electronics

MDPI

Article

Design and Development of Multi-Agent Reinforcement Learning Intelligence on the Robotarium Platform for Embedded System Applications

Lorenzo Canese, Gian Carlo Cardarilli, Mohammad Mahdi Dehghan Pir *, Luca Di Nunzio and Sergio Spanò

Department of Electronic Engineering, Tor Vergata University of Rome, Via del Politecnico 1, 00133 Rome, Italy; canese@ing.uniroma2.it (L.C.); g.cardarilli@uniroma2.it (G.C.C.); di.nunzio@ing.uniroma2.it (L.D.N.); spano@ing.uniroma2.it (S.S.)

* Correspondence: mohammadmahdi.dehghanpir@alumni.uniroma2.eu

Abstract: This research explores the use of Q-Learning for real-time swarm (Q-RTS) multi-agent reinforcement learning (MARL) algorithm for robotic applications. This study investigates the efficacy of Q-RTS in the reducing convergence time to a satisfactory movement policy through the successful implementation of four and eight trained agents. Q-RTS has been shown to significantly reduce search time in terms of training iterations, from almost a million iterations with one agent to 650,000 iterations with four agents and 500,000 iterations with eight agents. The scalability of the algorithm was addressed by testing it on several agents' configurations. A central focus was placed on the design of a sophisticated reward function, considering various postures of the agents and their critical role in optimizing the Q-learning algorithm. Additionally, this study delved into the robustness of trained agents, revealing their ability to adapt to dynamic environmental changes. The findings have broad implications for improving the efficiency and adaptability of robotic systems in various applications such as IoT and embedded systems. The algorithm was tested and implemented using the Georgia Tech Robotarium platform, showing its feasibility for the above-mentioned applications.

Keywords: reinforcement learning; Q-learning; multi-agent; Q-RTS; real-time swarm algorithm; robotics; IoT; embedded systems

Citation: Canese, L.; Cardarilli, G.C.; Dehghan Pir, M.M.; Di Nunzio, L.; Spanò, S. Design and Development of Multi-Agent Reinforcement Learning Intelligence on the Robotarium Platform for Embedded System Applications. *Electronics* **2024**, *13*, 1819. https://doi.org/10.3390/electronics13101819

Academic Editors: Hongfeng Wang, Muhammad Shahid Anwar, Maria Torres Vega and Muhammad Salman Pathan

Received: 1 March 2024
Revised: 24 April 2024
Accepted: 5 May 2024
Published: 8 May 2024

1. Introduction

It has been more than half a century since machine learning (ML) first captivated the attention of scientists, and the significance of ML applications cannot be overstated. In recent years, the remarkable availability of data made possible by the widespread use of the internet and the computational power offered by modern devices have further facilitated the proliferation of machine learning in several fields [1–8]. In recent years, reinforcement learning (RL) [9], a sub-field of ML, has gained significant popularity among researchers. This surge in interest is due to RL's ability to tackle tasks in a manner akin to human cognitive processes. Currently, RL has found diverse applications across domains such as finance, robotics [10], natural language processing, and telecommunications [11–14]. In RL, the agent's performance is evaluated through a feedback mechanism called a reward (R_t). Consequently, when pursuing a specific task, the agent strives to maximize its cumulative rewards to achieve the completion of the task [12,15].

The two major elements of RL are the agent (decision maker) and the environment, which should be considered a Markov decision process (MDP). The Markov property requires that the actions chosen by the agent in a particular state only influence the immediate reward and the subsequent next state, and this allows formalizing sequential decision making [15–17]. This model of the environment consists of states (S), actions

(A), and the probability of being in the next state (S'), which is denoted as $P(S'|S, A)$. Moreover, when the agent selects an action (**a**) in the state (**s**), the transition probability function $P(s'|s, a)$ describes the likelihood of transitioning to the next state (**s′**), and the associated reward function, denoted as $R(s, s', a)$, calculates the expected reward for this transition with respect to the discount factor γ. All these values are considered a tuple $[S, A, P(S'|S, A), R(s, s', a), \gamma]$ [11,12,15].

RL stands out from the crowd because it does not require prior knowledge of environmental dynamics. Instead, it learns through trial and error, exploring various actions to discover the best course of action to solve a problem without the need to solve the full complexity of the problem [15,17]. The agent or decision maker relies on a set of specific rules known as the policy (π) to select an action from the available actions (A_t) in the current state (s_t). Subsequently, the environment provides feedback in the form of a reward (r_t) and the next state (s_{t+1}), allowing the agent to refine its decision-making process. Through numerous iterations, this iterative state-action procedure ultimately leads to the discovery of a transition that maps states to optimal actions, referred to as the state-value function $V(s, a)$, with the goal of maximizing cumulative rewards. Initially, a random policy is assigned to the agent, and then the agent engages in a process of policy evaluation and improvement, culminating in determining an optimal policy for the desired behavior. On the other hand, the policy that maximizes the long-term reward is extracted from the optimal actions in each state recorded in the value function [12,15].

1.1. Related Works

Previously, engineers designed and fine-tuned robot controllers through a tedious manual procedure. However, in recent years, RL has gained widespread adoption in various robotic applications, including autonomous helicopters [18]; disaster management, such as fighting forest fires with a network of aerial robots [19]; surgery and medical assistant robots; pancake-flipping robots; space rovers; and more [20–22]. Ref. [23] conducted an investigation into the use of RL methods in robotics, where trained robots autonomously navigate to avoid obstacles and find the most efficient route to their goal. Furthermore, Ref. [12] emphasized that RL in robotics presents unique challenges, particularly due to the complex interactions between mechanical systems and their environments, especially when robots operate in close proximity to humans. Ref. [24] introduced a combination of Convolutional Neural Networks (CNNs) with RL to map raw image observations to control motor torque in robots. Ref. [20] categorized the applications of RL in robotics into three groups: underwater-based robots, air-based robots, and land-based robots. They highlighted that implementing RL in robotics involves various challenges, including defining the reward function, handling sensitive parameters, and dealing with high-dimensional continuous actions.

In many RL tasks, the model of the environment is unavailable, necessitating agents to acquire knowledge about it through experience. However, Q-learning, introduced in [16], is a prominent model-free RL algorithm. Unlike the direct bootstrapping approach of Temporal Difference, Q-learning employs a unique method by estimating the maximum discounted value for the next state-action pair, aiding the agent in making decisions aimed at maximizing anticipated future rewards. One of the Q-learning applications is path planning, which poses a significant challenge in mobile robot navigation and addresses the slow convergence of RL [22]; hence, the authors of [23] proposed "forgetting Q-learning", which considers navigation paths that might have been considered unacceptable through conventional Q-learning. This innovation fosters a greater inclination toward exploration. Moreover, Q-learning finds applications in robotic systems for tasks such as environment exploration and obstacle avoidance. For example, Ref. [25] employed a pre-developed deep Q-learning algorithm known as CNN-Q-Learning on the Turtlebot in a Gazebo simulation. This approach transformed raw sensor input from an RGB-D sensor into specific robot actions, enabling the Turtlebot to master obstacle avoidance in an unknown environment. The results demonstrated the robot's remarkable ability to learn effective strategies for

navigating and avoiding obstacles. However, modifications often prove to be the most effective strategy for enhancing a robot's performance in dynamic environments. In a related development, Ref. [22] proposed an Improved Q-Learning (IQL) method. Their work encompassed three key modifications to traditional Q-learning: the addition of a target distance measurement component, adjustments to Q-values to handle situations with no immediate reward or dead ends, and the introduction of a virtual target to circumvent dead-end states.

Using single-agent RL to solve a task can be cumbersome and time consuming when defining an optimal policy and evaluating an appropriate set of state-action pairs. Consequently, multi-agent reinforcement learning (MARL) has emerged as a solution to address this challenge. In MARL systems, multiple agents interact with the same environment simultaneously, reducing the time required to find the optimal policy and facilitating a more comprehensive exploration of the environment. However, managing interactions between agents in MARL poses its own challenges, leading various researchers to propose different algorithms to overcome the difficulties encountered in traditional RL. These systems have found applications in diverse domains, including traffic control, network packet routing, and robotics, where effective communication between agents is paramount to prevent potential calamities. For example, Ref. [19] introduced a Multi-Agent Deep Q-Network (MADQN) strategy for a fleet of Unmanned Aerial Vehicles (UAVs) aimed at autonomously combating forest fires. Furthermore, adaptability and stability among agents are fundamental characteristics of MARL. These traits are crucial as they facilitate continuous policy improvement through the exchange of data between agents [20].

