table 6: Some physical properties of the cars used in Experiment 2 that play an important role in determining their vehicular dynamics. “RWD” and “4WD” stand for “Rear Wheel Drive” and “Four Wheel Drive”, respectively.

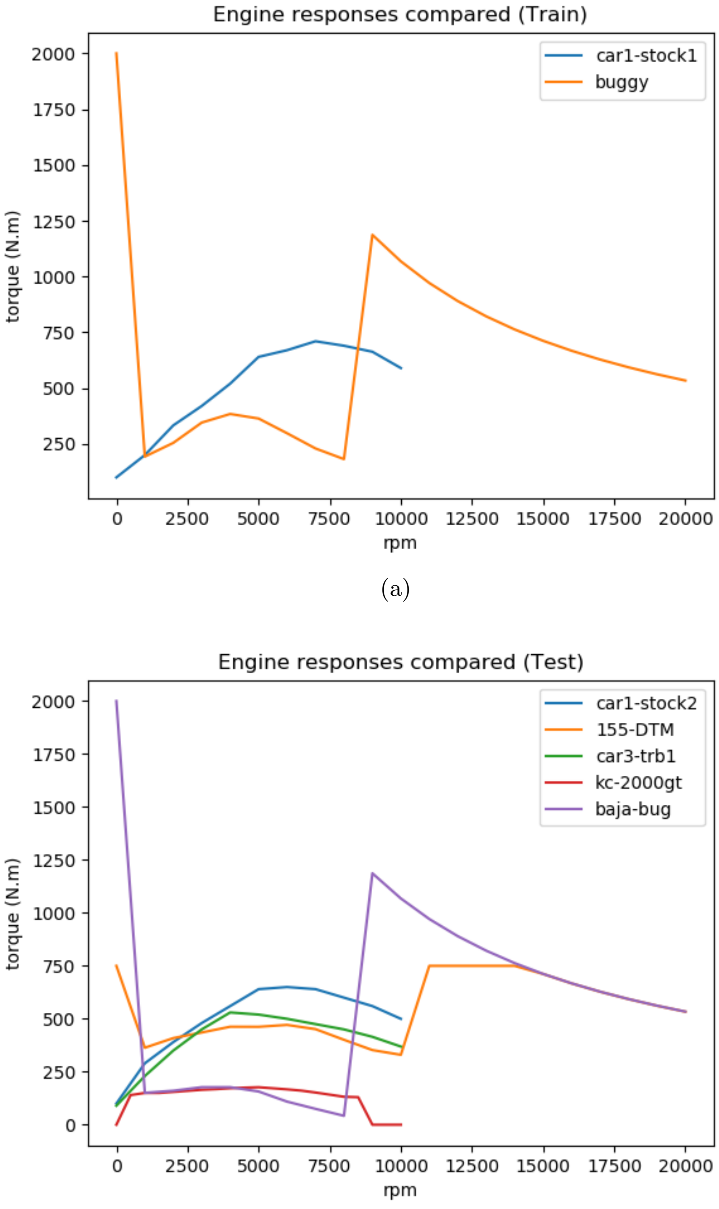
Car Name	Drive Type	Mass (Kg)	Height of CG (m)
car1-stock1	RWD	1550.0	0.3
car1-stock2	RWD	1550.0	0.3
155-DTM	4WD	1100.0	0.2
car3-trb1	RWD	1150.0	0.2
kc-2000gt	RWD	1200.0	0.25
buggy	RWD	650.0	0.45
baja-bug	RWD	600.0	0.35

Table 7: Generalization of an agent trained on `Alpine-1` to other road tracks (Experiment 1). S-A-B (Steering - Acceleration - Brake) and T-S (Track position - Speed) denote the control mode used.

		Avg. fraction of lap covered		**Avg. Speed**		**Lap completion rate**	
		S-A-B	**T-S**	**S-A-B**	**T-S**	**S-A-B**	**T-S**
Training Track	alpine-1	0.75	0.73	91.89	83.32	0.68	0.58
Test Tracks	aalborg	0.001	0.11	0.10	59.39	0.0	0.0
	alpine-2	0.38	0.31	89.95	72.64	0.04	0.0
	brondehach	0.001	0.72	0.1	81.01	0.0	0.3
	g-track-1	0.001	0.98	0.06	79.42	0.0	0.91
	g-track-2	0.002	0.97	0.11	75.99	0.0	0.95
	g-track-3	0.001	0.84	0.09	79.90	0.0	0.44
	corkscrew	0.0008	0.64	0.06	81.39	0.0	0.0
	e-road	0.001	0.94	0.11	85.63	0.0	0.88
	e-track-2	0.07	0.39	8.38	75.21	0.0	0.0
	e-track-3	0.31	0.68	25.88	77.96	0.03	0.57
	e-track-4	0.0005	0.95	0.08	78.41	0.0	0.85
	e-track-6	0.0009	0.83	0.09	80.65	0.0	0.58
	forza	0.001	0.79	0.08	71.63	0.0	0.70
	ole-road-1	0.29	0.40	101.22	78.06	0.0	0.11
	ruudskogen	0.97	0.97	100.87	81.15	0.95	0.93
	street-1	0.03	0.87	1.76	74.67	0.0	0.67
	wheel-1	0.0009	0.95	0.09	78.08	0.0	0.76
	wheel-2	0.36	0.81	81.69	81.51	0.0	0.64
	spring	0.14	0.29	104.76	82.55	0.0	0.0
	Average Scores (Test)	**0.14**	**0.71**	**27.12**	**77.64**	**0.04**	**0.49**

Table 8: Generalization of PPO policies using the S-A-B control mode across vehicles with different dynamics (Experiment 2). “random” refers to the setting in which the agent is presented with both `car1-stock1` and `buggy`, each with a probability of 0.5 during training.

		Avg. Fraction of Lap Covered			**Avg. Speed (km/h)**		
	Training Car	**car1-stock1**	**buggy**	**random**	**car1-stock1**	**buggy**	**random**
Test Cars	**155-DTM**	0.37	0.05	0.37	104.22	22.71	99.78
	car3-trb1	0.002	0.017	0.62	0.12	0.97	58.95
	kc-2000gt	0.77	0.013	0.30	80.44	0.71	22.02
	car1-stock2	0.001	0.016	0.54	0.09	0.91	50.23
	baja-bug	0.35	0.92	0.55	59.45	61.45	54.91
Average Scores		0.30	0.20	**0.48**	48.86	17.35	**57.18**



(a)

(b)

Figure 3: Variation of torque with engine RPM of cars studied in Experiment 2. (a) Torque-vs-RPM of the cars that we present our agent to drive during training with equal probability. (b) Torque-vs-RPM of the cars that we test our agent on.

