MODEL-BASED REINFORCEMENT LEARNING FOR EVOLVING SOCCER STRATEGIES

M. A. Wiering, **R. P. Sałustowicz**, and **J. Schmidhuber**
University of Utrecht, The Netherlands, marco@cs.uu.nl,
IDSIA, Lugano, Switzerland, rafal@idsia.ch,
IDSIA, Lugano, Switzerland, juergen@idsia.ch

We use reinforcement learning (RL) to evolve soccer team strategies. RL may profit significantly from world models (WMs). In high-dimensional, continuous input spaces, however, learning accurate WMs is intractable. In this chapter, we show that incomplete WMs can help to quickly find good policies. Our approach is based on a novel combination of CMACs and prioritized sweeping. Variants thereof outperform other algorithms used in previous work.

1 Introduction

Game playing programs have been a major focus of artificial intelligence (AI) research. How to represent and evaluate positions? How to use planning for exploiting evaluations to select the optimal next move (action)? Berliner's non-adaptive backgammon program (1977) had a prewired evaluation function (EF), costed many man-years of programming effort, but achieved only mediocre level of play. Tesauro's TD-Gammon program (1992), however, used reinforcement learning (RL) to *learn* the backgammon EF by playing against itself. After only three months of training on a RS6000, TD-Gammon played at human expert level. Back in 1959 Samuel already constructed a RL program which learned an EF for the game of checkers, resulting in the first game playing program that defeated its own programmer. Related efforts are described by Baxter (chess, 1997) , Thrun (chess, 1995) and Schraudolph (Go, 1994).

Soccer. We apply RL to a game quite different from board games: soccer. It involves multiple interacting agents and ambiguous inputs. We are partly motivated by the popularity of the International Soccer Robocup.

Most early research efforts in the field concentrated on implementing detailed behaviors exploiting the official Robocup soccer simulator. But recently machine learning (ML) and RL in particular have been used to improve soccer teams [31] — mostly to improve cooperation between players and to construct high-level strategies. The Robocup simulator, however, is too complex to evaluate and compare different RL methods for soccer teams learning from scratch, without prewired tactics and behaviors. Therefore we built our own simulator, which is simpler, faster and easier to comprehend.

Learning to play soccer. Our goal is to build teams of autonomous agents that learn to play soccer from very sparse reinforcement signals: only scoring a goal yields reward for the successful team. Team members try to maximize reward by improving their adaptive decision policy mapping (virtual) sensory inputs to actions. In principle there are at least two types of learning algorithms applicable to such problems: reinforcement learning (RL), e.g., [24], [32], [38], and [34], and evolutionary approaches, e.g., [11], [18], [9], and [20]. Here we describe a novel RL method and compare its results to those obtained by previous RL methods and an evolutionary approach.

Most existing RL algorithms are based on function approximators (FAs) learning value functions (VFs) that map states or state/action pairs to the expected outcome (reward) of a particular game [7], and [38]. In realistic, partially observable, multi-agent environments, learning value functions is hard though. This makes evolutionary methods a promising alternative. For instance, in previous work on learning soccer strategies [23] we found that *Probabilistic Incremental Program Evolution* (PIPE) [20], a novel evolutionary approach to searching program space, outperforms Q(λ) [17], [38], and [42] combined with FAs based on linear neural networks [22] or neural gas [21].

We identified several reasons for PIPE's superiority: (1) In complex environments such as ours RL methods tend to be brittle — once discovered, good policies do not stabilize but tend to get destroyed by subsequent "unlucky" experiences. PIPE is less affected by this problem because good policies have a large probability of surviving. (2) PIPE learns faster by isolating important features in the sensory input, combining them in programs of initially low algorithmic complexity, and subsequently refining the programs. This motivates our present approach: VF-based RL should also be able to (a) stabilize or improve fine policies (as opposed to unlearning them), (b) profit from the existence of low-complexity solutions, and (c) use incremental search to find more complex solutions where simple ones do not work.

Incomplete world models. *Direct* RL methods [7], and [38] use temporal differences (TD) [32] for training FAs to approximate the VF from simulated sequences of states (positions) and actions (moves) during a game. *Indirect* RL, however, learns a world model (WM) [15], and [40] estimating the reward function and the transition probabilities between states, then uses dynamic programming (DP) [5] or similar, faster algorithms such as prioritized sweeping (PS — which we will use in our experiments) [15] for computing the VF. This can significantly improve learning performance in discrete state/action spaces [15]. In case of continuous spaces, WMs are most effectively combined with *local* FAs transforming the input space into a set of discrete regions (core positions) and then learning the VF. Similarly, continuous action spaces can be transformed in a set of discrete actions. Previous work has already demonstrated the effectiveness of learning discrete world models for robotic localization and navigation tasks, e.g., [37]. Learning accurate WMs in high-dimensional, continuous, and partially observable environments is hard. However, this motivates our novel approach of learning useful but incomplete models instead.

