Extending the OpenAI Gym for robotics: a toolkit for reinforcement learning using ROS and Gazebo

Iker Zamora [*], **Nestor Gonzalez Lopez** [*], **Víctor Mayoral Vilches** [*], **and Alejandro Hernández Cordero** [*]

[*] Erle Robotics

Published as a whitepaper

This paper presents an extension of the OpenAI Gym for robotics using the Robot Operating System (ROS) and the Gazebo simulator. The content discusses the software architecture proposed and the results obtained by using two Reinforcement Learning techniques: Q-Learning and Sarsa. Ultimately, the output of this work presents a benchmarking system for robotics that allows different techniques and algorithms to be compared using the same virtual conditions.

Introduction

Reinforcement Learning (RL) is an area of machine learning where a software agent learns by interacting with an environment, observing the results of these interactions with the aim of achieving the maximum possible cumulative reward. This imitates the trial-and-error method used by humans to learn, which consists of taking actions and receiving positive or negative feedback.

In the context of robotics, reinforcement learning offers a framework for the design of sophisticated and hard-to-engineer behaviors [2]. The challenge is to build a simple environment where this machine learning techniques can be validated, and later applied in a real scenario.

OpenAI Gym [1] is a is a toolkit for reinforcement learning research that has recently gained popularity in the machine learning community. The work presented here follows the same baseline structure displayed by researchers in the OpenAI Gym (gym.openai.com), and builds a gazebo environment on top of that. OpenAI Gym focuses on the episodic setting of RL, aiming to maximize the expectation of total reward each episode and to get an acceptable level of performance as fast as possible. This toolkit aims to integrate the Gym API with robotic hardware, validating reinforcement learning algorithms in real environments. Real-world operation is achieved combining Gazebo simulator [3], a 3D modeling and rendering tool, with ROS [5] (Robot Operating System), a set of libraries and tools that help software developers create robot applications.

As benchmarking in robotics remains an unsolved issue, this work aims to provide a toolkit for robot researchers to compare their techniques in a well-defined (API-wise) controlled environment that should speed up development of robotic solutions.

Background

Reinforcement Learning has taken an increasingly important role for its application in robotics [6]. This technique offers robots the ability to learn previously missing abilities [4] like learning hard to code behaviours or optimizing problems without an accepted *closed* solution.

The main problem with RL in robotics is the high cost per-trial, which is not only the economical cost but also the long time needed to perform learning operations. Another known issue is that learning with a real robot in a real environment can be dangerous, specially with flying robots like *quad-copters*. In order to overcome this difficulties, advanced robotics simulators like Gazebo have been developed which help saving costs, reducing time and speeding up the simulation.

The idea of combining learning in simulation and in a real environment was popularized by the Dyna-architecture (Sutton, 1990), prioritized sweeping (Moore and Atkeson,1993), and incremental multi-step Q-Learning (Pengand Williams, 1996) in reinforcement learning. In robot reinforcement learning, the learning step on the simulated system is often called "mental rehearsal" [2]. Mental reharsal in robotics can be improved by obtaining information from the real world in order to create an accurate simulated environment. Once the training is done, just the resulting policy is transferred to the real robot.

Architecture

The architecture consits of three main software blocks: OpenAI Gym, ROS and Gazebo (Figure 1). Environments developed in OpenAI Gym interact with the Robot Operating System, which is the connection between the Gym itself and Gazebo simulator. Gazebo provides a robust physics engine, high-quality graphics, and convenient programmatic and graphical interfaces.

