Learning in Robot Soccer

Tobias Thiel
t.thiel@stud.tu-darmstadt.de
Department of Computer Science
Technische Universität Darmstadt

Abstract

This paper presents five case studies of successful applications of reinforcement learning methods to learn specific subtasks in robot soccer. The used learning algorithms are introduced, followed by descriptions of each case study on its own, by introducing the task, the machine learning techniques and finally the results. The case studies range from motor control over simulated soccer to actually kicking the ball.

1 Introduction

Robot soccer is an active field of research which aims to build a team of fully autonomous robots playing soccer. It is a domain which is fully distributed with multiple agents having both teammates and adversaries [1]. Additionally agents have noisy sensors and actions, which means that they do not have a complete view of the world around them and their actions are not always executed as commanded, making it difficult to model these real world effects [2]. When one tries to tackle this problem, another problem arises, as the state space in robot soccer is often very large and continuous, so that it becomes difficult to develop efficient agents. Often this problem is solved by modeling the state space using approximated functions. Algorithms to achieve that will be introduced later in Section 3. Before that some of the benefits of using machine learning in the domain of robot soccer will be discussed in Section 2. Then the five case studies are presented in the following sections:

- *4. RoboCup Soccer Keepaway* [2]
- *5. Learning powerful kicks* [3]
- *6. Learning aggressive defense behavior* [4]
- *7. Learning motor speed control* [4]
- *8. Learning to dribble* [4]

The *final section* will summarize and conclude this paper.

2 Why use machine learning?

Typically robot movements are hand-coded and manually tuned. This is a very challenging task, a lot of work and might not even lead to an optimal solution for the problem at hand, because developers are biased to the solution they think works and might not think of another solution that may be better. Using machine learning on the other hand reduces the manual effort to the implementation of the machine learning framework and modeling of the states. Above all machine learning algorithms remove the human bias from the solution and were successfully used in several large-scale domains just like robot soccer: e.g., backgammon [5], helicopter control [6] and elevator control [7]. This list focuses on successes with reinforcement learning methods, as these will be the main methods used in the presented case studies.

But there are also many successes using machine learning in robot soccer. The presented case studies [2] [3] [4] can be viewed as successful applications, but machine learning was also used to capture the ball by Fidelman and Stone [8], receiving passes by Kobayashi et al. [9], as well as walking by Quinlan et al. [10]. These are just some of the many examples of successful applications of machine learning algorithms in the robot soccer domain or overall, that should show that testing them might be an interesting next step.

3 Algorithms

In the following all the needed algorithms for the presented case studies will be introduced. In addition to algorithms from reinforcement learning, supervised learning and optimization algorithms are also used for learning.

Since the state space in robot soccer is rather large, there is also a need for function approximation, since rewards or expected values cannot be saved for every state and therefore are represented as functions [2] [4]. These approximation algorithms will also be introduced.

3.1 Reinforcement learning

A lot of reinforcement learning problems can be casted as a Markov Decision Process (MDP). A MDP M is a 4-tuple $M = (S, A, p, c)$ with S being a set of states and A a set of actions for the given problem. $c : S \times A \times S \to \Re$ is a function denoting the cost when taking a specific $a \in A$ in a state $s \in S$ and transitioning to the next state $s' \in S$. $p : S \times A \times S \to [0, 1]$ denotes the probability of ending up in state s' when performing action a in state s. The goal is to minimize the cost, by learning a decision policy $\pi : S \to A$ that returns the best action to take for a given state [4].

This section will introduce a few variations of *value iteration* and *Q-learning* used in almost all case studies.

3.1.1 Neural fitted value iteration

Neural fitted value iteration is based on value iteration and tries to calculate an optimal cost-to-go or value function $J^*(s)$, denoting the expected value of the sum

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

with t being a discrete time-step and $\gamma \in [0, 1]$ determining to which amount future costs are discounted compared to immediate ones. Once $J^*(s)$ is known it is easy to get the optimal control policy

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

The essential question therefore is how to obtain $J^*(s)$. Value iteration just calculates successive updates to the cost-to-go function for all states based on the previous approximation [4].

Neural fitted value iteration starts with an initial cost-to-go function J_0, which can be randomly initialized, and then iteratively approximates $J^*(s)$, by using a supervised learning module. First a training set $\mathcal{P}$ is extracted like this:

$$\tilde{J}_{k+1} := min_a \left\{ \sum_{s'} p_{ss'}(a)(c(s, a, s') + \gamma J_k(s')) \right\},$$
$$\mathcal{P}_{k+1} := \left\{ (s, \tilde{J}_{k+1}) | s \in I \right\}.$$

The experience set $I \subseteq S$ contains representative states to be used in the training set. The training set is then inserted into a supervised learning module, to calculate a new approximation of $J^*(s)$ called $J_{k+1}(s)$ which is used as the new starting point for the next iteration [4].

The advantage of using *Neural fitted value iteration* (NFV) compared to just value iteration is that NFV does not need to recalculate J_k every time for all available states, but can rather focus on representative and interesting regions in the state space to calculate good approximations.

3.1.2 Neural fitted Q-iteration (NFQ)

Just as Q-learning is the model-free version of the value iteration algorithm, *Neural fitted Q-iteration* (NFQ) is a version of NFV that does not require the world model p. The algorithm iteratively approximates a function $Q^*(s, a)$ just like NFV does [4]:

$$\tilde{Q}_{k+1} := c(s, a, s') + \gamma min_{a \in A(s)} Q_k(s', a),$$
$$\mathcal{P}_{k+1} := \{((s, a), \tilde{Q}_{k+1}(s, a) | (s, a, \cdot) \in I)\}.$$

The training set $\mathcal{P}$ is once again fed into a supervised learning module, to calculate a better approximation. Once a satisfying approximation of $Q^*(s, a)$ is reached the approximated optimal control policy can be determined like this [4]:

$$\pi^*(s) = argmin_{a \in A(s)} Q^*(s, a).$$

A disadvantage of NFQ compared to NFV is that NFVs experience set I does not need to be sampled by interaction with the actual system, because the set only contains the states to use. NFQs experience set additionally contains the corresponding actions to be taken and the resulting state, because the model cannot be used anymore. Therefore the set has to filled by interaction with the actual system, which slows the process of learning down.

3.1.3 Neural fitted policy iteration

Contrary to the two previous algorithms, *Neural fitted policy iteration* (NFP) relies upon an existing policy to calculate the cost-to-go function behind the policy, optimizing it and then using it to generate a new policy. The structure is nevertheless very similar to the other two algorithms. First the training set for the supervised learning module is filled:

$$\tilde{J}_k(s) := E\left\{\sum_t c(s_t, \pi_k(s_t), s_{t+1}) | s_o = s\right\},$$
$$\mathcal{P}_k := \{(s, \tilde{J}_k(s)) | s \in I\}.$$

This training set is then used to approximate a new cost-to-go function $J_k^\pi(s)$ using a supervised learning module, to then derive a new policy:

$$\pi_{k+1}(s) := argmin_{a \in A(s)} \sum_{s' \in S} p_{ss'}(a)(c(s, a, s') + \gamma J_k^\pi(s')).$$

This policy is then used in the next iteration of the algorithm. Another major difference of NFP is that the experience set is filled by sampling states alongside trajectories from different starting points by following the current policy π_k. Because of that the experience set has to be refilled from scratch in every iteration, since the experience cannot necessarily be reused with a different policy. Although this make NFP less data-efficient, Riedmiller et al. (2009) found that the algorithm was able to find very good policies in few iterations, especially when generating experience is not a problem, e.g., when having access to a simulator [4].

