J Intell Robot Syst
DOI 10.1007/s10846-014-0171-1

Batch Reinforcement Learning for Robotic Soccer Using the Q-Batch Update-Rule

João Cunha · Rui Serra · Nuno Lau · Luís Seabra Lopes · António J. R. Neves

Received: 7 July 2014 / Accepted: 17 November 2014

Abstract Reinforcement Learning is increasingly becoming a valuable alternative to tackle many of the challenges existing in a semi-structured, non-deterministic and adversarial environment such as robotic soccer. Batch Reinforcement Learning is a class of Reinforcement Learning methods characterized by processing a batch of interactions. By storing all past interactions, Batch RL methods are extremely data-efficient which makes this class of methods very appealing for robotics applications, specially when learning directly on physical robotic platforms.This paper presents the application of Batch Reinforcement Learning to obtain efficient robotic soccer controllers on physical platforms. To learn the controllers we propose the application of Q-Batch, a novel update-rule that exploits the episodic nature of the interactions in Batch Reinforcement Learning. The approach was validated in three different tasks with increasing difficulty. Results show the proposed approach is able to outperform hand-coded policies, for all the tasks, in a reduced amount of time. Additionally, for one of the tasks, a comparison between Q-Batch and Q-learning is carried out, and results show that, Q-Batch obtains better policies than Q-learning for the same amount of interaction time.

Keywords Batch reinforcement learning · Robot learning · Q-batch · Robotic soccer

J. Cunha · R. Serra · N. Lau · L. S. Lopes · A. J. R. Neves
Department of Electronics, Telecomunications, and Informatics, University of Aveiro, Aveiro, Portugal

J. Cunha (✉) · N. Lau · L. S. Lopes · A. J. R. Neves
Institute of Electronics and Telematics Engineering of Aveiro, University of Aveiro, Aveiro, Portugal
e-mail: joao.cunha@ua.pt

1 Introduction

The development of efficient robotic controllers is a demanding task. Classical approaches involve model-based solutions which require a prior expert knowledge of the system. Additionally, even in the presence of a reliable model, sensor and actuator noise create additional difficulties when fine tuning a controller. In this context, Reinforcement Learning approaches emerge as a valuable alternative. Guided by rewards and penalties the agent is able to learn directly from interacting with the environment. This greatly reduces the required input from the developer.

Robotic competitions such as the RoboCup robotic soccer leagues [1] present a valuable opportunity to apply RL approaches in the context of an extremely competitive and adversarial environment. When applying RL methods in robotics, data efficiency gains extreme importance. As opposed to simulation scenarios, gathering the interaction data in real world carries an associated cost. There are often situations during learning that can result in damage for the environment. Additionally, if the robot requires too many interactions to learn a task, the repeated execution of certain actions can result in a deterioration of the robot physical platform. Generalization is then a very important aspect of any RL method, avoiding the systematic exploration of the entire task search space. While Reinforcement Learning approaches have the potential to solve many problems faced in the multiple competitions of RoboCup, hand-coded solutions are still the norm.

Batch Reinforcement Learning is a class of RL methods that can be combined with function approximators to achieve fast and data-efficient learning solutions. Batch Reinforcement Learning has recently been applied with great success to learning in physical platforms [2–5]. This paper presents some recent developments in this field, namely the application of Q-Batch, a newly developed update-rule in a number of learning tasks involving physical robots. The learning tasks were developed in the context of the CAMBADA [6] project, an on-going research at the University of Aveiro, which developed a team of cooperative mobile robots that competes in the RoboCup Middle Size league.

The paper extends a previous conference publication [7]. Given the focus on the development of learning tasks, further experimental results are provided. With regards to the original publication, simulation results of all experiments are detailed that support the choice of task-specific parameters carried on to experiments in the real robots.

The remainder of this paper is structured as follows: Section 2 presents the main concepts of Reinforcement Learning. Section 3 discusses the proposed approach. The learning tasks developed are discussed in Section 4. Section 5 provides a concise overview of the related work in the application of Reinforcement Learning in Robotics and Section 6 concludes the paper.

2 Reinforcement Learning

Reinforcement Learning [8] is a sub-area of Machine Learning drawing inspiration on biology and animal behavior learning. Although not a new field of study, it has received a lot of attention in recent years from researchers worldwide having sprawled in a multitude of different methods [9, 10].

Reinforcement Learning is usually formalized using a Markov Decision Process (MDP). MDPs are defined by the tuple $\langle S, A, P, R \rangle$: a state set S, an action set A, a probability distribution model of the system dynamics $P(s'|s,a)$ and an immediate reward function $R(s,a,s')$. In every time step t the agent is in a state $s_t \in S$ and takes an action $a_t \in A$. In the following time step, $t+1$, the agent observes a transition to state s_{t+1} and collects a reward r_{t+1}. The key goal of Reinforcement Learning is to find an optimal policy π, defined as mapping from states to actions, $\pi : s \rightarrow a$, that maximizes the cumulative sum of rewards, or the return at time t, $R_t = \sum_{k=0}^{+\infty} \gamma^k r_{t+k+1}$, with a discount factor $\gamma \in [0,1]$.

One approach to solve RL problems involves computing the return by estimating value functions. A value function, defined as $V^\pi(s) = E\{R_t | s_t = s\}$, is a mapping from a given state s to the expected return when applying policy π at s. One solution to update a value function uses Dynamic Programming and Bellman equation and is given by Eq. 1:

$$V_{i+1}^{\pi}(s) = \max_a \sum_{s' \in S} P(s'|s,a)[R(s,a,s') + \gamma V_i^{\pi}(s')] \tag{1}$$

Consequently, given a value function $V(s)$ the corresponding policy $\pi(s)$ is given by $\pi(s) = \arg\max_a V(s)$. There are often times where the model of the environment, $P(s'|s,a)$, is unavailable or too complex to sample. In such cases, model-free approaches estimate action value functions, $Q(s,a)$, or Q-functions. As the name implies, a Q-Function, defined by $Q(s,a) = E\{R_t | s_t = s, a_t = a\}$, is a mapping from a given state s and a given action a to the expected return when applying a at state s and following policy π in the following states. The most common update-rule to estimate a Q-function, is Q-Learning [11], which is defined by Eq. 2:

$$\begin{aligned} Q(s_t, a_t) = \alpha(r_{t+1} + \gamma \max_b Q(s_{t+1}, b)) \\ +(1-\alpha) Q(s_t, a_t) \end{aligned} \tag{2}$$

with α setting the learning rate at which new estimations are incorporated in the existing Q-Function estimation.

The mentioned methods are able to estimate value functions that are guaranteed to converge to the optimal policy if all the states (or state-action pairs) are visited infinitely often. This is a major drawback in physical or real-world systems where the interaction with the environment may be rather costly. On the other hand, convergence is only guaranteed if the value function is represented by a table which limits the applicability of these methods to discrete state spaces.

