Double Deep Q-Network for Trajectory Generation of a Commercial 7DOF Redundant Manipulator

Enrico Marchesini, Davide Corsi, Andrea Benfatti, Alessandro Farinelli, Paolo Fiorini
Department of Computer Science
University of Verona, Strada le Grazie 15 - 37134 Verona, Italy
Email: alessandro.farinelli@univr.it

***Abstract*—Artificial Intelligence is constantly growing in all sectors of computer science and industrial automation applications. In several scenarios traditional controllers fail to adapt when a new and unfamiliar environment is presented. Recent studies to solve these problems have seen a shift to solutions using Reinforcement Learning policies. Building upon the recent success of Deep Q-Networks (DQNs), we present a comparison between DQNs and Double Deep Q-Networks (DDQNs) for the training of a commercial seven joint redundant manipulator in a real-time trajectory generation task. Experimental results demonstrate that the DDQN approach is more stable then the DQN one, and further optimization of this algorithm offers faster training and better performances. The proposed approach can also adapt to situations that are unseen during the training process such as different target positions. Moreover, we show that these policies, trained without any visual image of a simulator about the manipulator or the environment, can be directly applied not only to the official visualizer provided by the robot manufacturer, but it can also be applied to the real robot without any further training.**

I. INTRODUCTION

Robot technology advances are astonishing during these years, from the Boston Dynamics Atlas[1] which is capable of running and lift heavy weights to the Spot-Mini[2] that is capable of planning its own route and mapping the surrounding environment. Both these robots could be used in different situations (e.g. transport supplies in emergency cases or reach inaccessible places for humans.) This work will focus on robot manipulators that, despite their impressive capability of adapting to different industrial tasks and situations, will fail to adapt when a new and unknown environment is presented.

In addition to robotics advances, Reinforcement Learning (RL) has received great attention in learning control policies for autonomous agents. In particular, Deep Q-Network is a Q-learning variant able to learn policies exploiting images and scores from video games, as demonstrated by Google DeepMind team [1]. One problem in the DQN algorithm [2] is that the agent tends to overestimate the Q function value: the action with the highest positive error is selected and this value is subsequently propagated further to other states. This leads to positive value overestimation and can have a severe impact on stability of the learning process. A solution to this problem was proposed by Hado van Hasselt [3] and it is called Double Deep Q-Network.

[1]https://www.bostondynamics.com/atlas
[2]https://www.bostondynamics.com/spot-mini

Starting from the success of DQN and his further optimization with DDQN, we present an approach with a sparse reward policy, that does not use any visual image of the simulator about the manipulator in the training process, to train a commercial redundant 7-DOF robotic arm (Panda Franka Emika [3] in Figure 1). In more detail, we use the Panda's Denavit-Hartenberg (D-H) parameters as model, in a trajectory generation task with random target positions. We also present further optimization techniques such as an adaptive discount factor and an adaptation of the Priority Experience Replay [4] for our algorithm to improve performances and reduce training times. Moreover, we discuss a comparison between the two algorithms in terms of performances and training time. Our results demonstrate that both DQN and DDQN can be used to learn policies for a complex task: using these approaches the robotic arm is able to generalize the starting joint configuration, the target goal, the dimension of the path to follow as well as other details such as the step size. We also show that, using the D-H parameters given by the manufacturer as a model for the kinematics of the robotic arm, the generated trajectories can be directly applied to real-world robot without any modification or further training.

Summarizing, the contributions of this work are:

1) Learning complex control policies through the model of the robot with a sparse reward function. In particular we use both DQN and DDQN for trajectory planning, trained without a supervisor. An important feature for both approaches is that the robotic arm is able to generate valid trajectories for unseen target and starting pose of the manipulator.
2) Further optimization: giving the distance as a reward function the target task is way more difficult and time consuming with respect to a generic sparse reward. This kind of reward offers an intuitive solution for further optimization as the Priority Experience Replay, described in Section IV. Moreover, we employed an adaptive discount factor to reduce training times.
3) Transferring from simulation to the actual platform: without any additional training it is possible to use the control policy devised by the trained network both on the official visualizer and on the real robot. In more detail, we train the network by using a model that

[3]https://frankaemika.github.io/docs/

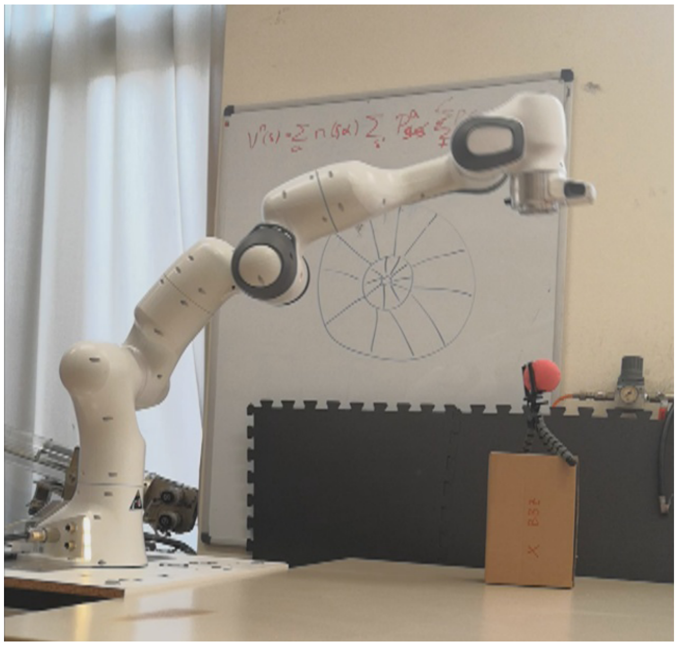
Fig. 1. The Panda robot manipulator used in this work.

