

Reinforcement Learning Control of Robot Manipulator

Lucas Pereira Cotrim
lucas.cotrim@usp.br¹

Marcos Menon José
marcos.jose@usp.br¹

Eduardo Lobo Lustosa Cabral
lcabral@ipen.br²

Abstract

Since the establishment of robotics in industrial applications, industrial robot programming involves the repetitive and time-consuming process of manually specifying a fixed trajectory, which results in machine idle time in terms of production and the necessity of completely reprogramming the robot when changing the task to be executed. The increasing number of robotics applications in unstructured environments requires not only intelligent but also reactive controllers, due to the unpredictability of the environment and safety measures respectively. This paper presents a robotic manipulator control architecture for positioning the end effector in obstacle-filled environments, with no previous knowledge of the obstacles' positions or of the robot arm dynamics. The controller is trained by methods of Reinforcement Learning, appropriate for applications in robotics thanks to their general formulation and capability of learning complex dynamics directly from environmental interaction. The sensory system consists of a camera that captures real-time images of the system's current state. The controller's performance is evaluated through matlab simulations of the KUKA-KR16 industrial robot and the results are compared to traditional methods.

Keywords – Robotics; Artificial Intelligence; Reinforcement Learning; Deep Convolutional Neural Networks; Computational Vision

1 Introduction

The diversity of modern industrial robotics applications requires the emergence of robots with different degrees of autonomy, appropriate for the execution of different tasks, such as welding, machining, assembly and cargo handling. The development of more sophisticated sensors, along with the increasing computational capacity of controllers and advances in the fields of computational vision and artificial intelligence have shifted the field of robotic manipulators: Repetitive and fixed pre-programmed routines have given way to flexible and more reactive controllers, capable of dynamically identifying the orientation of work pieces or learning optimal routines directly from data (ROSEN, 1999).

The objective of this work is to develop a route-planning and positioning control architecture for the KUKA-KR16 robot capable of functioning with minimal human intervention as an alternative to traditional methods of industrial robot programming. The agent is trained, through reinforcement learning methods, over successive interactions with an obstacle-filled environment coupled with an image acquisition system. For training it is only necessary to provide the initial specification of a reward function, which represents

¹Dept. of Mechatronic Engineering, University of São Paulo, São Paulo, Brasil

²Instituto de Pesquisas Energéticas Nucleares (IPEN), São Paulo, Brasil

the quality of actions taken by the agent and guides its exploration. After training, the agent is capable of positioning the robot's end effector in generic positions while avoiding obstacle collision based solely on images from the environment. The main focus of this work is a comparative analysis of common Reinforcement Learning algorithms: episodic REINFORCE and DQN. Different reward functions are tested and the robot's performance is evaluated.

The software chosen was MATLAB because of the native robotic functions that are used to simulate the KUKA-KR16. This saves time programming both the visualization and the computation of the dynamics, besides having all the necessary tools of reinforcement learning and neural networks. While python's reinforcement learning algorithms theoretically run faster, the implementation of complex robot kinematics and its interactions with the environment is simpler with MATLAB.

2 State of the Art

The recent development of Reinforcement Learning means that its practical applications are currently mostly restricted to simulation environments for testing and performance validation. The concept of state and action space exploring inherently requires large amounts of data to be processed and training directly in the real world may lead to accidents. Simulation-based training solves both issues by providing a risk free environment in which the control agent is capable of faster experience acquisition.

Several authors have tried to train RL agents in simulated environments and transfer the resulting model directly to real world applications. James and Johns (2016) were partially successful in the simulation based training and subsequent model transferring of a DQN agent for controlling a seven DOF robot in a cube locating and lifting task. The work environment was structured in a way that maximizes the similarity with the simulation environment in order to enable model transfer. The resulting RL agent was able to correctly locate the cube when applied directly to the real world robot, but subtle differences in the environment prevented it from grabbing and lifting it.

One of the biggest challenges associated with the implementation of Reinforcement Learning in industrial robotics is Low Sampling Efficiency. Most RL algorithms typically require a large volume of training data before optimal policies can be learned and the generation of data in real-world settings is often impractical, as it requires a long idle time. To work around this problem, hand crafted specific initial policies that capture the desired behavior are often used. However, this approach conflicts with the main advantage of RL, i.e. the autonomous learning of various behaviours with minimal human intervention. Shixiang Gu et. al (2016) present an innovative architecture of the DDPG (*Deep Deterministic Policy Gradient*) and NAF (*Normalized Advantage Function*) algorithms in which multiple robots interact with the environment, gain experience according to its current action politics and send data asynchronously to a server, that samples transitions and trains a DQN network. This architecture allows the robot to continue interacting with the environment and collecting state transitions while the DQN parameters are updated, promoting scalability for the inclusion of new robots. The authors validated the proposed architecture in learning the task of opening a door by manipulating robots with seven degrees of freedom and the action policy was obtained without previous demonstrations.

3 Problem Definition

A typical Reinforcement Learning framework consists in three interacting modules: environment, interpreter and agent. The environment's current condition is captured by an interpreter, which encodes it at time t as a state \mathbf{s}_t and assigns a reward value \mathbf{r}_t . The agent, based on the state and reward received by the interpreter, takes an action \mathbf{a}_t , which leads to a state transition according to the system dynamics. The application of this framework to the control of a robot manipulator is exemplified in diagram of figure 1.

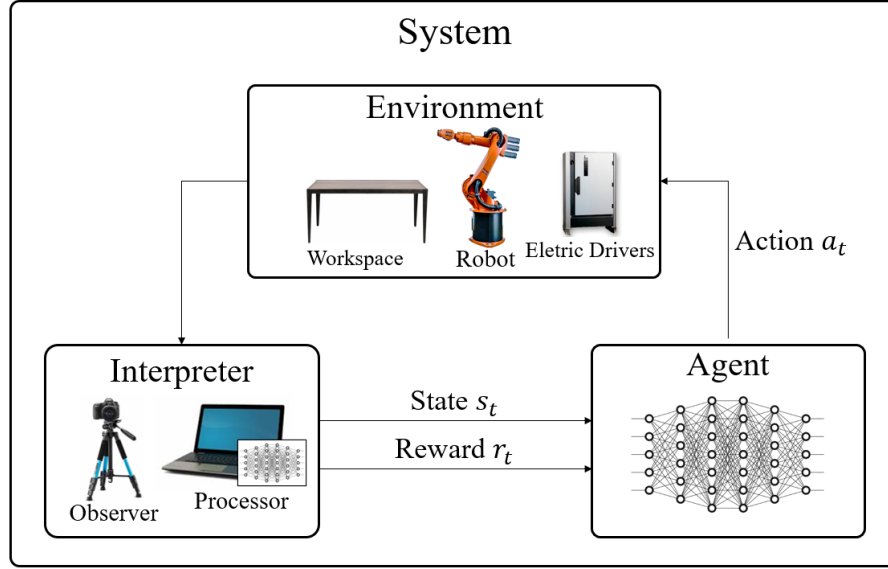


Figure 1: Simplified diagram of a real system environment trained by reinforcement learning.

In order to determine the best algorithm, reward function and hyperparameters, simplified versions of the problem were studied under two main classes of algorithms: Iteration over policy function π_θ and iteration over value function Q_θ . Figure 2 shows the four simplifications considered and the two algorithms implemented for each: Episodic REINFORCE and Q-Learning.

