Auton Robot (2009) 27: 55–73
DOI 10.1007/s10514-009-9120-4

Reinforcement learning for robot soccer

Martin Riedmiller · Thomas Gabel · Roland Hafner · Sascha Lange

Received: 7 November 2008 / Accepted: 4 May 2009 / Published online: 15 May 2009

Abstract Batch reinforcement learning methods provide a powerful framework for learning efficiently and effectively in autonomous robots. The paper reviews some recent work of the authors aiming at the successful application of reinforcement learning in a challenging and complex domain. It discusses several variants of the general batch learning framework, particularly tailored to the use of multilayer perceptrons to approximate value functions over continuous state spaces. The batch learning framework is successfully used to learn crucial skills in our soccer-playing robots participating in the RoboCup competitions. This is demonstrated on three different case studies.

Keywords Learning mobile robots · Autonomous learning robots · Neural control · RoboCup · Batch reinforcement learning

1 Introduction

Reinforcement learning (RL) describes a learning scenario, where an agent tries to improve its behavior by taking actions in its environment and receiving reward for performing well or receiving punishment if it fails. Since RL enables an agent to learn autonomously from its own experience, it is a highly attractive framework for learning behaviors of autonomous robots.

A current line of research is to establish efficient RL algorithms that are able to solve real-world problems. In this paper, we focus on the application of learning methods within the RoboCup robotic soccer domain. RoboCup was founded in 1997 to establish a challenging testbed for autonomous intelligent systems (Kitano 1997). In various leagues, where each one is focusing on different aspects of the complex overall problem, teams of robots are competing in a soccer-like setting. In 1998, our research group established the Brainstormers project, aiming at building highly competitive robot soccer teams that make extensive use of machine learning, in particular reinforcement learning methods. Since then, many of the crucial skills and behaviors of our agents have been learned from scratch and been applied in our competition teams, both in simulation league and in MidSize league.

The following reviews some of our recent work on the (neural) batch RL framework that constitutes the core concept of learning in our agents. Our concept is based on the idea of learning value functions by the application of dynamic programming methods. Due to the typical continuous nature of state spaces, multilayer perceptrons are used as function approximators that represent the value functions. We found them particularly powerful with respect to generalization and robust in their learning behavior, in particular when used in the batch RL framework.

The purpose of this paper is not to claim that batch reinforcement learning methods are necessarily superior to any other existing reinforcement learning algorithms. This would require a careful empirical investigation of various alternative learning approaches, which is far beyond the scope of this paper. Rather, the purpose of this paper is to give evidence to the thesis that batch reinforcement learning methods are a valuable piece in the growing toolbox of practically useful reinforcement learning algorithms and to show how the basic framework can be adapted to varying requirements in different application scenarios.

M. Riedmiller (✉) · T. Gabel · R. Hafner · S. Lange
Department of Computer Science, Albert-Ludwigs-Universität Freiburg, Freiburg, Germany
e-mail: martin.riedmiller@informatik.uni-freiburg.de

Fig. 1 Scene from a game in the RoboCup middle size league

The article is organized as follows: first, we will give a description of the batch RL framework and in particular describe three variants that have been proven useful for the application to challenging RL problems. Further, we will shortly discuss the concrete setup of neural batch RL controllers and the embedding within our software control architecture. Then we will discuss the concrete application of the concept exemplified in three case studies. Case Study I demonstrates the application for learning a complex and highly effective behavior for the simulation league from scratch. Case Study II shows the application for learning a low-level motor control directly on a real DC motor, and finally Case Study III discusses the learning of an effective dribbling behavior directly on our MidSize robot. At the end, we give an overview of the skills that we have learned so far, discuss their application within our competition team and then we present our conclusion.

2 Related work

Recently, reinforcement learning has drawn a lot of attention for becoming more relevant to solving real-world problems. The introduction of new fast policy search and policy gradient methods (Sutton et al. 2000; Peters and Schaal 2008b) has led to the development of several impressive real-world demonstrators including autonomously flying artistic maneuvers with a helicopter (Bagnell and Schneider 2001; Ng et al. 2004), teaching a robot to hit a baseball (Peters and Schaal 2006), operational space control (Peters and Schaal 2008a) and learning ball-in-a-cup with a robot arm (Kober et al. 2008).

In this paper, we advocate the value-function-based approach to reinforcement learning. Despite some promising early successes (Tesauro and Sejnowski 1989; Crites and Barto 1995), the rather small number of published applications to real-world problems has often led to skepticism regarding the practical relevance of these methods. Recently proposed *fitted* value iteration algorithms (Gordon et al. 1995), like Fitted Q-iteration (Ernst et al. 2006), Neural Fitted Q (NFQ) (Riedmiller 2005) and Least Squares Policy Iteration (LSPI) (Lagoudakis and Parr 2003), make more efficient use of the gathered experience by adapting the experience replay technique (Lin 1992). By drastically decreasing the number of necessary interactions with the system, these kinds of algorithms allow for solving more complex problems than have been feasible so far using only standard value-function-based methods. Some positive results of applying these methods to real-world problems already have been reported, for example naming steering an automobile (Riedmiller et al. 2007), controlling power systems (Wehenkel et al. 2005) and solving job-shop scheduling tasks (Gabel and Riedmiller 2007). In this paper, we present additional examples in the RoboCup domain.

The robot soccer competitions organized by the RoboCup Federation form an interesting and very challenging domain for the application of machine learning algorithms to real-world problems. Research groups have applied a variety of different machine learning methods to many aspects of autonomously soccer playing multirobot systems. Examples include evolutionary algorithms for gait optimization (Chernova and Veloso 2004; Röfer et al. 2004) or optimization of team tactics (Nakashima et al. 2005), unsupervised and supervised learning in computer vision tasks (Kaufmann et al. 2004; Li et al. 2003; Treptow and Zell 2004) and lower level control tasks (Oubbati et al. 2005). RL methods have been used to learn cooperative behaviors in the simulation league (Ma et al. 2008) as well as for real robots (Asada et al. 1999) and to learn walking patterns on humanoid robots (Ogino et al. 2004). Furthermore, Stone's keep-away-game is a popular standardized reinforcement learning problem derived from the simulation league (Stone et al. 2005).

A lot of learning tasks inspired by or directly derived from RoboCup have been used in proof-of-concepts of reinforcement learning methods (Asada et al. 1999; Stone et al. 2005). The goal of our efforts in RoboCup always has been to go a step further, learning skills that can be employed during the competition itself, ideally outclassing any other available hand-crafted or learned behavior.

3 Reinforcement learning for soccer robots

Reinforcement learning (Sutton and Barto 1998) follows the idea that an autonomously acting agent obtains its behavior policy through repeated interaction with its environment on a trial-and-error basis. In the following, we will delineate how and why this learning methodology can be profitably employed in the context of learning soccer robots.

3.1 Reinforcement learning background

In each time step an RL agent observes the environmental state and makes a decision for a specific action, which, on the one hand, may incur some immediate costs (also called reinforcement) generated by the agent's environment and, on the other hand, transfers the agent into some successor state.

Basic reinforcement learning problems are usually formalized as Markov Decision Processes (Puterman 2005). A Markov Decision Process (MDP) is a 4-tuple $M = [S, A, p, c]$ where S denotes the set of environmental states and A the set of actions the agent can perform. Function $c : S \times A \times S \to \mathbb{R}$ denotes immediate costs $c(s, a, s')$ that arise when taking action $a \in A$ in state $s \in S$ and transitioning to $s' \in S$. The probability $p_{ss'}(a) = p(s, a, s')$ of ending up in state s' when performing action a in state s is specified by the probability distribution $p : S \times A \times S \to [0, 1]$.

The agent's goal is not to minimize its immediate, but rather its long-term, expected costs. To do so, it must learn a decision policy that is used to determine the best action for a given state. Such a policy is a function $\pi : S \to A$ that maps the current state the agent finds itself in to an action from a set of viable actions.

3.1.1 Value iteration

When interacting with the MDP, an RL agent passes through a sequence of states $s(t)$ that are coupled to one another by the transition probabilities $p_{s_t,s_{t+1}}(a_t)$ and the actions $a_t = \pi(s_t)$ the agent takes, and the agent experiences a sequence of immediate costs $c(s_t, a_t, s_{t+1})$ incurred. The goal of the reinforcement learning agent is to minimize the expected value of the discounted sum

$$c_t = \sum_{k=0}^{\infty} \gamma^k c(s_{t+k}, a_{t+k}, s_{t+k+1})$$

of costs incurred over time, where $\gamma \in [0, 1]$ is a factor that determines to which amount future costs are discounted compared to immediate ones.

When conditioned on some specific state $s \in S$, the expected value $\mathbb{E}[c_t|\pi, s]$ is called the cost-to-go $J^\pi(s)$ of state s under policy π and it is recursively defined as

$$J^\pi(s) = \sum_{s' \in S} p_{ss'}(\pi(s))(c(s, \pi(s), s') + \gamma J^\pi(s')). \tag{1}$$

Accordingly, function J^π is called the cost-to-go function or value function for policy π.

The goal of learning is to find an optimal policy $\pi^\star$ that has less accumulated costs than all other policies π. It has been shown (Bertsekas and Tsitsiklis 1996) that for each MDP there exists an optimal policy $\pi^\star$ such that for any policy π, it holds that $J^{\pi^\star}(s) \leq J^\pi(s)$ for all states $s \in S$.

Given the optimal cost-to-go function $J^\star$, it is known that an optimal policy is given by greedily exploiting $J^\star$ according to

$$\pi^\star(s) = \arg\min_{a \in A} \left\{ \sum_{s' \in S} p_{ss'}(a)(c(s, a, s') + \gamma J^\star(s')) \right\}. \tag{2}$$

So, the crucial question is how to obtain the optimal cost-to-go function. To perform this task, dynamic programming methods may be employed, such as value iteration (Bellman 1957) which converges under certain assumptions to the optimal cost-to-go function $J^\star$. Value iteration is based on successive updates to the cost-to-go function for all states $s \in S$ according to

$$J_{k+1}(s) = \min_{a \in A} \left\{ \sum_{s' \in S} p_{ss'}(a)(c(s, a, s') + \gamma J_k(s')) \right\}, \tag{3}$$

where index k denotes the sequence of approximated versions of J, until convergence to $J^\star$ is reached.