3.1.4 SARSA

SARSA (State-Action-Reward-State-Action) is an on-policy learning algorithm introduced by Rummery & Niranjan [11], which is very similar to Q-learning. Q-learning updates are given by:

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

whereas Sarsa uses the following update function:

$$Q(s, a) := Q(s, a) + \alpha(c(s, a, s') + \gamma Q(s', b) - Q(s, a)).$$

α in these equations denotes a learning factor, γ as before a discount factor for future rewards. The difference is that Sarsa learns the Q values from the policy it follows itself, whereas Q-learning uses the actions of minimal cost to update the Q values while following potentially a different policy, if actions are not taken because of the hidden world model [2].

3.2 Supervised learning

Supervised learning is another machine learning effort, which needs training data containing the expected results for given input, which it can then use to optimize its procedure, to deliver the best generalized output possible. In this paper the only supervised learning method used are multilayer perceptrons, which are used to approximate functions.

3.2.1 Multilayer perceptrons (MLP)

Multilayer perceptrons are neural networks which can in contrast to normal perceptrons model non-linear functions accurately. The network contains neurons in layers and each neuron of a layer is connected to every neuron of the next layer, but there are no connections between neurons in a layer and no connections skipping a layer. Each connection from a node i of layer k to a node j of layer $k+1$ has a associated weight w_{ij}. The neurons of the input layer $k=0$ are assigned the input values for which the network calculates an output value y. The input value of a neuron x_i is used to calculate its output value y_i:

$$y_i = f(x_i).$$

The activation function f is most of the times a sigmoid function like the logistic function, because the calculation of the derivative is easy [12]:

$$f(x) = \frac{1}{1+e^{-x}}.$$

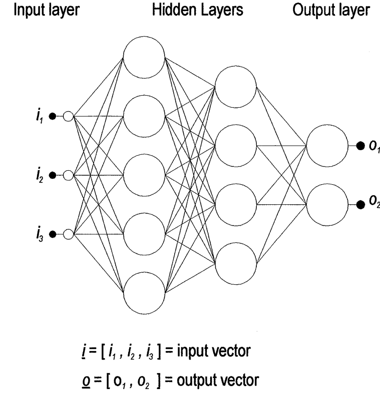

Figure 1: Example multilayer perceptron with one hidden layer [12]

Output values of a neuron are then propagated to the next layer by weighing the output value y_i with the associated weight w_{ij} and feeding the sum of all connections as input value to neuron j of the next layer.

In training mode the given expected output value T_j is used to calculate the error and then back-propagates changes to the weights. There are differing strategies to calculate the error and the resulting weight updates, so this paper will focus on the strategy presented in [13]. For the output layer this calculation is as follows:

$$\begin{aligned} x_j =& \sum_i y_i w_{ij}, \\ \epsilon_j =& (T_j - y_j) f'(x_j), \\ w_{ij} =& w_{ij} + \epsilon_j y_i \eta. \end{aligned}$$

x_j is the input value of the output neuron j and ϵ_j is the error propagated back, while η is the learning rate. f' is used to squash the values of ϵ_j. To calculate the weight updates in the previous layers the following calculation for ϵ_j is used [13]:

$$\epsilon_j = f'(x_j) \sum_k (\epsilon_k w_{jk}).$$

Thus instead of using the error to the expected output value, the error of all neurons from the next layer weighted by the connection weight is summed up and used in the calculation of the error in the hidden layers. Once the error in the output layer is within satisfying bounds the weight updates are aborted and the perceptron is ready to calculate values for arbitrary inputs.

The number of neurons in the input and output layers are dependent on the dimensionality of the function one wishes to approximate. The number of hidden layers and the neurons within are dependent on the task at hand and therefore need to be found through experimentation.

3.3 Optimization

Optimization algorithms select the best element from a set of available elements using a given criteria to evaluate the elements. This section will introduce two algorithms which will be used for learning to kick the ball in *Learning powerful kicks*.

3.3.1 Hill climbing

Hill climbing is an iterative optimization algorithm which finds a local optimum. It starts with an initial solution and then tries to generate a better solution by changing the parameters of the solution. Since it only makes minor changes to the parameters, it only explores neighboring configurations and therefore can only guarantee to find a local optimum.

The N dimensional hill climbing algorithm needs an initial policy $\pi = \{\theta_1, ..., \theta_j, ..., \theta_N\}$. It then generates t random new policies $R_i = \{\theta_1 + \delta_1, ..., \theta_j + \delta_j, ..., \theta_N + \delta_N\}$. Each δ_j is randomly determined to be either $+\epsilon_j$, 0 or $-\epsilon_j$, whose values are dependent on the actual task one wishes to learn, but are supposed to be small compared to θ_j. Next the new policies R_i are evaluated using a custom scoring function and the policy with the highest score is chosen as starting point for the next iteration [14].

Pseudo-code for the algorithm can be found below or in [14].

Algorithm 1: Hill Climbing pseudo-code

```
π ← InitialPolicy
while !done do
  {R_1, R2, ..., R_t} = t random permutations of π
  evaluate({R_1, R2, ..., R_t})
  π ← getHighestScoring({R_1, R2, ..., R_t})
end
```

3.3.2 Policy gradient

This version of the *policy gradient algorithm* basically extends the hill climbing algorithm, but instead of always choosing the best generated policy, the gradient of the objective function in the parameter space is approximated. There exist newer versions of this algorithm which use a different approximation of the gradient like in [15], but this paper will focus on the version used by the case studies to reach the described results.