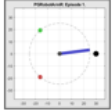
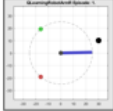
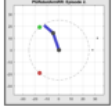
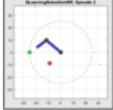
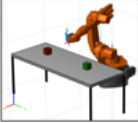
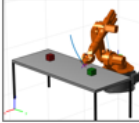
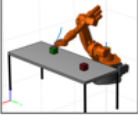
Test Projects	REINFORCE	Q-Learning
1 Degree of Freedom (R)		
2 Degrees of Freedom (RR)		
6 Degrees of Freedom (6R), fixed configuration		DQN 
6 Degrees of Freedom (6R), generic configurations		

Figure 2: Test projects developed for comparative analysis of algorithms.

In the first two simplified projects (1 and 2 DOF robots), the reduced dimension of the state \mathcal{S} and action \mathcal{A} spaces allowed the use of a Q-Learning algorithm known as Q-Tables, in which every possible state-action

combination is directly mapped to $\mathbf{Q}(\mathbf{s}, \mathbf{a})$, which is a function of state \mathbf{s} and action \mathbf{a} , given by a table. However, due to the increased dimensions of the last two projects, the more sophisticated algorithm DQN was implemented, in which a Feed Forward Neural Network approximates the state-action value function $\mathbf{Q}(\mathbf{s}, \mathbf{a})$.

4 Implemented Algorithms

The two of the major algorithms in reinforcement learning are the episodic REINFORCE and DQN, and both are implemented to verify their performance in robotic discrete applications.

4.1 First Implemented Algorithm: Episodic REINFORCE

The first implemented algorithm is the classic action policy function iteration algorithm, episodic REINFORCE proposed by Williams (1992), adapted to the manipulation robot positioning problem according to the pseudo code below. REINFORCE consists in the parameterization of the action policy function $\pi(\mathbf{s})$ as $\pi_\theta(\mathbf{s})$ using any function approximation method, such as neural networks or high-degree polynomials, and training by successive updates of the parameters θ in order to maximize the Performance function $J(\pi_\theta)$, which represents the quality of policy π_θ . Let $\tau = \{s_0, a_0, r_1, s_1, a_1, r_2, \dots, s_{T-1}, a_{T-1}, r_T\}$ be a trajectory generated by a generic policy π_θ , the performance function $J(\pi_\theta)$ can be defined as the expected value of discounted rewards over the trajectory:

$$J(\theta) = \mathbf{E}_{\tau \sim \pi(\tau)}[r(\tau)] \quad (1)$$

where $r(\tau) = \sum_{t=1}^T \gamma^{t-1} r_t = V_\pi(\mathbf{s}_0)$ is equivalent to the value of the initial state $V_\pi(\mathbf{s}_0)$ according to policy π_θ . Since knowledge of the environment and reward is gathered through environmental interaction, the gradient of the performance function $\nabla J(\theta)$ must be approximated by a sufficient number of trajectories N :

$$\nabla J(\theta) \approx \frac{1}{N} \sum_{n=1}^N \sum_{t=0}^{T-1} G_t \frac{\nabla \pi_\theta(\mathbf{s}_t, \mathbf{a}_t)}{\pi_\theta(\mathbf{s}_t, \mathbf{a}_t)} \quad (2)$$

where γ is known as the discount factor and the returns $G_t = \sum_{k=0}^T \gamma^k r_{t+k+1}$ are defined as the sum of discounted rewards from instant t onward.

Algorithm 1: Episodic REINFORCE

- Initialize Robot, setpoint, obstacle, initial state \mathbf{s}_0 and action space \mathcal{A} ;
- Initialize Hyperparameters (bonus and penalties, network size, number of timesteps, trajectories and epochs, discount factor γ and learning rate α);
- Initialize data structure to store epochs;
- Initialize parameterized action policy π_{θ_0} randomly;

Generate N trajectories $\{\tau_n\}_{n=1}^N$ from action policy π_{θ_0} , where
 $\tau_n = \mathbf{S}_0^{(n)}, \mathbf{A}_0^{(n)}, R_0^{(n)}, \dots, \mathbf{S}_{T-1}^{(n)}, \mathbf{A}_{T-1}^{(n)}, R_T^{(n)}$;

Determine returns $\{G_t\}_{t=0}^{T-1}$, where $G_i = \sum_{k=0}^T \gamma^k r_{i+k}$;

Store $\{\tau\}_{n=1}^N$ in EpochBuffer(1);

for $ep \leftarrow 2$ **to** $MaxEpoch$ **do**

Apply Gradient Ascent Method on $J(\theta)$ to get π_{ep} : $\theta_{ep} \leftarrow \theta_{ep-1} + \alpha \frac{1}{N} \sum_{n=1}^N \sum_{t=0}^{T-1} G_t \frac{\nabla \pi_\theta(\mathbf{s}_t, \mathbf{a}_t)}{\pi_\theta(\mathbf{s}_t, \mathbf{a}_t)}$;

Generate N trajectories $\{\tau_n\}_{n=1}^N$ from action policy $\pi_{\theta_{ep}}$, where $\tau_n = \mathbf{S}_0, \mathbf{A}_0, R_0, \dots, \mathbf{S}_{T-1}, \mathbf{A}_{T-1}, R_T$;

Determine returns $\{G_t\}_{t=0}^{T-1}$, where $G_i = \sum_{k=0}^T \gamma^k r_{i+k}$;

Store $\{\tau\}_{n=1}^N$ in EpochBuffer(ep);

end

4.2 Second Algorithm Implemented: DQN

DQN (Deep Q Network) can be seen as a generalization of the simple Q-Learning algorithm known as Q-tables. Rather than directly mapping each state-action pair to a value $Q(\mathbf{s}, \mathbf{a})$ and performing successive iterations on the resulting table, DQN performs the parameterization of the state-action value function $Q_\theta(\mathbf{s}, \mathbf{a})$ as a weighted neural network. The network is initialized arbitrarily with random weights θ , which are updated successively as state transitions $(\mathbf{s}, \mathbf{a}, r, \mathbf{s}')$ are observed by the agent. The DQN network is trained to satisfy the Bellman Equation (BELLMAN, 1967):

$$Q_\theta(\mathbf{s}_t, \mathbf{a}_t) = r(\mathbf{s}_t, \mathbf{a}_t) + \gamma \max_{\mathbf{a}' \in \mathcal{A}} Q_\theta(\mathbf{s}_{t+1}, \mathbf{a}') \quad (3)$$

which associates the value of a state-action pair to the maximum value of subsequent state-action pairs. DQN can be seen as a Supervised Learning Algorithm in which the *target* is non-stationary, as it depends on the $Q_\theta(\mathbf{s}, \mathbf{a})$ function itself, except for terminal states, in which the target is simply the reward $r(\mathbf{s}_t, \mathbf{a}_t)$. This is one of the major difficulties associated with this method and causes its convergence to depend on sufficient exploration of different actions across the entire state space. However, given sufficient exploration, the algorithm is proved to converge to the optimal state-action value function $Q^*(\mathbf{s}, \mathbf{a})$.