CMAC models. We will present a novel combination of CMACs and world models. CMACs [1] use *filters* mapping sensor-based inputs to a set of activated cells. Each filter partitions the input space into subsections in a prewired way such that each (possibly multi-dimensional) subsection is represented by exactly one discrete cell of the filter. For example, a filter might consist of a finite number of cells representing an infinite set of colors represented by cubes with 3 dimensions red, blue and green, and activate the cell which encloses the current color input component. For game playing, a filter may represent different but similar positions and the activated cell may represent the presence of a particular position.

In a RL context each cell has a Q-value for each action. The Q-values of currently active cells are averaged to compute the overall Q-values required for action selection. Previous work already combined CMACs with Q-learning [38] and Q(λ) methods [33], and [25]. Here we combine CMACs with WMs by learning an independent model for each filter. These models are then exploited by a version of prioritized sweeping (PS) [15], and [41] for computing the Q-functions. Later we will find that CMAC models can quickly learn to play a good soccer game and surpass the performance of PIPE and an approach combining CMACs and Q(λ).

Outline. Section 2 describes our soccer environment. Section 3 presents CMACs and describes how they can be combined with model-based learning. Section 4 describes experimental results. Section 5 concludes the chapter.

2 The Soccer Simulator

Our soccer simulator [23] runs discrete-time simulations involving two teams consisting of either 1 or 3 players per team. A game lasts from time $t = 0$ to time $t_{end} = 5000$. The field is represented by a two-dimensional continuous Cartesian coordinate system. As in indoor soccer the field is surrounded by impassable walls except for the two goals centered in the east and west walls. There are fixed initial positions for all players and the ball (see Figure 1).

Players/Ball. Each player and the ball are represented by a solid circle and a variable real-valued position and orientation. A player whose circle intersects the ball picks it up and then owns it. The ball owner can move or shoot the ball. A shot is in the direction of the player's orientation. When shot, the ball's initial speed is 0.12 units per time step. Each following time step the ball slows down due to friction by 0.005 units per time step (unless it is picked up by a player) - the ball can travel freely at most 1.5 units. At each discrete time step each player selects one of the following actions:

- *go_forward*: move 0.025 units in current direction.
- *turn_to_ball*: point player's orientation towards ball.

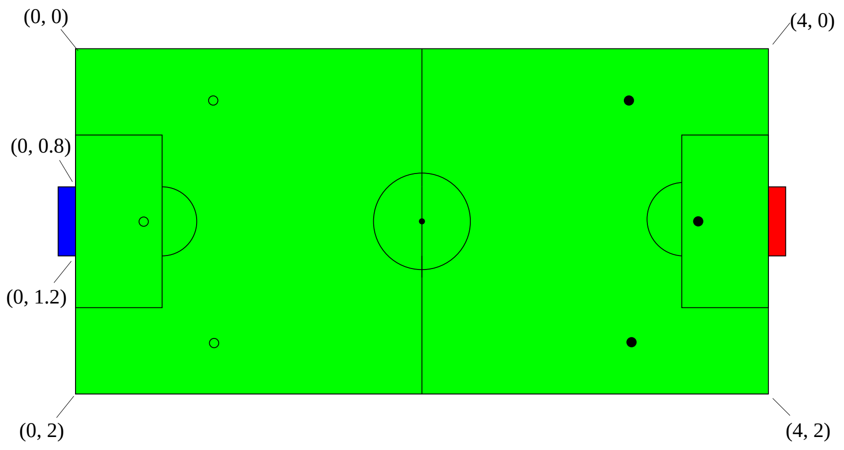

Figure 1. Players and ball (center) in initial positions. Players of a 1 player team are those furthest in the back.

- *turn_to_goal*: point player's orientation towards opponent's goal.
- *shoot*: if the player owns the ball then change player's orientation by a random angle from the interval $[-5°, 5°]$ (to allow for noisy shots), and shoot ball in the corresponding direction.

A player that makes a step forward such that its circle intersects another player's circle bounces back to its original position. If one of them owns the ball prior to collision then it will lose it to the collision partner.

Action framework. During each time step all players execute one action each, in randomly chosen order. Then the ball moves according to its current speed and direction. If a team scores or $t = t_{end}$ then all players and ball will be reset to their initial positions.