The physics engine needs a robot definition[1] in order to sim-

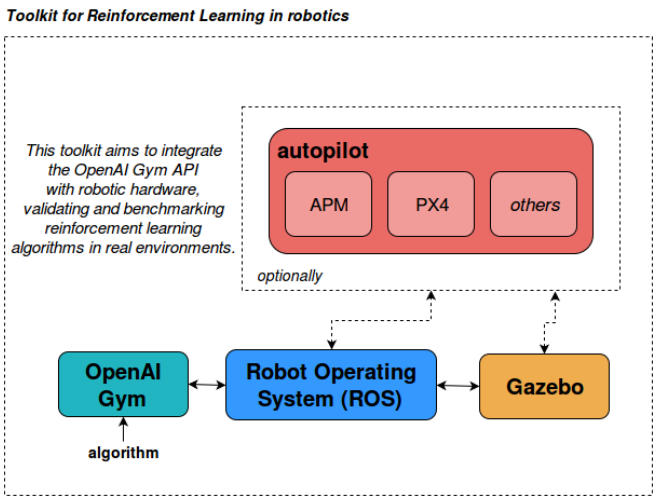


Fig. 1: Simplified software architecture used in OpenAI Gym for robotics.

[1] Unified Robot Description Format (URDF)

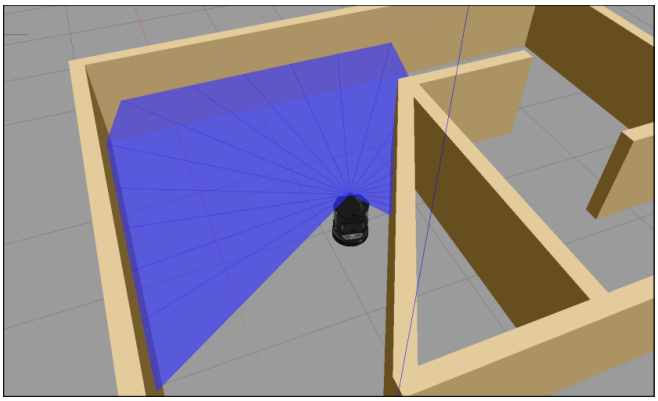


Fig. 5: Turtlebot in a virtual environment learning how to navigate autonomously using a LIDAR

ulate it, which is provided by ROS or a gazebo plugin that interacts with an autopilot in some cases (depends on the robot software architecture). The Turtlebot is encapsulated in ROS packages while robots using an autopilot like Erle-Copter and Erle-Rover are defined using the corresponding autopilot. Our policy is that every robot needs to have an interface with ROS, which will mantain an organized architecture.

Figure 1 presents the a simplified diagram of the software architecture adopted in our work. Our software structure provides similar APIs to the ones presented initially by OpenAI *Gym*. We added a new collection of environments called *gazebo* where we store our own gazebo environments with their corresponding assets. The needed installation files are stored inside the *gazebo* collection folder, which gives the end-user an easier to modify infrastructure.

Installation and setup consits of a ROS catkin workspace containing the ROS packages required for the robots (e.g.: Turtlebot, Erle-Rover and Erle-Copter) and optionally the appropriate autopilot that powers the logic of the robot. In this particular case we used the APM autopilot thereby the source code is also required for simulating Erle-Rover and Erle-Copter. Robots using APM stack need to use a specific plugin in order to communicate with a ROS/Gazebo simulation.

Environments and Robots

We have created a collection of six environments for three robots: Turtlebot, Erle-Rover and Erle-Copter. Following the desing decisions of OpenAI Gym, we only provide an abstraction for the environment, not the agent. This means each environment is an independent set of items formed mainly by a *robot* and a *world*.

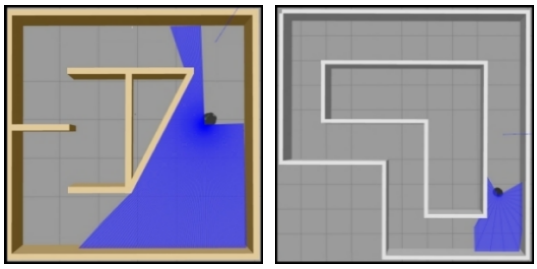
Figure 5 displays an environment created with the Turtlebot *robot* which has been provided with a LIDAR sensor using and a *world* called *Circuit*. If we wanted to test our reinforcement learning algorithm with the Turtlebot but this time using positioning information, we would need to create a completely new environment.