In order to improve numerical conditioning and allow for faster convergence, a technique known as Prioritized Experience Replay is implemented, where state transitions are stored in a experience buffer and sampled either randomly or prioritizing states in which the network's error is higher. State transition tuples are defined as $(\mathbf{s}, \mathbf{a}, r, \mathbf{s}', \text{bool}_{term})$ where \mathbf{s} is the system's current state, \mathbf{a} is the action taken by the agent, r the reward obtained and \mathbf{s}' is the subsequent state, in addition, a Boolean variable bool_{term} indicates if the state is terminal.

Algorithm 2: DQN

- Initialize the robot, *setpoint*, *obstacle*, initial state \mathbf{s}_0 and action space \mathcal{A} ;
- Initialize Hyperparameters (bonuses and penalties, network, number of *timesteps*, epochs and transitions at *Buffer*, discount factor γ , learning rate α) and ϵ ;
- Initialize epoch storage structure;
- Initialize parameterized DQN network Q_{θ_0} randomly;

for $ep \leftarrow 1$ **to** $MaxEpoch$ **do**

Initialize state: $\mathbf{s} \leftarrow \mathbf{s}_0$;

Fill *Experience Buffer* with N transitions given current network $Q_{\theta_{ep}}$ and ϵ -Greedy method ;

Sample random *mini-batch* of size N_{batch} from *Experience Buffer*;

for $i \leftarrow 1$ **to** N_{batch} **do**

Read i -th transition: $(\mathbf{s}, \mathbf{a}, r, \mathbf{s}', \text{bool}_{term})$;

if $\text{bool}_{term} == \text{true}$ **then**

$y = r$;

else

$y = r + \gamma \max_{\mathbf{a}' \in \mathcal{A}} Q_{\theta_{ep}}(\mathbf{s}', \mathbf{a}')$;

end

store expected output $q = Q(\mathbf{s}, \mathbf{a})$ and target y ;

end

Applies Gradient Descent to minimize cost function given by $\mathcal{L}(\theta) = \frac{1}{2}(Q_\theta(\mathbf{s}, \mathbf{a}) - y)^2$, that is:

$\theta_{ep+1} \leftarrow \theta_{ep} - \alpha \frac{1}{N_{batch}} \sum_{i=1}^{N_{batch}} \nabla_{\theta} \frac{1}{2}(Q_\theta(\mathbf{s}, \mathbf{a}) - y)^2$;

Clear *Experience Buffer*;

end

5 Reward Functions

Reward function engineering is critical in reinforcement learning applications. The reward function determines the quality of actions taken by the agent and influences not only the policies it is capable of learning but also the algorithm's convergence.

Over the test projects three different reward functions are implemented. In the first two test projects, agents trained with one reward function showed significantly better performance in comparison to the others. As a result, the Kuka projects focused on the implementation of this reward function. The following sections detail their mathematical implementations, key insights and intuition.

5.1 First Reward Function: Absolute Distances

The first reward function considered is inspired in the potential field method for path planning of mobile robots. The reward function depends on the euclidean distances between the end effector, the obstacle and the goal.

$$r_1(\mathbf{s}, \mathbf{a}) = k(r_{setpoint}(\mathbf{s}, \mathbf{a}) + r_{obstacle}(\mathbf{s}, \mathbf{a}) + c) + B_{goal}(\mathbf{s}, \mathbf{a}) + P_{joint\ boundary}(\mathbf{s}, \mathbf{a}) + P_{collision}(\mathbf{s}, \mathbf{a}) \quad (4)$$

where

$$\begin{cases} r_{setpoint} = -k_1 \|\mathbf{p}_{setpoint} - \mathbf{p}'_{ef}\|_2 \\ r_{obstacle} = k_2 \|\mathbf{p}_{obstacle} - \mathbf{p}'_{ef}\|_2 \end{cases} \quad (5)$$

and \mathbf{p}'_{ef} is the end effector's future position after action \mathbf{a} is taken. The remaining terms represent bonuses or penalties given to the agent based on the desired behaviour: B_{goal} is a bonus given when the desired object is reached, $P_{collision}$ is a penalty given when either the table or the red obstacle is hit and $P_{joint\ boundary}$ is a penalty given when one of the robot's joint's limit is reached.

5.2 Second Reward Function: Discrete under Approximation or Distancing

In order to correct problems observed in the first function, such as high magnitude and non-zero average value over all the possible actions at a given state, a second function is tested. The second reward function is dependent on the relative approximation or distancing between the end effector, the obstacle and the goal.

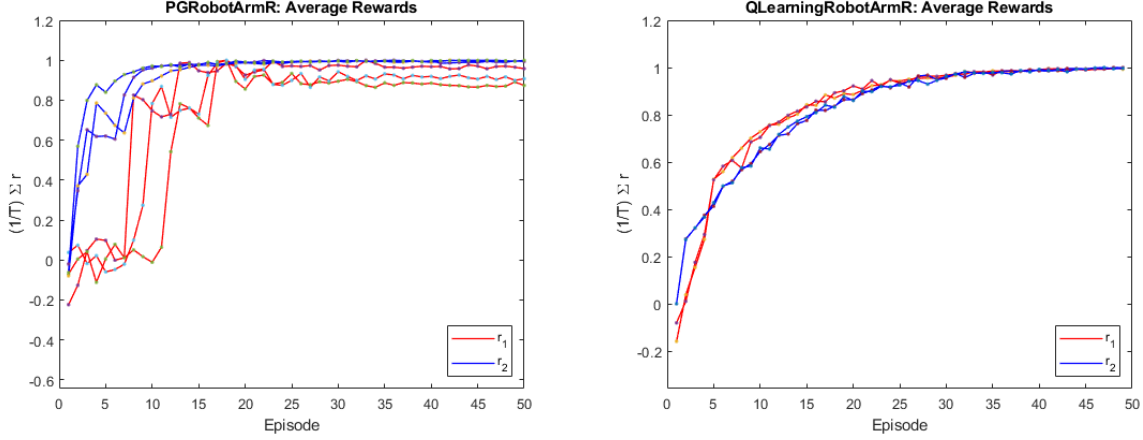
$$r_2(\mathbf{s}, \mathbf{a}) = (k_s r_{setpoint}(\mathbf{s}, \mathbf{a}) + k_o r_{obstacle}(\mathbf{s}, \mathbf{a})) + B_{goal}(\mathbf{s}, \mathbf{a}) + P_{joint\ boundary}(\mathbf{s}, \mathbf{a}) + P_{collision}(\mathbf{s}, \mathbf{a}) \quad (6)$$

where

$$r_{setpoint} = \begin{cases} -1, & \text{if } \|\mathbf{p}_{setpoint} - \mathbf{p}'_{ef}\|_2 > \|\mathbf{p}_{setpoint} - \mathbf{p}_{ef}\|_2 \\ 0, & \text{if } \|\mathbf{p}_{setpoint} - \mathbf{p}'_{ef}\|_2 = \|\mathbf{p}_{setpoint} - \mathbf{p}_{ef}\|_2 \\ 1, & \text{if } \|\mathbf{p}_{setpoint} - \mathbf{p}'_{ef}\|_2 < \|\mathbf{p}_{setpoint} - \mathbf{p}_{ef}\|_2 \end{cases} \quad (7)$$

$$r_{obstacle} = \begin{cases} 0, & \text{if } \|\mathbf{p}_{obstacle} - \mathbf{p}'_{ef}\|_2 > r_{infl} \\ \begin{cases} -1, & \text{if } \|\mathbf{p}_{obstacle} - \mathbf{p}'_{ef}\|_2 < \|\mathbf{p}_{obstacle} - \mathbf{p}_{ef}\|_2 \\ 0, & \text{if } \|\mathbf{p}_{obstacle} - \mathbf{p}'_{ef}\|_2 = \|\mathbf{p}_{obstacle} - \mathbf{p}_{ef}\|_2 \\ 1, & \text{if } \|\mathbf{p}_{obstacle} - \mathbf{p}'_{ef}\|_2 > \|\mathbf{p}_{obstacle} - \mathbf{p}_{ef}\|_2 \end{cases}, & \text{otherwise} \end{cases} \quad (8)$$

The discrete penalties and rewards given in case of collision are the same as defined in $\mathbf{r}_1(\mathbf{s}, \mathbf{a})$, given in eq. (2). Figure 3 illustrates the normalized average reward obtained by a REINFORCE agent for a 1-DOF robot (Test Project 1).



