Discrete Deep Reinforcement Learning for Mapless Navigation

Enrico Marchesini and Alessandro Farinelli

Abstract— **Our goal is to investigate whether discrete state space algorithms are a viable solution to continuous alternatives for mapless navigation. To this end we present an approach based on Double Deep Q-Network and employ parallel asynchronous training and a multi-batch Priority Experience Replay to reduce the training time. Experiments show that our method trains faster and outperforms both the continuous Deep Deterministic Policy Gradient and Proximal Policy Optimization algorithms. Moreover, we train the models in a custom environment built on the recent Unity learning toolkit and show that they can be exported on the TurtleBot3 simulator and to the real robot without further training. Overall our optimized method is 40% faster compared to the original discrete algorithm. This setting significantly reduces the training times with respect to the continuous algorithms, maintaining a similar level of success rate hence being a viable alternative for mapless navigation.**

I. Introduction

Since the groundbreaking release of the discrete action space algorithm Deep Q-Network (DQN) [1], astonishing results have been achieved applying Deep Reinforcement Learning (DRL) in a wide variety of fields. From video games [2] to robotic tasks using mobile robots [3] and manipulators [4]. The main reason behind the use of DRL in robotics is the possibility of adapting to the surrounding environment by generalizing from the training experiences. However, this has a significant cost in terms of training time for the network because robotic applications must cope with the uncertainties of physical hardware (and possibly low-cost sensors), hence they usually require a huge number of trials to achieve reasonable performances. For this reason, reducing training time, while maintaining a good level of performance, is of paramount importance.

Continuous state space control with DRL: (i) Deep Deterministic Policy Gradient (DDPG) [5] and (ii) Proximal Policy Optimization (PPO) [6] has been adopted to cope with the limited capability of Deep Q-Network to deal with physical control tasks, which are characterized by continuous and high dimensional action spaces. However, experimental evaluations demonstrate that these algorithms are time-consuming when compared to discrete action space algorithms (e.g., DQN). Given an exhaustive discrete set of actions sufficient to solve a problem, we show that it is possible to develop optimized and time-efficient discrete action space algorithms as a viable alternative to more computationally expensive solutions based on continuous methods.

In this paper, we focus on the mapless navigation problem, a well-known benchmark in recent DRL literature [7], [8], which aims at navigating the robot towards a random target using local observation and the target position, without a map of the surrounding environment or obstacles.

To address this problem we propose an approach based on Double Deep Q-Network (DDQN) [9] (an improved version of DQN). Our approach is explicitly designed to reduce the training phase and, to this end, we introduce an asynchronous parallel training phase [10] and a Priority Experience Replay (PER) [11] multi-batch memory management (see Section IV). We compare DDQN to DDPG and PPO for mapless navigation and empirically demonstrate that the discrete action space algorithm (i.e., DDQN) offers a time-efficient alternative to other continuous methods.

In more detail, we use the TurtleBot3[1], which is a widely used platform in several previous work focusing on robot navigation [7], [12]. Figure 1 shows our problem architecture and the robotic platform. We consider a sparse 13-dimensional range finder and the target position with respect to the mobile robot coordinate as input for the network. Traditional methods such as SLAM (Simultaneous Localization and Mapping) are based on dense laser range findings. However, the localization and the local cost-map prediction heavily depends on precise dense laser sensor. The laser sensor shipped with the TurtleBot3 is an LDS-01[2], which has an update rate of maximum 5Hz. This low rate causes planning issues using traditional methods (see Section V) where both the scan values and the localization of the robot have to be precise.

Another key factor of recent advances in DRL is the presence of increasingly realistic and complex simulation environments [13], [14], [15], [16], [17]. Along this line, Unity[3] has recently released a toolkit which enables rapid prototyping and development of simulation environments. In this work, we show that it is possible to export a model trained in our Unity environment, to the Robot Operating System (ROS) and furthermore, to the real robot (Figure 1).

Our results show that the optimized DDQN algorithm can be used for mapless navigation as it is able to learn how to safely drive our platform while generalizing on key aspects of the task such as the starting and goal positions, the configuration of the obstacles and the velocity of the robot. Crucially we show that the use of a discrete state space algorithm (DDQN) with our optimizations, significantly reduces the training time when compared to state of the art continuous state space algorithms (DDPG and PPO). Moreover, we show that our trained model can safely reach targets for obstacle

Authors are with the Department of Computer Science, University of Verona, 37135 Verona, Italy name.surname@univr.it

[1] http://www.robotis.us/turtlebot-3/

[2] http://emanual.robotis.com/assets/docs/LDS_Basic_Specification.pdf

[3] https://unity3d.com/company

configurations that the ROS navigation package $movebase$[4] can not handle.