The algorithm also starts with an initial policy π and generates t new random policies $R_i = \{\theta_1 + \delta_1, ..., \theta_N + \delta_N\}$ with δ_j either $+\epsilon_j$, 0 or $-\epsilon_j$. The next step is to build three sets out of the R_i by putting each one in either $S_{+\epsilon,n}$, $S_{0,n}$ or $S_{-\epsilon,n}$ like this:

$$R_i \in \begin{cases} S_{+\epsilon,n} & \text{if the } n\text{-th parameter of } R_i \text{ is } \theta_n + \epsilon_n \\ S_{0,n} & \text{if the } n\text{-th parameter of } R_i \text{ is } \theta_n + 0 \\ S_{-\epsilon,n} & \text{if the } n\text{-th parameter of } R_i \text{ is } \theta_n - \epsilon_n \end{cases}$$

Now all R_i and an average score for the three sets containing them can be calculated. These averages are $Avg_{+\epsilon,n}$, $Avg_{0,n}$ or $Avg_{-\epsilon,n}$ corresponding to the sets and express the benefit of altering parameter n by $+\epsilon_j$, 0 or $-\epsilon_j$. Using these scores the final adjustment vector A can be calculated as follows:

$$A_n = \begin{cases} 0 & \text{if } Avg_{0,n} > Avg_{+\epsilon,n} \\ & \text{and } Avg_{0,n} > Avg_{-\epsilon,n} \\ Avg_{+\epsilon,n} - Avg_{-\epsilon,n} & \text{otherwise} \end{cases}$$

After normalizing the vector it is multiplied with the scalar step-size η, to finally be added to the current policy π and starting the next iteration with the modified policy [14].

Pseudo-code for the algorithm as in [14] follows:

Algorithm 2: Policy Gradient pseudo-code

```
π ← InitialPolicy
while !done do
  {R_1, R2, ..., R_t} = t random permutations of π
  evaluate({R_1, R2, ..., R_t})
  for n = 1 to N do
    Avg_{+ε,n} ← average score of all R_i where the n−th parameter is θ_n + ε_n
    Avg_{0,n} ← average score of all R_i where the n−th parameter is θ_n + 0
    Avg_{−ε,n} ← average score of all R_i where the n−th parameter is θ_n − ε_n
```

```
    if  $Avg_{0,n} > Avg_{+\epsilon,n}$  and  $Avg_{0,n} > Avg_{-\epsilon,n}$
      $A_n \leftarrow 0$
    else
      $A_n \leftarrow Avg_{+\epsilon,n} - Avg_{-\epsilon,n}$
    end
  end
  $A \leftarrow \frac{A}{|A|} * \eta$
  $\pi \leftarrow \pi + A$
end
```

3.4 Function approximators

In this section methods to approximate functions will be introduced, which will be used to represent the large state space to generalize finite number of samples to a large state space. The only method that will be introduced is *Tile coding* since other approximation methods that are used, are supervised learning methods and therefore introduced there.

3.4.1 Tile coding

Tile coding is used as a function approximation algorithm to not have to save things like value functions for large state spaces. The procedure is to partition the space into tiles. One such partition is called a tiling and there can exist multiple axis-parallel tilings overlapping (see Figure 2). One value in the space then activates one tile in each tiling, called an active tile (highlighted tiles in Figure 2). The value is then represented by these active tiles and its location can be approximately reconstructed by building the intersection of all active tiles. Often tilings all have the same width, but are offset by a fixed amount from the previous tiling. This method has the advantage to generalize more broadly when looking at a individual tiling, but when taking all the tilings into consideration, the values can still be reconstructed without much loss. When using the described placement with a tile width w and number of tilings t with a constant offset, the resolution can be expressed as w/t. Additionally one can add weights to the individual tiles, to express the importance of one or multiple specific features in a tiling. [16].

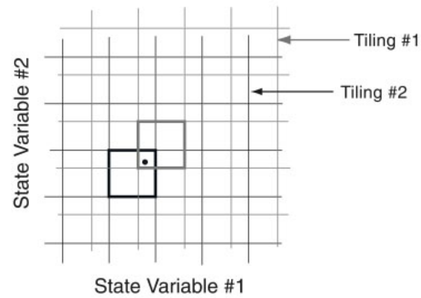

Figure 2: Tile coding example [2]

4 RoboCup soccer keepaway

The first case study which will be presented is *Reinforcement Learning for RoboCup Soccer Keepaway* from Stone, Sutton and Kuhlmann. The detailed report can be found in [2].

4.1 Task description

RoboCup Soccer Keepaway is a simulated soccer match, where two teams compete each other with the goal to hold the ball as long as possible in the own team. One team, called the keepers, try to hold the ball as long as possible without losing the ball to the other team, called the takers, who try to take the ball from the keepers. A time period where the keepers were able to hold the ball is called an episode. An episode ends when the takers get possession of the ball or when the ball leaves the playing field. Keepaway is obviously a subproblem of the whole robot soccer domain [2].

The simulator used is the standard RoboCup soccer simulator whose full details can be found in [17] [18]. Agents receive visual perceptions every 150ms, giving them the distance and angle to visible objects. They can send actions like turn(*angle*), dash(*power*) or kick(*power, angle*) every 100ms, forcing the agents to act asynchronously from their sensors. The simulator also inserts random noise into actions and sensory input. Agents belonging to the same team are only allowed

to communicate through the simulator which has bandwidth limitations, forcing the agents to learn independent policies. Thus the agents only have a limited world view, as well as noisy sensors and actions [17] [18].

The work of Stone, Sutton and Kuhlmann focuses on learning control policies for the keepers, while the takers use pre-specified and fixed policies [2].

4.2 Machine learning solution

The state was modeled from various distances and angles to other players. In the following an example for a state space with 3 keepers vs. 2 takers is presented (also see Figure 3). As will be seen the model can be easily modified for additional players. C will denote the center of the field, $K_1 - K_n$ are the keepers being ordered by their closeness to ball. The takers are $T_1 - T_m$, ordered by their distance to K_1. $dist(a, b)$ represents the distance between a and b. $ang(a, b, c)$ measures the angle between a and c with the vertex b [2].

- $dist(K_1, C); dist(K_2, C); dist(K_3, C)$
- $dist(T_1, C); dist(T_2, C)$
- $dist(K_1, K_2); dist(K_1, K_3)$
- $dist(K_1, T_1); dist(K_1, T_2)$
- $min(dist(K_2, T_1), dist(K_2, T_2))$
- $min(dist(K_3, T_1), dist(K_3, T_2))$
- $min(ang(K_2, K_1, T_1), ang(K_2, K_1, T_2))$
- $min(ang(K_3, K_1, T_1), ang(K_3, K_1, T_2))$

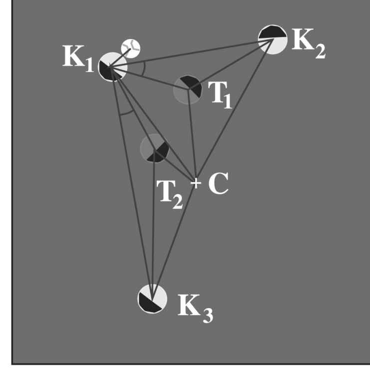

Figure 3: Example state representation. Various distances and angles are used [2]

The initial positions on the playing field were chosen so that all the takers start in one corner of the field (bottom left), whereas the remaining corners each contained one keeper, placing any remaining keeper in the center. The ball was always placed at the keeper in the top left corner.