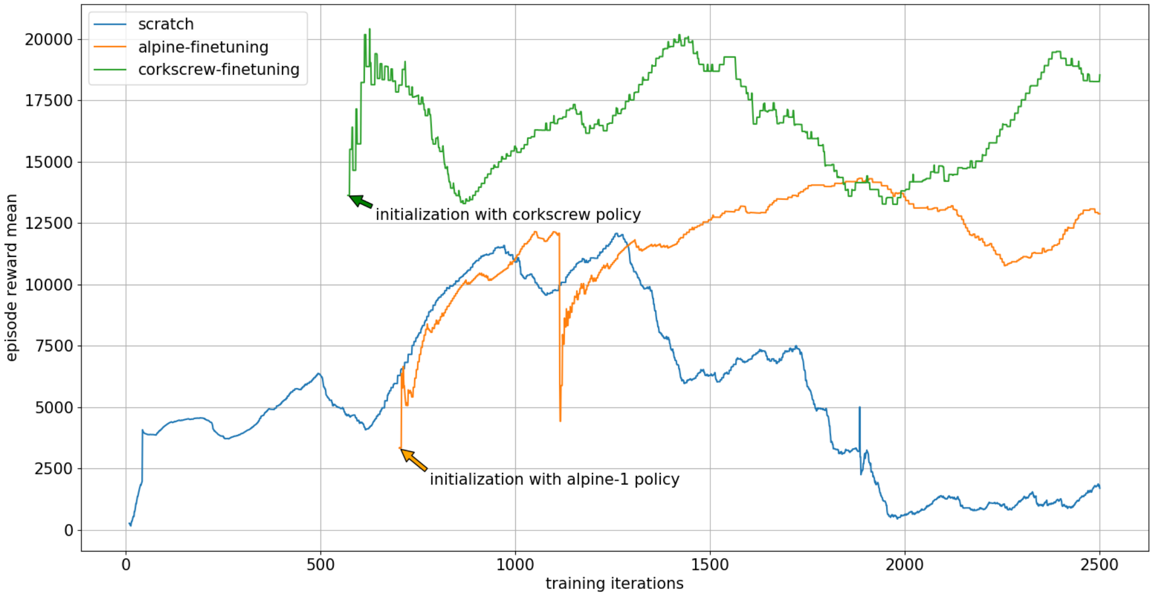


Figure 4: Variation of episode reward over iterations of PPO for learning from scratch on `spring` compared with first learning on simpler tracks – `alpine-1` and `corkscrew` – and then fine-tuning on `spring` (Experiment 3).

Table 9: Curriculum learning results for driving in `Spring` track (Experiment 3).

	Fraction of lap covered	**Average Speed (km/hr)**	**Lap completion rate (%)**
Training from scratch	0.18	101.9	0.0
Pre-training in Alpine-1	0.57	103.5	27.0
Pre-training in Corkscrew	0.54	100.6	45.8

even after 2500 iterations. When we use a curriculum of first training on `Alpine-1` or `Corkscrew` tracks followed by fine-tuning on `Spring` the agent learns to complete the entire lap with high success rates and average speed. In our curriculum learning experiments, we pick the policy that gives the highest mean trajectory reward in the first phase of training (obtained after 701 iterations in `Alpine-1` and 561 iterations in `Corkscrew`) and use it to initialize the policy in the second phase. The total number of training iterations and the total number of training samples for the curriculum learning strategies (considering both the pre-training and fine-tuning stages) are kept equal to that of training from scratch for fairness of comparison. For fine-tuning, we choose a learning rate of 1×10^{-6} for the `Alpine-1` policy and 5×10^{-7} for the `Corkscrew` policy. We evaluate the agents in terms of the average fraction of lap covered in an episode, average speed and successful lap completion rate.

Table 10: Results of a single PPO agent learning to drive in traffic by RL. The agent was trained to drive in the presence of 4 or 5 traffic cars with equal probability (Experiment 5).

Number of traffic agents	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Successful task completion rate	99.5%	98.1%	95.5%	96%	95.5%	95.7%	92.8%

Experiment 4: Learning under partial observability and stochastic outcomes of actions

In this experiment we compare the performances of PPO agents trained to drive `car1-stock1` around the `Corkscrew` track with and without observation noise under different levels of stochasticity of the outcome of actions. The training was performed with the primitive S-A-B action space. Observed variables, episode termination criteria and evaluation metrics are the same as in Experiment 1. The reward function is the same as in the Experiments 1-3 (see Table 5) with the weightage for angular acceleration penalty increased to 8. As described in Section 3, stochastic outcomes of actions is implemented by adding zero mean Gaussian noise to the actions. Figure 5 shows the learning curves. All these agents are tested in the same track `Corkscrew` in the presence of both observation noise and 0.5 standard deviation action noise. Table 11 compares the performance statistics. We observe that the agents trained in the presence of both observation and action noise perform better than the others. This demonstrates the ability of MADRaS to serve as a platform for evaluating the resilience of learning agents to observation noise and environmental stochasticity.

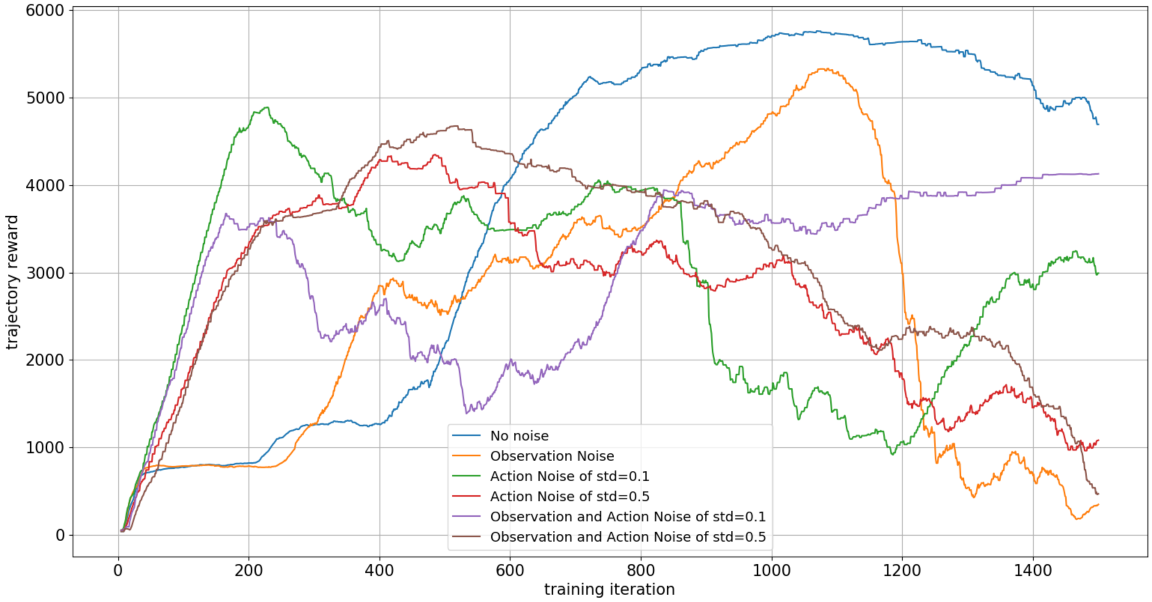


Figure 5: Learning to drive with under partial observability and stochastic outcomes of actions in `Corkscrew` track (Experiment 4).

Table 11: Learning to drive in the `corkscrew` track with and without observation noise and different levels of stochasticity in the outcome of actions and evaluation with observation noise and 0.5 std action noise (Experiment 4).

	Avg. Fraction of Lap Covered	**Avg. Speed (km/hr)**
No noise	0.38	52.54
Observation noise	0.19	30.78
Stochastic actions (noise std 0.1)	0.12	29.99
Stochastic actions (noise std 0.5)	0.64	48.67
Observation noise and Stochastic actions (noise std 0.1)	0.63	48.85
Observation noise and Stochastic actions (noise std 0.5)	0.68	46.91

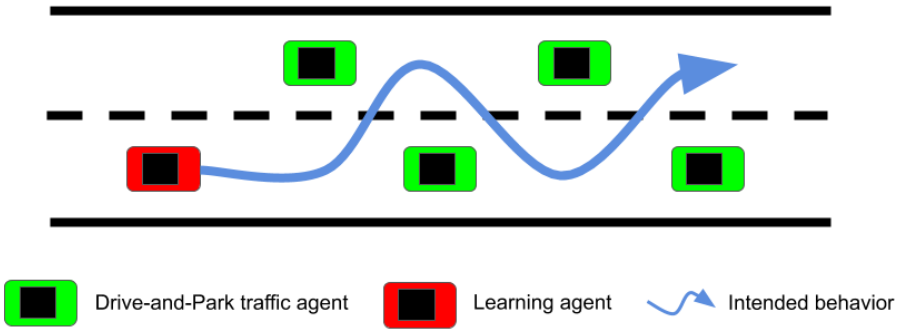


Figure 6: Schematic diagram of the environment design for Experiment 5. The task of the learning agent is to overtake all the traffic cars without colliding with any of them or going off track.