Summarizing, this paper makes the following contribution to the state of the art: (i) we show that our configuration of the mapless navigation problem using a discrete state space algorithm (DDQN) represents a viable alternative for motion planning as it maintains a comparable success rate to DDPG [7] while reducing the training time from 20 hours to less than one hour. (ii) The asynchronous parallel training phase and the multi-batch memory management PER, significantly accelerate the training phase of DDQN, which is $\approx 40\%$ faster compared to the original algorithm. (iii) We show that training the model in the Unity environment is significantly faster than using Gazebo. Crucially, we show that the model trained in Unity can then be used on Gazebo (using the ROS framework) and on the real robotic platform without additional training.

II. Related Work

Deep Reinforcement Learning has been widely applied by the robotics community, due to its ability to solve complex tasks both for mobile platforms (e.g., localization [18], coordination [2]) and for manipulators (e.g., trajectory generation [19]).

The use of Deep Learning approaches for robot navigation is a well-studied and promising area, however the need of a large amount of labeled data is a significant limitation for supervised solutions. To address this limitation, various approaches [20], [12] aim at exploiting human intervention to collect data from the deploy environment, in order to use the policy in a real scenario. However, most of these methods require image processing techniques and hence are computationally expensive [21], [22]. Reality gap [23] is also an important factor to consider when a policy learned in simulation is transferred to reality. Our neural network input, based on sparse laser scans and the target position instead of simulation or real images of the environment, limits the reality gap problem in our evaluation.

To cope with these problems, DRL has been widely adopted for robot navigation [24] and [5] proposes Deep Deterministic Policy Gradients, a DRL approach that can operate also with continuous state spaces and that uses both the policy and the value of the Q-function in the learning process. Recently, other policy-based algorithms, such as Proximal Policy Optimization [6] has been used in motion planning [25].

With the release of the continuous state space algorithms, many works on autonomous navigation exclusively focus on these approaches [7], [24]. However, a proper discretization of the action space may be beneficial in various robotic applications (e.g., trajectory generation for redundant manipulators) and the use of discrete method can result in a significantly shorter training time. In contrast to the training phase of 20 hours in [7], where DDPG was considered, our discrete approach train the network in less than one hour, using the environment built in Unity (see Section V for a detailed analysis).

In this work, we propose a discrete state space algorithm (based on DDQN) for mapless navigation, using a low-cost low-dimensional sparse laser range finder. We compare the performance of this method with the two state of the art continuous state space algorithms (PPO and DDPG). Our experimental evaluation shows that it is preferable to use these discrete approaches when the task does not strictly require a continuous state space. In this way, it is possible to obtain similar behaviors (see Section V) in a very limited amount of training time.

To further reduce training time we implement the considered algorithms in a parallel asynchronous fashion [10] (see Section IV), where the experiences and the back-propagation phase are executed in parallel in multiple instances of the environment, so to improve the sampling efficiency. Furthermore, we introduce a novel multi-batch [26] memory management system extending the original PER [11].

III. Problem Formulation and Background

We can consider the mapless motion planning problem as a decision making process.
At time $t \in [0, T]$, the robot chooses an action a_t according to the state s_t, it then executes the action, reaching a new state s_{t+1} and receiving a reward $r(s_t, a_t)$. Our goal is to maximize the total discounted reward from step t onward, given by $R_t = \sum_{i=t}^{T} \gamma^{i-t} r(s_t, a_t)$ with $0 < \gamma < 1$, where T is the time-step at which the experiment ends and γ is the discount factor.

a) Double Deep Q-Networks: DQN [1] is a learning algorithm based on Q-learning and deep neural networks which for a given state s_t at time t, estimates the value of the state-action pair (s_t, a_t). Given a policy $a_t = \pi(s_t)$ we can define $Q^{\pi}(s_t, a_t) = \mathbb{E}[R_t | s_t, a_t, \pi]$ and recursively compute $Q^{\pi}(s_t, a_t) = \mathbb{E}[r_t + \gamma Q^{\pi}(s_{t+1}, a_{t+1}) | s_t, a_t, \pi]$ using the Bellman equation. A deep neural network (parametrized by θ) estimates the Q-value through Q-learning and the goal is to minimize the loss function:

$$Loss(\theta_t) = [r_t + \gamma \max_a Q(s_{t+1}, a, \theta_t) - Q(s_t, a_t, \theta_t)]^2$$

However, Traditional Deep Q-Network is affected by an overestimation of Q-values and to handle the problem we can use two networks when we compute the Q-value, decoupling the action selection from the target Q value generation [9].