The agents did not choose the primitive actions the simulator exposes but rather choose macro actions. For the keepers these were [2]:

- **Receive**: If a teammate has control over the ball or is nearer to the ball, the agent moves to a position where he can receive a pass without being interrupted by opponents. Otherwise the ball is intercepted from the opponents or collected from its resting position. This action is repeated every time step until the keepers have possession of the ball.
- **HoldBall**: Stay as far away from the opponents as possible while keeping possession of the ball.
- **PassKThenReceive**: Pass the ball to teammate k and then behave as in the Receive action.

The reward signal used, was the time steps following the current action with possession of the ball. This means that instead of minimizing the costs, the reward/time was maximized, which makes no difference for the algorithms used. Additional actions were not penalized, since the agents were supposed to hold the ball as long as possible [2].

This case study uses *Sarsa* as the machine learning algorithm and *Tile coding* for functional approximation of the state space. Since the Robocup simulator is used, it has the control over when actions must be taken or sensor input is delivered. This leads to the fact the algorithms have to be broken up in different phases. In this case there are three of them: RLstartEpisode, RLstep and RLendEpisode. RLstartEpisode is responsible for choosing the first action to perform when a new episode starts and initializes the current state by finding the activated tiles. RLstep is called everytime a new action has to be chosen, but only when the keepers have the ball, because otherwise they just chose the receive

action. Once a new action is chosen, the reward for the last action is calculated and the Q-values updated. Finally RLendEpisode is called whenever an episode ends and just calculates the reward of the final action that was taken. *Tile coding* was used with 32 tilings per state variable each tiling with an offset of $1/32$ of the tile width. Distances used a tile width of 3.0m and angles of $10°$. Full details along with pseudo-code of all the phases can be found in [2].

4.3 Results

The results of the learned keeper were evaluated against a random policy which chooses a keeper macro-action randomly, a policy which always holds the ball, as well as a hand-coded policy fully specified in [2]. The evaluation took place with 3 keepers vs. 2 takers in a 20×20 region. The takers were always using a hand-coded policy, which in the case of just two takers is identical to all keepers go to ball and try to intercept it. The starting policy for the learning process is a random policy.

The first results presented were being measured when giving the learners full world-view rather than limited and noisy sensors. The random keeper policy was able to hold the ball for 5.5sec on average, the always hold policy for 4.8sec and the hand-coded policy for 5.6sec. The learned policy is able to outperform all these benchmark policies significantly. After having played a few thousand episodes they reach their peak performance and are able to hold the ball for around 12sec (Figure 4), but after playing only a few hundred episodes they already outperform all the benchmark policies. These episode numbers mean that after a few hours of training the learned policies outperform the benchmark ones and after approximately 15 hours they reach their peak performance [2].

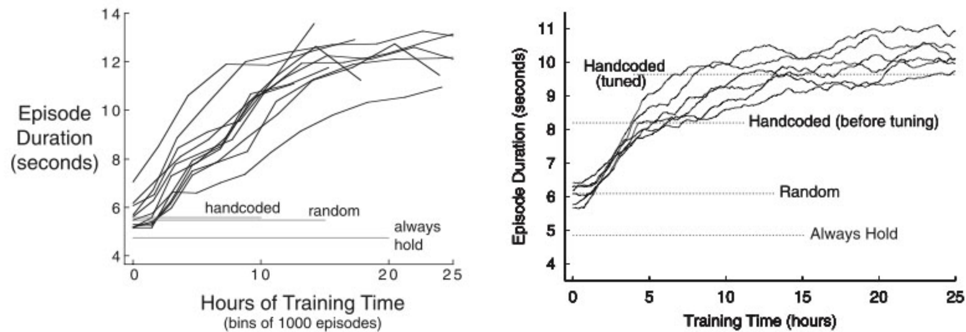

Figure 4: Results of keepaway training with multiple trials using full vision (left) and limited vision (right). The benchmark policies are the horizontal lines. [2]

When limiting the vision of the learners, the always hold policy can achieve episode lengths of 4.9sec, the random policy of 6.1sec and the hand-coded policy of 8.1sec on average. The learned policy is eventually able to outperform this by reaching episode lengths of 10 sec on average (Figure 4). This learning process took about 25 hours. It has to be noted that the hand-coded policy is able to outperform the learned policy when the field size is increased. This makes the problem of keeping the ball easier, as the keepers have more room to play the ball. In smaller field sizes the learned policy continues to outperform the hand-coded solution. Stone, Sutton and Kuhlmann argue that a possible explanation could be that larger field sizes have more intuitive solutions to the problem and can therefore be easily covered by hand-coded solutions [2].

Stone et al. successfully applied reinforcement learning to the robot soccer subdomain keepaway. Their learned policies where able to outperform other policies and still were learned in relatively few time. This paper focussed on the performance results of the learned policy, but Stone et al. [2] also reported results on a few other interesting topics regarding this learning process. Further studies could try to learn both the takers and keepers policy to learn better policies on both sides, challenging each other on the way.

5 Learning powerful kicks

In the following the case study *Learning powerful kicks* from Hausknecht and Stone will be described. A detailed report can be read in their paper [3].

5.1 Task description

Kicking is a crucial skill in robot soccer, as kicks are needed for passing to teammates as well as scoring. Typically this was hand-coded and manually tuned, but with the success of using machine learning for these tasks, as described in Section 2, Hausknecht and Stone [3] applied it to kicks.

To make the learning process as fast as possible, it was partially automated. The robot sat at the end of an inclined ramp and kicked the ball up the ramp. The inclination of the ramp was determined by trial and error, so that the robot could not kick the ball off the end of the ramp, but still being as flat as possible. Full autonomy of the learning process could not be achieved, because a human had to relay the distance of the kicks and reposition robot and/or ball. The reasons for this were that the kicks during the learning were violent and caused the robot to fall over and additionally that the ball would get stuck on the top of the ramp. With this setup Hausknecht and Stone were able to execute 500 kicks before the robot ran out of power, which is an average of one trial per seven or eight seconds [3].

Because of their varying usage a soccer player needs to be able to do several kicks of varying length and speed. Rather than learning multiple kicks, Hausknecht and Stone learned one kick and parameterized it, to vary the distance, which will be described in Results.

5.2 Machine learning solution

Machine learning always needs parameters it can change to optimize the task. The more parameters are provided the longer the learning time will be. If one uses too few parameters it can happen that the learning algorithm cannot effectively learn the task. Hausknecht and Stone used 10 parameters per pose in total. They reduced the totally available 18 joints by using the symmetry of the robot regarding the left and right front and back legs, as well as not using the two joints of the tail the robot had. This reduced the total 18 joints to only 10 per pose [3].

The tested machine learning algorithms used were *hill climbing* and *policy gradient*, which were run in succession. The random variations needed in the algorithms were in the range of 0 to one-tenth of the full range of the joint. The poker kick from *UT Austin Villa*, described below, was used as the initial policy for policy gradient, which was run for 65 iterations each generating and evaluating 10 policies and using a step-size of $\eta = 2.0$. The evaluation of the kick was done by letting each policy generate one kick up the ramp and then timing the return time to the robot as the reward signal, making higher times more powerful kicks. After these iterations the most powerful kick was chosen and hill climbing was used for 27 iterations each generating and evaluating 5 policies. In this phase two kicks were averaged to evaluate a policy. The numbers of iterations were found by running each algorithm until one battery charge was used [3].

