Journal of Intelligent and Connected Vehicles
2024, 7(3): 229–244
https://doi.org/10.26599/JICV.2023.9210039

Research Article

Development of deep-learning-based autonomous agents for low-speed maneuvering in Unity

Riccardo Berta, Luca Lazzaroni✉, Alessio Capello, Marianna Cossu, Luca Forneris, Alessandro Pighetti, Francesco Bellotti

Electrical, Electronics and Telecommunication Engineering and Naval Architecture Department (DITEN), University of Genoa, Genoa 16145, Italy

Received: November 4, 2023; Revised: December 20, 2023; Accepted: March 4, 2024

ABSTRACT: This study provides a systematic analysis of the resource-consuming training of deep reinforcement-learning (DRL) agents for simulated low-speed automated driving (AD). In Unity, this study established two case studies: garage parking and navigating an obstacle-dense area. Our analysis involves training a path-planning agent with real-time-only sensor information. This study addresses research questions insufficiently covered in the literature, exploring curriculum learning (CL), agent generalization (knowledge transfer), computation distribution (CPU vs. GPU), and mapless navigation. CL proved necessary for the garage scenario and beneficial for obstacle avoidance. It involved adjustments at different stages, including terminal conditions, environment complexity, and reward function hyperparameters, guided by their evolution in multiple training attempts. Fine-tuning the simulation tick and decision period parameters was crucial for effective training. The abstraction of high-level concepts (e.g., obstacle avoidance) necessitates training the agent in sufficiently complex environments in terms of the number of obstacles. While blogs and forums discuss training machine learning models in Unity, a lack of scientific articles on DRL agents for AD persists. However, since agent development requires considerable training time and difficult procedures, there is a growing need to support such research through scientific means. In addition to our findings, we contribute to the R&D community by providing our environment with open sources.

KEYWORDS: automated driving, autonomous agents, deep reinforcement learning, curriculum learning, modeling and simulation

1 Introduction

Motion planning is a key decision-making task in automated driving (AD) because it generates trajectories based on dynamic and kinematic models of vehicles (Dai et al., 2023; Gan et al., 2022; González et al., 2015; He et al., 2023). The literature typically divides the approaches into global and local planning. The former relies on prior knowledge (i.e., a map of the environment) and performs static and offline planning (Zhao et al., 2022). On the other hand, local planning exploits only real-time information from sensors and is typically used for implementing obstacle avoidance in limited areas (Peng et al., 2020).

Deep reinforcement learning (DRL) is an emerging method that combines the perception ability of deep learning (DL) with the decision-making support of reinforcement learning (RL) (Ding et al., 2022). These features make it particularly suited for addressing the typically dynamic, uncertain, and unstructured environment of local planning problems (Sun et al., 2021). As DRL agent policies are trained through a long and potentially very costly trial and error process, it is important to set up a proper simulation environment for pretraining before real-world deployment (Sun et al., 2021). Since custom simulators are expensive to develop and maintain, academics and industries are oriented toward a larger use of well-established open-source simulators. Some simulators are specifically dedicated to traffic (e.g., simulation of urban mobility (SUMO) (Lopez et al., 2018)) and automated driving development (e.g., car learning to act (CARLA) (Dosovitskiy et al., 2017) or highway-env (Leurent, 2018)). However, practitioners are also using more general, lower-level tools, such as game engines (e.g., Unreal (Unreal Engine, 2023), empowering CARLA, or Unity (Unity, 2023)), that provide the means for creating and operating real-time three-dimensional (3D) experiences and are supported by wide communities of developers, which makes these tools ever more appealing, especially for rapid prototyping.

Unity was developed for low-cost game development, and its use spread to various other industry areas, such as automotive, architecture and construction, and manufacturing. Unity has a relatively gentle learning curve and a flourishing developer community, which guarantees the availability of continuously updated documentation and code examples. This game engine also comes with ready-to-plug assets, even if not automotive specific, unlike CARLA, which features a variety of accurate vehicle models. While there is a wealth of information on blogs and forums reporting experiences with training and testing machine learning (ML) models with Unity, there is a lack of scientific articles about DRL agents for AD in Unity. On the other

✉ Corresponding author.
E-mail: luca.lazzaroni@edu.unige.it

hand, there is an increasing need to support such research through specific scientific means since agent development requires considerable training time and difficult procedures (e.g., possibly involving catastrophic forgetting (Goodfellow et al., 2015)). To address this information gap, the main novelty and aim of this article is to offer and discuss open code and data to support DRL agent development for low-speed AD maneuvering in Unity (Elios Lab, 2023). After providing some background information on DRL, this study dwells on two case studies: (1) parking in a garage and (2) static obstacle avoidance, reasoning about the strengths and weaknesses of the tool and of the methods. Along this path, this study will focus on some research questions that this study does not see adequately covered in the current scientific literature on Unity, such as the following: What are the main challenges to tackle in development? Can mapless navigation be extended to path lengths of tens of meters? How long is the training? Is curriculum learning (CL) useful? How is the computation distributed between the central processing unit (CPU) and graphics processing unit (GPU)? What is the generalization (i.e., knowledge transfer) ability of the agent? How does a trained DRL policy compare to a state-of-the-art policy, such as hybrid A*? What are the main lessons learned from the experience?

The remainder of this study is organized as follows. Section 2 presents related works, while Section 3 provides background on RL agent training methods. Section 4 describes the project workflow set up in Unity, while the experiments are presented and discussed in Section 5. Section 6 proposes a comparison of the usability of the proposed method with that of other tools used for DRL-based decision-making in automated driving. Conclusions on the work are drawn in Section 7.

2 Related works

Several different types of techniques have been developed for motion and path planning. In the area of global planning, a major role is played by the family of graph-search-based planning methods, which represent the state space of vehicles and other objects as an occupancy grid. Among these algorithms, we recall Dijkstra (Liu et al., 2021; Luo et al., 2020), A-Star (A*) (Boroujeni et al., 2017), Hybrid A* (Sedighi et al., 2019), and State Lattice (Pothan et al., 2017). Another group consists of sampling-based suboptimal planners who randomly sample the configuration or state space and search for connectivity (González et al., 2015). Among these methods, we cite the probabilistic roadmap method (PRM) (Kavraki et al., 1996) and the rapidly exploring random tree (RRT) (LaValle and Kuffner, 2001), which have been extensively used in automated driving. To overcome the suboptimality of RTT, Pohan et al. (2021) hybridized it with ant colony system (ACS) algorithms, achieving good performance and convergence speed. Another family of methods exploits curve interpolation, such as Bezier (Zheng et al., 2020) and splines (Wang et al., 2022a). Other algorithms employed for generating global trajectories simulate biological systems and behaviors, such as genetic algorithms (GAs) (Hao et al., 2020), ant colony algorithms (ACOs) (Nobahari and Nasrollahi, 2019), and particle swarm algorithms (Ling et al., 2021), which feature fast convergence and good robustness.

The pioneer algorithm in local path planning by Khatib (1990) is the artificial potential field (APF) algorithm. The algorithm drives the vehicle to the goal in an unknown environment by combining continuous attractive fields related to the goal-reaching task and repulsive fields generated by obstacles. Several variants have been proposed in the literature. For instance, Szczepanski et al. (2022) introduced a method for local minimum avoidance by predicting future movement to bypass obstacles in advance. Other employed techniques include simulated annealing (Grabusts et al., 2019) and fuzzy logic (Silva et al., 2021). Specific algorithms have been proposed for dealing with dynamic obstacles. For instance, the dynamic window approach with virtual manipulators (DWV) method (Kobayashi and Motoi, 2022) generates nonstraight line and nonarc candidate paths modified by predicting the position of dynamic obstacles.

The focus of this study is on DRL. The application of DRL to dynamic path planning in an unknown environment has been strongly enhanced by the application of the deep Q-network (DQN) (Mnih et al., 2015), which can learn successful policies directly from high-dimensional sensory inputs using end-to-end RL. Exploiting a lidar signal and local target position as the inputs, Lei et al. (2018) employed the double deep Q-network (DDQN) method (Van Hasselt et al., 2016) to train an agent that is able to reach the local target position successfully in an unknown dynamic environment. The agent policy features a convolutional neural network (CNN) for better generalization of the environment. Using asynchronous advantage actor-critic (A3C) RL (Mnih et al., 2016), Zhang et al. (2018) trained four-wheel robot navigation in urban search and rescue environments on rugged terrain.

Global and local planning can be combined for better overall performance. For instance, Faust et al. (2018) presented PRM-RL, a hierarchical method for long-range navigation task completion that associates sampling-based path planning with RL. The RL agents learn short-range, point-to-point navigation policies under the direction of PRM planning, enabling long-range navigation. Using a conceptually similar approach, Chiang et al. (2019) addressed long-range planning by combining RRT and a DRL agent that learns an obstacle avoidance policy that maps a robot's sensor observations to actions. The agent is used as a local planner during planning and as a controller during execution. The system demonstrates transferability to previously unseen experimental environments, making RL-RRT fast because the expensive computations are replaced with simple neural network inference. Gao et al. (2020) combined the twin delayed deep deterministic policy gradient (TD3) DRL algorithm with RPM global path planning to propose a novel path planner (PRM+TD3), which achieved promising results in terms of development efficiency and model generalizability. In Chu et al. (2023), DRL path planning based on DDQN was used to improve the planning of an autonomous underwater vehicle (AUV) under ocean current disturbance. Peng et al. (2021) applied DDQN for the path planning problem of an unmanned aerial vehicle (UAV), which exploits an edge server to execute computing tasks offloaded from multiple devices.

For a systematic review of the theory and applications of DRL methods in unstructured environments, we refer the readers to Sun et al. (2021). In Section 3, we provide the theoretical background for the DRL-based developments that we present in the experimental section of the paper.

Many simulation tools have been developed to test and exploit complete DRL pipelines and methods before deploying a learned agent in a real-world scenario. In this study, we examined several of the most popular open-source frameworks and analyzed their effectiveness. Highway-env is an open-source collection of environments for AD and tactical decision-making tasks implemented in Python by Leurent (2018). It relies on OpenAI's

Gym toolkit (Brockman et al., 2016), which allows a homogeneous approach to the development of RL methods and models. Highway-env environments are two-dimensional (2D) and are shown in a bird's-eye view. The project uses Stable-Baselines 3 (Raffin et al., 2021) as the library providing the implementation of state-of-the-art RL algorithms. The library, based on OpenAI baselines (Dhariwal et al., 2017), greatly supports hyperparameter tuning, comparison of existing methods, and the development of new methods. CARLA is an open-source simulator for developing AD systems in realistic urban and highway environments. The framework features a variety of physically accurate vehicle models and is able to generate highly detailed maps that reflect real-world road networks, complete with accurate traffic flow, infrastructure information, weather conditions, and pedestrian simulation. CARLA provides a comprehensive set of APIs and ancillary tools.

The web is rich in valid resources for DRL training on Unity (Buckley, 2023; Simonini and Sanseviero, 2023). There are also scientific articles (Almón-Manzano et al., 2022; Urmanov et al., 2019) and books on the topic (Majumder, 2021), but they typically deal with videogame applications/examples and do not address significant AD use cases. Experiences reported using other environments (Anzalone et al., 2022, on CARLA) are useful but not comprehensive given the specificities of the different simulation environments (e.g., timing and event management, vehicle and sensor models). On the other hand, there is a growing interest in the automotive industry (e.g., Martin and Zhai (2019)), which calls for more systematic and comprehensive literature contributions in the field.

