# Control of Open Mobile Robotic Platform using Deep Reinforcement Learning

Mihai-Daniel Pavel, Sabin Rosioru, Nicoleta Arghira and Grigore Stamatescu

**Abstract** Advanced control for mobile robotic platforms allows efficient real-time navigation in structured and unstructured environments in various industry applications. Deep reinforcement learning is an emerging control strategy where and agent is trained iteratively according to an optimisation objective by using reward and penalty actions. The agent generates the neural network weights used for computing the robot command towards the reference set point. We present an application for an open hardware mobile robotic platform navigation that integrates the sensing, communication, computing and control functions into a single system for navigation in unstructured environments. Implementation is performed through a dedicated software and communication layer that integrates the hardware platform with the MATLAB environment using standardized Robot Operating System (ROS) libraries. Quantitative testing results are presented, in order to prove the viability of the solution, by defining both simulation and laboratory setting scenarios.

Mihai-Daniel Pavel
Asti Automation, Calea Plevnei 139, 060011 Bucharest, Romania, e-mail: daniel.pavel@astiautomation.com

Sabin Rosioru
Department of Automation and Industrial Informatics, University Politehnica of Bucharest, 313 Splaiul Independentei, 060042 Bucharest, Romania e-mail: sabin.rosioru@stud.acs.upb.ro

Nicoleta Arghira
Department of Automation and Industrial Informatics, University Politehnica of Bucharest, 313 Splaiul Independentei, 060042 Bucharest, Romania e-mail: nicoleta.arghira@upb.ro

Grigore Stamatescu
Department of Automation and Industrial Informatics, University Politehnica of Bucharest, 313 Splaiul Independentei, 060042 Bucharest, Romania e-mail: grigore.stamatescu@upb.ro

# 1 Introduction

Navigation of mobile robots in dynamic, unstructured, environments is relevant for many industrial, operational and emergency recovery use cases. The main challenge lays in the orchestration of the sensing, computing and control functions that allow real-time object detection and avoidance while accounting for the tracking error against the control objectives, Choi et al (2019).

Several recent works describe development of the algorithms used for mobile robot platforms and include new approaches that are based on several types neural networks, Fu et al (2019). These range from fully connected networks up to convolutional deep networks. In Shabbir and Anwer (2018), the authors state that most of the applications built for navigation and control of mobile robots, are based on computer vision, laser sensors or a combination of both. In Wang (2021), the authors present one of the most commonly used method for interior navigation, which is mapping the environment and computing the distance between the start and finish points. In general, the computation of the trajectory is done using recursive algorithms applied on two levels: one that is global and one locally, in order to avoid object collision. This implementation does not use neural networks and has a main drawback, that the platform has difficulties in regards of the environmental changes that can occur. A novel approach for this consists of deploying a neural network based algorithm that will gather data for the sensors. Based on that information, the network will produce the best commands for the mobile platform to reach desired location, avoiding any perturbation that can appear in its path. All is done without prior knowledge of the surrounding space. The network is built using Reinforcement Learning for finding the optimal route taking into account any obstruction. Digital twin type simulation can contribute to the time and cost effective modelling of the algorithms performance using a realistically model of the robotic platform, Rosioru et al (2022). Industrial communication protocols, described in Luchian et al (2021a) and in Luchian et al (2021b), serve as a supporting technology over which the advanced control layer is implemented.

In this context, the main contributions of the work are argued to consist of: formulation and implementation of a deep reinforcement learning (DRL) control methodology for a open hardware mobile robotic platform using standardized software and communication components; testing and evaluation of the control performance of the DRL technique in a dedicated simulation environment and through implementation on the physical platform in a real scenario.

The rest of the paper is structured as follows. Section II presents the methodology of our work which includes both a theoretical background of the reinforcement learning technique, combined with deep neural network architecture training for parameter prediction, and a description of the open hardware mobile robotic platform used for the experiments. The in-depth implementation and analysis of results are described in Section III. The focus is on the step-wise implementation of the RL methodology, the heuristics used for the improvement of the control performance and the evaluation in both simulation and real scenarios. Section IV concludes the paper and lists potential relevant improvements for future work.

