

Heuristically-Accelerated Reinforcement Learning: A Comparative Analysis of Performance

Murilo Fernandes Martins^(✉) and Reinaldo A.C. Bianchi

Department of Electrical Engineering – IAAA Group,
Centro Universitário da FEI, São Paulo, Brazil
`murilo@ieee.org`, `rbianchi@fei.edu.br`

Abstract. This paper presents a comparative analysis of three Reinforcement Learning algorithms (Q-learning, $Q(\lambda)$ -learning and QS-learning) and their heuristically-accelerated variants (HAQL, $HAQ(\lambda)$ and HAQS) where heuristics bias action selection, thus speeding up the learning. The experiments were performed in a simulated robot soccer environment which reproduces the conditions of a real competition league environment. The results clearly demonstrate that the use of heuristics substantially improves the performance of the learning algorithms.

Keywords: Reinforcement learning · Heuristics · Robot soccer

1 Introduction

In the past decades a significant amount of algorithms for Reinforcement Learning (RL) have been proposed in the literature. Amongst the proposed techniques, the class of model-free algorithms is the most widely used, since such algorithms do not require a model of the environment with which the agents interact. The Q-learning algorithm [1, 2] is, perhaps, the most well-known model-free algorithm for RL. Although the implementation of the Q-learning algorithm is straightforward and its convergence to optimality has been mathematically proven [1], such convergence is rather slow, infinite time-bounded. This is due to the fact that only one state-action pair has its value updated at each iteration of the algorithm. Furthermore, the larger the state and action spaces, the more visits to the corresponding state-action pairs are necessary for the system to learn and hence the slower the learning process becomes.

Addressing the slow learning convergence of RL algorithms, many techniques have been proposed to speed up the time to converge to a (usually nearly) optimal action selection policy. This paper presents a systematic comparative analysis of performance between the Q-learning algorithm, two variants – $Q(\lambda)$ -learning [1, 3] (which makes use of temporal generalisations) and

The author acknowledges the current support of FAPESP – project n° 2012/12640-1.

QS-learning [4, 5] (which employs spatial generalisation) – and the heuristically-accelerated variants of such algorithms, herein denoted as HAQL [6], HAQ(λ) and HAQS. The experiments were performed in a complex, dynamic, stochastic and real-time simulated robot-soccer environment called SimuroSot¹, the official FIRA Middle League Robot Soccer simulation environment.

This paper is organised as follows. Section 2 presents a formal definition of RL and the algorithms studies, including some key implementation aspects. Next, Sect. 3 introduces the mathematical formulation of the class of Heuristically-Accelerated Reinforcement Learning (HARL) algorithms, whilst details of the experimental setup are presented in Sect. 4. Then, Sect. 5 presents the results obtained, along with a comparative analysis of performance between the implemented algorithms. Lastly, Sect. 6 concludes this paper and presents ongoing and future directions.

2 Preliminaries

Usually, algorithms for RL are formally defined as Markov Decision Processes (MDP) [2]. An MDP consists of a set of states \mathcal{S} , a set of actions \mathcal{A} (both usually finite), a reward function $\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ encoding the desired behaviour of the learning agent, and a transition function in the form $\mathcal{T} : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$. The optimal solution for a deterministic MDP is defined by a policy $\pi^* : \mathcal{S} \rightarrow \mathcal{A}$ which maximises the long term delayed rewards received by the learning agent.

2.1 The Q-Learning Algorithm

Perhaps the most popular RL algorithm, the Q-learning has been extensively studied and widely used across distinct areas. This algorithm defines a function $Q : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ representing the maximum cumulative reward value that can be received by the learning agent, as shown in Algorithm 1. The current state is represented by s , while s' is the resulting state of the environment after executing the action a in state s . The value function $V : \mathcal{S} \rightarrow \mathbb{R}$, where $V(s) = \max_a Q(s, a)$, is the maximum cumulative reward the learning agent can receive from s . The discount factor $\gamma \in [0, 1)$ defines a balance between immediate and future rewards. The learning rate α determines how much the current iteration should change the values in the action-value function $Q(s, a)$. In order for the Q-learning algorithm to converge to an optimal action selection policy (in a stochastic environment), in this paper the learning rate α is updated according to Eq. 1.

$$\alpha_n = \max \left(\frac{1}{1 + \text{visits}_n(s, a)}, 0.125 \right) \quad (1)$$

where α_n is the value of α at the n -th iteration and $\text{visits}_n(s, a)$ is the number of times the learning agent has visited state s and executed action a , also at the n -th iteration. However, in this implementation, whenever the learning rate

¹ <http://www.fira.net/?mid=simurocot>

α falls below $\alpha < 0.125$, its value is kept as $\alpha = 0.125$ so that the agent never stops learning. In This can also be understood as $\lim_{n \rightarrow \infty} \alpha_n = 0.125$.