In response to these challenges, Matta et al. [13] introduced an iteration-based Q-learning real-time swarm intelligence algorithm known as Q-RTS, which aims to use a technique to share knowledge between agents. This method not only addresses issues in MARL but also overcomes the timing limitations of traditional Q-learning and has proven effective in the field of robotics. In the Q-RTS algorithm, two types of data are utilized: local Q-values (Q_i), which are collected by each agent, and global Q-values (Q_{sw}), which are shared among all agents. During each iteration, the local Q-values are linearly combined with the global values, and the state-action pairs with the highest value are updated with the global Q-values. The independence factor, denoted as β, assesses the influence of the local and global Q-values using the agents during each iteration. Canese et al., in [11], identified key features of recent MARL algorithms. It is important to consider the non-stationarity of the environment when multiple agents are present in the same setting. Additionally, the scalability of these algorithms is often limited to a specific number of agents within the same environment [11]. Furthermore, Ref. [14]'s exploration of MARL led to the development of a novel approach, Decentralized Q-RTS, aimed at addressing challenges such as hardware implementation and achieving fully decentralized agent behavior. This innovative framework is particularly adept at handling scenarios where data transmission between agents fails or the number of available agents fluctuates, thereby ensuring robust operation in dynamic environments while facilitating the dissemination of knowledge among agents.

In the context of MARL, one common challenge is defining an appropriate reward function [17] The reward function is a crucial element of RL and Q-learning, as it guides the learning process and can significantly impact convergence. Designing an effective reward function often requires a deep understanding of the problem domain and can be a complex task. To address this, the authors initially establish a successful reward function for a single agent and then extend its use to multi-agent scenarios for further exploration.

1.2. Scope and Paper Organization

In this study, we implemented Q-RTS within a MARL system using Matlab. The Q-RTS method has emerged as one of the renowned techniques that theoretically demonstrate immense potential in MARL systems. However, this study primarily focused on finding an optimal policy and value function. Therefore, to enable further practical experiments,

it was imperative to devise a suitable reward function for the proposed robotic application. Moreover, addressing the issue of convergence time in MARL systems was a key concern in this research, and the results reveal significantly faster convergence through Q-value sharing among all available agents in the MARL system. Finally, the robustness of this approach concerning a rigorous reward function was tested by simulating dynamic environments. Despite the numerous potential applications of Q-RTS in robotics, particularly in path-finding and mapping robots, practical results are needed to demonstrate the algorithm's effectiveness in multi-agent reinforcement learning (MARL). Therefore, practical experiments were carried out on the Robotarium platform [26], offered by the Georgia Institute of Technology (Georgia Tech), which is a laboratory accessible worldwide. Robotarium features 20 unicycle robots known as GRITSBot-X, which are well suited for these experiments. Notably, this study marks the first instance of testing Q-RTS within this platform.

However, our focus was to evaluate the suitability of this algorithm for IoT and embedded system applications. Through analyzing the results, our goal was to validate all the aforementioned approaches.

The rest of this study is organized as follows. Section 2 outlines the methods used to evaluate the effectiveness of the techniques mentioned, the results will be discussed in Section 3, and Section 4 wraps up the article.

2. Methodology

The primary objective of this article was to present a Q-learning-based algorithm designed for an agent to initially locate a predefined track and maintain its position at the center. Furthermore, in a multi-agent implementation employing Q-RTS, multiple robots can communicate and exchange search information, aiming to minimize the overall search time. The Q-learning concept, as represented in Equation (1), involves adjustable parameters denoted as α and γ:

$$Q(S, A) \longleftarrow Q(S, A) + \alpha[R + \gamma \max_{a} Q(S', a) - Q(S, A)] \quad (1)$$

The learning rate, represented by alpha (α), typically falls between 0 (indicating no learning) and 1 (indicating fast learning). In this study, it was set to 0.2 to promote slow and consistent learning. Furthermore, the discount factor, denoted as gamma (γ), ranges from 0 to 1 and models the significance of future rewards relative to immediate rewards. A gamma of 1 implies that the agent values future rewards as much as current rewards, while a gamma of 0 suggests that the agent prioritizes only immediate rewards. For this study, $\gamma = 0.8$ was chosen, indicating that the value function relies predominantly on the subsequent state value.

To proceed with this investigation, it was vital to ensure that sensory and environmental factors remained quantifiable for the RL algorithm to compute the reward. Therefore, before progressing in the methodology, it was essential to incorporate certain configuration elements, including sensor and action settings, reward function specifications, the Q-RTS method, and the collision control algorithm.

2.1. Sensor Configurations

In this research, four ultrasonic sensors were used in the robots: three were located in the front and one in the back. In particular, these sensors measured the distance between the robot and the track border, irrespective of the robot's position inside or outside the track. These raw distance measurements, referred to as the sensor status, underwent conversion into distinguishable state criteria for utilization in the Q-learning. Consequently, at each time step, the sensor signal was transformed into a sensor state using the conversion method described in Table 1, supplemented by Figure 1.

Each crescent shape surrounding the robot in Figure 1 corresponds to a specific status and its equivalent state. Specifically, state-1 encompasses distances up to 0.2, state-2 ranges

from 0.2 to 0.5, and state-3 applies to distances greater than 0.5. It is noteworthy that distances beyond 0.5 were of lesser interest, as the primary focus was on proximity to the track, particularly for scenarios within the track, aiming for the middle.

Table 1. Sensor status and their related state configurations.

Status	State
$d \leq 0.2$	1
$0.2 < d \leq 0.5$	2
$d > 0.S5$	3

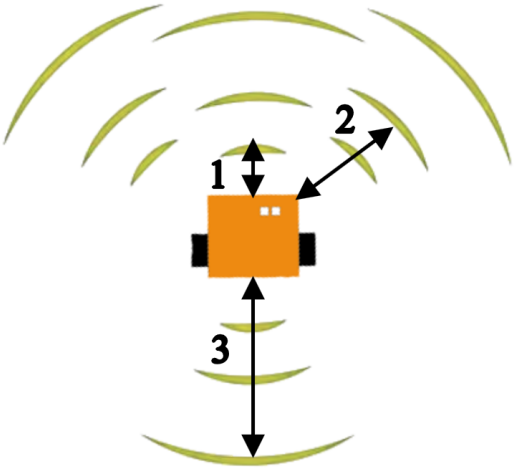

Figure 1. Schematic of equivalent sensor status and state.

2.2. Action Configurations

The robot decision-making process involves three distinct actions: turning right (action-1), turning left (action-2), and going straight (action-3). These actions are translated into motor commands by adjusting the rotational speed of the robot while maintaining a constant linear speed. Since the robot is omnidirectional, changing the speed of one wheel with respect to the opposite one will lead to a change in the angular direction of the robot, while the overall linear speed remains constant. In particular, changes in the rotational angle around the vertical axis result in directional adjustments.

In the context of Q-learning, the epsilon (ϵ) coefficient plays a crucial role in balancing exploration and exploitation. A higher ϵ value encourages random action selection, fostering exploration, while a lower ϵ value promotes the selection of actions with the highest known rewards, facilitating exploitation. The optimal strategy involves initiating the learning process with exploratory training and gradually reducing ϵ. This transition allows the agent to evolve from exploring various actions to favoring the most rewarding choices in each state.

In this study, a deliberate choice was made to set $\epsilon = 0.8$ during the learning phase, emphasizing exploration, and subsequently shift to $\epsilon = 0$ during the experimental phase to prioritize exploitation.

2.3. Reward Function

It has been emphasized that the reward function is central to RL as it serves as the primary means for the robot to interact with its environment, comprehend the outcomes of its actions, and subsequently make informed decisions. Consequently, establishing an accurate reward function based on the available sensor and actuator information in relation to the robot's goals is of paramount importance. It is worth noting that robot positioning is categorized into two distinct regions: inside the track and outside the track. As a result,

reward functions are defined on the basis of different sensor reading approaches in these two regions.