(a) Reward function comparison for REINFORCE agent. (b) Reward function comparison for Q-Learning agent.

Figure 3: Normalized average reward per epoch obtained by (a) REINFORCE and (b) Q-Learning agents for reward functions r_1 (red) and r_2 (blue) on test project 1.

5.3 Third Reward Function: Projection of Displacement Vector

Finally, the third reward function is similar to the second, but the terms $r_{setpoint}$ and $r_{obstacle}$ are no longer limited to $-1, 0$ and 1 , but are given by the projection of the displacement vector $\mathbf{n}_{ef \rightarrow ef'}$ in the directions that point to the goal $\mathbf{n}_{ef \rightarrow setpoint}$ and to the obstacle $\mathbf{n}_{ef \rightarrow obstacle}$.

$$r(\mathbf{s}, \mathbf{a}) = (k_s r_{setpoint}(\mathbf{s}, \mathbf{a}) + k_o r_{obstacle}(\mathbf{s}, \mathbf{a})) + B_{goal}(\mathbf{s}, \mathbf{a}) + P_{joint\ boundary}(\mathbf{s}, \mathbf{a}) + P_{collision}(\mathbf{s}, \mathbf{a}) \quad (9)$$

where

$$\begin{cases} r_{setpoint}(\mathbf{s}, \mathbf{a}) = (\mathbf{n}_{ef \rightarrow ef'} \bullet \mathbf{n}_{ef \rightarrow setpoint}) \\ r_{obstacle}(\mathbf{s}, \mathbf{a}) = \begin{cases} 0, & \text{if } \|\mathbf{p}_{obstacle} - \mathbf{p}'_{ef}\|_2 > r_{infl} \\ -(\mathbf{n}_{ef \rightarrow ef'} \bullet \mathbf{n}_{ef \rightarrow obstacle}), & \text{otherwise} \end{cases} \end{cases} \quad (10)$$

and $B_{goal}(\mathbf{s}, \mathbf{a})$, $P_{joint\ boundary}(\mathbf{s}, \mathbf{a})$ and $P_{collision}(\mathbf{s}, \mathbf{a})$ are defined the same way as in previous reward functions. Figure 4 illustrates a diagram of the third reward function for two different actions taken in the initial state \mathbf{s}_0 . \mathbf{s}'_A and \mathbf{s}'_B are the system's states after the agent has taken actions \mathbf{a}_A and \mathbf{a}_B respectively. The dotted lines represent the vectors that point to the desired position and to the obstacle. Finally, the reward is given by the corresponding projections.

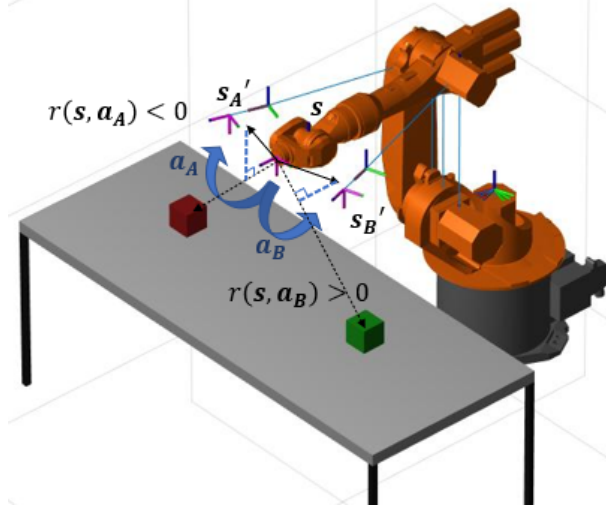
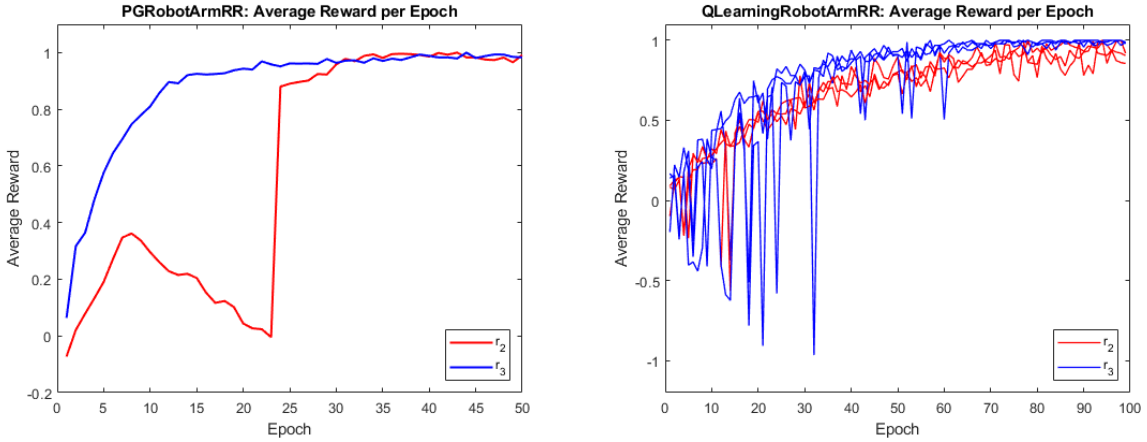


Figure 4: Diagram of the third reward function.

Overall, agents trained with the reward function r_3 showed better performance in comparison to those trained with r_2 . Figure 5 shows the average reward per epoch obtained during training of a 2-DOF robot. The REINFORCE agent trained with reward function r_3 presented significantly better performance in terms of convergence time and stability (Figure 5a) while the Q-Learning agent showed more frequent drops in its learning curve during epochs in which a collision with the obstacle occurred (Figure 5b), which is possibly due to r_3 's priority to direct paths to the goal combined with a goal-obstacle configuration in which a direct path is obstructed.



(a) Reward function comparison for REINFORCE agent. (b) Reward function comparison for Q-Learning agent.

Figure 5: Normalized average reward per epoch obtained by REINFORCE (a) and Q-Learning (b) agents for reward functions r_1 (red) and r_2 (blue) on test project 2.