b) Deep Deterministic Policy Gradient: DDPG [5] is also based on the use of neural networks to estimate the Q-value for each state and action pair using a critic network (parametrized by θ^c) and an actor network (parametrized by θ^{π}) to estimate optimal actions. This actor-critic architecture makes it suitable to work with a continuous action space. The critic network is trained in a similar fashion as DQN; the actor network is updated with policy gradient by applying the chain rule:

$$\nabla_{\theta^{\pi}} \pi \approx \mathbb{E}[\nabla_{\theta^{\pi}} Q(s, a|\theta^c)|_{s=s_t, a=\pi(s_t|\theta^{\pi})}] = E[\nabla_a Q(s, a|\theta^c)|_{s=s_t, a=\pi(s_t)} \nabla_{\theta^{\pi}} \pi(s|\theta^{\pi})|_{s=s_t}]$$

[4] http://wiki.ros.org/move_base

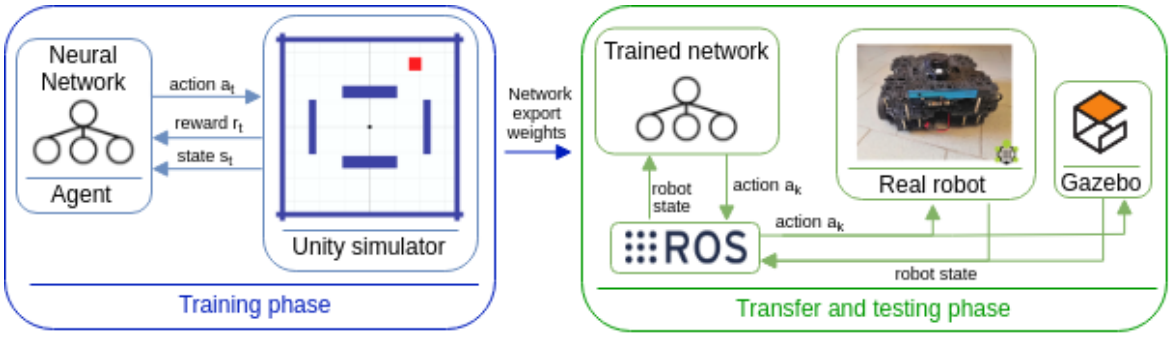

Fig. 1. Overall approach schema. All the considered algorithms train the network in the same fashion.

c) Proximal Policy Optimization:: PPO [6] attempts to control the policy change during learning updates by replacing the constraint of [27] in the optimization problem with a penalty term realized by a clipping in the objective function. In this paper, we consider the PPO implementation embedded in the Unity toolkit and realized by the authors of the framework. Further details of the algorithm can be found in [6].

IV. Methods

This paper aims to propose a discrete method based on DDQN the mapless navigation problem, to empirically demonstrate that discrete Deep Reinforcement Learning can be a viable, faster alternative to existing continuous techniques. Further improvements to speed-up the training phase of the DDQN approach are presented: an asynchronous parallel training phase based on the one introduced in [7], [10]; ii) a multi-batch memory management system to improve the original PER [11]. We also exploit a novel toolkit [28] for the creation of simulation environments, showing: (i) its performance (i.e., training times) compared to the same setting in a traditional robotics simulator; (ii) a porting of the trained models on the official robotic simulator (i.e., Gazebo) and to the real TurtleBot3.

A. Problem Encoding

Given the specifications of the TurtleBot3, we consider an angular velocity of max 90 deg/s provided by the output of our neural networks, and a constant linear velocity. One computation of the network represents an action that is directly mapped into the angular velocity of the robot in the case of discrete algorithms (Figure 2B), or it is used end-to-end in the case of continuous algorithms. For our training, a linear velocity = 0.15 m/s is chosen but, given the capability of generalization of the method, once the network is trained, it is possible to modify this value to obtain different behaviors. The decision-making frequency of the robot is set to 20Hz but, in our evaluation, the DDQN trained model computes on average $\approx$ 60 $actions/s$. This gives us a margin of improvement to further increase the control frequency and deal with faster robot or dynamic obstacles. The laser sensor mounted on the robot is a LDS-01 and it is used to provide a sparse 13-dimensional scan values, which are sampled between -90 and 90 degrees in a fixed angle distribution (Figure 2A). The maximum update rate of this component is 5Hz and it is given by the manufacturer. It is important to notice that this frequency causes localization issues using $amcl$[5] once the trained model is transferred in Gazebo and on the real robot. We show that this inaccuracy strongly influences the behavior of the robot in the testing phase and the continuous models can not handle this uncertainty, while our discrete method achieves better performances (see Section V). Previous works on continuous mapless motion planning [7] does not present this issue, as they employ superior laser sensors, which have higher update rate. Moreover, for our experiment, the laser data does not cover the back of the robot and we assume that backward movements are not allowed. The target goal coordinates are randomly chosen in a range of $(-3.5, 3.5)$ $meters$ (i.e., the size of the training arena in Figure 4D) and this area is guaranteed to be free (i.e., there are no obstacles in this area). To analyze the generalization skills of our models, the configuration of the obstacles is fixed and compact around the robot initial position (see the map depicted in Figure 1, left).