A class of methods that aim to address the shortcomings highlighted before is Batch Reinforcement Learning. Batch Reinforcement Learning [12] differs from other RL methods in that it estimates a policy π from a set $\mathcal{F}$ composed of all the N transitions sampled from the environment. A key characteristic of Batch RL methods is that the value function is updated synchronously in a global manner by processing the entire set $\mathcal{F}$ in a single step. While purely online methods, such as in Eq. 2, update the Q-function as soon as a transition is observed, Batch RL methods update the Q-function once for all state-action pairs in $\mathcal{F}$. To achieve a synchronous update, a Batch RL update-rule is applied, in a kernel fashion, to all collected transitions, generating a pattern $\mathcal{P}$ mapping state-action pairs to sample a Q-function value, $\mathcal{P} = \{(s_i, a_i), \bar{Q}(s_i, a_i) | i = 1, \ldots, N\}$. This presents an opportunity to apply batch supervised learning methods to build an approximate Q-function $\hat{Q}$, combining function approximators and regression, $\hat{Q} \leftarrow \texttt{approximation}(\mathcal{P})$. The approximate Q-function $\hat{Q}$ is then able to generalize the obtained return to states not visited in $\mathcal{F}$. Since the update is synchronous, the function approximator can be optimized in a global manner using gradient-descent based approaches such as RProp [13] to obtain a robust approximation. Figure 1 presents a visual representation of the Batch Reinforcement Learning framework, which is sub-divided in three distinct modules: the Interaction, Pattern Generation and Batch Supervised Learning. The modules can be implemented differently to realize different Batch Reinforcement Learning methods.

Among other methods, we highlight the Neural Fitted Q Iteration (NFQ) [14], an instance of the class of Fitted Iterations methods [15, 16], which relies on multilayer perceptrons, a powerful function approximator, to represent the approximate Q-function $\hat{Q}$. NFQ builds the pattern $\mathcal{P}$ by applying a Dynamic Programming adapted version of the Temporal Difference Q-Learning update rule (2), over all the collected transitions:

$$\bar{Q}_l(s_i, a_i) = r_i + \gamma \max_b \hat{Q}_l(s_i', b), \forall i \in 1..N \tag{3}$$

where s_i, a_i and s_i' are the current state, action chosen and following state of the i^{th} transition, respectively, while $\bar{Q}_l$ and $\hat{Q}_l$ correspond to the Q-function value samples and the approximate Q-function at loop l.

An analysis of Eq. 3 reveals that it is equal to Eq. 2 with maximum learning rate, $\alpha = 1$. While the learning rate is omitted, the underlying stochastic approximation present in Eq. 2 is solved by the squared error minimization during the approximation phase, effectively yielding an expected return.

A common approach on learning tasks using Batch RL methods involves building the batch incrementally, usually at the termination of an episode, appending the recently collected interactions to the existing batch. The batch is then processed in order to extract an improved policy. The process can then be repeated alternating data collection and policy improvements phases until the obtained policy can fulfill the learning task at hand. Assuming this workflow, then within the same episode the transitions are sampled along connected trajectories.

Assuming the presence of connected trajectories, we can conclude that an update-rule based on transitions, such as Eq. 3, is not particularly efficient at propagating large rewards through a possibly long trajectory. For example, a possible approach to provide

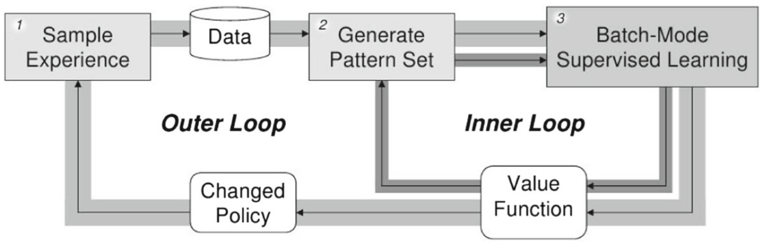

Fig. 1 Batch Reinforcement Learning framework, adapted from [12]

a good starting point to the learning process is to have interactions with the environment generated by a demonstrator policy. However, starting with a random $\hat{Q}_0$ Q-Function, we would need to perform a large number of inner loops, in a similar manner to Experience Replay [17], in order to propagate back the values all the way from the end of the trajectory to the initial states of the trajectory.

A tempting approach is the application of the Monte-Carlo update rules, based on trajectory rollouts [18] such as Eq. 4, since we could easily propagate the value from the final to the initial states of the trajectory.

$$\hat{Q}(s_i, a_i) = R_i, \forall i \in [1, N] \tag{4}$$

Analysing Eq. 4 shows that the computed sample values do not depend on the estimate of an existing value-function, so the update rule does not boostrap. This means the current estimates only depend on the data collected during the previous interaction phase and conversely, the data collected can only be used to evaluate the interacting policy. This imposes that all the experience present in $\mathcal{F}$ must be discarded after learning the value function. Considering applications involving real world physical systems, discarding collected data is highly undesirable.

A well known approach to unify Temporal Difference and Monte-Carlo methods is the application of eligibility traces [8]. An eligibility trace can be regarded as a temporary record of the occurrence of a transition. When performing a backup, eligible transitions propagate a fraction of the observed rewards towards the state being evaluated. While this concept has been extended to the *off-policy* case, such as Watkins' Q(λ), the methods rely on the existence of a policy, considered the current optimal policy, to propagate the eligible rewards. However before a close-to-optimal policy is obtained it may happen that some "good" rewards will not be propagated, which would accelerate convergence to the optimal policy faster.

A detailed and in-depth discussion of Reinforcement Learning is outside of the scope of this paper. Interested readers are referred to [9].

3 Q-Batch

The Q-Batch update-rule has been proposed recently [19] and aims to address the shortcomings highlighted before while assuming the presence of connected trajectories. This requires a rearrangement of the structure of data set $\mathcal{F}$. The basis of the proposed structure are the episodes. The set $\mathcal{F}$ is now composed of N episodes. Each episode i is then a time consistent sequence of T_i states, actions and rewards. This representation allows for a more compact data set $\mathcal{F}$ since the following state of a given transition and the current state of the following transition are now redundant information, thus we can avoid storing following states s' explicitly. Considering the timestep j of episode i, the corresponding state, action and reward are now represented by $s_{i,j}$, $a_{i,j}$ and $r_{i,j}$, respectively.

In essence, Q-Batch attempts to perform a rollout over the trajectory of the episode, while bootstrapping. Each rollout is evaluated according to the concept of n-step returns [11]. This concept builds on the basis that the return can be calculated from an intermediate number of n steps of real rewards and the estimated value of the state visited after n transitions. Therefore a one-step return is based on the first reward and the value of the state one step later, a two-step return is based on the two first rewards and the value of the state two steps later, and so on as shown in Eq. 5:

$$\begin{aligned}
R_t^1 &= r_{t+1} + \gamma V(s_{t+1}) \\
&= r_{t+1} + \gamma \max_{a \in A} Q(s_{t+1}, a) \\
R_t^2 &= r_{t+1} + \gamma r_{t+2} + \gamma^2 \max_{a \in A} Q(s_{t+2}, a) \\
&\vdots \\
R_t^n &= \sum_{i=0}^{n-1} \gamma^i r_{t+1+i} + \gamma^n \max_{a \in A} Q(s_{t+n}, a)
\end{aligned} \tag{5}$$

Historically, n-step returns were deprecated over other update rules, since a backup could only occur after n time steps had passed. In a Batch Reinforcement Learning context however, it provides a valuable alternative, since all backups are performed synchronously and all data required to evaluate n-step returns is available in $\mathcal{F}$.

