Deepbots: A Webots-Based Deep Reinforcement Learning Framework for Robotics

M. Kirtas, K. Tsampazis, N. Passalis(✉), and A. Tefas

Artificial Intelligence and Information Analysis Lab, Department of Informatics, Aristotle University of Thessaloniki, Thessaloniki, Greece
{eakirtas,tsampaka,passalis,tefas}@csd.auth.gr

Abstract. Deep Reinforcement Learning (DRL) is increasingly used to train robots to perform complex and delicate tasks, while the development of realistic simulators contributes to the acceleration of research on DRL for robotics. However, it is still not straightforward to employ such simulators in the typical DRL pipeline, since their steep learning curve and the enormous amount of development required to interface with DRL methods significantly restrict their use by researchers. To overcome these limitations, in this work we present an open-source framework that combines an established interface used by DRL researchers, the OpenAI Gym interface, with the state-of-the-art Webots robot simulator in order to provide a standardized way to employ DRL in various robotics scenarios. Deepbots aims to enable researchers to easily develop DRL methods in Webots by handling all the low-level details and reducing the required development effort. The effectiveness of the proposed framework is demonstrated through code examples, as well as using three use cases of varying difficulty.

Keywords: Deep Reinforcement Learning · Simulation environment · Webots · Deepbots

1 Introduction

Reinforcement Learning (RL) is a domain of Machine Learning, and one of the three basic paradigms alongside supervised and unsupervised learning. RL employs agents that *learn* by simultaneously *exploring* their environment and *exploiting* the already acquired knowledge to solve the task at hand. The learning process is guided by a reward function, which typically expresses how close the agent is to reaching the desired target behavior. In recent years, Deep Learning (DL) [8] was further combined with RL to form the field of Deep Reinforcement Learning (DRL) [17], where powerful DL models were used to solve challenging RL problems.

DRL is also increasingly used to train robots to perform complex and delicate tasks. Despite the potential of DRL on robotics, such approaches usually

Published by Springer Nature Switzerland AG 2020
I. Maglogiannis et al. (Eds.): AIAI 2020, IFIP AICT 584, pp. 64–75, 2020.
https://doi.org/10.1007/978-3-030-49186-4_6

require an enormous amount of time to sufficiently explore the environment and manage to solve the task, often suffering from low sample efficiency [20]. Furthermore, during the initial stages of training, the agents take actions at random, potentially endangering the robot's hardware. To circumvent these restrictions, researchers usually first run training sessions on realistic simulators, such as Gazebo [11], and OpenRAVE [4], where the simulation can run at accelerated speeds and with no danger, only later to transfer the trained agents on physical robots. However, this poses additional challenges [5], due to the fact that simulated environments provide a varying degree of realism, so it is not always possible for the agent to observe and act exactly as it did during training in the real world. This led to the development of more realistic simulators, which further reduce the gap between the simulation and the real world, such as Webots [15], and Actin [1]. It is worth noting that these simulators not only simulate the physical properties of an environment and provide a photorealistic representation of the world, but also provide an easily parameterizable environment, which can be adjusted according to the needs of every different real life scenarios.

Even though the aforementioned simulators provide powerful tools for developing and validating various robotics applications, it is not straightforward to use them for developing DRL methods, which typically operate over a higher level of abstraction that hides low-level details, such as how the actual control commands are processed by the robots. This limits their usefulness for developing DRL methods, since their steep learning curve and the enormous amount of development required to interface with DRL methods, considerably restricts their use by DRL researchers.

The main contribution of this work is to provide an open-source framework that can overcome the aforementioned limitations, supplying a DRL interface that is easy for the DRL research community to use. More specifically, the developed framework, called "deepbots", combines the well known OpenAI Gym [3] interface with the Webots simulator in order to establish a standard way to employ DRL in real case robotics scenarios. Deepbots aims to enable researchers to use RL in Webots and it has been created mostly for educational and research purposes. In essence, deepbots acts as a middle-ware between Webots and the DRL algorithms, exposing a Gym style interface with multiple levels of abstraction. The framework uses design patterns to achieve high code readability and reusability, allowing to easily incorporate it in most research pipelines. The aforementioned features come as an easy-to-install Python package that allows developers to efficiently implement environments that can be utilized by researchers or students to use their algorithms in realistic benchmarking. At the same time, deepbots provides ready-to-use standardized environments for well-known problems. Finally, the developed framework provides some extra tools for monitoring, e.g., tensorboard logging and plotting, allowing to directly observe the training progress. Deepbots is available at https://github.com/aidudezzz/deepbots.