3.1.2 Q Learning

Similar to Eq. 1, the expected cost-to-go $Q^\pi(s, a)$ of a state-action pair is defined, which is meant to express the expected costs arising after having taken action a in state s and following policy π subsequently:

$$Q^\pi(s, a) = \sum_{s' \in S} p_{ss'}(a)(c(s, a, s') + \gamma J^\pi(s')).$$

Using this relation and knowing that Bellman's equation can be interpreted as $J^\star(s) = \min_{a \in A(s)} Q^\star(s, a)$, the value iteration algorithm from above can be written as

$$Q_{k+1}(s, a) = \sum_{s' \in S} p_{ss'}(a) \times \left(c(s, a, s') + \gamma \min_{b \in A(s')} Q_k(s', b) \right). \tag{4}$$

If there is no explicit transition model p of the environment and of the cost structure c available, Q learning is one of the reinforcement learning methods of choice to learn a state-action cost function for the problem at hand (Watkins and Dayan 1992). In its simplest version, it directly updates the estimates for the costs-to-go of state-action pairs according to

$$Q(s, a) := (1 - \alpha) Q(s, a) + \alpha \left(c(s, a, s') + \gamma \min_{b \in A(s')} Q(s', b) \right)$$

where the successor state s' and the immediate costs $c(s, a, s')$ are generated by simulation or by interaction with a real process. For the case of finite state and action

spaces where the Q function can be represented using a look-up table, there are convergence guarantees that say that Q learning converges to the optimal cost-to-go function $Q^\star$, assumed that all state-action pairs are visited infinitely often and that the learning rate α diminishes appropriately. Given convergence to $Q^\star$, the optimal policy $\pi^\star$ can be induced by greedy exploitation of Q according to $\pi^\star(s) = \arg\min_{a \in A(s)} Q^\star(s, a)$.

3.1.3 Policy iteration

Policy iteration aims at finding the optimal policy by repeatedly evaluating the current policy and, after that, improving it. The former corresponds to the determination of the cost-to-go function J^π according to Eq. 1, which may be done in an interactive manner or, for larger-sized problems, based on simulation. Policy improvement refers to the determination of a new, improved policy π' that is greedy with respect to the current cost-to-go function J^π and is done in accordance to Eq. 2,

$$\pi'(s) = \arg\min_{a \in A} \left\{ \sum_{s' \in S} p_{ss'}(\pi(s))(c(s, \pi(s), s') + \gamma J^\pi(s')) \right\}. \tag{5}$$

The policy improvement theorem (Bertsekas and Tsitsiklis 1996) ensures that $J^{\pi'}(s) \leq J^\pi(s)$ for all states. If this process is iterated, the agent's policy is successively improved. Note that the value iteration algorithm described above can be interpreted as a version of policy iteration where the policy evaluation step is truncated and involves only one backup for all states.

3.2 Batch-mode reinforcement learning

For reasonably small state spaces, the above iteration methods (Eqs. 3, 4, 5) can be implemented using a table-based representation of the value functions. However, interesting RL problems typically have large and often continuous state spaces, where table-based methods are not applicable any more. One way to deal with that problem is to use function approximation to approximate the value function. Multilayer perceptrons (MLPs) are known to be a very useful and robust regression method to approximate value functions in a broad range of different applications (Riedmiller et al. 2007; Hafner and Riedmiller 2007). However, some peculiarities have to be considered in order to make them work properly in practical applications. One important characteristic results from the fact that they approximate the function in a global way, which means that—in contrast to local approximation schemes (like e.g. lookup tables or RBF networks)—changing the value at one point might well have impacts on the outcome of arbitrary other points far away in the input space. Therefore, we consider it as a crucial point for the successful application of multilayer perceptrons for RL, that they are used in a batch-mode type of method, where always a whole set of points is updated simultaneously.

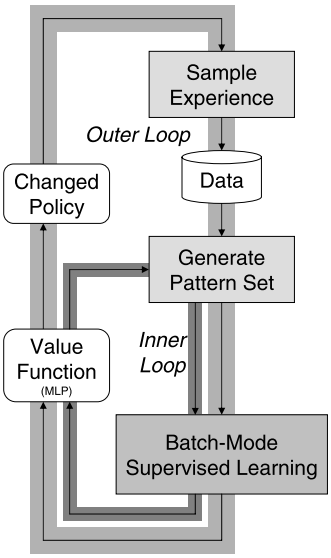

Fig. 2 A graphical sketch of the batch RL framework. It consists of three modules (sample experience, generate pattern set, apply batch-mode supervised learning) that are called sequentially in repeated loops. The core idea is to produce a training pattern set according to Dynamic Programming methods. Depending on the concrete realization, the implementation of the respective modules is adapted (see text for explanation)

Besides the argument of a simultaneous update of all samples, the batch learning scenario for MLPs additionally has the advantage that advanced learning algorithms exist that are typically much more powerful than the commonly used gradient descent technique. The method we use is the Rprop algorithm (Riedmiller et al. 1993), which aside from being very fast with respect to learning time, has also proven to find very good solutions and to be very robust with respect to its parameter settings.

Figure 2 shows a general framework for doing batch RL. It consists of three main steps, namely sampling experience, generating a training pattern set by the use of dynamic programming methods, and finally doing batch supervised learning to approximate the function represented by the training patterns. It is important to note that the individual components might look very different for each particular realization of a batch RL algorithm. We will describe the components in more detail for three batch RL algorithms in the following. The only component that is the same in all variants is the batch supervised training module.

We found batch RL algorithms in conjunction with multilayer perceptrons in general to be very effective with respect to the amount of training experience needed. All of our skills learned in the simulation league and middle size league are

based on one of the here described variants of the batch RL scheme (for an overview, see Sect. 7).

3.2.1 Neural fitted value iteration

The core idea of the fitted value iteration scheme is to iteratively compute new approximations of the value function, $\tilde{J}_{k+1}$, for a given set of points in the input space (Gordon et al. 1995). Therefore, the set of experiences just consists of different states. The states can, in principle, be determined completely without interaction with the system, e.g. lying on a regular grid or being drawn randomly. Alternatively, they might be recorded along actual trajectories, e.g. to focus on interesting parts of the state space. To generate the training pattern set, the according module here requires the model $p_{ss'}(a)$ of the system and the current estimate of the value function, $\tilde{J}_k$. Input patterns consist of the states of the experience set, while the respective target for each state is computed using the value function operator applied to the current value function estimate $\tilde{J}_k$. Thus, for all $s \in I$:

$$J_{k+1}^{target}(s) := \min_a \left\{ \sum_{s'} p_{ss'}(a)(c(s,a,s') + \gamma\, \tilde{J}_k(s')) \right\}.$$

Hence, the training data set is given by

$$\mathcal{P}_{k+1} := \{(s, J_{k+1}^{target}(s)) | s \in I\}.$$

This training set is then taken by the batch supervised learning module, which generates a new estimate for the value function.

The inner loop is repeated (index k counts inner loop iterations) until a satisfying policy has finally been learned. The outer loop is not necessarily required; however it can be used optionally, e.g. to sample states using a policy greedily exploiting the current value function. In many cases, this has turned out to be useful in order to focus on states in interesting regions of the state space.

3.2.2 Neural fitted Q-iteration (NFQ)

In the model-free case, the situation is quite similar to the fitted value iteration scheme described above. There are three major changes that characterize the Neural Fitted Q-iteration (NFQ) scheme (Riedmiller 2005). First, the set of experiences I are now triples of the type (state, action, successor state). Second, they are sampled by interaction with the system, and finally the value function uses the action as an additional argument. The target values are then computed by the application of the Q-learning operator to all $(s, a, s') \in I$[1]:

$$Q_{k+1}^{target}(s,a) := c(s,a,s') + \gamma \min_{a \in A(s)} \tilde{Q}_k(s', a).$$

The training data set is given by

$$\mathcal{P}_{k+1} := \{((s,a), Q_{k+1}^{target}(s,a)) | (s,a,\cdot) \in I\}.$$

As before, the training pattern set is now used by the batch supervised learning module to generate a new Q function. Also as before, the inner loop is repeated, until a satisfying policy has been learned, and the outer loop can be used optionally to generate experience triples in interesting regions of state-action space.

As a crucial point, please note that the validity of the experience set does not depend on the policy with which it is sampled. Therefore, the set can simply be aggregated over multiple passes of the outer loop. No sampled data has to be invalidated and disregarded. This fact makes the fitted methods particularly data-efficient.

3.2.3 Neural fitted policy iteration

In contrast to the above schemes, fitted policy iteration assumes the existence of a policy π to sample experiences. The general idea is to learn a value function of the policy, and then to improve the policy by greedily exploiting the value function—corresponding to one pass through the outer loop in Fig. 2.

We start with the description of a model-based version that works by sampling complete trajectories. This version is also used in Case Study I. Variants of this scheme, e.g. for the model-free case, will be discussed at the end of this section.

Sampling experience is done by repeatedly doing trajectories, in general from different starting points, following the current policy π_k (here, index k counts outer loop iterations).

Let I be the set of all states that occur within all of the collected trajectories. Then, for all states within I, the resulting accumulated path costs are computed by following the recorded trajectory and adding up the costs. This is called Monte Carlo policy evaluation in (Bertsekas and Tsitsiklis 1996),

$$J_k^{target}(s) := E\left\{ \sum_t c(s_t, \pi_k(s_t), s_{t+1}) | s_0 = s \right\}, \tag{6}$$

where the number of time steps t varies with the differing lengths of the trajectories. Accordingly, we obtain a pattern set

$$\mathcal{P}_k := \{(s, J_k^{target}(s)) | s \in I\}$$

[1] We usually adopt a definition, where the immediate cost c is not stored as part of the experience set I, thus is not an attribute of the environment but more related to the definition of the particular control-task and therefore is calculated within the agent. Actually, the environment very seldom delivers a cost signal that is distinct from the state signal in practice. Nevertheless, this is more a question of definitions and not a strict dogma; an attribution of the immediate costs to the environment may be absolutely reasonable and in some cases even necessary. All algorithms presented here are applicable in both cases.

that is used as input to the batch supervised learning process which produces as output an approximated version $\tilde{J}_k^\pi$ of the true cost-to-go function of the current policy π_k. Since the training target values, as computed by Eq. 6, do not depend on the estimate of the value function, the inner loop is meaningless in this case.

To prepare the next pass through the outer loop, a new policy has to be derived. In the model-based case, the next policy π_{k+1}, is computed by greedily exploiting $\tilde{J}_k^\pi$ according to

$$\pi_{k+1}(s) := \arg\min_{a \in A(s)} \sum_{s' \in S} p_{ss'}(a)(c(s,a,s') + \gamma \tilde{J}_k^\pi(s')). \quad (7)$$

Having determined the new policy, a new pass through the outer loop can be started by sampling experience according to π_{k+1}. A significant difference to the fitted value iteration schemes above is that the experience must be collected every time from scratch, and it cannot be added to the experience already sampled. This is due to the fact that the experience collected is only useful for evaluating the corresponding policy. This means that this method is, in principle, less data-efficient than the fitted value and Q-iteration methods described above. However, in many cases very good policies can already be found within only a few iterations of the outer loop. This method has turned out to be particularly useful in cases where sampling many experiences is not a problem, e.g. because a fast simulator of the environment is available. We will further explore this point in Case Study I (Sect. 4).