incorporates the D-H parameters of the robot (this is done in Keras[4]). We then validate the trajectories in the official visualizer to check for possible errors and finally execute the trajectory on the real platform.

The rest of this paper is structured as follows: Related work are discussed in Section II. Essential notation and existing algorithmic foundations are discussed in Section III. The proposed approach is described in Section IV. Experimental results are shown in Section V. Finally, conclusions are drawn in Section VI.

II. Related Work

Practical real-world applications of Reinforcement Learning [5] typically require significant engineering beyond the learning algorithm [6]: an appropriate representation for the value function must be chosen to achieve training times that are practical for physical hardware [7], and a lot of effort is spent to ensure safety concerns during training [8]. Among the wide variety of RL approaches we focus on Q-Learning which is one of the most widely used model free RL approach. Within this context, when a large state space is involved in the process, it is necessary to use an Artificial Neural Network (ANN) instead of the standard table to store the data [9]. Both DQN [2] and DDQN [3] need a tuning of their training parameters; given the amount of parameters to set, the usage of an ANN for an efficient and effective training can be challenging:

- Discount factor γ: determines the importance of future rewards. It is possible to design an adaptive method that change this parameter based on current state of the learning process [10], in this paper we will extend this approach using the current average reward to change the discount factor.
- Learning rate α: modern optimizer (e.g. Adam [11]) are designed to avoid blocking in local minimum or instability around them. Recent studies show how the learning rate should not remain a fixed value, but it should change during the training [12].
- Reward function: indicates how well an agent is doing in a certain step. This paper presents an extension of the context adaptive reward [13], using a sparse reward function.
- Prioritized Experience Replay (PER): introduced by Tom Schaul [4]. The idea is that some experiences may be more important than others for our training, but might occur less frequently. Given this premise, we adapted PER to work with our sparse reward policy.

[4]https://keras.io/

Learning methods for trajectory generation. State-of-the-art approaches based on learning from images, consider frames where the robot takes various configurations. Such approaches usually work by employing Kohonen Maps [14] learning directly from the images [15]. A similar approach based on possible configurations seemed effective [16] as well as the one based on the study of previously defined mathematical functions [17]. All these approaches need a data-set to perform the training. The generation of a data-set could be challenging given the requirement of a working model or human intervention to show the correct action to the ANN. Some learning methods are done on robot arm such as SCORBOT ER-4u and combine neural networks and D-H parameters of the robot to solve the inverse kinematics [18]. Deep RL approaches have been previously employed on 7-DOF robotic arms [19], however these methodologies consider a fixed target to reach such as the door handle in [19] or only simulated robots. In this work, we focus on a commercial 7-DOF redundant robotic arm and our approach can not only resolve inverse kinematics, but generate a complete trajectory to a random target in the robot work-space.

III. Background

In this section, we present the manipulator RL problem, introduce essential notation, and describe the existing algorithmic foundations on which we built our methods.

The goal is to control an agent, the Panda, that tries to maximize the reward function presented in Section IV. At state s_t in time t, the agent choose an action a_t according to the policy $\pi(a_t|s_t)$ and perform it, 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 time step t onward, given by:

$$R_t = \sum_{i=t}^{T} \gamma^{i-t} r(s_t, a_t) \text{ where } 0 < \gamma < 1.$$

where T is the time-step at which the experiment ends and γ is the discount factor.

The Q-learning algorithm is based on a function $Q : S \times A \rightarrow \mathbf{R}$ that calculates the quality of a coupl[illegible]$_t, a_t)$. At time $t = 0$, Q is initialized to a possibly arbitrary fixed value and at each time t it is updated with the information by the environment. The core of the algorithm is the value iteration update, using the weighted average of the old value and the new information:

$$Q(s_t, a_t) = (1 - \alpha) * Q(s_t, a_t) + \alpha[r_t + \gamma max_a Q(s_{t+1}, a)]$$

where r_t is the reward for the current state s_t, α is the learning rate ($0 < \alpha < 1$).

Our manipulation task is described as a set of joint step and results in a very large environment with millions of different states (see Section IV for detailed analysis); creating and updating a Q-table for such environment would not be efficient at all. In this case the best option is to create an ANN that will approximate the Q-table. This is the core idea behind the DQN and DDQN approaches that we investigate here.