The first phase of the process was used to move from the starting point in a generally better direction, as the policy gradient algorithm incorporates all the positive aspects of the generated policies into the next policy. In contrast Hill climbing in the second phase quickly moved to a specific policy, which was evaluated to be the best with higher accuracy since two kicks were used for evaluation.

After these two phases the learned kick was evaluated as described in Results below.

5.3 Results

The resulting kick was compared to the most powerful hand-coded kick (the *Power Kick* (PK)) of one of RoboCup's constant top competitors, *UT Austin Villa*, on flat ground. The comparison was done in 50 trials with 10 different robots. The learned kick (LK) was able to reach a distance of 373.66cm on average with a standard deviation of 105.51cm. PK only achieved 322.4cm on average with a standard deviation of 130.21cm (Figure 5). Thus the LK was significantly better than the PK, as the increase in distance was proven to be statistically significant [3]. Additionally the LK was able to always move the ball at least 200cm contrary to PK (Figure 5). Another important

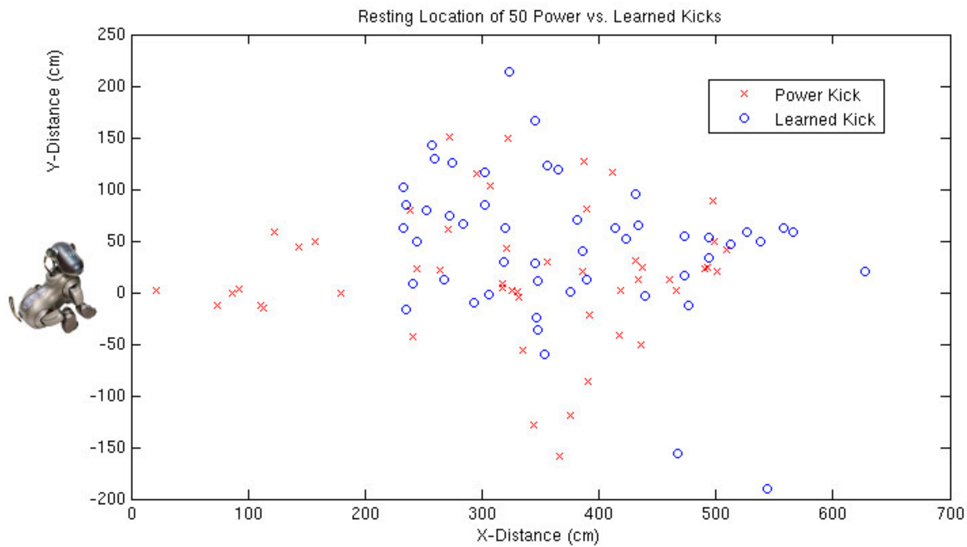

Figure 5: The robot kicks from origin in positive X-direction [3]

aspect of kicking aside from the distance is the accuracy of the kick, which was measured using the deviation from the X-axis per meter of kick distance. The PK was slightly more accurate achieving an average deviation of 15.82cm(standard deviation 14.88cm) per meter, contrary to the LK with 19.17cm (standard deviation 16.32cm). It is likely this is the result of not considering the accuracy of the kick in the learning process [3].

Another research point of Hausknecht and Stone was how much influence the starting point of the learning process has to the final result, since their starting point was an already tuned kick. They used a kick which was not able to move the ball forward as a starting point and after a few iterations the algorithms were able to find a kick which moved the ball forward. Ultimately the kick was optimized to reach the end of the ramp, which corresponds to a distance of approximately 200cm. But Hausknecht and Stone also admit that not every starting point will result in a working kick [3].

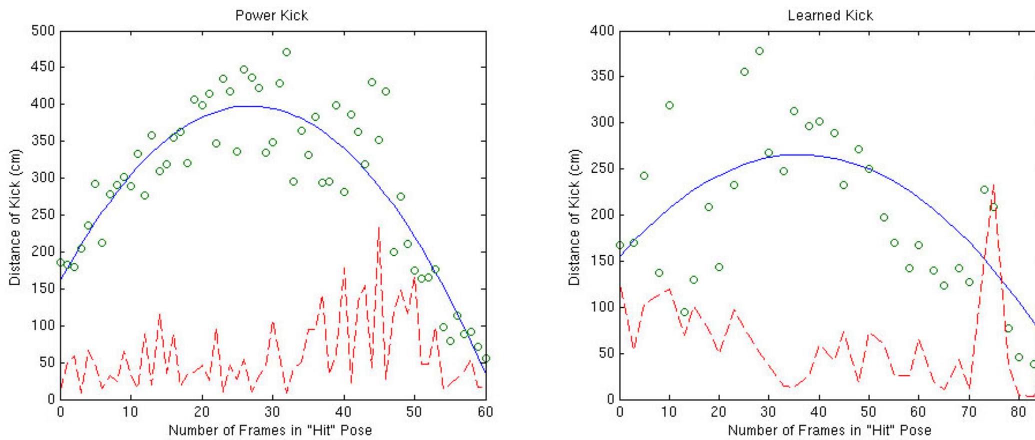

Figure 6: Quadratics (solid line) fit to kick distance data points (circles) and standard deviations of the kicks comprising each data point (dashed line) [3]

Since varying kicks are necessary to effectively play soccer and in order to not increase the learning time, by learning multiple kicks, the learned kick was parameterized: This was achieved by altering the transition time before the pose that makes contact with the ball, to allow the robot to execute variable distance kicks. After trying several different transition times, a quadratic function was used

to approximate the relation between transition time and distance, which allowed the learned kick to accurately kick within 45.5cm with a standard deviation of 39.3cm (1 - 2 robot lengths), evaluated by 20 kicks in the range of 60cm to 400cm (Figure 6) [3].

Hausknecht and Stone successfully showed that machine learning can be effectively used to learn robots to kick a ball and also showed how a parameterized kick can be created from any kick. To make the result and the learning process even more effective, it would be interesting to further automate the learning process, as well as adding accuracy feedback to the machine learning solution.

6 Learning aggressive defense behaviour

This case study will describe *Learning an aggressive defense behavior* from Riedmiller, Gabel, Hafner and Lange [4]. It will focus on simulating defense behaviour through the Soccer Server [18] rather than controlling real robots.

6.1 Task description

The task for the agent is to stop the adversary having the ball, ideally by stealing the ball to be able to launch a counter-attack. This is rather difficult since the defense behavior is highly dependent on the adversary and to much optimization could over-specialize the strategy and therefore perform poorly against some teams. Additionally failing to steal the ball in a duel can lead to a scoring opportunity by the opposing team. The case study focuses on the situations where one player has to interfere with an adversary having the ball to mitigate an attack.