A major aspect of our analysis concerns CL. CL is a machine learning (ML) strategy that trains a model from easier data to harder data, which imitates the meaningful learning order in human curricula, improving the generalization capacity and convergence rate of various types of models (Wang et al., 2022b). The original concept of CL was first proposed by Bengio et al. (2009). It starts by optimizing a heavily smoothed version of the loss function and gradually moves to the target (more complex) function. Tracking the local minima throughout the training guides the model toward better convergence (Bengio, 2014). In Florensa et al. (2017), CL enables RL agents to solve hard goal-oriented problems that cannot be solved without curricula. In the AD domain (but not using RL), the superiority of a semiautomatic CL strategy has been demonstrated for the estimation of the roll and sideslip angles of autonomous vehicles (Bae et al., 2021).

3 Reinforcement learning algorithm

Together with supervised and unsupervised learning, RL is a major paradigm of ML. RL trains an autonomous agent to perform optimal actions in a given environment. Learning occurs in a trial-and-error process through interactions between the agent and the environment. The interactions include the environmental observations periodically made by the agent and the corresponding actions taken by the agent. The consequent rewards provided by the environment are processed, together with the observations and actions, by an RL algorithm that refines the agent's policy (i.e., its decision strategy) during the various training iterations (Fig. 1).

The policy is a map between observations (i.e., environmental states) and actions and can be implemented as a look-up table, a complex function, or even a stochastic function providing probabilities. In each training episode, the agent has the goal of maximizing the total cumulative reward. To this end, it has to

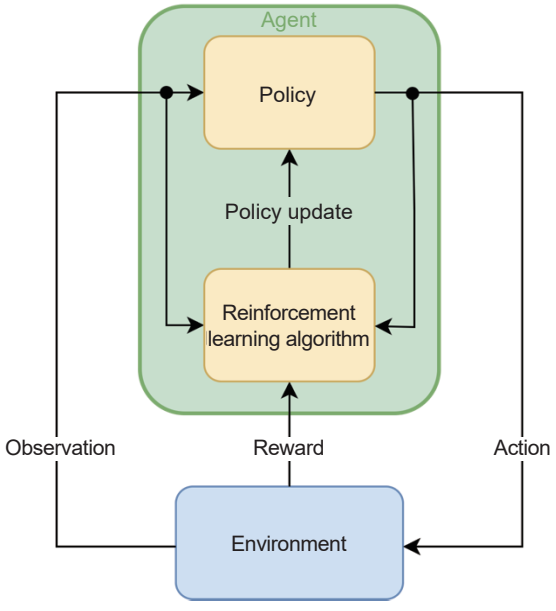


Fig. 1 Reinforcement learning loop.

prefer, state by state, those actions that it experienced to provide a good reward in that state. However, to discover such actions, it must also try actions not previously chosen. In other words, the agent must exploit what is already learned but also explore new actions to ensure that the best possible choice is made. This is the well-known exploration–exploitation trade-off (Sutton and Barto, 1998).

An RL problem is typically modeled as a Markov decision process (MDP) involving a set of states S, a transition function T, and a reward function R. At each time step, an agent is in a state s, executes an action a and enters the next state s' with a transition probability $T(a, a, s') \in [0, 1]$ and a reward $R(s, a)$. The agent learns a stochastic policy, which associates, in the state space, a probability on the set of the available actions $p(a|s)$. The goal is to find the best policy p^* that maximizes the expected sum of the rewards in an episode, which can be expressed as Eq. (1):

$$\pi^* = \underset{\pi}{\operatorname{argmax}} \mathbb{E}_\pi \left\{ \sum_{k=0}^{H-1} \gamma^k r_{t+k} \middle| s_t = s \right\} \tag{1}$$

where π is the policy, r is the reward, and t is the current timestep. The discount factor $\gamma \in [0, 1]$ controls how the agent considers future rewards. Low values encourage the agent to maximize short-term rewards, and large values imply a longer-term perspective. H is the horizon, intended as the number of steps in the MDP. It can be set to an infinite or a finite number if the episode ends after a given number of steps or as soon as a given final state is reached. In this last case, a γ close to 1 is usually preferred to incentivize the agent to reach the goal. Lower γ values are preferred in infinite time domains to balance long- and short-term rewards.

In typical decision-making problems, the size of the MDP state space is large, and deep neural networks are frequently used to model policy or other learned functions (deep reinforcement learning (DRL)). Various types of RL training methods have been developed over the years, and they can be classified from different perspectives. The first major distinction is between value-based and policy-based algorithms. Value-based methods, such as Q-learning (Sutton and Barto, 1998), use a value network to estimate the Q-value, which is the overall utility of each single (s, a) pair (i.e., the expected total reward), assuming that the agent

continuously follows a policy π. The *Q*-table is randomly initialized and iteratively updated using a learning rate between 0 and 1. The table update policy can be expressed as Eq. (2):

$$Q(s,a) = Q(s,a) + \alpha\left(r + \gamma \max_{a' \in A(s')} Q(s',a') - Q(s,a)\right) \quad (2)$$

where Q is the state-action value function, r is the reward, α is the learning rate, and γ is the discount factor. Given its structure, *Q*-learning is unable to scale with the number of states and actions, which is a clear issue, particularly with continuous-action problems (Sutton and Barto, 1998). Moreover, because of greedy action selection, value-based RL algorithms cannot address stochastic policy problems (Gu et al., 2022). In policy-based methods, instead, a policy is learned directly, typically through gradient descent optimization of the weights of a neural network, to maximize the expected reward. The policy may be deterministic (e.g., deterministic policy gradient (DPG) (Silver et al., 2014)) or, in the case of probabilistically selected actions, stochastic (e.g., proximal policy optimization (PPO) (Schulman, et al., 2017b)). Actor-critic methods combine the benefits of value-based and policy-based approaches (Wang et al., 2017, 2020). The 'actor' network implements the actual policy, being able to deal with a continuous action space, while the 'critic' network implements a value function, allowing the update of the policy with a lower variance of trajectories, which is the historical sequence of states, actions, and rewards.

Another difference is between on-policy and off-policy methods. During training, off-policy methods follow a behavior distribution that ensures adequate exploration of the state space and thus differs from the 'target policy', which is the one actually optimized. For instance, *Q*-learning follows an $\mathcal{E}$-greedy strategy, in which the agent selects a random action with probability $\mathcal{E}$ and chooses the action with the highest *Q*-value with probability $1 - \mathcal{E}$ (Mnih et al., 2015), implementing the exploration–exploitation trade-off.

A final distinction is between model-based and model-free methods, depending on whether or not they build a model of the environment. A nonexhaustive taxonomy of RL algorithms is shown in Fig. 2, highlighting the categorization in terms of *Q*-learning and policy-based methods.

This work focuses on the policy gradient/actor-critic proximal policy optimization (PPO) algorithm, which supports a continuous action space, as per our problem's requirements. PPO was introduced in 2017 (Schulman, et al., 2017b) and has become a reference for continuous control problems. In policy gradient methods, the value estimation network is subject to noise, as it is updated for every experience sample. A previous method, trust region policy optimization (TRPO), limits the policy gradient step to restrict policy variation (Schulman, et al., 2017a). PPO features

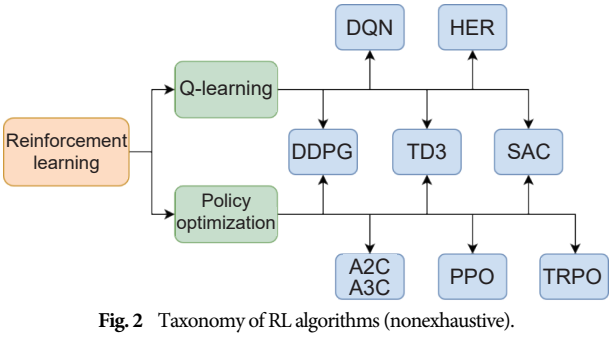


Fig. 2 Taxonomy of RL algorithms (nonexhaustive).

an objective function that enables multiple epochs of minibatch updates, thus achieving some of the advantages of TRPO but with better implementation simplicity and generalizability and higher sample efficiency.

4 ML-Agents' project workflow

We developed our agents in Unity, a popular cross-platform game engine that is used for the efficient development of games and simulations in various industrial domains (Unity, 2023). Machine learning agents (ML-Agents) (Juliani et al., 2020) are an open-source toolkit that allows the use of Unity as a simulation environment to create and train autonomous agents. ML-Agents provides a python application programming interface (API) to an implementation, based on the PyTorch library, of the main RL algorithms. These are the key components of ML-Agents:

- Learning environment: This is the Unity scene providing the environment in which the agent observes, acts, and learns. The ML-Agents Toolkit SDK allows wrapping any Unity scene as a learning environment, defining the agents and their behaviors. In a learning environment, it is possible to train more than one agent concurrently, significantly reducing the training time. The training area must of course be built so that the agents do not interact with each other.
- Mlagents-learn: This is the ML-Agents' core utility, which manages the training. The utility is launched with a *.yaml* configuration file. The file is divided into several sections: behaviors (specifies the training algorithm and its hyperparameters), environment (specifies the environment path, the environment arguments, if any, and the number of environments to be parallelized), engine (specifies the rendering settings, i.e., the screen dimension, the render quality, the time scale, and whether to render the scene or not), checkpoint (contains info about the creation of checkpoints during the training), and torch (specifies whether to use the CPU or the GPU for the training).
- Python low-level API: This API allows communication with the Unity scene during training.
- External Communicator: This component, which is internal to the learning environment, allows communication with a Python API.
- Python trainers: contains the algorithms used to train the agents. It provides a command-line utility (namely, mlagents-learn) and is interfaced with the python low-level API only.

Fig. 3 shows the architecture of an example ML-Agent learning environment. In the example, two agents (namely, A1 and A2) are trained by the python trainer through a python API. The communicator module connects the agents to the python API while also retrieving the environmental parameters needed for training (e.g., the target coordinates and the agent location). For the sake of completeness, two other agents are included in the environment. The first one is an agent featuring a policy implemented through a neural network (NN), which was already trained in a previous RL session. The second is a heuristic agent, which implements a behavior determined by a set of heuristic rules.

Fig. 4 provides an overview of the typical ML-Agents project workflow. The initial design and implementation step concerns the definition of fundamental aspects such as simulation settings; observations, actions, and reward signals; and NN hyperparameters. The subsequent training consists of executing several simulation episodes, with possible tweaks (e.g., in the reward function). When the per-episode reward gained by the

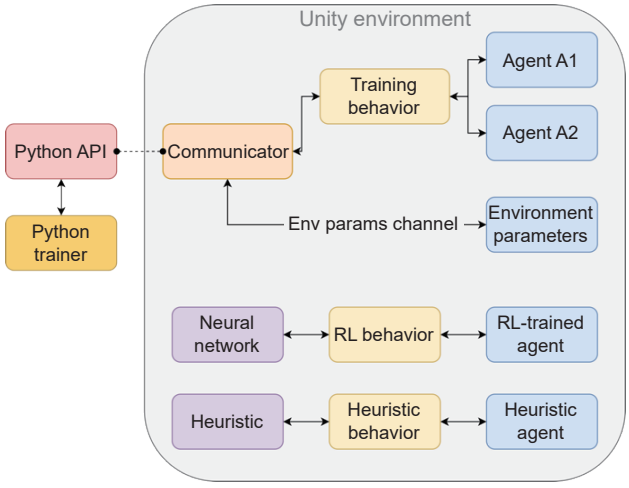


Fig. 3 Example of an ML-Agent learning environment

agent reaches a suitable level, the training is concluded, and the learned policy can be tested via pure inference. If the test is successful, the model can be deployed; otherwise, training has to restart with a model updated by the designer according to the gained experience.