Sensory input. At any given time a player's input vector $\vec{x}$ consists of 16 (1 player) or 24 (3 players) components:

- Three Boolean input components that tell whether the player/a team member/opponent team owns the ball.

- Polar coordinates (distance, angle) of both goals and the ball with respect to the player's orientation and position.
- Polar coordinates of both goals relative to the ball's orientation and position.
- Ball speed.
- Polar coordinates of all other players w.r.t. the player ordered by (a) teams and (b) distances to the player.

Policy-sharing. All players share the same Q-functions or PIPE-programs. Still their behaviors differ due to different, situation-specific inputs. Policy-sharing has the advantage of greatly reducing the number of adaptive free parameters, which tends to reduce the number of required training examples (learning time) and increase generalization performance, e.g., [16]. A potential disadvantage of policy sharing, however, is that different players cannot develop truly different strategies to be combined in fruitful ways.

3 CMAC Models

CMACs [1] use multiple, *a priori* designed filters to quantize the input space. Each filter consists of several cells with associated Q-values. Applying the filters to the current input yields a set of activated cells (a discrete distributed representation of the input). Their Q-values are averaged to compute the overall Q-value.

Filter design. In principle filters may yield arbitrary divisions of the input space, such as hypercubes. To avoid the curse of dimensionality one may use hashing to group a random set of inputs into an equivalence class, or use hyperslices omitting certain dimensions in particular filters [33]. Although hashing techniques may help to overcome storage problems, we do not believe that random grouping is the best we can do. Since our soccer simulation involves a fair number of input dimensions (16 or 24), we use hyperslices to reduce the number of adjustable parameters. Our filters divide the state-space by splitting it along single input dimensions into a fixed number of cells — input components are treated in a mutually independent way. Of course we could also construct filters combining different input features, and this is what we would have to do for representing the high amount of context-dependency in a game such as chess. Finally, we apply multiple filters to the same input component to allow for smoother generalization.

Partitioning the input space. We use two filters for each input component, both splitting the same component. Input components representing Boolean values, distances (or speeds), and angles, are split in various ways (see Figure 2): (1) Filters associated with a *Boolean* input component just return its value. (2) *Distance* or *ball-speed* input components are rescaled to values between 0 and 1. Then the filters partition the components into n_c or n_c+1 quanta. (3) *Angle* input components are partitioned in n_c equal quanta in a circular (and thus natural) way — one filter groups the angles $359°$ and $0°$ to the same cell, the other separates them by a cell boundary.

Selecting an action. Applying all filters on a player's current input vector at time t returns the active cells $\{f_1^t, \dots, f_z^t\}$, where z is the number of filters. The Q-value of selecting action a given input $\vec{x}$ is calculated by averaging all Q-values of the active cells:

$$Q(\vec{x}, a) = \sum_{k=1}^{z} Q_k(f_k^t, a)/z,$$

where Q_k is the Q-function of filter k. Instead of just averaging the Q-values of all filters, we might also weigh them according to particular active strategies or the predictive ability of each filter, or alternatively we may use a voting scheme in which each filter votes for a specific action.

After computing the Q-values of all actions we select an action according to the Max-random exploration rule: select the action with maximal Q-value with probability P_{max}, and a uniformly random action otherwise.

Learning with WMs. Learning accurate models for high-dimensional input spaces is hard. Usually there are so many possible successor states that storing all of them for each different input would be infeasible and updates would cost a lot of time. Instead we introduce a novel combination of model-based RL and CMACs. We use a set of independent

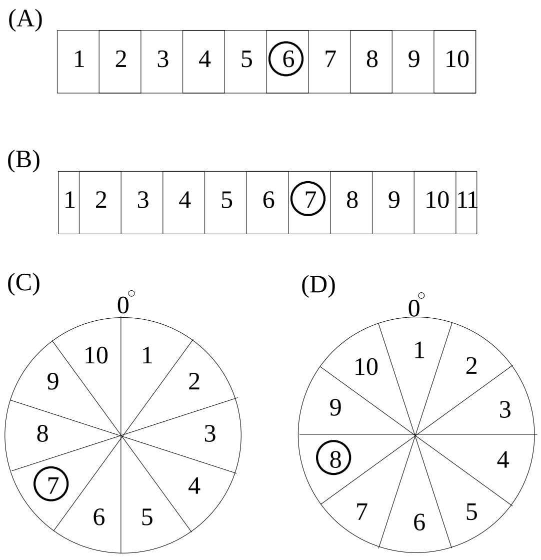

Figure 2. We use two filters for each input component, resulting in a total of 32 (1 player) or 48 (3 players) filters. Filters of a Boolean input component just return the Boolean value as cell number. The figure (activated cells are marked) illustrates decompositions of (A) a continuous distance input component into 10 discrete cells, (B) the same component into 11 cells, (C) a continuous angle component into 10 cells, (D) the same component into 10 different cells.