Experiment 5: Learning to drive in traffic

In this experiment we use the ability of MADRaS to generate custom traffic to train an agent to navigate through a narrow road without colliding with any traffic car – moving or parked. Figure 6 shows a schematic diagram of the training environment. We choose the `Aalborg` track for this study since it is one of the narrowest tracks of TORCS and further reduce its width to half resulting in an effective track width of 5m.

The traffic agents used in this experiment are `DriveAndParkAgents` (see Table 2). MADRaS positions the traffic cars ahead of the learning car at the start of the race. When an episode begins, the `DriveAndParkAgents` start driving at their given target speeds (50 km/hr) towards their given parking locations (specified in terms of distance from the start of the race and track position) using PID controllers. This way, the learning agent sees moving cars in the beginning and parked cars towards the end of each episode. This forces it to learn to avoid collision with both static and moving obstacles. We set the parking

Table 12: RL training criteria for Experiment 5. Please refer to (Loiacono et al., 2013) for details on the observed variables.

	Reward Function Component	**Weightage**
Reward function	Progress Reward	1.0
	Average Speed Reward	1.0
	Collision Penalty	10.0
	Turn Backward Penalty	10.0
	Angular Acceleration Penalty	1.0
	Overtake Reward	5.0
	Rank 1 Reward	100.0
Observed variables	angle, track, trackPos, speedX, speedY, speedZ, opponents	
Done criteria	Rank 1, Time Out, Collision, Turn Backward, Out of Track	

locations of the traffic cars on alternate sides of the road so that the the agent must learn to turn both left and right to overtake all the traffic cars. We maintain a gap of at least 10m between consecutive parking locations along the length of the road to make sure that the learning car has enough space to maneuver between the traffic cars. To create variance in the environment, we randomly vary each parking location within an area of 5m along the track length and 0.25m along the track width. We also switch the number of traffic cars between 4 and 5 with equal probability. Changing the number of traffic cars also makes sure that the learning agent gets initialized in the left and right halves of the track with equal probability. We use the T-S control mode and set the target speed of the learning agent to 50 km/h. Table 12 gives the training criteria for this experiment.

The agent gets an *Overtake Reward* every time it overtakes a traffic agent and *Rank 1 Reward* at the end of the episode if it manages to overtake all the traffic agents. The agent is evaluated in terms of the fraction of times it overtakes all the traffic cars successfully.

Table 10 presents the results of this experiment. We observe that the agent learns to generalize to both fewer and more traffic agents than it encountered during training and navigate its way through them collision-free with a high success rate. Figure 12 in the Appendix shows how the instances of the agent colliding and driving off-track reduces as training progresses while the frequency of it emerging Rank 1 and successfully completing the episode increases.

Experiment 6: Learning to navigate safely through a traffic bottleneck by multi-agent cooperation and RL

One of the biggest aspirations of autonomous driving is the avoidance of traffic congestion through cooperation. In this experiment we utilize the multi-agent training infrastructure of MADRaS and its framework for inter-vehicular communication to solve a simplified ver-

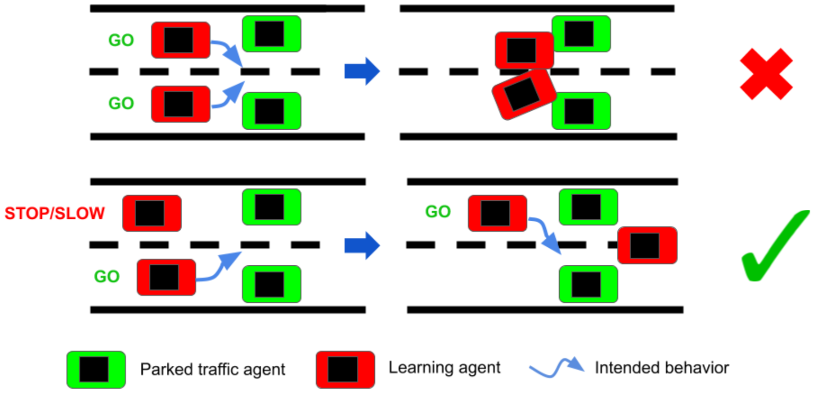


Figure 7: Schematic diagram of the multi-agent task studied in Experiment 6. The task for the two learning agents is to coordinate with each other and pass through the gap between the parked traffic cars without making any collision. The top row shows an example of undesirable behavior in which both the agents attempt to pass through the bottleneck at the same time and result in a collision. The bottom row gives a feasible solution to the problem in which one of the agents stops or slows down to wait for the other agent to pass through the gap. Only after the gap is clear does it attempt to pass through – thus avoiding collision with any of the other cars.

Table 13: Dimensions of cars used in Experiment 6.

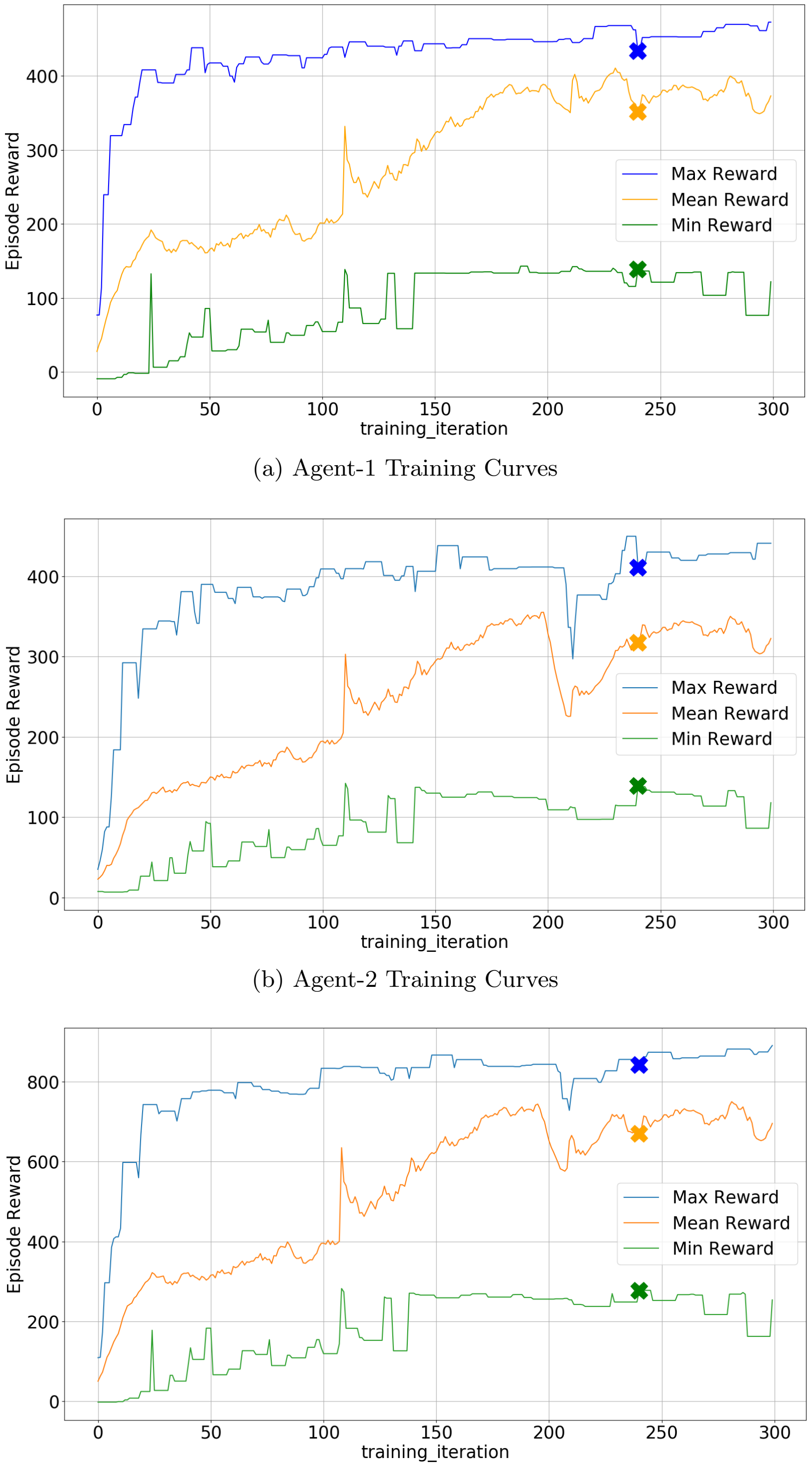
	Car Model	**Length (m)**	**Width (m)**
Traffic Car	`car1-trb1`	4.52	1.94
PPO Agent-1	`car3-trb1`	4.55	1.95
PPO Agent-2	`car5-trb1`	4.67	1.94

sion of this task by multi-agent reinforcement learning.

