Quasi-Online Reinforcement Learning for Robots

Bram Bakker*, Viktor Zhumatiy†, Gabriel Gruener‡, and Jürgen Schmidhuber†§
*Informatics Institute, University of Amsterdam, the Netherlands, bram@science.uva.nl
†IDSIA, Manno-Lugano, Switzerland, {viktor, juergen}@idsia.ch
‡CSEM, Alpnach, Switzerland, gabriel.gruener@csem.ch
§TU Munich, Germany, juergen.schmidhuber@in.tum.de

***Abstract*— This paper describes quasi-online reinforcement learning: while a robot is exploring its environment, in the background a probabilistic model of the environment is built on the fly as new experiences arrive; the policy is trained concurrently based on this model using an anytime algorithm. Prioritized sweeping, directed exploration, and transformed reward functions provide additional speed-ups. The robot quickly learns goal-directed policies from scratch, requiring few interactions with the environment and making efficient use of available computation time. From an outside perspective it learns the behavior online and in real time. We describe comparisons with standard methods and show the individual utility of each of the proposed techniques.**

I. Introduction

Reinforcement learning (RL) [9] is an attractive technique for robots, because it allows them to autonomously learn a great variety of tasks based on a straightforward trial and error process and simple scalar reward signals. However, one of the main problems is that standard RL techniques require many learning iterations, i.e. many state-action-reward-next state interactions with the environment. In the case of robots, each interaction with the environment is typically very expensive (in terms of time), making standard RL techniques impractical.

This paper reports on work on accelerating RL using an innovative combination of the following techniques and applying them to a real robot:

- Dyna-like online model learning [8], [6]
- Prioritized Sweeping (PS) [2], [5]
- Directed exploration [8], [2], [6]
- Transformed reward functions [3]

In a sense, we investigate how well reinforcement learning from scratch can work for robotics, using these techniques, without resorting to extensive *a priori* learning or programming.

One contribution of this paper lies in the novel combination of techniques. Secondly, these techniques have, by themselves, rarely or never been investigated on real robots, and we describe adaptations to make them applicable and particularly well-suited to real robots. Thirdly, we provide a systematic evaluation of their individual contributions, and compare them to standard techniques.

The investigated combination of techniques allows for *quasi-online reinforcement learning*: as the robot is exploring its environment, in the background (in the control computer's idle time) a model of the environment is built on the fly as new experiences arrive, and the policy is trained concurrently based on this model. Directed exploration based on the model and policy learned so far, and a reward function transformed in a principled way, provide additional speed-ups. The result is that the robot learns very quickly, both in terms of required environment interactions and in terms of computation, and from an outside perspective learns the behavior online and in real time.

In this work we formalize the robot task as a Markov Decision Process (MDP). The next section briefly reviews MDPs and corresponding standard solution methods, value iteration and Q-learning. Section III describes, in turn, each of the investigated techniques to accelerate RL. Section IV describes the robot and its task. Section V describes experiments done in simulation and with a real robot. Section VI, finally, presents a general discussion of the results and possible future work.

II. MDPs and standard solution techniques

A. MDPs

The robot task is formalized as a Markov Decision Process (MDP). An MDP $\mathcal{M}$ is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} \rangle$. $\mathcal{S}$ is a finite set of states s, some of which may be terminal states. $\mathcal{A}$ is a finite set of actions a, whose availability may depend on the state. $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ defines the state transition function that describes the probability $p(s'|s, a)$ that the system will move from state s to s' after performing the action $a \in \mathcal{A}$. $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$ defines the expected immediate real-valued reward $r(s, a, s')$ when action a is taken in state s and the transition to s' is made.

The objective is to determine a policy $\pi : S \rightarrow A$ which at discrete time step t selects an action a_t given the state s_t and which maximizes the expected discounted future cumulative reward, or return: $r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... = \sum_i \gamma^i r_{t+i}$, where $\gamma \in [0, 1]$ is a factor which discounts future rewards.

B. Value iteration

MDPs with known state transition functions and reward functions can be solved optimally using dynamic programming methods. Because it uses a model, this approach may be called model-based RL.