models to estimate the dynamics of each filter. To estimate the transition model for filter k, we count the transitions from activated cell f_k^t to activated cell f_k^{t+1} at the next time-step, given the selected action. These counters are used to estimate the transition probabilities $P_k(c_j|c_i, a) = P(f_k^{t+1} = c_j|f_k^t = c_i, a)$, where c_j and c_i are cells, and a is an action. For each transition we also compute the average reward $R_k(c_i, a, c_j)$ by summing the immediate reinforcements, given that we make a step from active cell c_i to cell c_j by selecting action a.

Prioritized sweeping (PS). We could immediately apply dynamic programming (DP) [5] to the estimated models. Online learning with DP, however, is computationally expensive. But fortunately there are more efficient update management methods. We will use a method similar to prioritized sweeping (PS) [15] which may be the most efficient available update mechanism. PS updates the Q-value of the filter/cell/action triple with the largest update size before updating others. Each update is made via the usual Bellman backup [5]:

$$Q_k(c_i, a) \leftarrow \sum_j P_k(c_j|c_i, a)(\gamma V_k(c_j) + R_k(c_i, a, c_j))$$

where $V_k(c_i) = \max_a Q_k(c_i, a)$ and $\gamma \in [0, 1]$ is the discount factor. After each player action we update all filter models and use PS to compute the new Q-functions. PS uses a parameter to set the maximum number of updates per time step and a cutoff parameter ϵ preventing tiny updates. Note that PS may use different numbers of updates for different filters, since some filters tend to make larger updates than others and the total number of updates per time step is limited. The complete PS algorithm is given in Appendix A.

Non-pessimistic value functions. Policy sharing requires the fusion of experimental data from different players into a single representation. This data, however, is generated by different player histories. In fact, certain experiences of certain players will probably never occur to others — there is no obvious and straightforward way of data fusion. For instance, the unlucky experience of one particular player may cause the VF approximation to assign low values to certain actions for all players. After having identified this problem, we tried a heuristic solution to overcome this weakness. We compute *non-pessimistic* value functions: we decrease the probability of the worst transition from each cell/action and renormalize the other probabilities. Then we apply PS to the adjusted probabilities (details of the algorithm are given in Appendix B). The effect is that only frequently occurring bad experiences have high impact on the Q-function. Experiments showed small but significant improvements over the basic algorithm. The method is quite similar to Model-Based Interval-Estimation [41], an exploration algorithm extending Interval Estimation [12] by computing optimistic value functions for action selection.

Multiple restarts. The method sometimes may get stuck with continually losing policies which hardly ever score and fail to prevent (many) opponent goals (also observed with our previous simulations based on linear networks and neural gas). We could not overcome this problem by adding standard exploration techniques (evaluating alternative actions of losing policies is hard, since the perturbed policy will usually still lead to negative rewards). Instead we reset Q-functions and WMs once the team has not scored for 5 successive games but the opponent scored during the most recent game (we check these conditions every 5 games). After each restart, the team will gather different experiences affecting policy quality. We found that multiple restarts can significantly increase the probability of finding good policies.

We use $P_{max} = 1.0$ in the Max-random exploration rule, since that worked best. The reason multiple restarts works better without exploration is that it makes the detection of losing policies easier. Hopeless greedy policies will loose 0-something, whereas with exploration our agents may still score although they remain unable to improve their policy from the generated experiences. Thus, using greedy policies we may use a simpler rule for restarting.

Learning with Q(λ). Possibly the most widely used RL algorithm is Q-learning [38], which tries out sequences of actions through state/action space according to its policy and uses environmental rewards to estimate the expected long-term reward for executing specific actions in particular states. Q-learning repeatedly performs a one-step lookahead backup, meaning that the Q-value of the current state/action pair (SAP) becomes more like the immediately received reward plus the estimated value of the next state.

Q(λ)-learning [38], [17], and [42] combines TD(λ) methods [32] with Q-learning to propagate state/action updates back in time such that multiple SAPs which have occurred in the past are updated based on a single current experience. Q(λ)-learning has outperformed Q-learning in a number of experiments [14], [19], and [42]. For purposes of comparison we also use online Q(λ)-learning for training the CMACs to play soccer. The details of the algorithm are given in Appendix C.