6 Results

From the partial results obtained during training of a 1 and 2-DOF robot, agents trained with the third reward function presented significantly better performance. As a result, only r_3 is implemented in the last two projects, which focus on a comparative analysis between both classes of algorithms. In this section the

training frameworks and detailed results obtained from the application of both classes of algorithms to the Kuka test projects are presented. The results are followed by a brief comparative analysis and summarized in the end of the section.

A side by side comparison of both algorithms in increasingly more sophisticated applications is valuable as it allows us to focus on the algorithms' foundations, eliminating sources of instability or non-convergence and comparing both in identical settings. Another advantage is the possibility to exploit modular, object oriented program design, since most functions are shared by the various test projects and can be easily adapted to other applications. As shown in figure 2, the test projects are characterized by the simplifications considered: The first two implement 1 and 2 DOF robots, while the last two implement the 6-DOF KUKA KR16 robot, but with initially fixed and then generic configurations of goal and obstacle.

6.1 Test Project 3: KUKA KR16 - Fixed Configuration

After comparing both the reward function and the algorithm on robots with a reduced number of degrees of freedom, a 3-dimensional simulation and visualization environment for the KUKA-KR16 robot was implemented on Matlab through by loading *RigidBodyTree* object representation of the robot based on its urdf file and stl meshes.

In this project, the agent's task is to control the robot's first five degrees of freedom to position its end effector on the fixed goal position (green) while avoiding collision with a known obstacle (red) and the table, which is unknown and is only detectable through interaction. Due to overall better performance observed on agents trained with the third reward function (equation 9) on both algorithms, the following projects implement only r_3 as the reward function and focus on a comparative analysis between algorithms as well as on techniques to overcome the dimensionality issue on real robotics applications.

The State Space is now given by: $\mathcal{S} = \left(\prod_{i=1}^5 \mathcal{S}_i \right) \times \mathcal{R}^3 \times \mathcal{R}^3$, where $\mathcal{S}_i = \{\theta_{i_{inf}} + i\Delta\theta | i = 1, \dots, \frac{\theta_{i_{sup}} - \theta_{i_{inf}}}{\Delta\theta}\}$.

Similarly to previous test projects, the State Space is the combination of possible angular positions for each controllable rotating joint \mathcal{S}_i and all possible Cartesian positions for the goal and the obstacle in three-dimensional space \mathcal{R}^3 . Table 1 indicates Kuka KR16's joint limits and the implemented limits give the table workspace.

Joint	Angular Limits	$[\theta_{i_{inf}}, \theta_{i_{sup}}]$
A1	$[-185^\circ, 185^\circ]$	$[-30^\circ, 40^\circ]$
A2	$[-65^\circ, 125^\circ]$	$[-20^\circ, 40^\circ]$
A3	$[-220^\circ, 64^\circ]$	$[-30^\circ, 50^\circ]$
A4	$[-350^\circ, 350^\circ]$	$[-40^\circ, 40^\circ]$
A5	$[-130^\circ, 130^\circ]$	$[-40^\circ, 40^\circ]$
A6	$[-350^\circ, 350^\circ]$	not controlled

Table 1: Kuka KR16's joint limits and implemented angular limits.

The Action Space is: $\mathcal{A} = \prod_{i=1}^5 \mathcal{A}_i$, where $\mathcal{A}_i = \{-1, 0, 1\}$, which corresponds to all possible positive, neutral and negative increments for all joint actuators. As a result, the number of possible actions is $3^5 = 243$

(equation 11). Figure 6 illustrates the simulation environment.

$$\mathcal{A} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & -1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & -1 \\ 1 & 1 & 1 & -1 & 1 \\ 1 & 1 & 1 & -1 & 0 \\ 1 & 1 & 1 & -1 & -1 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ -1 & -1 & -1 & -1 & 1 \\ -1 & -1 & -1 & -1 & 0 \\ -1 & -1 & -1 & -1 & -1 \end{bmatrix} \quad (11)$$

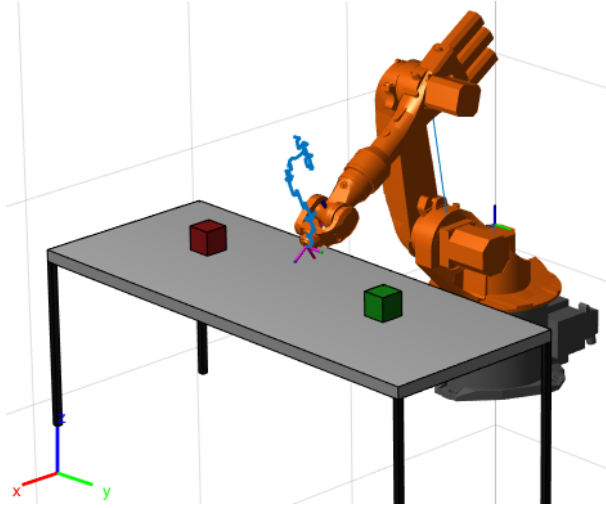


Figure 6: Simulation Environment developed for test projects 3 and 4. Goal and obstacle are represented by green and red cubes and a random trajectory is shown in blue.

6.1.1 Episodic REINFORCE

The algorithm’s generic formulation allowed for relatively simple adaptation to the new project. The policy function $\pi_{\theta}(\mathbf{s}, \mathbf{a})$ neural network complexity was increased in order to allow for the abstraction of more complex policies. A three-layer feedforward network with 104000 trainable parameters was implemented. Table 5 summarizes the project’s hyperparameters.

Parameter	Description	Value
Psetpoint	Goal Position (m)	(1.05, 0.45, 0.75)
Pobstacle	Obstacle Position (m)	(1.05, -0.55, 0.75)
$\Delta\theta$	Minimum Joint Angle Step	1°
α	Learning Rate	0.0005
B_{goal}	Goal Bonus	400
$P_{collision}$	Collision Penalty	-400
$P_{joint\ boundary}$	Joint Limit Penalty	-400
r_{infl}	Obstacle Influence Radius (m)	0.50
$MaxEpoch$	Maximum Number of Training Epochs	50
N	Number of Trajectories per Epoch	30
T	Maximum Number of Actions per Trajectory	70
γ	Discount Factor	0.3
$dim(s)$	State s Dimension	11
$dim(a)$	Action a Dimension	5
$size(\mathcal{A})$	Action Space \mathcal{A} Size	243
$(n_{in}, n_{h1}, n_{h2}, n_{out})$	$\pi_\theta(\mathbf{a} \mathbf{s})$ Network Neurons per Layer	(11, 100, 300, 243)
k_s	Goal Multiplicative Factor	100
k_o	Obstacle Multiplicative Factor	70

Table 2: Variables and Hyperparameters of Episodic REINFORCE implementation of Test Project 3

Similarly to previous test projects, the direct approximation and training of the policy function $\pi_\theta(\mathbf{a}|\mathbf{s})$ yielded a smooth, monotonically increasing average reward curve (figure 13). As opposed to value iteration algorithms, which search for the optimal state-action value function $Q_\theta(s, a)$ and derive the optimal policy by taking the action of most value at each state.

In order to study the agent’s increasing preference for optimal actions during training, the probability distribution of actions in \mathcal{A} given the initial state was plotted for epochs 1, 4, 8 and 12 (figure 14).