Dynamic programming iteratively computes the value function $Q(s, a)$, which represents the estimate of the expected return attainable from each state. It is guaranteed to converge to the optimal value function $Q^*(s, a)$, which represents the maximum attainable expected return. One well-known method,

value iteration, repeatedly sweeps through the state-action set of the MDP and updates each state-action value according to

$$Q(s,a) \leftarrow \sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma \max_{a'} Q(s',a')] \quad (1)$$

until the largest change in value of any of the state-action pairs, Δ, is smaller than a small constant threshold. After convergence, the optimal policy is followed by simply taking the greedy action in each state s: $a^* = \arg\max_a Q^*(s,a)$. With n states and a maximum of m admissible actions for any state, value iteration requires for each sweep through the state space at most $O(mn)$ operations in the deterministic case and $O(mn^2)$ operations in the stochastic case. Because of this, it can become very slow with large numbers of states.

C. Q-learning

When a model of the environment is not available, one may learn value functions and/or policies directly from experience, without using a model. This is called direct or model-free RL.

The most widely used model-free RL algorithm is Q-learning [10]. The basic idea is to incrementally estimate values of state-action pairs, Q-values, based on experienced rewards in the environment and the currently estimated Q-values. When an action a is taken in state s, next state s' is observed, and reward r is received, the corresponding Q-value is updated by

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)] \quad (2)$$

where α is a learning rate parameter. Convergence to the optimal values is guaranteed under certain standard conditions [10], [9]. However, typically many interactions with the environment are required for convergence to a good or optimal value function and policy.

III. Techniques for accelerating RL

A. Dyna

The Dyna framework [8], [6] assumes, like Q-learning, that no model of the environment is available *a priori*, and learns from experience. However, unlike Q-learning, it uses each state-action-reward-next state experience to not only update the policy, but also to simultaneously learn a predictive model of the environment; and this model is used concurrently to train the policy. In Sutton's [8] simple Dyna version, each real experience leads to one value update (eq. 2), as it does in standard Q-learning, but now each real experience is followed by n simulated experiences produced by the model, all leading to additional standard Q-learning updates.

A related, standard approach in applying RL and other learning techniques to robots [4] is to build or learn a model of the environment in an initial, system identification phase. In the next phase, model-based learning techniques are used to learn the policy offline. Finally, the resulting policy is transferred to the robot. In contrast to that approach, Dyna allows the RL method to explore and learn truly autonomously and online, without different phases assigned by humans. Model and policy are learned concurrently with exploration of the environment, which means that model uncertainty and the current policy can guide exploration. In this way model uncertainty can automatically be corrected, and the model can become most precise in those areas of the state-action space which are most relevant for the policy, because exploration will focus on those areas. The disadvantage of Dyna, on the other hand, is that early experience can lead to an initially imperfect and incomplete model, which may bias the learning of the policy in inappropriate ways. Below we provide solutions for this issue.

B. Prioritized Sweeping

Prioritized Sweeping (PS) [2], [5] can be understood as an extension of the Dyna framework. It assumes that the model that is learned can be used for value iteration, i.e. it must contain explicit state transition probabilities $p(s'|s,a)$ and expected rewards $r(s,a,s')$.

Initialize $Q(s,a)$, $p(s'|s,a)$, and $r(s,a,s')$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$
loop
  $s \leftarrow$ current (nonterminal) state
  $a \leftarrow$ exploration(s,Q)
  Promote (s,a) to top of priority queue
  Execute action a
  while there is time and priority queue not empty **do**
    Remove top state-action pair from priority queue and call it (s_p,a_p)
    $Q_p \leftarrow \sum_{s'_p} p(s'_p|s_p,a_p)[r(s_p,a_p,s'_p)+\gamma \max_{a'_p} Q(s'_p,a'_p)]$
    $\Delta \leftarrow |Q_p - Q(s_p,a_p)|$
    $Q(s_p,a_p) \leftarrow Q_p$
    if $Q(s_p,a_p) = \max_{a'_p} Q(s_p,a'_p)$ and $\Delta > \theta$ (a tiny threshold) **then**
      for all $(s''_p,a''_p) \in$ predecessors(s_p) **do**
        $P \leftarrow p(s_p|s''_p,a''_p)\Delta$
        if $P > \theta$ and (s''_p,a''_p) not on priority queue, or P exceeds current priority of (s''_p,a''_p)) **then**
          promote (s''_p,a''_p) to new priority P
  Observe state s' and reward r resulting from action a
  Update model, $p(s'|s,a)$ and $r(s,a,s')$, based on s',r

Algorithm 1: Pseudocode of parallel anytime Prioritized Sweeping, as it used in this paper.