The following are the initial environments and robots provided by our team at Erle Robotics. Potentially, the amount of supported robots/environments will will grow over time.

Turtlebot. TurtleBot combines popular off-the-shelf robot components like the iRobot Create, Yujin Robot's Kobuki, Microsoft's Kinect and Asus' Xtion Pro into an integrated development platform for ROS applications. For more information, please see turtlebot.com .

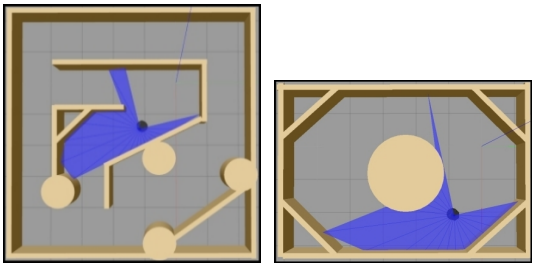
The following are the four environment currently available for the Turtlebot:

- *GazeboCircuitTurtlebotLIDAR-v0* (Figure 2.a) : A simple circuit with a diagonal wall, which increases the complexity of the learning.
- *GazeboCircuit2TurtlebotLIDAR-v0* (Figure 2.b) : A simple circuit with straight tracks and 90 degree turns. Note that the third curve is a left turn, while the others are right turns.
- *GazeboMazeTurtlebotLIDAR-v0* (Figure 2.c) : A complex maze with different wall shapes and some narrow tracks.
- *GazeboRoundTurtlebotLIDAR-v0* (Figure 2.d) : A simple oval shaped circuit.



(a) A simple circuit with a diagonal wall, which increases the complexity of the learning.

(b) A simple circuit with straight tracks and 90 degree turns. Note that the third curve is a left turn, while the others are right turns.



(c) A complex maze with different wall shapes and some narrow tracks.

(d) A simple oval shaped circuit.

Fig. 2: Environments available using Turtlebot robot and a LIDAR sensor.

Erle-Rover. A Linux-based smart car powered by the APM autopilot and with support for the Robot Operating System erlerobotics.com/blog/erle-rover .

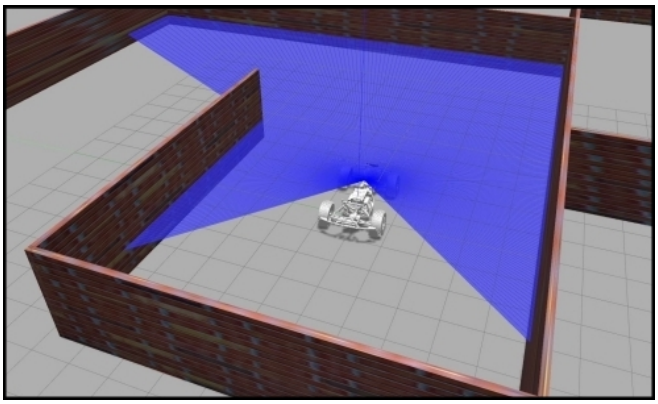


Fig. 3: Erle-Rover learning to avoid obstacles in a big maze with wide tracks and 90 degree left and right turns, an environment called *GazeboMazeErleRoverLIDAR-v0*.

Erle-Copter. A Linux-based drone powered by the open source APM autopilot and with support for the Robot Operating System erlerobotics.com/blog/erle-copter .

