RL-Studio: a tool for Reinforcement Learning Methods in Robotics

Pedro Fernández de Cabo, Rubén Lucas, Nacho Arranz, Sergio Paniego, and José M. Cañas

Universidad Rey Juan Carlos, Madrid, Spain,
p.fernandezd.2016@alumnos.urjc.es, r.lucasz@alumnos.urjc.es,
n.arranz.agueda@gmail.com, sergio.paniego@urjc.es, josemaria.plaza@urjc.es

Abstract. This paper introduces *RL-Studio*, an open-source software that eases the implementation of reinforcement learning algorithms to solve a problem. This software enables the integration of any simulator to recreate the problem where a reinforcement learning algorithm can be applied. Some relevant advantages of this tool are the generalization of common software components and the unified architecture. RL-Studio permits to just focus on fine tuning the hyperparameters of the algorithm and redefine the goal and reward function of the previously integrated projects. In case a non integrated scenario, algorithm or simulator is needed, RL-Studio also provides some already integrated libraries that can be reused to save time. It has been experimentally validated in research projects and some of the canonical problems such as Robot Mesh or Mountain Car.

Keywords: Reinforcement learning, software tools, open source, ROS, Gazebo, robotics, autonomous vehicles

1 Introduction

The use of autonomous agents has become popular in recent years. It is already common to find robots in extensive areas of society such as logistics automation and warehouse robots, self-driving cars and aerial unmanned vehicles and even home vacuum robots among many examples. It is now possible to travel in driverless vehicles of the Waymo company in San Francisco or acquire home cleaning robots such as Roomba® or Braava®. Traditionally the solution has been built through modular handcrafted pieces of hardware and software, where each one has been designed to solve a minimal case, and altogether has been able to obtain the solution. But autonomous agents are designed to help and even replace human beings in tasks carried out in multidimensional fields. Therefore, the complexity of agent control systems has increased in such a way that it is very difficult to design them with those explicit programming methods. Traditional methods do not scale well nor only in other fields, but even in their application domain. Deep learning (DL) as a function approximator and feature learning has demonstrated its excellence in finding low-dimensional patterns in multidimensional data so has made possible to face real problems in complex environments.

For this reason, DL is used extensively in visual object detection and perception. Supervised and semi-supervised learning methods, using these DL techniques, process labeled datasets to find similar patterns in the input data. Even with no enough labeled data, data augmentation can help to generate synthetic data that help neural networks achieve good performance. DL has demonstrated robustness in end-to-end solutions based on the direct understanding of the input information for delivering a complete and functional result. There are real business solutions implemented like Tesla Autopilot, in garbage collection to identify containers, in waste recycling or even in garment recognition. However, DL needs tons of labeled data and with these techniques alone it is not easy to guarantee that a neural net understands the real complex environments. It is also necessary to learn to simulate complex scenarios where there could be dangerous events. A dataset which undersamples these important cases might yield a classifier which fails on them. Reinforcement learning (RL) is based on behavioral psychology where the agent learns by interacting in an environment through trial and error seeking to maximize reward and traditionally has been used successfully in control tasks. In this type of problems an agent, a car or robot, is interacting in an environment, such as a road, a city, or a room, at time-step to obtain a reward. The agent has to learn which are the best decisions and actions at each moment to learn the best strategy, the policy, that leads to optimizing the rewards in the long term. The reward is one of the most important distinctions in RL, since the objective sought is the maximization of the expected values of the cumulative sum of a signal. The design of the reward function in an RL algorithm is a mixture of art and science, achieved through testing hard work. Unlike the previous methods, RL algorithms do not need massive training data to learn. The agent receives the information just having contact with the environment in which it operates. But one of the main issues of robotic learning is to collect a large amount of data autonomously and safely. Modern simulation tools and RL methods, such as addressing partial observations, domain-randomization or domain adaptation help reducing the reality gap in sim-to-real transfer scenarios. In RL methods we let the agent find the best solution so that it can be generalized to similar domains without having to program explicitly. But creating a good policy representation is not a trivial problem, and now other kinds of complex issues appear to solve as sample efficiency, credit assignment, exploration vs. exploitation, or representation. To help in domains with high-dimensional states and continuous actions, the use of DL algorithms, as the representation learning part, within RL framework has created the new field of Deep Reinforcement Learning (DRL). Developing RL models involves different steps. One is to select the agent with the appropriate characteristics and sensors to correctly read the information from the environment. Another is to choose the RL algorithm that suits the goal. There are dozens and each of them has its own parameters that must be adjusted. Additionally, each simulator has features that make it suitable for a specific problem, and finally, the work process requires continuous trial and error testing, evaluating the training results, tweaking parameters, or even changing the design of certain components.