Instead of trying to find an optimal value for n, which is data dependent, Q-Batch seeks the rollout that yields the maximum return, in other words, $\bar{Q}$ is the maximum n-step return found.

$$\bar{Q}(s_{i,j}, a_{i,j}) = \max_k R^k_{i,j} = \max_k \left(\sum_{l=0}^{k-1} (\gamma^l r_{i,j+1+l}) + \gamma^k \max_{b \in A} \hat{Q}(s_{i,j+k}, b) \right), \quad \forall i \in 1..N, \forall j \in 1..T_i - 1, \forall k \in 1..T - j \tag{6}$$

Performing roll-outs through the trajectories of the episodes increases the complexity of the backups. To reduce this complexity we rewrite the maximum n-step return in the following recursive form:

$$\max_k R^k_t = \max(R^1_t, r_{t+1} + \gamma \max_{k'=k-1} R^{k'}_{t+1}) \tag{7}$$

Combining Eq. 6 with Eq. 7, yields:

$$\begin{aligned}\bar{Q}(s_{i,j}, a_{i,j}) &= \max_k R^k_t \\ &= \max(R^1_t, r_{t+1} + \gamma \max_{k'} R^{k'}_{t+1}) \\ &= \max(R^1_t, r_{t+1} + \gamma \bar{Q}(s_{i,j+1}, a_{i,j+1})) \\ &= r_{t+1} + \gamma \max(\max_b \hat{Q}(s_{i,j+1}, b), \\ &\quad \bar{Q}(s_{i,j+1}, a_{i,j+1}))\end{aligned} \tag{8}$$

Analyzing Eq. 8, we observe that the estimated Q-function value depends on the next immediate reward, the approximated maximum Q-function value in the following state, and the new Q-function value of the following transition. Considering that all backups are performed synchronously, we can take advantage of *memoization*, to store the value of $\bar{Q}(s_{i,j+1}, a_{i,j+1})$, and the recursive relation between maximum n-step returns, to apply Q-Batch in the *inverse* order for every episode in $\mathcal{F}$, thus achieving a constant complexity, comparable to Eq. 3.

In the following section, this update-rule is applied in the development of the robotic soccer controllers. Q-Batch was implemented in the Pattern Generation phase of the Batch Reinforcement Learning framework (Fig. 1) replacing the commonly applied Q-Learning batch variant (3), while the remaining modules of the framework were developed as reported in the literature [2].

4 Learning Tasks

This section presents the developed learning tasks and discusses the obtained controller performance after the learning procedure. All the tasks were applied in the Middle Size League robots of the CAMBADA [6] team. In this league, two teams composed of 5 robots play on a 18 m × 12 m field. The environment is structured, with the elements of the game constituted mainly by uniform colors. The field is green with white field lines, the ball has a dominant color (usually orange or yellow) and the robots are mainly black. Each robot must not exceed a squared base of 52 cm with a maximum height of 80 cm. The robots are completely autonomous being the human-interaction restricted to the human referee commands which are relayed to the robots through a control station via a WiFi network. All sensors of the robots are on-board along with the computing components. The robots can also share information with each other and a control station that acts as a coach.

Each CAMBADA robot, presented in Fig. 2, uses a camera as the main sensor mounted in a catadioptric system, ensuring a 360 degrees field of view around the robot. This allows the robot to perceive most of the environment without requiring the robot to face any particular direction. The robot moves by means of an omnidirectional drive composed of three swedish wheels. Additionally the robot possesses an active mechanism for ball retention.

A key aspect in applying Reinforcement Learning in real robotics, highlighted by Lauer et. al, is to ensure the Markov property. Similarly to other approaches [20], to cope with system delays, we predict the state of the world after the actions effects, to a point where the current decision being made will take effect [21].

In the design of the learning tasks, an available simulation environment was used since it allowed for an easier gathering of data and tuning of the learning parameters. After learning in simulation, the procedure was repeated in the physical robots. To learn the different tasks the Neural Fitted Q Iteration method was applied, using a multilayer perceptron (MLP) to approximate the Q-Function. MLPs output a smooth approximation of training target data. With this in mind, throughout the learning tasks we will apply

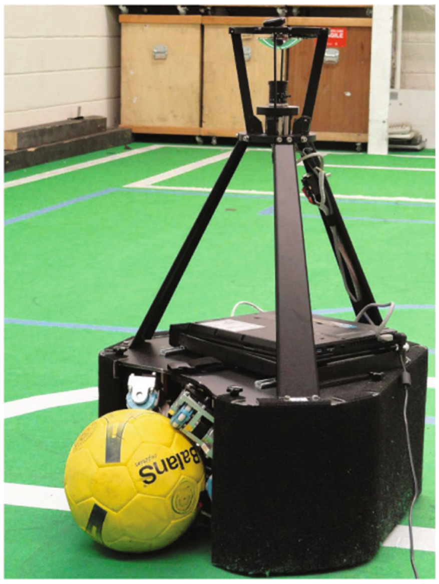

Fig. 2 A CAMBADA Middle Size League robot

a smooth reward function, as proposed in [5], that can generate sample Q-function values easier to approximate by this type of function approximator. Having defined a target state s^{target}, a region of width δ is defined around the target where the cost decreases smoothly, from 95% of a base cost, C down to 0. Instead of rewards, we found it more intuitive to define the learning tasks in terms of costs. Therefore learned policies will choose the actions that minimize the sum of costs. The reward function is defined by Eq. 9 and represented in Fig. 3.

$$r(s,a,s') = C \times \tanh^2\left(|s^{target} - s| \times \frac{\tanh^{-1}(\sqrt{0.95})}{\delta}\right) \tag{9}$$

When possible we followed the guidelines proposed in [22], to specify some of the learning parameters, such the neural networks structure or training epochs. As reported, this choice allows to obtain learned controllers without having to perform an extensive parameter search.

4.1 Rotating to a Point

The first task defined involves having the robot rotate around itself to face a given orientation. While far

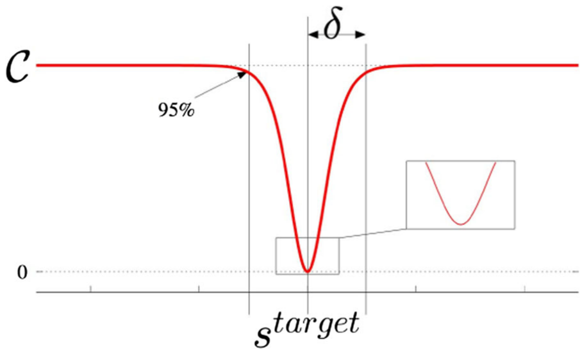

Fig. 3 Smooth reward function used in the learning tasks, adapted from [5]

from a great challenge this presents an opportunity to develop a proof-of-concept learning task in the CAMBADA team.

To minimize the state dimensions and increase generalization, the problem is defined according to a robot centered coordinate frame. Since no linear movement is required, the state is a two-dimensional vector composed of the angular error to the target, and the robot angular velocity, $\langle \theta_e, \omega \rangle$. The action set is composed of angular velocities, in the robot frame. These are then converted to motor setpoints, using differential inverse kinematics, in a separate module of the CAMBADA software architecture. The problem was simplified by providing the agent the observations of the absolute value of its orientation error, and a modified angular velocity value, where whether it is positive or negative means that the velocity is directed to the target orientation or away from it, $s = \langle |\theta_e|, sign(\theta_e) \cdot \omega \rangle$. Additionally, we also restrict the actions to be values greater than or equal to zero, so that the actual command sent to the motors of the robot is always towards minimizing the orientation error. After the action is chosen, it is modified by the sign of the robot orientation error, $\pi(s) = sign(\theta_e) \cdot \arg\max_a \hat{Q}(s,a)$, meaning that when the orientation error is greater than zero, the action that is returned by the agent is applied directly; otherwise, we apply a negative action with the same magnitude as the one returned. This way, we are able to reduce the orientation error range from $[\pi, \pi]$ to $[0, \pi]$. We settled for an action set with three actions: $[0, \frac{\pi}{2}, \pi]$ rad/s, visually represented in Fig. 4.

In this learning task, we intend to minimize the time it takes for the robot to face the desired orientation.