In this research, the environment was also partitioned into two segments, each requiring distinct reward scenarios. The first scenario entailed the robot being inside the track, with the goal of maintaining a middle pathway within it. The second scenario involved the robot being outside the track, with the objective of locating the track and navigating toward it. To design effective reward functions, a comprehensive analysis of all possible states and poses for the robot was performed, both inside and outside the track. For each pose, rewards were assigned as a constant value or, in certain cases, determined by a reward function based on the minimum distance between the robot and the track borders. To aid in understanding the agent's behavior in various scenarios, a schematic representation of the robot in different situations is provided graphically in Figure 2.

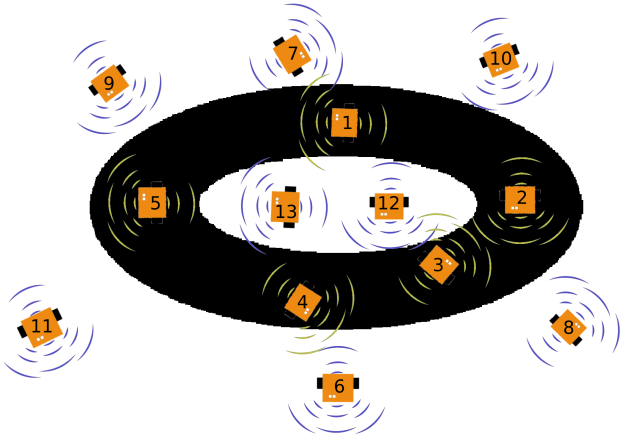

Figure 2. Different robot positions inside and outside of the track.

Inside the track scenario. In these cases, the agent will receive a general reward for being inside the track. Moreover, positions 1, 2, 3, 4, and 5 are considered the inside-track modes of the robot, which are the straight in the middle (cases 1 and 2, the highest rewards); turning to the borders (cases 3 and 4, low reward); and, finally, straight, heading to the border, which may consequently go outside the track (case 5, the lowest reward). The check for being in the middle path is performed by comparing the status of two-sided sensors, and returning to the track or heading to the borders can be distinguished using the statuses of the two front and back sensors. Therefore, assigning different rewards for each position provides more intuitions about the agent's behavior. The green box in the flowchart depicted in Figure 3 signifies this aspect of the reward function.

Outside the track scenario. An overall penalty is assigned for being outside the track. This penalty will persuade the robot to seek the track. However, 6 to 13 positions in Figure 2 are outside the track modes. In addition, for a better understanding of the robot's behavior, four different positions are considered: just comes outside the track (cases 6 and 7), passes by the track (case 8), heads to the track (cases 9 and 10), and, finally, does not see the track or is far away from the track (case 11, the highest penalty). This section of the reward function is indicated by the red box in Figure 3. Additionally, the remaining instances (12 and 13) represent modes classified as outside-in-the-middle, where all sensors can see the track, and their statuses are all-sensor $\neq$ inf, but their differences reveal the robot's situation. In mode-12, the robot is in the process of exiting the track, while in mode-13, it is heading toward the track. The rewards assigned for these cases are illustrated by the black box in Figure 3. However, in every scenario, the sensor status distinguishes the modes, and in each mode, a different reward has been allocated. For example, the status of the back sensor shows if the robot just comes out of the track (back sensor $\neq$ inf) or is heading toward the track (back sensor $=$ inf), or the combination of front and side sensors reveals if the track is on the left or right of the robot.

Given the primary goal of the robot, which is to stay on the track and return if it deviates, the agent requires a higher level of intuition to initiate a turn back before it reaches a critical point. Consequently, the utilization of a combination of sensor states and statuses not only addresses the challenge of diverse positional situations but also ensures that the distribution of the state-action value function is effectively organized.

Figure 3 illustrates the reward distribution flowchart.

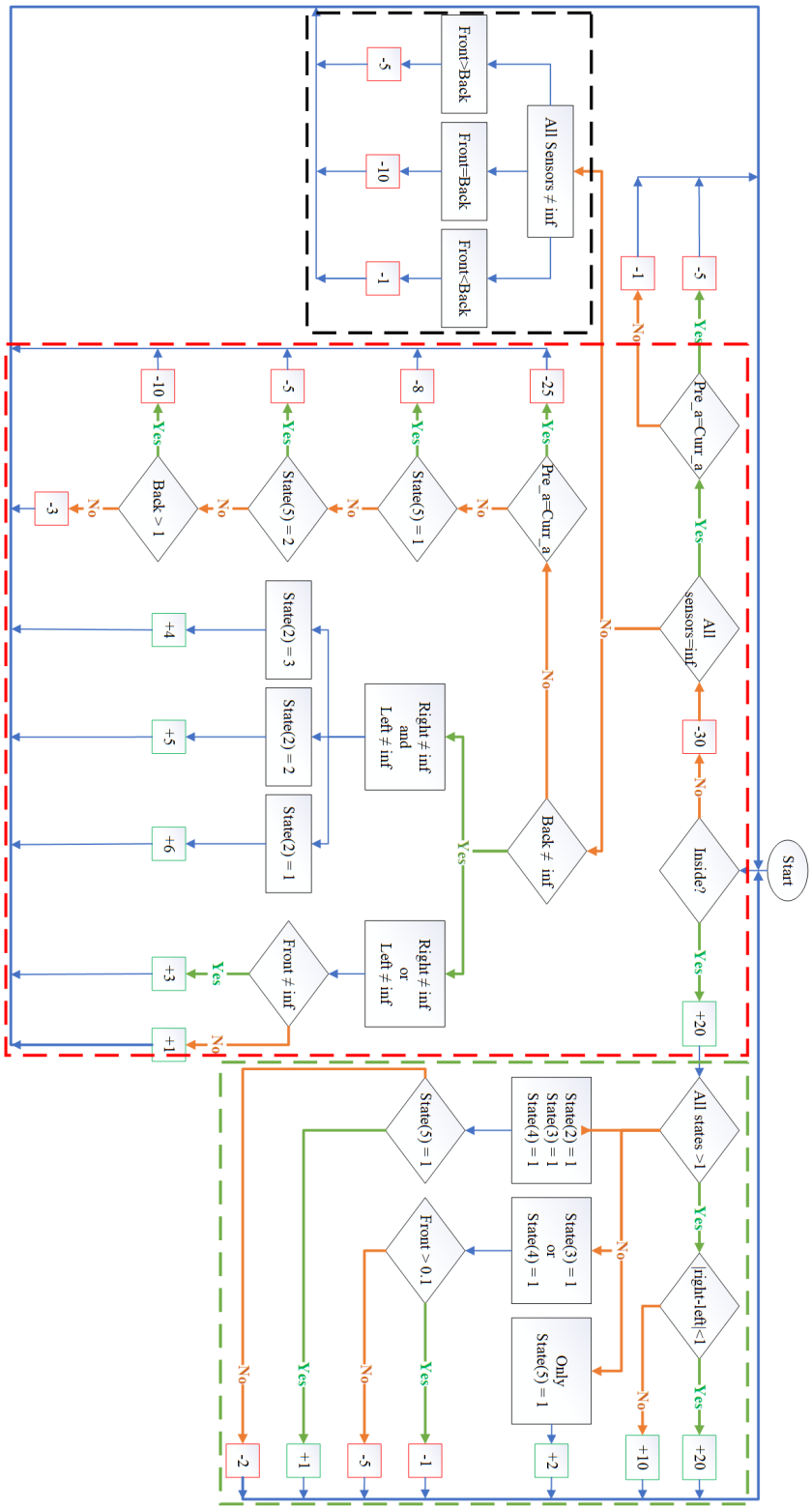

Figure 3. Reward distribution flowchart.

As mentioned above, the reward function is divided into two distinct areas, and the robot must navigate back to the track, especially at the beginning of the exit. The green box in the flowchart represents the reward distribution for the inside-the-track scenario, while the red box represents the reward function for the outside-of-the-track scenario. The black box corresponds to the case of a somewhat-outside-the-track-area scenario on the map, where all sensors can detect the track, but the robot is not on the track itself.

In instances where the agent has moved far away from the track, a strategy is employed to influence the agent's actions even when facing an infinite direction. In such cases, the robot selects different actions in each iteration compared to the previous one. This strategy ensures that the agent avoids deviating too far from the optimal return path, forcing it to turn around and head back towards the track. It is important to note that this method does not halt the Q-learning process; instead, it compels the agent to choose different available actions randomly. Subsequently, in the next iteration, the previous action is disabled. In this manner, the robot engages in a pure searching process to rediscover the track while enhancing its learning capabilities and updating Q-values.