# 2 Methodology

## 2.1 Reinforcement Learning

Reinforcement learning (RL) can be applied on general learning problems that optimize a metric in a sequential way. Thus, reinforcement learning is suited for optimal control and operation in robotic systems. It has close ties with statistics, optimization, game theory etc. and can be used in many scientific scenarios. (Li (2022)). RL is built by implementing a logic policy by simulating different case studies in which an agent optimizes the cost function. The positive actions are rewarded while the ones that have no benefit to the global goal are penalized. The objective of the optimization problem is to minimize the cost function in order to produce a desirable control policy, reducing to:

$$G = R_0 + R_1 + ... + R_k = \sum_{t=0}^{\infty} R_t \tag{1}$$

Where $G$ is the total reward at a moment of time. In a practical example, the infinite horizon of steps becomes a finite one since we want the algorithm to run for a fixed number of steps. The cost function can be written as $R_t = \gamma^t r_t$, where $\gamma^t$ is the discount factor.

$$\pi : A \times S \rightarrow [0,1]; \pi(a|s) = P(a_t = a|s_t = s) \tag{2}$$

Function $\pi$ represents the logic in which the probability $P$ will implement the action $a$ as a state $s$. According to Peng et al (2020) and Han (2018) and introducing two new performance evaluation functions, the state evaluation function, described in equation 3, and the action evaluation function described in equation 4, we can say that the critical network is trained according to Bellman model described in equation 5, and the actor network is updated according to Bellman model described in equation 6.

$$V_\pi(s) = E_\pi\{G|s = s_t\} = E_\pi\{R_t|s = s_t\} + E_\pi\{\gamma V_\pi(s)|s = s_{t+1}\} \tag{3}$$

where $E\{\}$ is the statistical averaging operator, $E_\pi\{R_t|s = s_t\}$ is the reward of state $s$ at the current $t$ or "immediate" time, denoted by $R_{im}$, and $E_\pi\{\gamma V_\pi(s)|s = s_{t+1}\}$ is the expected reward at the immediate next step, scaled by the discount factor $\gamma$,

$$Q_\pi(s,a) = E_\pi\{R_t|s = s_t, a = a_t\} + E_\pi\{\gamma Q_\pi(s,a)|s = s_{t+1}, a = a_{t+1}\} \tag{4}$$

with the notations mentioned in 3,

$$V_\pi^*(s) = \sum_{a \in A} \pi(a|s)(R_{im} + \gamma \sum_{s' \in S} P(s \rightarrow s', a) V_\pi^*(s')) \tag{5}$$

where $P(s \rightarrow s', a)$ is the probability of reaching state $s'$ from the current state $s$ by action/transition $a$, and $V_\pi^*(s)$ is the optimal value of the state evaluation function, considering the optimal $\pi(a|s)$ policy found,

$$Q_\pi^*(s,a) = R_{im} + \gamma \sum_{s' \in S} P(s \rightarrow s', a) \sum_{a' \in A} \pi(a'|s') Q_\pi^*(s', a') \tag{6}$$

with the same notations as in the equation 5.

Figure 1 illustrates how the RL algorithm must contain the two main components: the agent and the training environment and also the interactions between the network and the hardware equipment.

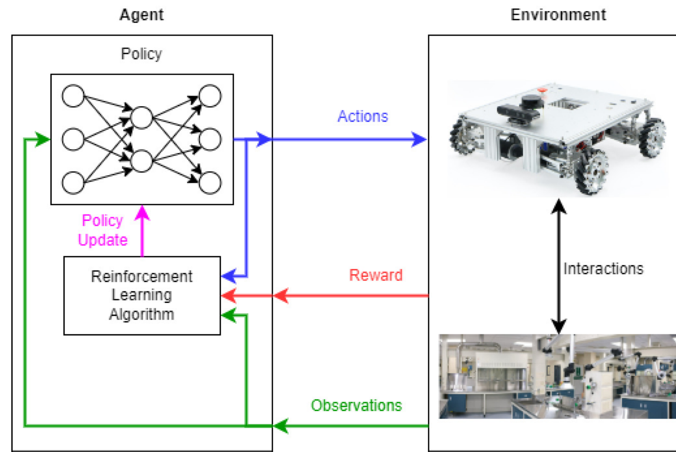



Fig. 1: Reinforcement Learning Architecture

## 2.2 Open Hardware Mobile Robotic Platform

This section presents the design of the robotic platform used for the study and the implementation of the deep reinforcement algorithm. The mobile robotic platform consists of:

- Omnidirectional wheels: based on the complex geometrical arrangement of them, the platform has a high degree of freedom, allowing for complex movements;

- Suspension system: perturbations caused by the vibration of the platform are reduced in order to precisely position to the desired location;
- Lidar: used for scanning in order to accurate create a 3D map of the surrounding environment;
- Depth camera: used for complex task as object identification or spatial orienting; in combination with the integrated NVIDIA Jetson module, the platform is capable of handling some of the latest and most complex optimization scenarios;
- Drive system: the mobile robot it is equipment with two drivers needed by the motors for their synchronization.

In addition to the physical equipment, a complex software architecture runs in parallel. ROS, or Robot Operating System, is the open-source programming language used for mobile robot programming. Three main component have been determined to compose an robot system: the perception of the surrounding, the logic used for calculations and the output used by the physical equipment, as shown in Figure 2. In order to evaluate the environment the mobile platform is equipped multiple sensors, including the depth camera enabling a large field of possible applications. In the logic part of the structure, either C++ or Python programming languages can be used, each with its one advantages and disadvantages. In general, a mobile robot uses DC or stepper motors to execute movement commands. In our case, the robot is using 4 DC motors to move. The operation of the DC motors is done by the integrated CANOpen module that receives the computed commands from the NVIDIA board and transforms them into usable information for the drivers. A summary of the main relevant technical specifications of the mobile robotic platform components is listed in Table I.

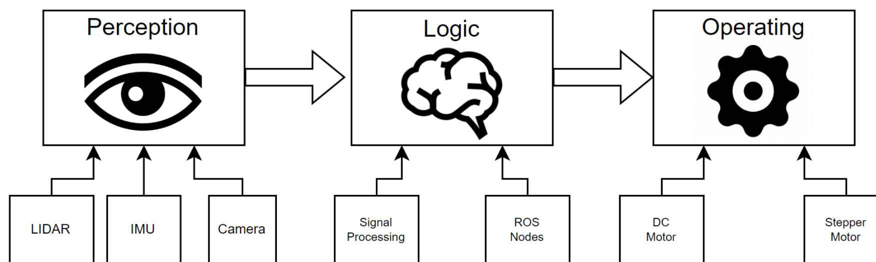



Fig. 2: Conceptual control system pipeline

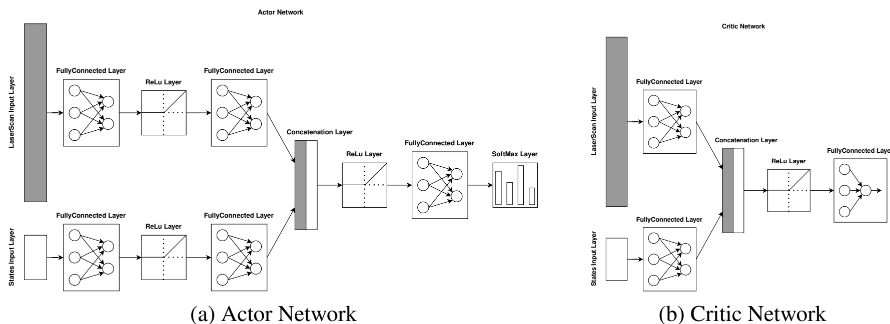
| Component | Model | Specification |
|---|---|---|
| Main board | NVIDIA Jetson Tegra X2 | Maxwell graphics processing unit and ARM A57 processor |
| Lidar | RP LIDAR A2M8 360 | Resolution: 0.5mm-1.5m at a maximum range: 12m |
| Camera | ASTRA PRO Depth | Distance: 0.6m-8m 1280 x 720 @30fps |
| Motor | MD60 100W DC | Speed:175rpm/67rpm / /power: 100W |
| Wheel | Mecanum omnidirectional | Weight: 700g // Rolls number: 16 |
| Driver | DFR0601 Motor Driver | Motor type: Brushed DC |
| Screen | SPI OLED LCD | Resolution: 128px x 64px |

