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: enrico.marchesini@univr.it

***Abstract*—Recent studies to solve industrial automation applications where a new and unfamiliar environment is presented 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 DQN. Moreover, we show that these policies can be directly applied to the official visualizer provided by the robot manufacturer and to the real robot without any further training.**

I. INTRODUCTION AND RELATED WORK

Robot manipulators are used for several different industrial tasks. However, despite the widespread use of these platforms, the approaches to control them usually can not adapt to new and unknown environments.

Reinforcement Learning (RL) has received great attention in learning control policies for autonomous agents and several approaches have been applied to a wide variety of robotic platforms (including manipulators) with the aim of realizing flexible and adaptive control policies. One of the most popular method for RL in complex robotic system is the Deep Q-Network algorithm (DQN) [1]. However, such approach tends to overestimate the Q function value: the action with the highest positive error is selected and this value is propagated further to other states. A solution to this problem has been recently proposed by Hado van Hasselt [2] and it is called Double Deep Q-Network.

We present an approach with a sparse reward policy to train the commercial redundant 7-DOF robotic arm Panda 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 adaptation of the Priority Experience Replay (PER) [3] for our algorithm to improve performances and reduce training times.

II. METHODS

Our goal is to train the Panda to generate a real-time trajectory to reach a random target position. To generalize the starting pose of the Panda, we do not reset the robot to an initial configuration between two consecutive experiment, where for experiment we intend the trajectory generation process for a new random target. For both the DQN and DDQN algorithms, the network maintains the same size in terms of hidden layers (3 with 64 neurons each) and input/output layers structure. The input layer has 9 nodes, one 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), and the last three for the target coordinates $< x, y, z >$. The output layer has 12 nodes and computes a single action to perform at every computational step. As in the literature, each joint has 2 nodes to decide if it should move of ω degrees clockwise or anti-clockwise.

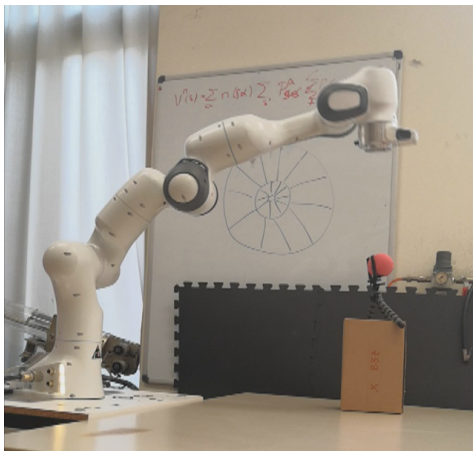

Fig. 1. The Panda robot manipulator used in this work.

We encode the work-space of the manipulator by considering a range of ± 120 degree for each of the first six joints.

Algorithm. The proposed approach is based on a neural network (or two in the case of DDQN [2]) 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 and to select important experiences for the PER implementation. 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 Panda in the official simulation tool and the real robotic platform.

Reward structure. The loss function defines how to penalize the output and how poorly the model behaved. It also determines the gradients from the final layer to be propagated so the model can actually learn. We used a standard loss function [4]:

$$Loss = [(reward + \gamma * Q_{\max}(S_t, A_t)) - Q(S_t, A_t)]^2 \quad (1)$$

Considering the 6-DOF, a pose that is close to the target may not be a good candidate. In fact, given the redundancy of the robot, there might be configurations that are close to the goals but can not reach the desired final configuration. Giving a high reward to such configurations would provide the network with false positive information and ruining the training phase. To avoid this problem we designed the following sparse reward function:

$$Reward = \begin{cases} -1 & \text{if timeout is reached} \\ 0 & \text{intermediate step} \\ 1 & \text{end-effector has reached the target} \end{cases} \quad (2)$$

Both the DQN and DDQN approaches maximize the expected long term reward in our task to reach a random spawning target position.

Priority Experience Replay. The basic idea behind PER is that some experiences or actions may be more important than others for our training, but they might occur less frequently. If we sample the batch uniformly (i.e. selecting the experiences randomly), the crucial experiences that occur rarely have a very low probability of being selected. We then modify the priority system introducing two batches: one for the winning samples (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 drawn from both the batches.

III. EXPERIMENTAL RESULTS

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. We consider how many correct trajectory are generated for a batch of one hundred episodes. In our context, a correct trajectory is a sequence of joint positions that led the end-effector to the target position with a tolerance ϵ.

In the following graph, we report on the x-axis the number of learning episodes, where a learning episode is the complete sequence of actions for a single trajectory. On the y-axis we report the success rate (i.e., the ratio between correct trajectories and all generated trajectories averaged over 100 repetitions). 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.

Figure 2 shows that the proposed DDQN optimization offers more stability during the training phase, and, as described by van Hasselt [2] this lead to better results when performing complex task. The method stabilizes at over 98% of success rate after about 80000 iterations that corresponds to $7-9$ hours of training.

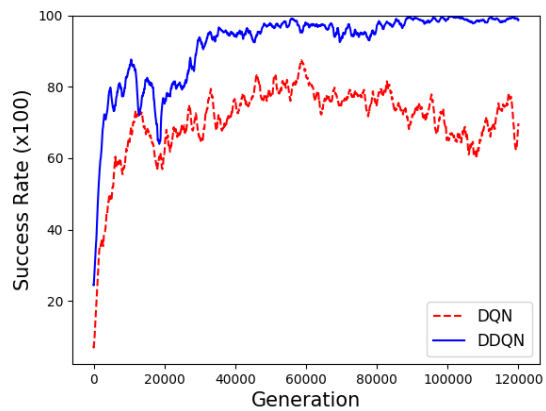

Fig. 2. Comparison of average success rate during the training phase for trajectory generation between DDQN and DQN algorithm (both using PER).

Validation on the real platform We have validated our approaches on the Panda Franka Emika manipulator. 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 as well as 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.

This is important because it suggests that also other tasks can be executed on the platform in a similar fashion. Moreover, these results confirms that performing the training phase in the simulator and then using the trained network on the real robot is possible. 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.

IV. ACKNOWLEDGMENT

This work has been partially supported by the project of the Italian Ministry of Education, Universities and Research (MIUR) "Dipartimenti di Eccellenza 2018-2022".

REFERENCES

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

[2] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," *AAAI*, 2016.

[3] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *ICLR*, 2016.

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