Deep Q-Network. Is based on a neural network that for a given state s at time t, returns a vector of action values $Q(s_t, a, \theta_t)$, where θ are the parameters of the network.

As detailed in [2] two important features of DQNs are the adaptation to the task, thanks to a multi-layer network structure, and the experience replay, which is responsible to train the network using previous experience. Note that the task depends on the network weights and our goal is to minimize the loss functions:

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

Similar to standard Q-Learning, this algorithm is off-policy because it learns the greedy strategy $a = max_a Q(s, a, \theta)$, while following a behaviour distribution that ensures an adequate exploration of the state space. The behavior is selected by an ϵ-greedy strategy that follows the greedy with probability $(1 - \epsilon)$ and selects a random action with probability ϵ. We will adapt the experience replay to our reward function design in Section IV.

However the "max" operator in the standard Q-learning and in DQN uses the same parameters for both selecting and evaluating an action. This makes the approach more likely to select overestimated values, resulting in overoptimistic value estimates. To prevent this, we can split the selection process from the evaluation.

Double Deep Q-Networks. To handle the problem of the overestimation of Q-values we use two networks when we compute the Q-value, decoupling the action selection from the target Q value generation. As described by van Hasselt [3], the weights of the second network θ'_t are replaced with the ones of the target network θ_t for the evaluation of the policy. The update phase of the target network is the same as the DQN approach, and remains a periodic copy of the online network.

IV. METHODS

Here we present the proposed approach in more detail, starting with the setup of the network and the reasons behind these choices. The goal is to train the real Panda to generate an adaptive trajectory to a random target position in its work-space, without the grasping phase.

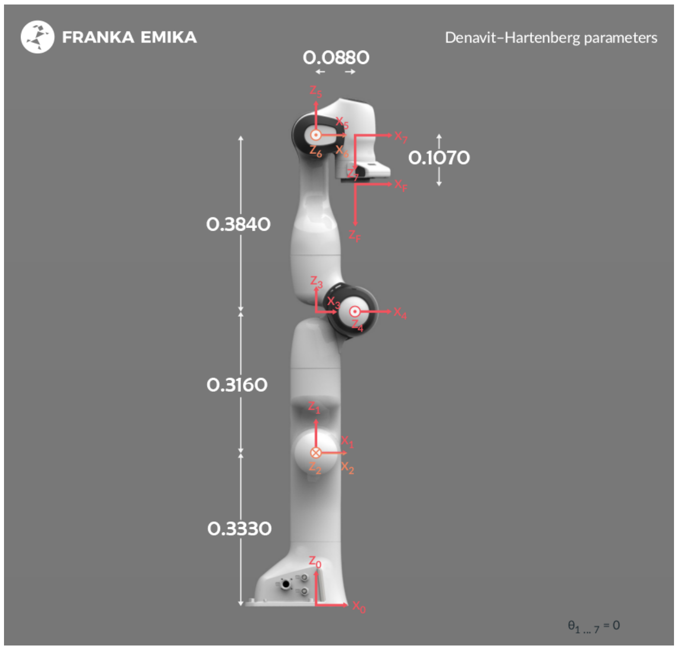

Fig. 2. Pandas kinematic chain. Image from https://www.franka.de/panda.

A. Problem Encoding

Given the kinematic chain of the Panda in Figure 2, we encode the work-space of the manipulator by considering a range of ±120 degree for each of the first six joints, excluding the last one in the wrist, which is responsible of the grasping phase that we do not consider. Moreover a single computation of the network represents an action that move a single joint of ω degrees, where ω is a user-defined parameter in the training phase. For our training, an $\omega = 2$ is chosen but, given the capability of generalization of the agent, once the network is trained, it is possible to generate a trajectory with a different value of ω. We refer to the position of the six joints with an integer vector $q = q_0, \cdots, q_5$, where q_i is the current position of the joint i in degree. The target point is expressed in the work-space of the robot and the coordinates are represented as a triple $< x, y, z > \in \mathbf{R}^3$, where each element is in the interval $(-1, 1)$ that is the size in meters of the works-pace of the Panda robot. Joint positions and target coordinates are the only information provided to the input layer of the neural network which use is justified by the large dimension of the state space:

$$\left(\frac{120}{2}\right)^6 \times x \times y \times z, where x, y, z \in range(-1, 1) \sqsubseteq \mathbf{R} \quad (1)$$

B. ANN structure.

The task to complete is to reach a generalized target position q_f starting from an initial pose $q_0 = [(0, 40, 0, -40, 0, 90)]$. To generalize the starting joint configuration of the Panda, we do not reset the robot to its initial pose q_0 between two consecutive experiment, where for experiment we intend the trajectory generation process for a new random target position. At the end of the training the robot will be able to follow a moving target in its work-space. For both the DQN and DDQN algorithms, the network maintains the same size in terms of hidden layers and input/output layers structure:

- Input layer: 9 nodes, one for each joint and the last three for the target coordinates $< x, y, z >$.
- Output layer: 12 nodes. This layer computes a single action to perform at every computational step. Each joint has 2 nodes to decide if it should move of ω degrees clockwise or anti-clockwise.