The training environment consists of two PPO agents and two traffic agents on the `Corkscrew` track. The PPO agents communicate their actions to each other at every step. We park the traffic agents next to each other with a small gap in between that is sufficient only for one car to pass through. The task of the PPO agents is to pass through the gap one by one without colliding with each other or with any traffic agent (see Figure 7). Thus the agents must learn a collaborative strategy in which the agent trying to pass through

Table 14: Curriculum for multi-agent RL in Experiment 6.

Iterations of training	**Parking Distance (m)**	**Gap Width (m)**
1–240	30–40	2.76–4.06
240–300	30–35	2.76–3.46



(a) Agent-1 Training Curves

(b) Agent-2 Training Curves

(c) Joint learning Curves

Figure 8: Learning curves for multi-agent training in Experiment 6. The cross symbol denotes transition point in the agent's curriculum where the first task ends and the second task begins.

Table 15: RL training criteria for Experiment 6. *peerActions* refers to the actions of the other learning agent from the previous time step. Please refer to (Loiacono et al., 2013) for details on the other observed variables.

	Reward Function Component	**Weightage**
Reward function	Progress Reward	1.0
	Average Speed Reward	1.0
	Collision Penalty	10.0
	Turn Backward Penalty	10.0
	Angular Acceleration Penalty	5.0
Observed variables	angle, track, trackPos, opponents, speedX, speedY, speedZ, peerActions	
Done criteria	Race Over, Time Out, Collision, Turn Backward, Out of Track	

the gap first should be given enough time to pass through completely by the other agent before it makes its own attempt.

Table 13 gives the cars assigned to the learning and traffic agents and their dimensions. Both the PPO agents have T-S action space. Table 14 describes the curriculum used for the training. We randomly vary the parking distance of each traffic car and the gap between them at the start of each episode for improved generalization. Table 15 gives the details of the observed variables, reward functions and done criteria. The agents must learn the following distinct skills to be able to accomplish this task.

- Running forward without going off track.
- Not colliding with each other.
- Not colliding with any of the parked cars.
- Learning to collaborate and pass through the bottleneck one by one.

We jointly evaluate the agents in terms of the rate of successful passage of both the agents through the traffic bottleneck. Figure 8 shows the individual and joint learning curves respectively during training. The final evaluation is done over 100 episodes of stochastically parked agents and the PPO agents demonstrate a joint task completion rate of 83.3%.

5. Conclusion

In this paper we present MADRaS, an open-source Multi-Agent Driving Simulator for autonomous driving. MADRaS builds on TORCS, a popular car racing platform, and adds a suite of features like hierarchical control modes, domain randomization, custom traffic, partial observability, stochastic outcomes of actions and support for multi-agent training. We present a suite of experiments that illustrate how MADRaS can be used to simulate rich highway and track driving scenarios of high variance and complexity that are valuable for autonomous driving research and investigating the robustness and generalization abilities of RL algorithms. We compare primitive and abstract (or, high-level) control-modes at the task of generalizing to a multitude of driving tracks and observe that the abstract control-mode achieves superior generalization while the primitive control-mode offers tighter handling. We learn a policy that generalizes to a wide range of vehicular dynamics simply by training on two car models from the extreme ends of the spectrum and leveraging MADRaS's ability to change the agent's car in every episode. We use the ability of MADRaS to inject varying levels of noise into the observation and action spaces to study driving under stochasticity and partial observability. MADRaS offers a powerful set of tools for simulating traffic. We present experiments on learning to navigate through static and moving traffic without colliding or going off track and learning multi-agent cooperation for passing through traffic bottlenecks safely. We wish to develop features specific to fuel management and vehicular safety in the future.

Acknowledgements

The authors would like to thank Professor Pabitra Mitra of the Department of Computer Science and Engineering, IIT Kharagpur for his helpful feedback on the structure of the paper and Manish Prajapat of ETH Zurich for his useful tips on the implementation of inter-vehicular communication in MADRaS. The authors would also like to thank Intel Labs India for incubating the early stage of this project. Anirban Santara's work in this project was supported by Google India, under the Google India PhD Fellowship grant, and Intel Inc. under the Intel Student Ambassador Program. Anirban Santara and Sohan Rudra contributed equally to this project.

Appendix A. Configuring MADRaS

The structure of MADRaS focuses on the ease of use and encourages custom modifications. In this section we describe the configuration variables of MADRaS. All these variables are specified in the `envs/data/madras_config.yml` file. The 'yaml' (or 'yml') format provides a powerful yet convenient way of specifying most data types and basic data structures like lists and dictionaries.

The `madras_config.yml` file has three sections:

1. **Server configuration:** In this section contains the global configurations of the MADRaS environment. Since MADRaS can randomly vary the driving tracks, model of car for the learning agents, and the number of traffic cars between episodes, these terms are specified as lists and ranges. The maximum number of cars in the environment (including learning and traffic agents) can be specified as `max_cars` and the minimum number of traffic cars by `min_traffic_cars`. The number of learning agents (N_l) is specified in the "agent configuration" section.

   $$N_l + \texttt{min\_traffic\_cars} \leq \texttt{max\_cars}$$

   The list of car models to choose for the learning agent can be specified in `learning_car`. The list of tracks to choose for each episode can be specified in `track_names`. If `randomize_env = True` the car model, track and the number of traffic agents is chosen randomly for each episode.

2. **Agent configuration:** The `agents` section, contains the configurations of the learning agents. The `target_speed`, `pid_settings` for the low level controller if `pid_assist` is `True`, configuration of the observation space (according to the modes in `utils/observation_handler.py`), reward function (to be parsed by `utils/reward_handler.py`) and done function (to be parsed by `utils/done_handler.py`) can be specified individually for each agent in this section.

3. **Traffic configuration:** The `traffic` section can be used to specify the details of the traffic agents in the environment. If N_t traffic agents need to be chosen in a given episode, their configurations will be set to the first N_t elements from the list of agents in this section. These configurations are parsed by `traffic/traffic.py`. The `target_speed`, `target_lane_pos`, collision avoidance properties and `pid_settings` of the traffic cars can be specified here. If the traffic agents need to be parked in certain locations (specified in terms of their distance from the start line and track position) of the track before the start of each episode, that can also be specified in this section.

The full list of the configuration variables is available in Tables A1, A2 and A3.

MADRaS supports inter-vehicular communication (IV-Comm) between the learning agents. The settings for the IV-Comm system can be specified in `envs/data/communications.yml`. The user can specify the list of variables (`vars`) that each learning agent wants to observe from a list of communicating agents (`comms`) for a given number of previous time steps (`buff_size`).