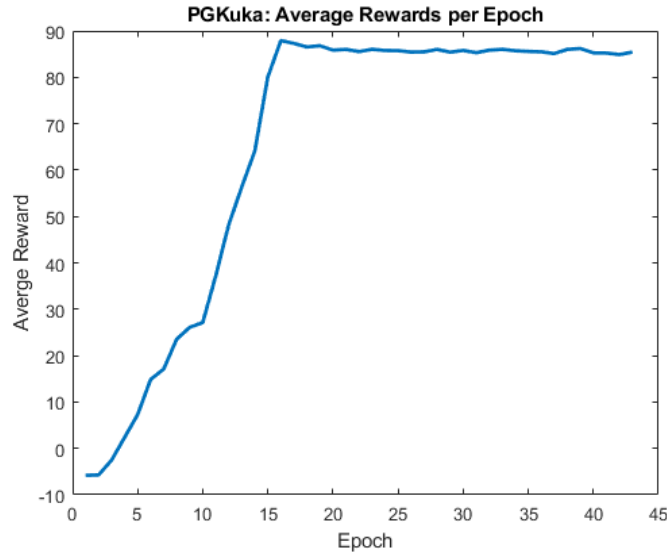


Figure 7: Normalized average reward per epoch obtained by REINFORCE agent trained with reward function r_3 .

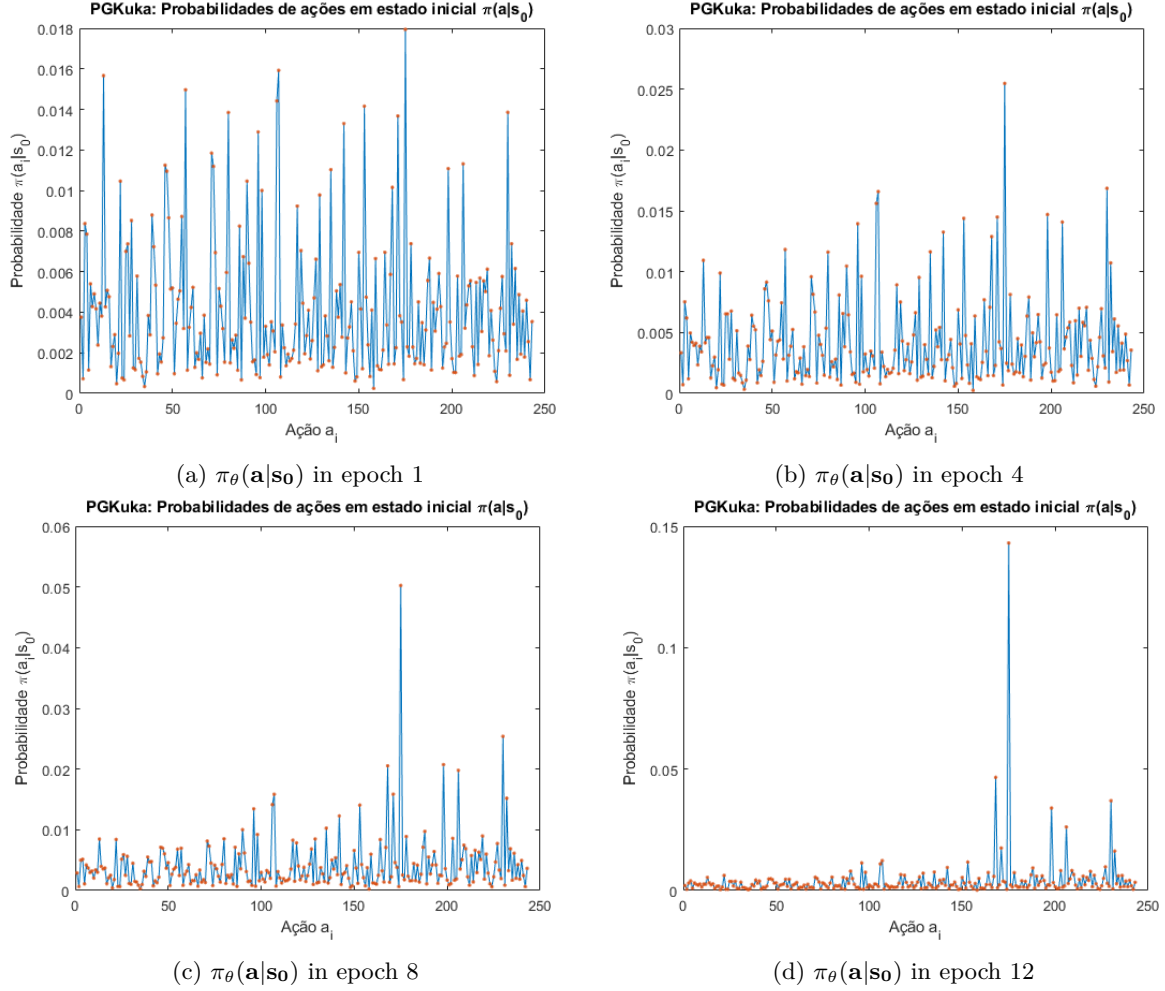


Figure 8: Probability Distribution $\pi_\theta(\mathbf{a}|\mathbf{s}_0)$ over action space \mathcal{A} for initial state \mathbf{s}_0 during training in epochs 1, 4, 8 and 12.

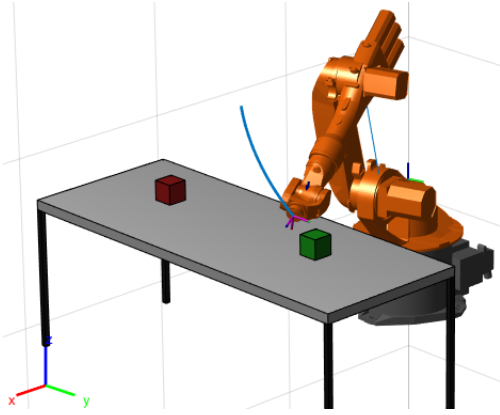
6.1.2 DQN

Due to exponentially increasing state-action space dimension as the number of degrees of freedom increases, a Q-table algorithm is impracticable as a result of memory and computation limitations. In order to overcome the dimensionality issue, the state-action value function $\mathbf{Q}_\theta(\mathbf{s}, \mathbf{a})$ was represented as a Multi-layer Perceptron (MLP). The algorithm's formulation, detailed in section 4.2, consists in applying gradient descent in order to minimize the mean squared error between the network's current output $\mathbf{Q}_\theta(\mathbf{s}, \mathbf{a})$ and the target $r(\mathbf{s}, \mathbf{a}) + \gamma \max_{\mathbf{a}' \in \mathcal{A}} \mathbf{Q}_{\theta_{\text{ep}}}(\mathbf{s}', \mathbf{a}')$, where \mathbf{s}' denotes the state reached after action \mathbf{a} is executed in state \mathbf{s} . Table 6 summarizes the algorithm-specific hyperparameters implemented.