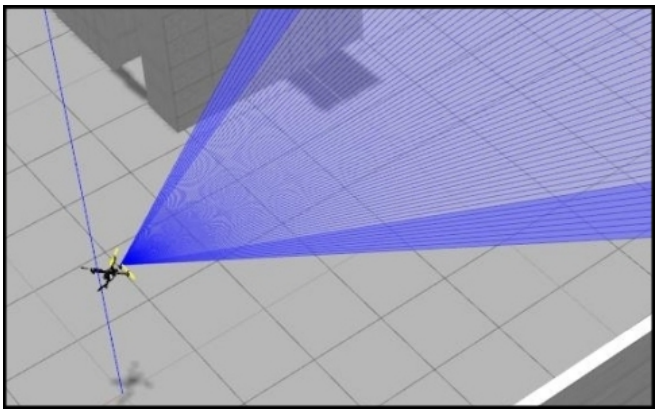


Fig. 4: Erle-Copter learning to avoid obstacles in ErleRobotics office without ceiling, an environment called *GazeboOfficeErleCopterLIDAR-v0*.

Results

We have experimented with two Reinforcement Learning algorithms, Q-Learning and Sarsa. The turtlebot has been used to benchmark the algorithms since we get faster simulation speeds than robots using and autopilot. We get around 60RTF (Real Time Factor), which means 60 times normal simulation speed and 30RTF when we launch the visual interfaze. This benchmarks have been made using a i7 6700 CPU and non-GPU laser mode.

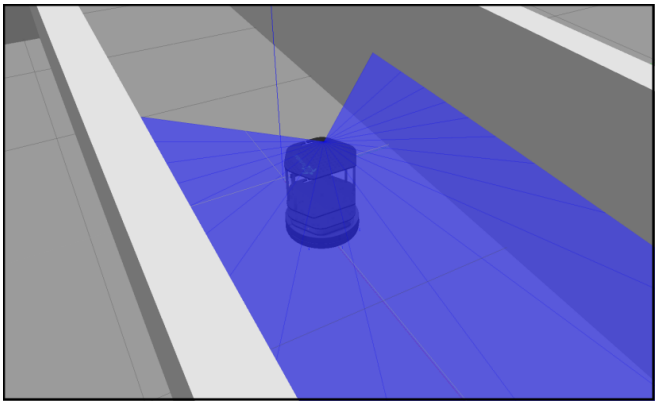


Fig. 6: Turtlebot with LIDAR in *GazeboCircuit2TurtlebotLIDAR-v0 environment.*

We will use *GazeboCircuit2TurtlebotLIDAR-v0* (Figure 2.b & 6) to perform the benchmarking. This environment consists of a simple straight lined circuit with five right turns and one left turn. We will use just a LIDAR sensor to train the Turtlebot, no positioning or other kind of data will be used. Both algorithms will use the same hyperparameters and exact environment. The actions and rewards forming the environment have been adapted to get an optimal training performance.

Actions

- Forward: v = 0.3 m/s
- Left/Right: v = 0.05 m/s , w = +-0.3 rad/s

Taking only three actions will lead to a faster learning, as the 'Q' function will fill its table faster. Obstacle avoidance could be performed with just two turning actions, but the learnt movements would be less practical.

Left and right turns have a small linear velocity just to accelerate the learning process and avoid undesired behaviours. Setting the linear velocity to zero, the robot could learn to turn around itself constantly as it would not crash and still earn positive rewards. As Atkeson and Schaal point out:

> Reinforcement learning approaches exploit such model inaccuracies if they are beneficial for the reward received in simulation.

This could be avoided changing the reward system, but we found the optimal values for this environment are the following.

Rewards

- Forward: 5
- Left/Right: 1
- Crash: -200

Forward actions take five times more reward than turns, this will make the robot take more forward actions as they give more reward. We want to take as many forward actions as possible so that the robot goes forward in straight tracks,

which will lead to a faster and more realistic behaviour.

Left and right actions are rewarded with 1, as they are needed to avoid crashes too. Setting them higher would result in a zigzagging behaviour.

Crashes earn very negative rewards for obvious reasons, we want to avoid obstacles.

Q-Learning

Q-Learning[10] is an Off-Policy algorithm for Temporal Difference learning. Q-Learning learns the optimal policy even when actions are selected according to an exploratory or even random policy [7].