The Reinforcement Learning Studio (RL-Studio) tool, the main contribution of this article, is designed with the aim of facilitating the creation of RL models. It is modular, it allows working with different agents, in any environment, and with any RL algorithm. Any type of sensor suitable for collecting information can be designed and allows to integrate any simulation tool. All these features allow research experimental validation. RL-Studio toolkit is open-source and easily downloaded[1].

2 Related Works

In this section we provide a brief perspective of the state-of-art algorithms, methods and tools in RL. Regarding algorithms, there are two main taxonomies in classical RL, *sample-based learning methods* or also denominated *value-based algorithms* and *approximate solution methods* or *policy-based algorithms*. The former is used in actions and states in low-dimensionality problems and inside this category we can find dynamic programming, Monte-Carlo methods, and temporal-difference methods. The latter can work with real scenarios with continuous actions and states space and the solution they provide is only approximate. Policy gradients are the most popular family of methods [7]. DRL algorithms fall in both categories and we discard other methods such as *imitation-based algorithms* where the agents try to mimic expert actions.

2.1 RL Software Frameworks

There are several frameworks that allow learning RL models without having to program from scratch, although some have different characteristics. Basically, frameworks are modular, cross-platform and ease of running with many pre-implemented RL algorithms allowing easy reproduction of training and results, making it painless to understand and to prototype new research ideas. *OpenAI baselines*, supported by OpenAI company, allows working with different state-of-the-art RL algorithms. It was design specifically for RL research providing built-in simulated environments such as Atari games, board games, 2D and 3D physical simulations, so it allows easily train agents, develop new RL algorithms and compare them. *Keras-rl* based on Keras DL framework and developed by Google, with easy integration with Tensorflow and OpenAI environments. *Ray lib* is an open-source project developed at UC Berkeley that, in addition to having the preceding characteristics, allows building distributed applications in large infrastructures and has been included in the major cloud providers. *ReAgent* by Meta company and thus supports an easy integration with Pytorch. *Acme* developed by DeepMind focus on as a starting block for novel researchers and designed to run single-stream or distributed agents.

[1] https://github.com/JdeRobot/RL-Studio

2.2 Simulation Tools

Planning and control strategies in real robots are often expensive, time-consuming and sometimes dangerous. So, powerful and realistic dynamic simulation tools have been developed improving Sim2Real practice. The next simulators have been chosen considering their level of maturity, modularity, and popularity among engineers and researchers in RL. All of them allow flexible level of control and can run in a fast and distributed manner in order to provide the iteration time required for experimental research. *Carla* is a domain-specific platform and has been built for the development of algorithms for autonomous cars. It includes very real scenarios to test results thus it is widely used in the RL research community. *Mujoco* is a free and open source physics engine designed for the purpose of model-based optimization recently acquired by DeepMind. It has been developed with mind in optimization through contacts. *Unity ML-Agents Toolkit* is a real-time 3D development general platform based in Unity game engine, able to perform sensory, physical and task logic complexity. *Gazebo* is a simulation environment with a built-in physics engine called Ignition. Ignition allows you to simulate the robot with realistic physics. Since Gazebo integrates very well with Robotic Operating System (ROS) which is the main robotics framework, it has been widely adopted in robotics community.