2.4. Q-RTS

One of the main goals of this research was to implement a state-of-the-art Q-RTS algorithm into the robot. This method collects the highest returned state-action pairs and shares them among multi-agents to reduce the time needed to train agents in the environment. The Q-RTS algorithm consists of two primary components: the update process and global matrix computation.

The update process was facilitated using a local combination (linear) function (Equation (2)) to create the updated local matrix Q'_i. During each environmental search time interval, each agent leverages two Q-values, the individual local Q_i and the shared swarm Q_{sw} in a linear function with an independent factor $\beta = 0.1$. This process enables the agent to estimate the updated local matrix Q'_i and select a higher-valued state-action pair from it. Consequently, the agent's decision-making process in its current state is primarily influenced by the shared swarm Q-value using a factor of 0.9. This independent factor can take values between 0 and 1, and for this research, it was set to 0.1. Consequently, the agent relies more on the shared knowledge than on its individual research efforts. The rationale behind selecting this value stems from the intrinsic characteristics of Q-learning and its bootstrapping technique. Consequently, within the same state, different agents may opt for distinct actions, resulting in subsequent transitions to future states with associated state values. This consequential value has the potential to influence the current estimated state-action pair. Through simulation, the interaction of these states with diverse agents accelerates the updating process, leading to quicker convergence toward an optimal value. On the other hand, by setting the independent factor $\beta = 0.1$, we introduced a kind of new degree of exploration–exploitation trade-off within a system of multiple agents, allowing them to balance between exploiting their own high-value actions and exploring potentially better alternatives shared by the other agents.

$$\begin{cases} Q_i(s_t, a_t) \leftarrow (1-\alpha)Q'_i(s_t, a_t) + \alpha[(r_t + \gamma \max_a Q'_i(s_{t+1}, a_t)] \\ \\ Q'_i(s_t, a_t) = \beta Q_i(s_t, a_t) + (1-\beta)Q_{sw}(s_t, a_t) \end{cases} \tag{2}$$

Moreover, in MARL, different agents may choose different actions in the same state, leading to diverse trajectories and state-value estimates. By incorporating $\beta = 0.1$, we encourage agents to leverage shared knowledge from the swarm while still maintaining individual exploration. This facilitates faster updates and convergence to an optimal policy by leveraging insights from multiple agents' experiences. Therefore, it reduces the need to search the environment for each agent individually.

The global matrix computation (Equation (3)) is achieved through an aggregation non-linear function that merges the agents' knowledge for updating the shared swarm

matrix Q_{sw}. This means that, prior to reaching the successful state, the algorithm evaluates all agents and their current states, updating the shared swarm matrix Q_{sw} by comparing it with the current estimated Q_i (available on the set $\prod$ of all the local matrices Q_i) to select the highest state-action value.

$$\begin{cases} \max\limits_{Q_i \epsilon \prod} Q_i(s_t, a_t), & \text{if} |\max\limits_{Q_i \epsilon \prod} Q_i(s_t, a_t)| \geq |\min\limits_{Q_i \epsilon \prod} Q_i(s_t, a_t)| \\ \min\limits_{Q_i \epsilon \prod} Q_i(s_t, a_t), & \text{otherwise} \end{cases} \tag{3}$$

This aggregation process occurs sequentially after each agent's decision-making phase, ensuring that the Q_{sw} is continually updated throughout each episode.

2.5. Collision Control Algorithm

The agent's collision avoidance algorithm is of great importance in our system. Since collision avoidance is not part of the decision-making algorithm, it is crucial to pause the agent training process during collision avoidance. In each iteration, the distances between agents are calculated to assess the likelihood of collision between two adjacent agents within a range of 0.3.

As depicted in Figure 4, the 6th state (used to identify the presence of another robot nearby) serves as a condition to halt the training and activate the collision avoidance algorithm. Based on the position and orientation of each pair of adjacent agents, a goal position is assigned to each agent within a range of 0.3 ahead of their current position and in the same direction as their last movement. To achieve these new goals while avoiding collisions, three predefined Robotarium certificates (create-si-position-controller(), create-si-to-uni-dynamics(), and create-uni-barrier-certificate2()) are employed.

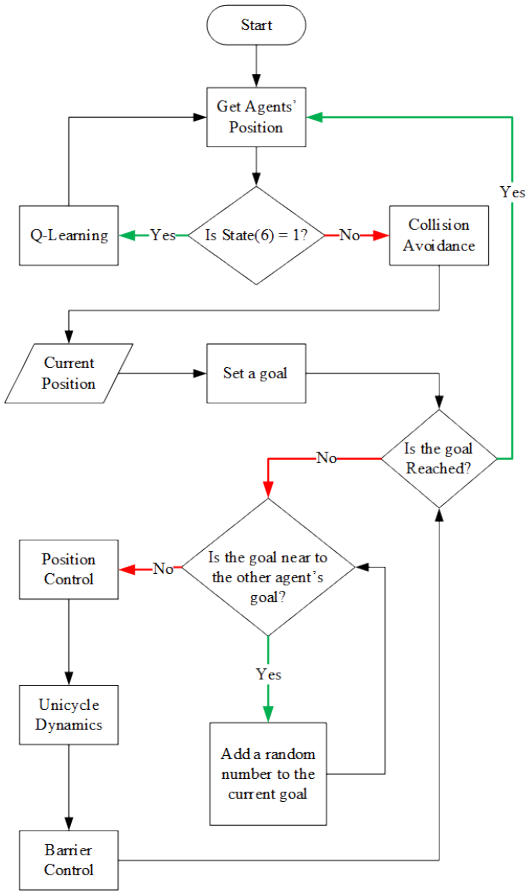

Figure 4. Collision avoidance flowchart.

While the collision avoidance algorithm utilizes the goal position of potential collision partners, it also considers the real-time positions of other agents. As a result, Robotarium certificates automatically determine optimal linear and angular velocities for agents at risk of collision. This not only ensures the avoidance of collision, but also prevents interference with other agents. The pseudocode that describes the collision avoidance algorithm is provided in Aglorithm 1.

Algorithm 1 Pseudocode for collision avoidance algorithm.

procedure COLLISION AVOIDANCE(agents position, x, y, θ, state(6))
  while state(6)=2 **do** ▷ Loop for each step until goal is reached
    Goal $\leftarrow (x + \Delta x \cdot \cos\theta, y + \Delta y \cdot \sin\theta)$
    Check the current position and goal position
    Transform the position command to unicycle dynamics
    Check the barrier between two adjacent agents
    $V(agent) \leftarrow (v_{linear}, v_{angular})$
  end while
end procedure

The first certificate computes the distance between the current position and the goal position for each agent, determining the linear and angular velocity in each iteration. Subsequently, the second certificate transforms the agent's position control into the unicycle dynamic signal, regulating the acceleration and speed of the agents' wheels. Finally, the third certificate manages the safe barrier between two agents to ensure effective collision avoidance.

2.6. Simulation and Experimental Design for Result Discussion

This study encompassed both computational and experimental outcomes, specifically focusing on the cumulative rewards obtained in each iteration by 1, 4, and 8 agents. Although RL typically employs the same training environment for testing (a static environment), delving into dynamic environments ensures robust performances by trained agents. It is crucial to note that the results are based on 200 simulations, each comprising 5000 iterations, resulting in a total of one million recorded rewards.

2.6.1. Reward as a Figure of Merit to Compare and Mean Cumulative Reward

In the initial phase, the algorithm was tested with a single agent, recording rewards for each iteration for subsequent analysis. Once robust behavior was established in a single-agent scenario, the agent count was increased to 4 and 8, employing the Q-RTS method. Notably, in multi-agent simulations, agents collaborate by sharing environmental knowledge, aiming to optimize the search process. To facilitate a meaningful comparison between single- and multi-agent simulations, the Mean Cumulative Reward (MCR) was calculated using the recorded rewards for each agent in every iteration.

$$MCR = \frac{\sum_{N=i}^{n} R_N}{n} \tag{4}$$

This rigorous experimental design allowed for a comprehensive algorithm performance assessment across various agent scenarios.

2.6.2. Movmean Function

To mitigate the impact of noise inherent in the simulation, the **movmean** function, available in the Matlab package, was employed. The function, **M = movmean(A,k)**, calculates the mean value of neighboring elements within a matrix A and stores them in a new matrix (M) of the same size as A. As illustrated in Figure 5, a moving window traverses the k-adjacent neighbors, computing the mean value over this window.