The workflow for each episode (including the setting of the parameters that are randomized to stimulate the agent's generalization capabilities) is sketched in Fig. 5. In the first phase, the environment is initialized according to the user definitions, and the vehicle and target positions are randomly generated. Then, the episode loop performs the typical RL loop with the choice of an action by the agent and the consequent modification of the environment. The new PPO iteration is fed by the observation of the environment and the gained reward. Finally, the episode ends either by a collision, a timeout or the reaching of the goal.

5 Experiments

To implement the vehicle that hosts the DRL agent, we exploited a simple 3D model available at Dev (2023). The vehicle's collider component, which allows the detection of collisions, has the shape of a parallelepiped, with a size of 4.90 m × 1.90 m. Given the scope of the low-speed maneuver, we employ a simple kinematic bicycle model for the vehicle (Polack et al., 2017), with Eq. (3) (see also Fig. 6):

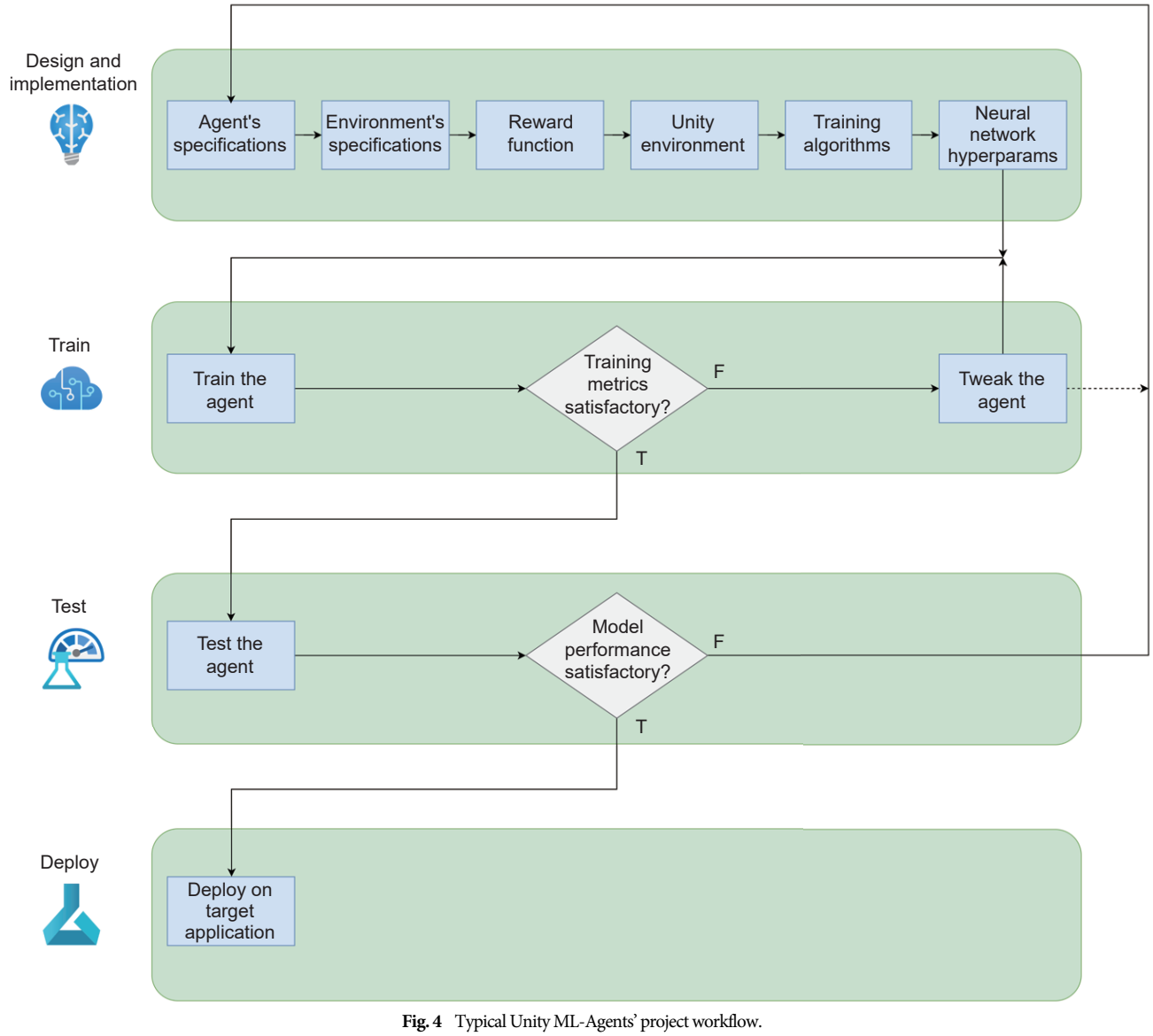


Fig. 4 Typical Unity ML-Agents' project workflow.

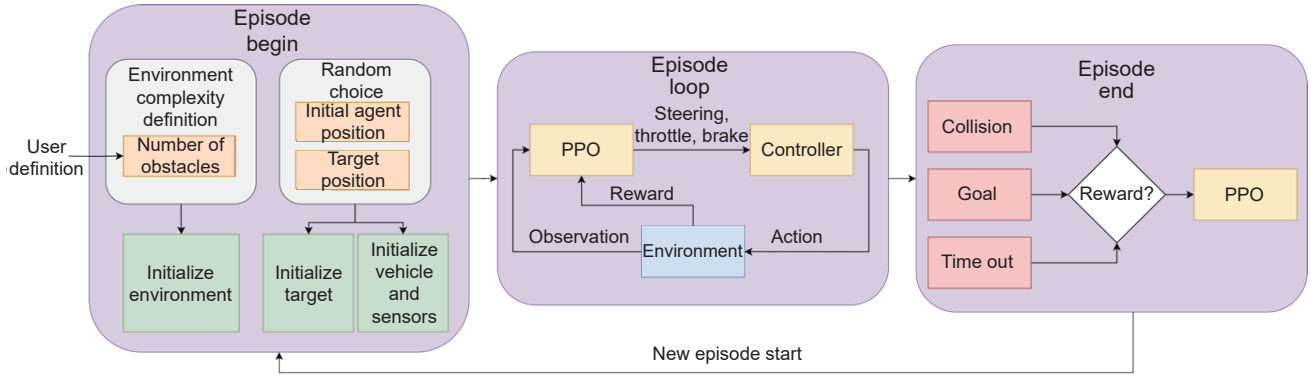


Fig. 5 ML-Agents' training episode workflow.

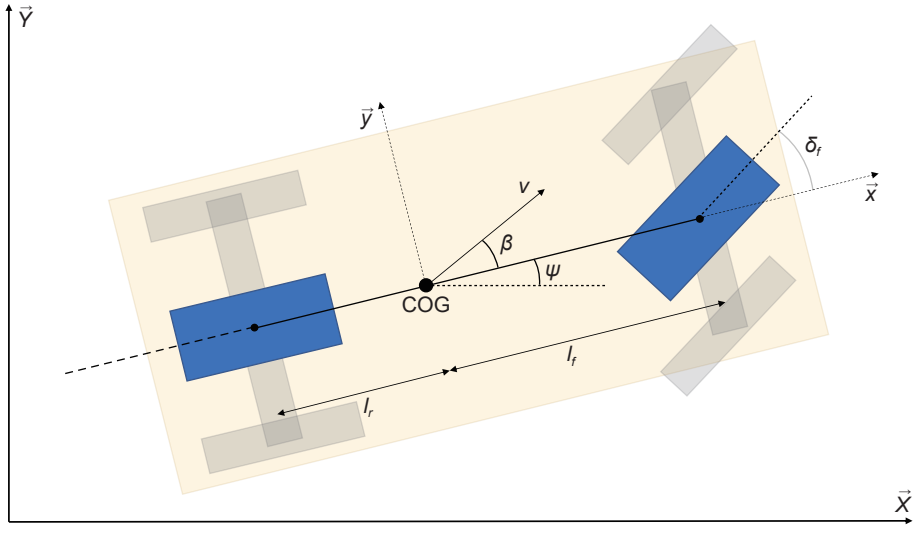


Fig. 6 Kinematic bicycle model.

$$\begin{cases} \dot{X} = V\cos\left(\psi + \beta\left(u_2\right)\right) \\ \dot{Y} = V\sin\left(\psi + \beta\left(u_2\right)\right) \\ \dot{V} = u_1 \\ \dot{\psi} = \dfrac{V}{l_r}\sin\left(\beta\left(u_2\right)\right) \end{cases} \tag{3}$$

where u_1 represents the acceleration command and u_2 is the front wheel angle used as a steering command. $\beta\left(u_2\right)$ is the slip angle at the center of gravity:

$$\beta\left(u_2\right) = \arctan\left(\tan\left(u_2\right)\frac{l_r}{l_f + l_r}\right) \tag{4}$$

The agent is supplied by the environment with information about the position of the vehicle and the target and the speed vector. Moreover, it receives a stream of data from a lidar and/or a camera. The lidar, with a 360° field of view and an angular resolution of 10° (Fig. 7), is placed on the top and in the center of the roof to produce a 1D array. The radius length is 5 m (we intentionally adopted such a short range to try to enhance the generalization capability of the agent). The camera, placed over the windscreen, has a field of view of 60° vertically and 120° horizontally and a resolution of 84 × 84 pixels. The lidar is implemented through the ML-Agents raycast sensor component and the camera through a camera component. The raycast sensor provides a vector containing the distances (d) of the last stacked observation (s) for each of the n rays. The camera sensor provides a 2D vector (size x times y) of the recorded image. The complete observation vector is provided in Eq. (5):

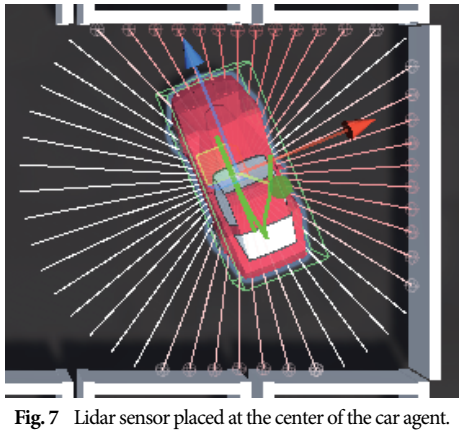


Fig. 7 Lidar sensor placed at the center of the car agent.

$$\begin{cases} S_{\text{lidar}} = (n, d, s) \\ S_{\text{camera}} = (x, y) \\ S_{\text{env}} = \left(\text{agent}_{\text{vel}}, \text{agent}_{\text{dir}}, \text{target}_{\text{dir}}, \text{distance}_{\text{agent}\to\text{target}}\right) \\ S = \left(S_{\text{env}}, S_{\text{camera}}, S_{\text{lidar}}\right) \end{cases} \tag{5}$$

The agent can control the motor torque, the brake torque, and the direction of its front wheels. The steering angle range is $[-\pi/6, \pi/6]$ rad, the brake torque range is [0, 300] Nm, and the motor torque range is [−400, 400] Nm, allowing the vehicle to perform maneuvers forward and backward.

The NN model chosen is a multilayer perceptron (MLP) with two hidden layers of 256 neurons each. When the camera sensor is present, the NN has two additional convolutional layers to

Table 1 Characteristics of the NN architecture

Layer	Size	Description
Input layer	(408 or 84 × 84 × 3) + 8	408 lidar values or an 84 × 84 RGB image + 2D values of distance between agent and goal, agent's speed, agent's heading, and goal's heading
Convolutional layer	1st Kernel size = [8, 8] 1st Stride size = [4, 4] 2nd Kernel size = [4, 4] 2nd Stride size = [2, 2]	2 convolutional layers to pre-process the camera input, when provided
Dense layer	256	2 layers
Output layer	3	Possible agent's actions: throttle, steering, brake

preprocess the visual signal. Table 1 synthesizes the values characterizing the network architecture.