2.3 RL in Robotics

In recent years, RL has achieved human-level control in fields as diverse as video games, Go game, financial industry, traffic and smart cities, education system and health as some examples. In robotics the advantages of RL in online adaptability and self-learning features of complex systems have developed considerable research in learning simple manipulation tasks such as opening doors and stacking Lego blocks, or in to complete object rearrangement tasks such as grasping an apple and putting it on a plate, moving object flexible grasping [8], pushing objects to desired locations [6], visuomotor control [5], object pushing and trajectory following [12], for satisfying relative object positioning tasks [10], and for cloth manipulation such as folding different fabric types [1]. In legged locomotion tasks, RL models are a good promise to design high performance controllers for legged robots [4] and applications in which the robot can learn walking and turning from scratch in the real world [11]. In navigation, RL methods enable mobile robots to self-explore and self-learn by interacting with the dynamic environment. While in traditional programming every tasks involved could be carefully design, such as environment perception, localization, pose estimations, obstacle avoidance and path planning, with RL techniques have been possible build accurate end-to-end models [13] In global and local path planning, RL algorithms have achieved improvements in accuracy, stability and robustness [9]. In Trajectory and route tracking, where the robots track in a partially observable nonlinear dynamic environment, RL methods have improved the real time adaptability of obstacle avoidance and navigation by trajectory and route tracking techniques [3]. And finally, in autonomous vehicles RL methods have had

remarkable success in end-to-end models ending in commercial proposals such as [2].

3 RL-Studio tool

RL-Studio is a modular open-source multi-platform command line software application supported in Linux and Mac OS and developed in Python language. It currently works with Gazebo and OpenAi Gym simulators, although Carla will be available shortly. It enables the flexibility of configuring the agents, algorithms, environments and simulation parameters by just modifying the configuration file used at launch time. It enables two main use cases: training and inference mode. In training mode, RL-Studio permits to develop robotic brains that best adapt to a specific domain in stochastic environments. In inference mode, RL-Studio allows to choose any of the trained robotics brains and test them in the same or any other different environments.

3.1 Software Architecture

RL-Studio has been designed with four main logic components in mind. ***Agent*** contains the definition of the training and inference agents and the necessary software utilities for execution. ***RL algorithm*** where algorithms are found. ***Env*** holds the environment features and simulation tools applied in every project. Based on OpenAI Gym, it acts as an information processing engine. And finally in ***Simulator*** the agent features, sensors and actuators to be launched by the simulator are found. The simulator processes the dynamics physics knowledge of the environment simulating the real world. The information flowing through the nodes is managed by ROS, a standard robotic communication framework, that acts as middleware between agents and environment.

To build a new project from scratch, the following classes must be implemented: *Environment class*, located in the *Env* package, interfaces the simulator sensors and actuators to both execute the actions indicated by the agent and to provide the reward and next state using the *step* and *reset* methods. *Algorithm class*, located in the *algorithm* package, is in charge of learning from the experience to finally select the action which maximizes the reward and leads the agent to its objective. *Agent class*, located in the *agents* package as the core class of the project, is in charge of building the simulation according to the models and worlds configurations, loading the trained algorithm in inference mode, calling the environment class methods to execute the selected actions, feeding the algorithm with the resulting rewards, saving the trained algorithm in training mode and tracking the performance making use of the logging and monitoring libraries.

In RL-Studio operating cycle, each agent acts as an independent node in ROS, obtaining the data through the onboard sensors, e.g. a camera, and publishing it through the ROS topics. The RL-Studio environment subscribes to the agent topics and receives the data that is used to generate the necessary information

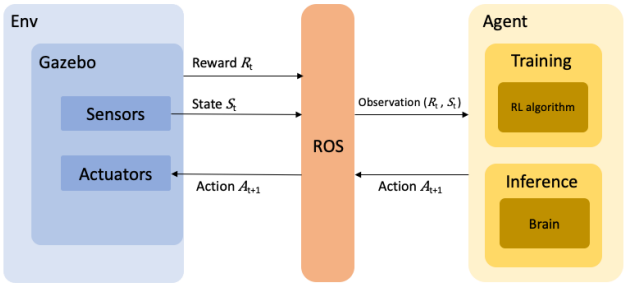


Fig. 1: RL-Studio satisfies the standard RL cycle. ROS handles information between nodes through topics. Highlight on the right, an agent can actuate in training mode where the agent executes the corresponding RL algorithm. Or in inference mode, the already trained brain is executed.

that it sends back to the agent actuators, also through the ROS topics. Therefore, RL-Studio satisfies the standard RL cycle where the environment generates the state and the reward at each time step and the agent gives back the corresponding action as shown in fig. 1. RL-Studio currently supports the simulator Gazebo and the algorithms Q-learning and DQN.