Table A1: Server Configuration Parameters

Parameters	**Description**	**Possible Values**
`torcs_server_port`	For setting the port of communication with the TORCS Server.	$\mathbb{Z}^+$
`max_cars`	Max number of vehicles to be spawned.	$\mathbb{Z}^+$
`min_traffic_cars`	Min number of traffic cars to be spawned.	$\mathbb{Z}^+$
`track_names`	List of tracks on which the simulation will run.	List of track names
`track_limits`	Restrict the agent to remain within a given range of `track_pos` values.	$(\mathbb{R}, \mathbb{R})$
`distance_to_start`	Starting distance of the cars from the start line.	$\mathbb{Z}^+$
`torcs_server_config_dir`	The location of the TORCS server racing config directory.	Path string
`scr_server_config_dir`	The location of available cars config directory	Path string
`traffic_car`	The type of car to be used for traffic	car name
`learning_car`	List of car models for using as the learning agent.	List of car names
`randomize_env`	Flag for turning randomization on.	boolean
`add_noise_to_actions`	Flag for adding a small Gaussian Noise to the actions before sending to the TORCS server.	boolean
`action_noise_std`	Specifies the standard deviation of the Gaussian for the noise addition.	$[0, 1]$
`noisy_observations`	Toggles the TORCS flag for enabling noisy observations.	boolean
`visualise`	Flag for setting the display on and off.	boolean
`no_of_visualisations`	To visualize multiple training instances	$\mathbb{Z}^+$
`max_steps`	Maximum steps that the environment will take before resetting.	$\mathbb{Z}^+$

Appendix B. PID Response

In this section we describe our implementation of the PID controller used for low level control in our experiments with the track-position – speed control mode of MADRaS. Please note that this implementation can be easily swapped out for a more sophisticated one by creating a derived class of `PIDController` defined in `controllers/pid.py`. The error function (e_{TP}) for track-position PID controller is defined as a function of the track-position (TP) and the angle (θ) that the car's heading makes with the center line. The output of this controller is the steer-angle of the vehicle for the current time-step (t) that would bring the car closer to the desired track-position ($TP_{desired}$).

$$e_{TP}(t) = \theta(t-1) - (TP(t-1) - TP_{desired}) * scale \tag{6}$$

Table A2: Agent Configuration Parameters

Parameters	**Description**	**Possible Values**
`vision`	Flag for activating visual input instead of the usual sensor based one.	boolean
`throttle`	Flag for activating throttle control on and off.	boolean
`gear_change`	Flag for activating gear control on and off.	boolean
`client_max_steps`	Maximum steps that the client is available to take.	$\mathbb{Z}^+ \cup \{-1\}$
`target_speed`	Target speed setting of the agent car.	$\mathbb{Z}^+$
`state_dim`	Dimension of the Observation Space.	$\mathbb{Z}^+$
`normalize_actions`	Toggle to turn on action normalization.	boolean
`pid_assist`	Toggle to turn on T-S control mode.	boolean
`pid_settings[accel_pid]`	K_p, K_i, K_d for throttle PID.	List of floats
`pid_settings[accel_pid]`	K_p, K_i, K_d for steering PID.	List of floats
`accel_scale`	Acceleration Scaling.	$\mathbb{R}^+$
`steer_scale`	Steering Scaling.	$\mathbb{R}^+$
`pid_latency`	Number time-steps the control command sticks to the server.	$\mathbb{Z}^+$
`observations[mode]`	Name of the Observation Class.	string
`observations[multi_flag]` *(multi-agent mode only)*	Toggle for turning on communication for the agent i,	boolean
`observations[buff_size]`	Specifies the buffer size of action.	$\mathbb{Z}^+$
`observation[normalize]`	Toggle to tun on observation normalization.	boolean
`obs_min`	Minimum values for certain observation attributes.	dict
`obs_max`	Maximum values for certain observation attributes.	dict
`rewards[name, scale]`	List of the Reward classes and a scaling factor of the rewards.	list of names and dict
`dones`	Done conditions currently in use.	list of dones

Table A3: Common Traffic Configuration Parameters

Parameters	**Description**	**Possible Values**
`name`	Traffic Agent Type,	string
`target_speed`	Traffic Agent Speed.	$\mathbb{R}^+$
`initial_distance`	Traffic Agent initial distance from start line (range).	2-Tuple of Floats
`initial_trackpos`	Traffic Agent initial track-position (range).	2-Tuple of Floats
`track_len`	Length of the Current Track.	$\mathbb{R}^+$
`pid_settings[accel_pid]`	K_p, K_i, K_d values for acceleration.	List of Floats
`pid_settings[steer_pid]`	K_p, K_i, K_d values for steering.	List of Floats
`accel_scale`	Acceleration scaling.	$\mathbb{R}^+$
`steer_scale`	Steering scaling.	$\mathbb{R}^+$
`collision_time_window`	Describes the collision region for the traffic agent	$\mathbb{R}^+$