In the Q-learning algorithm, the optimal policy is defined as $\pi^*(s, a) \equiv \arg \max_{a \in \mathcal{A}} Q(s, a)$. To ensure random exploration of the environment by the learning agent, as opposed to pure exploitation, the action selection rule denoted as $\epsilon - Greedy$ is used (Eq. 2).

$$\pi(s) = \begin{cases} \arg \max_a Q(s, a) & q \leq p \\ a_{random} & \text{otherwise} \end{cases} \quad (2)$$

where $q \in [0, 1]$ is a random value sampled from a uniform distribution and $p \in [0, 1]$ is a parameter determining the exploration/exploitation rate. Also, the action a_{random} is randomly sampled from a uniform distribution of \mathcal{A} .

Algorithm 1. Q-learning algorithm

Input: \mathcal{S} : set of states, \mathcal{A} : set of actions, $r(s, a)$: reward function, $V(s)$: value function, $Q(s, a)$: action-value function.

1. $\forall s \in \mathcal{S}, V(s) \leftarrow 0$ and $\forall s \in \mathcal{S} \wedge \forall a \in \mathcal{A}, Q(s, a) \leftarrow 0$
 2. Observe current state s
 3. **loop**
 4. Select an action $a \in \mathcal{A}$ using the $\epsilon - Greedy$ rule
 5. Execute selected action a in current state s
 6. Receive the reward $r(s, a)$ and then observe next state s'
 7. Calculate temporal difference error $TD(0)$: $e' = r(s, a) + \gamma * V(s') - Q(s, a)$
 8. Update $Q(s, a) \leftarrow Q(s, a) + \alpha_n * e'$ and $V(s) = \max_a Q(s, a)$
 9. $s \leftarrow s'; n \leftarrow n + 1$
 10. **end loop**
-

2.2 The $Q(\lambda)$ -Learning Algorithm

The $Q(\lambda)$ -learning [1, 7] is an algorithm which combines the Q-learning algorithm with temporal generalisations – the Temporal Difference method $TD(\lambda)$ [2] – back-propagating the outcome of a single iteration to a history of multiple state-action pairs recently visited, known as eligibility trace [2].

In the approach presented in [1], the eligibility trace must always be reset whenever an action is randomly selected by the $\epsilon - Greedy$ rule, whereas the approach proposed in [7] does not differentiate a random action from an action that follows a greedy policy. As a consequence, for a fixed, non-greedy policy π , the Q function in the $Q(\lambda)$ -learning will neither converge to Q^π nor to the optimal Q^* , but to some hybrid policy in between. However, according to [2], for a policy which becomes greedy over time the $Q(\lambda)$ -learning proposed in [7] may converge to the optimal Q^* and, furthermore, result in a significantly better performance than the $Q(\lambda)$ -learning algorithm proposed by [1].

In the $Q(\lambda)$ -learning algorithm, the $\lambda \in [0, 1]$ factor is used to discount the temporal difference error $TD(\lambda)$ of the next steps when updating the action-value function $Q(s, a)$. These error values are incrementally calculated using the eligibility trace e . In this approach, a value $l(s, a)$ of eligibility trace is stored for each state-action pair. The $Q(\lambda)$ -learning algorithm implemented in this paper follows the approach defined in [7], which is also detailed in [8]. The implemented algorithm is shown in Algorithm 2.

Algorithm 2. $Q(\lambda)$ -learning algorithm

Input: \mathcal{S} : set of states, \mathcal{A} : set of actions, $r(s, a)$: reward function,
 $V(s)$: value function, $Q(s, a)$: action-value function,
 L : list of (s, a) pairs, that is, the eligibility trace.