The fitted policy iteration method can be modified in various ways. A model-free variant of this algorithm can be realized by following the idea of rollout policies (Tesauro and Galpering 1995; Bertsekas and Tsitsiklis 1996). In another variant, one may use iterative backups of the value functions instead of Monte Carlo evaluations to approximate $\tilde{J}_k^\pi$ or $\tilde{Q}_k^\pi$ respectively (Bertsekas and Tsitsiklis 1996). Doing so then makes multiple passes through the inner loop reasonable again. This variant has the advantage, that experiences do not have to be provided in complete trajectories, but one can in principle deal with partial trajectories or even with single transitions.

3.3 Setting up the neural RL controller

Here, we provide an overview on some issues that are characteristic for our work on neural RL controllers in the robotic domain.

The learning task is formulated either as a terminal-goal-state problem or as a setpoint-regulation problem. In the first case, we assume that a successful controller can control the system to a goal state, and then the control task is terminated. Accordingly the agent then receives terminal costs of zero (see Case Studies I and III). In the setpoint-regulation problem, no terminal goal state exists, but the agent has to actively keep the system within the set of goal states. This is specified by giving immediate zero costs for every transition, where the state is within the target region (Case Study II). In contrast to the terminal-state framework, the episode here is not terminated and no terminal costs are given. In general, we consider the setpoint-regulation problem to be more difficult to learn, since the agent not only has to reach a certain target, but also has to take into account that the state can be held there in the future. Also, since no state is explicitly evaluated with terminal costs of zero, some precautions have to be taken to avoid that the output of the neural network always tends to increase. We tackle that problem by introducing artificial training patterns, where we know the value function is equal to 0 ('*hint-to-goal*' heuristic, see Riedmiller 2005).

As optimization criterion, we typically use a minimum time formulation, since it results in a very simple and task-independent choice of the immediate cost function, i.e.

$$c(s,a,s') := \begin{cases} 0.0, & \text{if } s \in S^+, \\ c, & \text{else} \end{cases} \quad (8)$$

where S^+ denotes the set of goal states and c is some small positive real number, e.g. $c = 0.01$. In the case of explicit constraints for the states that must not be violated (S^-, failures), the above definition is extended by $c(\cdot,\cdot,s') = 1.0$ if $s' \in S^-$. The reason for choosing the low costs for regular transitions and equal to 1 for transitions into failure states, is that the output function of the multilayer-perceptron is a sigmoid with range $(0, 1)$ (empirically, we found learning with a sigmoidal output to be considerably more robust than when using a linear activation function). Input values to the neural network are always normalized to have zero mean and a standard deviation of 1. Typically, in our experiments we use no discounting, i.e. $\gamma = 1$.

The neural network used is a multilayer-perceptron, with the number of input units corresponding to the number of state variables plus the number of action variables in the Q-learning case. Typically, we use two hidden layers with sigmoidal activation functions and a single output neuron. No shortcuts were used here. Although not shown in this paper, in several studies we found that learning behavior is very robust with respect to the concrete choice of the network structure (see e.g. Riedmiller 2005).

For the purpose of training neural networks, we exclusively make use of the Rprop algorithm (Riedmiller et al. 1993), which is considerably faster than ordinary gradient descent. Moreover, its robustness with respect to the choice of its learning parameters allows us to employ Rprop using its standard settings for a wide range of RL applications and, in particular, in all the case studies described below.

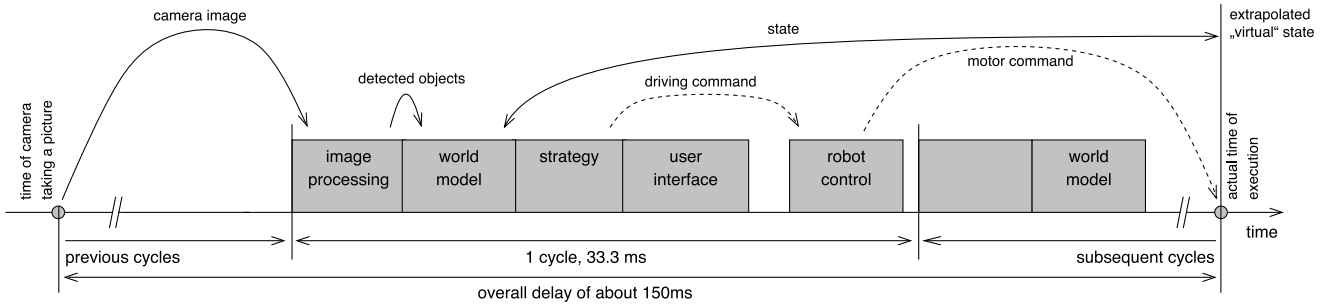

Fig. 3 Sequential processing of the different control modules during one cycle of the control loop (MidSize robot). The control loop is repeated 30 times a second. The overall delay from sensing to the first physical effects of the robot's reaction has been determined empirically to measure about 150 ms. In order to establish a state description resembling the Markov property as close as possible, the world model uses robust regression techniques to predict, from the measurements and previously selected actions, the state of the environment when the reaction has its first effect. In this architecture, the strategy module works on the predicted state information only, allowing the decision-making to neglect any delays

In the applications presented here, discrete action sets are used. If the restriction of the amount of training experience is an issue (as it typically is when learning on a real system, see Case Study II and III), we try to keep a balance between a small action set and the quality of the resulting policy. If collecting experiences is easy, e.g. because a fast simulator is available, the action set can become quite large (e.g. in the range of 50–100 actions or more; see Case Study I) in order to learn a highly optimized policy.

3.4 An architecture for soccer-playing robots

Using reinforcement learning on real robots has several important implications on the overall architecture of the control software. A prerequisite for standard RL methods is the modeling of the learning task as a Markov Decision Process (MDP). Accordingly, we introduced a (closed) control loop with discrete time steps, as proposed in classical digital control theory. During each step of the control loop all modules are processed sequentially (see Fig. 3).

Another issue is providing an internal representation of the environment's state that has the Markov property. In order to provide reliable estimates, the robots' world model deals with noisy measurements (imprecision, vibration) and partial observability (field of view, occlusions, bad lighting conditions) implementing a sensor fusion process specifically tailored to our needs. An overview can be found in (Gabel et al. 2006).

Having only non-stationary sensors available, a precise estimate of the robot's own position on the field (self-localization) is especially important, since all other estimates are directly affected by errors in this position estimate and the derived robot velocity. A newly developed precise and very efficient self-localization method using the Rprop algorithm (Riedmiller et al. 1993) for locally fitting a position estimate to sensory data (Lauer et al. 2005) forms the solid basis of our world model that all other estimators are built on.

To overcome the problem of non-neglectable temporal delays (typically 150 to up to 200 ms, see Fig. 3), we implemented a predictive world model utilizing robust regression techniques and testing of alternate hypotheses. This world model allows for extrapolating the movement of all relevant objects into the future (Lauer et al. 2006). MLPs for predicting parts of the world model have been included as well, following an idea of (Behnke et al. 2003). Thus, decision-making in our architecture never works on the state of the environment as it was sensed by the sensors, but always uses the state extrapolated to the moment in future when a decision actually will become active.

Establishing this virtual state description that approximates the Markov property as close as possible proved a necessary prerequisite for successfully applying value-function-based RL methods to the soccer robots.

The structuring of the decision-making module is another important aspect. We followed a behavior-based approach for implementing this module, adapting it to the specific needs of reinforcement learning. Fundamental skills like dribbling and shooting are implemented in distinct behavior modules. Higher levels of this hierarchical architecture realize more abstract tactical abilities, making use of lower level behaviors. This decomposition allows (a) for learning on all levels of the hierarchy, including individual skills like dribbling, but also including higher level tactical abilities and (b) for easily isolating learnable sub-tasks from the overall strategy. Furthermore, learned modules can be easily combined with other learned or hand-coded modules and integrated into the soccer robot's strategy.

In the following sections, we present three different examples of learning complex real-world problems using the techniques described here.

4 Case study I: learning an aggressive defense behavior

The focus of this case study is laid upon RoboCup's 2D simulation league, where two teams of simulated soccer-playing agents compete against one another using the Soccer Server (Noda et al. 1998), a real-time soccer simulation system.

4.1 The environment

The Soccer Server allows autonomous software agents to play soccer in a client/server-based style. The server simulates the playing field, communication, the environment and its dynamics, while the clients—eleven agents per team—are permitted to send their intended actions (e.g. a parameterized kick or dash command) once per simulation cycle to the server via UDP. Then, the server takes all agents' actions into account, computes the subsequent world state and provides all agents with (partial and noisy) information about their environment via appropriate messages over UDP (cf. Fig. 4). The simulation is based on discrete time steps of 100 ms length.

The special challenge in soccer simulation is its complexity. Rules and simulation constraints are geared to make the simulation resemble real soccer as much as possible. Consequently, a large number of complex tasks arises, when developing simulated soccer agents. These range from handling imperfect and noisy (simulated) vision and the difficulties in creating low-level soccer skills of high technical quality to aspects of team tactics and strategies. Moreover, multiagent coordination, cooperation, and reasoning in a decentralized and partially observable environment represent special challenges. Additionally, the intricacy of developing powerful components for a simulated soccer player is significantly magnified by the fact that the simulation environment has remained stable for a number of years. By now, most teams possess numerous very strong hand-tuned behaviors into whose development and steady improvement months or even years of effort have been invested.

Accordingly, the utilization of learning approaches in this context is very appealing. On the one hand, given the high degree of competitiveness in soccer simulation, human development efforts may sometimes not suffice to outperform certain capabilities some opponent team has implemented. Therefore, the usage of machine learning may be the only viable option to get ahead. On the other hand, given that a simulation is available, the utilization of reinforcement learning algorithms is not complicated, for example, by expensive interaction between robot hardware and the environment. Nevertheless, the state and action space sizes to be dealt with, when learning for autonomous soccer agents, render the problems we are focusing on here as very hard ones.

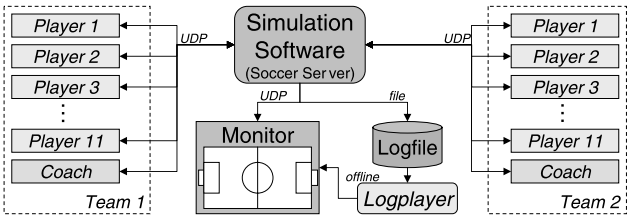

Fig. 4 Soccer simulation environment. The events on the 2D playing field are simulated by a standard software; the course of action during a match can be visualized using an additional monitor program

The case study at hand addresses a defense scenario of crucial importance: We focus on situations where one of our players must interfere and disturb an opponent ball leading player in order to thwart the opponent team's attack at an early stage and, even better, to eventually get possession of the ball and, thus, initiate a counter attack. In so doing, we rely on batch-mode RL for enabling our players to autonomously acquire such an *aggressive defense behavior* (ADB), we carefully test it, and integrate it into our competition team's defensive strategy.