PIPE. The other competitor is *Probabilistic Incremental Program Evolution* (PIPE) [20]. PIPE is a novel technique for automatic program synthesis. It combines probability vector coding of program instructions [26], [27], and [28], Population-Based Incremental Learning [2], and tree-coded programs like those used in some variants of Genetic Programming (GP) [9], [10], and [13]. PIPE iteratively generates successive populations of functional programs according to an adaptive probability distribution over all possible programs. In each iteration it lets all programs play one soccer game; then the best program is used to refine the distribution. Thus PIPE stochastically generates better and better programs. All details can be found in [23].

4 Experiments

We compare the CMAC model to CMAC-Q(λ) and PIPE [20], which outperformed Q(λ)-learning combined with various FAs in previous comparisons [21], and [23].

Task. We train and test the learners against handmade programs of different strengths. The opponent programs are mixtures of a program which randomly executes actions (random program) and a (good) program which moves players towards the ball as long as they do not own it, and shoots it straight at the opponent's goal otherwise. Our five opponent programs, called *Opponent(P_r)*, use the random program to select an action with probability $P_r \in \{0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1\}$, respectively, and the good program otherwise.

CMAC model set-up. We play a total of 200 games. After every 10 games we test current performance by playing 20 test games against the opponent and summing the score results. The reward is +1 if the team scores and -1 if the opponent scores. The discount factor is set to 0.98. After a coarse search through parameter space we chose the following parameters: 2 filters per input component (total of 32 or 48 filters) number of cells $n_c = 20$ (21 for the second filters of distance/speed input components). Q-values are initially zero. PS uses $\epsilon = 0.01$ and a maximum of 1000 updates per time step. We only compute non-pessimistic value functions for the 3-player teams for which we use z_α = 1.96.

CMAC Q(λ) set-up. We play a total of 200 games. After every 20 games we test current performance of the policy (during tests we continue selecting actions according to the current exploration scheme) by playing 20 test games against the opponent and summing the score results. The reward is +1 if the team scores and -1 if the opponent scores. The discount factor is set to 0.98. We conducted a coarse search through parameter space to select learning parameters. We use online Q(λ) with replacing traces [30] and $\lambda = 0.8$ for the 1-player case, and $\lambda = 0.5$ for the 3-player case. The initial learning rate is set to $\alpha_c = 1.0$, the learning rate decay parameter β (see Appendix C) is set to 0.3.

We use Max-random exploration with P_{max} linearly increased from 0.7 in the beginning of the simulation to 1.0 at the end. As for CMAC-models we use two filters per input component (total of 32 or 48 filters). The number of cells is set to $n_c = 10$ (11 for the second filters of distance/speed input components). All Q-values are initially zero. In general, learning performance does not very sensitively depend on the used parameters. E.g., using n_c = 20 results in only slightly worse performance. Small values for λ (< 0.3) do make things worse though.

PIPE set-up. For PIPE we play a total of 1000 games. After every 50 games we test performance of the best program found during the most recent generation. Parameters for all PIPE runs are the same as in the previous experiments [23].

Results : 1-Player case. We plot number of points (2 for scoring more goals than the opponent during the 20 test games, 1 for a tie, and 0 for scoring less) against number of training games in Figure 3.

We observe that on average our CMAC model wins against almost all training programs. Only against the best 1-player team ($P_r = 0$) it wins as often as it loses, and often plays ties (it finds a blocking strategy leading to a 0-0 result). Against the worst two teams, CMAC model always finds winning strategies.

CMAC-Q(λ) finds programs that on average win against the random team, although they do not always win. It learns to play about as well as the 75% random and 50% random teams. CMAC-Q(λ) is no match against the best opponent, and although it seems that performance jumps up at the end of the trial, longer trials do not lead to better performances.

PIPE is able to find programs beating the random team and quite often discovers programs that win against 75% random teams. It encounters great difficulties in learning good strategies against the better teams, though: although PIPE may execute more games (1000 vs. 200), the probability of generating programs that perform well against the good opponents is very small.