To decide the hidden layers size, tests on different dimensions are required [20]. Figure 3 shows the preliminary experiments to choose the size of the network. In particular, it is clear that the network with 3 hidden layers with 64 neurons each, seem to offer the best performances in terms of success over training time, where for success we intend a generated trajectory that led the end-effector of the Panda to the target point with an error τ, where τ is a user-defined parameter in the training phase. For our training, an $\tau = 5cm$ is chosen but as shown in Section V, the unsuccessful trajectory will led the end-effector to the target as well but with a greater error (i.e. every generated trajectory bring the end-effector to the target but some trials have $\tau \pm 2cm$ of error).

It is also important to keep values in the ANN as small as possible: it is not time efficient to keep the original data (i.e. 150 degrees on a particular joint) in these approaches because the neuron weight has to reach those values before it can start to learn (with small changes in the network biases, it is easier to reach a normalized value 0.15 compared to an original value 150). Thus, neuron weights are $\in (-1, 1)$ and all joint degrees are normalized in that range. Given this premise, instead of the standard sigmoid, it is more time efficient to use the hyperbolic tangent activation function:

$$tanh(x) = \frac{2}{1 + e^{-2x}} - 1 \tag{2}$$

Moreover, a linear activation function is used to cope with the variations in the values of the activation function of the output.

C. ANN shaping.

Loss Function. The loss function defines how to penalize the neural network output and how poorly the model behaved and, in the context of back-propagation and neural networks, it also determines the gradients from the final layer to be propagated so the model can actually learn. The used loss function is the standard one presented by Sutton and Barto [21] and shown in Section III:

$$Loss = [(reward + \gamma * Q_{\max}(S_t, A_t)) - Q(S_t, A_t)]^2 \tag{3}$$

Reward function. State-of-the-art approaches in Section II prefer to use a dense reward to give to the algorithm precise and detailed information about each couple (state, action) and to let the network learning at each computational step. This is possible when the training phase is based on the images from the simulator and the policy can consider visual information on one or more side of the simulator to give a reward. One of our goal is to optimize the training time for the trajectory task and the usage of a simulator view is expensive in terms of network size, memory and especially training time.

To avoid such complexity we prefer to use only the manipulator model in the training phase. To this end we could provide as a reward the distance from the end-effector to the target position, normalized for convergence times purposes as explained earlier in this Section (see Eq. 4).

$$reward_1 = e^{\text{-distance}} - 1 \tag{4}$$

Given the chosen work-space of the Panda, equation 4 gives values in the range $(-1, 0]$ so the network obtains a good reward (i.e. 0) only if the robot reaches the correct location

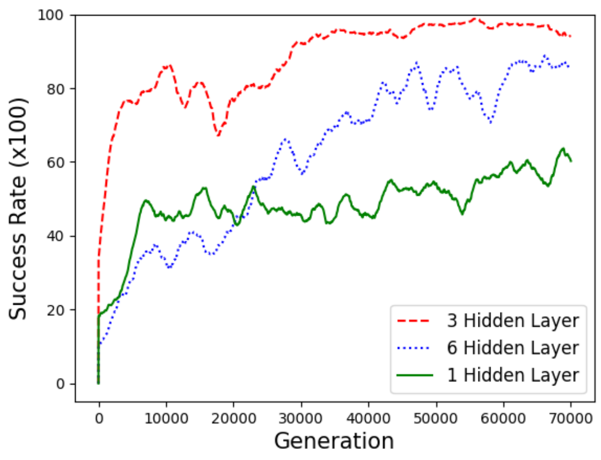

Fig. 3. Preliminary results on ANN sizing test.

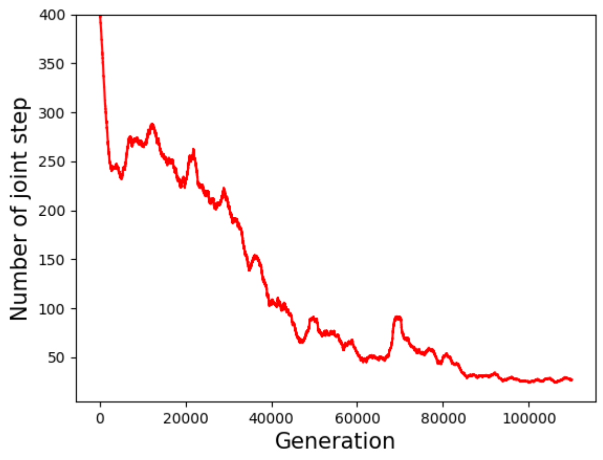

Fig. 4. Number of joint moves with the advancing of the training phase.

within τ. However, when considering the redundancy of the 6-DOF configuration, a pose that is close to the target may not be a good candidate. In fact, given the redundancy of the robot, it can fall into a configuration that never leads to the goal, providing the network with false positive information and ruining the training phase. To avoid this problem we provide the network with a sparse reward:

$$Reward = \begin{cases} -1 & \text{if timeout is reached} \\ 0 & \text{intermediate step} \\ 1 & \text{end-effector has reached the target} \end{cases} \quad (5)$$

This approach simplifies the training process because the network receives information about bad sequences of moves that do not reach the target (e.g., the training ends after a fixed number of computations i.e.,a time-out or when the joints reach configurations that are not feasible), or good sequence of moves that reaches the target with an error ϵ. The Deep Q-learning and the Double Deep Q-learning algorithms maximize the expected long term reward in our task to reach a random spawning target position.