1. $\forall s \in \mathcal{S}, V(s) \leftarrow 0; \forall s \in \mathcal{S} \wedge \forall a \in \mathcal{A}, Q(s, a) \leftarrow 0$ and $L \leftarrow \emptyset$
2. Observe current state s
3. **loop**
4. Select an action $a \in \mathcal{A}$ using the $\epsilon - Greedy$ rule
5. Execute selected action a in current state s
6. Receive the reward $r(s, a)$ and then observe next state s'
7. Calculate temporal difference error $TD(0)$: $e' = r(s, a) + \gamma * V(s') - Q(s, a)$
8. Calculate temporal difference error $TD(\lambda)$: $e = r(s, a) + \gamma * V(s') - V(s)$
9. **for all** $(v, u) \in L$ **do**
10. Calculate trace decay $l(v, u) = \gamma * \lambda * l(v, u)$
11. Update $Q(v, u) = Q(v, u) + \alpha_n * e * l(v, u)$
12. **if** $l(v, u) < \zeta$ **then**
13. $L \leftarrow L \setminus (v, u); visited(v, u) \leftarrow 0$
14. **end if**
15. **end for**
16. Update $Q(s, a) = Q(s, a) + \alpha_n * e$
17. Update eligibility trace:
18. $\forall u \neq a, l(s, u) \leftarrow 0; l(s, a) \leftarrow 1$
19. **if** $visited(s, a) = 0$ **then**
20. $visited(s, a) \leftarrow 1; L \leftarrow L \cup (s, a)$
21. **end if**
22. $s \leftarrow s'$
23. **end loop**

In this case, α is updated using the rule defined in Eq. 1, $v \in \mathcal{S}$ is a state, $u \in \mathcal{A}$ is an action and (v, u) are the recently visited state-action pairs. Such pairs are inserted into a doubly-linked list L and, in case the eligibility trace $l(v, u) < \tau$, the corresponding (v, u) pair is removed from the list. Here, $\zeta \geq 0$ is a systematically defined threshold. Removing the (v, u) pairs from the list when their values fall below τ has been observed to considerably speed up each iteration of the algorithm. In order to ensure that the list L does not have duplicate state-action pairs, a binary function $visited(v, u)$ is used to indicate whether a given (v, u) pair has already been visited recently and hence is in L .

By making use of replacing eligibility traces [2] (lines 24 and 25 of Algorithm 2), it is possible to define the maximum number of state-action pairs in L using the values $\gamma\lambda$ and τ . As a result, the complexity of each iteration of the algorithm does not grow linearly as the learning agent visits new state-action pairs, but it is, in the worst case, bounded to a constant length L and a manageable number of updates to the action-value function Q . Also, it is worth noticing that if $\lambda = 0$, the $Q(\lambda)$ -learning algorithm becomes identical to the Q -learning.

2.3 The QS-Learning Algorithm

Whilst the $Q(\lambda)$ -learning algorithm makes use of temporal generalisations, the QS-learning algorithm [4] takes advantage of *a priori* knowledge of spatial similarities employing spatial generalisation to improve the performance of the vanilla Q -learning algorithm. Depending on the similarities between state-action pairs, one single iteration of the algorithm may update more than one (s, a) pair in Q . This similarity is determined by a spreading function $\sigma(v, u, s, a) \in [0, 1]$, which may occur both in the state space \mathcal{S} or the action space \mathcal{A} . However, in this paper, as in [4, 5], only similarities in \mathcal{S} are considered. The spreading function is defined in Eq. 3.

$$\sigma(v, u, s, a) = g(v, s)\delta(u, a), \text{ with } g(v, s) = \tau^d \quad (3)$$

where $\delta(u, a)$ is the Kronecker delta: $\delta(u, a) = 1$ if $u = a$, and $\delta(u, a) = 0$ if $u \neq a$. The function $g(v, s) = \tau^d$ defines the similarity between $v \in \mathcal{S}$ and $s \in \mathcal{S}$, where τ is a constant and d is a factor which quantifies the similarity between v and s . The proof of convergence of the QS-learning algorithm is detailed in [4], and the QS-learning algorithm implemented in this paper is described in Algorithm 3.

Algorithm 3. QS-learning algorithm

Input: \mathcal{S} : set of states, \mathcal{A} : set of actions, $r(s, a)$: reward function,

$V(s)$: value function, $Q(s, a)$: action-value function,

1. $\forall s \in \mathcal{S}, V(s) \leftarrow 0$ and $\forall s \in \mathcal{S} \wedge \forall a \in \mathcal{A}, Q(s, a) \leftarrow 0$
 2. Observe current state s
 3. **loop**
 4. Select an action $a \in \mathcal{A}$ using the $\epsilon - Greedy$ rule
 5. Execute selected action a in current state s
 6. Receive the reward $r(s, a)$ and then observe next state s'
 7. **for all** $v \in \mathcal{S}, u \in \mathcal{A}$ **do**
 8. **if** $\sigma(v, u, s, a) \neq 0$ **then**
 9. Calculate temporal diff. error TD(0) $e' = r(s, a) + \gamma * V(s') - Q(v, u)$
 10. Update $Q(v, u) = Q(v, u) + \sigma(v, u, s, a) * \alpha_n * e'$
 11. **end if**
 12. **end for**
 13. $s \leftarrow s'$
 14. **end loop**
-