3.2 Configuration

The configuration validation has been generalized to just focus on adding, removing or modifying parameters that will affect our agents both in learning or inferencing mode. These configuration parameters have to be provided in a yaml file, where the agent, algorithm, environment, simulation tool and execution mode, training or inference, are setting. RL-Studio builds the scenario and starts its execution, and for that it needs some of the following input parameters: 1. Agent: width, height, weight, sensors, etc. 2. Actions: number of actions and values 3. Environment: start pose, world specifications, maximum steps per iteration... 4. Algorithm hyper-parameters: learning rate, discount factor, future rewards relevance... 5. Training configuration: save model flag, logs folder, logs level... 6. Inference configuration : inferences file name and actions file name. Depending on the mode input parameter, RL-Studio runs in training or inference mode.

3.3 Training and Inference mode

Training mode allows to create reinforcement learning models that will be saved for later use. In the training mode execution flow, first RL-Studio loads the training configuration settings and validates the integrity. Then, it initializes the project implementation and starts the training: the simulator engine and the configured training algorithm are launched, the configured environment is built from an existing or newly created implementation and monitoring utilities

are called. The environment, as an OpenAI Gym based, plays an important role in this mode, and it is used as a library to interact with the simulated world. It implements the reset and steps execution behaviour which retrieves the state the agent can read from sensors and the rewards obtained in every state.

The goal of inference mode is to test whether robotic brains can generalize to other environments. Therefore, the execution flow is similar to the training mode, except that the trained brain to be tested must be included as a parameter in the configuration file.

3.4 Logging and Monitoring

Every training carries out with RL-Studio is registered in the desired location through the configuration file. Directories are automatically created, in case they do not exist, to save the trained brains as well as the log files necessary to analyze training or inference execution. Also some error messages and common helpful information have been generalized to be imported in any integrated project. Some libraries have been implemented to monitor the evolution of training in the RL-Studio projects: LivePlot library is based on matplotlib[2] and renders a graph of either episode rewards or episode lengths. Stats_recorder stores in json format the episode lengths, episode rewards, episode types and their timestamps. Video_recorder optionally renders a movie of a rollout.

4 Experiments

RL-Studio has been validated building and collaboratively evolving some of the following research projects.

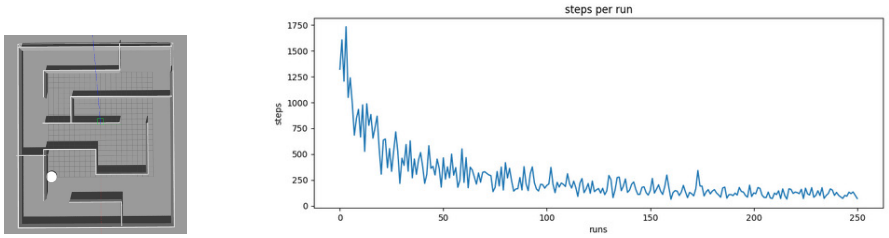
4.1 Robot Mesh

The problem to solve is to make an agent learn the optimal path to the goal as soon as possible inside a mesh. The provided environment permits performing one of four actions, ahead, back, left and right, and each simulation step tried to get closer to the location/state in the board which was configured as goal. This problem has been validated in RL-Studio shown in this video[3] with Q-learning algorithm based on the hyperparameters obtained after several tests, $\gamma = 0.95$, $\alpha = 0.9$, $exploration\ min = 0.00001$ and $exploration\ decay = 0.999$, a sparse reward function with 1 when the goal is reached and 0 otherwise and the input sensory information has been the position of the agent, x and y coordinates, running in Gazebo simulation tool 2a.

As we can see in the figure 2b, the agent learned the optimal path to the goal in approximately 100 iterations. In this problem, RL-studio has been especially useful to easily training and validating the inference mode.