Table 1: List of the hardware components

## 3 Results

### *3.1 Implementation*

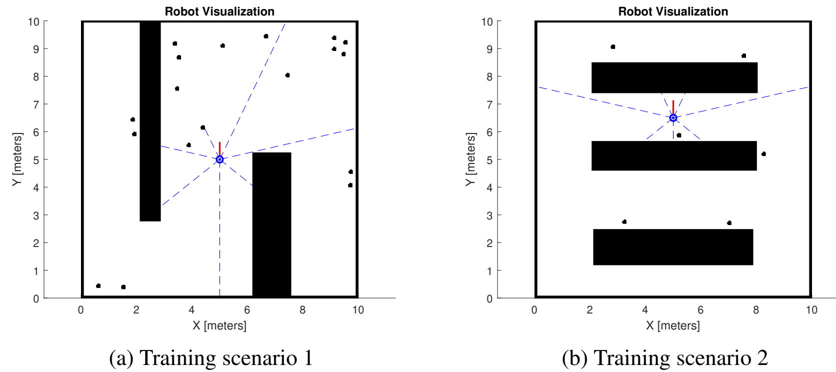
The first step in developing our application is to build a neural network that will represent the brain of the robot. Using MATLAB Deep Network Designer we were able to create an architecture for both networks, the actor (Figure 3a) and the critic (Figure 3b). The input data to those two networks is the same and it is distributed on two channels and each channel will serve for a different purpose. The first channel will use all the data needed to avoid the obstacles meaning the data received from the LiDAR sensor available as an vector of 720 values representing the distance in meters from the robot to the obstacle. The second channel serve for guiding the robot to the destination point and it uses the information about the current state of the robot (current position, current velocity, target position and distance to the target) also available as an vector of 12 values. To help the agent develop a policy based on minimizing the distance between the robot and the target, we chose to also feed the data from the last 3 sample times.




(a) Actor Network (b) Critic Network

Fig. 3: Agent Neural Networks

After successfully building the network, we needed to create a simulation where the agent could start the training. Having access to the MATLAB Reinforcement Learning Toolbox, we created a function based environment to have maximum flexibility in parametrization of training scenarios. The simulated agent is controlled by the neural network and it can interact with the simulated environment just like the real robot could, reading data and moving around according to the inverse kinematics of the real robot. We created more scenarios for the robot to train (Figure 4), each scenario representing a different occupancy grid map and obstacles scattered around.




(a) Training scenario 1 (b) Training scenario 2

Fig. 4: Examples of training scenarios

Last step before starting the training is represented by the parametrization of the learning algorithm. We can set all common parameters of a learning algorithm like the learning rate, the gradient threshold, the discount factor and even the algorithm used for training. In this application we chose the Adam (Adaptive Movement Estimation) algorithm using a exploration policy for the agent training with a sample time of 0.05 seconds, the learning rate is set to $10^{-4}$ and the discount factor is set to 0.99.

To accelerate the training process, we used the parallel training feature available in MATLAB. After activating this option, the learning process will create agents up to the NVIDIA multiprocessor count value; in our case we had 6 available processors. When using the parallel training we need to choose between the synchronous or asynchronous training. The synchronous training will make the agents pause their execution until all others are finished. This option is often used for a gradient-based parallelization where the main process updates the actor and critic weights according to the results of all agents. The asynchronous training uses the experiences sent by each agent to update the weights and as soon as a parallel process ends its episode it will receive a new set of updated parameters to start a new episode. We used the asynchronous method to speed up the training process because in our application, an

agent ends its episode if it reaches the destination or if it hits an obstacle. At first, it is not a problem for the agents to wait for each other to finish their episode since the chances of hitting an obstacle are higher than reaching the destination, but thinking about the future, when the agents end up making their way to the destination more and more often, it will be a waste if an agent hits an obstacle early and it is forced to wait for the others to finish.