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma max\alpha, Q(s_{t+1}, a_t) - Q(s_t, a_t)]$$

Let's see how the Turtlebot learns using Q-Learning in *GazeboCircuit2TurtlebotLIDAR-v0* environment. Simplified code is presented below:

```
env = gym.make('GazeboCircuit2TurtlebotLIDAR-v0')
qlearn = qlearn.Qlearn(alp=0.2,gam=0.9,eps=0.9)
for x in range(3000):
    observation = env.reset()
    state = ''.join(map(str, observation))
    for i in range(1500):
        action = sarsa.chooseAction(state)
        observation, reward, done = env.step(action)
        nextState = ''.join(map(str, observation))
        qlearn.learn(state, action, reward,
                     nextState)
        if not(done):
            state = nextState
        else:
            break
```

After selecting environment we want to test, we have to initialize Q-learn with three parameters. Small changes in these hyperparameters can result in substantial changes in the learning of our robot. Those parameters are the following:

- α, Learning rate:. Setting it to 0 means the robot will not learn and a high value such as 0.9 means that learning can occur quickly.
- γ, Discount factor:. A factor of 0 will make the agent consider only current rewards, while a factor approaching 1 will make it strive for a long-term high reward.
- ϵ, Exploration constant:. Used to randomize decisions, setting a high value such as 0.9 will make 90% of the actions to be stochastic. An intersting technique is to set an epsilon decay, where the agent starts taking more randon actions (exploration phase) and ends in an exploitation phase where all or most of the actions performed are selected from the learning table instead of being random.

The selected initial hyperparameters are, $\alpha = 0.2$, $\gamma = 0.9$ and $\epsilon = 0.9$. In this example we use the epsilon decay technique, being the decay $\epsilon * 0.9986$ every episode until it reaches our minimum epsilon value, 0.05 in this case.

We want to run the simulation for 3000 episodes. Each episode the simulation will be resetted and the robot will start again from its initial position. Every episode we try to make a maximum of 1500 iterations, which means the robot has not crashed. Every iteration we choose an action, take a step (execute an action for a short time or distance) and receive feedback. That feedback is called *observation* and it is returns the next state to be taken and the received reward.

To sum up, every episode the robot tries to take as many steps as possible, learning every step from the obtained rewards.

The following graph shows the results obtained through 3000 episodes.

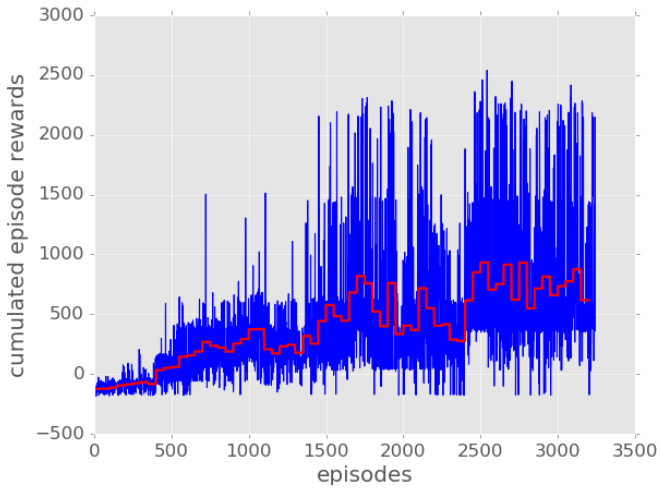


Fig. 7: Cumulated reward graph obtained from the monitoring of *GazeboCircuit2TurtlebotLIDAR-v0* (Figure 2.b) environment using Q-Learning. The blue line prints the whole set of readings while the red line shows an approximation to the averaged rewards.

We get decent results after 1600 episodes. Cumulated rewards around 2000 or higher usually mean the robot did not crash or gave more than two laps.

Sarsa

Sarsa [9] is an On-Policy algorithm for Temporal Difference Learning. The major difference between it and Q-Learning, is that the maximum reward for the next state is not necessarily used for updating the Q-values. Instead, a new action, and therefore reward, is selected using the same policy that determined the original action [7].