In the following, we describe the two experiments we conducted to verify our approach: parking in a garage and random obstacles.

5.1 Parking in a garage

Using 3D settings, we built from scratch a Unity scene representing a garage environment with ten parking lots in a 400 m^2 area (Fig. 8).

Each lot, 5.3 m × 3.5 m in size, is delimited by walls. The central corridor, which is drivable in both directions, is 20 m × 8 m. At each episode, only one parking lot is open. To speed up the training, we replicated the same environment 10-fold in a single scene, allowing the use of up to 10 vehicles simultaneously. The goal of the DRL agent is to define and execute a path from a random initial position to the target lot, avoiding any possible collision with the walls.

A key factor in autonomous agent design is represented by the reward function, which is the means through which the environment shapes the behavior of the agent. We started by reproducing the reward function (Eq. (6)) of a popular open-source project based on ML-Agents (Raju, 2020). This reward penalizes the agent for collisions and rewards it when it is moving and when it reaches the target. Alignment of the vehicle to the parking lot is also rewarded.

$$\text{reward} = \begin{cases} 0.2 \times |\text{alignment}|, & \text{goal and car parked facing the wall} \\ 0.8 \times |\text{alignment}|, & \text{goal and car parked facing the road} \\ -0.01, & \text{collision} \\ 0.001 \times \text{target}_{\text{dir}} \times \text{target}_{\text{velocity}} \end{cases} \tag{6}$$

where $\text{alignment} = \|\text{agent}_{\text{vel}}\| \times \|\text{agent}_{\text{dir}}\| \times \cos\theta$ and θ is the angle between the ego heading vector and the ego-target conjunction vector, respectively.

Through various experimental iterations, we refined the reward function to achieve better results in terms of both accuracy and convergence time:

$$\text{reward} = \begin{cases} -1, & \text{collision} \\ 0, & \text{goal} \\ c_1 \times \left[\left(c_2 \times d^2 \right) + \dfrac{c_3 \times (1 - \cos\theta)}{d + 1} \right] \end{cases} \tag{7}$$

where $c_1 = 0.01$, $c_2 = -0.01$, and $c_3 = -1.5$.

The function sums two rewards: two sparse rewards (i.e., given when a specific event occurs) and one dense reward (i.e., given at every simulation step). The first sparse reward penalizes the agent in case of collision. The second one rewards the achievement of the goal. An additional goal reward is added proportionally to the final alignment of the vehicle to the parking lot. The dense reward aims at favoring the approach to the goal and consists of two components that correspond to the two targets of final position and final orientation. The first term penalizes the distance d from the target, and the second term penalizes the misalignment from the parking lot. The second term is divided by the distance d, as controlling misalignment is meaningful only when the vehicle is close to the target. We use the Manhattan distance, which is more appropriate for the environment. The dense reward is normalized in the [–2, 0] interval. The sparse and dense rewards are carefully weighted through the c_1, c_2, and c_3 factors to empirically achieve a good balance.

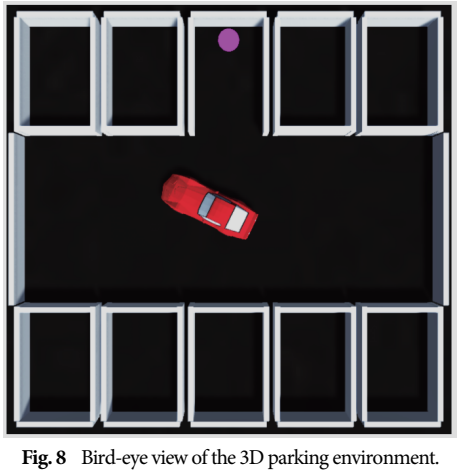


Fig. 8 Bird-eye view of the 3D parking environment.

For the DRL method, we used the PPO, which is the reference for continuous control problems. The best values we could empirically find for the training hyperparameters are reported in Table 2.

The scene is reinitialized at each training episode. One parking lot is selected as the target, and its door is opened; the agent is set in a random position and orientation in the central corridor. An episode ends when one of these conditions occurs: (1) the vehicle reaches the goal; (2) the vehicle collides with an obstacle (this criterion is not applied in the initial training phase, when we prefer to let the agent continue the exploration after a collision); or (3) the maximum number of steps for an episode is reached without arriving at the goal.

We considered the following performance metrics for the

Table 2 Best hyperparameter values for the experimental environment, empirically found

Hyperparameter	Value
Batch size	512
Buffer size	51,200
Learning rate schedule	Linear
Learning rate (start)	1×10^{-4}
Time horizon	128

training phase:

- Cumulative reward. It is the sum of all the rewards accumulated by the agent in an episode. The evolution in time of this metric is a major indicator of the effectiveness of the training. As rewards are given only during training, this quantity, unlike the next quantity, is not applicable in testing.
- Goal rate. The ratio between the number of episodes that ended up reaching the goal and the total number of episodes.
- Collision rate. The ratio between the total number of collisions and the total number of episodes. As we will see, in some cases, collisions are terminal; in others, they are not, depending on the training policy design choice.
- Timeout rate. The ratio between episodes ending with a timeout (i.e., doing the maximum number of steps without reaching the goal) and the total number of episodes.

For our analysis of the garage scenario, we conducted three subexperiments, which are described below.

- Subexperiment No. 1: There is a single target lot for all the training episodes, thus with limited generalizability. Collision is not a terminal event. The vehicle is equipped with a lidar.
- Subexperiment No. 2: The lidar is replaced with a camera, and the NN architecture is modified to manage the different inputs. Specifically, two convolutional layers are added to process the camera signal, and the output is concatenated to the other inputs.
- Subexperiment No. 3: The vehicle is equipped with a lidar and trained to reach one of the ten possible parking lots, which are randomly selected at the beginning of each episode. A collision is not a terminal event until the success rate reaches a certain threshold; then, the collision is set as a terminal event.

For all the subexperiments, the testing phase consists of 100 episodes in which the vehicle is spawned in a random position and orientation and has to reach one randomly selected lot among the 10 available. The relevant results are reported in Table 3.

Regarding the distribution of computations during training, in this experiment (and in the other experiments), we did not notice a significant improvement by using a dedicated GPU. We argue that this is because the NN used is not so heavy in terms of depth and neurons. Moreover, RL is CPU intensive due to sequential agent-environment interactions (Fig. 1), and rendering is suppressed when a camera is not used. The training speeds measured are approximately 1.4 M step/h, using a device equipped with an Intel Xeon W2223 processor, 32 GB of RAM, and an NVIDIA Quadro RTX 4,000 GPU.

Table 3 Testing results of each experiment

Subexp. No.	Sensor	Number of target parking lots in training	Success rate in test
1	Lidar	1	50%
2	Camera	1	80%
3	Lidar	10	94%

For subexperiment No. 1, we noticed a critical phase in the first 8 M steps, with low/unstable reward and episode lengths dominated by collisions and then by timeouts. In particular, we observed that in several cases, the vehicle repeatedly executed sudden gear inversions, substantially remaining in the same position. This corresponds to a local minimum, which prevents the agent from improving its performance. After 8 M steps, the agent performance quickly reaches a stable state with a high reward and short episode lengths. However, the test phase, where the target can be any of the 10 lots, reveals that the trained agent is not able to make a proper generalization (Table 3).

In the second subexperiment, we substituted the lidar with a camera. The training proved effective. However, in this case, the agent's generalization capability to multiple target test cases is not satisfactory (Table 3). We thus tried to restart the training by randomly simulating all the possible target lot cases, which slowed learning without leading to significant improvements. Additionally, the idea of penalizing the timeouts did not provide significant advantages.

In the third subexperiment, we returned to use a lidar instead of a camera, which reduced the input size and the training time. We also changed the values of the constants and normalized the rewards. For the tuning of such hyperparameters, it was very important to create and analyze additional TensorBoard charts, as shown in Fig. 9. In particular, we logged in Tensorboard the value of all the constituents of the reward function (i.e., collision, goal, distance, and alignment) to quantitatively check agent behavior. This was key to efficiently assessing various alternatives and tuning the reward function and simulation settings, which is a very time-consuming task.

This analysis was also very useful for tuning the CL. CL in RL is tricky because there is no predefined dataset, and the difficulty of evaluating samples can be evaluated by a difficulty measurer (Wang et al., 2022b). Thus, similar to Anzalone et al. (2022), we used a one-pass algorithm (Bengio et al., 2009), with increasing difficulty levels associated with more complex versions of the learning environment. We could have also gradually increased the agent's capabilities in different training stages Wei et al. (2023), but preliminary results showed that the increased training complexity did not provide benefits. In contrast to Anzalone et al. (2022), we did not set a predefined number of steps for each stage but adopted a performance criterion (Wang et al., 2022b). Moreover, at each stage, we not only increased the context complexity (e.g., type of goal, condition of episode termination) but also tweaked the agent's reward function by carefully analyzing the Tensorboard charts of various training attempts, as mentioned above. For this first experiment, we implemented two stages. In the first stage, the agent is trained to reach a single target lot. Collisions are penalized but do not terminate the episode to allow the agent to explore the environment without frequent interruptions (Lazzaroni et al., 2022). When the goal is achieved with a rate greater than 99.5%, the second stage starts. In this stage, the other 9 targets are randomly activated, collisions become terminal events for an episode, and the collision penalty is increased (the reward decreases from –1 to –10). The introduction of these factors causes the sharp performance drop of the goal reward that we can observe at the stage transition in Fig. 9b. However, the training resumes effectively (even if more slowly because of the much greater complexity of the learning environment) and, after a sufficient number of iterations, allows satisfactory performance in the target environment to be reached.

The first stage is completed at 25 M, while further steps at 57 M

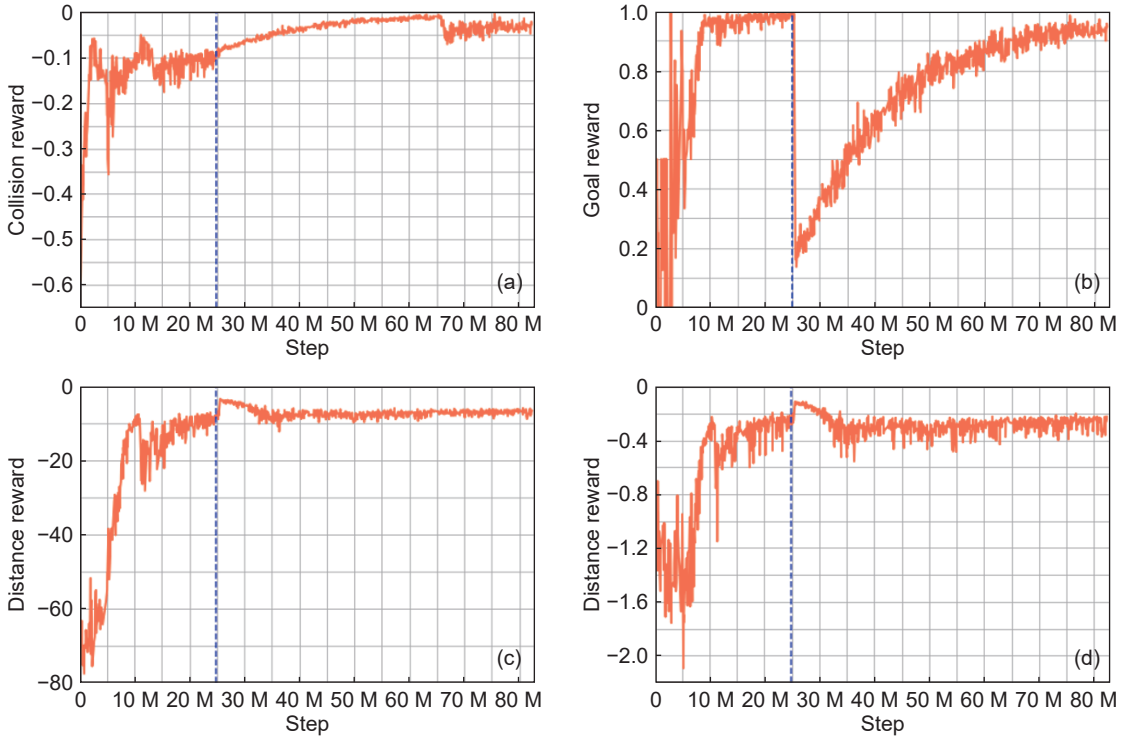


Fig. 9 Tensorboard plot of the single components of the reward function in the third subexperiment: (a) the collision reward, (b) the goal, (c) the distance, and (d) the alignment. The dashed blue lines indicate the switch from the first to the second stage.