[2] https://matplotlib.org/

[3] https://www.youtube.com/watch?v=36Zko9hSY6k

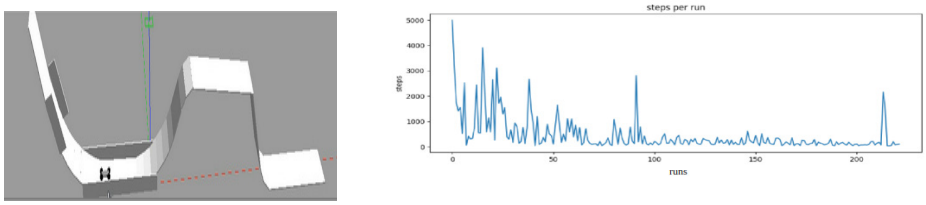


(a) (b)

Fig. 2: (a) Robot Mesh in RL-Studio using Gazebo simulator. (b) Training steps per iteration, showing model stabilization after 100 episodes.

4.2 Mountain Car

The objective has been to teach a car how to get out of the valley and reach the peak of a mountain, fig. 3a, in less than 300 steps just applying one action, force to right, to left or do nothing, every 0.5 seconds. It has been needed to get farther from the goal to be able to reach it. Thus, defining the problem, action and states, and the hyper-parameters configuration has been more complicated so the agent could learn either the optimal or a feasible sequence of steps to accomplish the objective. In this case we have solved the problem using Gazebo, Q-learning algorithm, the state space defined by discretized vehicle coordinates received by the GPS-simulated input sensor, $x = [-10, 10]$, and the discretized linear velocity, $v = [-100, 100]$. The reward function is 1 when the goal is reached and 0 otherwise and the best hyper-parameters configuration tested has been $\gamma = 0.9$, $\alpha = 0.2$, $exploration\ min = 0.01$ and $exploration\ decay = 0.99995$



(a) (b)

Fig. 3: (a) Mountain-car in RL-Studio using Gazebo simulator. (b) Training steps per iteration, starting model stabilization after 100 episodes.

As we can see in the figure 3b, the agent considerably improved its performance as it learns. In this problem, RL-Studio has been especially useful to build the problem. Thanks to the easy way of configuring the actions to be applied, the agent to be used and the monitoring libraries already integrated, we could

perform multiple tests to solve the canonical mountain car problem as it can see in this video[4].

4.3 Formula1 Visual Follow-Line

The agent is based on a real Formula1 (F1) car and it is endowed with a built-in color camera as input sensory data for reading the information retrieved by environment. The environments designed are several circuits with different characteristics of length and curves as can be seen in fig. 4. They are inspired by real F1 circuits and each one has different light conditions, road textures, size straights and pronounced curves what helps us understand the learning generalization capabilities of the model[5].

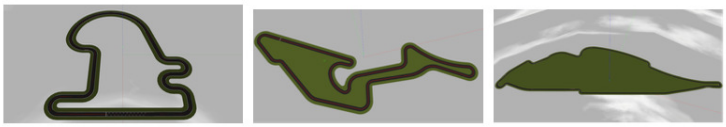
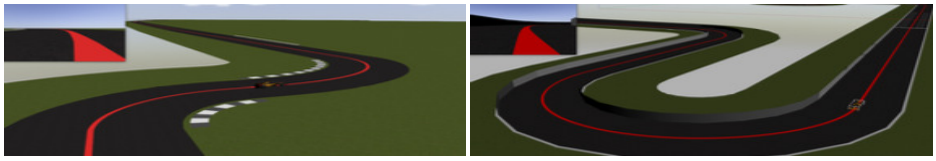


Fig. 4: Some of the circuits in RL-Studio. From left to right: Simple circuit, Nürburgring and Montreal.

The robot goal has been to follow the center red line of the road as closely as possible and without leaving the road as shown in Fig. 5. The environment retrieved a reward that tells the agent how good is the action taken, following the standard RL cycle.



(a) Simple circuit (b) Montreal circuit

Fig. 5: (a) F1 circulating in the Simple circuit and taking a curve to the left from two perspectives. (b) The same F1 agent in Montreal circuit.