As with Q-learning and $Q(\lambda)$ -learning, in the QS-learning algorithm the learning rate α is updated according to Eq. 1. In order to satisfy the conditions to guarantee the convergence of the algorithm, the function $\sigma(v, u, s, a)$ must decay at a faster rate than α . It is important to highlight that when $g(v, s)$ does not define any similarity between v and s , its value is thus $g(v, s) = 0$ for any state v other than s , resulting in a spreading function $\sigma(v, u, s, a) = 0$. As a consequence, the QS-learning algorithm becomes identical to the Q-learning.

3 Heuristically-Accelerated Reinforcement Learning

The use of heuristics to accelerate RL algorithms has firstly been proposed in [6], where the Q-learning algorithm was extended to take advantage of a static heuristic function defined *a priori*. This technique has also been extensively explored in goal-driven navigation tasks [9], in the multiagent robot soccer scenario [10]. The Heuristically-Accelerated Reinforcement Learning (HARL) approach has also been combined with a market-based approach applied to the Robocup 2D simulated domain [11] and with case-based reasoning [12]. In addition, in the Markov Games domain, a Heuristically-Accelerated minimax-Q algorithm (HAMMQ) was proposed in [10] and extensively analysed in domains of distinct complexity [13]. The HAMMQ algorithm is an extension of the minimax-Q algorithm proposed by [14], which is essentially the Q-learning algorithm with a *minimax* rule replacing *max* in the well-known Bellman equations [2].

In the HARL, a heuristic function is defined as $\mathcal{H} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ and is used to bias the action selection during the learning process. This function determines how desirable would the selection of a given action $a \in \mathcal{A}$ be whilst in $s \in \mathcal{S}$. Furthermore, this function may be stationary or non-stationary, as discussed in [9]. Although the heuristic function could be extracted automatically, or from demonstrations of a teacher, it has invariably been defined *a priori* by a specialist, using the knowledge of the domain.

In order for the heuristic function to bias the action selection, the ϵ – *Greedy* rule must be modified, as shown in Eq. 4.

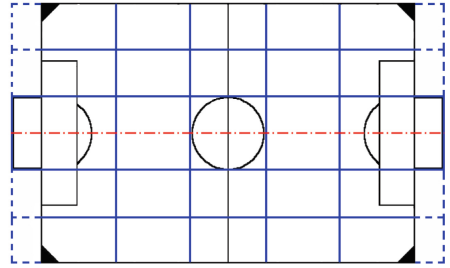
$$\pi(s) = \begin{cases} \arg \max_a [Q(s, a) + \xi * \mathcal{H}(s, a)] & q \leq p \\ a_{random} & \text{otherwise} \end{cases} \quad (4)$$

where $\xi \in \mathbb{R}$ is a real-valued number which weighs the influence of the heuristics in use and it is necessary to guarantee the convergence of the algorithm.

In this paper, the Q-learning variant HAQL, proposed in [6], as well as the herein proposed HAQ(λ) (Heuristically-Accelerated Q(λ)-learning) and HAQS (Heuristically-Accelerated QS-learning) – variants of the Q(λ)-learning and QS-learning, respectively – were implemented using the action selection rule defined in Eq. 4, rule which is the only modification necessary for the vanilla algorithms to take advantage of the HARL approach.



(a) Screenshot of SimuroSot

(b) Discrete 7×5 macro-cells**Fig. 1.** The SimuroSot and diagrammatic representation of the pitch

4 Experimental Setup

A simulated robot soccer domain was proposed in [14], which consists of a two-dimensional grid world of dimensions 5×4 , determining the state space \mathcal{S} , along with 5 possible actions: *Move North (N)*, *Move South (S)*, *Move East (E)*, *Move West (W)* and *Idle (I)*. In this simplistic domain there are no hidden variables, and two players (agent and opponent) compete against each other, with the ball always being in possession of one of them. This domain has been used as testbed in several studies, e.g., [6, 10].

In contrast with the aforementioned domain, the FIRA SimuroSot simulator (Fig. 1a), also used as testbed in other studies (e.g., [13]), is significantly more complex. Firstly, the game is dynamic, which means the players must act upon the current state in real-time, since the ball will not stop moving whilst the players decide which action to execute next. In addition, the SimuroSot simulates cube-shaped robots with 7.5 cm sides and differential-drive kinematics, with 2 teams of 5 robots each, in a 220×180 cm pitch. As in the simplistic domain, only two robots are considered in the game, agent i and opponent j .