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

Let's now compare how the Turtlebot learns using Sarsa in *GazeboCircuit2TurtlebotLIDAR-v0* environment. Simplified code is presented below

```
env = gym.make('GazeboCircuit2TurtlebotLIDAR-v0')
sarsa = sarsa.Sarsa(alp=0.2,gam=0.9,eps=0.9)
for x in range(3000):
    observation = env.reset()
    state = ''.join(map(str, observation))
    for i in range(1500):
        action = sarsa.chooseAction(state)
        observation, reward, done = env.step(action)
        nextState = ''.join(map(str, observation))
        nextAction = sarsa.chooseAction(nextState)
        sarsa.learn(state, action, reward,
                    nextState, nextAction)
        if not(done):
            state = nextState
        else:
            break
```

After selecting environment we want to test, we have to initialize Sarsa with three parameters: alpha, gamma and epsilon. They work in the same way as Q-learn, so we used the same as in the previous Q-leaning test . The selected initial hyperparameters are, $\alpha = 0.2$, $\gamma = 0.9$ and $\epsilon = 0.9$ with $\epsilon * 0.9986$ epsilon decay.

We want to run the simulation for 3000 episodes. Every episode we try to make a maximum of 1500 iterations, which means the robot has not crashed. Every iteration we choose an action, take a step (execute an action for a short time or distance) and receive feedback. That feedback is called *observation* and it is used to build the next action to be taken.

Since we are using Sarsa (on-policy) and not Q-learn (off-policy), we need to choose another action before we learn. This is done by selecting a new action from the previously built next state.

As explained in the Q-Learning section, every episode the robot tries to take as many steps as possible, learning every step from the obtained rewards.

The following graph shows the results obtained through 3000 episodes.

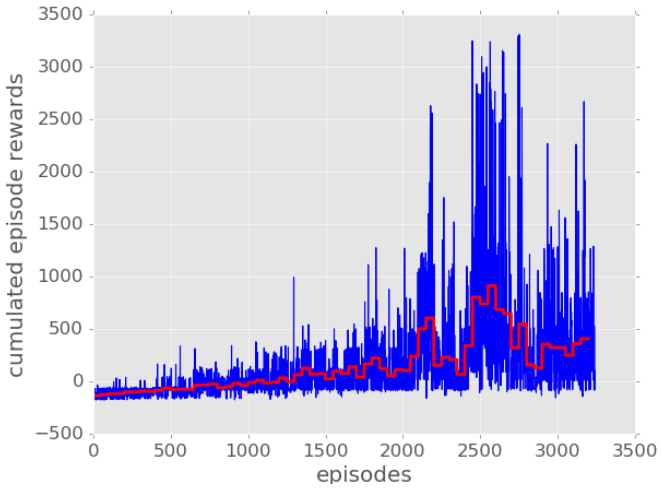


Fig. 8: Cumulated reward graph obtained from the monitoring of *GazeboCircuit2TurtlebotLIDAR-v0* (Figure 2.b) environment using Sarsa. The blue line prints the whole set of readings while the red line shows an approximation to the averaged rewards.

We get decent results after 2500 episodes. Cumulated rewards around 2000 or higher usually mean the robot did not crash or gave more than two laps.

Benchmarking

The learning in Q-Learning occurs faster than in Sarsa, this happens because Q-Learning is able to learn a policy even if taken actions are chosen randomly. However, Q-learning shows more risky moves (taking turns really close to walls) while in Sarsa we see a smoother general behaviour. The major difference between Sarsa and Q-Learning, is that the maximum reward for the next state is not necessarily used for updating the Q-values (learning table). Instead, a new action, and therefore reward, is selected using the same policy that determined the original action [7]. This is how Sarsa is able to take into account the control policy of the agent during learning. It means that information needs to be stored longer before the action values can be updated, but also means that our robot is going to take risky actions much frequently [8]. This smoother behaviour where forward actions are being exploited in straight tracks leads to higher maximum cumulated rewards. We get values near 3500 in Sarsa while just get cumulated rewards around 2500 in Q-Learning. Running Sarsa for more episodes will cause to get higher average rewards.