For the mathematical modelling of the desired behavior we used the branched reward function shown in the equations 7 and 8. Each term is assigned an experimentally determined weight to encourage the good behaviors like ”keeping a safe distance from the objects” and ”approaching the target”. Another way to help the agent develop a good policy is to feed a positive constant reward for reaching the target and a negative reward for hitting the obstacles.

$$R_t = Af_t^2 - 2Ar_t^2 + 0.5min(scans) + 0.1[dist(pos_t, target_t)]^{-1} \tag{7}$$

$$R_t = \begin{cases} 1 & for\ dist(pos_t, target_t) < 0.1[m] \\ -10 & for\ min(scans) < 0.3[m] \end{cases} \tag{8}$$

Once the training process begins, the environment we built will show the progress of the agents highlighting the reward accumulated by an agent on each episode, the average reward and the estimate of the discounted long-term reward Q0. We can also stop the training process if something goes wrong or if the agents does not make any progress, using this graphical interface. The training algorithm allows us to set the flags to stop the process automatically, like the average reward or even the episode reward exceeds some value. Another useful flag we can set is the condition to auto-save an agent based on the average reward, or the episode count, which will save all agents after that set value.

Table 2: Training results

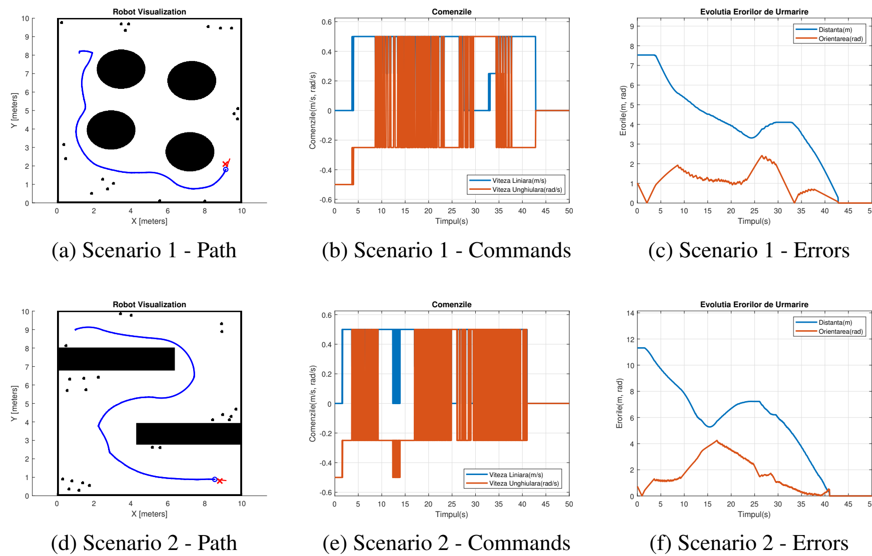
| Episode | Ep. Reward | Avg. Reward | Ep. Q0 | Elapsed Time |
|---|---|---|---|---|
| 2500 | -112.13 | -2392.29 | -24.32 | 2h |
| 5000 | -216.77 | -1107.12 | -33.18 | 4.5h |
| 7500 | 304.19 | -974.02 | -19.12 | 7h |
| 10000 | 417.71 | -1109.16 | -42.51 | 9.2h |

To deploy the network on the real-world robot, we needed to adapt the MATLAB environment to communicate with the ROS environment. The ROS Master needs to have access to the raw data of the sensors, meaning the main process needs to run directly on the robot, but the embedded system does not support the latest versions of MATLAB and that means we need the robot to send the ROS messages with the data from the sensors to the external station running the neural network and send back the ROS messages containing the commands for the movement. Using the MATLAB ROS toolbox and Robotics System Toolbox we were able to adapt

our application for the real-world robot deployment. As long as the two stations are in the same wireless network, the ROS Master and the MATLAB ROS package can exchange messages without any trouble. Because our simulation was build around the real-world robot model, we had no problem adapting the code for the ROS environment. Instead of reading the data from the simulated environment we created the subscribers to access this information from ROS messages and instead of using the inverse kinematics to update the position of the agent in the simulation we created the ROS message that will be sent directly to the robot to execute the real action.