4.2 Task description & learning system setup

The task of the agent is to disturb the opposing agent having the ball, in particular, to prevent it from moving ahead, and, if possible, to steal the ball. A general strategy to achieve these goals is difficult to implement because

- the task itself is far beyond trivial and its degree of difficulty heavily depends on the respective adversary,
- there is a high danger of creating an over-specialized behavior that works well against some teams, but performs poorly against others, and
- duels between players (one without and the other with the ball in his possession) are of high importance to the team as a whole, since they may bring about ball possession, but also bear some risk, if, for example, a defending player loses his duel, is overrun by the dribbling player, and thus opens a scoring opportunity for the opposing team.

Within the RL framework, we model the ADB learning task as a terminal state problem with both terminal goal S^+ and failure states S^-. Intermediate steps are punished by constant costs of $c = 0.05$, whereas $J(s) = 0.0$ for $s \in S^+$ and $J(s) = 1.0$ for $s \in S^-$ by definition (cf. Eq. 8).

The state space is 9-dimensional and covers, in a compressed form, information about positions and velocities of both players involved as well as of the ball. Additionally, some information is incorporated to indicate where on the playing field the current situation is located. The learning agent is allowed to use dash(x) and turn(y) commands where the domains of both commands' parameters ($x \in [-100, 100]$, $y \in [-180°, 180°]$) are discretized such

that in total 76 actions are available to the agent at each time step (60 turn actions and 16 dashes of differing power).

We have to distinguish between different types of goal and failure states.

Successes A dueling episode can be considered successful, i.e. finished by reaching a terminal state $s \in S_1^+$, if the ball has been brought into the learning player's kickable area. Moreover, in soccer simulation there is the so-called tackle command which is meant to simulate a straddle and only succeeds with a certain probability, depending on the relative position of the ball to the player. Thus, we also consider an episode successful, if the learning agent has managed to position itself in such a manner that issuing a tackle command yields a successful tackle for the ball with very high probability.

It may also happen that the opposing agent having the ball simply kicks it away (usually forwards) as soon as the ADB learning agent has approached or hassled him too much, or if it simply considers his situation to be too hopeless to continue dribbling. Consequently, if an opponent executes such a panic kick, the episode under consideration may be regarded as a semi-success, since the learning agent has managed to effectively interfere with the dribbler, though it has not secured the ball (goal state set S_2^+).

Failures The ADB player is said to fail (entering a failure state $s \in S^-$) if the opponent has remained in ball possession, has overrun the learning agent and escaped at least 7 m from him, or approached the goal such that a goal shot might become promising.

For the learning statistics we report on below, we also distinguish episodes without a clear winner that were ended by a time-out (maximal episode duration of 35 time steps). We will refer to such duels as semi-failures, because the learning agent was not effective in interfering with the dribbling player—in a real match the player advancing the ball may have had the chance to play a pass to one of his teammates within that time.

4.3 Special features

In soccer simulation, the transition model p is given, since the way in which the Soccer Server simulates a match is known. We exploit this advantage to the largest degree possible by using model-based instead of model-free learning methods and, thus, simplifying the learning task (e.g. by having to represent a cost-to-go function over S only, instead of over $S \times A$). For the ADB learning task at hand, however, the situation is aggravated due to the presence of an adversary whose next actions cannot be controlled and hardly be predicted. Consequently, the heretofore known model represents merely an approximation $\tilde{p}$ of the true model of the process.

Of course, we might enhance the accuracy of $\tilde{p}$ by incorporating knowledge about the opponent's behavior, such as building an opponent model or assuming an optimally dribbling adversary. As it turns out, however, assuming an idle opponent that does not contribute to state transitions at all is a very robust approach. This no-op assumption is appropriate, because a single opponent action only has a minor influence on the transition of the state and because the cost-to-go function for the problem at hand is rather flat over wide regions. As a consequence, an approximate prediction of the successor state is possible and, therefore, also the realization of a policy improvement step by greedily exploiting the current cost-to-go function according to Eq. 7.

For the calculation of $\hat{J}_k^\pi$, i.e. for the evaluation of the current policy, we rely on Monte Carlo-based estimation methods as described in Sect. 3.2.3. This matches very well with the fact that in soccer simulation an arbitrary number of evaluative episodes based on the current policy can be easily performed, thanks to the availability of a fast simulator.

4.4 Learning procedure

We designed a specialized set of starting states S_0 for the learning agent ($|S_0| = 5000$), which is visualized in Fig. 5a. It basically consists of two semicircles across which the opposing agent having the ball is placed randomly, whereas our learning player resides in the center. While the semicircle that lies in the direction towards our goal (defensive direction) has a radius of 3.0 m, the one in the opposite (offensive) direction is larger (5.0 m). The intention behind this design of starting situations is that, on the one hand, an agent possessing the ball typically starts immediately to dribble towards our goal, whereas the ADB learning agent must interfere and try to hinder him from making progress. On the other hand, the intended aggressive defense behavior shall be primarily applied in situations where our player is closer to our goal or where the opponent only has a small head start. Regarding the remaining state space dimensions, the ball is always randomly placed in the opponent's kickable area with zero velocity, and the velocities of both players as well as their body angles are chosen randomly, as well.

Moreover, we defined four *training regions* on the playing field, as sketched in Fig. 5b. The midfield training region is situated at the center of the field, the central defensive region is halfway towards our goal. Finally, there are a left wing and a right wing defensive region that are placed near the corners of the field with a distance of 25 meters to our goal. The idea behind this definition of different training and testing places is that dribbling players are very likely to behave differently, depending on where they are positioned on the field. As a consequence, a duel for ball possession may proceed very differently, depending on the current position on the field. To this end, the exploitation of symmetries in

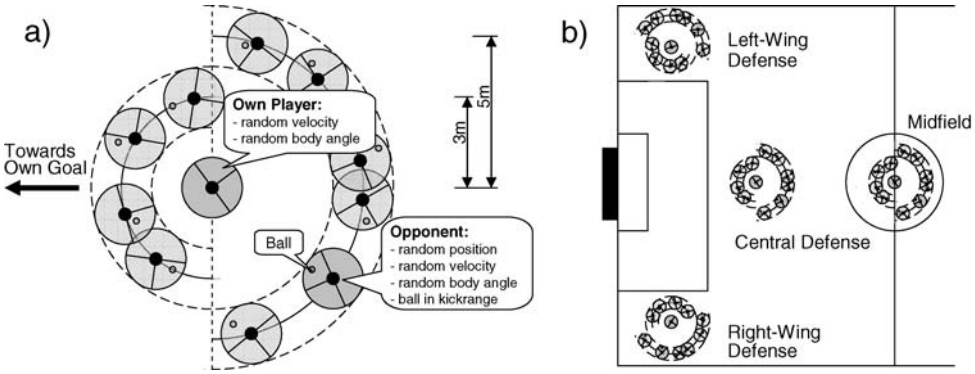

Fig. 5 Customized sets of training start situations and training regions

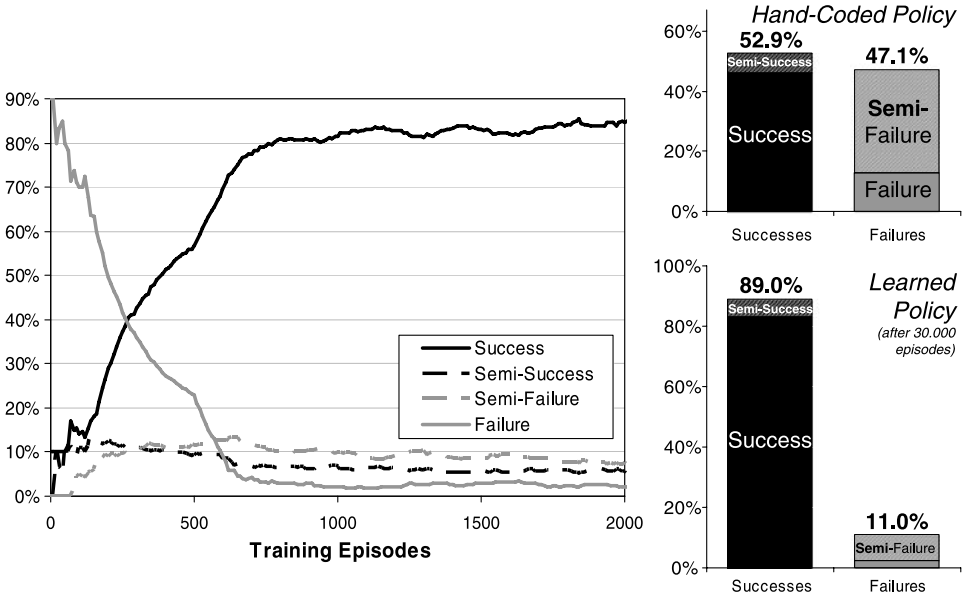

Fig. 6 This figure shows the ADB learning progress (against team WrightEagle). The *right part* opposes the resulting performance of our hand-coded dueling behavior and the learned ADB policy. The share of episodes where our player manages to steal the ball from the opponent is almost doubled, whereas there is only one in 42 episodes where the dribbling opponent still overruns the learning player

the tasks turned out to be problematic, as opponent dribbling players behaved differently in seemingly symmetric situations, e.g. in defensive left and right scenarios.

The learning agent starts with a cost-to-go function J_0, represented by a randomly initialized multilayer perceptron neural network with one hidden layer consisting of 18 units with sigmoidal activation functions (9:18:1-topology). During interaction with the environment, this function is always exploited greedily, i.e. realizing policy π_{k+1}, and simulated experience is collected. New estimates for $\hat{J}^{\pi}_{k+1}(s)$ are calculated according to Eq. 6 for successful episodes, failure states $s \in S^- \cap I$ in the experience set are associated with maximal costs of 1.0, and semi-success as well as semi-failure episodes (which play a negligible role as learning moves on) are disregarded for evaluating π_{k+1}.

Central to the learning process is that we perform neural network training in batch-mode. After having simulated a larger number of training episodes and, in so doing, having built up a set of representative states $I \subset S$, where for each $s \in I$ we have an estimated value $\hat{J}^{\pi}_{k+1}(s)$, the next cost-to-go function is determined by invoking the underlying batch supervised learning process. For neural network training, we employ the back-propagation variant Rprop using default parameters. Performing this step completes one iteration of fitted policy iteration (one pass through the outer loop of Fig. 2).

As can be seen in Fig. 6, the learning process is quite effective. After about 700 training episodes (corresponding to 16 fitted policy iterations and, hence, as many batch-mode neural net trainings), the agent consistently succeeds in capturing the ball with a probability of about 80%. The fact that the learned ADB policy tremendously outperforms our hand-coded behavior can be read from the right part of this figure. While successes and failures used to be in balance when employing the hand-coded policy, after learning, successes now outweigh failures at the rate of 9:1.