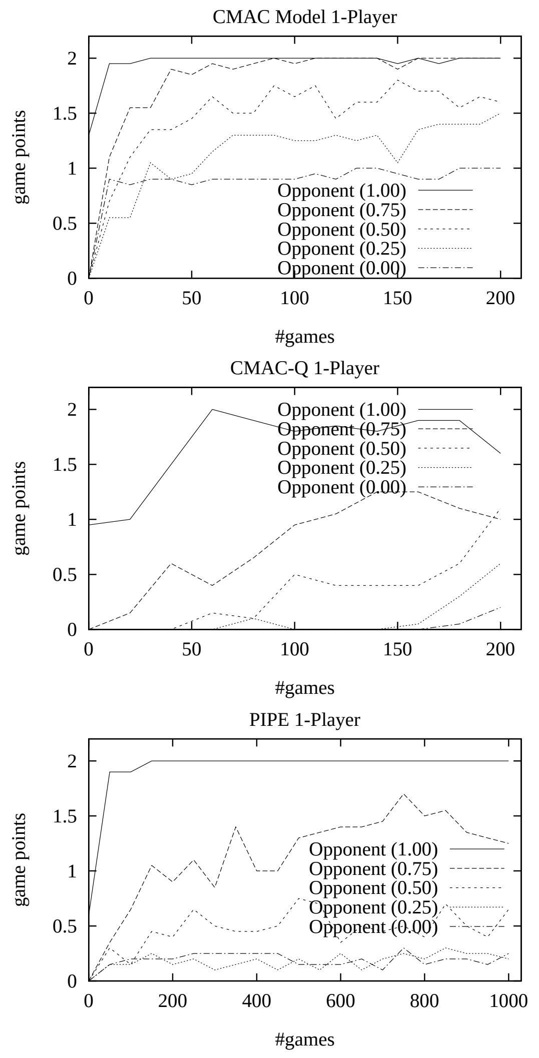

Figure 3. Number of points (means of 20 simulations) during test phases for teams consisting of 1 player. Note the varying x-axis scalings.

It tends to learn from the best of the losing programs. This in turn does not greatly facilitate the discovery of winning programs.

Results : 3-Players case. We plot number of points (2 for scoring more goals than the opponent during the 20 testgames) against number of training games in Figure 4.

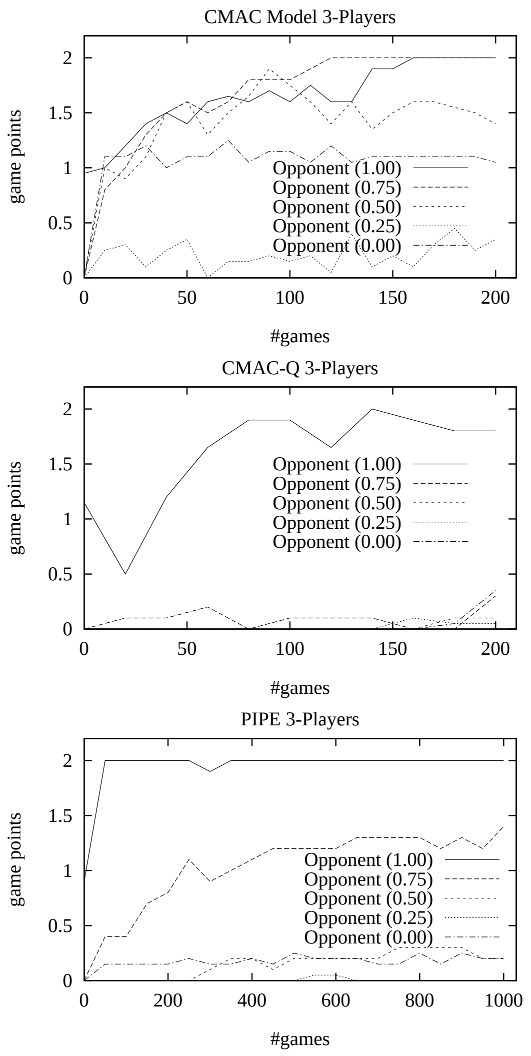

Figure 4. Number of points (means of 20 simulations) during test phases for teams consisting of 3 players. Note the varying x-axis scalings.

Again, CMAC model always learns winning strategies against the worst 2 opponents. It loses on average against the best 3-player team (with $P_r = 0.25$) though. Note that this strategy mixture works better than always using the deterministic program ($P_r = 0$) against which CMAC model plays ties or even wins. In fact, the deterministic program tends to clutter agents such that they obstruct each other. The deterministic opponent's behavior also is easier to model. All of this makes the stochastic version a more difficult opponent.

CMAC-Q is clearly worse than CMAC model — it learns to win only against the worst opponent.

PIPE performs well only against random and 75% random opponents. For the better opponents it runs into the same problems as mentioned above.

Score differences. We show maximal obtained score differences in Table 1 (1 player) and Table 2 (3 players). Although PIPE performs better against the weakest opponent than CMAC-models or CMAC-Q, PIPE often cannot score against strong opponents. CMAC-models, however, do score against the good opponents, and are able to find (at least once) winning policies against all opponents.

We should keep in mind that score differences may have a large variance. For instance, in some experiments with the 1-player CMAC model, opponent (0.0) may continuously win 760-0 in test matches. This extreme score difference is caused by resetting the (losing) CMAC model just before testing. PIPE has a small advantage here, since it uses the best program of the last generation for testing and thus almost lets vanish the probability of testing a really bad policy.