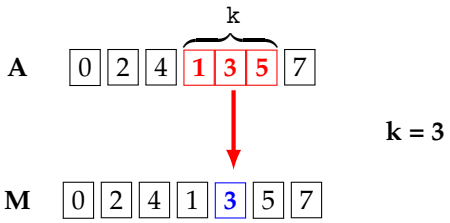

Figure 5. Movmean function in Matlab package. A window (k) reveals the number of arrays that were taken to calculate the average neighboring elements.

However, a distinction arises between odd and even windows. In the case of an odd-numbered window (k), it centers on the element at the current position. On the contrary, for an even-numbered window, the center is shared between the current and previous elements. If the window centers on the first element, the preceding sub-windows are treated as zero. At the matrix's end, right-sided sub-windows are truncated, and the average is computed over the elements within the window.

It should be noted that when a single agent navigates the environment independently, its allocated reward exhibits considerable variability, leading to inevitable fluctuations in the MCR over iterations. Consequently, a larger **movmean** window is necessary (e.g., k = 50,000). On the contrary, when multiple agents are involved, the observed fluctuation is less due to collaborative environment exploration and information sharing among agents. This cooperative behavior leads to the use of prior knowledge, emphasizing exploitation over exploration, and ultimately allows a smaller window size (for example, k = 15,000) in the reward analysis.

2.6.3. Step Delay

Another facet of this study involved introducing delays in the Q-learning process during simulations. This means that in every iteration where Q-learning was applied, the agent subsequently employed identical action and velocity vectors for a certain number of iterations, without receiving additional rewards. Here are some reasons why step delays can be beneficial:

- **Exploration–Exploitation Balance:** Q-learning involves a trade-off between exploration and exploitation; therefore, introducing step delays can encourage the agent to explore different actions for a longer duration before updating its Q-values, potentially leading to a better balance between exploration and exploitation.
- **Stability and Consistency:** Delaying the application of Q-learning for a certain number of iterations can provide stability and consistency in the agent's decision-making process. This could be particularly useful in dynamic environments where rapid changes may lead to suboptimal decision making if the agent updates its Q-values too frequently.
- **Memory and Learning Efficiency:** In holding the agent's action and velocity vectors constant for a certain number of iterations, the agent effectively "memorizes" its current strategy. This can be advantageous in situations where maintaining a consistent strategy over a short period is beneficial for learning more complex patterns in the environment.
- **Simulation Realism:** In some scenarios, introducing delays might align more closely with real-world situations. For example, if an agent in the real world cannot update its strategy instantaneously, introducing delays in the simulation could mimic this real-world constraint.
- **Comparative Analysis:** This strategy facilitates a comparative analysis between simulations with step delays and no-delay simulations across varying numbers of agent setups. This comparative analysis aims to gain insights into the impact of delays on the agent's accumulated reward, providing valuable information on whether delayed Q-learning results in an improved or diminished performance.

However, within the scope of this research, simulation step delays ranging from 1 to 5 were implemented and compared with simulations without delays in terms of the agent's accumulated reward. As a result, the recorded rewards remained unchanged, consistent with the rewards earned in the previous Q-learning execution.

Considering the benefits of accounting for step delays, limiting the step delays to 5 steps was primarily due to the higher amount of time needed to complete the simulation. Although there may be slight improvements in cumulative rewards with higher step delays, some state-action values remained unchanged and were not updated through the step-delay times. This issue is mostly addressed in multi-agent simulations, where one-step delay outperforms the other step delays. On the other hand, in the case of being outside of the track, this causes the agents to go farther away from the track (since the agent uses the previous action for further consecutive states) and reduces their chance to correctly find their path to go back to the track, as mentioned in the dynamic environment experiment.

2.7. Robotarium

The Robotarium is a global, freely accessible testbed for swarm robotics provided by the Georgia Institute of Technology (GeorgiaTech) for both experimental and educational purposes. Within this facility, there are 20 remotely accessible unicycle robots known as GRITSBot-X. We chose to conduct the experimental aspect of our research using this platform.

To incorporate our data into the designated robots, the Robotarium has an established Matlab simulation platform. This platform allowed us to evaluate our concepts before submitting the files through the Robotarium website. Subsequently, our scripts and simulations were validated by the Robotarium team, following which the experiment was conducted on the Robotarium platform. The resulting data, along with a video documenting the experiment, were then sent and made accessible through the user profile.

3. Results

After analyzing the data from a single-agent simulation, the program incorporates the step-delay notion. Additionally, four agents were used in the simulation, each with a different step delay. The influence of the Q-RTS approach with respect to the number of agents was then determined by examining the best step-delay case in the eight-agent simulation and comparing it to the same step delay in all agent case scenarios. The best-case situation was ultimately sent to Robotarium for experimental examination.

3.1. One-Agent Simulation

The simulation started with one agent with no step delay, and the study was completed by increasing the step delay to five. Figure 6 shows the agent's reward for each iteration, which was recorded for every simulation.

As mentioned above, our simulations involved recording 1 million rewards for each simulation. To facilitate a more nuanced comparison, we segmented the performance graph into three distinct phases: a black box for the initial phase (up to 200,000 iterations), a red box for the middle phase (200,000 to 800,000 iterations), and the final phase, which is in green (more than 800,000 iterations). In the initial phase, the no-step-delay agent exhibited a better performance. However, throughout the simulation, the other agents demonstrated comparable performances. In particular, the agent with a one-step delay displayed significant fluctuations in the middle phase, leading to slower convergence compared to the agent with a five-step delay, which exhibited greater stability and achieved fast convergence within 700,000 iterations. Surprisingly, in the final phase, despite its fluctuating reward distribution, the one-step delay agent achieved the highest total reward, while the other agents obtained comparatively lower rewards. This can be attributed to the one-step delay agent's ability to navigate its environment more freely, benefiting from an additional step in its decision-making process. However, the no-delay agent maintained a middle-ground position in reward allocation, which is indicative of its inherent balance in searching principles. However, as the delay step increased, the performances of the

other agents in the final phase decreased, with the five-step delay agent demonstrating less fluctuation and faster convergence, albeit with a lower total reward.

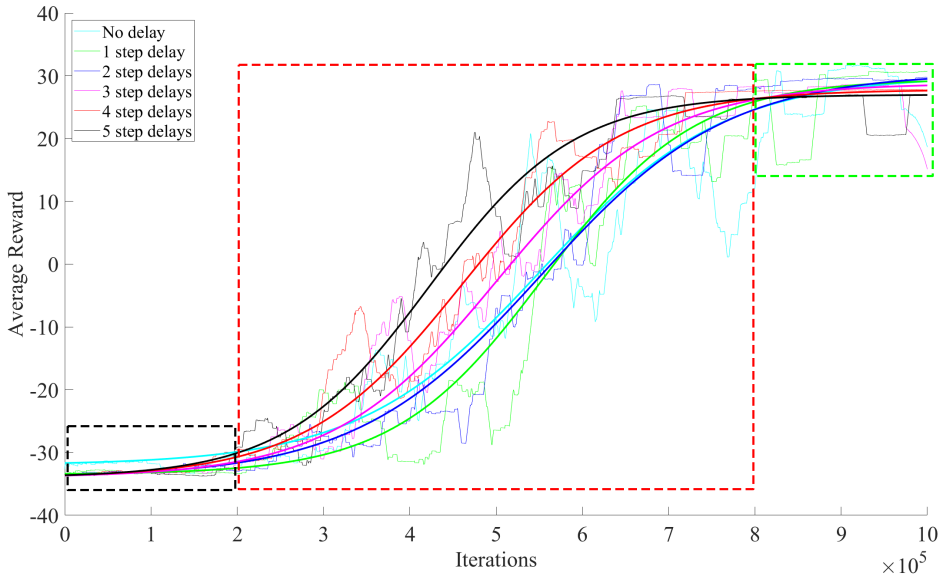

Figure 6. Cumulative reward for one agent with different numbers of step delays.

3.2. Four-Agent Simulation

The Q-RTS method was employed in multi-agent simulations to expedite search times through collaborative information sharing among agents. Given the effective performance of the reward function in single-agent simulations, the four-agent Q-RTS simulation utilized the same reward function, and the outcomes, as indicated by the MCR, shed light on the algorithm's efficacy. In Figure 7, the MCR graph for the four-agent scenario is partitioned into three distinctive windows. These windows demarcate the initial phase in the black box (up to 200,000 iterations), the red middle-phase box (from 200,000 to 600,000 iterations), and the final phase in green (exceeding 600,000 iterations). Specifically, the middle-phase window encapsulates the MCR results from the conclusion of the initial phase until convergence in the MCR was attained.