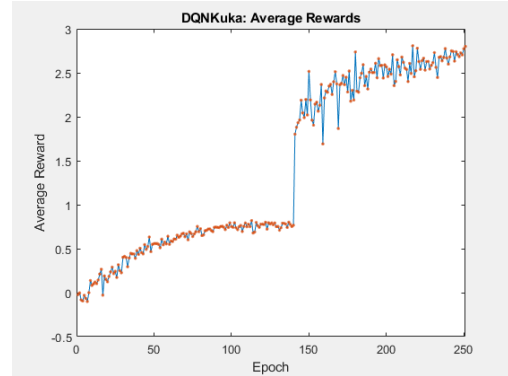
Parameter	Description	Value
α	Learning Rate	0.002
$MaxEpoch$	Maximum Number o Training Epochs	250
$Ntrajs$	Number of Trajectories per Epoch	10
$MiniBatchSize$	Number of Sampled Transitions for Training	200
T	Maximum Number of Actions per Trajectory	70
γ	Discount Factor	0.3
$dim(s)$	State s Dimension	11
$dim(a)$	Action a Dimension	5
$size(\mathcal{A})$	Action Space \mathcal{A} Size	243
$(n_{in}, n_{h1}, n_{h2}, n_{out})$	$\pi_{\theta}(\mathbf{a} \mathbf{s})$ Network Neurons per Layer	(11, 100, 300, 243)

Table 3: Variables and Hyperparameters of DQN implementation on Test Project 2

Figure 15 illustrates the trajectory taken by the DQN agent after convergence and the average reward performance curve during training.



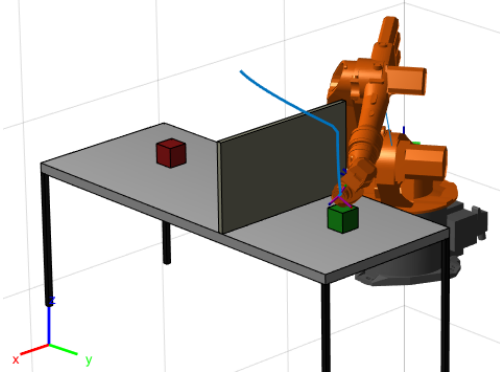
(a) Trajectory taken by DQN agent after training



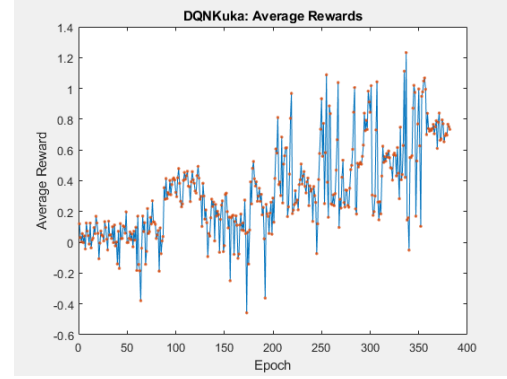
(b) DQN agent performance curve

Figure 9: Optimal trajectory taken by agent (a) and average rewards obtained per epoch during training (b) in DQN implementation on Test Project 3.

In order to analyze the agent's behavior in cases where a direct path to the goal is blocked by an unknown object, a wall was placed between the effector's initial position and the goal. Similarly to table collision, wall collision is incorporated into the state transition function and terminates a trajectory if the robot's end effector is sufficiently close to the wall. A negative reward of $P_{collision}$ is given during collision and the wall's position can only be learned through environmental interaction. Figure 16 illustrates the optimal trajectory found by the agent and the corresponding learning curve during training. As expected, an increased number of epochs was necessary for the abstraction of a more complex behavior, but the DQN agent was able to dodge the wall correctly with no algorithmic changes.



(a) Optimal Trajectory found by DQN agent in environment with unknown blocking obstacle



(b) Average rewards per epoch obtained during training

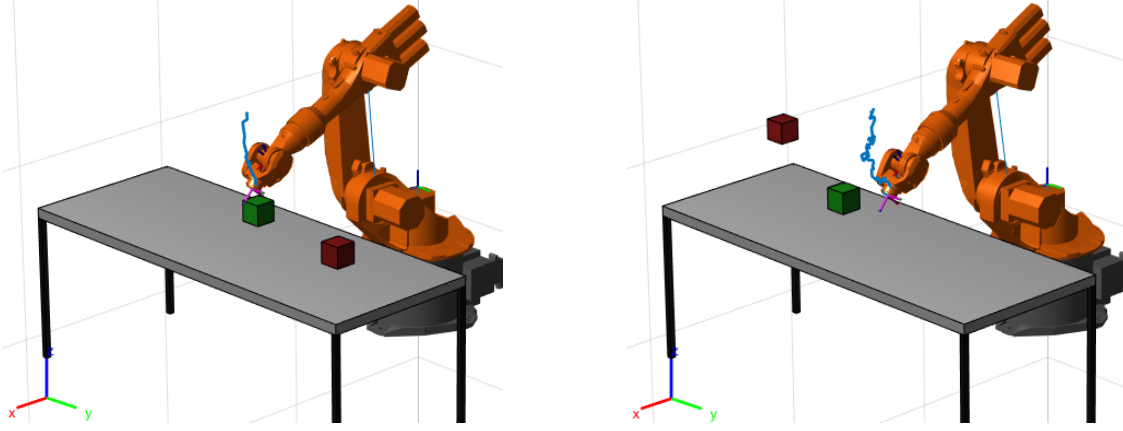
Figure 10: Optimal Trajectory (a) and learning curve during training (b) obtained by DQN agent implementation on variation of Test Project 3 with unknown obstacle.

6.2 Test Project 4: KUKA KR16 - Generic Configurations

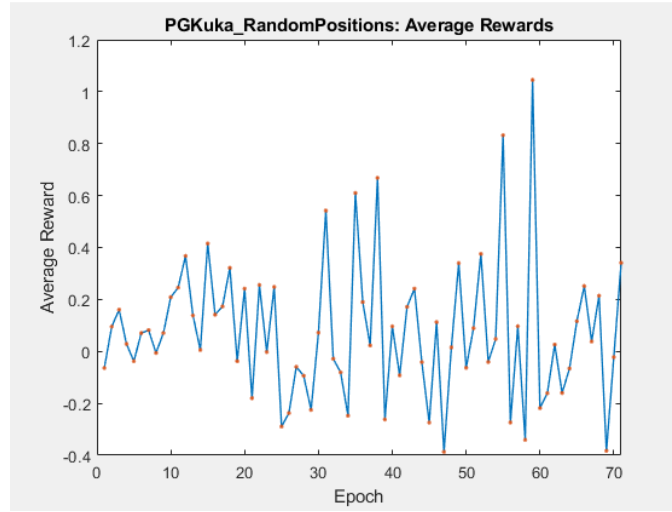
The main advantage of adaptive learning applied to the control of industrial robots is the flexibility in unexpected scenarios, the scalability provided by training over time and the abstraction of complex and often non intuitive policies with minimal human intervention. In order to investigate both agents' capability of learning an efficient positioning task for objects randomly located on the work space, both the goal's (green) and the known obstacle's (red) positions were changed randomly during training. A subspace $\mathcal{W} \subset \mathcal{R}^3$ of the robot's work volume, defined as $\mathcal{W} = \{(x, y, z) \in \mathcal{R}^3 | 0.80 < x < 1.4, -0.90 < y < 0.90, 0.80 < z < 1.00\}$, was chosen for the possible goal and obstacle positions. During testing, planar goal-obstacle configurations often did not require the agent to avoid the obstacle, as a direct path to the goal was frequently present. In order to test the RL agent on more challenging scenarios, a three-dimensional volume of possible goal obstacle configurations was chosen, giving the impression that some objects are floating.