Figure 7 visualizes the cost-to-go function acquired after 30000 training episodes for a small, two-dimensional fraction of the 9-dimensional state space. While zero object velocities and constant player body angles are assumed, the plot shows how desirable each position on the pitch would be for the learning agent. Obviously, positions where the

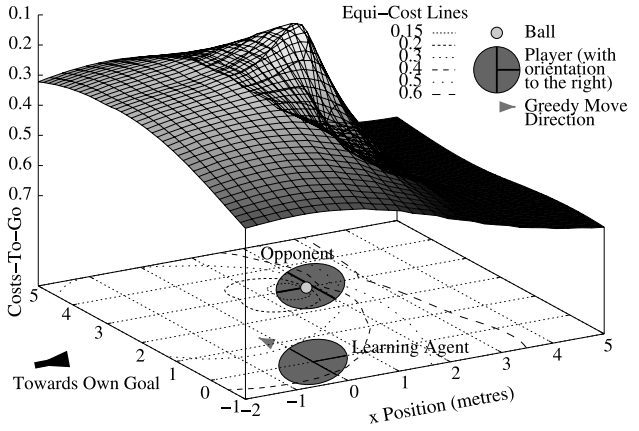

Fig. 7 Learned value function for the dueling behavior, see the text for a detailed explanation

learning player blocks the dribbler's path towards the goal are of high value, whereas high costs are to be expected if the opponent has already overrun our player. From the shape of $\tilde{J}$ and the resulting equi-cost lines, it can be concluded that—from the learning agent's perspective—the most desirable direction into which to move is the one indicated by the gray-colored arrow. Accordingly, this kind of greedily exploiting $\tilde{J}$ (cf. Eq. 2) by following the steepest ascent brings the ADB learner onto a promising interception course (assuming a rational, i.e. forward-moving opponent).

4.5 Final performance

Table 1 summarizes the performance of the learned aggressive defense behavior, when fighting against a selection of adversary teams, subject to various test situations at different places on the playing field. Although we performed a number of series of learning experiments with differing training opponents, we report here on results only that were obtained when training against team WrightEagle which features a rather strong dribbling skill.

For reasons of readability, failures and semi-failures as well as successes and semi-successes are not distinguished. We note, however, that the rate of (full) failure episodes for the learned policy is only about 8.2%, compared to 19.7% by the hand-coded policy. If we compare the success rates (leaving out semi-successes, i.e. only considering succeeding in obtaining the ball) of the learned and the hand-coded policy, we find a significant performance boost. Their share has been incremented from 38.1% to 62.3% (averaged over 5 opponent teams and 4 different test regions on the pitch). What can also be read from Table 1 is that WrightEagle obviously has the most highly developed dribbling behavior against which our hand-coded routine for interfering with

Table 1 This table opposes the performance of our hand-coded dueling behavior, which we used during competitions until 2006, and the new RL-based aggressive defense behavior ADB. The latter significantly outperforms the former for all test opponent teams considered and varying test situation sets

	Hand-Coded				Learned			
	Test Region-Specific Performance							
	DefL	DefR	DefC	Midf	DefL	DefR	DefC	Midf
Success Rate	56%	56%	55%	53%	84%	83%	85%	84%
Failure Rate	44%	44%	45%	47%	16%	17%	15%	16%
Opponent	Opponent-Specific Performance (Success:Failure)							
CZU	57.2% : 42.8%				90.2% : 9.8%			
STEP	61.0% : 39.0%				79.1% : 20.9%			
TokyoTech	61.2% : 38.8%				80.9% : 19.1%			
UvA	43.2% : 56.8%				85.9% : 14.1%			
WrightEagle	41.5% : 58.5%				82.9% : 17.1%			
Joint Average	52.8% : 47.2%				83.8% : 16.2%			

the dribbler performed worst.[2] After learning, however, the acquired ADB policy clearly outperforms any opponent at any place on the pitch.

During the run of a standard game (6000 time steps), our team players start on average 66 dueling episodes. Therefore, even under the very conservative assumption that only about half of all attempts are successful, we can draw the conclusion that the learned behavior allows for stealing the ball at least 30 times per game.

During competition matches, any agent utilizes the acquired policy for dueling with opponents in possession of the ball on average in about 14.8% of the time our team is defending (for obvious reasons, it is not employed when attacking). This is quite a considerable usage share, when taking into account that most of the time a defending player has to pursue different tasks, such as covering opponents or obstructing potential pass lanes. Taking the perspective of the dribbling player, the situation is even clearer. An opponent is being disturbed by one of our players employing the learned aggressive defense behavior during approximately 41.2% of the time he is in ball possession.

The deployment of the learned ADB policy in our competition team, Brainstormers, clearly improved its defense capabilities and had a strong impact on winning the world championships tournaments RoboCup 2007 and 2008. Further details on the learning procedure, system set-up, and experimental results for the learning task examined in this case study can be found in (Gabel et al. 2008).

[2]The performance gain brought about by utilizing ADB was of special subtlety, insofar as we faced team WrightEagle both in the final match of the world championships 2007 and 2008.

5 Case study II: learning motor speed control

5.1 The environment

Fast, accurate and reliable motor speed control is a central requirement in many real world applications, especially for mobile, wheel-based robots. In most applications, this low level motor control behavior is a crucial prerequisite for higher level tasks to be efficient and successful. Especially changing load situations, dependent on the overall system behavior, are challenging for general control schemes and are often dedicated to immense effort of designing an appropriate control law with its structure and parameters.

The RoboCup MidSize league provides a competitive testbed for a broad range of autonomous mobile robot control tasks. While Sect. 6 presents a case study for learning a competitive skill for the MidSize league, the objective of this case study is to learn the low level control of our omnidirectional MidSize robot (see Fig. 13) and is therefore not only specific to RoboCup. To be more precise, the goal is to learn reliable and accurate speed control of the DC motors that operate the robot. On the omnidirectional robot, we have three DC motors in a specific configuration, each driving an omnidirectional wheel. The motion and dynamics of the robot depend directly on the speeds of the three motors. This gives the robot a high maneuverability and avoids nonholonomic constraints in motion planning. On the other hand, the interaction of the three motors causes highly changing load situations which a motor controller has to cope with.

5.2 Task description & learning system setup

Our goal is to learn a fast, accurate and reliable controller for regulating the speed of each DC motor of the omnidirectional mobile robot, solely by interaction with the real robot.

Instead of learning a specialized controller for each motor, we show a setup where we learn a single controller that operates independently on each of the three motors. From the point of view of the controller, it gets the state information and set point of a single motor and answers with an action that will regulate this motor from its current state to the desired speed. In other words, there is only one single DC motor in the view of the controller that has to be controlled in a broad range of dynamical load situations. On the other hand, from the point of view of the robot, there is one controller that can take an action for each of the three motors separately in each time step. This procedure is legitimate, since the three motors and wheels are of the same type of construction. In the following, we will describe the learning system setup in more detail. For doing so, we take the viewpoint of the controller, so the task is DC motor speed regulation, based on the state information of a single motor, to arbitrary set points under a broad range of load situations.

The control problem considered can be described as a Markovian Decision Problem (MDP). As our goal is to learn in interaction with the real motor, we use Neural Fitted Q-iteration to have a fast and robust learning setup.

The input to the RL controller must represent the current state of the DC motor, such that the Markovian property of the task is captured. The state of a general DC motor can be sufficiently described by two variables, namely the current motor speed $\dot{\omega}$ and the armature current I. Since our final controller has to deal with arbitrary target speeds, the information about the desired speed must also be incorporated into the input. In principle, we can do this by directly using the value of the target speed. However, here we are using the error between the actual speed and the target speed, $E := \dot{\omega}_d - \dot{\omega}$.

The immediate cost function $c : S \times U \to \mathbf{R}$ defines the control behavior eventually desired. Here, we are facing a set point regulation task, since no terminal states exist, but instead regulation is an ongoing, active control task. The control objective here is to first bring the motor speed close to the target value as fast as possible and then to actively keep it at the desired level. In terms of the immediate cost function, this can be expressed by the following choice of c:

$$c(s, a, s\prime) = c(s) = \begin{cases} 0 & \text{if } |\dot{\omega}_d - \dot{\omega}| < \delta, \\ 0.01 & \text{else.} \end{cases} \tag{9}$$

The first line denotes the desire to keep the motor velocity $\dot{\omega}$ close to its target value $\dot{\omega}_d$, where the allowed tolerance is denoted by $\delta > 0$.

The second line expresses the desire for the minimization of the time of the system being not close to its target value. The above framework specifies our demand for a time-optimal controller to a region close to the target value, which reflects our desire for a fast and accurate control behavior.

5.3 Special features

The accurate regulation of the motor speed at arbitrary target values would, in principle, require the output of continuous voltages by the controller. Therefore, even if we accept a certain tolerance in accuracy, a very large action set of control voltages is needed. However, dealing with large action sets means having a large number of potential candidates for each decision, and this drastically increases the complexity of learning an appropriate control policy.

This is a very common problem for many control tasks, not only in robotics. To overcome this problem, we use an integrating output, i.e. a special form of a dynamic output element (Riedmiller 1997). The idea is that the controller does not output the voltage directly, but instead just decides about the decrease or increase of the voltage by a certain amount

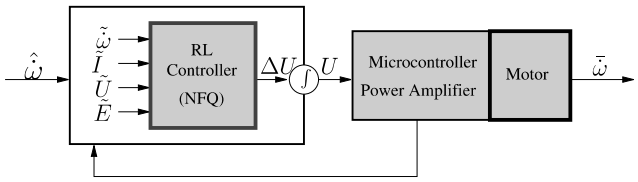

Fig. 8 Scheme of the reinforcement learning controller for the speed control of a DC motor

ΔU. By applying this trick, a wide range of resulting voltages can be produced, whereas the set of actions available to the RL controller remains relatively small. The final action set of the controller is

$$\Delta U \in \{-0.3, -0.1, -0.01, 0.0, 0.01, 0.1, 0.3\}.$$

As a consequence, the state of the MDP that the controller sees is increased by the current state of the integrating output U. This adds one additional component to the input vector of the RL controller.

The final input to the controller consists of the four-dimensional continuous vector $(I, U, E, \dot{\omega})$. Finally, Fig. 8 shows the overall structure of the RL controller.

5.4 Learning procedure

To train the controller, we use NFQ, described in Sect. 3, with a multilayer perceptron (MLP) whose topology consists of 5 neurons in the input layer (4 as a state description, one for the action), 2 hidden layers with 10 neurons each, and a single output neuron (denoting the Q-value of the respective state-action pair).

As a proof of concept for the presented structure of the RL controller, we conducted a first experiment with a single, free-running and therefore load-free real DC motor. For this experiment, we choose a transition sampling procedure that interleaves learning phases and data collection, starting with a randomly initialized neural Q-function. In the data collection phase, the DC motor is controlled by exploiting the current Q-function in an ϵ-greedy manner (exploration is done in 20% of time steps) for 100 time steps. Afterwards, the NFQ-procedure is executed on the set of all transition samples collected so far, and a new data collection phase begins with a randomly selected set point. In Fig. 9, the system behavior in the initial learning phase is shown. Learning was achieved very efficiently, i.e. after only 198 s (little more than 3 minutes) of interaction time with the real system, an effective RL controller was learned from scratch. The resulting performance of the learned reinforcement learning controller is highly satisfying, both with respect to speed and accuracy (Figs. 10 and 11).