The paper is structured as follows. First, Sect. 2 provides a brief overview of existing tools and simulators that are typically used for training DRL algorithms and highlights the need for providing a standardized DRL framework over the

simulators to lower the barrier for accessing these tools by DRL researchers. Then, a detailed description of deepbots is provided in Sect. 3, while a set of already implemented examples, along with results achieved by well-established baseline RL algorithms, are provided in Sect. 4. Finally, Sect. 5 concludes this paper.

2 Related Work

First, the well-established OpenAI Gym toolkit, as well as the Webots simulator, are briefly introduced. Then, a number of related frameworks are also discussed and compared to the proposed deepbots framework.

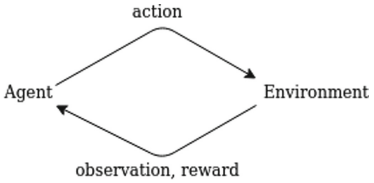


Fig. 1. OpenAI Gym interface

The OpenAI Gym, or simply "Gym", framework has been established as the standard interface between the actual simulator and RL algorithm [3]. According to the OpenAI Gym documentation, the framework follows the classic "agent-environment loop", as shown in Fig. 1, and it defines a set of environments. An environment for each step receives an action from the agent and returns a new observation and reward for the action. This procedure is repeated and separated in an episodic format. Except that, Gym standardizes a collection of testing environments for RL benchmarking. Even though OpenAI Gym is an easy-to-use tool to demonstrate the capabilities of RL in practice, it comes only with toy, unrealistic and difficult to extend scenarios. It needs several external dependencies to build more complex environments, like the MuJoCo simulator [21], which is a proprietary piece of software, which barriers its use and ability to be extended.

As RL problems become more and more sophisticated, researchers have to come up with even more complicated simulations. Self-driving cars, multi-drone scenarios, and other tasks with many more degrees of freedom synthesize the new big picture of RL research. Consequently, that leads to the need of even more accurate and realistic simulations, such as Webots [16], which is a free and open-source 3D robot simulator. Webots provides customizable environments, the ability to create robots from scratch, as well as high fidelity simulations with realistic graphics and is also Robot Operating System (ROS) compliant. It comes preloaded with several well-known robots, e.g., E-puck [7], iCub [14], etc. Robots can be wheeled or legged and use actuators like robotic arms, etc. An array of sensors is also provided, e.g., lidars, radars, proximity sensors, light sensors, touch sensors, GPS, accelerometers, cameras, etc. These capabilities allow

it to cover a wide range of different applications. Robots are programmed via controllers that can be written in several languages (C, C++, Python, Java and MATLAB). However, even though Webots can be used for DRL, it comes with a set of limitations. The mechanisms which are used to run the different scripts are not friendly for those with DRL background, requiring a significant development overhead for supporting DRL algorithms, while there is no standardization regarding the way DRL methods are developed and interface with Webots.

Note that there is also an increasing number of works that attempt to formalize and facilitate the usage of RL in robotic simulators. However, none of these works target the state-of-the-art Webots simulator. For example, Gym-Ignition [6] is a work which aims to expose an OpenAI Gym interface to create reproducible robot environments for RL research. The framework has been designed for the Gazebo simulator and provides interconnection to external third party software, multiple physics and rendering engines, distributed simulation capabilities and it is ROS compliant. Other than that, [22] extends the Gym interface with ROS compliance and it uses the Gazebo simulator as well. The latest version of this work [13] is compatible with ROS 2 and is extended and applied in more real world oriented examples. All of these works are limited by the low quality graphics provided by the Gazebo simulator, rendering them less useful for DRL algorithms that rely on visual input. Finally, Isaac Gym [9] is a powerful software development toolkit providing photorealistic rendering, parallelization and is packed as a unified framework for DRL and robotics. However, its closed source nature can render it difficult to use, especially on scenarios that deviate from its original use cases. To the best of our knowledge this is the first work which t provide a generic OpenAI Gym interface for Webots, standardizing the way DRL algorithms interface with Webots and provide easy access to a state-of-the-art simulator.