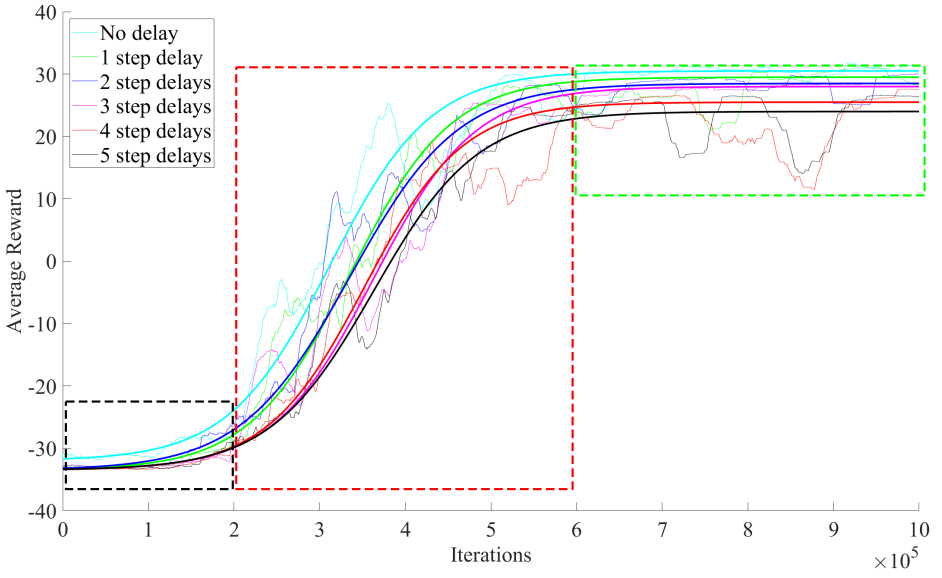

Figure 7. Average cumulative reward for four agents with different numbers of delays.

Similar to the single-agent scenario, multi-agent simulations commenced with low MCR values, resulting in closely aligned graphs for various step delays. However, during the middle iteration in the red window, the distinctions in the accumulated reward graphs become evident due to differing step delays. Notably, the no-delay agent exhibits a substantial upward trajectory during this phase, showcasing faster convergence to the final MCR, particularly after 450,000 iterations. Despite the rapid ascent, this agent's graph displays minimal fluctuations, and the early plateau observed until the conclusion of the simulation underscores the robustness of the Q-RTS method, leading to improved convergence without any delay.

Although the presence of delays in the simulation adversely affects the MCR and the overall performance of the Q-learning algorithm, a notable similarity emerged with the one-agent simulations. Specifically, a one-step delay in the four-agent simulations demonstrates a performance comparable to that of the no-delay scenario. The diminished performance in simulations with delays can be attributed to the improved coverage of the environment by multi-agents, resulting in the reduced necessity to bypass the Q-learning process and, consequently, the Q-RTS method.

3.3. Single-Agent vs. Multi-Agents with One-Step Delay

The choice of a one-step delay was made for the eight-agent simulation employing Q-RTS. The subsequent comparison, varying the number of agents, provides insights into the performance of Q-RTS relative to the agent count. Notably, despite an initial phase where all simulations exhibit similar performances, as shown in Figure 8, the utilization of eight agents illustrates an expedited convergence, depicted by a sharply ascending graph up to 400,000 iterations, where the MCR convergence initiates and persists to a plateau at 500,000 iterations until the conclusion of the simulation.

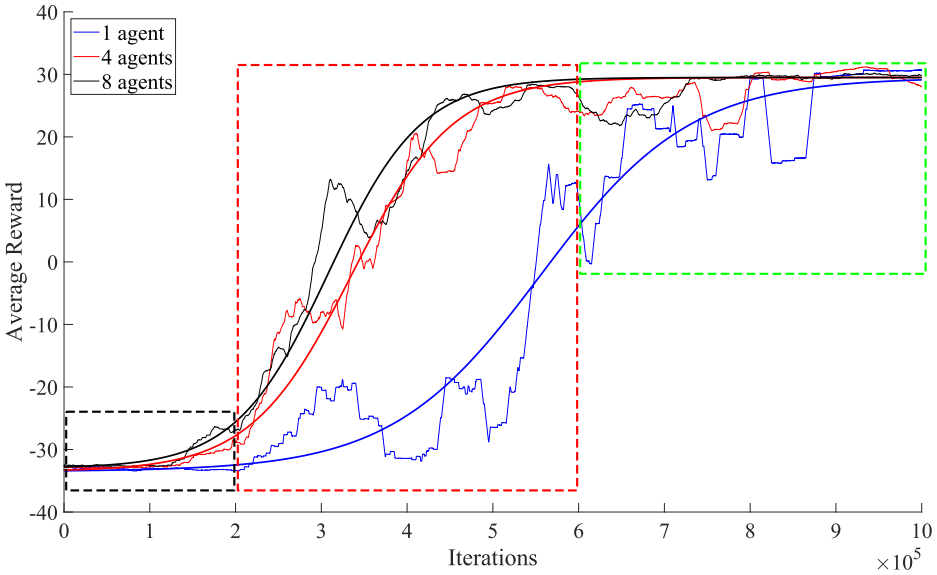

Figure 8. Average cumulative reward comparison for one, four, and eight agents with one-step delay.

However, the simulation with four agents exhibits a similar pattern but converges to the same MCR value within 600,000 to 650,000 iterations. In contrast, the single-agent scenario requires an extended duration to converge, and even at the end of the simulation, full MCR convergence was not attained.

As the simulations progressed to the final stage, convergence to a nearly identical MCR was observed across all scenarios. Intriguingly, the single-agent simulation lagged in terms of convergence speed. The incorporation of eight agents in Q-learning, coupled with Q-RTS, showcases a superior performance among all simulations, achieving a quicker

convergence and better-accumulated rewards. This outcome underscores the efficacy of Q-RTS within the Q-learning framework.

3.4. Success Rate

The success rate serves as a metric for comparing different scenarios and assessing whether the proposed algorithms and methods enhance results. In this study, the one-step delay scenario was utilized across three agent setups (one, four, and eight agents), encompassing 100 training simulations with 5000 iterations each.

$$\text{SRP} = \frac{\text{Number of successful iterations}}{\text{Total iterations}} \times 100 \tag{5}$$

The success rate percentage (SRP) shown in Equation (5) is calculated by dividing the number of iterations in which the agent remained inside the track by the total number of iterations within each simulation. The results of this comparison are illustrated in Figure 9.

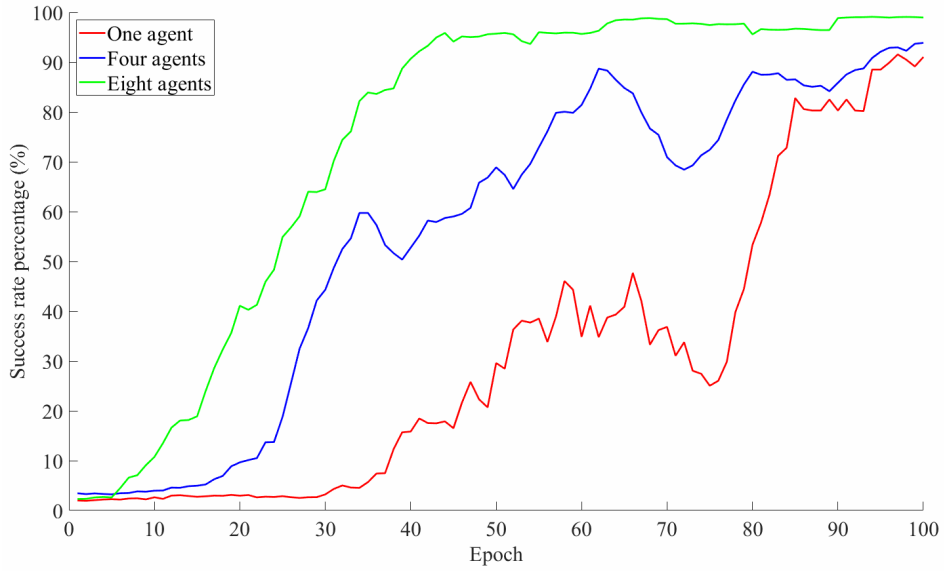

Figure 9. Success rate percentage comparison for one, four, and eight agents with one-step delay, over 100 simulations with 5000 iterations each.