This first experiment showed a good performance in the special case of a single free-running motor. The learned controller, however, fails in controlling the motors of the real robot. This is mainly because it has never seen dynamically

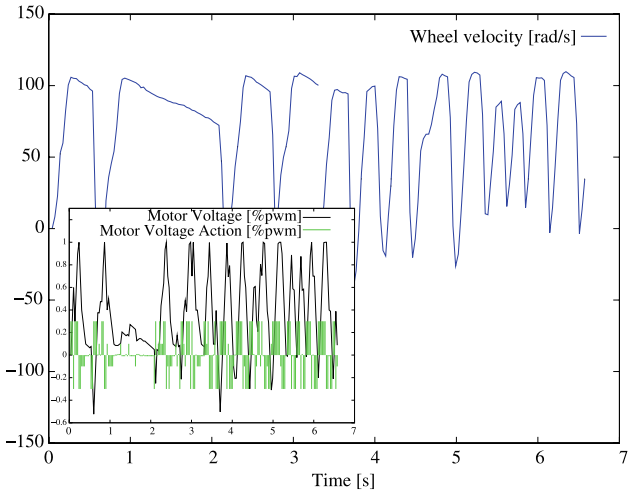

Fig. 9 (Color online) Behavior of the reinforcement learning controller in the early stages of learning. The behavior of the motor speed is rather arbitrary, since the controller has not yet learned a useful policy. All the transition data is collected, stored and used for training. The *small figure* shows how the integration of the RL actions works. The *green lines* denote the actions that are selected by the RL controller, which are integrated to result in the voltage finally applied to the motor (black signal)

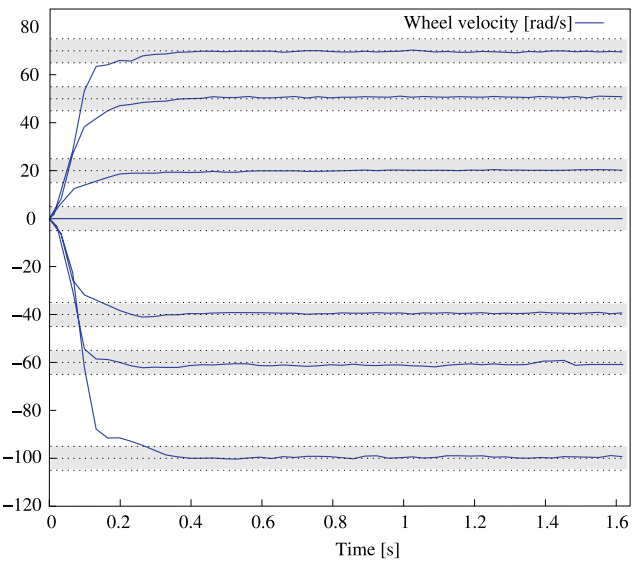

Fig. 10 Performance of the learned controller on the free running real DC motor. Different set points are reached quickly and accurately

changing load situations. In addition the dynamics of the load free motor is almost noise free, allowing smooth control and a very small steady state error. This changes drastically on the real robot, where the interaction of the omnidirectional wheels with the ground cause severe noise due to the shape of the omnidirectional wheels. The interaction of the three motors causes dynamic changing loads for the motors.

To collect data with typical load situations for the controller in its final working range, we have to collect them di-

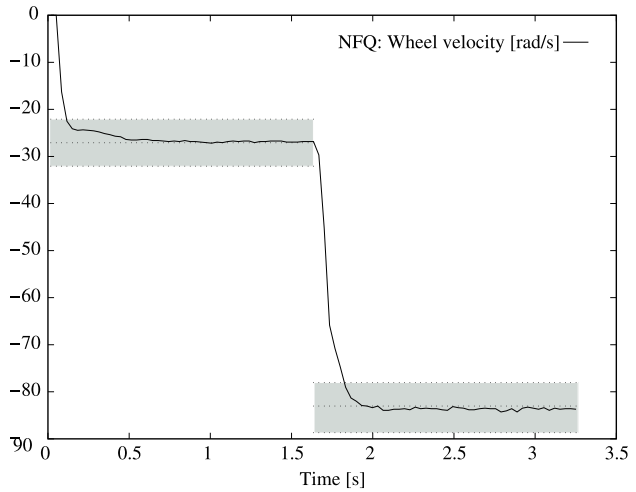

Fig. 11 A trajectory with changing set point. The controller reaches each set-point solely by learning through interaction with the real DC motor

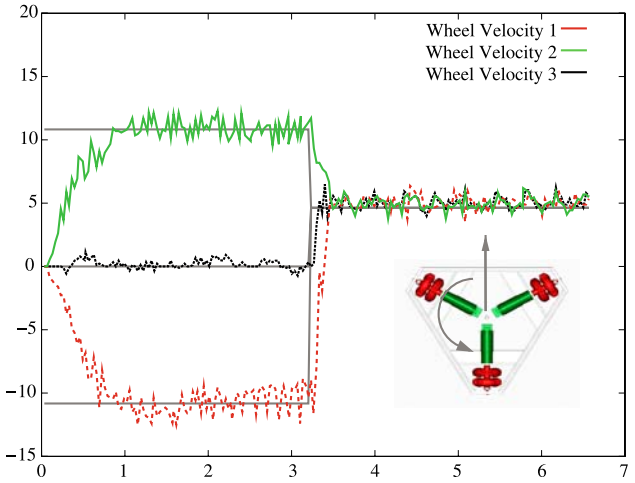

Fig. 12 The learned controller tested on a follow-up control on the real robot. The robot was driven forward with 0.5 m/s changing to a rotational velocity of 2 rad/s. The controller is able to achieve the velocities for all three motors under the presence of noise generated from the wheels

rectly in interaction with the real omnidirectional robot. This was done by putting the robot on the ground and driving it around by applying the controller structure independently of the three motors of the robot. Following this approach, we can collect three different transition samples in different load conditions in each time step. In contrast to the transition sampling procedure that interleaves learning phases and data collection, we decided to pursue another strategy here. Data was collected completely at random, i.e. random control signals were emitted to each of the three motors on trajectories of an episode length of 150 time steps. A new set point was randomly selected for each motor after each data collection episode in order to collect a broad range of set points. After all data-collecting episodes are finished, the controller is trained by NFQ in a purely off-line manor. Using random sampling exclusively is a valid procedure, since NFQ does not require that samples are collected in a certain fashion.

For the application of the controller on the real omnidirectional robot, we ran 50 data collection episodes (each corresponding to a duration of 5 seconds), applying purely random control signals. This gave an overall of $50 \cdot 150 = 7500$ transition samples collected for each motor. Since the data was collected simultaneously for all three motors, this resulted in an overall of $3 \cdot 7500 = 22500$ transition samples that can be used for training the controller within the NFQ framework. The whole process of data collection on the real robot needed only $50 \cdot 5\ \text{s} = 250\ \text{s}$, which is little more than 4 minutes. After only 30 iterations through the NFQ loop, a highly effective controller was learned.

5.5 Final performance

In Fig. 12, the learned controller is shown running on the real robot. The global drive commands used as a demonstration here are ‘drive forward with 0.5 m/s’ and then ‘rotate by 2 rad/s’. The inverse kinematics are used to deliver the respective target speeds for each motor. The task of the learned controller is then to regulate each motor to the desired motor speed.

As shown in Fig. 12, the neural controller has learned to control the motors very quickly and reliably to their desired target speeds. A highly satisfying fact is that the learned speed controller works reliably under the wide range of actual loads that occur within the real robot movement. It has even learned to deal with the considerable noise that occurs in the measured data due to the shape of the wheels. For the complete setup and more information, see (Hafner and Riedmiller 2007).

6 Case study III: learning to dribble on a MidSize robot

6.1 The environment

In the MidSize league, two teams of 6 robots play on a 12 m × 18 m field. The robots are completely autonomous, doing all sensing, information-processing and decision-making on board. Our robot (see Fig. 13) uses a camera as its main sensor and has an omnidirectional drive based on three motors. The sense-act loop is carried out with 30 Hz, which means that a new motor command is sent every 33 ms. A motor command consists of three values denoting v_y^{target} (target forward speed relative to the coordinate system of the robot), v_x^{target} (target lateral speed) and v_θ^{target} (target rotation speed). These values are then transformed into the target motor speeds of the three motors by the use of the inverse kinematics model of the robot.

Fig. 13 Brainstormers MidSize league robot. The difficulty of dribbling lies in the fact that, according to the rules, at most one-third of the ball may be covered by the robot. Not losing the ball while turning, therefore requires a sophisticated control of the robot motion

In our framework, robot skills (such as driving to a certain position, intercepting a ball, dribbling a ball) are implemented as discrete-time closed-loop controllers. The conventional way to design such a skill is to program a base-controller and then fine-tune parameters until the skill works satisfactorily. Obviously, reinforcement learning in this case is highly attractive, not only that computer power can be used instead of man power to develop a skill, but also the learning results can be expected to be superior, even more so if learning is based on an optimization approach.

Learning directly on a real robot has the additional advantage that the controller is directly tuned to the behavior of the actual hardware instead of an idealized model. This is particularly true here, since some real-world effects are extremely difficult to model, e.g. the interaction of the ball with the robot's dribbling device. On the other hand, learning on real robots requires highly data-efficient learning methods, since collecting experience is usually expensive with respect to both time and abrasion of material. The following describes the application of a neural RL controller that directly learns to dribble a ball on a real robot from scratch.

6.2 Task description & learning system setup

Dribbling in the context of this experiment means keeping the ball in front of the robot, while turning to a given target. Since according to the rules of the MidSize league, only one-third of the ball may be covered by a dribbling device, this is quite challenging. The dribbling behavior must carefully control the robot such that the ball does not role away when the robot changes direction. In previous years, we used a hand-coded and carefully hand-tuned routine for this, which was already quite successful (e.g. using it, we won the world championship in 2006 and several European titles). However, it showed weaknesses by failing to execute turns sharply and by occasionally losing the ball.

Within the RL framework, we model the dribbling problem as a terminal state problem with both a terminal goal state and terminal failure states. Intermediate steps are punished by constant costs of $c = 0.01$. We use the Neural Fitted Q Iteration method as the core learning algorithm. The computation of the target value for the batch training set thus becomes:

$$Q^{target}(s,a) := \begin{cases} 1.0, & \text{if } s' \in S^-, \\ 0.01, & \text{if } s' \in S^+, \\ 0.01 + \min_b \tilde{Q}(s',b), & \text{else} \end{cases} \quad (10)$$

where S^- denotes the states, where the ball is lost, and S^+ denotes the states, where the robot has the ball and heads towards the target. State information for the real robot is computed from the camera input and the internal odometry sensors of the robot (see Sect. 3.4). For dribbling, we use a six-dimensional vector with the following real valued entries: speed of the robot in relative x and y direction, rotation speed, x and y ball position relative to the robot and finally the heading direction relative to the given target direction. A failure state $s \in S^-$ is encountered if the ball's relative x coordinate is larger than 50 mm or less than -50 mm, or the relative y coordinate exceeds 100 mm. A success state is reached whenever the absolute difference between the heading angle and the target angle is less than 5 degrees.