3 Deepbots

Deepbots follows the same agent-environment loop as the OpenAI Gym framework, with the only difference being that the agent, which is responsible for choosing an action, runs on the supervisor and the observations are acquired by the robot. This master-minion protocol is not problem-specific and thus has the advantage of generalization, due to the fact that it can be used in more than one examples. That makes it easier to construct various use cases and utilize them as benchmarks. In this way, the deepbots framework acts as a wrapper, meaning that it wraps up and hides certain operations from the users, so that they are able to focus on the DRL task, rather than handling all the technical simulator-specific details. At the same time, deepbots also enriches the training pipeline with live monitoring features, which helps researchers get early observations about the fundamental parts of the training process. All these features contribute into providing a powerful DRL-oriented abstraction over Webots, allowing researchers to quickly model different use cases and simulation environments, as well as employ them to develop sophisticated DRL algorithms.

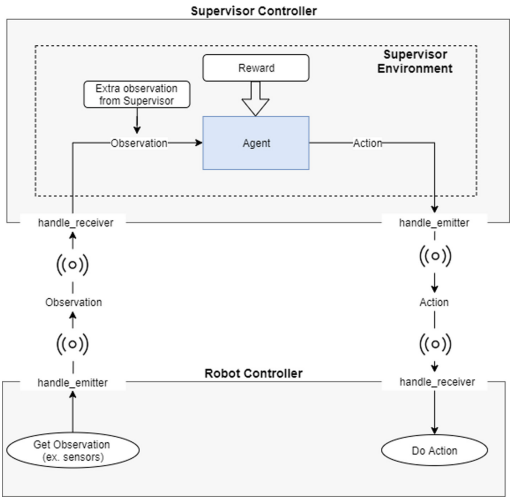


Fig. 2. Deepbots supervisor-controller communication

Before describing deepbots in detail, it is useful to briefly review the way Webots handles various simulation tasks. Webots represents scenes with a tree structure in which the root node is the world and its children nodes are the different items in the world. Consequently, a robot should be a node under the root node which contains a controller. Controllers are scripts responsible for the node's functionality. A robot for example has a controller in order to read values from sensors and act accordingly. For RL purposes, it is necessary to include a special supervisor node which has full knowledge of the world and it can get information for and modify every other node. For example, the supervisor can be used to move items in the world or measure distances between a robot and a target.

With respect to the aforementioned logic, deepbots gives the ability to easily construct DRL tasks with minimal effort. The basic structure of deepbots communication scheme is depicted in Fig. 2, where the supervisor controller is the script of the supervisor and the robot controller is the script of the robot. The communication between supervisor and robot is achieved via emitters/receivers, which broadcast and receive messages respectively. Without loss of generality, the supervisor is a node without mass or any physical properties in the simulation. For example, in a real case scenario, a supervisor could be a laptop which transmits actions to the robot, but without interacting with the actual scene. Furthermore, the emitter and the receiver could be any possible device, either cable or wireless, properly set up for this task.

Deepbots works as follows: first of all the simulator has to be reset in the initial state. On the one hand, the robot collects the first set of observations and by using its emitter sends the information to the supervisor node. On the other hand, the supervisor receives the observations with its receiver component