PS modifies the way of doing value iteration updates, compared to standard value iteration. Rather than doing full sweeps through the entire state-action set, PS focuses computational effort where it can do the most good. That is, it gives priority to those state-action pairs whose Q-values are most likely to have the largest changes. This is implemented by maintaining a *priority queue*, and placing state-action pairs on the priority queue depending on the change in Q-values of their successors. After all, if their successors have large changes in value, these state-action pairs will likely have large changes in values as well (see eq. 1). Value iteration updates are done on state-action pairs in the order indicated by the priority queue. The

priority queue is itself continuously updated after each value iteration update.

In a sense, PS can achieve the best of the worlds of model-based and model-free RL methods. Like model-based methods, it maximally exploits information from real experiences to learn a model and maximally exploits the model by doing full value iteration updates rather than sample-based value updates. At the same time, like model-free methods it focuses its efforts on updating *relevant* state-action pairs' Q-values rather than dispersing the efforts evenly and inefficiently over the entire state-action space.

Standard Dyna and Prioritized Sweeping [2], [5] correspond to *serial* algorithms that first obtain an experience, subsequently allow n iterations of value updates based on the model, then obtain an experience again, etc. In real robots, the execution of actions and the reading out of sensors take a significant amount of time, during which the RL control computer's CPU is usually idle for much of the time. We adapt Dyna/PS to make it a *parallel anytime* algorithm: all idle computer time is used for iterations of value iteration according to PS, until the next robot action must be selected or the priority queue is empty. When there is no sufficient computation time for convergence to the value function that is optimal given the current model, this parallel anytime PS algorithm focuses computation on the most important state-action pairs, and it always yields a viable value function. Algorithm 1 provides pseudocode of parallel anytime PS, as it used in this paper.

C. Directed exploration

Exploration is an inherent aspect of any RL method that does not assume an a priori model. Undirected exploration, which explores evenly around the currently policy, is often less efficient than directed exploration, which attempts to direct exploration towards "interesting" parts of the state-action space. Directed exploration is especially beneficial in the case of Dyna and PS [8], [2], [5]: with standard, straightforward, undirected exploration, the system may converge prematurely to suboptimal policies based on an imperfect, incomplete model learned from limited early experience.

We use a combination of and variation on two methods [2], [8], resulting in a directed exploration technique that is particularly effective for robot RL. It requires the storage of some extra information for each state-action pair. Using the terminology of [8], an *exploration bonus* is added to the estimated Q-values, which reflects the added value of selecting a particular state-action value for the sake of exploration, and action selection is done based on these Q-values plus exploration bonuses:

$$Q^+(s,a) = Q(s,a) + \epsilon\sqrt{m(s,a)}/n(s,a) \tag{3}$$

where ϵ is a constant, $m(s,a)$ is the number of time steps since action a was last tried in state s, and $n(s,a)$ is the number of times action a was tried in state s at all. If $n(s,a) = 0$ it is replaced by a constant $\nu \in [0,1)$. $Q^+(s,a)$ is used for standard Boltzmann exploration, which assigns probabilities of action selection $p(s,a)$ in proportion to values [9]:

$$p(s,a) = \frac{e^{Q^+(s,a)/\tau}}{\sum_{a'} e^{Q^+(s,a')/\tau}} \tag{4}$$

where τ is the so-called temperature parameter.

The net result is exploration that favors actions which are promising with respect to rewards (the effect of Boltzmann exploration), that favors actions that have not been tried often (the effect of $n(s,a)$), and that favors actions that have not been tried for a while (the effect of $m(s,a)$). Once $n(s,a)$ becomes very large, the exploration bonus vanishes.

D. Transformed reward functions

Appropriate reward functions can speed up RL significantly, because they may give "encouragement" for imperfect but approximately correct behavior, and thus direct exploration toward promising parts of the state-action space. However, many authors have reported problems when using such functions. The main problem is that imperfect behavior is rewarded and the system may converge to behavior which obtains a lot of reward but fails to accomplish the desired task. An example is a soccer-playing Robocup robot which was given a small reward for touching the ball (since possession of the ball is important in soccer) and which learned to remain next to the ball and "vibrate", i.e. touching it as often as possible [3].