The robot is controlled by a three-dimensional action vector, denoting target translational and rotational speeds.

6.3 Special features

When dealing with a real-world system, data acquisition typically becomes a non-trivial issue. Most important, the learning problem has to be formulated such that good solutions can be found while reasonably restricting the degrees of freedom to keep the number of interactions with the real system at a reasonable amount.

Following the above argumentation, we implemented an 'intelligent' interpretation of the rotation value. If the difference between the current body angle and the target direction was negative, the rotation speed was automatically set to a negative value. Therefore, we could restrict the action set to positive rotation speeds. In the same, way the lateral target speed v_x^{target} switched its sign if the rotation speed is negative. Here, however, also negative values for v_x^{target} were allowed.

Overall, 5 different action triples have been used, $U = \{(2.0,\ 0.0,\ 2.0),\ (2.5,\ 0.0,\ 1.5),\ (2.5,\ 1.5,\ 1.5),\ (3.0, 1.0, 1.0),\ (3.0, -1.0, 1.0)\}$, where each triple denotes $(v_x^{target}, v_y^{target}, v_\theta^{target})$ (see Fig. 14).

Actions are selected greedily according to the neural Q value function. For dribbling, it turned out that it suffices if actions are reconsidered every second time step. This re-

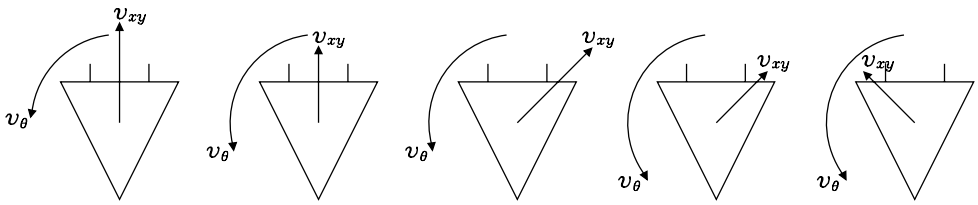

Fig. 14 Action set of the dribbling controller. 5 different combinations of forward and lateral speed (v_{xy}) and rotation speed (v_θ) are available for choice

duces the number of decisions per trajectory and therefore simplifies the learning problem.

Input to the Neural Fitted Q-iteration method is a set of transition triples of the form (state, action, successor state). A common procedure to sample these transitions is to alternatively train the Q function and then sample new transitions episode-wise by greedily exploiting the current Q function. However, on the real robot, this means, that between each data collection phase one has to wait until the new Q function has been trained. This can be annoying, since putting the ball back on the field requires human intervention. Therefore, we decided to go for a batch-sampling method, which collects data over multiple trials without relearning.

At the beginning of each trial, the robot waits until the ball is put onto the middle of the field, before moving to a starting position 2 m away from the ball. Next, it drives towards the ball and as soon as it gets there, the dribbling trial is started. In every trial, a different target direction is given. Here, we collected batches of 12 trials each without retraining the neural controller within a batch. After each batch, the sampled transitions are added to the data set, and learning is started. If the set of target values used for the 12 trials are the same for each batch, then concurrently to data sampling, the performance of the controllers can be evaluated and compared. This was the case here. An extension of this method is to do some exploration in one part of the batch and be greedy in another part. Then, performance can be judged when the controller is greedily exploited, and seeing a diversity of data is guaranteed by the exploration part. However, for this study, always greedily following the value function turned out to be sufficient.

6.4 Learning procedure

For learning, we use the Neural Fitted Q framework described in Sect. 3. The value function is represented by a multilayer perceptron with 9 input units (6 state variables and 3 action variables), 2 hidden layers of 20 neurons each and 1 output neuron. After each batch of 12 trials, we did 10 NFQ iterations, where in each iteration we computed the target values according to equation 10. Learning the target values was done in 300 epochs of supervised batch learning, using the Rprop learning method with standard parameters. After learning was finished, the new controller was used to control the robot during the next data collection phase. After

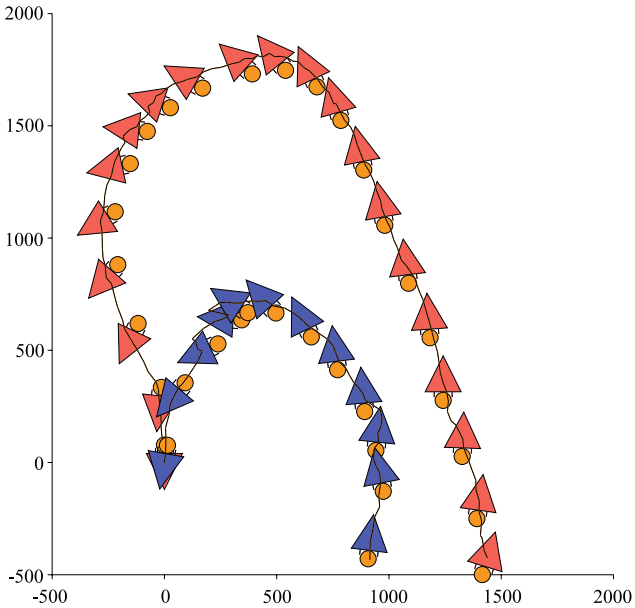

Fig. 15 Comparison of hand-coded (*outer trajectory*, *red*) and neural dribbling behavior (*inner trajectory*, *blue*) when requested to make a U-turn. The data was collected on our real robot. When the robot gets the ball, it typically has an initial speed of about 1.5 to 2 m/s in forward direction. The positions of the robot are displayed every 120 ms. The U-turn of the neural dribbling controller is much sharper and faster

11 batches (= 132 trials), a very good controller was learned. The complete learning procedure took about one and a half hour, including the time used for offline updating of the neural Q function. The actual interaction time with the real robot was about 30 minutes, including preparation phases.

6.5 Final performance

The neural dribbling controller is implemented as an autonomous skill within our robot control architecture. The behavior is called with a certain target direction and the current state information. It returns a three-dimensional drive vector consisting of rotation speed v_θ, the relative forward speed v_y, and the relative lateral speed v_x. It was the first behavior in our MidSize robot that was completely learned on the real robot. For intercepting the ball, we also use a learned behavior, but it was learned by the use of a simulator. The neural dribbling skill performed significantly better than the previously used hand-coded and hand-tuned dribbling routine, especially in terms of space and time needed to turn to the desired target direction (see Fig. 15).

The neural dribbling skill has been successfully used in our competition team since 2007. With its help, we won the world championship 2007 in Atlanta and became third at the world championship in 2008 in Suzhou, China.

7 RL in Brainstormers' competition teams

The Brainstormers project was started in our research group in 1998 with the goal to develop autonomous soccer robots that are able to learn to act by the extensive use of machine learning techniques, with a particular focus on reinforcement learning methods. By participating in the annual RoboCup championships, we aim at demonstrating the effectiveness of learning approaches in a highly competitive field. We see this competitive evaluation as one important contribution for reaching maturity of learning methods with respect to practical applicability.

The skills that have been learned (see Table 2 for an overview) range from direct motor control skills (like motor speed control, Case Study II) to individual skills (like dribbling, Case Study III or intercepting the ball), to skills interfering with opponents (like the aggressive-defense behavior, Case Study I) and complex multiagent behaviors (like cooperative attack play, considering 7 attacking team players and 8 defending opponent players, Riedmiller et al. 2003). In order to finally yield a competitive software agent, the individual skills must also prove their superiority to alternative approaches. Therefore, all the learned skills that made it inside our competition code have proven their superiority compared to previous hand-coded versions. When all the learned skills are activated in our simulation league agent, up to 80 percent of the decisions in a game are triggered by neural networks (e.g. when activating all neural skills in our 2005 agent, a neural network is involved in decision-making on average in 56.8% (defender), 73.0% (sweeper), 84.4% (midfielder), 82.6% (attacker), of its total number of actions).

Table 2 Overview of a selection of behaviors that were learned by neural reinforcement learning methods for the Brainstormers' simulation league and MidSize league teams over the years from 2000 to 2008. Filled dot denotes the application in the competition team, empty dot denotes that the skill was successfully learned, but finally did not make it into the competition team. Many of the skills have been improved from year to year

	'00	'01	'02	'03	'04	'05	'06	'07	'08
Simulation League									
NeuroKick	•	•	•	•	•	•	•	•	•
NeuroIntercept	•	•	•	•		○	○		
NeuroGo2Pos	•	•	•	•	•				
NeuroADB								•	•
NeuroAttack	○	•	•	•	•		•	•	•
NeuroPenalty				•	•	•	•	•	•
Rank	2	2	3	3	2	1	2	1	1
MidSize League									
NeuroMotorSpeed								○	○
NeuroGo2Pos							○	○	○
LmapIntercept							•	•	•
NeuroDribble								•	•
Rank							1	1	3

In almost all cases, we used neural networks for representing the value function, using the batch RL framework described in Sect. 3 as the general learning framework. The state dimension typically ranges from 5 to 10 real valued state variables, the number of discrete actions is typically in the range of up to 10 for real robot learning and up to several hundreds of actions for learning in the simulation league agent.

8 Conclusions

Real-world applications of reinforcement learning methods require highly data-efficient and robust learning algorithms. The batch RL paradigm discussed in Sect. 3 provides a useful framework that can be adapted in various ways according to the concrete requirements of the learning task to be solved. We demonstrated the concrete application of this concept in three case studies, all within the context of our Brainstormers' robotic soccer project. Learned skills have been vastly and successfully applied over more than 8 years in our competition teams. The learning tasks faced typically have continuous state spaces, a considerable amount of state dimensions and rich action sets.

The success of the Brainstormers project is not only documented by the record of rankings in the competitions (we won 5 world championships, several European titles, and multiple 2nd and 3rd places), but also by the development of the learning methods with respect to their increased practical applicability. While at the beginning of our project, often hundreds of thousands of episodes had to be done to learn a successful policy, the increased robustness and data-efficiency of the algorithms has lead to learning systems that are now able to actually learn on real robots from scratch.

Acknowledgements Thanks to Christian Müller for helping with the neuro dribbling experiments on our MidSize league robot. We gratefully acknowledge the support of our industrial sponsors, in particular Harting Technology Group, who substantially supports our work both technically and financially. This work was sponsored in part by a DFG-grant within the SPP 1125.

References

Asada, M., Uchibe, E., & Hosoda, K. (1999). Cooperative behavior acquisition for mobile robots in dynamically changing real worlds

via vision-based reinforcement learning and development. *Artificial Intelligence*, *110*(2), 275–292.

Bagnell, J., & Schneider, J. (2001). Autonomous helicopter control using reinforcement learning policy search methods. In *Proceedings of the 2001 IEEE international conference on robotics and automation (ICRA 2001)* (pp. 1615–1620), Seoul, South Korea. New York: IEEE Press.