## *3.2 Evaluation*

As mentioned before, we used the parallel computation to run the training simulations. At the end of each training session we obtain the evolution graph where we can see how the agent started to accumulate more and more reward, meaning the policy developed becomes more accurate and the agent behavior is approaching the one we want. To improve our network, we saved the agents with the most prominent result and used them as the starting point of the new training sessions.




(a) Scenario 1 - Path (b) Scenario 1 - Commands (c) Scenario 1 - Errors

(d) Scenario 2 - Path (e) Scenario 2 - Commands (f) Scenario 2 - Errors

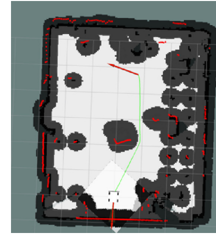
Fig. 5: Validation experiments on simulator

The final results are shown in the figure 5 where we mention that the validation scenarios are different from the training scenarios. We can see that even with no pre-

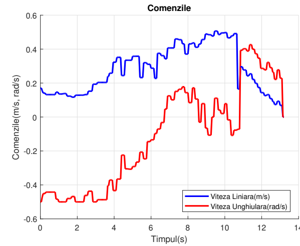
vious experiences the agent we obtained can find a way to the destination, keeping a safe distance from the obstacles scattered around the map.

The first position of the agent in each of the two scenarios was set to look at the upper left corner of the map. We see that the first action of the robot was to rotate until it would look in the direction of the target then proceed to search for the path to reach the destination. Every time it gets too close to the obstacles the policy developed chooses to stop in place and move around the walls until it finds a clear path to the objective. When the robot arrives at the destination the simulation stops and, in real-world, the robot receives a ”stop” command.

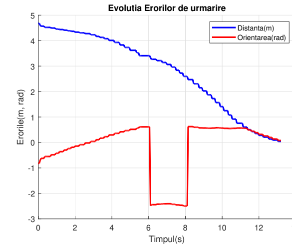
(a) Initial state scenario 1



(b) Final state scenario 1




(c) Received commands scenario 1

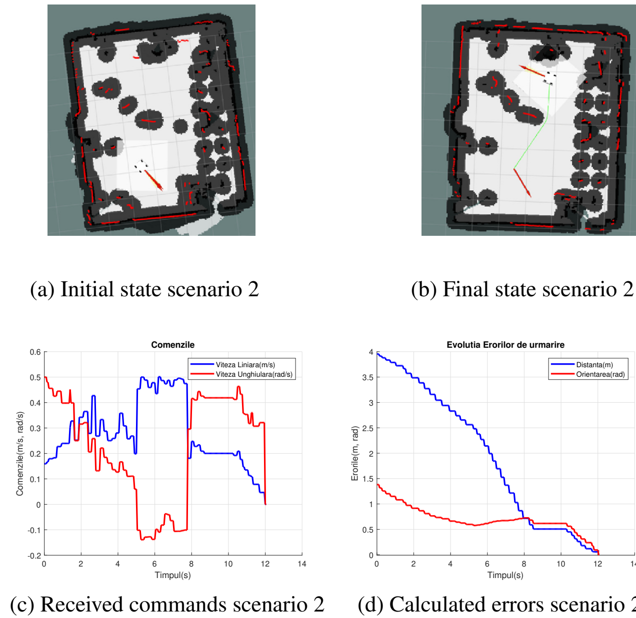



(d) Calculated errors scenario 1

Fig. 6: Validation experiment 1 on real-world environment

With this final agent we started testing in real-world environment and the results are shown in the figure 6 and figure 7 along with the graphs for the executed commands and the graphs of the tracking errors. The commands are sent with a frequency of $20Hz$ and the data is also received at $20Hz$. As we can see from these two sets of experiments, the agent is able to find a path to the destination even in the real environment. The sudden jump in the 6d is due to the change of orientation from $180^o$ to $-179.9^o$ when it is moving backwards.