6.2.1 Episodic REINFORCE

A REINFORCE agent with $\pi_\theta(\mathbf{a}|\mathbf{s})$ policy network architecture equal to Test Project 3 was implemented and trained. However, there was no noticeable increase in performance or convergence to the optimal policy, as shown in figure 17 c). The agent's performance presented an undesired high sensitivity to goal-obstacle configuration, performing the positioning task correctly on specific configurations (figure 17 a) and incorrectly on others (figure 17 b). Moreover, proximity between the goal and obstacle resulted in poor performance and increased chance of table collision.



(a) Correct positioning trajectory performed by the agent. (b) Table collision trajectory performed by the agent.

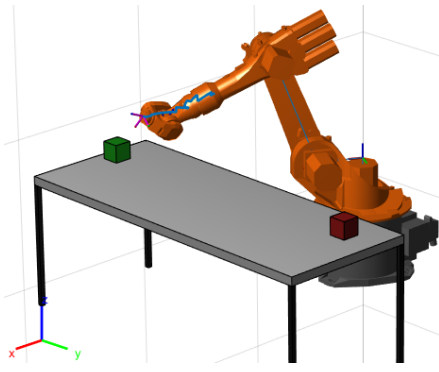


(c) Average rewards per epoch during training

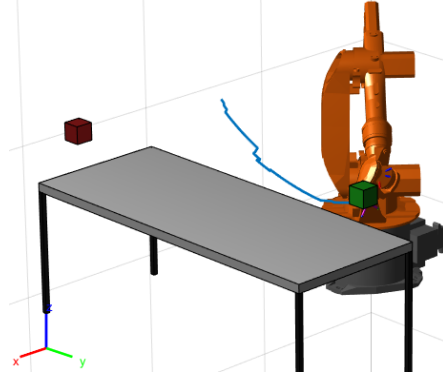
Figure 11: Trajectories generated by the REINFORCE agent after training for different goal-obstacle configurations (a),(b) and learning curve (c) during training.

6.2.2 DQN

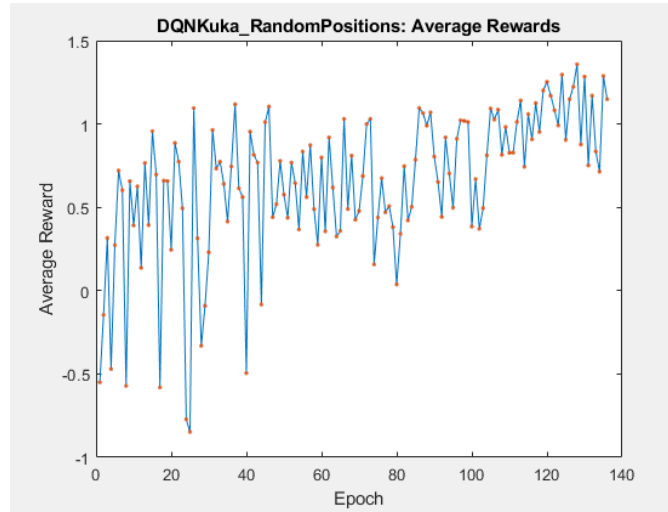
Finally, a DQN agent was trained in the same environment, with goal and obstacle positions being randomly generated on \mathcal{W} for each trajectory of a given training epoch. The process of randomly sampling transitions from different trajectories for mini batch composition during training allowed the agent to gradually learn the positioning task for generic goal-obstacle configurations, as shown in figure 18. As expected, an increased number of training epochs was required relative to the fixed configuration case.



(a) Trajectory performed by DQN agent in epoch 132 of training.



(b) Trajectory performed by trained agent on random goal-obstacle configuration.



(c) Average rewards per epoch during training

Figure 12: Trajectories performed by DQN agent during (a) and after training (b) and associated learning curve (c) during training.

6.3 Algorithms Comparative Analysis

In order to compare both classes of algorithm three main criteria were analyzed: convergence rate over the test projects, execution time and smooth increase of average reward. The policy-iteration algorithm REINFORCE outperformed DQN in the latter while the value-iteration algorithm showed better results both in execution time and convergence. Table 7 illustrates the execution time until convergence for both algorithms on the four test projects. DQN showed better scalability with increasing application complexity while REINFORCE agents presented longer training times.

Project	REINFORCE	Q-Learning
1 Degrees of Freedom (R)	6,8min	6,9 min
2 Degrees of Freedom (RR)	11.7min	7.4min
6 Degrees of Freedom (6R), fixed configuration	30h	16h
6 Degrees of Freedom (6R), random configuration	Did not converge	25h

Table 4: Average training times associated with both agents on each test project.

7 Conclusion

The application of newly developed methods, especially in consolidated industries where sensitive operations require that safety conditions must be met, is subject to an extensive research process in simulated settings and controlled environments. Reinforcement Learning is a relatively new field with promising results in control and game theory. The main contribution of this work is the evaluation of two classes of RL algorithms applied to a typical industrial robotics task and the development of a modular simulation architecture which allows for simplicity in further investigation of similar problems.

A comparative analysis of both classes of algorithms on increasingly complex environments also highlighted their main limitations and points to improve on future research: sensitivity to reward function, state-action space exponential increase in dimensions, low sample efficiency and consequently high training time. Reward function engineering is where human expert analysis is fundamental, the dimensionality issue is often overcome by algorithmic changes, such as the replacement of Q-tables with DQN or by modelling the action space \mathcal{A} as continuous and having the policy network $\pi_{\theta}(\mathbf{a}|\mathbf{s})$ output allow for the mapping onto continuous actions, commonly done in algorithms such as REINFORCE, DDPG and Actor-Critic (SUTTON, 2017).

The non-convergent behavior obtained by the REINFORCE agent on the last project can be explained by common limitations associated with policy iteration algorithms in general, such as high sensitivity to learning rate and exploratory variance (KORMUSHEV, 2013). DQN's overall better performance in shorter training periods is possibly due to higher frequency network updates and the implementation of an experience replay from which state transitions are randomly sampled (LIN, 1993).

8 References

BELLMAN, R. E. **Introduction to the Mathematical Theory of Control Processes**, volume 40-I. Academic Press, New York, NY, 1967.

GU, S et al. **Deep Reinforcement Learning for Robotic Manipulation with Asynchronous Off-Policy Update**. Google Brain, 2016.

JAMES, S.; JOHNS, E. **3D Simulation for Robot Arm Control with Deep Q-Learning**, Imperial College London, UK, 2016.

KOBER, J.; BAGNELL, A. J.; PETER J. **Reinforcement Learning in Robotics: A Survey**, 2013.

KORMUSHEV, P.; CALINON, S.; CALDWELL, D.G. **Reinforcement Learning in Robotics: Applications and Real-World Challenges**. Department of Advanced Robotics, Istituto Italiano di Tecnologia, 2013.

LIN, L. **Reinforcement Learning for Robots Using Neural Networks**, School of Computer Science, Carnegie Mellon University, 1993.

SUTTON, R. S.; BARTO A. G. **Reinforcement Learning: An Introduction**. 2nd Edition. The MIT Press, 2017.

9 Code

The entire project is open source and all codes can be found in:

[<https://github.com/MMenonJ/Controle_Robo_Manipulador>](https://github.com/MMenonJ/Controle_Robo_Manipulador)