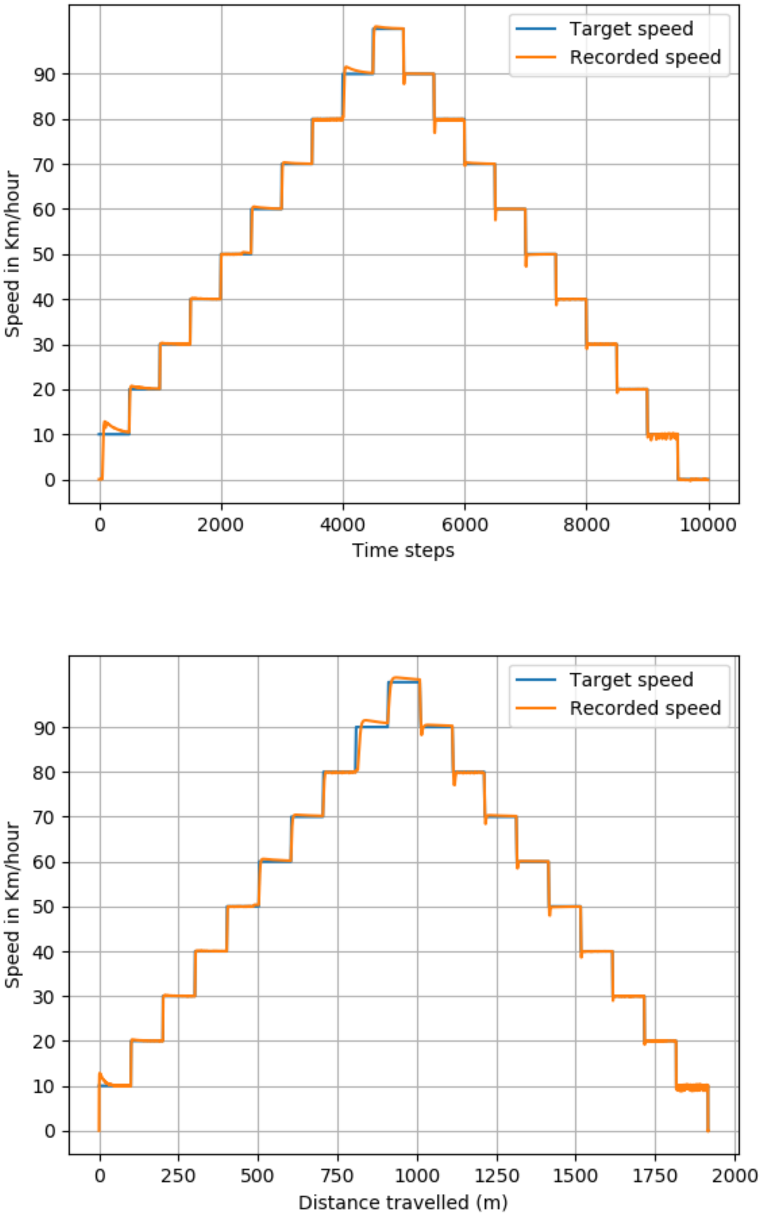
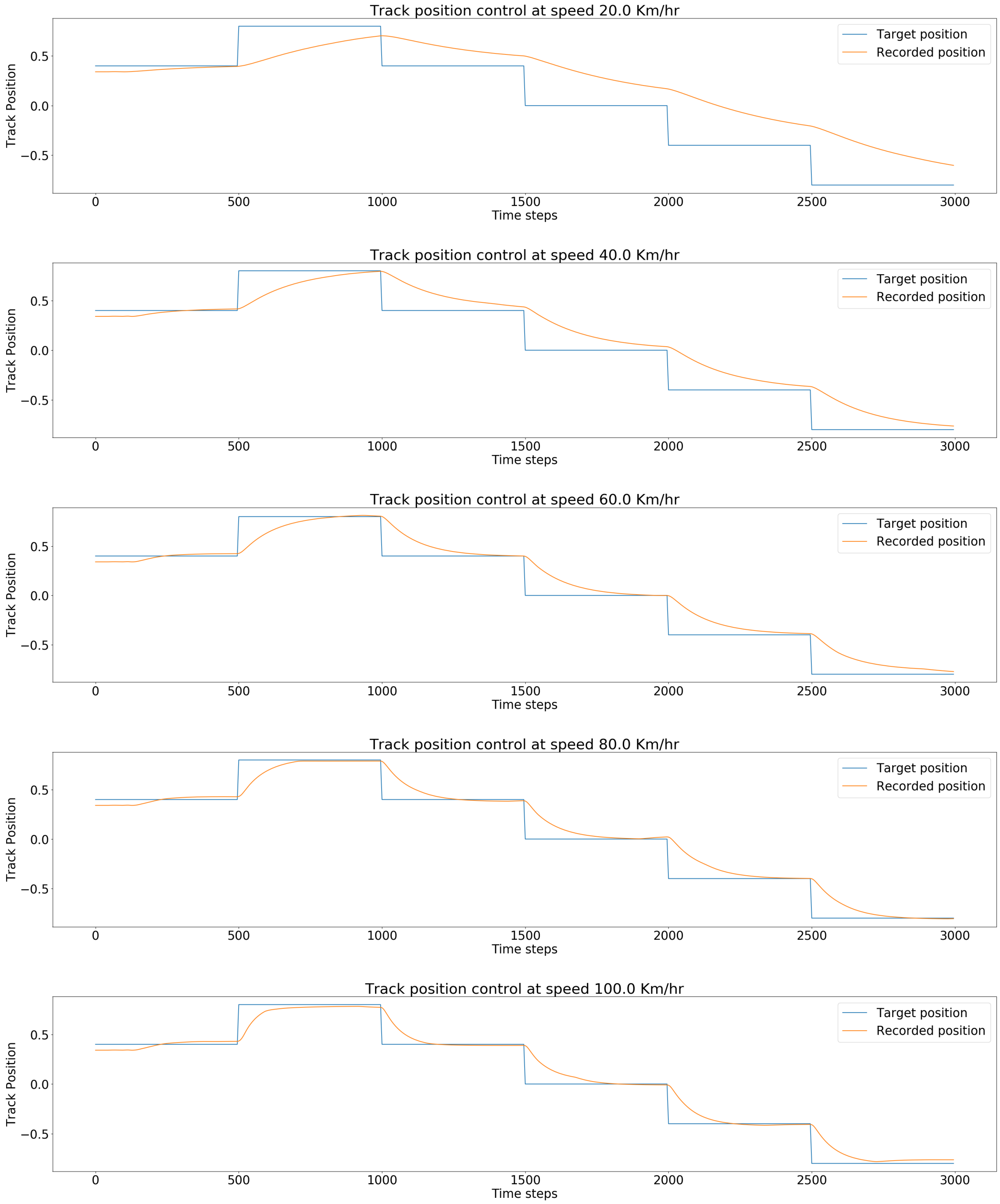


Figure 9: Speed control accuracy and convergence of our PID controller at different initial speeds over time-steps and distance travelled. The `PID_latency` is set to 5. The track used in this study is the "f-speedway" oval track. The lane-position command is fixed at 0.0 which refers to the center of the track.



Track position control at speed 20.0 Km/hr

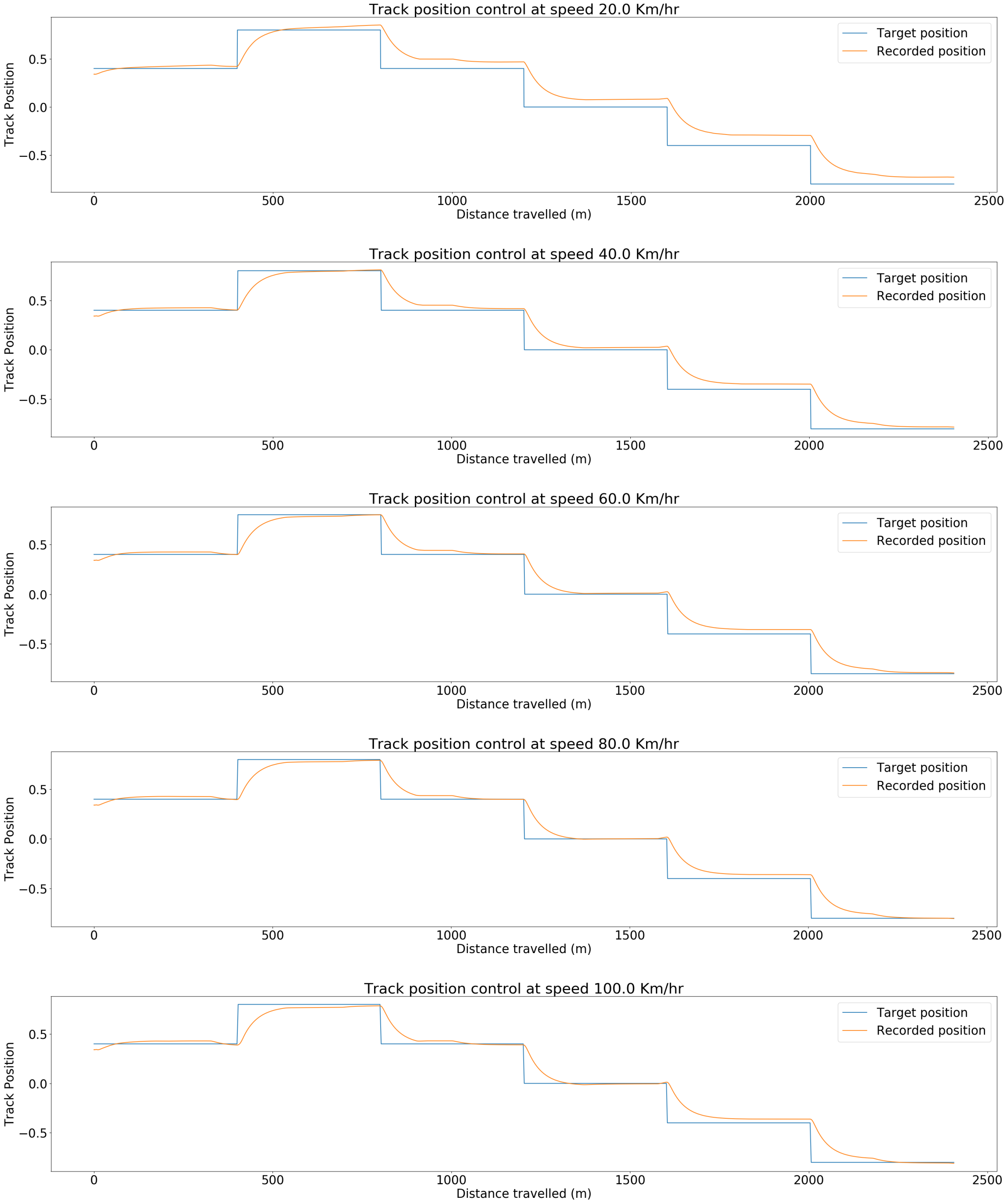
Track position control at speed 40.0 Km/hr

Track position control at speed 60.0 Km/hr

Track position control at speed 80.0 Km/hr

Track position control at speed 100.0 Km/hr

Figure 10: This plot demonstrates the position control accuracy and convergence over *time* at different speeds of the PID controller used in our experiments. The track used is the “f-speedway” oval track and `PID_latency` is set to 5. The plot covers the range of speeds used in our experiments.