Fig. 10 Evolution of the training (success rate) in the third subexperiment. The dashed blue line indicates the switch from the first to the second stage. The sharp performance drop is due to the increased collision penalty, and the episode stops in the case of collision.

are needed for the second stage. Fig. 10 reports the evolution of the success rate (i.e., goal reach, without collisions) during training, with the abovementioned sharp performance drop at the change of the stage. The test results show that training on multiple targets appears to be necessary for an agent to reach multiple targets (Table 3), which means that training on a single target causes the agent to overfit the actual position of the target.

We assessed the impact of CL by comparing the achieved results with those of an agent's training starting from the beginning with the same conditions and rewards of our stage 2 agent. Impressively, even after 80 M iterations, the success rate persistently remains at 0. We argue that a pivotal factor for CL is the possibility of having initial episodes with nonterminal collisions. This not only promoted exploration of the environment (characteristics of the movement area and effects of the actions) but also exposed the agent to the rewards associated with reaching the target. On the other hand, the agent heavily penalized since the beginning of collisions learned that staying was still more advantageous than moving in a risky environment. Thus, CL fosters an explorative learning process, steering the agent away from overly cautious behaviors and toward a more adaptive learning strategy.

We compared the test results from the third subexperiment with those of Hybrid A*. Hybrid A* is a variant of the well-known A* search algorithm, applied to the 3D kinematic state space of the vehicle, which guarantees kinematic feasibility of the path through a state-update rule that takes into account the continuous state of the vehicle in the A* nodes (Dolgov et al., 2008; Petereit et al., 2012).

As a Unity implementation of the Hybrid A* algorithm is available open-source (Nordeus, 2022), we applied it to our test environment and achieved results that are presented in Table 4. To generate the path, Hybrid A* discretizes the map in 25 cm squared cells. side. Every cell in the map represents a node with which a cost is associated in the path generation process. A heuristic function accounts for the vehicle's possible movements, considering a discrete set of actions given by three steering angles: [−30°, 0°, 30°]. A flow field around the obstacles is generated to avoid collisions. The value of its extension can be controlled by modifying the safety margin. The results show that Hybrid A* achieves a 100% success rate, which decreases to 85% in the actual path following phase because of the discretization of the map (Table 4). Performance measured on an Intel Xeon W2223 machine with 32 GB of RAM indicates that more than 0.8 s are needed to compute the path (which is not needed by the DRL agent). Achieving more real-time performance with Hybrid A* (almost 0.2 s) requires increasing the size of the cells to 1.2 m, which, however, leads to a decrease in the target reach rate to 21%. Table 4 also shows that the DRL agent can reduce the number of gear inversions needed to reach the target by 25% on average. Thus, the DRL agent achieves better performance (goal reach rate, gear inversion rate, latency) and does not need a map.

As important lessons learned, our experience stresses that some

Table 4 Comparison of 100 episodes of Hybrid A* PPO and DRL PPO in the Garage environment

Path planning algorithm	Success rate	Gear inversions per episode	Time for path planning
Proposed DRL	94%	3.36	—
Hybrid A*	85%	4.48	0.857 s

factors are key to properly setting up the environment and avoiding systematic (and difficult-to-spot) errors. The simulation tick (i.e., the time between two consecutive updates of the environment) should be fast enough to guarantee a faithful physical simulation (otherwise, some collisions and/or goal reach may be detected too late, if not missed) but should be tuned according to the actual speed of the simulated items and the CPU capacity. The decision period (i.e., the number of ticks between two consecutive agents' decisions) is the other time-related setting that needs careful tuning. A short decision period leads to the vehicle being stationary. Our interpretation is that, in this case, the agent does not have the time to see the effect of its actions. With a too long period, on the other hand, the agent becomes insufficiently reactive to properly manage the vehicle's dynamics. To speed up the training phase, we used the possibility offered by ML agents running multiple training environments at the same time. This increases the ratio of steps/second and the load on the CPU. For this reason, the number of parallel environments should not be too high because after a certain (machine-dependent) threshold, the step/second ratio begins to decrease.

The choice of the sensor(s) has a strong impact on how the agent learns and behaves but also on the training time. The decision between a camera sensor and the lidar has a strong impact because the camera requires the rendering of the scene at each timestep, reducing the number of instructions per second and increasing the training time. A camera also requires a larger NN (two convolutional layers were added to process the signal), which also impacts the training time. Since the goal-reach results between the camera and lidar for our experiments were similar, we decided to prioritize the second one. Concerning generalization, we noticed some clear limits: when something changes in the configuration (width or length of the driving area, position of the parking lots, etc.), performance drops significantly, and the agent needs retraining.

5.2 Random obstacles

The previous experiment trained an agent to reach any target parking lot, starting from a random position in a garage. As a second investigation, we explored an environment characterized by the presence of random obstacles before the target, with longer origin–destination distances. To analyze this, we set up another Unity environment, which is analyzed in this subsection.

The input type, neural network configuration, and RL training method are the same as those for Garage subexperiment No. 3. The goal of the agent is to reach a target random position in an 80 m × 80 m space, avoiding collisions with obstacles placed within the scene. The number of obstacles ranges between 3 and 9, and each obstacle is placed between the starting position of the agent and the goal, with a random orientation. The presence of obstacles aims at adding variety to the environment, requiring the planning of a proper path to avoid them (e.g., Fig. 11).

For the reward function, we used a similar approach as for parking but also added a low-speed penalty. This addition was chosen to avoid situations, as experimentally observed, in which the agent stops close to the target, without ever reaching it, and is seemingly satisfied with the short distance reached (likely compared to the risk of a collision).

For the training, we used a one-step CL procedure, as in the first experiment, with increasing difficulty in terms of the number of obstacles present in the scene. Once the agent masters a difficulty level, the training continues in more complicated settings. The difficulty levels are reported in Table 5. The weakening of hyperparameters (including the reward function) may also occur at these levels, based on the agent's behavioral observations.

The TensorBoard chart in Fig. 12 shows the evolution of the training success rate across the different difficulty levels. Regarding the first three levels, it appears that the entry level is accomplished in 11 M steps, the medium level after an additional 36.5 M steps, and the third level is completed after an additional 363.5 M steps. The decrease in performance at each level shows a lack of generalization ability on the agent's side. In fact, at each stage, the agent seems to learn to reach a target with the current obstacle schema but not to reach a target without collisions (which is our ultimate generalization goal). However, the drop decreases at higher difficulty levels, suggesting that eventually, the agent learns to generalize. As it appears, every level switch requires specific training, which is slowly but clearly achieved by the agent.

Fig. 13 shows the performance of a system directly trained at the final difficulty level (Table 5). In this case, unlike in garage parking, agents trained with no CL are able to achieve considerable performance, even if they still have a gap with respect to CL (95% vs. 98%). We argue that the difference with the Garage case study could be attributed to the larger spaces between obstacles in this second environment. Although collisions are terminal since the training outset, they are not frequent, so the "novice" agent can explore the environment and learn the task. A comparison with Fig. 12, however, shows that the agents start learning much later because of the higher environmental

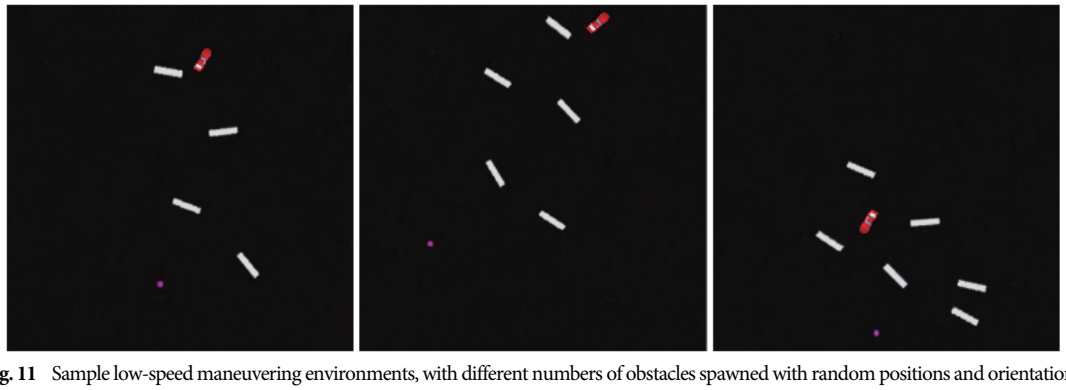


Fig. 11 Sample low-speed maneuvering environments, with different numbers of obstacles spawned with random positions and orientations.

Table 5 Difficulty levels

Difficulty level	Description
1	No obstacles.
2	One obstacle midway between the spawning point and the target.
3	Like level 2 but with two additional obstacles on the sides.
4	Like level 3 but an additional obstacle is placed in a second series of obstacles.
5	Like level 3 but two additional obstacles are placed in the second series of obstacles.
6	Like level 3 but three additional obstacles are placed in the second series of obstacles.

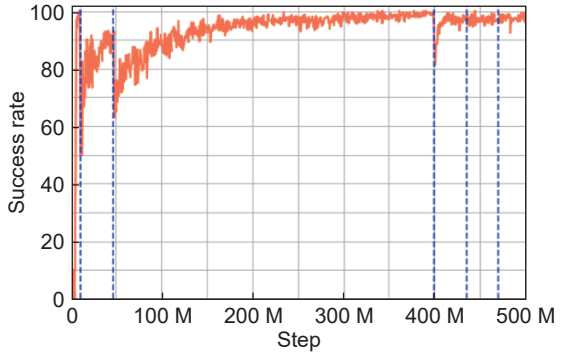


Fig. 12 Evolution of the training success (i.e., goal reached) rate at the six different difficulty levels (Table 5). Difficulty level switches are indicated by the dashed blue line.

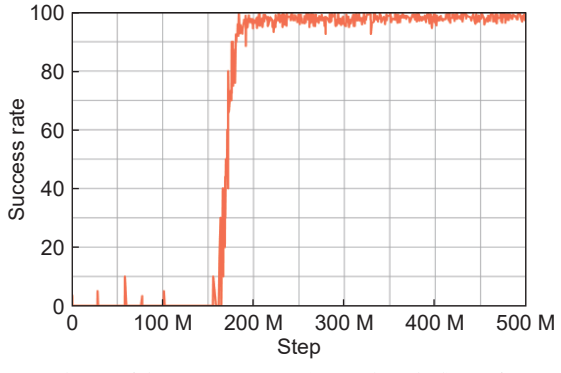


Fig. 13 Evolution of the training success (i.e., goal reached) rate for an agent trained directly at the final difficulty level (Table 5).

complexity, which would be a problem if fast feedback was needed (e.g., because of investigating different hyperparameters on shared/pay-on-demand hardware).