However, it is possible to design appropriate reward functions and avoid such undesirable and pathological behavior. The first step is to design a straightforward reward function which genuinely reflects the goal of the task. For instance, if the robot is a soccer-playing robot, it may get a reward of 1 if it scores a goal and -1 if the opponent scores a goal.

In the second step we *transform* this original reward function to make it more informative during learning. The key idea is that the optimal policy according to the transformed reward function must be identical to the one according to the original reward function. This is the case if, and only if, the transformed reward $r(s,a,s')$ has the following form [3]:

$$r(s,a,s') = r_{orig}(s,a,s') + F(s,a,s') \tag{5}$$

where $r_{orig}(s,a,s')$ is the original reward and F is a *difference of potentials*:

$$F(s,a,s') = \gamma\Phi(s') - \Phi(s) \tag{6}$$

where Φ is a *potential function* defined over states.

This still leaves room for different potential functions. The design of a particular potential function is problem-dependent. But for any given problem, various potential functions can speed up learning immensely compared to the original reward function, while guaranteeing that the optimal policy remains the same as the one according to the original reward function [3].

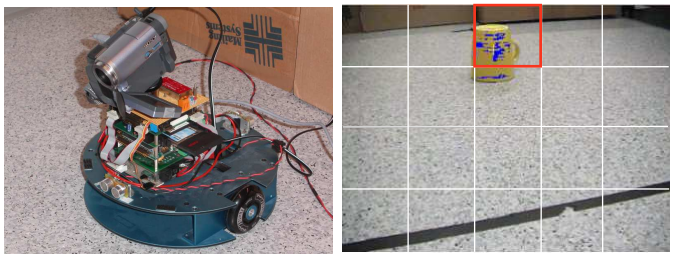

Fig. 1. Fig. a (left). The wheeled mobile robot, equipped with a standard forward looking camcorder camera, and front and back ultrasound sensors. Fig. b (right). Camera image obtained by the robot. The target object, the yellow cup, is in sight. Detected target color pixels are indicated in blue. The small yellow cross marks the center of the cluster of target color pixels. This information is quantized using a regular 5 by 4 grid (white lines). Bold red lines indicate the grid cell corresponding to the quantized state information.

IV. Learning task

A. Task and robot

The techniques designed to accelerate RL described above were investigated by applying them to a real robot's RL task. The task is to find a specific object (a yellow coffee cup in our experiment, see figure 1b), to move towards it, and to bump into it, while avoiding bumping into walls. The task is inspired by the Robocup task of finding a ball, moving towards it, and kicking it. The robot moves around inside a fenced area (see figure 3).

The robot is a small wheeled mobile robot equipped with ultrasound sensors in the front and the back and a forward looking standard Camcorder camera (see figure 1a). It has an onboard low-level motor controller which implements 6 basic actions: go forward (approx. 20 cm), go backward (approx. 20 cm), turn left (approx. 20 degrees), turn right (approx. 20 degrees), turn left and go forward, turn right and go forward. Which of these actions is to be executed during each iteration (taking approx. 0.4 seconds) is determined by the RL system running on an offboard computer. Another offboard computer is used for dedicated processing of visual data coming from the camera.

B. State information

The robot's camera images are processed as follows. A simple color vision algorithm looks for clusters of densely packed pixels of certain color (yellow in this case) in the image. If and only if there is such a cluster, it marks coordinates of the center of this cluster (see figure 1b). This process has some (difficult to quantify) noise, especially with increasing distance to the object of interest. A regularly spaced 5 by 4 grid divides the entire image into 20 regions. All coordinate values marking the presence of the cluster of specific color (the yellow coffee cup) within one grid region are assumed to correspond to one state. Using this standard, BOXES-style state aggregation method [1], the continuous state is quantized into discrete states.

Additional state information is provided by the ultrasound sensors, one of which is mounted in the front, and one of which is mounted in the back. Each sensor provides noisy estimates of distance to the nearest object, if this object is less than approx. 1m away. For each of the ultrasound sensors, this information is quantized into 3 discrete categories: no obstacle (obstacle $>$ .30 m), obstacle near (obstacle $>$ .15 m and $<$ .30 m), and obstacle bump (obstacle $<$ .15 m).[1] Together with the visual information this leads to a total of $3 \times 3 \times 21$ (20 grid cells + not seeing the target object) $= 189$ possible states.