The wheels of the real robot are subjected to friction and at the same time the system gains inertia when it moves in relation to the simulator where the agent moves at a constant speed. These effects can be seen in figure 6c and figure 7c. The




(a) Initial state scenario 2

(b) Final state scenario 2

(c) Received commands scenario 2

(d) Calculated errors scenario 2

Fig. 7: Validation experiment 2 on real-world environment

goal of the robot is set manually at a laptop where we run the robot visualization program RViz included in ROS packages. It is interesting to see how the agent reacts to the real-life scenarios, since the LiDAR sensor has an error dispersion of about $1cm$. Even if the environment is no longer as accurate as on the simulator, the agent still manages to comply with its policy, keeping a distance of about 10cm from the nearest obstacle and looking for a way to the set target.

## 4 Conclusion

The article presented a reinforcement learning approach to train a fully connected neural network for the control of an open hardware mobile robotic platform. The implementation includes the control design, communication between the MATLAB environment using ROS functions for integration with the robot hardware. Evaluation of the results was performed in a dedicated simulation environment as well as through implementation on the hardware robot system in a laboratory scenario.

Future work will be dedicated to comparing the DRL scheme to other classical (PID) and intelligent control methodologies such as genetic algorithms, other nature-inspired heuristics, fuzzy logic. The potential for deploying an online training

algorithm directly on the robotic platform using the onboard computing resources will be explored. This will enable the robotic edge computing paradigm with or without a online cloud support system. This should allow the continuous improvement of the control performance based real-time acquisition and perception of the changing environment.

**Acknowledgements** Financial support from the Competitiveness Operational Program 2014-2020, Action 1.1.3: Creating synergies with RDI actions of the EU's HORIZON 2020 framework program and other international RDI programs, MySMIS Code 108792, project acronym "UPB4H", financed by contract: 250/11.05.2020 is gratefully acknowledged.

## References

Choi J, Park K, Kim M, Seok S (2019) Deep reinforcement learning of navigation in a complex and crowded environment with a limited field of view. In: 2019 International Conference on Robotics and Automation (ICRA), pp 5993–6000, DOI 10.1109/ICRA.2019.8793979

Fu Y, Jha DK, Zhang Z, Yuan Z, Ray A (2019) Neural network-based learning from demonstration of an autonomous ground robot. Machines 7(2):24, DOI 10.3390/machines7020024, URL http://dx.doi.org/10.3390/machines7020024

Han X (2018) A mathematical introduction to reinforcement learning

Li Y (2022) Deep reinforcement learning: Opportunities and challenges. arXiv preprint arXiv:220211296

Luchian RA, Rosioru S, Stamatescu I, Fagarasan I, Stamatescu G (2021a) Enabling industrial motion control through iiot multi-agent communication. In: IECON 2021 – 47th Annual Conference of the IEEE Industrial Electronics Society

Luchian RA, Stamatescu G, Stamatescu I, Fagarasan I, Popescu D (2021b) Iiot decentralized system monitoring for smart industry applications. In: 2021 29th Mediterranean Conference on Control and Automation (MED), pp 1161–1166

Peng Y, Zhang X, Jiang Y, Xu X, Liu J (2020) Leader-follower formation control for indoor wheeled robots via dual heuristic programming. In: 2020 3rd International Conference on Unmanned Systems (ICUS), pp 600–605, DOI 10.1109/ICUS50048.2020.9274823

Rosioru S, Mihai V, Neghina M, Craciunean D, Stamatescu G (2022) Prosim in the cloud: Remote automation training platform with virtualized infrastructure. Applied Sciences 12(6):3038

Shabbir J, Anwer T (2018) A survey of deep learning techniques for mobile robot applications. CoRR abs/1803.07608, 1803.07608

Wang B (2021) Path planning of mobile robot based on a algorithm. In: 2021 IEEE International Conference on Electronic Technology, Communication and Information (ICETCI), pp 524–528, DOI 10.1109/ICETCI53161.2021.9563354