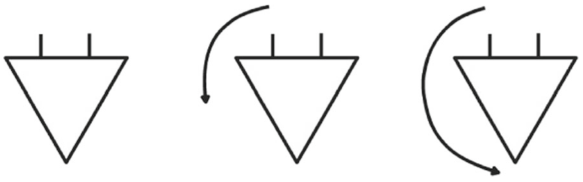

Fig. 4 Action set specified for the rotating learning task

Not only has the robot to minimize the orientation error as fast as possible it also has to actively maintain the error close to zero afterwards. We are then facing a regulation problem. The reward function chosen follows Eq. 9, with $C = 0.01$ and $\delta = 0.1$ rad.

4.1.1 Learning Procedure

For this task, we use a neural network with 3 input neurons, 2 layers of 10 hidden neurons each, and a single output neuron. The 3 input neurons account for the 2 state dimensions and 1 action dimension. The pattern set was approximated with 300 epochs of the RProp algorithm. The Q-Batch update-rule was applied with no discounting, that is $\gamma = 1.0$. A common technique applied in the NFQ framework is the addition of *hint-to-goal* patterns. These patterns are added to the pattern set $\mathcal{P}$ with output zero, that hint the learning agent to the specified states, while also helping to clamp the value of the $\hat{Q}$ close to zero in these states. The *hint to goal* heuristic was used, with an artificial experience set composed by the target state ($\langle \theta_e = 0, \omega = 0 \rangle$), repeated 100 times for each action from the action set.

The learning procedure interleaves phases of experience gathering with policy evaluation. After each interaction phase, 10 loops of policy evaluation are performed, to speed up the learning procedure. Each iteration of experience collection is composed by 10 learning episodes, with each episode lasting for 70 control cycles. In the CAMBADA platform, a new control cycle is started every 0.033 seconds, so, since there are no terminal states, each learning experiment accounts for around 23 seconds of interaction time. At the start of each episode a random target orientation is chosen sampled from a uniform distribution between $[\pi, \pi]$.

4.1.2 Results

This task was applied initially in the simulation environment. A controller able to reliably complete the task was found after 38 learning iterations, which accounts for roughly 14.5 minutes of interaction time. Figure 5 shows a comparison between the learned controller and a hand-tuned PD controller, usually running on the robot, averaged after 10 experiences each. The comparison shows that the hand-coded controller is more robust. We attribute this mainly to the reduced number of actions and also because, for situations when the orientation error is small, these are too strong, i.e. the controller is over actuated. Nonetheless, an acceptable result is achieved. This goes to show how, even with a less-than-optimal action set, it is possible to learn a suitable controller. However, if we were to use this behavior in competition situations, or if our sole objective was to surpass the performance of the explicitly coded behavior, we would need to find a better action set. As this is only a proof-of-concept, we chose to focus our efforts in the remaining tasks.

Learning on the real robot lasted for 24 learning iterations, during which 16800 experience tuples were gathered. This accounts for around 9 minutes of interaction time. In Fig. 6, we can see a comparison between the explicitly coded and the learned behaviors in terms of absolute orientation error, averaged after 10 experiments. Again, we see that the learned behavior is able to reach the target orientation very quickly. The robot omniwheels appear to have a tighter grip on the field surface, so the robot never overshoots as seen in the simulated environment.

Since successful learning controllers can be obtained with a relative ease for this task, we took the opportunity to compare the learning performance of Q-Batch with batch version of Q-Learning, (3). In order to draw conclusions over their performance, 10 learning experiments with a maximum of 50 learning iterations, which interleave one episode of experience collection with one policy evaluation phase, were carried out for each learning rule. The results were aggregated and are displayed in Fig. 7. As we can see, the Q-Batch update rule is faster to achieve a lower mean cost per cycle. Also, after the first 5 learning iterations, the average results for Q-Learning were never better than the average Q-Batch results. This is

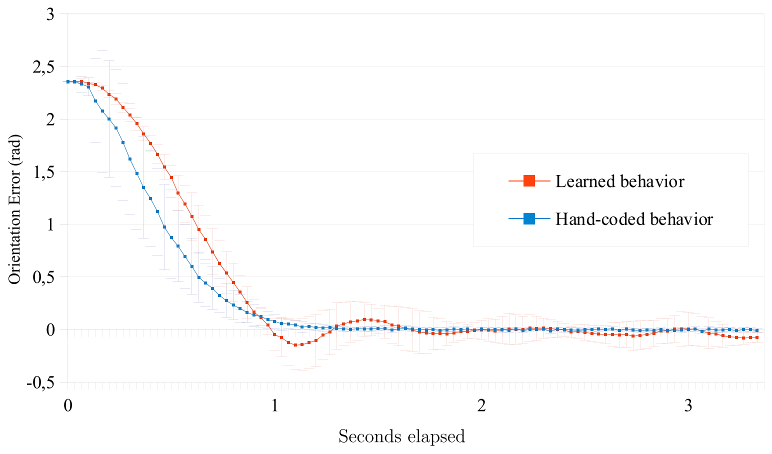

Fig. 5 Performance comparison between the learning and the hand-coded controllers in simulation on the rotating learning task

an interesting observation, because it shows that Q-Batch can reach better performances than Q-Learning in short learning times.

4.2 Dribbling the Ball

Dribbling is one of the basic set of skills of a soccer robot. However to dribble across the field while performing sharp turns without losing the ball is a very challenging task. Additionally, the Middle Size League rules stipulate that a robot may not cover more than a third of the ball, which means that the contact surface between the ball and the robot is very small.

In this task, we intend to develop a low-level controller that is able to dribble a ball in a given direction in the minimum time possible. While the CAMBADA robots have a dribbling behavior, the hand-coded existing solution performs large turns. The controller has to learn how to perform a sharp turn without losing the ball. The task is defined as a regulation problem, this means that after the angular error is close to zero the robot has to maintain the dribbling heading. This can be seen as an extension of an existing learning task in the literature [2].

The state vector of this behavior includes the robot relative orientation error to the target direction, θ, the robot linear velocity in the robot frame (v_x, v_y), where v_y points to the front of the robot, the robot angular velocity,ω, and a signal indicating whether the ball is engaged or not, b. This binary signal is further filtered, so that it is only negative after five sequential control cycles in which the ball was perceived as not engaged by the robot. This is because the robot may lose control of the ball for only one or

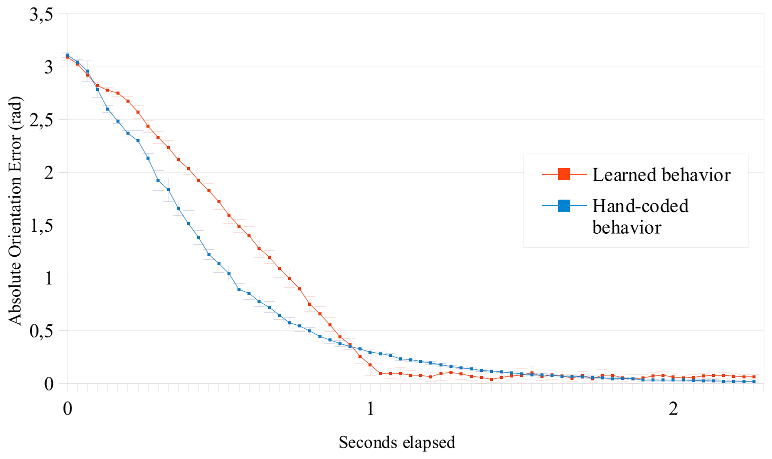

Fig. 6 Performance comparison between the learning and the hand-coded controllers in a real environment on the rotating learning task