and in turn passes them to the agent. In this step, if needed, the supervisor can augment the observation with extra information, e.g., Euclidean distances with respect to some ground truth objects, which are unavailable to the robot and its sensors. Except for the observation, the supervisor can pass the previous action, reward and/or any other state information to the agent. It should be mentioned that deepbots is not bound to any DL framework and the agent can be written in any Python-based DL framework, e.g., PyTorch, TensorFlow, Numpy, etc. After that, the agent produces an action which is going to be transmitted to the robot via the emitter. Finally, the robot has to perform the action which was received, with its actuators. The aforementioned procedure is performed iteratively until the robot achieves its objective or for a certain number of epochs/episodes, or whatever condition is needed by the use case.

```
class FindTargetSupervisor(SupervisorEmitterReceiver):
    def get_observation(self):...
    def get_reward(self, action):...
    def is_done(self):...
    def reset(self):...
    def step(self, action):...

    env = FindTargetSupervisor()
    env = TensorboardLogger(env)
    agent = DDPG(...)
    for i in range(EPOCHS):
        done = False
        score = 0
        obs = env.reset()
        while not done:
            act = agent.choose_action(obs)
            obs, reward, done, info = env.step(act)
            agent.remember(obs, action, reward, done)
            agent.learn()
            score += reward
```

Code Example 1.1: Supervisor controller code example

In order to implement an agent, the user has to implement two scripts at each side of the communication channel and the framework handles the details. On the supervisor side, the user has to create a Gym environment with the well known abstract methods and train/evaluate the DRL model, as shown in Code Example 1.1. While on the other side, a simple script has to be written for reading values from sensors and translating messages to the actual values needed by the actuators. A typical script for this task is shown in Code Example 1.2. The deepbots framework runs all the essential operations needed by the simulation, executes the step function and handles the communication between the supervisor and the robot controller in the background. In addition, by following the framework workflow, cleaner code is achieved, while the agent logic is separated from the actuator manipulation and it is closer to the physical cases. Furthermore, the framework logic is similar to ROS and can be integrated with it with minimal effort.

```
class FindTargetRobot(RobotEmitterReceiver):
  def create_message(self):
    message = []
      for rangefinder in self.rangefinders:
        message.append(rangefinder.value())
    return message
  def use_message_data(self, message):
    gas = float(message[1])
    wheel = float(message[0])
    ...
robot_controller = FindTargetRobot()
robot_controller.run()
```

Code Example 1.2: Robot controller code example

As the deepbots framework mostly aims to be a user-friendly tool for educational and research purposes, it has different levels of abstraction. An overview of the abstraction level class diagram of deepbots is provided in Fig. 3. For example, users can choose if they would use JSON emitters and receivers or if they want to go on with an implementation from scratch. At the top of the abstraction hierarchy is the *SupervisorEnv*, which is the OpenAI Gym abstract class. Below that level is the actual implementation which resolves the communication between the supervisor and the robot. Similarly, the robot has also different levels of abstraction. A user can choose among certain types of message formats to transmit actions and observations. Extra features can be added to the framework as decorator classes by implementing the OpenAI Gym interface, as demonstrated in line 9 of Code Example 1.1. This design pattern could be used to stack different controls, monitoring and other functionalities.

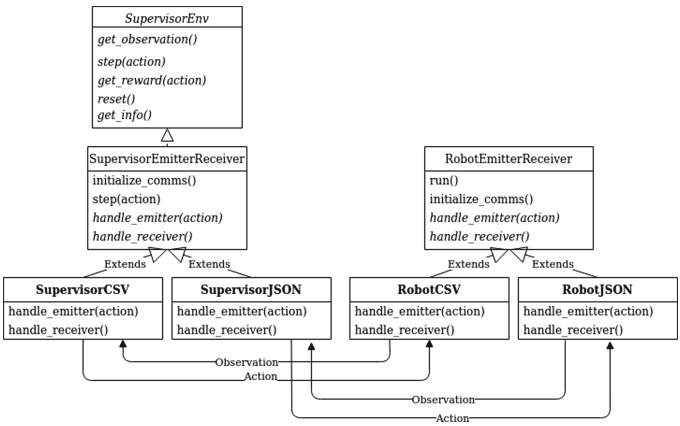


Fig. 3. Abstraction level class diagram

4 Example Environments

Deepbots contains a collection of ready-to-use environments, which showcase uses of the framework in toy or complicated examples. On the one hand, the community can contribute new environments and use cases to enrich the existing collection. On the other hand, this collection can be used by researchers to benchmark RL algorithms in Webots. Three environments of varying complexity are presented in this Section.