As shown in Figure 9, the single-agent approach requires more simulations to train the agent to navigate the track and maintain its position within it. Initially, its success rate is low, but after 30 epochs, it begins to improve, reaching a 90% success rate after 95 consecutive epochs.

In contrast, the shared knowledge method employed in the Q-RTS algorithm for four and eight agents demonstrates exceptional performance in terms of success rate percentage and learning time. The four-agent configuration begins to enhance its performance after 15 epochs, ultimately achieving a 93% success rate. However, the eight-agent configuration surpasses both setups with a 98% success rate. Notably, success rate improvement starts early in this setup, reaching its peak after 45 epochs. This early progress can be attributed to the collaborative nature of the Q-RTS algorithm, which accelerates the search for an optimal performance within the environment.

In Table 2, the success rate measured after training is presented along with the standard deviation over 50 simulations. For all the configurations, the rate is over 90%. We can notice that the success rate decreases as the number of agents grows; this is due to the fact that if more agents are in the track, they will need to avoid collision more often. Since the collision avoidance algorithm has a higher priority than being on the track for some iterations, the agents will exit the track to avoid collision and re-enter after.

Table 2. Final success rates after training for one-, four-, and eight-agent configurations. Percentage and standard deviation for 100 simulations with 5000 iterations each.

Number of Agents	Success Rate
1	$92.66 \pm 1.83\%$
4	$98.29 \pm 1.12\%$
8	$99.63 \pm 0.49\%$

3.5. Dynamic Track

The other aspect of this research focused on changing the track in the middle of the simulation process. For this purpose, the scenario with four agents and a one-step delay was chosen. The following wo different implementations were considered:

1. Two tracks far away from each other ($(a1)$ and $(a2)$ in Figure 10);
2. Two tracks in the same area but with different shapes ($(b1)$ and $(b2)$ in Figure 10).

Figure 10 depicts the setup for these two experiments. In this scenario, the simulation begins with $(a1)$, and in the middle of the simulation, the track position changes to $(a2)$. This demonstrates that the robots are unable to locate the track because the new position is beyond their sensor range, resulting in the apparent loss of the track from their reach.

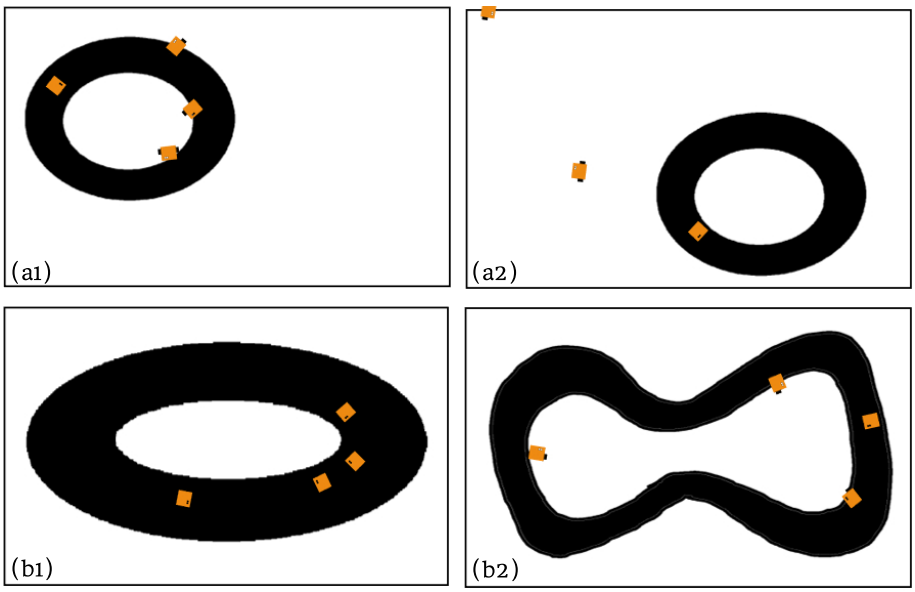

Figure 10. Dynamic track simulation: (**a**) tracks are simulated far away from each other, from the start (**a1**) to the final (**a2**) situation; (**b**) two different tracks are simulated in the center, for which (**b1**) shows the beginning and (**b2**) shows the middle of the simulation.

On the contrary, in Figure 10(b1,b2), when the track changes shape but remains within the range of the robot sensor, the agents can easily adapt to the new situation, find the track, and stay in the middle of it. This experiment demonstrated the stability and robustness of the algorithm in response to environmental changes. It highlights the algorithm's excellent compatibility with dynamic environments, showcasing a high likelihood of trained agents successfully seeking out a new track.

3.6. Robotarium Implementation

In the final stage of this investigation, the Q-values of four trained agents were sent to the Robotarium testbed provided by Georgia Tech to examine the efficiency of the reward function and Q-RTS in practice. The results were obtained in terms of a recorded video by the laboratory, with Figure 11 illustrating key moments in the experiment. Initially,

all the agents were positioned on the platform (a), and then the track was projected onto the field (b). The robots began to move randomly (c), signaling an imminent collision, prompting the collision avoidance algorithm to activate (d). After a while, the robots successfully located the center of the track and started following each other at a safe distance to prevent possible further collisions (e). The provided video and the returned data show a high correlation with our simulations, as expected.

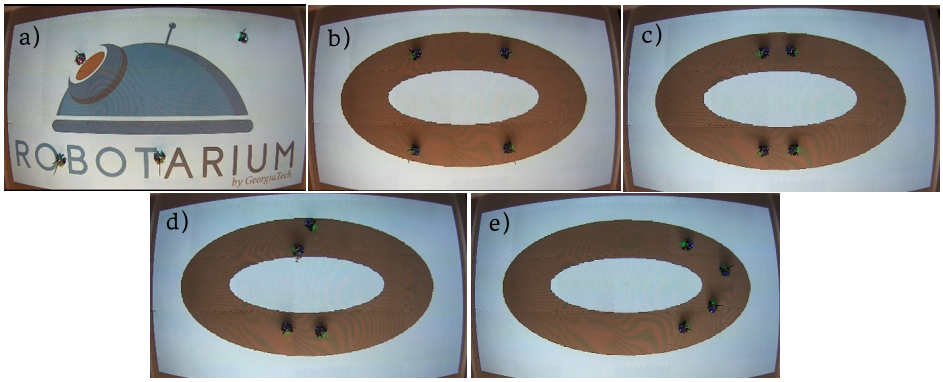

Figure 11. Robotarium experimental simulation, implemented into GRITSBot-X: (**a**) start of the experiment, (**b**) positioning of robots, (**c**) start of collision avoidance algorithm, (**d**) collision avoidance procedure, and (**e**) robots are in the middle of the track and follow each other.

4. Conclusions

This study reveals that the Q-RTS method is highly effective in reducing the search time of robots, resulting in a more efficient achievement of the environmental objectives. The significance of this finding lies in its potential to enhance the real-world applicability of robotic systems, where efficiency is paramount. This research underscores the critical role of a well-designed reward function in the success of the Q-learning algorithm. By considering diverse agent postures, the algorithm becomes more nuanced and able to adapt to complex scenarios. This finding has broad implications for the development of future robotic systems that rely on RL. The scalability of the Q-RTS method was demonstrated through the successful implementation with four and eight trained agents. This scalability is pivotal for scenarios where a team of robots is required to work collaboratively on tasks, showcasing the method's adaptability to different team sizes. This is a crucial feature if the algorithm should be applied to applications of IoT and embedded systems. Furthermore, this study highlights the robustness of trained agents in adapting to dynamic environmental changes. This implies that robotic systems can maintain performance and adaptability even when faced with unforeseen alterations in their operating environment. However, this research implies the following insights as well:

1. **Real-World Applicability:** The Q-RTS method's efficiency suggests its potential application in real-world scenarios, such as search-and-rescue missions or autonomous exploration, where quick decision making is crucial;
2. **Enhanced Learning Algorithms:** The emphasis on the reward function design implies that future developments in robotic learning algorithms should prioritize a nuanced understanding of environmental cues, leading to more adaptable and intelligent robotic systems;
3. **Flexible Team Deployment:** The scalability of the Q-RTS method allows for the flexible deployment of robotic teams, making it suitable for various scenarios, from small-scale tasks to more extensive collaborative efforts;
4. **Adaptive Systems:** The demonstrated robustness in the face of dynamic changes suggests that robotic systems equipped with Q-RTS can operate in environments where conditions might change unexpectedly, increasing the reliability of these systems.