6.2 Machine learning solution

The state representation for the machine learning algorithm is nine dimensional and contains the positions and velocities of both players involved as well as from the ball. Additionally information about where on the field the situation takes place is inserted, since an attack in front of the goal might be handled differently than one on the left wing of the field. This information is compressed into the mentioned 9 dimensions [4].

States can either be success states (S_1^+), semi-success (S_2^+), semi-failure (S_1^-), failure states (S_2^-) or intermediate states. Success states contain the states where the agent either has the ball in its kickable area or could execute a succeeding tackle with a high probability. If the opposing player simply kicks the ball away, e.g. because he considers the situation hopeless, the defense was effective, but the agent could not secure the ball, which is considered a semi-success. Failure states are the states where the opposing player has overrun the agent at least 7m or approached the goal such that a goal shot might be promising. Semi-failures are considered states where no clear winner could be distinguished because defense took more than 35 time steps and therefore was not effective in interfering with the adversary [4].

The agents were allowed to perform dash(x) and turn(y) commands, where $x \in [-100, 100]$ was discretized into 16 different actions and $y \in [-180°, 180°]$ was discretized into 60 different actions [4].

The machine learning algorithms used were *Fitted policy iteration* with a *Multiplayer perceptron* as the supervised learning module to approximate the cost-to-go function. Because a simulator is used, generation of experience as needed in every iteration of the policy iteration algorithm is not a problem. Additionally policy iteration needs a world model p, so the known model of the simulator was used as world model. This model does not include adversary behavior and is therefore only an approximation. The multilayer perceptron contains 9 input neurons, one hidden layer of 18 units and 1 output neuron. Initially the cost-to-go function J_0 is initialized with the perceptron having random input values. Every intermediate step was punished by costs of $c = 0.05$, otherwise [4]:

$$J(s) = \begin{cases} 0.0 & \text{for } s \in S^+ \\ 1.0 & \text{for } s \in S^- \end{cases}$$

The schema from which initial states where generated can be seen in Figure 7a and b.

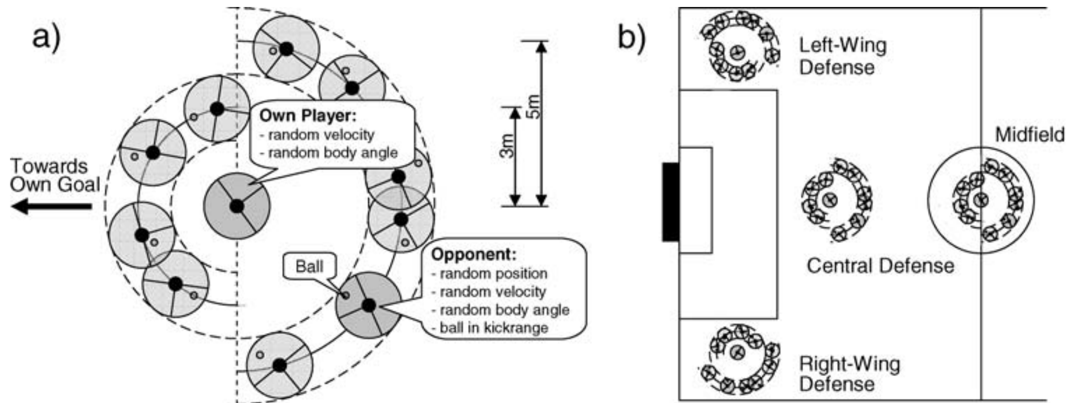

Figure 7: Defense behavior learning: Initial states schema [4]

6.3 Results

The learning process was eventually evaluated by measuring the number of successes, semi-success, semi-failures and failures when playing against team WrightEagle. This team was able to play very effectively against the hand-coded defense players of Riedmiller et al (2009) [4]. After about 700 training episodes, the defense player is able to succeed in capturing the ball (no semi-successes) with a probability of 80% [4].

After complete training the learned policy is able to successfully defend (success and semi-success) in all positions with a probability of 84% (hand-coded 55%) against team WrightEagle. Against other teams the policy is also able to reach such high successes in the range of 80% - 90% (hand-coded 40% - 60%) (average results in Figure 8). This a very significant improvement, so that the learned policy was deployed in the competition team of Riedmiller et al [4].

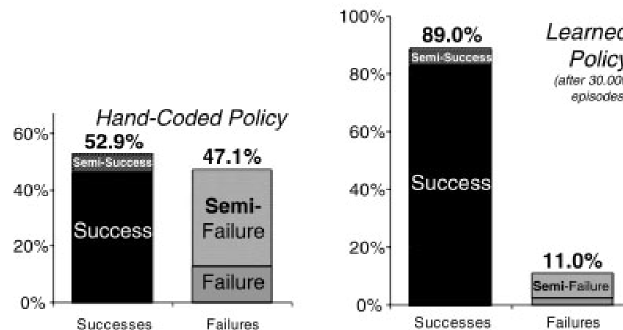

Figure 8: Resulting performance of hand-coded policy vs. learned policy [4]

Riedmiller et al. showed that machine learning can be used to extensively improve the defensive behavior of single agents. One next step could be to try to make this a team effort, meaning that a team of defense agents try to stop an adversary to further increase the success rate.

7 Learning motor speed control

This section will describe *Learning motor speed control* from Riedmiller, Gabel, Hafner and Lange [4]. Instead of high-level task such as kicking or defending, this case study focuses on accurate controlling of DC motors, to allow precise movements of robots. Additional more detailed information about the study can also be found in [19].

7.1 Task description

Reliable and accurate motor speed control is a crucial task in any scenario where robots are used. Without being able to accuratly control the motor high-level tasks cannot perform as good, since they would have to take inaccuracies of the motor controller into consideration. The task will be to learn a controller who can control three DC motors of an omnidirectional robot independently. Since three motors have to be controlled at once, the controller has to be able to handle a higher load than a normal controller responsible for only a single motor. The controller got the current state information from a motor and then responded with an action for the motor to bring it to the desired speed [4].

7.2 Machine learning solution

The state of a DC motor can be described using only two variables: the current motor speed $\dot{\omega}$ and the armature current I. Since the controller needs also to know the desired speed, the error to the desired motor speed will be used: $E := \dot{\omega_d} - \dot{\omega}$. Because motor control is an ongoing task, the costs to the desired states will be modeled such that the motor will be brought close to the target speed (with tolerance $\delta > 0$) rather fast and then will be held there [4]:

$$c(s, a, s') = \begin{cases} 0.0 & \text{if } |\dot{\omega_d} - \dot{\omega}| < \delta \\ 0.01 & \text{else} \end{cases}$$

To reduce the amounts of actions the controller can choose from, the actions did only output a few discrete values representing the voltage change to be applied, rather than allowing every value on the continuous scale to be an action. The available actions were $A = \{-0.3, -0.1, -0.01, 0.0, 0.01, 0.1, 0.3\}$ [4].