B. Network Architecture

All the considered algorithms share the same input layer structure: 13-sparse laser scans and the target position (a similar setting is used in [7], [25]); these values are normalized in range $(0, 1)$. The target position is expressed in polar coordinates, reporting the distance and angle (in degrees) with respect to our TurtleBot3 agent (Figure 2A). We did explore other encodings for the problem, increasing the number of sparse scan range up to 25 and decoupling the output of the network in two streams to computes different linear and angular velocities. However, the higher complexity of the problem causes longer training times but the success rate was similar. Hence, we preferred a coarse discretization of the action space to maintain a fast training phases.

To determine the size of the hidden layers, we performed tests on different network dimensions [29]. In particular, we performed multiple trials with different random seeds and network sizes. Figure 2 shows the results of the chosen

[5]http://wiki.ros.org/amcl

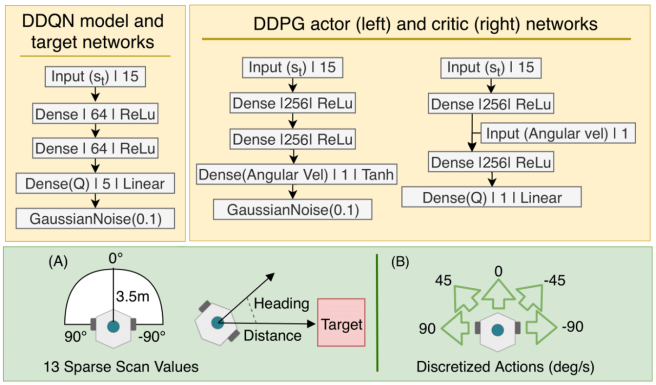

Fig. 2. On top the network structures for DDQN and DDPG (the PPO has a similar structure). Every layer is represented by type, dimension and activation function in Keras nomenclature. (A) Represents the input of the networks. (B) Represents the output layer of the discrete algorithm.

hidden layer and the output layer structure (DDPG and PPO hidden layers are identical).

Figure 2B shows the output layer actions for the discrete algorithm. We use 5 nodes to encode the possible angular velocities for the discrete DDQN ($[-90, -45, 0, 45, 90]$ deg/s) with a linear activation function. For the continuous DDPG and PPO a single node with a hyperbolic tangent activation function is used. This value multiplied by an hyper-parameter, is used end-to-end as angular velocity in a range $(-90, 90)$ deg/s. Section V present the experimental results in terms of: (i) travel distance; (ii) successful attempts; (iii) training times. It is important to notice that given the low update rate of the laser sensor, both $movebase$ and the continuous models can fail in some cases where the discrete model is able to succeed.

We define a successful attempt as a collision-free trajectory that moves the robot in a position that is less than τ cm from the target goal, where τ is an user-defined parameter in the training phase. For our training $\tau = 15cm$. We discover that models trained with a greater τ ($40cm$ in our initial setting), in the testing phase tend to perform a correct trajectory towards the goal, but then they ignore obstacles that are close to the goal and collide with them in order to reach the target.

a) Reward function: In order to get consistent results, the three algorithms share the same reward function r_t. There are three conditions for the reward: two sparse value in case of reaching the target R_{reach} or crashing R_{fail} which terminates an episode (resetting the robot to its starting position), and a dense part used during the travel:

$$r_t = \begin{cases} R_{fail} & \text{if crashes or timeout} \\ R_{reach} & \text{if reaches the target} \\ \omega(d_{t-1} - d_t) & \text{otherwise} \end{cases} \quad (1)$$

where d_{t-1}, d_t indicate the distance between robot and goal at two consecutive time step and ω is a multiplicative factor. In our experiments $R_{reach} = 1$, $R_{fail} = -1$ and $\omega = 10$. The tuning of ω causes different behaviors of the robot: (i) setting $\omega = 15$ when the robots perform actions with angular velocity = 0 we obtain straight movements and smooth trajectories; (ii) maintaining a constant $\omega = 10$ for every action, causes a non-smooth path, because unnecessary actions with angular velocity $\neq 0$ are selected. The attached video shows both these robot behaviors.