In this project Q-learning has been used as RL algorithm and Gazebo tool as a simulator. Q-learn is a tabular RL algorithm admitting a space of actions and states of low dimensionality. Therefore, the raw input sensory data must have been processed to obtain the appropriate information, simplifying it to a

[4]https://www.youtube.com/watch?v=KZjDe6N-d0k
[5]https://www.youtube.com/watch?v=3jdxZTjPCss

set of positions that indicates how close or far the car is from the center of the red line. Every input frame is separated in 17 vertical columns, from -8 to 8 including 0, each of them representing a position or state. In this project, the F1 has been trained with 1 point of perception, so the state will be between -8 to 8. Knowing only the lower half of the image captured by the camera sensor contains the road information, and after various tests, we take as a perception point a distance of 60 pixels below the middle of the image, fig. 6. So, the distance from the center of the image to the center red line of the road will give us the state of the car at each time step. In RL-Studio is very easy to configure different points of perception.

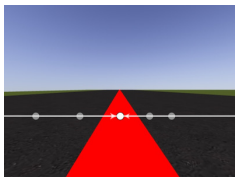


Fig. 6: Simplified perception with one point

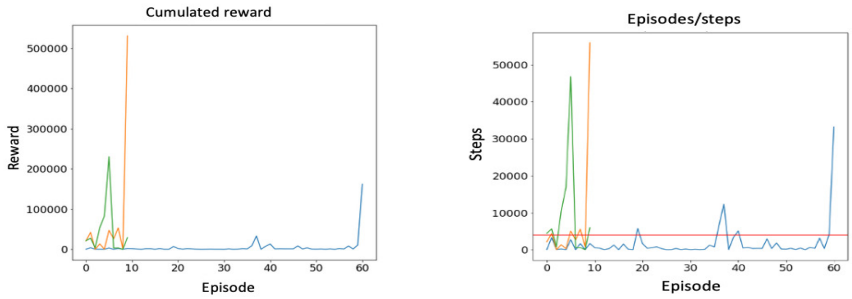
Likewise, we have created a set of discrete actions defined as a vector $a = [v, w]$ which contains the two degrees of freedom that a planar vehicle needs, linear and angular velocity in m/sec and rad/sec respectively. So, we have defined three actions, *action* $A = [3, 0]$, *action* $B = [2, 1]$ and *action* $C = [2, -1]$. Training with low speeds allows us have not taking into account the kinematics and dynamics of the vehicle for the model.

The below defined reward function R, maximizes the value being close to the center, c, of the road line, decreasing as it moves further away. A high penalty has been defined in case the car goes off the track to help make learning faster. Remember that F1 does not have the stop action defined to prevent unwanted behavior.

$$R = \begin{cases} 10 & c < \mid 0.2 \mid \\ 2 & \mid 0.4 \mid > c > \mid 0.2 \mid \\ 1 & \mid 0.9 \mid > c > \mid 0.4 \mid \\ -100 & c > \mid 0.9 \mid \end{cases}$$

Three training have been executed in the Simple circuit for two hours each one and 4000 steps in every episode, green, yellow and blue in fig. 7a, with hyperparameters optimized to $\gamma = 0.9$, $\alpha = 0.8$ and *exploration decay* $= 0.998$. In all of them, the agent has been able to complete the circuit and in two of the three trials, green and yellow, converge very quickly in just less than 10 epochs and accumulating high reward. Namely it spends more time in regions nearby the center of the line where the reward returns the highest values. Likewise, the number of steps taken in every training is showing in fig. 7b.

Table 1 contains more information about the three experiments. We can see the algorithm has learned by improving in each one of the episodes not only in



(a) Cumulated reward in three Simple circuit training

(b) Number of steps in every training in Simple circuit training

Fig. 7: Results of the same three training sessions in the Simple circuit, with a simplified perception point and three discretized actions.

completed lap time, but in smaller number of episodes and in less exploration time given by the variable ϵ, where an ϵ nearby to 1 tells the exploration time is low. *Time per lap (min.)* shows time in minutes to complete the first lap. Total epochs shows number of episodes carried out in two hours.

Table 1: Three different training in Simple circuit with one point of perception and the three defined actions A, B and C.