To train the controller the described Neural fitted Q-iteration (NFQ) was used alongside a Multiplayer perceptron with 5 input neurons, 2 hidden layers each containing 10 neurons and one output neuron. The inputs to the neural network are the current state $(I, U, E, \dot{\omega})$, as well as the action a, together denoting $Q(s, a)$, the output is the new approximated Q-value [4].

7.3 Results

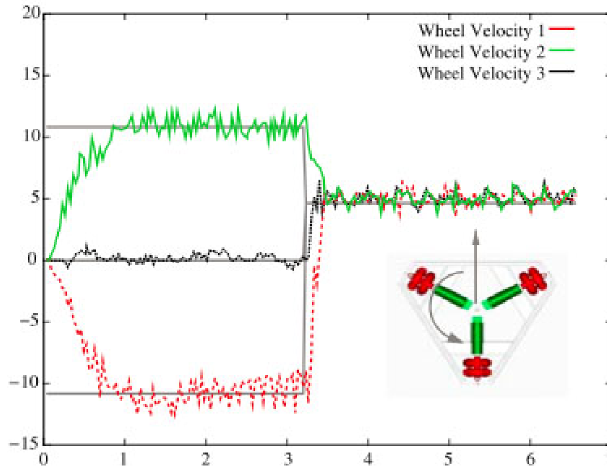

Figure 9: Final performance of the learned motor controller. Straight lines represent ideal motor control [4]

The learned controller was first trained using a single free-running motor, but although the controller performed good on the free-running motor, it failed effectively controlling the robot. The likely cause was that the controller never experienced noise or dynamically changing load situations. In a second learning procedure the real robot was used to sample experience by issuing random control commands to all three motors at once. After a little more than 4 minutes enough samples from all motors were collected, to start off-line learning [4].

The final performance of the controller was quite good, such that it was able to reach the right target speeds in under approximately 0.5sec on average, even under the noise present on the real robot (Figure 9). More results especially regarding the eventually unsatisfying free-running learning process can be found in [19].

Riedmiller et al. showed that machine learning can even be a valuable addition in low-level tasks such as motor control. Unfortunately no comparison with hand-coded motor controllers was described, which is a crucial comparison when deciding if machine learning provided additional value by outperforming previous solutions.

8 Learning to dribble

This case study will describe *Learning to dribble on a MidSize robot* by Riedmiller, Gabel, Hafner and Lange [4].

8.1 Task description

This case study focuses on dribbling with a real robot, that means keeping the ball in front of the robot, while turning to a specific target. This task is challenging as the rules for MidSize robots state that only one-third of the ball can be covered by something helping the robot to dribble. While turning, the robot has therefore take care so that the ball cannot roll away and always stays in its control [4].

8.2 Machine learning solution

States in this case study are encoded in a six dimensional vector containing: speed of the robot in relative x and y direction, rotation speed, x and y position of the ball relative to the robot and heading direction relative to the target. A state is regarded as a failure state if the absolute distance of the ball to the robot in x direction exceeds 50mm or exceeds 100mm on the y coordinate. A state is regarded successful whenever the absolute difference of heading and target angle is less than 5 degrees.

To reduce the number of possible actions, the rotation value of an action triple $(v_x^{target}, v_y^{target}, v_\theta^{target})$ was interpreted differently depending on the difference between the robots current angle and the target direction. If this difference was negative, the rotation value was interpreted negatively, therefore only positive values were allowed. The target speed v_x^{target} was interpreted similarly, but also allowed negative values. This lead to only five different possible actions, also illustrated in Figure 10:

1. $(2.0, 0.0, 2.0)$
2. $(2.5, 0.0, 1.5)$
3. $(2.5, 1.5, 1.5)$
4. $(3.0, 1.0, 1.0)$
5. $(3.0, -1.0, 1.0)$

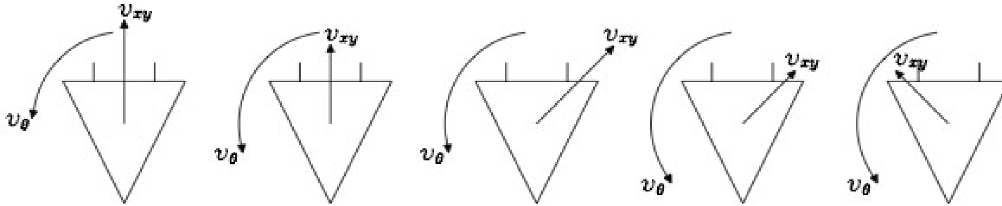

Figure 10: Available dribbling actions with different combinations of speed v_{xy} and rotation v_θ [4]

Every intermediate step while dribbling was punished with cost $c_I = 0.01$, while success states also costed $c_{S^+} = 0.01$, but terminated the sequence. Failure states were punished with $c_{S^-} = 1.0$.

Just as the previous section, Neural fitted Q-iteration (NFQ) along with a Multiplayer perceptron is used as a learning framework. The MLP had 9 input neurons (6 state, 3 action), 2 hidden layer each containing 20 neurons and 1 output neuron for the Q-value.

Training was done by first collection 12 trials of experience without retraining, to not have to intervene by resetting robot and ball to initial positions after each iteration and then learning with this experience.

8.3 Results

The final performance was evaluated by letting the learned dribbling policy compete against the previously used hand-coded version executing an U-turn. While the hand-coded policy drove a big arc to not loose the ball, the learned policy successfully turned in a significantly smaller way and was therefore also faster (Figure 11). These results were achieved with 1.5 hours of training of which about 30min were interaction time with the robot.

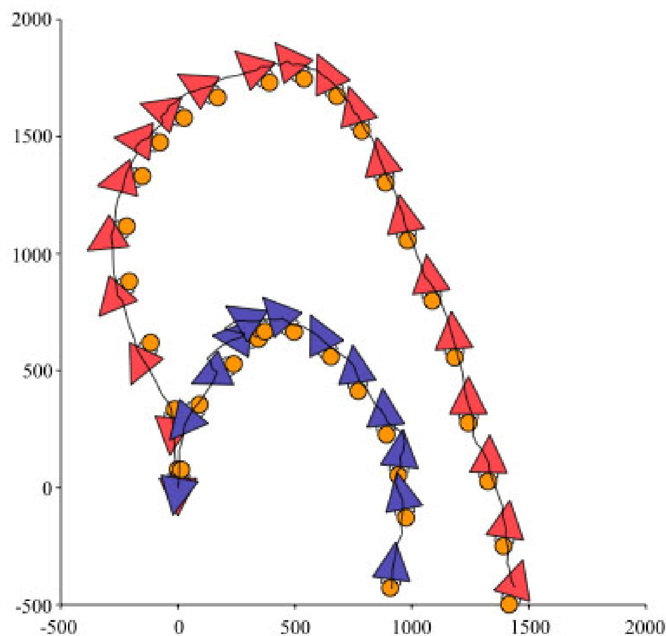

Figure 11: Dribbling evaluation by doing an U-turn, hand-coded policy in red, learned policy in blue [4]

Riedmiller et al. showed how to successfully apply machine learning to ball dribbling and were able to outperform their previous hand-coded solution. Further study might try to give the agent more movement freedom than the 5 basic acions used here, to boost the final performance further. Another idea might be to incorporate the actual time a specific action took, to learn the agent to make his move as fast as possible time-wise and not with the least actions possible, since certain direct turns might take longer than taking a little detour.