C. Reward function

The ultrasound sensors are used to detect bumping into objects (walls) other than the target object, and this automatically stops forward or backward motion and leads to a negative reward of -1, but how to avoid bumping must be discovered by the learning algorithm. Similarly, the ultrasound sensors and visual information are used to automatically detect bumping into the target object, the yellow cup, which leads to a positive reward of 4 and the end of the episode. The reward discount parameter $\gamma = .95$.

This original reward function, which directly reflects the basic goals of the task, is transformed, as described above, using a potential function. In this task, the potential function Φ is a function which increases as the coordinates of the grid cell in which the detected target object lies, (x, y), are closer to the center of the image ($x = 0$), meaning that the robot is facing the target object more directly, or closer to the bottom of the image ($y = 0$), meaning that the robot is closer to the target object:

$$\Phi(s) = \begin{cases} 6 - |x| - y & \text{if target in view} \\ 0 & \text{otherwise.} \end{cases} \quad (7)$$

V. Experiments

A. Simulation experiment

The state and action representations, reward function, and learning algorithms described above were tested in simulation before applying them to the real robot. The simulations were *not* used to train the controller beforehand and apply an already trained controller to the real robot. Instead, the simulations were used to systematically compare different methods in many runs, which would be impossible in the real robot. We used a simple simulation of the continuous world of the robot in its fenced area, with randomly placed robot starting positions, target positions, and several obstacles (see figure 2a). Actions were as described above. Limited field of view ($90°$) vision and ultrasound sensors were simulated based on the simulated robot's orientation and its distance to the target object, obstacles, and walls.

Using the state and action representations described above, we systemically investigated the utility of the described extensions to the standard RL framework. Starting with standard Q-learning, we consecutively add Dyna, Prioritized Sweeping, directed exploration, and the transformed reward function. We

[1] To avoid damage to the robot and obstacles, we do not want the robot to actually bump into obstacles and the target object, which is why we build in this safety margin of 15 cm.

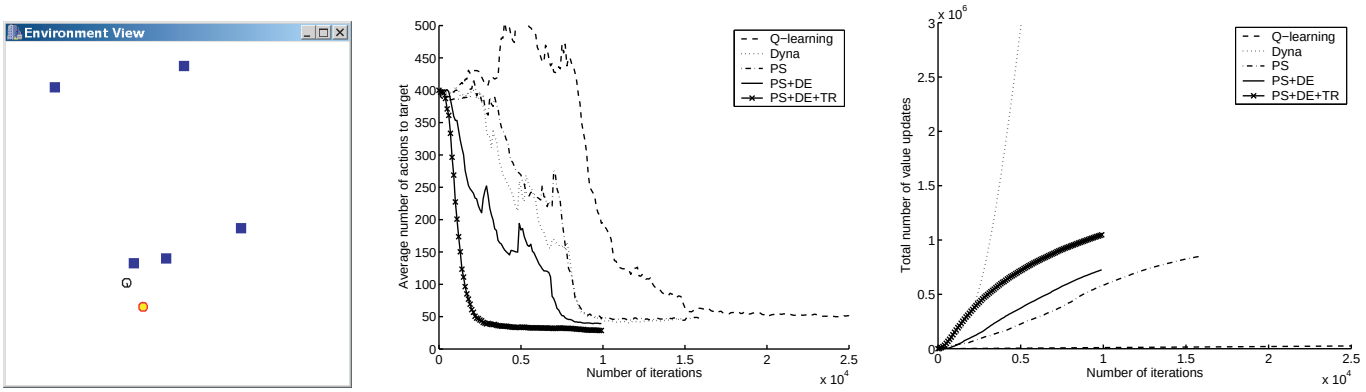

Fig. 2. Fig. a (left). Screenshot of the simulated robot and its environment. The simulated robot is close to and oriented towards the round, yellow target object. Also shown are square obstacles. Fig. b (middle). The average number of actions needed to reach the target, for each investigated method, as a function of state-action-reward-next state iterations. See text for explanation. Fig. c (right). The cumulative number of value updates for each investigated method, as a function of state-action-reward-next state iterations. See text for explanation.

measure performance for each variation in terms of required numbers of state-action-reward-next state iterations and numbers of value updates.