Additionally, for this experiment, we compare our DRL agent with a Hybrid A* implementation (Table 6). Here, Hybrid A* slightly outperforms our DRL agent in terms of the goal reach rate (99% vs. 98%). We argue that this is due to the lower complexity of the maneuvers needed to reach the target. However, reducing the latency from 0.56 to 0.17 s requires increasing the cell side size from 0.5 to 1.2 m, causing a decrease in the success rate to 69%.

Within this experiment, we also conducted a specific study to better assess the agent's generalizability. After concluding the training at the first three levels, we tested the learned policy in environments with more obstacles. The new obstacles are identified with the same logic as for the first three levels but closer to the target (Fig. 11). Table 7 reports the results of over 1,000 test episodes of the agent trained using 3 obstacles in different settings.

The outcomes highlight that the additional difficulty is not properly managed by the agent, with a clear increase in the collision rate with the first added obstacle. We thus extended the training to the cases of 4, 5, and 6 obstacles. The results, provided in Tables 8, 9, and 10, respectively, show that training with the

Table 6 Comparison of 100 episodes between Hybrid A* and DRL PPO in the Random Obstacles environment at the final difficulty level

Path planning algorithm	Success rate	Time for path planning
Proposed DRL	98%	—
Hybrid A*	99%	0.561 s

Table 7 Testing results obtained using different environmental complexities with the agent trained with 3 obstacles

Env.	Goal ratio	Collision ratio	Timeout ratio
3 obstacles	99.3%	0.7%	0%
4 obstacles	66.6%	33.4%	0%
5 obstacles	65.7%	34.2%	0.1%
6 obstacles	60.8%	39.1%	0.1%

Table 8 Testing results obtained using different environmental complexities with the agent trained with 4 obstacles

Env.	Goal ratio	Collision ratio	Timeout ratio
3 obstacles	96.9%	2.4%	0.7%
4 obstacles	98.2%	1.7%	0.1%
5 obstacles	93.7%	6.1%	0.2%
6 obstacles	93.4%	6.1%	0.1%

Table 9 Testing results obtained using different environmental complexities with the agent trained with 5 obstacles

Env.	Goal ratio	Collision ratio	Timeout ratio
3 obstacles	98.3%	1.1%	0.6%
4 obstacles	97.8%	1.5%	0.7%
5 obstacles	98.3%	1.4%	0.3%
6 obstacles	96.3%	3.1%	0.6%

Table 10 Testing results obtained using different environmental complexities with the agent trained with 6 obstacles

Env.	Goal ratio	Collision ratio	Timeout ratio
3 obstacles	98.6%	1.3%	0.1%
4 obstacles	97.6%	2.1%	0.3%
5 obstacles	98.4%	1.5%	0.1%
6 obstacles	97.8%	1.9%	0.3%

fourth obstacle is already almost sufficient for tackling the most complex cases. As anticipated, this fact is reflected in Fig. 12, which shows the evolution of the training. We see a performance drop at approximately 400 M steps, in which the 4-obstacle training starts, but we do not see significant drops later when switching to training with 5 or 6 obstacles. From Tables 8–10, we also see that a policy trained in a more complex environment also works substantially well in simpler environments. In summary, and particularly referring to Fig. 12, given the criticality of the training time, we argue that the agent is (relatively) quickly able to

reach the target (11 M steps); then, as an obstacle is added, the reward abruptly drops, substantially because the agent discovers the concept of an obstacle before the target. The agent learns to avoid a single obstacle in 36.5 M steps. Further addition of two lateral obstacles does not degrade the performance as much as in the case of the first obstacle, but it takes the agent many more steps (363.5 M) to slowly learn how to address the increased complexity. We argue that this is necessary for the agent to generalize the concept of obstacle avoidance. Then, for additional obstacles, the drop is relatively smaller (and ever smaller, with increasing complexity), and a shorter tuning stage is necessary to regain the top performance.

This study provides a quantitative measurement of how to abstract the notion of avoidable obstacles, an agent needs training in an environment with a sufficient number of obstacles. This is akin to the concept of class representativeness in supervised learning datasets but in the more complex RL context, where the dataset is built dynamically through the agent's experience.

6 Usability comparison with other frameworks

In this section, we compare our development experience in Unity with that in two other environments that can be effectively employed for building DRL-based AD solutions, namely, highway-env and CARLA.

Unity is a multi-industry graphic engine that can be flexibly modified in terms of its simulation environment and settings, with a variety of easily pluggable automotive sensors, such as lidars, radars, cameras, and semantic cameras. Additionally, the generation of different observations is straightforward. The graphical user interface makes the interaction pleasant and efficient, without the need to explore the source code. A Unity simulation implies physics computation and rendering, at least when the input to the agent includes a camera, leading to much higher computational demand. This issue (which is common to all environments) can be efficiently handled because Unity offers the possibility of instantiating several environments in a single process (scene) and/or executing several scenes simultaneously, which significantly reduces the training time by exploiting the different cores of the CPU. As a major limitation, the types of selectable NN architectures (and tunable parameters) are very limited (e.g., dense and convolutional NNs, which do not implement state-of-the-art mechanics such as attention). Unity can be interfaced through the ml-agents Python library as an Open AI Gym environment, thus allowing the exploitation of any NN architecture or custom. However, the available documentation is limited, and we are not aware of reports on this topic in the literature.

Highway-env, on the other hand, is an automotive-specific DRL environment that directly supports high-level concepts such as vehicles and lanes. The framework offers several predefined simulation environments representing different driving scenarios, such as driving on a highway, navigating a roundabout, and parking. In addition to the learning agent (i.e., the ego vehicle), such environments can also be easily populated by concurrent nonplayer vehicles (NPVs). The built-in hook to OpenAI's Gym API enables very efficient and fast model training and policy tuning. All environments are indeed rendered in 2D with a bird's-eye view and can be easily interfaced through the Pygame Python library. For cars, highway-env employs the bicycle kinematic model (Polack et al., 2017), a linear acceleration model based on the intelligent driver model (IDM) (Treiber et al., 2000), and lane-changing behavior based on the MOBIL (Kesting et al., 2007) model. The behavioral decisions of the nonego cars are made through heuristics. One of the main advantages of highway-enhanced methods is their simplicity and low computational cost. This makes it a great tool for developing new and benchmarking DRL algorithms without the added complexity of more sophisticated environments (Bellotti et al., 2023). The goal, reward functions, and observation space for the learning agent can be easily set and adjusted at the user's discretion to best fit the problem at hand. Modifying the environment (e.g., the number of lanes, the topology of roads, and intersections) and implementing variations of the tasks (e.g., overtaking, reaching a goal in a narrow space) is possible by operating at the source code level. On the other hand, the realism and variety of scenarios and objects are limited (e.g., there are no pedestrians or motorbikes/bicycles), which are necessary for proper training and testing of a realistic agent.

CARLA is an open-source platform for simulating and testing automated driving systems. It provides a high degree of flexibility in terms of customization and control of the environment's physics, supporting the design of complex scenarios, which is very useful for DRL experimentation. CARLA provides predefined maps and tools (e.g., Scenario Runner (CARLA Scenario Runner, 2023) and map editor (CARLA Map Editor, 2023)) for developing simulations in highly realistic and/or customized settings. It also features an impressive catalog of carefully simulated vehicle models, with parameters including the maximum RPM, the moment of inertia of the vehicle's engine, the gear switch time, and the drag coefficient of the vehicle's chassis. This is also possible thanks to the reliance on the Unreal Engine game engine (Unreal Engine, 2023), which provides high-quality 3D rendering and physics simulations. CARLA uses a modified Unreal 4.26 fork, which contains patches specific to CARLA and requires powerful hardware. Vehicles' dynamic models are based on NVIDIA PhysX (NVIDIA-Omniverse, 2023), which is the standard Unreal Engine 4 vehicle model. The tool's learning curve is quite steep. The overall sophistication and complexity of the environment also have implications for training, which is sensitive to parameters such as the vehicle type, the observations, and, particularly, the timing of the simulation world advancement and the agent decision.

In summary, the choice of the simulation framework depends on the specific needs and goals of the research project. If realism and physical precision are the main priorities, we argue that CARLA would be the ideal choice. Highway-env, on the other hand, is a very convenient option for easy deployment and rapid prototyping and seems particularly suitable as a starting benchmark for novel AD DRL algorithms. Finally, the Unity ML-Agents toolkit may provide an excellent trade-off between the fidelity of VR representations and the efficiency of modeling, training, and simulation.

7 Conclusions and future work

This study investigated the training of an AD DRL agent in two map-less path planning tasks to reach a destination by relying on real-time sensor information only. Although mapless planning is usually employed for local navigation, as opposed to global navigation, our experience has shown that a DRL agent can also cover distances of several tens of meters, avoid randomly placed static objects, and perform a specific maneuver, such as parking. We employed Unity as a simulation environment, which proved to be an excellent tool for efficient prototyping, given its effective

user interface, easy configurability of sensors, overall efficiency of 3D modeling, and efficient RL management due to the parallelization of the scenes. There is a limit on the type of NN directly employable through the UI, which can be overcome through manual coding.

The main contribution of this paper is a systematic and quantitative analysis of the training of a DRL agent for low-speed AD maneuvering in Unity. In particular, our experience highlights some key factors in successful training. First, CL allows the agent to overcome local minima of a complex loss function (Fig. 1 in Bengio (2014)). This technique was necessary for the garage case study and highly beneficial for obstacle avoidance. It is also useful for reducing training time and the use of computational resources. It involved carefully modifying, at various stages, factors such as the type of goal, terminal condition, complexity of the learning environment, and reward function hyperparameters.

Two simulation parameters are particularly important for training: the simulation tick and the decision period. Setting the latter requires a trade-off between the agent's responsiveness and availability for the agent to have enough time to observe the effects of its actions and thus learn from experience, which is key to RL. For the simulation tick (i.e., world update frequency), the trade-off is between the environment's feedback (e.g., for collisions or goal reach) and the simulation time (thus, the training time). In our experiments, we found a good trade-off with the following values: a simulation time equal to 0.02 s and a decision period of 10 ticks. It is important to stress that we also experienced similar (and even more marked) sensitivities in other environments, such as CARLA (Lazzaroni et al., 2024) and highway-env (Bellotti et al., 2023).

The size of the neural network should be decided considering the input and output size; for our experiments, a small NN, with two hidden layers of 256 neurons each, was sufficient to achieve satisfactory results. Careful hyperparameter tuning is key to achieving a working policy. This concerns ML/RL hyperparameters (e.g., learning rate, gamma, batch size) and, particularly, reward function hyperparameters (the coefficients c_1, c_2, and c_3 in Eq. (7), which weight each contribution). Plotting through tensorboard the evolution across episodes of relevant quantities (e.g., the values of single components of the reward function) is very useful for addressing the key difficulty of reward function design, including its hyperparameters. Only an empirical iterative process, performed carefully by observing the outcome of long training simulations, could lead to an effective policy (which has to be defined for every stage of a CL approach). In particular, an accurate balance must be achieved between dense and sparse rewards, as the former tends to infuse expert knowledge into the policy, which may be useful for accelerating training, while the latter considers the final outcome of the agent process, thus giving the agent more freedom to learn his or her own patterns. With this shrewdness, we achieved training that is quite robust and not too long (approximately 12 days on a device equipped with an Intel Xeon W2223 processor, 32 GB of RAM, and an NVIDIA Quadro RTX 4000 GPU). According to our experience, the bottleneck is clearly the CPU because of the prevalence of the simulation computation burden.