9 Discussion & Conclusion

This paper described five different case studies for applications of machine learning in the domain robot soccer. Every case study can be viewed as a successful case of applying machine learning, as most of the times previous hand-coded solutions where outperformed. Additionally the presented case studies where just a few picked ones under a lot more [8] [9] [10]. A lot of the teams in the RoboCup competitions use machine learning at least for a few of their tasks and are able to achieve quite good results with them, such as the team from Riedmiller et al [4]. This shows that

machine learning and specifically reinforcement learning is a viable addition to robot soccer, as all case studies used some form of reinforcement learning.

Reinforcement learning is clearly the way to go, as the formalization to MDPs fits most cases rather well and the results speak for themselves. Supervised learning is not really feasible all the time, since annotated data cannot be provided and unsupervised learning is likely to not reach the results of reinforcement learning, since all the feedback provided during the reinforcement learning process would have to be learned too.

Future work in the domain should be to try to improve even more tasks with reinforcement learning methods, but even more important might instead be to further automate learning processes incorporating real robots, because human interaction during the learning process slows the process significantly down and therefore limits possible results.

Additonally the presented case studies also leave room for further studies. In the keepaway subdomain training of the takers is an obvious task which might even improve the keepers, when learned simultaneously or agents could be given more action decisions and learn how to move and position themselves when they do not have the ball. A major disadvantage of the learned kicks is that their accuracy is rather poor since these measurements were not included in the feedback during the learning process, which could be a logical next step, that can also lead to a better parameterized kick. The defense task already was able to achieve a major improvement in contrast to previous manual approaches. Further improvement might be achieved when letting multiple defense agents act as a team against an opponent attacking with the ball. Dribbling could be further studied by allowing more fine-grained actions to give the learning process more freedom to find good policies, but also incorporating time as a reward signal, since number of actions and the actual passed time can be quite different in situations, where actions might take longer.

To conclude one can say reinforcement learning is a powerful tool to improve robot movement beyond human-optimized policies. The results of the presented case studies clearly show, that it is worth trying to incorporate reinforcement learning in various tasks occurring in robot soccer.

References

[1] Peter Stone and Herbert A. Simon. Layered learning in multi-agent systems. Technical report, 1998.

[2] Peter Stone, Richard S. Sutton, and Gregory Kuhlmann. Reinforcement learning for robocup soccer keepaway. *Adaptive Behavior*, 13(3):165–188, 2005.

[3] Matthew Hausknecht and Peter Stone. Learning powerful kicks on the aibo ers-7: The quest for a striker. In Javier Ruiz-del Solar, Eric Chown, and Paul Plöger, editors, *RoboCup 2010: Robot Soccer World Cup XIV*, volume 6556 of *Lecture Notes in Computer Science*, pages 254–265. Springer Berlin / Heidelberg, 2011. 10.1007/978-3-642-20217-9_22.

[4] Martin Riedmiller, Thomas Gabel, Roland Hafner, and Sascha Lange. Reinforcement learning for robot soccer. *Autonomous Robots*, 27:55–73, 2009. 10.1007/s10514-009-9120-4.

[5] Gerald Tesauro. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Comput.*, 6(2):215–219, March 1994.

[6] J.A. Bagnell and J.G. Schneider. Autonomous helicopter control using reinforcement learning policy search methods. In *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, volume 2, pages 1615 – 1620 vol.2, 2001.

[7] Robert H. Crites and Andrew G. Barto. Elevator group control using multiple reinforcement learning agents. *Machine Learning*, 33:235–262, 1998. 10.1023/A:1007518724497.

[8] Peggy Fidelman and Peter Stone. The chin pinch: A case study in skill learning on a legged robot. In Gerhard Lakemeyer, Elizabeth Sklar, Domenico Sorrenti, and Tomoichi Takahashi, editors, *RoboCup 2006: Robot Soccer World Cup X*, volume 4434 of *Lecture Notes in Computer Science*, pages 59–71. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-74024-7_6.

[9] Hayato Kobayashi, Tsugutoyo Osaki, Eric Williams, Akira Ishino, and Ayumi Shinohara. Autonomous learning of ball trapping in the four-legged robot league. In Gerhard Lakemeyer, Elizabeth Sklar, Domenico Sorrenti, and Tomoichi Takahashi, editors, *RoboCup 2006: Robot*

Soccer World Cup X, volume 4434 of *Lecture Notes in Computer Science*, pages 86–97. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-74024-7_8.

[10] Michael J. Quinlan, Stephan K. Chalup, and Richard H. Middleton. Techniques for improving vision and locomotion on the sony aibo robot. 2003.

[11] G. A. Rummery and M. Niranjan. On-line q-learning using connectionist systems. 1994.

[12] M.W Gardner and S.R Dorling. Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences. *Atmospheric Environment*, 32(14–15):2627 – 2636, 1998.

[13] P.D. Wasserman and T. Schwartz. Neural networks. ii. what are they and why is everybody so interested in them now? *IEEE Expert*, 3(1):10 –15, spring 1988.

[14] Nate Kohl and Peter Stone. Machine learning for fast quadrupedal locomotion. In *Proceedings of the 19th national conference on Artifical intelligence*, AAAI'04, pages 611–616. AAAI Press, 2004.

[15] Jan Peters and Stefan Schaal. Policy gradient methods for robotics. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, pages 2219 –2225, oct. 2006.

[16] Alexander Sherstov and Peter Stone. Function approximation via tile coding: Automating parameter choice. In Jean-Daniel Zucker and Lorenza Saitta, editors, *Abstraction, Reformulation and Approximation*, volume 3607 of *Lecture Notes in Computer Science*, pages 901–901. Springer Berlin / Heidelberg, 2005. 10.1007/11527862_14.

[17] M. Chen, E. Foroughi, F. Heintz, S. Kapetanakis, K. Kostiadis, J. Kummeneje, I. Noda, O. Obst, P. Riley, T. Steffens, Y. Wang, and X. Yin. Users manual: Robocup soccer server manual for soccer server version 7.07 and later. Available at http://sourceforge.net/projects/sserver/.

[18] Itsuki Noda, Hitoshi Matsubara, Kazuo Hiraki, and Ian Frank. Soccer server: A tool for research on multiagent systems. *Applied Artificial Intelligence*, 12(2-3):233–250, 1998.

[19] R. Hafner and M. Riedmiller. Neural reinforcement learning controllers for a real robot application. In *Robotics and Automation, 2007 IEEE International Conference on*, pages 2098 –2103, april 2007.