Track position control at speed 20.0 Km/hr

Track position control at speed 40.0 Km/hr

Track position control at speed 60.0 Km/hr

Track position control at speed 80.0 Km/hr

Track position control at speed 100.0 Km/hr

Figure 11: This plot demonstrates the position control accuracy and convergence over *distance* at different speeds of the PID controller used in our experiments. The track used is the "f-speedway" oval track and `PID_latency` is set to 5. The plot covers the range of speeds used in our experiments.

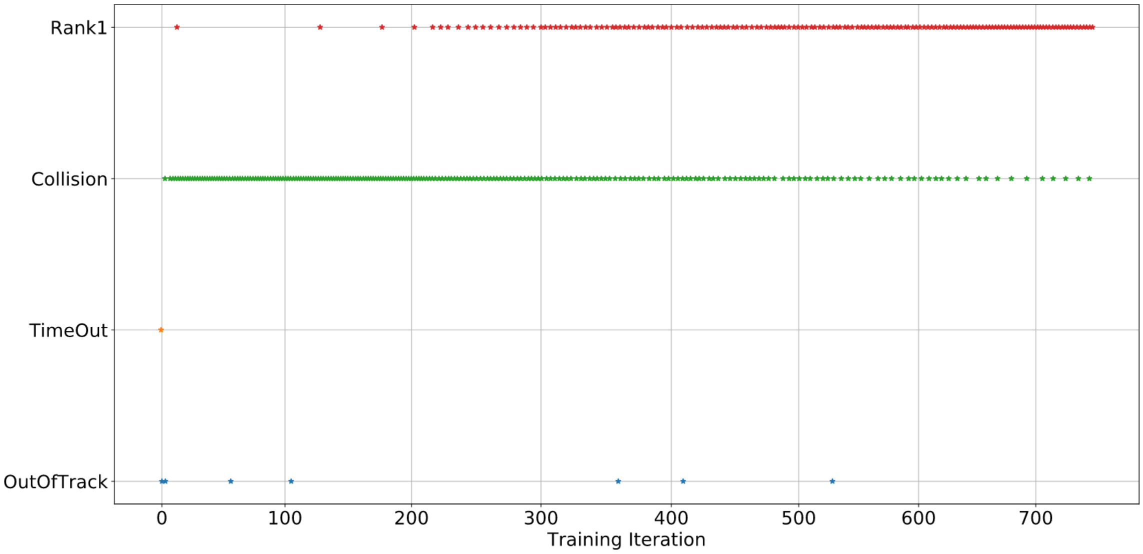


Figure 12: This plot demonstrates how the causes of episode termination (done reason) varies as the agent makes progress in training. We observe that collisions and out-of-track frequencies drop while Rank 1 frequency increases as training progresses. This experiment was a replica of Experiment 5 with 2 or 3 traffic cars at equal probability.

The error function for the Speed PID controller (e_V) is a function of the forward velocity (V). The output of the controller is the value of acceleration and braking that would bring the speed closer to the target speed of the vehicle (V_{target}).

$$e_V(t) = (V(t-1) - V_{target}) * scale \tag{7}$$

Figure 9, 10 and 11 show the responses of the PID controller used in our experiments with the high level track-position – speed action space. For testing the controller response over time, we change the input signal (track-position or speed) every 500 steps and monitor the output. For testing the controller response over distance, we change the input signal after the agent has driven every 100 meters for speed control and 400 meters for track-position control. We use the “f-speedway” oval track for this study. For speed control (Figure 9) we change the target signal in incremental steps of 10 km/hour from 0 km/hour to 100 km/hour and back to 0 km/hour keeping the track-position input fixed at 0.0, the center of the track. For position control we increment the signal in steps of 0.4 starting from 0.4 (default initial track-position) towards the extreme left (up to 0.8) and then towards the extreme right (up to −0.8). We observe that the controller responds faithfully within the range of speeds and track positions used in our experiments.

Appendix C. Initial State Distribution

The initial state of an episode in MADRaS can be configured in terms of the set of parameters listed below. The `madras_config.yml` file has the `randomize_env` flag that can be enabled to randomly assign values for these parameters at the start of each episode.

- **Vehicle Model:** The model of the car assigned to the learning agent(s) can be specified using the learning_car field. This can also be randomly selected from a categorical distribution over a list of car models when randomize_env = True.

- **Number of Traffic Cars:** The number of traffic cars can be specified using the min_traffic_cars field. When randomize_env = True the number of traffic cars is assigned randomly between min_traffic_cars and (max_cars - (number of learning agents)).

- **Track Position of Traffic Cars:** Some traffic cars can be assigned a certain track position to stick to. For ParkedAgent, it can be specified as the parking_lane_pos while for ConstVelTrafficAgent, SinusoidalSpeedAgent and RandomStoppingAgent it can be specified using the target_lane_pos field. If randomize_env = True the track position is sampled randomly from a continuous uniform distribution between specified high and low limits for these parameters.

- **Parking Distance of Traffic Agents from the Start line:** The distance from start of ParkedAgent traffic agents can be set using the parking_dist_from_start parameter. When randomize_env = True it is sampled uniformly from a fixed range specified by high and low values for the same parameter.

References

Andrychowicz, O. M., Baker, B., Chociej, M., Jozefowicz, R., McGrew, B., Pachocki, J., Petron, A., Plappert, M., Powell, G., Ray, A., et al. (2020). Learning dexterous in-hand manipulation. *The International Journal of Robotics Research*, *39*(1), 3–20.

Argall, B. D., Chernova, S., Veloso, M., & Browning, B. (2009). A survey of robot learning from demonstration. *Robotics and autonomous systems*, *57*(5), 469–483.

Bansal, M., Krizhevsky, A., & Ogale, A. (2018). Chauffeurnet: Learning to drive by imitating the best and synthesizing the worst..

Behere, S., & Törngren, M. (2016). A functional reference architecture for autonomous driving. *Information and Software Technology*, *73*, 136–150.

Bengio, Y., Louradour, J., Collobert, R., & Weston, J. (2009). Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pp. 41–48. ACM.

Bojarski, M., Yeres, P., Choromanska, A., Choromanski, K., Firner, B., Jackel, L., & Muller, U. (2017). Explaining how a deep neural network trained with end-to-end learning steers a car..

Bowling, M., & Veloso, M. (2000). An analysis of stochastic game theory for multiagent reinforcement learning. Tech. rep., Carnegie-Mellon Univ Pittsburgh Pa School of Computer Science.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). Openai gym..

Brown, A., et al. (2018). Udacity self-driving car simulator. In *GitHub Repository* *https://github.com/udacity/self-driving-car-sim*.

Bu, L., Babu, R., De Schutter, B., et al. (2008). A comprehensive survey of multiagent reinforcement learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, *38*(2), 156–172.

Chen, C., Seff, A., Kornhauser, A., & Xiao, J. (2015). Deepdriving: Learning affordance for direct perception in autonomous driving. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 2722–2730. IEEE.

Da Silva, F. L., & Costa, A. H. R. (2019). A survey on transfer learning for multiagent reinforcement learning systems. *Journal of Artificial Intelligence Research*, *64*, 645–703.

de Bruin, T., Kober, J., Tuyls, K., & Babuška, R. (2018). Integrating state representation learning into deep reinforcement learning. *IEEE Robotics and Automation Letters*, *3*(3), 1394–1401.