Concerning generalization (i.e., knowledge transfer), the agent can reach different targets by avoiding obstacles (walls or other shapes) with random positions (origin, destinations, and placement of the obstacles) and with difficult maneuvers (parking and narrow spaces among obstacles), provided that it is accurately trained to do that (same type of maneuver, training with random positions, a similar configuration/complexity of the environment). The abstraction of concepts such as obstacle avoidance requires training in a sufficiently complex environment (i.e., an environment with a sufficient number of obstacles).

The present work has shown the clear value of Unity as a simulation environment for prototyping DRL agents. This, together with some limitations of our study (particularly on curriculum training and agent generalization support because of the large amount of computational resources needed to explore alternatives, which led us to implement simple case studies), opens significant possibilities for future research in the same or other DRL environments.

We have proven the crucial effectiveness of CL for low-speed AD maneuvering. Future research could explore the automation of the process (e.g., difficulty measurement or training scheduling (Wang et al., 2022b)), which would reduce the time spent investigating and tweaking the numerous hyperparameters involved. Other research directions may involve the use of higher-level AD decision making (Forneris et al., 2023), multiagent systems (Fei et al., 2024; Ryou et al., 2022), and language models for trajectory generation (Kwon et al., 2023; Liu et al., 2023; Qu et al., 2023). Additionally, we limited our study to static obstacles, for the sake of clarity and because this is typical in low-speed maneuvering, but now the analysis could be extended to dynamic contexts, e.g., with moving vehicles and pedestrians, in more realistic contexts (motorways and urban areas), thus involving other types of maneuvers. Finally, transfer learning should be explored in real-world vehicles, which is the final target and poses clear challenges in terms of modeling the behavior of vehicle(s) and sensors.

Replication and data sharing

To support the R&D community in the field, we release our environment and data open source at https://github.com/Elios-Lab/pathfollowing.

Acknowledgements

The authors would like to thank Nicola Poerio, of CRF Stellantis, for his invaluable support on the subject of reinforcement learning for automated driving. Sincere appreciation is also owed to Andrea Alberti and Gianluca Gatti for their considerable efforts toward this research.

Declaration of competing interest

The authors have no competing interests to declare that are relevant to the content of this article.

References

Almón-Manzano, L., Pastor-Vargas, R., Troncoso, J. M. C., 2022. Deep reinforcement learning in Agents' training: Unity ML-agents. In: International Work-Conference on the Interplay Between Natural and Artificial Computation, 391–400.

Lopez, P. A., Wiessner, E., Behrisch, M., Bieker-Walz, L., Erdmann, J., Flotterod, Y. P., et al., 2018. Microscopic Traffic Simulation using SUMO. In: 2018 21st International Conference on Intelligent Transportation Systems (ITSC), 2575–2582.

Anzalone, L., Barra, P., Barra, S., Castiglione, A., Nappi, M., 2022. An end-to-end curriculum learning approach for autonomous driving scenarios. IEEE Trans Intell Transport Syst, 23, 19817–19826.

Bae, J., Kim, T., Lee, W., Shim, I., 2021. Curriculum learning for vehicle lateral stability estimations. IEEE Access, 9, 89249–89262.

Bellotti, F., Lazzaroni, L., Capello, A., Cossu, M., De Gloria, A., Berta, R., 2023. Explaining a deep reinforcement learning (DRL)-based automated driving agent in highway simulations. IEEE Access, 11, 28522–28550.

Bengio, Y., 2014. Evolving culture versus local minima. In: Growing Adaptive Machines: Combining Development and Learning in Artificial Neural Networks, 109–138.

Bengio, Y., Louradour, J., Collobert, R., Weston, J., 2009. Curriculum learning. In: Proceedings of the 26th Annual International Conference on Machine Learning, 41–48.

Boroujeni, Z., Goehring, D., Ulbrich, F., Neumann, D., Rojas, R., 2017. Flexible unit A-star trajectory planning for autonomous vehicles on structured road maps. In: 2017 IEEE International Conference on Vehicular Electronics and Safety (ICVES), 7–12.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., et al., 2016. OpenAI Gym. https://arxiv.org/abs/1606.01540

Buckley, D., 2023. Unity ML-Agents Tutorials. https://gamedevacademy.org/unity-machine-learning-agents-tutorials

CARLA Map Editor, 2023. https://github.com/carla-simulator/carla-map-editor

CARLA Scenario Runner, 2023. https://github.com/carla-simulator/scenario_runner

Chiang, H. T L., Hsu, J., Fiser, M., Tapia, L., Faust, A., 2019. RL-RRT: Kinodynamic motion planning via learning reachability estimators from RL policies. IEEE Robot Autom Lett, 4, 4298–4305.

Chu, Z., Wang, F., Lei, T., Luo, C., 2023. Path planning based on deep reinforcement learning for autonomous underwater vehicles under ocean current disturbance. IEEE Trans Intell Veh, 8, 108–120.

Dai, C., Zong, C., Zhang, D., Li, G., Chuyo, K., Zheng, H., et al., 2023. Human-like lane-changing trajectory planning algorithm for human-machine conflict mitigation. J Intell Connect Veh, 6, 46–63.

Dev, Y., 2023. Chevrolet Corvette 1980 Different colours. https://sketchfab.com/3d-models/chevrolet-corvette-1980-different-colours-7e428bdb3ab54b4e9ac610e545fd9d03

Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., et al., 2017. OpenAI Baselines. https://github.com/openai/baselines

Ding, H., Li, W., Xu, N., Zhang, J., 2022. An enhanced eco-driving strategy based on reinforcement learning for connected electric vehicles: Cooperative velocity and lane-changing control. J Intell Connect Veh, 5, 316–332.

Dolgov, D., Thrun, S., Montemerlo, M., Diebel, J., 2008. Practical search techniques in path planning for autonomous driving. AAAI Work Tech Rep, WS–08–10, 32–37.

Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., Koltun, V., 2017. CARLA: An open urban driving simulator. In: Conference on robot learning, 1–16.

Elios Lab., 2023. Pathfollowing. https://github.com/Elios-Lab/pathfollowing.git

Faust, A., Oslund, K., Ramirez, O., Francis, A., Tapia, L., Fiser, M., et al., 2018. PRM-RL: Long-range robotic navigation tasks by combining reinforcement learning and sampling-based planning. In: 2018 IEEE International Conference on Robotics and Automation (ICRA), 5113–5120.

Fei, Y., Shi, P., Li, Y., Liu, Y., Qu, X., 2024. Formation control of multi-agent systems with actuator saturation via neural-based sliding mode estimators. Knowl Based Syst, 284, 111292.

Florensa, C., Held, D., Wulfmeier, M., Zhang, M., Abbeel, P., 2017. Reverse curriculum generation for reinforcement learning. In: Proceedings of the 1st Annual Conference on Robot Learning, 482–495.

Forneris, L., Pighetti, A., Lazzaroni, L., Bellotti, F., Capello, A., Cossu, M., et al., 2023. Implementing deep reinforcement learning (DRL)-based driving styles for non-player vehicles. Int J Serious Games, 10, 153–170.

Gan, N., Zhang, M., Zhou, B., Chai, T., Wu, X., Bian, Y., 2022. Spatio-temporal heuristic method: A trajectory planning for automatic parking considering obstacle behavior. J Intell Connect Veh, 5, 177–187.

Gao, J., Ye, W., Guo, J., Li, Z., 2020. Deep reinforcement learning for indoor mobile robot path planning. Sensors, 20, 5493.

González Bautista, D., Pérez, J., Milanés, V., Nashashibi, F., 2015. A review of motion planning techniques for automated vehicles. IEEE Trans Intell Transp Syst, 17, 1135–1145.

Goodfellow, I. J., Mirza, M., Xiao, D., Courville, A. Bengio, Y., 2015. An empirical investigation of catastrophic forgetting in gradient-based neural networks. https://doi.org/10.48550/arXiv.1312.6211

Grabusts, P., Musatovs, J., Golenkov, V., 2019. The application of simulated annealing method for optimal route detection between objects. Procedia Comput Sci, 149, 95–101.

Gu, Y., Cheng, Y., Philip Chen, C. L., Wang, X., 2022. Proximal policy optimization with policy feedback. IEEE Trans Syst Man Cybern Syst, 52, 4600–4610.

Hao, K., Zhao, J., Yu, K., Li, C., Wang, C., 2020. Path planning of mobile robots based on a multi-population migration genetic algorithm. Sensors, 20, 5873.

He, Y., Liu, Y., Yang, L., Qu, X., 2023. Deep adaptive control: Deep reinforcement learning-based adaptive vehicle trajectory control algorithms for different risk levels. IEEE Trans Intell Veh, 9, 1654–1666.

Juliani, A., Berges, V.P., Teng, E., Cohen, A., Harper, J., Elion, C., Goy, C., et al., 2020. Unity: A general platform for intelligent agents. https://doi.org/10.48550/arXiv.1809.02627

Kavraki, L. E., Svestka, P., Latombe, J. C., Overmars, M. H., 1996. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. IEEE Trans Robot Automat, 12, 566–580.

Kesting, A., Treiber, M., Helbing, D., 2007. General lane-changing model MOBIL for car-following models. Transp Res Rec, 1999, 86–94.

Khatib, O., 1990. Real-time obstacle avoidance for manipulators and mobile robots. In: Cox IJ, Wilfong GT, Autonomous Robot Vehicles, 396–404.

Kobayashi, M., Motoi, N., 2022. Local path planning: Dynamic window approach with virtual manipulators considering dynamic obstacles. IEEE Access, 10, 17018–17029.

Kwon, T., Di Palo, N. and Johns, E., 2023. Language models as zero-shot trajectory generators. https://doi.org/10.48550/arXiv.2310.11604

LaValle, S. M., Kuffner, Jr., J. J., 2001. Randomized kinodynamic planning. Int J Robot Res, 20, 378–400.

Lazzaroni, L., Bellotti, F., Capello, A., Cossu, M., De Gloria, A., Berta, R., 2022. Deep reinforcement learning for automated car parking. In: International Conference on Applications in Electronics Pervading Industry, 125–130.

Lazzaroni, L., Pighetti, A., Bellotti, F., Capello, A., Cossu, M., Berta, R., 2024. Automated parking in CARLA: A deep reinforcement learning-based approach. In: International Conference on Applications in Electronics Pervading Industry, 352–357.

Lei, X., Zhang, Z., Dong, P., 2018. Dynamic path planning of unknown environment based on deep reinforcement learning. J Robot, 2018, 5781591.

Leurent, E., 2018. An environment for autonomous driving decision-making. https://github.com/Farama-Foundation/HighwayEnv

Ling, Y., Yang, N., Yu, H., Zhu, Y., 2021. Novel Bayesian network incremental learning method based on particle swarm optimization algorithm. In: International Conference on Intelligent and Interactive Systems and Applications, 941–947.

Liu, L. S., Lin, J. F., Yao, J. X., He, D. W., Zheng, J. S., Huang, J., et al., 2021. Path planning for smart car based on dijkstra algorithm and dynamic window approach. Wirel Commun Mob Comput, 2021, 1–12.

Liu, Y., Wu, F., Liu, Z., Wang, K., Wang, F., Qu, X., 2023. Can language models be used for real-world urban-delivery route optimization? Innovation, 4, 100520.