Discount Factor. The discount factor determines the importance of future rewards. State-of-the-art approaches for robot manipulators mentioned in Section II usually set the discount γ to a fixed value, following the early works on DQN [2]. In the specific application of deep Q-learning algorithms for trajectory generation, where the goal is to maximize the future expected reward, we can notice in Figure 4 that the number of step in the trajectory tends to be minimized in the advanced stages of the training, because the network learns to reach the target as fast as possible to get the only positive reward available (i.e. $Reward = 1$). For this reason with the proceeding of the training, we can scale this value, assuming that over time we will need less moves to perform the trajectory. With this method the training success rate results 10-15% better with respect to the one with a fixed discount factor (Figure 5).

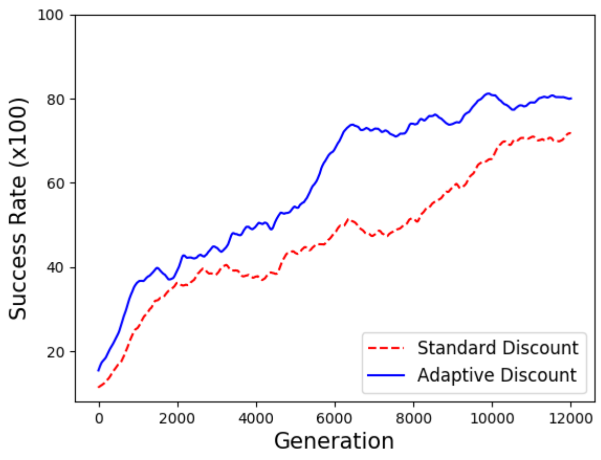

Fig. 5. Overview of the optimization with adaptive discount factor.

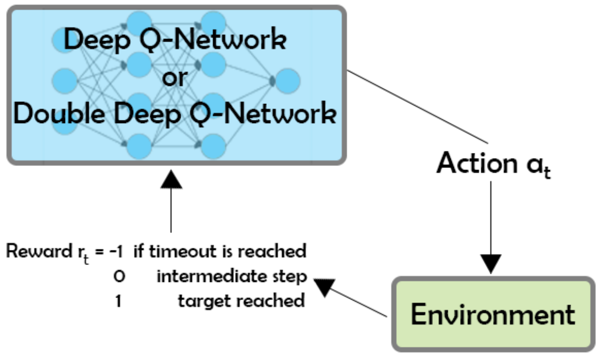

Fig. 6. Overall schema of the used approach.

D. Algorithm

The overall scheme of the proposed approach is presented in Figure 6. It is based on a neural network (or two in the case of DDQN) that is trained by interacting with the environment. In particular, during the training phase, the network calculates the next best action a_t and the model of the environment is updated according to that action. A reward function takes as input the new environment configuration to provide a reward signal r_t to the network. This reward signal is then used to perform the back-propagation process. Once the training is complete, we can use the network in feed-forward as a controller for the robotic arm. Crucially, the same network can be directly applied to control the simulated robot both in Matlab and in the official Panda simulation tool and the real robotic platform.

Priority Experience Replay. Given our sparse reward, we adapt a modify version of the PER, introduced by Schaul [4] to improve the success rate. The idea is that some experiences or actions may be more important than others for our training, but might occur less frequently. Sampling the batch uniformly (i.e. selecting the experiences randomly), the crucial experiences that occur rarely have a very low probability to be selected. In the initial stage of the training, where a trajectory is composed by several moves (i.e., about 100), the network receives a positive reward only in the final state if successful. Standard PER uses a criterion to define the priority of each couple (state, action) of experience. In our case we simplified this priority system introducing two batches: one for the winning couples (i.e. the ones with $Reward = 1$) and one for the losing ones. The final batch of experiences is then composed by the same amount of random samples taken both from the winning and the losing initial batches. Figure 7 shows the difference in performances between the DQN algorithm with and without PER. Notice that in our particular case of sparse reward, it seems that PER is crucial for the goodness of the training phase; the DQN without PER does not learn at all.