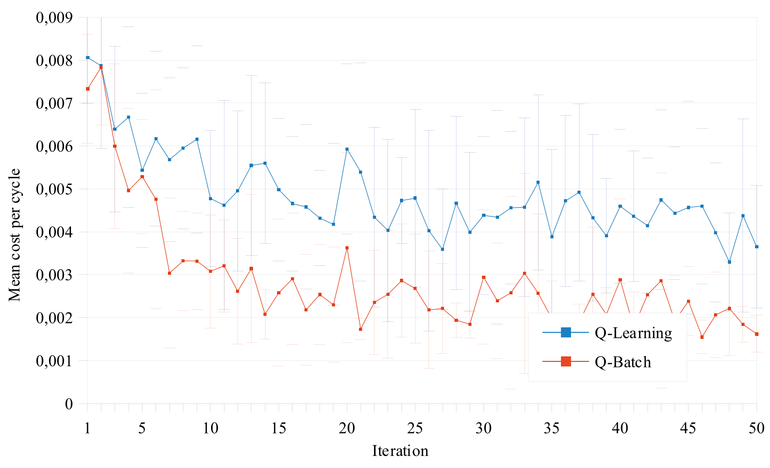

Fig. 7 Learning performance over time of the two update-rules used on the rotating learning task

two control cycles, but regain control right after. In these situations, we choose not to punish the agent, and instead consider as if control of the ball was never lost.

As was described in previous task, we can take advantage of symmetry in order to reduce the number of actions. In this behavior, we again only provide the agent with the observation of absolute value of the orientation error. The state vector is given by, $s = \langle |\theta_e|, sign(\theta_e) \cdot v_x, v_y, sign(\theta_e) \cdot \omega, b \rangle$. Actions are 3-tuples consisting of desired velocity values in the robot coordinate frame, and follow the form $\langle v_x, v_y, \omega \rangle$. Six actions were defined: $\langle 0.0, 2.5, 0.0 \rangle$, $\langle 0.0, 2.0, 2.0 \rangle$, $\langle 0.0, 1.5, 2.5 \rangle$, $\langle 1.5, 1.5, 2.5 \rangle$, $\langle 1.0, 1.0, 3.0 \rangle$ and $\langle 1.0, 1.0, 3.0 \rangle$. As in the previous task the actions correct θ_e only in the range of $[0; 0\pi]$. The actions are then modified to compensate the full range Fig. 8 presents a visual representation of the action set.

The reward function was designed to have the robot reduce the heading error as fast as possible without losing the ball. To achieve this the reward function gives a harsh penalty every time the robot loses the ball, as presented in Eq. 10

$$r(s,a,s') = \begin{cases} 1 & \text{if ball lost} \\ 0.01 \times \tanh^2 \left(|\theta_e| \times \frac{\tanh^{-1}(\sqrt{0.95})}{0.1} \right) & \text{otherwise} \end{cases} \tag{10}$$

4.2.1 Learning Procedure

The learning procedure alternates between 15 episodes of interactions with 10 policy evaluation phases. Each episode lasted for a maximum of 100 cycles, but would terminate earlier if the robot lost the ball during the experiment. The neural network topology used for this behavior consists of 8 input nodes, 2 hidden layers of 20 nodes each and one single output node. The pattern set was approximated over 300 epochs of the RProp algorithm. The Q-Batch update rule was used to generate the pattern set with a discounting factor of $\gamma = 0.99$. An artificial set of "hint" transition tuples was introduced, with state $\langle \theta, v_x, v_y, \omega, b \rangle = \langle 0, 0, 2.5, 0, 1 \rangle$, repeated 100 times for each action of the action set, and target output of zero. At the start of each episode the ball was placed away from the robot to ensure an initial velocity upon ball possession. This way the robot can not perform very sharp turns without loosing the ball. After the ball was engaged, an uniformly random dribbling direction was chosen signaling a start of experience collection.

4.2.2 Results

Learning in the simulated environment was achieved after 34 iterations. Around 17500 experience tuples

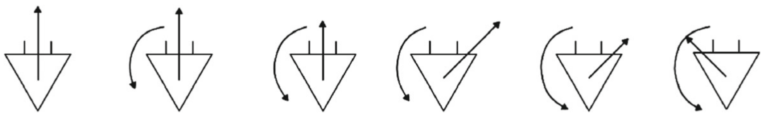

Fig. 8 Action set specified for the dribbling learning task

were gathered during learning, which means the controller was learned under 10 minutes of interaction time. Figure 9 presents a comparison of both the hand-coded behavior and the learned behavior for scenario where the robot had to perform a 180 degrees turn. A square is placed every second on each trajectory, in order to compare how long each behavior took to finish the test.

As observed, the learned behavior is much more sharper than the hand-coded one. In fact, using the learned behavior, the robot is able to rotate around the ball, even while carrying significant speed. This way, the robot could complete a full 180 degree turn in less space. Of particular note is the fact that the learning controller was able to exploit the approximate model of the simulator which accounts for the difference in performance compared to the hand-coded solution, which is tuned to the model of the physical robot.

Learning in the real robot required 18 learning iterations. During learning, the robot collected 6535 experience tuples, or little over 3.5 minutes of experimentation were needed. The whole learning process took around 1.5 hours, including batch training, preparation and execution of learning experiments. Figure 10 shows the performance of the learning controller and the hand-coded solution on the real robot. The visualization follows the same logic as Fig. 9. As we can see, the learned behavior is sharper, and was able make a harder turn. Additionally, it achieved faster speeds after having reached the target orientation, and completed the test after 134 control cycles. The explicitly coded behavior, on the other hand, is softer and needs more space to achieve the same curve, and never moves as fast once it is facing the target direction. It finished the test after 164 cycles. However the increased speed comes at a cost, since the learned controller still looses the ball 30 % of the time as opposed to 10 % on the hand-coded controller.

4.3 Receiving a Pass

Passes are a very important part of a soccer game. For robots to succeed in performing this task, receiving the ball efficiently must be achieved. While humans do it naturally, for a robot this is a very challenging task [23].

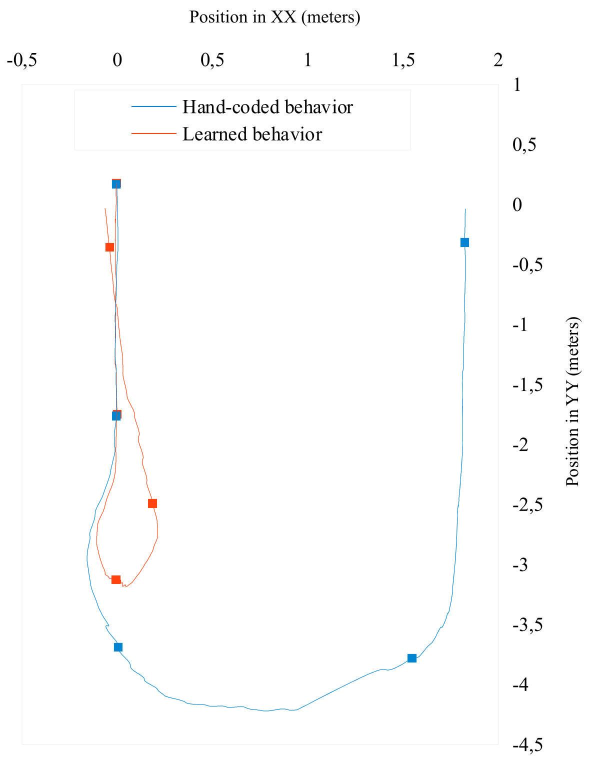

Fig. 9 Performance comparison between the learning and hand-coded controllers in simulation on the dribbling task. Squares are placed on each trajectory every second, allowing for a temporal comparison. The robot started in the (0, 0) position. The dribbling behavior started after the robot reached an YY position of less than -2 meters

Note that the task is not to gain possession of the ball as quickly as possible, in other words to intercept the ball. To receive a pass, the robot has to gain control of the ball with the minimum *movement* possible. This way a pass can be used to gain a strategic advantage during a game. The objective of this behavior is for the robot to position itself in the ball path and absorb the ball speed upon contact, with the minimum required movement, so as to not forfeit its current position. This greatly increases the difficulty of the task. Since what we are trying to optimize is only the linear movement of the agent, we chose to delegate the task of facing the ball to a PID, thus simplifying the problem.