The Q-learning method used the standard update rule of eq. 2. We tested learning rates α in the range $[0.1, 0.8]$, and used the best value for this task, $\alpha = 0.3$. The Dyna method used standard value iteration. The model, i.e. the state transition function and reward function, was learned based on maximum likelihood, i.e. by counting and averaging over experienced state transitions and rewards. After each state-action-reward-next state iteration, a maximum of 2000 value updates were allowed for value iteration on the current model. Our Prioritized Sweeping method was similarly allowed 2000 value updates after each state-action-reward-next state iteration, and priority threshold $\theta = .0001$. Next, Prioritized Sweeping was extended with the directed exploration method described above, using exploration parameters $\epsilon = .02$ and $\nu = .1$. Finally, the transformed reward function was added as described above.

Figure 2b shows learning performance as a function of state-action-reward-next state iterations for each of the methods, averaged over 10 runs. Learning performance is measured as the average number of actions it takes the robot to bump into the target object (the system always learned to avoid bumping into walls/obstacles as well).

The Q-learning method needed the largest number of iterations to reach good performance. The Dyna method needed only about one third of Q-learning's number of iterations to learn the task well. This is in large part due to the fact that in contrast with Q-learning, Dyna can do a lot of *latent learning*, i.e. learning about the characteristics of the environment, regardless of whether any target rewards are obtained. Once Dyna experiences the target reward, all this latent learning allows it to estimate a reasonable policy for the whole state space almost immediately.

The Prioritized Sweeping method (*PS*) did not improve much on the plain Dyna method in terms of number of iterations to learn the task. This is probably because 2000 value updates between each state-action-reward-next state iteration was sufficient for Dyna with standard value iteration. Prioritized Sweeping with directed exploration (*PS+DE*), however, did significantly reduce the number of iterations needed to learn the task. This is the case because the directed exploration method allows the system to more systematically explore the state-action space, such that the target object is encountered sooner. Adding the transformed reward function, finally, led to by far the lowest number of iterations needed to learn the task (*PS+DE+TR*). Now just seeing the target object leads to rewards, such that value function learning can start earlier and the robot explores the state-action space even more efficiently, guided by the task's requirements.

Figure 2c shows total, cumulative numbers of value updates for each of the methods, again as a function of state-action-reward-next state iterations. Standard Q-learning makes as many value updates as state-action-reward-next state iterations. The Dyna method, in contrast, made many millions of value updates. With larger state-action spaces or less time between state-action-reward-next state iterations, that would make this Dyna method with standard value iteration impractical. All Prioritized Sweeping variations required significantly fewer value updates, so they scale much better with larger problems and stringent real-time requirements. The versions with directed exploration and with transformed rewards made slightly more value updates, because they encounter more rewards or encounter rewards earlier.

B. Real robot experiment

Based on the results of the simulations, the complete RL system with Prioritized Sweeping, directed exploration, and transformed reward function was implemented in the real robot. The value function/policy learned in simulation were not used, the real robot learned from scratch. As described above, parallel anytime PS was used, using all idle time of the RL computer while executing actions and sensing. The