Double deep Q-Network. Given the success of the DQN and the DDQN algorithm, we present our adaptation of the DDQN

Algorithm 1 Double Deep Q-learning with Experience Replay

```
    Input: current position of joints and target coordinates
    Output: Q-value (for policy and action selection)
 3: Initialize Experience Replay memory B, $B_{win}$, $B_{lose}$
    Initialize action-value function Q with random weights $\theta$
    Initialize target action-value function Q' with weights $\theta'$
 6: for episode = 1 to M do
      for t = 1 to T do
        Following $\epsilon$-greedy policy, select
        $a_t = \begin{cases} \text{a random action} & \text{with probability } \epsilon \\ argmax_a Q(s_t, a; \theta) & \text{otherwise} \end{cases}$
 9:     Execute action $a_i$ and observe reward $r_t$ and new state $s_{t+1}$
        {Experience replay}
        if $r_t == 1$ then
          Store transition $(s_t, a_t, r_t, s_{t+1})$ in $B_{win}$
12:     else
          Store transition $(s_t, a_t, r_t, s_{t+1})$ in $B_{lose}$
        end if
15:     Build B with random minibatch of transitions $(s_t, a_t, r_t, s_{t+1})$ from $B_{win}$ and $B_{lose}$
        if episode terminates at step $j+1$ then
          $y_j = r_j$
18:     else
          $a_j = argmax(s_{t+1}, a, \theta)$
          $y_j = r_j + Q(s_{t+1}, a_j, \theta')$
21:     end if
        Perform a gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$ w.r.t the network parameter $\theta$
        {Periodic update of target network}
        Every C steps reset $Q' = Q$, i.e. set $\theta' = \theta$
24:   end for
    end for
```

algorithm of van Hasselt [3] that, as shown in Section V, offers the best performances in our scenario.

V. EXPERIMENTAL RESULTS

The training phase was realized using Keras[5], a high-level neural networks API in Python, capable of running on top of TensorFlow[6]. We exported the trained network both on our simulator built with the Robotic Toolbox in Matlab[7] and on the official Franka visualizer. Moreover, we tested the same network on the real robot building a ROS node using rospy[8].

Our goal is to evaluate the proposed approach of trajectory planning in free space generalizing three main parameters: i) target position; ii) starting joint position, iii) step size. In every scenario we consider:

- Success rate: how many correct trajectory are generated on a batch of one hundred episodes. For correct trajectory we mean a sequence of joint positions that led the end-effector to the target position with a tolerance ϵ.
- Average reward: useful to collect data about the trend of the reward function. Both our DQN and DDQN approaches have to maximize this value over the long term.

[5]https://keras.io/
[6]https://www.tensorflow.org/
[7]ttps://it.mathworks.com/products/matlab.htm
[8]http://www.ros.org/

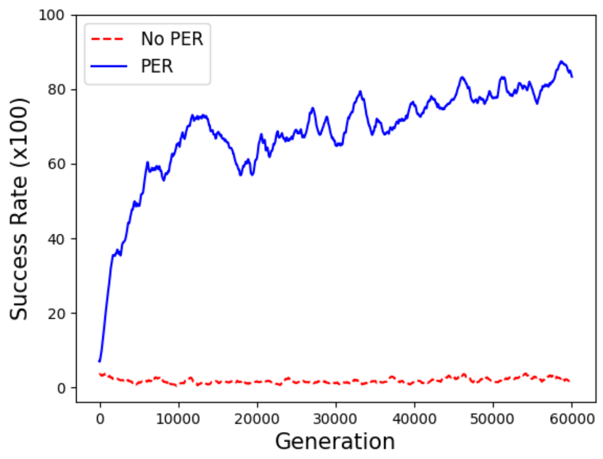

Fig. 7. Experimental evaluation of our DQN algorithm with and without Priority Experience Replay system.

In every following graph, we report on the x-axis the learning episodes where a learning episode is the complete sequence of actions for a single trajectory. A learning episode can end in three cases: when a correct trajectory is generated, when we reach a fixed timeout of actions, or when the joint configuration is unfeasible (i.e., it is outside the Panda works-pace): this last case allowed the network to learn the limit of the Panda work-space. Given the improvement in performances given by our adaptive discount factor γ, the following results will consider only training with adaptive γ.

A. *Quantitative results in the official visualizer*

In this section we compare different version of our approaches using the robot model both in the training and testing phase. This is important to have a repeatable quantitative evaluation of the methodology.

Adaptive Trajectory Planning Training Figure 7 reports results comparing the DQN approach with and without PER. We can notice that the training without PER did not give any usable results so the next experiments consider only training with PER. The method reaches 75% of success rate after about 80000 iterations that corresponds to $6-8$ hours of training. In particular, this shows that the network is able to adapt the trajectory generation to the different positions of the target and to different initial positions of the robotic arm. This is crucial to obtain an approach that can be used in practical applications.