The inherent incremental error of odometry sensors is simulated, also giving the SimuroSot a stochastic characteristic. At each iteration, the simulator provides somewhat accurate information referring to the robots' position and orientation, as well as the position of the ball, playing the role of a bird's-eye view computer vision system. Also, low level position control and path planning are not readily available and had to be implemented. The whole system (RL algorithms, low level control and path planning) was implemented in C++.

4.1 Definition of Set of States \mathcal{S} and Actions \mathcal{A}

Although the position and orientation of robots and ball are continuous variables, using such values would result in a markedly large state space, with memory requirements which are not practical when using the vanilla tabular

representation of states. The state space \mathcal{S} was thus discretised to 7×5 symmetric macro-cells (Fig. 1b). Since such regions are larger than the size of robots and ball, both robots and the ball may occupy the same region at the same time. Hence, a given state $s = \langle x_i, y_i, x_j, y_j, x_b, y_b \rangle$ consists of the position of the robots (learning agent i and opponent j) and the ball b .

In this paper, due to the dynamic and stochastic characteristics of the SimuroSot, the action space \mathcal{A} was implemented as a set of behaviour-based actions rather than simple transitions from one discrete region to another. Behaviour-based actions are necessary because, for instance, moving regions whilst controlling the ball in the SimuroSot is far from trivial, since the ball moves freely and game only stops when a goal is scored. Furthermore, the noise from odometry sensors and position of robots and ball may result in a stochastic transition to distinct s' when repeatedly executing a given action a from state s . As a result, executing an implemented behaviour-based action $a \in \{N, S, W, E\}$ means navigating from the macro-cell at which the robot is currently located to the centre of the macro-cell corresponding to the action being executed, whilst I denotes no movement at all. Also, if the execution of an action would result in trespassing the boundaries of the pitch, the execution is considered to be a failure and the outcome is akin to executing I .

In addition, the action space \mathcal{A} defined in [14] was extended with two additional actions: *Fetch Ball* (F) and *Kick to Goal* (K). Executing F results in moving from whatever macro-cell the robot is located towards the ball (regardless of which region the ball is in). The precise desired location of the robot when executing F is defined as immediately behind the ball (considering the scoring goal side). Executing K , on the other hand, requires that the robot be in the same macro-cell as the ball, resulting in failure (i.e., no movement) otherwise. With this pre-condition fulfilled, K results in the robot hitting the ball from such an angle which allows for pushing the ball in a straight line towards the centre of the scoring goal.

Regarding the behaviour-based actions, a vanilla PID controller was implemented for the low level position control of the robots, whilst the navigation layer – which generates waypoints to move from current to desired position and orientation – was implemented using the well-known cubic Bézier curves, which presented very good results at a very low computational cost.

4.2 The Reward (\mathcal{R}), Spreading (σ) and Heuristic (\mathcal{H}) Functions

The reward function \mathcal{R} was determined in such a way that whenever a goal was scored by the agent, a large positive reward value was received. Similarly, whenever a goal was suffered, the agent received a large negative reward. In addition, in order to avoid a sub-optimal, stationary behaviour by the agent, a small-valued negative reward was given for every action which did not result in a goal scored. Also, in order to discourage the agent from selecting the actions which would result in failure (representing potential collisions with the pitch boundary walls – extremely undesirable with real robots), another negative reward value was given when the outcome of an action was a failure.

The values of the reward function \mathcal{R} were defined through experimentation. If $s' = \text{goal scored}$, $r(s, a) = +1000$; when $s' = \text{goal suffered}$, $r(s, a) = -1000$; also, $r(s, a) = 10, \forall s' \neq \text{goal scored} \wedge a \neq \text{failure}$; and if $a = \text{failure}$, $r(s, a) = -50$.

Regarding the parameter values used in this paper, the learning rate α was initially set to 1, decaying according to Eq. 1. The exploration/exploitation ratio $p = 0.2$, as well as the discount factor $\gamma = 0.9$ are values commonly used in the literature [10, 13, 14]. For the algorithms $Q(\lambda)$ -learning and $HAQ(\lambda)$, the factor $\lambda = 0.3$ was determined systematically, following the remarks made in [3]. Similarly, for the QS -learning and $HAQS$ algorithms the spreading function $\sigma(v, u, s, a)$ was defined based on [5]. However, the initial value $\tau = 0.7$ decays according to the rule defined in Eq. 5a.

$$\tau_n = [0.7 - 0.1 * \text{visits}_n(v, u)]^d \quad (5a)$$

$$\lim_{n \rightarrow \infty} \tau_n = 0 \quad (5b)$$

where τ_n is the value of τ at the n -th iteration and $\text{visits}_n(s, a)$ is the number of times the learning agent has visited state v and executed action u at the n -th iteration as well. This way, the rate at which τ decays as the number of iterations n increases is faster than that of α , guaranteeing the convergence requirements, noticing Eq. 5b and recalling that α will never be smaller than 0.125 (Eq. 1).