Discussion. Despite treating all components independently the CMAC model is able to learn good reactive soccer strategies preferring actions that activate those cells of a filter which promise highest average reward. The use of a model often quickly stabilizes good strategies: given sufficient experiences (5-20 learning games), the policy will hardly change anymore. The reason is that deterministic policies generate similar experiences.

Table 1. Best average score differences for different learning methods against 1-player opponents of varying strengths. * = Although CMAC-models were sometimes able to score 7 goals, they also sometimes lost 0-760.

Learning Alg.	1.0	0.75	0.5	0.25	0.0
CMAC model	85-2	68-3	27-1	6-15	1-146*
CMAC-Q	92-6	52-23	10-7	1-4	0-13
PIPE	225-18	127-35	19-13	0-3	0-10

Table 2. Best average score differences for different learning methods against 3-player opponents of varying strengths.

Learning Alg.	1.0	0.75	0.5	0.25	0.0
CMAC model	161-31	236-100	84-70	6-20	0.3-0
CMAC-Q	111-26	36-73	13-58	3-23	0-24
PIPE	297-18	163-64	30-31	0-11	0-21

Early experiences with a random initial policy may greatly impact the final policy's quality. They may result in continually losing policies (especially against better opponents) that are unable to improve, due to the near-impossibility of learning from bad experiences. For such reasons we performed multiple restarts (1 up to more than 10). Since tested strategies often remain either winners or losers, the step-wise improvements shown in the learning curves are mainly due to multiple restarts. The ups and downs in the learning curves are caused by unstable policies with unstable score results.

CMAC model tends to be quite robust under variations of filter design (e.g., combining multiple input components) and number of cells. Conducting additional experiments with filters combining distance and angle input components, or 10/11 instead of 20/21 cells per filter, we obtained similar levels of performance.

Multiple restarts helped CMAC model to avoid getting stuck with losing policies. When we tried CMAC-Q with multiple restarts and without exploration, it often found blocking strategies (leading to 0-0 results) against all 1-player opponents, but did not learn to win. Learning blocking strategies against multi-agent teams, however, is much harder.

All methods perform better in the single agent case. This can probably be explained by the fact that the multi-agent case yields more significantly different game configurations so that finding a policy that works fine for all of them is more difficult.

5 Conclusion

Model-based RL is a promising method for learning to control autonomous agents. Since learning accurate world models in high dimensional, continuous spaces is difficult, we have focused on learning useful but incomplete models instead. Here we have described a novel combination of CMACs and incomplete world models which allows for discovering successful soccer strategies and tends to outperform both PIPE and a Q(λ)/CMAC combination. Especially against better opponents CMAC models proved superior.

In some environments and for different games certain more complex filters grouping multiple context-dependent input components may be necessary. Filters combining many different, mutually dependent input components for a particular task may require a lot of storage space. Many of the possible input combinations, however, will never be experienced. A more space-efficient approach will use decision tree models to keep track of rewards and transition probabilities between leaf nodes defining “interesting” input component combinations. Starting with an initial set of low-complexity decision trees consisting of single root components, new leaf nodes may be generated online using statistical tests as done in, e.g., the G-algorithm [8]. This may lead to more and more informative patterns which could be useful for describing the main characteristics of positions in games such as chess or soccer. Finally, filter hierarchies could be constructed such that combinations of active cells of different filters activate higher-level filter cells, thus allowing for more context-dependent yet compact representations. This could make our approach suitable for learning evaluation functions of a wide variety of games.

Appendix A: Prioritized Sweeping

An efficient method determining which updates to perform is prioritized sweeping (PS) [15]. PS assigns priorities to updating the Q-values of different states according to a heuristic estimate of the size of the Q-values' updates. The algorithm keeps track of a “backward model” relating states to predecessor state/action pairs. After the update of a state value the state's predecessors are inserted in a priority queue which is then used for updating the Q-values of actions that can be performed in those states which have the highest priority.

Our Prioritized Sweeping. Moore and Atkeson's PS (M+A's PS) [15] calculates the priority of some state by checking all transitions to updated successor states and identifying the one whose update contribution is largest. Our variant allows for computing the *exact* size of updates of state values since they have been used for updating the Q-values of their predecessors, and yields more appropriate priorities. Unlike our PS, M+A's PS cannot detect large state-value changes due to many small update steps, and will not process the corresponding states. A complete description of both algorithms is given in [40].