C. Double Deep Q-Network Extensions

In this section we present our improvements to the original methods, to accelerate the training phase of the three algorithms:

a) Exploration using Gaussian Noise and Target Soft Update: in the original DDPG implementation [5], the exploration was managed introducing noise in the last fully connected layer of the actor network [30] (without the necessity of tuning the ϵ decay hyper-parameter for a standard ϵ greedy exploration strategy). In addition to this, the target actor and critic network do not update their values every u episodes (where u is an user-defined hyper-parameter), but the update is partially performed at each time step of the network as explained in [31]. In order to balance the differences between the original DDPG/PPO and DDQN, we implemented these two features in our discrete DDQN algorithm. Figure 3A show the difference in terms of mean reward between the original DDQN and the optimized version, applied to our robotic task.

b) Asynchronous Parallel Learning: We separate the experience collecting process to another thread which trains the network [19]. The nature of the Unity engine is to manage and optimize the efficiency of multiple concurrent game threads, so we exploited the original asynchronous fashion of the game engine for the training process. Moreover, we create multiple independent instances of the training environment to obtain parallel agents that collect samples simultaneously. Figure 3B show the effectiveness of the asynchronous parallel DDQN version compared to the original one in a setting with 1, 2 and 4 parallel environments (the effectiveness of this approach on continuous state space algorithms was previously shown in [7], [10]).

c) Memory Management in Priority Experience Replay: given our mixed reward structure, we use a modified version of the PER, introduced by Schaul [11] to accelerate training times. We modified the original priority system by introducing three different batches all managed by the priority system: one for the winning couples (i.e. the ones with R_{reach}) of size 5000, one for the losing ones (i.e. the ones with R_{fail}) of the same size and one with the other experiences of size 20000. The final batch of experiences on which the models are trained is then composed of the same amount of samples taken from the three batches. With this implementation, we are able to train the network with a complete batch of experiences from all the possible behaviors of the reward function. Figure 3C show the difference in terms of success rate (i.e., how many targets the robot reached in the last 100 episodes) between the original PER and this version, applied to the DDQN algorithm in our robotic task.

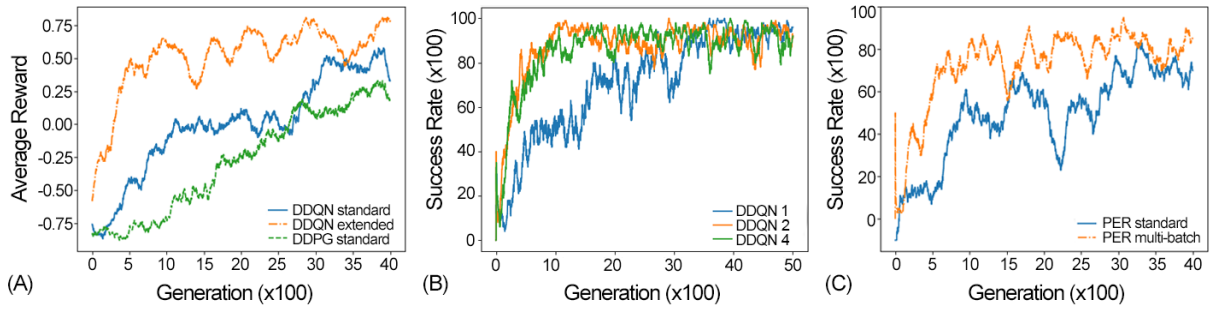

Fig. 3. (A) Average reward for training episode using the standard DDQN, DDPG, and the extended DDQN. (B) Success Rate over the last 100 training episodes between the standard DDQN (i.e., 1 agent) and the asynchronous parallel version (i.e., 2 and 4 parallel agents). (C) Success Rate over the last 100 training episodes between the standard PER and our implementation with multiple batches.

V. Experimental Results

The training environment (Figure 4D) is built using Unity and the learning phase exploits its new Python interface to perform the training of the three algorithms with Keras, a high-level neural networks API. We transferred the trained networks both on a reproduction of the same environment in Gazebo and on the real TurtleBot3, to prove that the model can be ported without the need of additional training. To communicate with the Gazebo environment and the robot, we developed a ROS node, localizing the robot using $amcl$. The collected data are related to training phases performed on a notebook with an Intel Core i7-8550U and a NVIDIA GeForce MX150, using Adam [32] as optimizer with the learning rate set to 0.0003.