Analysing the learnig task, it is possible to reduce the state space by considering a coordinate frame centered on the ball with the XX axis pointing towards

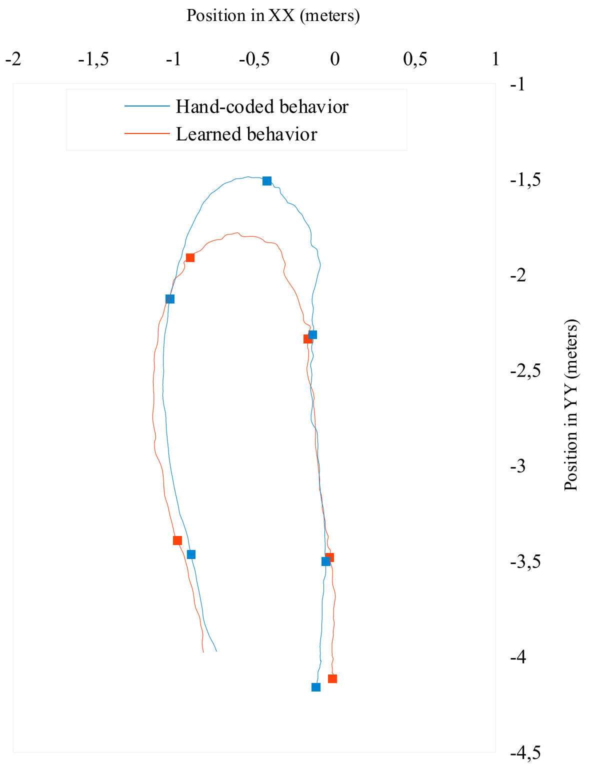

Fig. 10 Performance comparison between learning and hand-coded controllers in a real environment on the dribbling learning task. Squares are placed on each trajectory every second, allowing for a temporal comparison. For each trial, the robot started close to the (0, -4) position. It then grabbed the ball and moved forward. The dribbling behavior started after the robot reached an YY position of greater than -2.5 meters

the movement of the ball. If the ball stops the XX axis points towards the robot. The final formulation of the state vector includes 6 features: the ball speed $\|v_b\|$, robot position (p_x, p_y), robot velocity (v_x, v_y) and the number of control cycles the ball is engaged. We also take advantage of symmetry along the XX axis of the coordinate frame, allowing to represent the component YY of the robot position in absolute values and the same component of the robot velocity to indicate if it is pointing towards the ball path or not.

The fact that we model the problem by considering coordinates as being in the axis of movement of the ball, allows us to simplify actions and reduce the size of the action set. Actions are defined to be two dimensional vectors of the form $\langle v_x, v_y \rangle$, which correspond to target velocity values in the specified coordinate frame. This means that actions should have the same effect regardless of the orientation of the robot, thus taking advantage of the CAMBADA holonomic motion. The action set includes 6 actions: $\langle 0, 0 \rangle$, $\langle 0, 1 \rangle$, $\langle 1, 0 \rangle$, $\langle 1, 1 \rangle$, $\langle 0, 0.5 \rangle$ and $\langle 1, 0.5 \rangle$. Positive v_x actions move the robot away from the ball along its path to absorb the ball incoming velocity, and negative v_y actions move the robot towards the ball path. Figure 11 presents a visual representation of the pass learning task.

The reward function encodes a series of penalties that are combined to give rise to the reception of the ball. We provide a penalty according to YY component of the robot position, p_y, to force the robot onto the ball path as soon as possible. To prevent the movement of the robot along the axis of the ball movement, a penalty is added according to the robot velocity in XX component, v_x. A constant term is added to force the robot to move if the ball stops. Additionally to ensure that the robot attempts to get the ball, a large penalty is given if the ball moves too far away from the robot, $p_x < -0.5$, where we consider that the pass has failed. If the ball is engaged for more than 5 cycles no penalty is given. The overall reward function is presented in Eq. 11.

$$
\begin{aligned}
r(s,a,s') &= \begin{cases} 0 & \text{ball received} \\ 1 & \text{if ball lost} \\ 0.01 + r_y + r_x & \text{otherwise} \end{cases} \\
r_y &= 0.01 \times \tanh^2\left(|p_y| \times \frac{\tanh^{-1}(\sqrt{0.95})}{0.1}\right) \\
r_x &= 0.01 \times \tanh^2\left(|v_x| \times \frac{\tanh^{-1}(\sqrt{0.95})}{0.1}\right)
\end{aligned}
\qquad (11)
$$

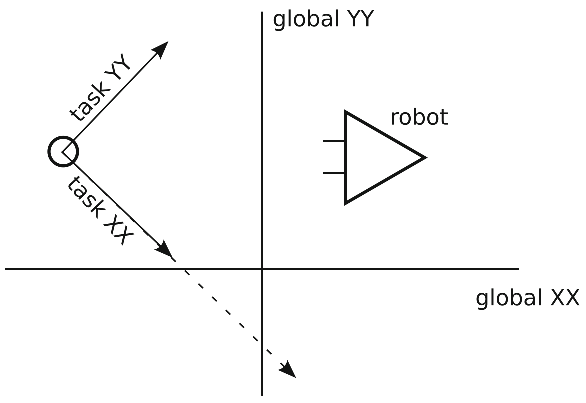

Fig. 11 Visual representation of the pass learning task

4.3.1 Learning Procedure

To learn this behavior, the robot collects experiences in sets of 10 episodes, each with a maximum of 100 control cycles. An episode ends sooner if the ball is lost or caught. After the interaction phase ended, 10 iterations of policy evaluation were performed. The Q-Batch update rule was used to generate pattern sets, with a discount factor of $\gamma = 1.0$. The pattern set was approximated over 300 RProp epochs. The neural network used to approximate the Q-function has 8 input nodes, 2 hidden layers of 20 nodes each and an output layer with a single node.

Learning in the simulator allowed for a controlled repeatability of the experiments. During learning the robot positioned in the center of the field and with the ball placed randomly, being the initial distance between them no less than 4 meters and no greater than 6 meters. The ball starts moving to a target location, which is determined randomly around the robot, being at most 1 meter away from it. This way, we try to reproduce situations where the passer produces a somewhat inaccurate kick. We allow for some control cycles to pass before starting the actual learning episode, to reduce noisy measurements of the ball velocity.

When learning in the real platform the ball was rolled down a ramp at different heights to ensure different initial ball velocities. Similarly to the simulated environment, the ball was thrown at different directions around the robot to ensure a robust ball reception controller.

4.3.2 Results

Using the simulator, 34 learning iterations were needed to obtain a good control policy. The learning agent needed less than 9 minutes of interaction time, and a total number of gathered experience tuples close to around 16000. The hand-coded behavior is able to receive 80 % of the passes performed. The learning behavior is only able to capture 50 % of the passes. Since our goal in simulation is to obtain a proof of concept of the task definition, we chose to advance to learning in the real platform, even though there is still room for improving the learning policy. Figure 12 presents a comparison of the performance of the two controllers. For a better visualization, squares and circles are placed on the robot and ball trajectories, respectively, every third of a second. Also, circles with a dark outline represent the ball under the control of the robot. We can observe that compared to the hand-coded behavior, the learned behavior, attempts to minimize the backward movement, as expected from the specification of the rewards function.