Behnke, S., Egorova, A., Gloye, A., Rojas, R., & Simon, M. (2003). Predicting away robot control latency. In D. Polani, B. Browning, A. Bonarini, & K. Yoshida (Eds.), *LNCS. RoboCup 2003: robot soccer world cup VII* (pp. 712–719), Padua, Italy. Berlin: Springer.

Bellman, R. (1957). *Dynamic programming*. Princeton: Princeton University Press.

Bertsekas, D., & Tsitsiklis, J. (1996). *Neuro dynamic programming*. Belmont: Athena Scientific.

Chernova, S., & Veloso, M. (2004). An evolutionary approach to gait learning for four-legged robots. In *Proceedings of the 2004 IEEE/RSJ international conference on intelligent robots and systems (IROS 2004)*, Sendai, Japan. New York: IEEE Press.

Crites, R., & Barto, A. (1995). Improving elevator performance using reinforcement learning. In *Advances in neural information processing systems 8 (NIPS 1995)* (pp. 1017–1023), Denver, USA. Cambridge: MIT Press.

Ernst, D., Geurts, P., & Wehenkel, L. (2006). Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research*, *6*(1), 503–556.

Gabel, T., & Riedmiller, M. (2007). Adaptive reactive job-shop scheduling with learning agents. *International Journal of Information Technology and Intelligent Computing, 2*(4).

Gabel, T., Hafner, R., Lange, S., Lauer, M., & Riedmiller, M. (2006). Bridging the gap: learning in the RoboCup simulation and midsize league. In *Proceedings of the 7th Portuguese conference on automatic control (Controlo 2006)*, Porto, Portugal.

Gabel, T., Riedmiller, M., & Trost, F. (2008). A case study on improving defense behavior in soccer simulation 2D: the NeuroHassle approach. In Iocchi, L., Matsubara, H., Weitzenfeld, A., & Zhou, C. (Eds.), *LNCS. RoboCup 2008: robot soccer world cup XII*, Suzhou, China. Berlin: Springer.

Gordon, G., Prieditis, A., & Russell, S. (1995). Stable function approximation in dynamic programming. In *Proceedings of the twelfth international conference on machine learning (ICML 1995)* (pp. 261–268), Tahoe City, USA. San Mateo: Morgan Kaufmann.

Hafner, R., & Riedmiller, M. (2007). Neural reinforcement learning controllers for a real robot application. In *Proceedings of the IEEE international conference on robotics and automation (ICRA 07)*, Rome, Italy. New York: IEEE Press.

Kaufmann, U., Mayer, G., Kraetzschmar, G., & Palm, G. (2004). Visual robot detection in RoboCup using neural networks. In D. Nardi, M. Riedmiller, C. Sammut, & J. Santos-Victor (Eds.), *LNCS. RoboCup 2004: robot soccer world cup VIII* (pp. 310–322), Porto, Portugal. Berlin: Springer.

Kitano, H. (Ed.). (1997). *RoboCup-97: robot soccer world cup I*. Berlin: Springer.

Kober, J., Mohler, B., & Peters, J. (2008). Learning perceptual coupling for motor primitives. In *Proceedings of the 2008 IEEE/RSJ international conference on intelligent robots and systems (IROS 2008)* (pp. 834–839), Nice, France. New York: IEEE Press.

Lagoudakis, M., & Parr, R. (2003). Least-squares policy iteration. *Journal of Machine Learning Research*, *4*, 1107–1149.

Lauer, M., Lange, S., & Riedmiller, M. (2005). Calculating the perfect match: an efficient and accurate approach for robot self-localization. In A. Bredenfeld, A. Jacoff, I. Noda, & Y. Takahashi (Eds.), *LNCS. RoboCup 2005: robot soccer world cup IX* (pp. 142–153), Osaka, Japan. Berlin: Springer.

Lauer, M., Lange, S., & Riedmiller, M. (2006). Motion estimation of moving objects for autonomous mobile robots. *Kunstliche Intelligenz*, *20*(1), 11–17.

Li, B., Hu, H., & Spacek, L. (2003). An adaptive color segmentation algorithm for Sony legged robots. In *The 21st IASTED international multi-conference on applied informatics (AI 2003)* (pp. 126–131), Innsbruck, Austria. New York: IASTED/ACTA Press.

Lin, L. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, *8*(3), 293–321.

Ma, J., & Cameron, S. (2008). Combining policy search with planning in multi-agent cooperation. In L. Iocchi, H. Matsubara, A. Weitzenfeld, & C. Zhou (Eds.), *LNAI. RoboCup 2008: robot soccer world cup XII*, Suzhou, China. Berlin: Springer.

Nakashima, T., Takatani, M., Udo, M., Ishibuchi, H., & Nii, M. (2005). Performance evaluation of an evolutionary method for RoboCup soccer strategies. In A. Bredenfeld, A. Jacoff, I. Noda, & Y. Takahashi (Eds.), *LNAI. RoboCup 2005: robot soccer world cup IX*, Osaka, Japan. Berlin: Springer.

Ng, A., Coates, A., Diel, M., Ganapathi, V., Schulte, J., Tse, B., Berger, E., & Liang, E. (2004). Autonomous inverted helicopter flight via reinforcement learning. In *Experimental robotics IX, the 9th international symposium on experimental robotics (ISER)* (pp. 363–372), Singapore, China. Berlin: Springer.

Noda, I., Matsubara, H., Hiraki, K., & Frank, I. (1998). Soccer server: a tool for research on multi-agent systems. *Applied Artificial Intelligence*, *12*(2–3), 233–250.

Ogino, M., Katoh, Y., Aono, M., Asada, M., & Hosoda, K. (2004). Reinforcement learning of humanoid rhythmic walking parameters based on visual information. *Advanced Robotics*, *18*(7), 677–697.

Oubbati, M., Schanz, M., & Levi, P. (2005). Kinematic and dynamic adaptive control of a nonholonomic mobile robot using a RNN. In *Proceedings of the 20005 IEEE international symposium on computational intelligence in robotics and automation (CIRA 2005)* (pp. 27–33). New York: IEEE Press.

Peters, J., & Schaal, S. (2006). Policy gradient methods for robotics. In *Proceedings of the IEEE/RSJ international conference on intelligent robots and systems (IROS)*, Beijing, China. New York: IEEE Press.

Peters, J., & Schaal, S. (2008a). Learning to control in operational space. *The International Journal of Robotics Research*, *27*(2), 197–212.

Peters, J., & Schaal, S. (2008b). Reinforcement learning of motor skills with policy gradients. *Neural Networks*, *21*(4), 682–697.

Puterman, M. (2005). *Markov decision processes: discrete stochastic dynamic programming*. New York: Wiley-Interscience.

Riedmiller, M. (1997). Generating continuous control signals for reinforcement controllers using dynamic output elements. In *Proceedings of the European symposium on artificial neural networks (ESANN 1997)*, Bruges, Belgium.

Riedmiller, M. (2005). Neural fitted Q iteration—first experiences with a data efficient neural reinforcement learning method. In *Machine learning: ECML 2005, 16th European conference on machine learning*, Porto, Portugal. Berlin: Springer.

Riedmiller, M., & Braun, H., (1993). A direct adaptive method for faster backpropagation learning: the RPROP algorithm. In H. Ruspini (Ed.), *Proceedings of the IEEE international conference on neural networks (ICNN)* (pp. 586–591), San Francisco.

Riedmiller, M., & Merke, A. (2003). Using machine learning techniques in complex multi-agent domains. In I. Stamatescu, W. Menzel, M. Richter, & U. Ratsch (Eds.), *Adaptivity and learning*. Berlin: Springer.

Riedmiller, M., Montemerlo, M., & Dahlkamp, H. (2007). Learning to drive in 20 minutes. In *Proceedings of the FBIT 2007 conference*, Jeju, Korea. Berlin: Springer.

Röfer, T. (2004). Evolutionary gait-optimization using a fitness function based on proprioception. In Nardi, D., Riedmiller, M., Sammut, C., & Santos-Victor, J. (Eds.), *LNCS. RoboCup 2004: robot soccer world cup VIII* (pp. 310–322), Porto, Portugal. Berlin: Springer.

Stone, P., Sutton, R., & Kuhlmann, G. (2005). Reinforcement learning for RoboCup-soccer keepaway. *Adaptive Behavior*, *13*(3), 165–188.

Sutton, R., & Barto, A. (1998). *Reinforcement learning. An introduction*. Cambridge: MIT Press/A Bradford Book.

Sutton, R., McAllester, D., Singh, S., & Mansour, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems 12 (NIPS 1999)* (pp. 1057–1063), Denver, USA. Cambridge: MIT Press.

Tesauro, G., & Galpering, G. (1995). On-line policy improvement using Monte Carlo search. In *Neural information processing systems (NIPS 1996)* (pp. 206–221), Denver, USA. Berlin: Springer.

Tesauro, G., & Sejnowski, T. (1989). A parallel network that learns to play backgammon. *Artificial Intelligence*, *39*(3), 357–390.

Treptow, A., & Zell, A. (2004). Real-time object tracking for soccer-robots without color information. *Robotics and Autonomous Systems*, *48*(1), 41–48.

Watkins, C., & Dayan, P. (1992). Q-learning. *Machine Learning*, *8*, 279–292.

Wehenkel, L., Glavic, M., & Ernst, D. (2005). New developments in the application of automatic learning to power system control. In *Proceedings of the 15th power systems computation conference (PSCC05)*, Liege, Belgium.

Martin Riedmiller studied computer science at the University of Karlsruhe, Germany, where he received his diploma in 1992 and his PhD in 1996. In 2002 he became a professor for Computational Intelligence at the University of Dortmund. From 2003 to 2009 he was heading the Neuroinformatics Group at the University of Osnabrueck. Since April 2009 he is a professor for Machine Learning at the Albert-Ludwigs-University Freiburg. His research interests are machine learning, neural networks, reinforcement learning and robotics.

Thomas Gabel (diploma degree in Computer Science 2003 at the University of Kaiserslautern) is a researcher at the Machine Learning Group at the University of Freiburg. He works in the fields of machine learning and reinforcement learning, with a focus on multi-agent systems, as well as in knowledge management and case-based reasoning. He is involved in RoboCup activities as team leader of the robotic soccer simulation team Brainstormers, and won several world championships. He is organizing chair of the RoboCup 2009 simulation league in Graz.

Roland Hafner studied computer science at the University of Karlsruhe, Germany, where he received his diploma in 2002. In 2009 he will finish his PhD at the University of Osnabrueck. His research interests are Reinforcement Learning in feedback control applications, Neural Networks and autonomous robots.

Sascha Lange studied Cognitive Science at the University of Gothenburg and at the University of Osnabrueck, where he received his master degree in 2004. Currently he is a research assistant in the Neuroinformatics Group at the University of Osnabrueck. His research topics include machine learning, computer vision and autonomous robots.