Our goal is to to provide a configuration of the mapless navigation problem for the DDQN method, in order to empirically demonstrate that discrete Deep Reinforcement Learning can be a viable, faster alternative to existing continuous algorithms, when the task does not strictly require physics interactions (e.g., manipulation [19]).

To motivate the usage of the novel Unity toolkit, we performed the DDQN and DDPG training phase both on our Unity environment and in the same one built in Gazebo. Figure 4C report the difference in terms of training time between the two software. We tested our implementation of DDQN and DDPG and it is clear that Unity is faster than Gazebo (the training phase on Unity is ≈ 4 time faster than the one on Gazebo).

It is important to notice that our results in Figures 3, 4 show the average of three training runs with initial random seeds. This number of runs is sufficient given that the deviation between different runs of the same algorithm is not significant (on average there is a distance of under 9 points in terms of success rate between the best run and the worst one).

After the training phase, the models were able to learn to navigate exploiting the minimal information in the input layer of the network, generalizing: (i) robot starting position, (ii) target position, (iii) velocity. The laser scan based navigation also allows the TurtleBot3 to be able to navigate in unknown environments with different obstacles, which is a key feature for motion planning. For each algorithm, we consider (i) success rate: how many successful obstacles-free trajectories are performed on a batch of one hundred episodes. (ii) training times, (iii) path length.

Given the improvement in performance given by our optimizations described in Section IV, the following results will consider only training with these optimizations: (i) noisy exploration and target soft update, (ii) asynchronous parallel learning and (iii) multi-batch PER.

A. Quantitative results in the Unity simulation environment

In this section, we compare the different performances obtained by the three algorithms in the Unity environment. This is important to have a repeatable quantitative evaluation of the methodology and to prove that discrete state space algorithms can represent a viable alternative in robotics. Figure 4A shows that the discrete DDQN offers better performances in terms of success rate over the same generation. This result considers an experiment with $\tau = 15cm$, and the training stabilizes at over 95% of success rate after about 3000 iterations that correspond to 50 minutes of training. A crucial evaluation of the viability of discrete algorithms is presented in Figure 4C, where we show that to reach similar performance (i.e., 95% of success rate), continuous algorithms require at least 4 times the training time of DDQN. We did run experiments using DDPG-discrete with the discrete setup on the output layer. However, DDPG-discrete results gave us no improvements in the training time of DPPG. In the case of a relatively simple motion planning scenario, the four neural networks used for the DDPG implementation [5] leads to a very time-consuming training phase. We also test the two families of algorithm in the training environment (Figure 4D) and in a testing environment of size 3.5x$10.5m$ (Figure 4E) that present obstacles that the robot has never seen (e.g., cylinders). These figures report the trajectories generated by two exemplary executions of the trained models aiming at providing a visual representation of the behaviors for the different models and are consistent with our evaluations. Given a very similar behavior of both DDPG and PPO, we show the path of just one continuous approach. In particular, we see that there is a very close correspondence of the robot motion in Figure 4D, but the discrete algorithm offers a shorter path ($\approx 14m$