Figure 8 shows that the proposed DDQN optimization offers more stability during the training phase; as described by van Hasselt [3] this lead to better results when performing complex

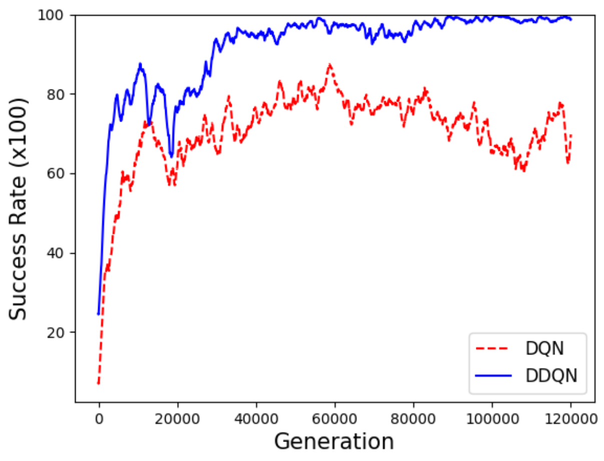

Fig. 8. Comparison of average success rate during the training phase for trajectory generation between DDQN and DQN algorithm with PER (the target position randomly varies in the whole work-space across learning episodes).

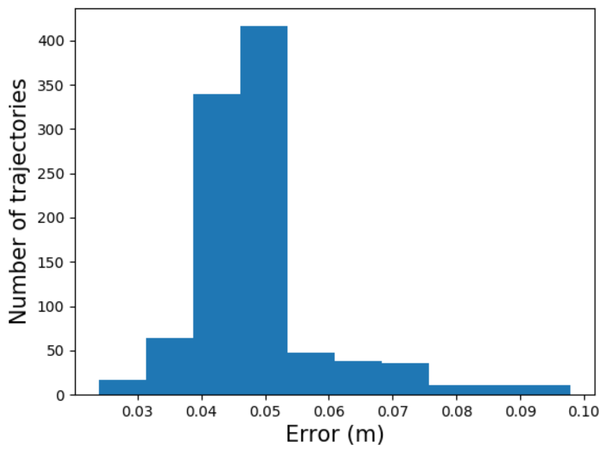

Fig. 9. DDQN performances after the training phase on 1000 random target.

task. The method stabilizes at over 98% of success rate after about 80000 iterations that corresponds to $7 - 9$ hours of training.

Adaptive Trajectory Planning Performances Figure 9 shows the performances of the network with DDQN in the testing phase. As discussed in Section IV, the trained network on 100 different target is able to generate a trajectory towards the random target coordinates in every experiment. The success rate is respected as in the DDQN case 900 trajectories over 1000 test reach the target within an error τ defined *a priori* before the training phase. We also note that even when the network does not reach the target within the specified tolerance (5cm), the error is most of the time lower than 7cm and never higher than 10cm.

B. *Validation on the real platform*

We have validated our approaches on the Panda Franka Emika manipulator (see Figure 10). The test on the actual platform involved the generation of the trajectory to a random target point; video of the comparison between the simulation test and the actual platform test and the source code of the project are available at: https://bitbucket.org/emarche/ddqn-irc/src/master/. A key outcome of the validation is that the movement of the real robot shows a very close correspondence with the simulation environment.

As discussed in Section IV, we used the same trained network on the official visualizer and on the real robot at the same time. The performed actions are almost identical for both the platforms. On the one hand, this is important because it suggests that the other tasks can be executed on the platform in a similar fashion. Moreover, this results confirms that performing the training phase in the simulator and then use the training network on the real robot is possible. As mentioned above, this is crucial not only because we can significantly reduce the time required for training but also because we do not have safety issues related to the need of running a not trained network on the real platform.

VI. CONCLUSIONS

We propose the comparison between the application of a DQN and a DDQN based approaches to plan the trajectories of a commercial 7-DOF robotic manipulator. More in details, the key elements for our approach are the design of the Artificial Neural Network and our further training optimization to encode our robotic manipulation task. In particular, the sparse reward structure, combined with the Priority Experience Replay memory and the use of DDQN provide promising results achieving a success rate of over 98%.

The size and shape of the network has been devised by empirical evaluation and the sparse reward function with PER has been designed to reduce the error in the training process. Moreover, we added specific features and optimization to the network and to the reward function to obtain a faster training.

The training of the network has been performed using Keras and the model of the Panda robot. Our approach has been evaluated both in the simulation environment and on the real platform. Results show that the method is able to generalize from the training phase hence being able to generate trajectories in real time for target points that are not included in the training set.

The training performed on the simulator resulted in successful trajectory generation on the real platform. This is important as it significantly reduces the training time and allows to perform safe training (avoiding possible damages to the robot and surrounding environment).

Overall our work suggests that the use of DDQN methods on commercial robotic platforms are a viable approach to realize high level complex tasks to deal with unexpected changes of the environment.