The values of the similarity quantifier factor d are defined according to the macro-cell at which the opponent is located within v in relation to its location in s , $d = 0$ if $v = s$; $d = 1$ if v_j is adjacent in any cardinal direction of s_j ; $d = 2$ if v_j is adjacent in any diagonal direction of s_j ; $d = \infty$ otherwise, where $v_j \subset v$ and $s_j \subset s$ represent the position $\langle x_j, y_j \rangle$ of the robot opponent (ignoring the orientation) in states v and s , respectively.

Hence, up to 9 updates to the action-value function Q may be done per iteration. In addition, an imaginary horizontal line (axis of symmetry: $y = 2$ – dash-dotted red line in Fig. 1b) geometrically divides the pitch in half, such that experiences of a single iteration in the upper part of the pitch can be spread out to the bottom part by mirroring states and actions, and vice-versa, thus making greater use of spatial generalisation. Mirroring states means calculating the Point Reflection, that is, the isometric involutive affine transformation in the Euclidean space \mathbb{R}^2 with one fixed point determined by $\rho = \langle x_k, y \rangle, \forall k \in \{i, j, b\}$ and $y = 2$, for agent i , opponent j and the ball b . Thus, for each of a maximum of 9 updates, the corresponding state v is symmetrically mirrored to $v^\rho \equiv \text{Ref}_\rho(v) = 2\rho - v$.

To illustrate this case, suppose both robots and the ball are located in the upper-leftmost macro-cell within the pitch, that is, $s = \langle x_i = 0, y_i = 4, x_j = 0, y_j = 4, x_b = 0, y_b = 4 \rangle$. By mirroring this state, both robots and the ball would be located at the bottom-leftmost macro-cell, i.e., $s^\rho = \langle x_i = 0, y_i = 0, x_j = 0, y_j = 0, x_b = 0, y_b = 0 \rangle$.

Mirroring actions, on the other hand, follows a rather simple rule: $a^\rho = S$ if $a \equiv N$; $a^\rho = N$ if $a \equiv S$; otherwise $a^\rho = a$.

In regards to the heuristic function \mathcal{H} , $\xi = 1$ and, as in most of the studies involving HARL algorithms, in this paper \mathcal{H} was defined *a priori*, being as intuitive and concise as possible, as defined in Eq. 6.

$$\mathcal{H}(s, a) = \begin{cases} 100 & \text{if } s_i = s_b \wedge a \equiv K \\ 0 & \text{otherwise} \end{cases} \quad (6)$$

where $s_i = \langle x_i, y_i \rangle$ is the position of the learning robot in s and $s_b = \langle x_b, y_b \rangle$ is the position of the ball, also in s . Thus, the learning robot should tend to select action $a \equiv K$ whenever it is in the same macro-cell as the ball. Notice that using \mathcal{H} as a controller is not a solution and would only be marginally better than random walk. This is because $\forall s_i \neq s_b, \mathcal{H}(s, a) = 0$ and Eq. 4 ($\arg \max_a [\mathcal{H}(s, a)]$) would often result in randomly sampling $a \in \mathcal{A}$ from a uniform distribution.

5 Results and Discussion

In order to compare the performance of the RL algorithms herein discussed, 5 trials of 500 games each were executed for each algorithm: Q-learning, Q(λ)-learning, QS-learning, HAQL, HAQ(λ) and HAQS. The virtual learning robot (the agent) always played against an opponent which chooses actions randomly.

In the SimuroSot each game consists of 5 min regardless of number of goals scored. The timer only stops when a goal is scored, then continuing once the ball and robots are automatically placed on their initial positions. As a result, each trial of each algorithm consumed 72 h (~ 42 h of actual gameplay and ~ 30 h of overhead for game set up, restarting, file saving and so on). Conducting simulations in the SimuroSot requires several mouse clicks, and hence the procedure was automated using a third-party commercial software.

The results of the trials of the Q-learning and HAQL algorithms are shown in Fig. 2, whilst Fig. 3 shows the learning curves of QS-learning and HAQS and Fig. 4 presents the results of Q(λ)-learning and HAQ(λ). The graphs consist of mean and standard deviation (of the 5 trials) of the cumulative goal difference over 500 games.