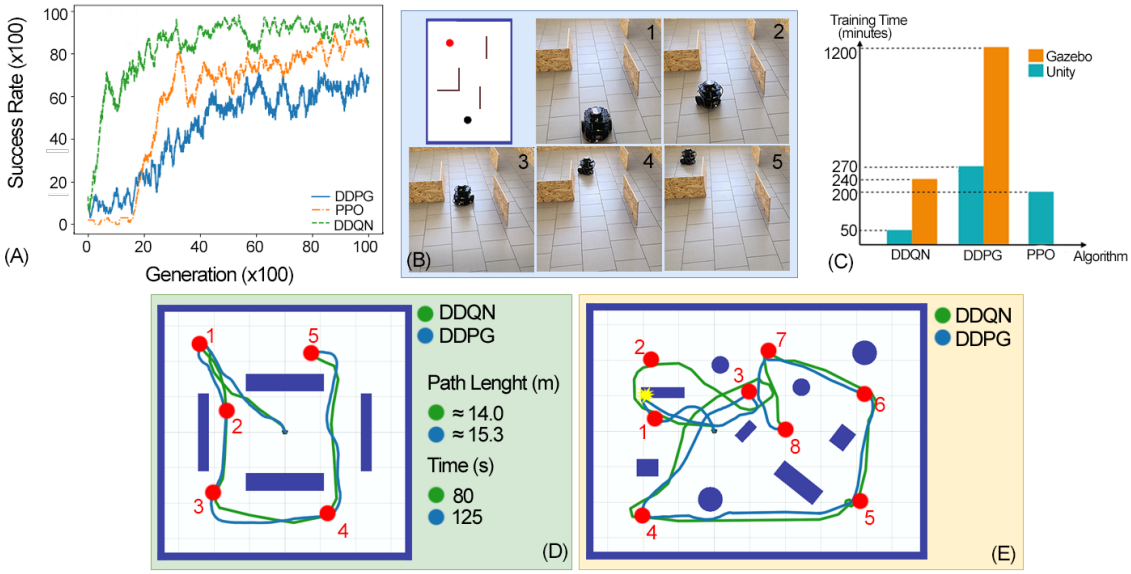

Fig. 4. (A) Average success rate between DDQN-DDPG-PPO. (B) Porting on the real robot. (C) Training times comparison in both Unity and Gazebo (we did not re-implement the PPO of the Unity toolkit for the Gazebo evaluation, given the similar behavior with DDPG). (D) Comparison between the traveled path between DDQN-DDPG in the training environment. (E) Comparison between the traveled path between DDQN-DDPG in the testing environment.

vs $\approx 15.5m$) and takes less time ($80s$ vs $125s$). Figure 4E, shows the path of the two algorithms in a new environment. Here we can see another important aspect of our evaluation, where the DDPG model fails to reach the second target goal (even with a training phase that is 4 times longer); this is caused by the slow update rate of the laser sensor. In our evaluation, we found that the discretization offered by DDQN deals better with the lag introduced by the LDS sensor with respect to DDPG which works in a continuous action space. Notice that this issue is not related to the learning algorithm or our network architecture; the movebase motion planner (which is included in the ROS distribution) also fails for some obstacle configuration.

B. Transfer on Gazebo and on the real platform

We validate the training performed on the Unity toolkit, transferring the training models on Gazebo and on a real TurtleBot3 (Figure 4B). In this simulator and in the real scenario, we used the manufacturer ROS package to retrieve the laser sensor values and $amcl$ to localize the robot. Given the update rate of the TurtleBot3 laser sensor (i.e., 5Hz), the robot localization retrieved with $amcl$ is imprecise. The attached video show that this can lead the traditional $movebase$ motion planner to fail without fine-tuning of its parameters when the discrete trained model succeeds without human intervention.

VI. Conclusions

We propose a discrete DRL approach based on DDQN for the mapless motion planning problem for a TurtleBot3 platform, and we compare such approach with state of the art continuous methods (DDPG and PPO). We extend DDQN by employing asynchronous parallel training and a multi-batch Priority Experience Replay. This extension provides promising results, achieving a success rate of over 95% in a short time (i.e., 50 minutes). Training times of continuous DDPG and PPO are significantly longer (i.e., ≈ 4.5 and ≈ 3.5 hours) and the resultant model performed worse than the discrete one in terms of success rate and path length. Another key element of our approach is the validation of the Unity toolkit as a viable alternative to Gazebo.

Our approach has been evaluated both in simulation environments (Unity and Gazebo) and on the real platform. Overall our work suggests that the use of a discrete action space algorithm is a viable alternative to the continuous ones for mapless navigation tasks, given its performance and short training times. Moreover, our optimizations have further speed up the training phase, managing to train a functional mapless motion planner in less than an hour (when previous work required several hours of training).

It is also important to notice that previous multi-agent work [33] required the realization of a simulator from scratch, given the inefficiency of Gazebo. Unity natively supports multi-agent environments hence paving the way for further research targeting multi-agent scenarios.

As future directions, we intend to investigate the ability of the agent to avoid dynamic obstacles with different velocities, compare the Unity framework performance to recent navigation frameworks [34] and explore different tasks to further investigate the discrete-continuous trade-off.

VII. Acknowledgement

The research has been partially supported by the projects "Dipartimenti di Eccellenza 2018-2022", funded by the Italian Ministry of Education, Universities and Research (MIUR).

REFERENCES

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning," in *NIPS*, 2013.

[2] A. Tampuu, T. Matiisen, D. Kodelja, I. Kuzovkin, K. Korjus, J. Aru, J. Aru, and R. Vicente, "Multiagent cooperation and competition with deep reinforcement learning," *PLOS ONE*, 2017.

[3] L. Tai and M. Liu, "Towards Cognitive Exploration through Deep Reinforcement Learning for Mobile Robots," *CoRR*, 2016.

[4] E. Marchesini, D. Corsi, A. Benfatti, A. Farinelli, and P. Fiorini, "Double Deep Q-Network for Trajectory Generation of a Commercial 7DOF Redundant Manipulator," in *IRC*, 2019.

[5] T. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," in *CoRR*, 2015.

[6] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," *CoRR*, 2017.