Simple Circuit			
Training	1	2	3
Time per lap (min.)	20:06	6:53	4:26
Epochs to complete	20	2	1
Total epochs	60	10	10
ϵ	0.8	0.84	0.85
Circuit completed	100%	100%	100%

We have taken the *time per lap* as a quantitative metric to know how the algorithm has learned and the *percentage of circuit completed*, as a qualitative metric, has allowed to know if the agent has completed the circuit correctly. In this last case as all training converged, the values have been 100%.

5 Conclusions

In this paper we have presented a tool to build and test RL models applied to robotics. The proposed architecture, modular-based, has been validated in three different robotics projects where it has demonstrated its robustness, effectiveness and usability. It allows indirect programming with RL based methods in such a way that defining the space of states and actions, robots can find the policy that

maximizes the sought objective. RL-Studio covers the cycle of any standard RL model and is easily configurable for training and inference, allowing to configure all the pieces that make up an RL model. As future work, we plan to integrate other simulators used in the scientific robotics community like Carla, along with more state-of-the-art algorithms in RL such as DDPG and PPO which allow to work with real continuous spaces. We will also take advantage of its modularity along with Docker to create end-to-end solutions with hardly any changes in the configuration.

References

1. Hietala, J., Blanco-Mulero, D., Alcan, G., Kyrki, V.: Learning visual feedback control for dynamic cloth folding. arXiv preprint arXiv:2109.04771 (2021)
2. Kendall, A., Hawke, J., Janz, D., Mazur, P., Reda, D., Allen, J.M., Lam, V.D., Bewley, A., Shah, A.: Learning to drive in a day. In: 2019 International Conference on Robotics and Automation (ICRA), pp. 8248–8254. IEEE (2019)
3. Kong, X., Xia, Y., Hu, R., Lin, M., Sun, Z., Dai, L.: Trajectory tracking control for under-actuated hovercraft using differential flatness and reinforcement learning-based active disturbance rejection control. Journal of Systems Science and Complexity **35**(2), 502–521 (2022)
4. Lee, J., Hwangbo, J., Wellhausen, L., Koltun, V., Hutter, M.: Learning quadrupedal locomotion over challenging terrain. Science robotics **5**(47), eabc5986 (2020)
5. Lee, J.W., Kim, K.W., Shin, S.H., Kim, S.W.: Vision-based collision avoidance for mobile robots through sim-to-real transfer. In: 2022 International Conference on Electronics, Information, and Communication (ICEIC), pp. 1–4. IEEE (2022)
6. Serhan, B., Pandya, H., Kucukyilmaz, A., Neumann, G.: Push-to-see: Learning non-prehensile manipulation to enhance instance segmentation via deep q-learning. Institute of Electrical and Electronics Engineers (2022)
7. Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction. MIT press (2018)
8. Tu, Z., Yang, C., Wu, X., Zhu, Y., Wu, W., Jia, N.: Moving object flexible grasping based on deep reinforcement learning. In: 2022 8th International Conference on Control, Automation and Robotics (ICCAR), pp. 34–39. IEEE (2022)
9. Wang, B., Liu, Z., Li, Q., Prorok, A.: Mobile robot path planning in dynamic environments through globally guided reinforcement learning. IEEE Robotics and Automation Letters **5**(4), 6932–6939 (2020)
10. Xie, A., Singh, A., Levine, S., Finn, C.: Few-shot goal inference for visuomotor learning and planning. In: Conference on Robot Learning, pp. 40–52. PMLR (2018)
11. Yang, Y., Caluwaerts, K., Iscen, A., Zhang, T., Tan, J., Sindhwani, V.: Data efficient reinforcement learning for legged robots. In: Conference on Robot Learning, pp. 1–10. PMLR (2020)
12. Yen-Chen, L., Bauza, M., Isola, P.: Experience-embedded visual foresight. In: Conference on Robot Learning, pp. 1015–1024. PMLR (2020)
13. Zhou, Q., Lyu, L., Liu, H.: Deep reinforcement learning with long-time memory capability for robot mapless navigation. In: 2022 IEEE 25th International Conference on Computer Supported Cooperative Work in Design (CSCWD), pp. 1215–1220. IEEE (2022)