In the real platform, a suitable policy was obtained after 28 learning iterations. This amounted to close to 12 minutes of interaction time, during which 21303 experience tuples were collected. The actual time involved, including preparation and off-line Q-function approximation, was around 2 hours.

The learned solution outperforms the existing hand-coded solution, when receiving passes on the real platform. While the hand-coded solution is only able to receive 10 % of the passes, the learned controller was able to receive 50 % of the performed passes. This is mostly justified by the amount of noise present in the ball velocity estimation, to which the hand-coded solution seems to be more susceptible. Figure 13 presents a comparison of the performance of the two controllers. The visualization follows the same logic as in Fig. 12. As it is visible, the learned behavior is faster to move to its final position, where it grabs the ball safely. The hand-coded behavior, on the other hand, starts by moving sideways, but then changes the direction of its movement and starts backing up. In this trial, the hand-coded behavior failed to grab the ball, as it was not aligned with its trajectory at the moment of impact.

5 Related Work

Reinforcement Learning has been the focus of intense research as a solution to sequential decision problems. While classical applications focused on finite discrete environments, Reinforcement Learning has since been applied to continuous domains, specially in Robotics.

The application of Reinforcement Learning approaches in the Robotics field raises several challenges. Learning environments can easily be composed by a large number of continuous dimensions. Additionally, one can rarely observe the true state of the environment, either because of incomplete observability or due to noise corrupting observations. Another challenge is interaction with the environment using physical platforms is simultaneously tedious

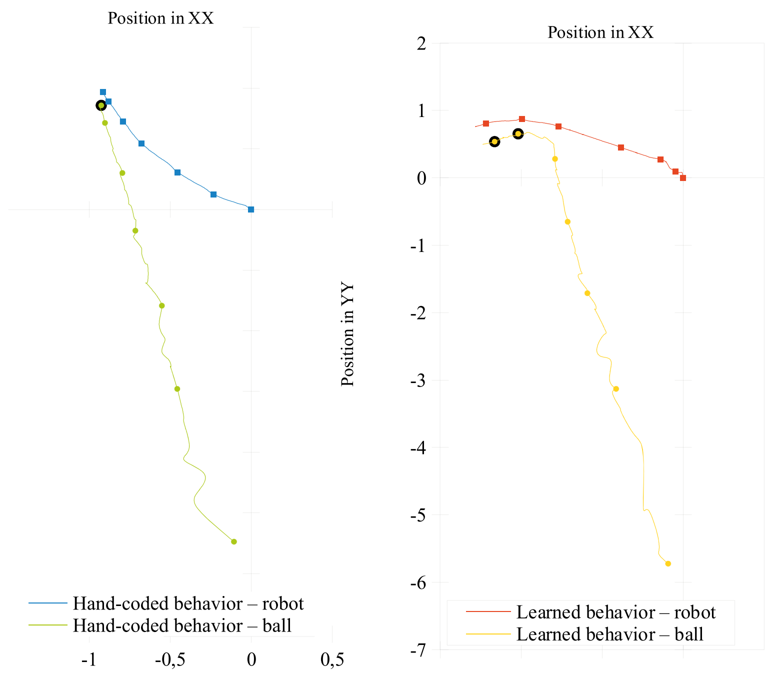

Fig. 12 Comparison of the hand-coded and the learned behavior when receiving a pass in simulation. Both the robot and the ball trajectories are marked with squares and circles every third of a second. A circle with a black outline signifies the robot has grabbed the ball. The receiving behavior was enabled shortly after the robot detected that the ball started moving

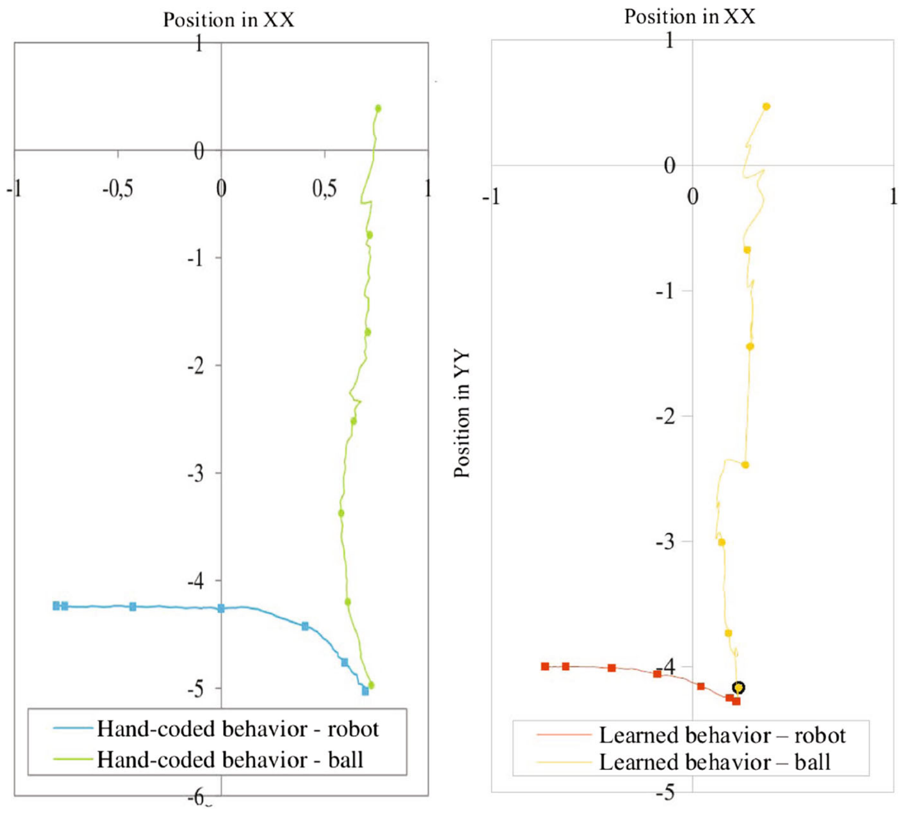

Fig. 13 Comparison of the hand-coded and the learned behavior when receiving a pass in the real platform. Both the robot and the ball trajectories are marked with squares and circles every third of a second. A circle with a black outline signifies the robot has grabbed the ball. The receiving behavior was enabled shortly after the robot detected that the ball started moving

and costly. To minimize the required experience, domain knowledge is usually exploited in various forms, from state and action representations to the specification of reward functions.

Two opposing trends dominate the literature: policy search methods and value function based methods, both with their inherent strengths and weaknesses. Applications are usually related to robotic control and remarkable successful examples include, among others, helicopter control [24], table tennis [25], autonomous driving [4, 26] and also robotic soccer [2, 27]. In the context of RoboCup, we highlight the Brainstromers Tribots team that successfully applied Reinforcement Learning to solve diverse sub-tasks in the Simulation and the Middle Size leagues, winning several championships. A thorough and recent overview of example applications of Reinforcement Learning in Robotics can be found in [28].

6 Conclusion

This paper presents the application of the Q-Batch update-rule to learn robotic soccer controllers in the context of Batch Reinforcement Learning. Q-Batch was recently proposed, and assumes the agent interacts with the environment in episodes composed of connected trajectories. The update-rule performs trajectory rollouts to propagate rewards to the initial states faster. This introduces a minimal computational cost increase in the form of an additional maximization, when compared to Q-learning while being able to reuse the data collected, and to perform off-policy backups. This paper represents the first application of this update-rule in real robotics.