By analysing Figs. 2 and 3, it is possible to notice the similarity between the Q-learning and QS-learning, as well as the HAQL and

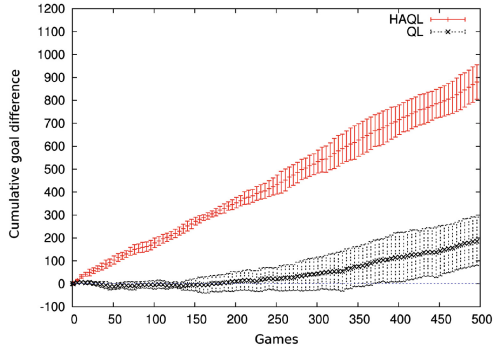


Fig. 2. Q-learning and HAQL

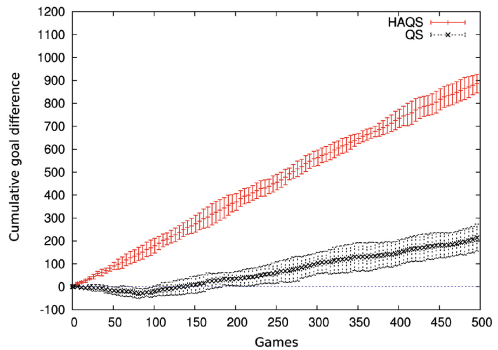


Fig. 3. QS-learning and HAQS

HAQS. This similarity, also noticed by [5], is due to the fact that spreading of experiences only occurs at the beginning of the learning, when the agent has little experience with few large rewards received (scoring or suffering a goal). Since the spreading function decays rapidly (to guarantee convergence), the QS-learning and HAQS quickly become equivalent to the Q-learning and HAQL, respectively. Despite the markedly similar performance of QS-learning and HAQS to Q-learning and HAQL, the advantage of spatial spreading of experiences can be observed by the resulting smaller standard deviation; at the early stages of learning, when visiting previously unvisited states the agent will have received rewards from the spatial spreading, thus preferring to select certain actions over others.

On the other hand, the use of temporal generalisation has a great positive impact in performance when comparing the $Q(\lambda)$ -learning (Fig. 4) with Q-learning and QS-learning. This improvement is also noticeable when comparing the $HAQ(\lambda)$ (Fig. 4) with HAQL and HAQS, though not as prominent. Furthermore, two important remarks can be made from analysing the graphs in Figs. 2, 3 and 4. Firstly, the use of a good heuristic function \mathcal{H} positively biased the action selection of the agent with little experience, thus avoiding an excessively exploratory behaviour at the beginning of the learning process. Secondly, by approximating the curves to straight lines, a considerably higher slope is noticeable with the algorithms using heuristics. The slope denotes the rate at which the agent improves its performance over time towards an optimal policy, and the difference between slopes is the result of the heuristics accelerating the learning process during the early stages. In the longer term, the influence of \mathcal{H} (Eq. 4) will tend to be residual when compared to the value of Q and the slopes of the vanilla algorithms will become equivalent to their heuristically-accelerated counterparts.

In regards to the temporal generalisations in comparison with the spatial generalisations, it is worth noticing that whilst $Q(\lambda)$ -learning and $HAQ(\lambda)$ spread the reward received over a trace of recently visited state-action pairs (multiple iterations) aiming at maximising the reward, the QS-learning and HAQS algorithms spread only a single position of robots and ball, and action executed (single iteration) to other states with a certain similarity (according to σ) with the original state. Therefore, propagation of high long term delayed reward values over time in the QS-learning and HAQS will be similarly slow to the propagation in the Q-learning and HAQL, respectively. On the other hand, the propagation of a trace of experiences, rather than a single experience, directs

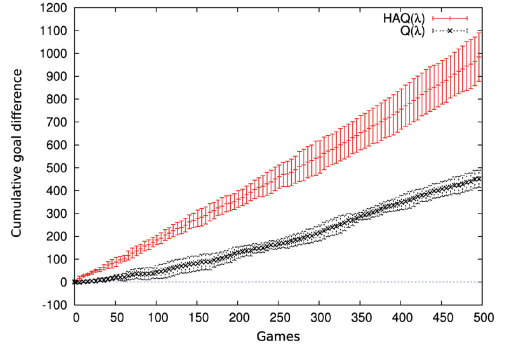


Fig. 4. $Q(\lambda)$ -learning and $HAQ(\lambda)$

the agent towards a path of higher long term rewards, explaining why the $Q(\lambda)$ -learning and $HAQ(\lambda)$ outperformed the other algorithms by a markedly long margin.