(a) (b) (c)

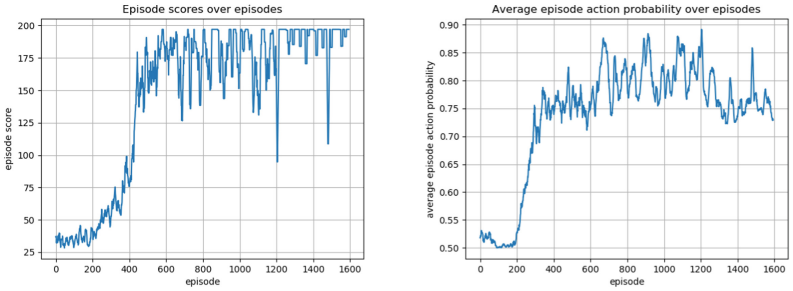
Fig. 4. (a) CartPole: the x axis is the cart motion axis, the y axis is the pole rotation axis, (b) PitEscape: the robot inside the pit, (c) FindBall: the robot searching for the yellow ball (Color figure online)

4.1 CartPole

The CartPole example is based on the problem described in [2] and adapted to Webots. In the world exists an arena, and a small four wheeled cart that has a long pole connected to it by a free hinge, as shown in Fig. 4. The hinge contains a sensor to measure the angle the pole has off vertical. The pole acts as an inverted pendulum and the goal is to keep it vertical by moving the cart forward and backward. This task is tackled with the discrete version of the Proximal Policy Optimization (PPO) RL algorithm [19].

The observation contains the cart position and velocity on the x axis, the pole angle off vertical and the pole velocity at its tip. The action space is discrete containing two possible actions for each time step, move forward or move backward. For every step taken, the agent is rewarded with +1 including the termination step. Each episode is terminated after a maximum 200 steps or earlier if the pole has fallen $\pm 15°$ off vertical or the cart has moved more than ± 0.39 m on the x axis. To consider the task solved, the agent has to achieve an average score of over 195.0 in 100 consecutive episodes.

The learning curve using the PPO algorithm, as well as the average action probability over the training process are depicted in Fig. 5. The actor and critic consist of small two-layered neural networks with 10 ReLU neurons on each layer and the agent was able to solve the problem after running for a simulated time of about 2.5 h.



(a) (b)

Fig. 5. CartPole: (a) reward accumulated for each episode, and (b) average probability of selected actions per episode

4.2 Pit Escape

The Pit Escape example that can be seen in Fig. 4 is a problem taken from *robotbenchmark*[1].

The Pit Escape world comprises of a pit with a radius of 2.7, where inside it lies a BB-8 robot with a spherical body that acts as a wheel [16]. The robot contains a pitch and a yaw motor and can move by rolling forward or backward (pitch) or by turning left and right (yaw). The task is to escape from the pit, but its motors are not powerful enough to escape by just moving forward, so it needs to move in some other way. This task is also tackled with the discrete version of the PPO RL algorithm [19].

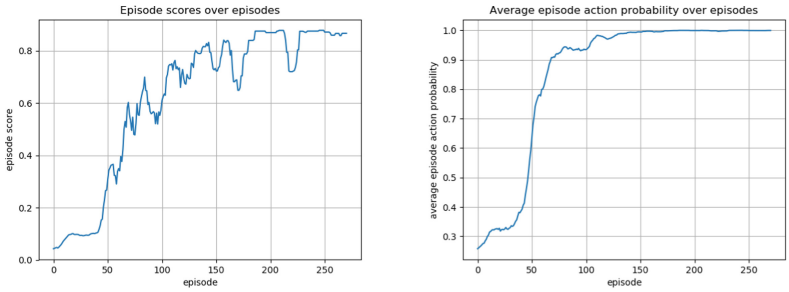
This problem is very similar to the Mountain Car one [18], but in three dimensions and has more complex observation and action spaces. The robot contains a gyroscope and an accelerometer which provide the observation. Thus, the observation contains the robot orientation and acceleration in the x, y, z axes, i.e., a total of 6 values. The action space is discrete containing 4 possible actions for each time step. With each action the robot can set its motor speeds to their maximum or minimum values. Each episode lasts 60 s and the reward function is based on a metric M:

$$M = \begin{cases} 0.5\frac{d}{R} & d < R \\ 0.5 + 0.5\frac{T-t}{T} & d > R \end{cases}, \tag{1}$$

where d is the maximum distance achieved from the center of the pit until now in the episode, R is the radius of the pit, T is the maximum time allowed per episode (60 s), and t is the time until now in the episode. M only changes when a higher distance from the center is achieved during the episode. For each time step, based on the change between the previous step and current step metrics, the reward R_i for step i is calculated as $R_i = M_{old} - M$, where M_{old} is the previous step metric and M is the current step metric. An episode terminates

[1] Robotbenchmark, https://robotbenchmark.net.

after 60 s or if the robot has escaped the pit, which is calculated by the distance between the robot and the pit center.



(a) (b)

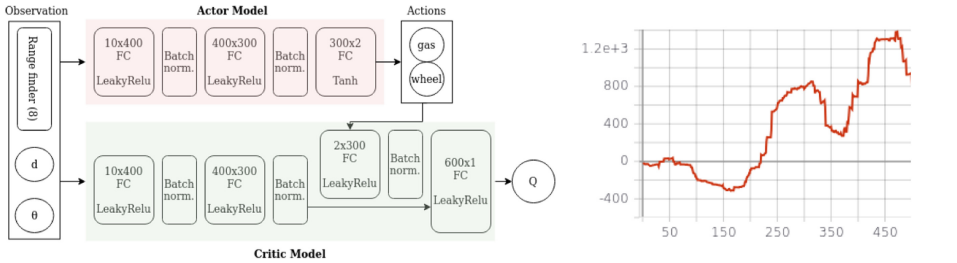
Fig. 6. Pit Escape: (a) reward accumulated for each episode, and (b) average probability of selected actions per episode

Two-layered networks with 60 ReLU neurons on each layer were used for the actor and critic models, while the learning curves are provided in Fig. 6. The agent achieved an average episode score of over 0.8 after training for a simulated time of about 3 h.

4.3 Find the Ball and Avoid Obstacles

The last example is a typical find target and avoid obstacles task with a simple world configuration. For this task the E-puck robot is used [7], which is a compact mobile robot developed by GCtronic and EPFL and is included in Webots. The world configuration contains an obstacle and a target ball. Different world configurations with incremental difficulty have been used in the training sessions for better generalization. It has been observed that the convergence of training algorithms can be improved by incrementing the difficulty of the problems [10]. The E-puck robot uses 8 IR proximity distance sensors and it has two motors for moving. The agent, apart from the distance sensor values, also receives the Euclidean distance and angle from the target. Consequently, the observation the agent gets is an one-dimensional vector with 10 values. On the other hand, the actuators are motors, which means that the outputs of the agent are two values controlling the forward/backward movement and left/right turning respectively (referred to as gas and wheel).

In order to deal with the continuous action space problem, the Deep Deterministic Policy Gradient (DDPG) algorithm was used to tackle this task [12]. The architecture of the models is described in Fig. 7. The reward function used for training the agent is calculated as:



(a) (b)

Fig. 7. FindBall: (a) DDPG models architecture, and (b) reward accumulated for each episode

$$R = \begin{cases} -10 & s > T_{steps} \\ +10 & d < T_{distance} \\ -1 & crashed \\ \frac{1}{d} - \frac{T_{steps}}{s} & otherwise \end{cases}, \tag{2}$$

where s is the current step, T_{steps} the maximum allowed steps, d the current distance from target and $T_{distance}$ the minimum distance between the robot and the target which is considered as reaching the goal. This reward function takes into account both the distance from the target and the number of steps elapsed, while when the robot crashes on an obstacle or does not find the target after certain steps, it provides a negative reward and the episode is terminated.