Three tasks were developed, with increasing difficulty. The design of the learning tasks is described, with a focus on hardware abstraction, allowing other teams to implement the tasks on different platforms.

Since there are a considerable number of parameters to tune before learning can be efficiently obtained, a simulation environment was used in an initial step to obtain a good set of parameters. This proved very valuable as it prevented a possible over-use of the real platform during the parameter tuning phase.

After the learning parameters were obtained, learning was performed on the real robot. The obtained results show that efficient controllers able to perform the desired tasks were obtained in a reduced amount of time. The obtained learned policies were also able to outperform existing hand-coded controllers. In one of the task a comparison between Q-Batch and Q-Learning was carried out, with the former being able to obtain policies that on average incur in less penalties than the latter, for the same interaction time.

Future work will involve a broader comparison of Q-Batch in different learning tasks, with existing update-rules in the literature, in order to better perceive the advantages and limitations of the update-rule. The successful application of the Q-Batch, update-rule also opens up research opportunities for the development of additional Reinforcement Learning tasks in the CAMBADA project, addressing learning more complex control tasks, and also learning at higher levels of reasoning.

Acknowledgments This work was supported by project Cloud Thinking (funded by the QREN Mais Centro program, ref. CENTRO-07-ST24-FEDER-002031).

References

1. Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I., Osawa, E., Robocup, H.M.: A challenge problem for ai. AI mag. **18**(1), 73 (1997)
2. Riedmiller, M., Gabel, T., Hafner, R., Lange, S.: Reinforcement learning for robot soccer. Auton. Robot. **27**(1), 55–73 (2009)
3. Bonarini, A., Caccia, C., Lazaric, A., Restelli, M.: Batch reinforcement learning for controlling a mobile wheeled pendulum robot. In: Bramer, M. (ed.) Artificial Intelligence in Theory and Practice II, IFIP 20th World Computer Congress, vol. 276 of IFIP, pp. 151–160 Milano, Italy, Springer. (2008)
4. Lauer, M.: A case study on learning a steering controller from scratch with reinforcement learning. In: Intelligent Vehicles Symposium (IV), 2011 IEEE, pp. 260–265. IEEE (2011)
5. Hafner, R., Riedmiller, M.: Reinforcement learning in feedback control. Mach. Learn. **84**, 137–169 (2011)
6. Neves, A.J.R., Azevedo, J.L., Cunha, B., Lau, N., Silva, J., Santos, F., Corrente, G., Martins, D.A., Figueiredo, N., Pereira, A., Almeida, L., Lopes, L.S., Pinho, A.J., Rodrigues, J.M.O.S., Pedreiras, P.: Robot Soccer, chapter CAMBADA soccer team: from robot architecture to multiagent coordination, pp. 19–45. I-Tech Education and Publishing, Vienna (2010)
7. Cunha, J., Serra, R., Lau, N., Lopes, L.S., Neves, A.J.R.: Learning robotic soccer controllers with the q-batch update-rule. In: Proceedings of International Conference on Autonomous Robot Systems and Competitions (ICARSC 2014), pp. 134–139. Espinho, Portugal (2014)

8. Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction. MIT Press, Cambridge (1998)
9. Wiering, M.A., van Otterlo, M. (eds.).: Reinforcement Learning: State of the Art, volume 12 of Adaptation, Learning, and Optimization. Springer, Berlin (2012)
10. Szepesvári, C.: Algorithms for Reinforcement Learning. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool (2010)
11. Watkins, C.J.C.H.: Learning from Delayed Rewards PhD thesis. University of Cambridge, Cambridge (1989)
12. Lange, S., Gabel, T., Riedmiller, M.: Batch reinforcement learning. In: Wiering, M., van Otterlo, M. (eds.) Reinforcement Learning: State of the Art, chapter 2, pp. 45–74. Springer, Berlin (2012)
13. Riedmiller, M., Braun, H.: A direct adaptive method for faster backropagation learning: the RPROP algorithm. In: Ruspini, H. (ed.) Proceedings of the IEEE International Conference on Neural Networks, pp. 586–591. San Francisco, CA (1993)
14. Riedmiller, M.: Neural fitted Q iterationfirst experiences with a data efficient neural reinforcement learning method. In: Gama, J., Camacho, R., Brazdil, P., Jorge, A., Torgo, L. (eds.) Proceedings of the european conference on machine learning, vol. 3720 of lecture notes in computer science, pp. 317–328, Springer (2005)
15. Gordon, G., Prieditis, A., Russel, S.: Stable function approximation in dynamic programming. In: Proceedings of the 12th Internation Conference on Machine Learning (ICML 1995), pp. 261–268, Tahoe City, USA (1995)
16. Ernst, D., Geurts, P., Wehenkel, L.: Tree-based batch mode reinforcement learning. J. Mach. Learn. Res. **6**, 503–556 (2005)
17. Lin, L.-J.: Self-improving reactive agents based on reinforcement learning, planning and teaching. Mach. Learn. **8**(3-4), 293–321 (1992)
18. Tesauro, G., Galperin, G.R.: On-line policy improvement using Monte Carlo search. In: Neural information processing systems (NIPS), pp. 206–221, Denver (1996)
19. Cunha, J., Lau, N., Neves, A.J.R.: Q-Batch: initial results with a novel update rule for Batch Reinforcement Learning. In: Advances in Artificial Intelligence - Local Proceedings, XVI Portuguese Conference on Artificial Intelligence, Azores, Portugal, pp. 240–251 (September 2013)
20. Lauer, M., Langue, S., Riedmiller, M.: Motion estimation of moving objects for autonomous mobile robots. In: Kunstliche Intelligenz, vol. 20, pp. 11–17 (2006)
21. Cunha, J., Lau, N., Rodrigues, J.M.O.S., Cunha, B., Azevedo, J.: Predictive control for behavior generation of omni-directional robots. In: Progress in Artificial Intelligence, 14th Portuguese Conference on Artificial Intelligence, vol. 5816 of Lecture Notes in Artificial Intelligence, pp. 275–286, Aveiro, Portugal. Springer-Verlag Berlin / Heidelberg. (2009)
22. Riedmiller, M.: 10 steps and some tricks to set up neural reinforcement controllers. In: Neural Networks: Tricks of the Trade (2nd ed.), pp. 735–757 (2012)
23. Corrente, G., Cunha, J., Sequeira, R., Lau, N.: Cooperative Robotics: Passes in robotic soccer. In: Proceedings of 13th International Conference on Autonomous Robot Systems and Competitions, pp. 82–87. Lisbon, Portugal (2013)
24. Ng, A.Y., Coates, A., Diel, M., Ganapathi, V., Schulte, J., Tse, B., Berger, E., Liang, E.: Autonomous inverted helicopter flight via reinforcement learning. In: International Symposium on Experimental Robotics, pp. 363–372. Springer, Singapore (2004)
25. Peters, J., Schaal, S.: Policy gradient methods for robotics. In: Proceedings of the IEEE/RSJ international conference on intelligent robots and systems. IEEE Press, Beijing, China (2006)
26. Riedmiller, M., Montemerlo, M., Dahlkamp, H.: Learning to drive in 20 minutes. In: Proceedings of the FBIT 2007 conference. Springer, Jeju, Korea (2007)
27. Hester, T., Quinlan, M., Stone, P.: Generalized model learning for reinforcement learning on a humanoid robot. In: IEEE International Conference on Robotics and Automation (ICRA) (2010)
28. Jens Kober, J., Bagnel, A., Peters, J.: Reinforcement learning in robotics: A survey . Int. J. Robot. Res. **32**(11), 1238–1274 (2013)