Dikmen, M., & Burns, C. M. (2016). Autonomous driving in the real world: Experiences with tesla autopilot and summon. In *Proceedings of the 8th international conference on automotive user interfaces and interactive vehicular applications*, pp. 225–228. ACM.

Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., & Koltun, V. (2017). Carla: An open urban driving simulator..

Dossa, R. F. J., Lian, X., Nomoto, H., Matsubara, T., & Uehara, K. (2019). A human-like agent based on a hybrid of reinforcement and imitation learning. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8. IEEE.

Dresner, K., & Stone, P. (2008). A multiagent approach to autonomous intersection management. *Journal of artificial intelligence research*, *31*, 591–656.

Fayjie, A. R., Hossain, S., Oualid, D., & Lee, D. (2018). Driverless car: Autonomous driving using deep reinforcement learning in urban environment. In *2018 15th International Conference on Ubiquitous Robots (UR)*, pp. 896–901.

Kaushik, M., Prasad, V., Krishna, K. M., & Ravindran, B. (2018). Overtaking maneuvers in simulated highway driving using deep reinforcement learning. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pp. 1885–1890. IEEE.

Koutník, J., Cuccu, G., Schmidhuber, J., & Gomez, F. (2013). Evolving large-scale neural networks for vision-based reinforcement learning. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pp. 1061–1068. ACM.

Koutník, J., Schmidhuber, J., & Gomez, F. (2014). Evolving deep unsupervised convolutional networks for vision-based reinforcement learning. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, pp. 541–548. ACM.

LaValle, S. M. (2006). *Planning algorithms*. Cambridge university press.

Li, Y., Song, J., & Ermon, S. (2017). Infogail: Interpretable imitation learning from visual demonstrations. In *Advances in Neural Information Processing Systems*, pp. 3812–3822.

Liang, E., Liaw, R., Moritz, P., Nishihara, R., Fox, R., Goldberg, K., Gonzalez, J. E., Jordan, M. I., & Stoica, I. (2018). Rllib: Abstractions for distributed reinforcement learning..

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., & Wierstra, D. (2019). Continuous control with deep reinforcement learning..

Lin, X., Adams, S. C., & Beling, P. A. (2018). Multi-agent inverse reinforcement learning for general-sum stochastic games. *ArXiv*, *abs/1806.09795*.

Lin, X., Beling, P. A., & Cogill, R. (2017). Multiagent inverse reinforcement learning for two-person zero-sum games. *IEEE Transactions on Games*, *10*(1), 56–68.

Littman, M. L. (1994). Markov games as a framework for multi-agent reinforcement learning. In *Machine learning proceedings 1994*, pp. 157–163. Elsevier.

Liu, G.-H., Siravuru, A., Prabhakar, S., Veloso, M., & Kantor, G. (2017). Learning end-to-end multimodal sensor policies for autonomous navigation. In Levine, S., Vanhoucke, V., & Goldberg, K. (Eds.), *Proceedings of the 1st Annual Conference on Robot Learning*, Vol. 78 of *Proceedings of Machine Learning Research*, pp. 249–261. PMLR.

Loiacono, D., Cardamone, L., & Lanzi, P. L. (2013). Simulated car racing championship: Competition software manual..

Loiacono, D., Lanzi, P. L., Togelius, J., Onieva, E., Pelta, D. A., Butz, M. V., Lonneker, T. D., Cardamone, L., Perez, D., Sáez, Y., et al. (2010a). The 2009 simulated car racing championship. *IEEE Transactions on Computational Intelligence and AI in Games*, *2*(2), 131–147.

Loiacono, D., Prete, A., Lanzi, P. L., & Cardamone, L. (2010b). Learning to overtake in torcs using simple reinforcement learning. In *IEEE Congress on Evolutionary Computation*, pp. 1–8. IEEE.

Lowe, R., Wu, Y., Tamar, A., Harb, J., Abbeel, P., & Mordatch, I. (2017). Multi-agent actor-critic for mixed cooperative-competitive environments. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, p. 6382–6393, Red Hook, NY, USA. Curran Associates Inc.

Minster, G., Haghighat, S., Chu, K., & Vogt, K. (2018). System and method for autonomous vehicle driving behavior modification.. US Patent 10,035,519.

Neider, J., Davis, T., & Woo, M. (1993). *OpenGL programming guide*, Vol. 14. Addison-Wesley Reading, MA.

Onieva, E., Cardamone, L., Loiacono, D., & Lanzi, P. L. (2010). Overtaking opponents with blocking strategies using fuzzy logic. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, pp. 123–130. IEEE.

Parisi, G. I., Kemker, R., Part, J. L., Kanan, C., & Wermter, S. (2019). Continual lifelong learning with neural networks: A review. *Neural Networks*, *113*, 54–71.

Pomerleau, D. A. (1989). Alvinn: An autonomous land vehicle in a neural network. In *Advances in neural information processing systems*, pp. 305–313.

Richter, S. R., Hayder, Z., & Koltun, V. (2017). Playing for benchmarks. In *International conference on computer vision (ICCV)*, Vol. 2.

Richter, S. R., Vineet, V., Roth, S., & Koltun, V. (2016). Playing for data: Ground truth from computer games. In *European Conference on Computer Vision*, pp. 102–118. Springer.

Ros, G., Sellart, L., Materzynska, J., Vazquez, D., & Lopez, A. M. (2016). The synthia dataset: A large collection of synthetic images for semantic segmentation of urban scenes. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 3234–3243.

Santara, A., Naik, A., Ravindran, B., Das, D., Mudigere, D., Avancha, S., & Kaul, B. (2018). Rail: Risk-averse imitation learning. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '18, p. 2062–2063, Richland, SC. International Foundation for Autonomous Agents and Multiagent Systems.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. *CoRR*, *abs/1707.06347*.

Seel, N. M. (2011). *Encyclopedia of the Sciences of Learning*. Springer Science & Business Media.

Shah, S., Dey, D., Lovett, C., & Kapoor, A. (2018). Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and service robotics*, pp. 621–635. Springer.

Shalev-Shwartz, S., Shammah, S., & Shashua, A. (2016). Safe, multi-agent, reinforcement learning for autonomous driving..

Shalev-Shwartz, S., & Shashua, A. (2016). On the sample complexity of end-to-end training vs. semantic abstraction training..

Sharifzadeh, S., Chiotellis, I., Triebel, R., & Cremers, D. (2016). Learning to drive using inverse reinforcement learning and deep q-networks. *CoRR*, *abs/1612.03653*.

Sulkowski, T., Bugiel, P., & Izydorczyk, J. (2018). In search of the ultimate autonomous driving simulator. In *2018 International Conference on Signals and Electronic Systems (ICSES)*, pp. 252–256. IEEE.

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

Talpaert., V., Sobh., I., Kiran., B. R., Mannion., P., Yogamani., S., El-Sallab., A., & Perez., P. (2019). Exploring applications of deep reinforcement learning for real-world autonomous driving systems. In *Proceedings of the 14th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications - Volume 5: VISAPP,*, pp. 564–572. INSTICC, SciTePress.

Wymann, B., Espié, E., Guionneau, C., Dimitrakakis, C., Coulom, R., & Sumner, A. (2000). Torcs, the open racing car simulator. *Software available at http://torcs.sourceforge.net*, *4*(6).

Yoshida, N. (2016). Gym-torcs. `https://github.com/ugonama-kun/gymtorcs`.

You, C., Lu, J., Filev, D., & Tsiotras, P. (2019). Advanced planning for autonomous vehicles using reinforcement learning and deep inverse reinforcement learning. *Robotics and Autonomous Systems*, *114*, 1 – 18.

Yu, L., Song, J., & Ermon, S. (2019). Multi-agent adversarial inverse reinforcement learning. In *International Conference on Machine Learning*, pp. 7194–7201. PMLR.