Finally, this research marks a novel implementation of artificial intelligence, specifically using RL and the innovative Q-RTS as a MARL technique, successfully integrated into the Robotarium platform. To the best of the authors' knowledge, this study represents the first instance of such an implementation, contributing to the advancement of AI methodologies in the Robotarium environment.

Author Contributions: L.C.: resources, formal analysis, validation, and writing—review and editing; G.C.C.: supervision; M.M.D.P.: data curation, formal analysis, investigation, methodology, visualization, and writing—original draft; L.D.N.: validation and writing—review and editing; S.S.: investigation, resources, validation, visualization, and writing—review and editing. All authors have read and agreed to the published version of the manuscript.

Funding: This work has been partially supported by Project ECS 0000024 Rome Technopole, CUP B83C22002820006, NRP Mission 4 Component 2 Investment 1.5, Funded by the European Union—NextGenerationEU.

Acknowledgments: This paper is based on Mohammad Mahdi Dehghan Pir's Master's Thesis: "Design and Development of Multi-Agent Reinforcement Learning Intelligence on the Robotarium Platform".

Data Availability Statement: Dataset available on request from the authors.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

RL	Reinforcement Learning;
MARL	Multi-Agent Reinforcement Learning;
Q-RTS	Q-learning Real-Time Swarm;
MCR	Mean Cumulative Reward.

References

1. Schmidhuber, J. Deep learning in neural networks: An overview. *Neural Netw.* **2015**, *61*, 85–117. [CrossRef]
2. Al-Haddad, L.A.; Jaber, A.A.; Al-Haddad, S.A.; Al-Muslim, Y.M. Fault diagnosis of actuator damage in UAVs using embedded recorded data and stacked machine learning models. *J. Supercomput.* **2024**, *80*, 3005–3024. [CrossRef]
3. Cardarilli, G.C.; Di Nunzio, L.; Fazzolari, R.; Giardino, D.; Matta, M.; Patetta, M.; Re, M.; Spanò, S. Approximated computing for low power neural networks. *Telkomnika Telecommun. Comput. Electron. Control* **2019**, *17*, 1236–1241. [CrossRef]
4. Simonetta, A.; Paoletti, M.C.; Nakajima, T. The SQuaRE Series as a Guarantee of Ethics in the Results of AI systems. In Proceedings of the 11th International Workshop on Quantitative Approaches to Software Quality, Seoul, Republic of Korea, 4 December 2023.
5. Jaber, A.A.; Bicker, R. The optimum selection of wavelet transform parameters for the purpose of fault detection in an industrial robot. In Proceedings of the 2014 IEEE International Conference on Control System, Computing and Engineering (ICCSCE 2014), Penang, Malaysia, 28–30 November 2014; IEEE: Piscataway, NJ, USA, 2014; pp. 304–309.
6. Bertazzoni, S.; Canese, L.; Cardarilli, G.C.; Di Nunzio, L.; Fazzolari, R.; Re, M.; Spanò, S. Design Space Exploration for Edge Machine Learning featured by MathWorks FPGA DL Processor: A Survey. *IEEE Access* **2024**, *12*, 9418–9439
7. EL_Rahman, S.A.; AlRashed, R.A.; AlZunaytan, D.N.; AlHarbi, N.J.; AlThubaiti, S.A.; AlHejeelan, M.K. Chronic Diseases System Based on Machine Learning Techniques. *Int. J. Data Sci.* **2020**, *1*, 18–36. [CrossRef]
8. Gyunka, B.A.; Oladele, A.T.; Adegoke, O. Adaptive Android APKs Reverse Engineering for Features Processing in Machine Learning Malware Detection. *Int. J. Data Sci.* **2023**, *4*, 10–25. [CrossRef]
9. Kaelbling, L.P.; Littman, M.L.; Moore, A.W. Reinforcement learning: A survey. *J. Artif. Intell. Res.* **1996**, *4*, 237–285. [CrossRef]
10. Martínez-Marín, T.; Duckett, T. Fast reinforcement learning for vision-guided mobile robots. In Proceedings of the 2005 IEEE International Conference on Robotics and Automation, Barcelona, Spain, 18–22 April 2005; pp. 4170–4175.
11. Canese, L.; Cardarilli, G.C.; Nunzio, L.D.; Fazzolari, R.; Giardino, D.; Re, M.; Spanò, S. Multi-agent reinforcement learning: A review of challenges and applications. *Appl. Sci.* **2021**, *11*, 4948. [CrossRef]
12. Polydoros, A.S.; Nalpantidis, L. Survey of model-based reinforcement learning: Applications on robotics. *J. Intell. Robot. Syst.* **2017**, *86*, 153–173. [CrossRef]
13. Matta, M.; Cardarilli, G.C.; Nunzio, L.D.; Fazzolari, R.; Giardino, D.; Re, M.; Silvestri, F.; Spanò, S. Q-RTS: A real-time swarm intelligence based on multi-agent Q-learning. *Electron. Lett.* **2019**, *55*, 589–591. [CrossRef]
14. Canese, L.; Cardarilli, G.C.; Di Nunzio, L.; Fazzolari, R.; Re, M.; Spanò, S. Resilient multi-agent RL: Introducing DQ-RTS for distributed environments with data loss. *Sci. Rep.* **2024**, *14*, 1994. [CrossRef]

15. Sutton, R.S.; Barto, A.G. *Reinforcement Learning: An Introduction*; MIT Press: Cambridge, MA, USA, 2018; Volume 13.
16. Watkins, C.J.; Dayan, P. Technical Note, Q-Learning. *Mach. Learn.* **1992**, *8*, 279–292. [CrossRef]
17. Kober, J.; Bagnell, J.A.; Peters, J. Reinforcement learning in robotics: A survey. *Int. J. Robot. Res.* **2013**, *32*, 1238–1274. [CrossRef]
18. Bagnell, J.A.; Schneider, J.G. Autonomous helicopter control using reinforcement learning policy search methods. *Int. Conf. Robot. Autom. (ICRA)* **2001**, *2*, 1615–1620.
19. Haksar, R.N.; Schwager, M. Distributed deep reinforcement learning for fighting forest fires with a network of aerial robots. In Proceedings of the International Conference on Intelligent Robots and Systems (IROS), Madrid, Spain, 1–5 October 2018; pp. 1067–1074.
20. Singh, B.; Kumar, R.; Singh, V.P. Reinforcement learning in robotic applications: A comprehensive survey. *Artif. Intell. Rev.* **2022**, *55*, 945–990. [CrossRef]
21. Kormushev, P.; Calinon, S.; Caldwell, D.G. Reinforcement learning in robotics: Applications and real-world challenges. *Robotics* **2013**, *2*, 122–148. [CrossRef]
22. Low, E.S.; Ong, P.; Low, C.Y.; Omar, R. Modified Q-learning with distance metric and virtual target on path planning of mobile robot. *Expert Syst. Appl.* **2022**, *199*, 117–191. [CrossRef]
23. Yen, G.G.; Hickey, T.W. Reinforcement learning algorithms for robotic navigation in dynamic environments. *ISA Trans.* **2004**, *43*, 217–230. [CrossRef]
24. Levine, S.; Finn, C.; Darrell, T.; Abbeel, P. End-to-end training of deep visuomotor policies. *J. Mach. Learn. Res.* **2016**, *17*, 1334–1373.
25. Tai, L.; Liu, M. A robot exploration strategy based on q-learning network. In Proceedings of the 2016 IEEE International Conference on Real-Time Computing and Robotics (RCAR), Angkor Wat, Cambodia, 6–10 June 2016; pp. 57–62.
26. Wilson, S.; Glotfelter, P.; Wang, L.; Mayya, S.; Notomista, G.; Mote, M.; Egerstedt, M. The Robotarium: Globally Impactful Opportunities, challenges, and Lessons Learned in remote-access, distributed Control of multi-robot Systems. *Mach. Learn.* **2020**, *40*, 26–44. [CrossRef]