The agent has been trained for 500 episodes and the accumulated reward is presented in Fig. 7. The training session lasted for about 1 h of wall clock time and about 3 h of simulated time. Although the agent solved the problem, it fails to generalize in more complicated scenes, highlighting the challenging nature of this baseline, that can be used for benchmarking future DRL algorithms.

5 Conclusions

Even though there have been attempts to formalize the use of RL in robotic simulators, none of them targets the state-of-the-art simulator Webots. The deepbots framework comes to fill that gap for anyone who wants to apply RL and DRL in a high fidelity simulator. Deepbots provides a standardized way to apply RL on Webots, by focusing only on parts that are important for the task at hand. Deepbots can fit a high variety of use cases, both research and educational, and can be extended by the community due to its open-source nature. Together with Webots, it provides a test bed for algorithm research and task solving with RL, as well as a practical platform for students to experiment with and learn about RL and robotics.

Acknowledgments. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871449 (OpenDR). This publication reflects the authors' views only. The European Commission is not responsible for any use that may be made of the information it contains.

References

1. Actin. https://www.energid.com/actin. Industrial-Grade Software for Advanced Robotic Solutions
2. Barto, A., Sutton, R., Anderson, C.: Neuron like elements that can solve difficult learning control problems. IEEE Trans. Syst. Man Cybern. **13**, 834–846 (1970)
3. Brockman, G., et al.: OpenAI Gym (2016)
4. Diankov, R.: Automated construction of robotic manipulation programs. Ph.D. thesis, Carnegie Mellon University, Robotics Institute, August 2010. http://www.programmingvision.com/rosen_diankov_thesis.pdf
5. Dulac-Arnold, G., Mankowitz, D.J., Hester, T.: Challenges of real-world reinforcement learning. CoRR abs/1904.12901 (2019). http://arxiv.org/abs/1904.12901
6. Ferigo, D., Traversaro, S., Metta, G., Pucci, D.: Gym-ignition: reproducible robotic simulations for reinforcement learning (2019)
7. Gonçalves, P., et al.: The e-puck, a robot designed for education in engineering. In: Proceedings of the Conference on Autonomous Robot Systems and Competitions, vol. 1, January 2009
8. Goodfellow, I.G., Bengio, Y., Courville, A.C.: Deep learning. Nature **521**, 436–444 (2015)
9. Gym, I.: https://www.nvidia.com/en-in/deep-learning-ai/industries/robotics
10. Hacohen, G., Weinshall, D.: On the power of curriculum learning in training deep networks (2019)
11. Koenig, N., Howard, A.: Design and use paradigms for gazebo, an open-source multi-robot simulator. In: Proceedings of the 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems, vol. 3, pp. 2149–2154 (2004)
12. Lillicrap, T.P., et al.: Continuous control with deep reinforcement learning (2015)
13. Lopez, N.G., et al.: Gym-gazebo2, a toolkit for reinforcement learning using ROS 2 and Gazebo (2019)
14. Metta, G., Sandini, G., Vernon, D., Natale, L., Nori, F.: The iCub humanoid robot: an open platform for research in embodied cognition. In: Performance Metrics for Intelligent Systems (PerMIS) Workshop, January 2008
15. Michel, O.: Webots: professional mobile robot simulation. J. Adv. Robot. Syst. **1**(1), 39–42 (2004)
16. Michel, O.: WebotsTM: professional mobile robot simulation. Int. J. Adv. Robot. Syst. **1**, 40–43 (2004)
17. Mnih, V., et al.: Playing Atari with deep reinforcement learning (2013)
18. Moore, A.: Efficient memory-based learning for robot control, June 2002
19. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms (2017)
20. Steckelmacher, D., Plisnier, H., Roijers, D.M., Nowé, A.: Sample-efficient model-free reinforcement learning with off-policy critics (2019)
21. Todorov, E., Erez, T., Tassa, Y.: MuJoCo: a physics engine for model-based control. pp. 5026–5033, October 2012
22. Zamora, I., Lopez, N.G., Vilches, V.M., Cordero, A.H.: Extending the OpenAI Gym for robotics: a toolkit for reinforcement learning using ROS and Gazebo (2016)