Luo, M., Hou, X., Yang, J., 2020. Surface optimal path planning using an extended dijkstra algorithm. IEEE Access, 8, 147827–147838.

Majumder, A., 2021. *Deep Reinforcement Learning in Unity: With Unity ML Toolkit*. Berkeley, CA: Apress.

Martin, E. Zhai, Y., 2019. Top 5 ways real-time 3D is revolutionizing the automotive product lifecycle. https://unity.com/resources/whitepaper-top-5-use-cases-for-rt3d

Mnih, V., Badia, A.P., Mirza, M., Graves, A., Lillicrap, T.P., Harley, T., Silver, D., et al., 2016. Asynchronous methods for deep reinforcement learning. In: International conference on machine learning, 1928–1937.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., et al., 2015. Human-level control through deep reinforcement learning. Nature, 518, 529–533.

Nobahari, H., Nasrollahi, S., 2019. A terminal guidance algorithm based on ant colony optimization. Comput Electr Eng, 77, 128–146.

Nordeus, E., 2022. Self driving vehicle. https://github.om/Habrador/Self-driving-vehicle

NVIDIA-Omniverse., 2023. NVIDIA PhysX. https://github.com/NVIDIA-Omniverse/PhysX

Peng, T., Liu, X., Fang, R., Zhang, R., Pang, Y., Wang, T., et al., 2020. Lane-change path planning and control method for self-driving articulated trucks. J Intell Connect Veh, 3, 49–66.

Peng, Y., Liu, Y., Zhang, H., 2021. Deep reinforcement learning based path planning for UAV-assisted edge computing networks. In: 2021 IEEE Wireless Communications and Networking Conference (WCNC), 1–6.

Petereit, J., Emter, T., Frey, C.W., Kopfstedt, T., Beutel, A., 2012. Application of hybrid A* to an autonomous mobile robot for path planning in unstructured outdoor environments. In: ROBOTIK 2012; 7th German Conference on Robotics, 1–6.

Pohan, M. A. R., Trilaksono, B. R., Santosa, S. P., Rohman, A. S., 2021. Path planning algorithm using the hybridization of the rapidly-exploring random tree and ant colony systems. IEEE Access, 9, 153599–153615.

Polack, P., Altche, F., d'Andrea-Novel, B., de La Fortelle, A., 2017. The kinematic bicycle model: A consistent model for planning feasible trajectories for autonomous vehicles? In: 2017 IEEE Intelligent Vehicles Symposium (IV), 812–818.

Pothan, S., Nandagopal, J. L., Selvaraj, G., 2017. Path planning using state lattice for autonomous vehicle. In: 2017 International Conference on Technological Advancements in Power and Energy (TAP Energy), 1–5.

Qu, X., Lin, H., Liu, Y., 2023. Envisioning the future of transportation: Inspiration of ChatGPT and large models. Commun Transp Res, 3, 100103.

Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., Dormann, N., 2021. Stable-baselines3: Reliable reinforcement learning implementations. J Mach Learn Res, 22, 1–8.

Raju K., 2020. Autonomous car parking using ML-agents. https://medium.com/xrpractices/autonomous-car-parking-using-ml-agents-d780a366fe46

Ryou, G., Tal, E., Karaman, S., 2022. Cooperative multi-agent trajectory generation with modular Bayesian optimization. https://doi.org/10.48550/arXiv.2206.00726

Schulman, J., Levine, S., Moritz, P., Jordan, M.I., Abbeel, P., 2017a. Trust region policy optimization. In: International conference on machine learning, 1889–1897.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O., 2017b. Proximal policy optimization algorithms. https://doi.org/10.48550/arXiv.1707.06347

Sedighi, S., Nguyen, D. V., Kuhnert, K. D., 2019. Guided hybrid A-star path planning algorithm for valet parking applications. In: 2019 5th International Conference on Control, Automation and Robotics (ICCAR), 570–575.

Silva, F. L., Filgueira da Silva, S., Mazzariol Santiciolli, F., Eckert, J. J., Silva, L. C. A., Dedini, F. G., 2021. Multi-objective optimization of the steering system and fuzzy logic control applied to a car-like robot. In: International Symposium on Multibody Systems and Mechatronics, 195–202.

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., Riedmiller, M., 2014. Deterministic policy gradient algorithms. In: Proceedings of the 31st International Conference on International Conference on Machine Learning-Volume 32, 387–395.

Simonini, T., Sanseviero, O., 2023. The hugging face deep reinforcement learning class. https://github.com/huggingface/deep-rl-class.git

Sun, H., Zhang, W., Yu, R., Zhang, Y., 2021. Motion planning for mobile robots—Focusing on deep reinforcement learning: A systematic review. IEEE Access, 9, 69061–69081.

Sutton, R. S., Barto, A. G., 1998. Reinforcement learning: An introduction. IEEE Trans Neural Netw, 9, 1054.

Szczepanski, R., Tarczewski, T., Erwinski, K., 2022. Energy efficient local path planning algorithm based on predictive artificial potential field. IEEE Access, 10, 39729–39742.

Treiber, M., Hennecke, A., Helbing, D., 2000. Congested traffic states in empirical observations and microscopic simulations. Phys Rev E Stat Phys Plasmas Fluids Relat Interdiscip Topics, 62, 1805–1824.

Unity, 2023. Unity. https://unity.com

Unreal Engine, 2023. Unreal Engine. https://www.unrealengine.com/en-US

Urmanov, M., Alimanova, M., Nurkey, A., 2019. Training unity machine learning agents using reinforcement learning method. In: 2019 15th International Conference on Electronics, Computer and Computation (ICECCO), 1–4.

Van Hasselt, H., Guez, A., Silver, D., 2016. Deep reinforcement learning with double Q-learning. Proc AAAI Conf Artif Intell, 30, 1.

Wang, D., Ha, M., Qiao, J., 2020. Self-learning optimal regulation for discrete-time nonlinear systems under event-driven formulation. IEEE Trans Automat Contr, 65, 1272–1279.

Wang, D., He, H., Liu, D., 2017. Adaptive critic nonlinear robust control: A survey. IEEE Trans Cybern, 47, 3429–3451.

Wang, P., Yang, J., Zhang, Y., Wang, Q., Sun, B., Guo, D., 2022a. Obstacle-avoidance path-planning algorithm for autonomous vehicles based on B-spline algorithm. World Electr Veh J, 13, 233.

Wang, X., Chen, Y. Zhu, W., 2022b. A survey on curriculum learning. IEEE Trans Pattern Anal Mach Intell, 44, 4555–4576.

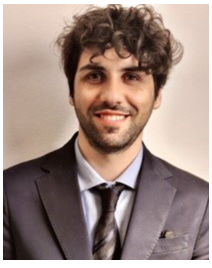
Wei, Y., Zhang, H., Wang, Y., Huang, C., 2023. Autonomous maneuver decision-making through curriculum learning and reinforcement learning with sparse rewards. IEEE Access, 11, 73543–73555.

Zhang, K., Niroui, F., Ficocelli, M., Nejat, G., 2018. Robot Navigation of Environments with Unknown Rough Terrain Using deep Reinforcement Learning. In: 2018 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR), 1–7.

Zhao, J., Ma, X., Yang, B., Chen, Y., Zhou, Z., Xiao, P., 2022. Global path planning of unmanned vehicle based on fusion of A* algorithm and Voronoi field. J Intell Connect Veh, 5, 250–259.

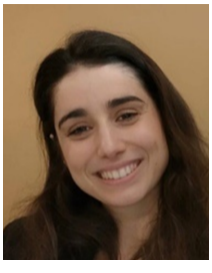
Zheng, L., Zeng, P., Yang, W., Li, Y., Zhan, Z., 2020. Bézier curve-based trajectory planning for autonomous vehicles with collision avoidance. IET Intell Transp Syst, 14, 1882–1891.

Riccardo Berta is an Associate Professor with the ELIOS Laboratory, DITEN, University of Genoa. He has authored approximately 120 papers in international journals and conferences. His main research interests include applications of electronic systems, in particular, in the fields of machine learning, the Internet of Things (IoT), and serious gaming. He is a founding member of the Serious Games Society. He is a Publications Chair of the International Conference Series GALA (Game and Learning Alliance) and the Applications in Electronics Pervading Industry, Environment and Society (ApplePies) International Conference. He is an Associate Editor of the International Journal of Serious Games.



Luca Lazzaroni received the B.Sc. degree in electronic engineering and information technologies, the M.Sc. degree in electronic engineering, and the Ph.D.degree in science and technology for electronic and telecommunication engineering from the University of Genoa, Genoa, Italy, in 2017, 2020, and 2024, respectively, where he is currently a Research Fellow with the ELIOS Laboratory, Department of Electrical, Electronic, Telecommunication Engineering and Naval Architecture. He has authored approximately 30 papers in international journals and conferences. His research interests include deep learning, edge computing, and automated driving.

Alessio Capello received the B.Sc. degree in electronic engineering and information technologies and the M.Sc. degree (cum laude) in electronic engineering from the University of Genoa, Genoa, Italy, in 2017 and 2021, respectively, where he is currently pursuing the Ph.D. degree in science and technology for electronic and telecommunication engineering (STIET), ELIOS Laboratory, Department of Electrical, Electronic, Telecommunication Engineering and Naval Architecture. His main research interests include big data management and deep learning.



Marianna Cossu received the B.S. degree in electronic engineering and information technologies, in 2019, and the M.S. degree in electronic engineering, in 2021. She is currently pursuing the Ph.D. degree in science and technology electronic and telecommunication engineering (STIET) with the ELIOS Laboratory, DITEN, University of Genoa. Her main research interests include machine learning, deep learning, and automated driving.

Luca Forneris is a Ph.D. student at ELIOS Lab, Department of Electrical, Electronic, Telecommunication Engineering and Naval Architecture, University of Genoa, Genoa, Italy. He obtained the B.Sc. degree in computer science in 2019 and the M.Sc. degree (cum laude) in computer science – data science & engineering: artificial intelligence in 2022 from the University of Genoa. He is pursuing the Ph.D. degree in Joint Doctoral Programme in Interactive and Cognitive Environments (JD-ICE) in collaboration with the University of Genoa (UniGe) and the Queen Mary University of London (QMUL). His research activities include deep learning, computer vision and autonomous driving.

Alessandro Pighetti is a Ph.D. student at ELIOS Lab, Department of Electrical, Electronic, Telecommunication Engineering and Naval Architecture, University of Genoa, Genoa, Italy. He obtained the B.Sc. degree in computer science in 2019 and the M.Sc. degree in computer science – data science & engineering: artificial intelligence in 2022 from the University of Genoa. He is enrolled in a Joint Doctoral Programme in Interactive and Cognitive Environments (JD-ICE) in collaboration with the University of Genoa (UniGe) and the Queen Mary University of London (QMUL). His research activities include deep learning, serious games, and autonomous driving.

Francesco Bellotti received the M.Sc. degree (cum laude) in electronic engineering and the Ph.D. degree in electrical engineering from the University of Genoa, in 1997 and 2001, respectively. He is an Associate Professor with DITEN, University of Genoa, and currently teaching computational intelligence and cyber-physical systems. He seats in the Didactic Board of the international joint Ph.D. program in Interactive Cognitive Environments. He has coauthored more than 250 scientific publications in journals, international books, and conference proceedings. He has been responsible for the design and implementation WPs of several European and Italian industrial research projects, particularly in the field of intelligent transportation systems. His main research interests include automated driving, machine learning, cyber-physical systems, edge computing, and human–computer interaction.