6 Conclusion and Future Work

This paper presented a comparative analysis of performance of well-known RL algorithms – Q-learning, $Q(\lambda)$ -learning and QS-learning – and their heuristically-accelerated variants – the previously proposed HAQL and, to the best of the authors’ knowledge, the novel algorithms $HAQ(\lambda)$ and HAQS. The results (obtained in a dynamic, stochastic and fairly realistic simulator) clearly demonstrated that the use of heuristics significantly improves the performance in all the cases from the very beginning, avoiding excessive exploration by the agent at the early stages of the learning process.

Work has already begun testing the HARL approach with real robots, where not only the use of heuristics will be evaluated, but also the transfer of policies $\pi : \mathcal{S} \rightarrow \mathcal{A}$ learnt in simulation to the real robots. Future work will explore the effect of poorly defined heuristic functions (potentially leading to poorly selected actions), as well as the use of human demonstrations as heuristics. The use of function approximations for state space representation will also be addressed.

References

1. Watkins, C.: Learning from delayed rewards. Ph.D. thesis, University of Cambridge, England (1989)
2. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. Adaptive Computation and Machine Learning. MIT Press, Cambridge (1998)
3. Wiering, M., Schmidhuber, J.: Fast online $q(\lambda)$. *Mach. Learn.* **33**(1), 105–115 (1998)
4. Ribeiro, C., Szepesvári, C.: Q-learning combined with spreading: convergence and results. In: ISRF-IEEE International Conference on Intelligent and Cognitive Systems (Neural Networks Symposium), pp. 32–36 (1996)
5. Ribeiro, C., Pegoraro, R., Costa, A.: Experience generalization for concurrent reinforcement learners: the minimax-qs algorithm. In: Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems, pp. 1239–1245. ACM, NY (2002)
6. Bianchi, R.A.C., Ribeiro, C.H.C., Costa, A.H.R.: Heuristically accelerated Q-learning: a new approach to speed up reinforcement learning. In: Bazzan, A.L.C., Labidi, S. (eds.) SBIA 2004. LNCS (LNAI), vol. 3171, pp. 245–254. Springer, Heidelberg (2004)
7. Peng, J., Williams, R.: Incremental multi-step q-learning. *Mach. Learn.* **22**(1–3), 283–290 (1996)
8. Wiering, M., van Hasselt, H.: Ensemble algorithms in reinforcement learning. *IEEE Trans. Syst. Man Cybern. Part B* **38**(4), 930–936 (2008)
9. Bianchi, R., Ribeiro, C., Costa, A.: Accelerating autonomous learning by using heuristic selection of actions. *J. Heuristics* **14**(2), 135–168 (2008)

10. Bianchi, R., Ribeiro, C., Costa, A.: Heuristic selection of actions in multiagent reinforcement learning. In: Proceedings of the 20th International Joint Conference on Artificial Intelligence, pp. 690–696. Morgan Kaufmann Publishers Inc. (2007)
11. Gurzoni Jr, J.A., Tonidandel, F., Bianchi, R.A.C.: Market-based dynamic task allocation using heuristically accelerated reinforcement learning. In: Antunes, L., Pinto, H.S. (eds.) EPIA 2011. LNCS, vol. 7026, pp. 365–376. Springer, Heidelberg (2011)
12. Bianchi, R.A.C., Ros, R., Lopez de Mantaras, R.: Improving reinforcement learning by using case based heuristics. In: McGinty, L., Wilson, D.C. (eds.) ICCBR 2009. LNCS, vol. 5650, pp. 75–89. Springer, Heidelberg (2009)
13. Bianchi, R., Martins, M., Ribeiro, C., Costa, A.: Heuristically-accelerated multiagent reinforcement learning. *IEEE Trans. Cybern.* **44**(2), 252–265 (2013)
14. Littman, M.L.: Markov games as a framework for multi-agent reinforcement learning. In: Proceedings of the 11th International Conference on Machine Learning (ML-94), pp. 157–163. Morgan Kaufmann, New Brunswick (1994)

Towards Autonomous Robotic Systems

14th Annual Conference, TAROS 2013, Oxford, UK,

August 28--30, 2013, Revised Selected Papers

Natraj, A.; Cameron, S.; Melhuish, C.; Witkowski, M.

(Eds.)

2014, XIII, 486 p. 239 illus., Softcover

ISBN: 978-3-662-43644-8