As future directions, we intend to investigate other optimization approaches to further reduce the training time and

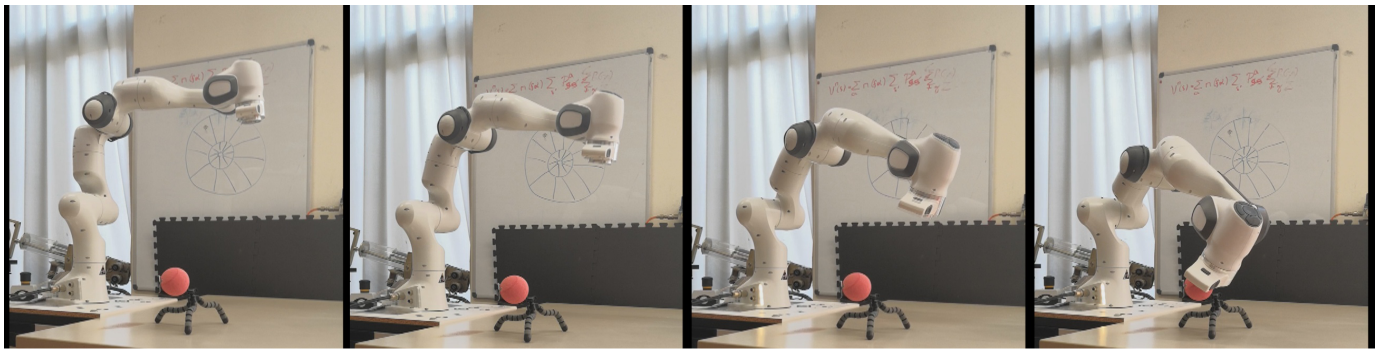

Fig. 10. Test on the real robot.

to exploit synergies with traditional controllers to enhance the success rate of the generated trajectories. Fault management and adaptive obstacle avoidance are also future direction of study using the Double Deep Q-Network algorithm.

References

[1] Y. Liang, M. C. Machado, E. Talvitie, and M. H. Bowling, "State of the art control of atari games using shallow reinforcement learning," *CoRR*, vol. abs/1512.01563, 2015. [Online]. Available: http://arxiv.org/abs/1512.01563

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013.

[3] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," *CoRR*, vol. abs/1509.06461, 2015. [Online]. Available: http://arxiv.org/abs/1509.06461

[4] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *CoRR*, vol. abs/1511.05952, 2015. [Online]. Available: http://arxiv.org/abs/1511.05952

[5] C. Robinson, "Modified reinforcement learning for sequential action behaviors and its application to robotics," in *IEEE SOUTHEASTCON 2014*, March 2014, pp. 1–8.

[6] D. Sabo and X.-H. Yu, "A new pruning algorithm for neural network dimension analysis," in *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, 2008.

[7] N. D. Nguyen, T. Nguyen, and S. Nahavandi, "System design perspective for human-level agents using deep reinforcement learning: A survey," *IEEE Access*, vol. 5, 2017.

[8] R. Y. Putra, S. Kautsar, R. Y. Adhitya, M. Syai'in, N. Rinanto, I. Munadhif, S. T. Sarena, J. Endrasmono, and A. Soeprijanto, "Neural network implementation for invers kinematic model of arm drawing robot," in *2016 International Symposium on Electronics and Smart Devices (ISESD)*, Nov 2016, pp. 153–157.

[9] P. A. Rao, B. N. Kumar, S. Cadabam, and T. Praveena, "Distributed deep reinforcement learning using tensorflow," in *2017 International Conference on Current Trends in Computer, Electrical, Electronics and Communication (CTCEEC)*, 2017.

[10] N. Yoshida, E. Uchibe, and K. Doya, "Reinforcement learning with state-dependent discount factor," 2013.

[11] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014. [Online]. Available: http://arxiv.org/abs/1412.6980

[12] W. An, H. Wang, Y. Zhang, and Q. Dai, "Exponential decay sine wave learning rate for fast deep neural network training," 2017.

[13] C. Finn, S. Levine, and P. Abbeel, "Guided cost learning: Deep inverse optimal control via policy optimization," *CoRR*, 2016.

[14] S. Cavalcanti and O. Santana, "Self-learning in the inverse kinematics of robotic arm," 2017.

[15] R. Szab and A. Gontean, "Robotic arm joint recognition in space using a neural network trained with pattern matching in labwindows/cvi with vision development module," 2016.

[16] O. F. Alcin, F. Ucar, and D. Korkmaz, "Extreme learning machine based robotic arm modeling," 2016.

[17] C. Stanton, A. Bogdanovych, and E. Ratanasena, "Teleoperation of a humanoid robot using full-body motion capture, example movements, and machine learning," 2012.

[18] R. R. Kumar and P. Chand, "Inverse kinematics solution for trajectory tracking using artificial neural networks for scorbot er-4u," in *2015 6th International Conference on Automation, Robotics and Applications (ICARA)*, Feb 2015, pp. 364–369.

[19] S. Gu, E. Holly, T. Lillicrap, and S. Levine, "Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, May 2017, pp. 3389–3396.

[20] C.-T. Chen and W.-D. Chang, "A feedforward neural network with function shape autotuning," *Neural Networks*, vol. 9, no. 4, pp. 627 – 641, 1996.

[21] R. S. Sutton and A. G. Barto, "Reinforcement learning: An introduction," *The MIT Press*.