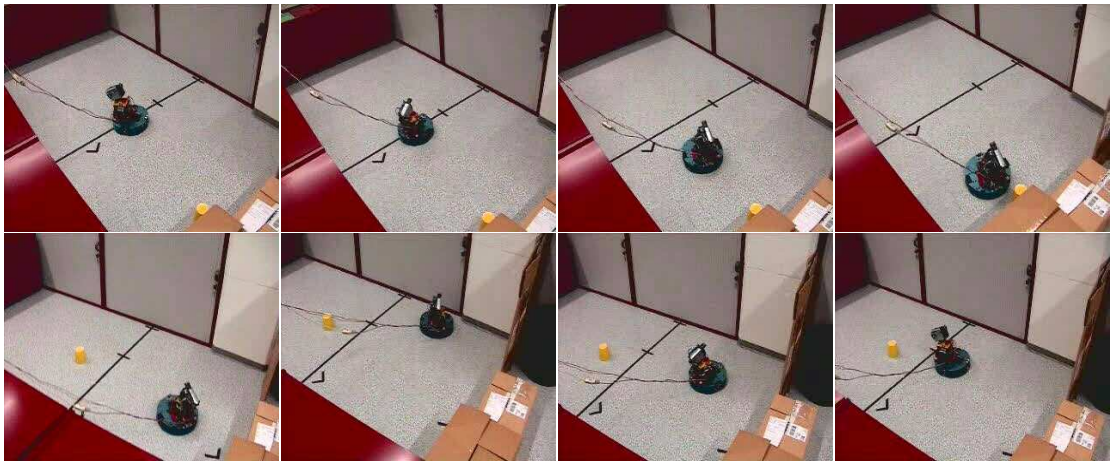

Fig. 3. Snapshots (ordered like text) of robot moving around inside the fenced area containing the target object, a yellow cup. Top row: after approximately 20 minutes of learning, the robot searches for the target object effectively, and approaches it directly. Bottom row: the robot generalizes successfully to novel starting positions and target object positions.

priority threshold $\theta = .0001$. The directed exploration method used exploration parameters $\epsilon = .02$ and $\nu = .1$.

This combination of techniques allowed the robot to learn the task in around 20 minutes of learning. In this period, there was time for approximately 4000 real state-action-reward-next state experiences. But sufficient learning of the task required around a million of value updates in the background using our anytime PS algorithm.

Figure 3 shows a number of snapshots of the robot during its task execution.[2] The robot learns to, first of all, successfully avoid bumping into walls (or obstacles placed in the robot's way), excluding occasional bumping due to exploration. The robot's strategy for finding the target object is to usually simply turn around in one direction until it sees the target object, and to occasionally move forward or backward (note that the robot continues to use its exploration algorithm). This may be likened to "searching" behavior. Once the robot sees the target object, it approaches the target directly, making small corrections to keep the target in the center of its field of view, until its sensors indicate it has bumped into the target.

Importantly, experience with only a limited number of different robot starting positions and target object positions allowed the robot to generalize its behavior successfully to other robot starting positions and target object positions.

VI. Discussion

The combination of techniques investigated in this study allowed a real robot to do quasi-online reinforcement learning from scratch. From an outside perspective the robot learned a nontrivial task online and in real time. The robot learned very efficiently, both in term of required environment interactions and in terms of computation, compared to standard RL techniques.

[2] `http://www.science.uva.nl/~bram/RobotCup.htm` has videos of the robot's behavior, both during the initial stages of learning and after sufficient learning.

Possible further improvements to the methods explored in this paper include more sophisticated generalization techniques [7], [6], [4] and techniques for easily adding more background knowledge [7]. In general, successful applications of RL to robots will, as in this paper, most likely require a combination of techniques that work well together.

Acknowledgments

The work described in this paper was done while the first author was at IDSIA, and it was supported by the Swiss Center for Electronics and Microtechnology (CSEM) and EU project FP6-511 931.

References

[1] D. Michie and R. A. Chambers. BOXES: An experiment in adaptive control. In Dale E and Michie D., editors, *Machine Intelligence 2*, pages 137–152, Edinburgh, 1968. Oliver and Boyd.

[2] A. Moore and C. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning*, 13:103–130, 1993.

[3] A. Y. Ng, D. Harada, and S. Russell. Policy invariance under reward transformations: theory and application to reward shaping. In *Proc. 16th International Conf. on Machine Learning*, pages 278–287, 1999.

[4] A. Y. Ng, H. J. Kim, M. Jordan, and S. Sastry. Autonomous helicopter flight via reinforcement learning. In *Advances in Neural Information Processing Systems 16*, 2004.

[5] J. Peng and R. J. Williams. Efficient learning and planning within the dyna framework. *Adaptive Behavior*, 1 (4):437–454, 1993.

[6] J. Schmidhuber. Curious model-building control systems. In *Proceedings of the International Joint Conference on Neural Networks, Singapore*, volume 2, pages 1458–1463. IEEE press, 1991.

[7] W. D. Smart and L. P. Kaelbling. Practical reinforcement learning in continuous spaces. In *Proc. 17th International Conf. on Machine Learning*, pages 903–910. Morgan Kaufmann, San Francisco, CA, 2000.

[8] R. S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proc. 7th ICML*, pages 216–224, 1990.

[9] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction.* MIT Press, Cambridge, MA, 1998.

[10] C. J. C. H. Watkins. *Learning from delayed rewards.* PhD thesis, Cambridge University, 1989.