The table below provides a numerical comparison of Q-Learning and Sarsa representing the average reward value over 200 consecutive episodes. From the data, one can tell that learning occurs much faster using the Q-Learning technique:

Table 1: Average reward value over a 200 episode interval in 3000 episode long tests using Q-Learning and Sarsa.

Episode interval	Q-Learning	Sarsa
0-200	-114	-124
200-400	-79	-98
400-600	72	-75
600-800	212	-43
800-1000	239	-43
1000-1200	282	-6
1200-1400	243	55
1400-1600	439	65
1600-1800	676	104
1800-2000	503	127
2000-2200	510	361
2200-2400	345	164
2400-2600	776	698
2600-2800	805	550
2800-3000	685	240

Variance. Although the variance presented in Figures 7 & 8 is high[2], the averaged plots and the table above show that out robot has learnt to avoid obstacles using both algorithms.

Iterating 2000 episodes more will not make the variance disappear, as we have almost reached the best behaviour possible in this highly discretized environment. Laser values are discretized so that the learning does not take too long. We take only 5 readings with integer values, which are taken uniformly from the 270° lasers horizontal field of view. To sum up, using a simple reinforcement learning technique and just a LIDAR as an input, we get quite decent results.

[2] common thing in this scenario

Future directions

The presented toolkit could be further improved in the following directions:

- Support more autopilot solutions besides APM such as PX4 or Paparazzi.
- Speed up simulation for robots using autopilots. Currently, due to limitations of the existing implementation, the simulation is set to normal(real) speed.
- Pull apart environments and agents. Testing different robots in different environments (not only the ones built specifically for them) would make the toolkit more versatile.
- Provide additional tools for comparing algorithms.
- Recommendations and results in *mental reharsal* using the presented toolkit.

ACKNOWLEDGMENTS.

This research has been funded by Erle Robotics.

References

[1] Greg Brockman et al. "OpenAI Gym". In: *arXiv preprint arXiv:1606.01540* (2016).

[2] Jens Kober, J Andrew Bagnell, and Jan Peters. "Reinforcement learning in robotics: A survey". In: *The International Journal of Robotics Research* (2013), p. 0278364913495721.

[3] Nathan Koenig and Andrew Howard. *Gazebo-3d multiple robot simulator with dynamics*. 2006.

[4] Petar Kormushev, Sylvain Calinon, and Darwin G Caldwell. "Reinforcement learning in robotics: Applications and real-world challenges". In: *Robotics* 2.3 (2013), pp. 122–148.

[5] Morgan Quigley et al. "ROS: an open-source Robot Operating System". In: *ICRA workshop on open source software*. Vol. 3. 3.2. Kobe, Japan. 2009, p. 5.

[6] *Reinforcement learning in robotics*. [Online; accessed 6-August-2016]. URL: http://blog.deeprobotics.es/robots,/ai,/deep/learning,/rl,/reinforcement/learning/2016/07/06/rl-intro/.

[7] *Reinforcement Learning: Q-Learning vs Sarsa*. [Online; accessed 7-August-2016]. URL: http://www.cse.unsw.edu.au/~cs9417ml/RL1/algorithms.html.

[8] *Reinforcement Learning: Sarsa vs Qlearn*. [Online; accessed 7-August-2016]. URL: https://studywolf.wordpress.com/2013/07/01/reinforcement-learning-sarsa-vs-q-learning/.

[9] Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*. University of Cambridge, Department of Engineering, 1994.

[10] Christopher JCH Watkins and Peter Dayan. "Q-learning". In: *Machine learning* 8.3-4 (1992), pp. 279–292.