[7] L. Tai, G. Paolo, and M. Liu, "Virtual-to-real deep reinforcement learning: Continuous control of mobile robots for mapless navigation," in *IROS*, 2017.

[8] E. Marchesini and A. Farinelli, "Genetic deep reinforcement learning for mapless navigation," in *AAMAS*, 2020.

[9] H. van Hasselt, A. Guez, and D. Silver, "Deep Reinforcement Learning with Double Q-learning," in *AAAI*, 2016.

[10] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous Methods for Deep Reinforcement Learning," *ICML*, 2016.

[11] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized Experience Replay," in *ICLR*, 2016.

[12] L. Tai, S. Li, and M. Liu, "A deep-network solution towards model-less obstacle avoidance," in *IROS*, 2016.

[13] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The Arcade Learning Environment: An Evaluation Platform for General Agents," *JAIR*, 2012.

[14] M. Kempka, M. Wydmuch, G. Runc, J. Toczek, and W. Jaskowski, "ViZDoom: A Doom-based AI Research Platform for Visual Reinforcement Learning," *CoRR*, 2016.

[15] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI Gym," *CoRR*, 2016.

[16] E. Todorov, T. Erez, and Y. Tassa, "MuJoCo: A physics engine for model-based control," *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2012.

[17] M. J. Johnson, K. Hofmann, T. Hutton, and D. Bignell, "The Malmo Platform for Artificial Intelligence Experimentation," in *IJCAI*, 2016.

[18] R. Clark, S. Wang, H. Wen, A. Markham, and N. Trigoni, "VINet: Visual-Inertial Odometry as a Sequence-to-Sequence Learning Problem," *AAAI*, 2017.

[19] S. Gu, E. Holly, T. Lillicrap, and S. Levine, "Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates," in *ICRA*, 2017.

[20] Y. LeCun, U. Muller, J. Ben, E. Cosatto, and B. Flepp, "Off-road Obstacle Avoidance Through End-to-end Learning," in *NIPS*, 2005.

[21] M. Pfeiffer, M. Schaeuble, J. Nieto, R. Siegwart, and C. Cadena, "From perception to decision: A data-driven approach to end-to-end motion planning for autonomous ground robots," in *ICRA*, 2017.

[22] A. Wahid, A. Toshev, M. Fiser, and T. E. Lee, "Long range neural navigation policies for the real world," *CoRR*, 2019.

[23] S. Koos, J. Mouret, and S. Doncieux, "The Transferability Approach: Crossing the Reality Gap in Evolutionary Robotics," *IEEE Transactions on Evolutionary Computation*, 2013.

[24] L. Xie, S. Wang, S. Rosa, A. Markham, and N. Trigoni, "Learning with Training Wheels: Speeding up Training with a Simple Controller for Deep Reinforcement Learning," in *ICRA*, 2018.

[25] P. Long, T. Fanl, X. Liao, W. Liu, H. Zhang, and J. Pan, "Towards Optimally Decentralized Multi-Robot Collision Avoidance via Deep Reinforcement Learning," in *ICRA*, 2018.

[26] S. Han and Y. Sung, "Multi-Batch Experience Replay for Fast Convergence of Continuous Action Control," *CoRR*, 2017.

[27] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, "Trust Region Policy Optimization," *International Conference on Machine Learning*, 2015.

[28] A. Juliani, V. Berges, E. Vckay, Y. Gao, H. Henry, M. Mattar, and D. Lange, "Unity: A General Platform for Intelligent Agents," *CoRR*, 2018.

[29] C.-T. Chen and W.-D. Chang, "A feedforward neural network with function shape autotuning," in *Neural Networks*, 1996.

[30] M. Fortunato, M. G. Azar, B. Piot, J. Menick, I. Osband, A. Graves, V. Mnih, R. Munos, D. Hassabis, O. Pietquin, C. Blundell, and S. Legg, "Noisy Networks for Exploration," *CoRR*, 2017.

[31] R. Fox, A. Pakman, and N. Tishby, "Taming the Noise in Reinforcement Learning via Soft Updates," *AUAI*, 2015.

[32] D. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," in *ICLR*, 2014.

[33] C. Pinciroli, V. Trianni, R. O'Grady, G. Pini, A. Brutschy, M. Brambilla, N. Mathews, E. Ferrante, G. Di Caro, F. Ducatelle, M. Birattari, L. M. Gambardella, and M. Dorigo, "ARGoS: a modular, parallel, multi-engine simulator for multi-robot systems," *Swarm Intelligence*, 2012.

[34] H. L. Chiang, A. Faust, M. Fiser, and A. Francis, "Learning Navigation Behaviors End to End," *RA-L*, 2019.