Our implementation of CMAC models uses a set of predecessor lists $Preds_k(j)$ containing all predecessor cells of cell j in filter k. We denote the priority of cell i of filter k by $|\Delta_k(i)|$, where the value $\Delta_k(i)$ equals the change of $V_k(i)$ since the last time it was processed by the priority queue. To calculate it, we constantly update all Q-values of predecessor cells of currently processed cells, and track changes of $V_k(i)$.

The model-based update of the Q-value $Q_k(c,a)$, **Q-update(**k, c, a**)** looks as follows:

$$Q_k(c,a) \leftarrow \sum_j P^k_{cj}(a)(R_k(c,a,j) + \gamma V_k(j)),$$

where $P^k_{cj}(a) = P_k(j|c,a)$. The details of our PS look as follows:

```
**Our-Prioritized-Sweeping(x)** :
  1) Compute active cells: $f_1, \ldots, f_z$;
  2) For $k$ = 1 to $z$ do:
    2a) Update $f_k$ --- $\forall a$ do:
      2a.1) **Q-update(**$k, f_k, a$**)**;
    2b) Set $|\Delta_k(f_k)|$ to $\infty$;
    2c) Promote $(k, f_k)$ to top of queue;
  3) *While* ($n < U_{max}$ & queue $\neq$ nil)
    3a) Remove top $(k, c)$ from the queue;
    3b) $\Delta_k(c) \leftarrow 0$;
    3c) $\forall$ Predecessor cells $k, i$ of $k, c$
do:
      3c.1) $V'_k(i) \leftarrow V_k(i)$;
      3c.2) $\forall a$ do:
        3c.2.1) **Q-update(**$k, i, a$**)**;
      3c.3) $V_k(i) \leftarrow \max_a Q_k(i, a)$;
      3c.4) $\Delta_k(i) \leftarrow \Delta_k(i) + V_k(i) - V'_k(i)$
      3c.5) If $|\Delta_k(i)| > \epsilon$
        3c.5.1) Insert $i$ at priority
$|\Delta_k(i)|$;
    3d) $n \leftarrow n + 1$;
  4) Empty queue, but keep $\Delta_k(i)$ values;
```

Here U_{max} is the maximal number of updates to be performed per update-sweep. The parameter $\epsilon \in I\!R^+$ controls update accuracy. Note that another difference to M+A's PS is that we remove all entries from the queue after having processed all updates.

Appendix B: Non-Pessimistic Value Functions

To compute non-pessimistic value functions we decrease the probability of the worst transition from each filter/cell/action and then renormalize the other probabilities. Then we use the adjusted probabilities to compute the Q-functions. Thus we substitute the following for **Q-update(**k, c, a**)**:

```
Q-update-Non-Pessimistic(k,i,a):
    1) $m \leftarrow \arg\min_j \{R_k(i,a,j) + \gamma V_k(j)\}$;
    2) $n \leftarrow C_i^k(a)$;
    3) $P \leftarrow \hat{P}_{im}^k(a)$;
    4) $P_{im}^k(a) \leftarrow \frac{(P - \frac{z_\alpha^2}{2n} + \frac{z_\alpha}{\sqrt{n}}\sqrt{P(1-P) + \frac{z_\alpha^2}{4n}})}{1 + \frac{z_\alpha^2}{n}}$;
    5) $\Delta_P \leftarrow P_{im}^k(a) - \hat{P}_{im}^k(a)$;
    6) $\forall j \neq m$
        6a) $P_{ij}^k(a) \leftarrow \hat{P}_{ij}^k(a) - \frac{\Delta_P C_{ij}^k(a)}{C_i^k(a) - C_{im}^k(a)}$;
    7) Q-update($k, i, a$);
```

Here $C_{ij}^k(a)$ counts the number of transitions of cell i to j in filter k after selecting action a and $C_i^k(a)$ counts the number of times action a was selected and cell i of filter k was activated. We obtain $\hat{P}_{ij}^k(a)$, the estimated transition probability, by dividing them.

The variable z_α determines the step size for decreasing worst transition probabilities. To select the worst transition in step 2, we only compare existing transitions (we check whether $\hat{P}_{ij}^k(a) > 0$ holds). Note that if there is only one transition for a given filter/cell/action triplet then there will not be any renormalization. Hence the “probabilities” may not sum up to 1. Consequentially, if some filter/cell/action has not occurred frequently then it will contribute just a comparatively small Q-value and thus have less impact on the computation of the overall Q-value.

Appendix C: Q(λ)-learning

Q-learning [38], and [39] enables an agent to learn a policy by repeatedly executing actions given the current state. At each time step the algorithm uses 1-step lookahead to update the currently selected filter/cell/action pairs (FCAPs):

```
Q-learning(k, c_t, a_t, r_t, c_{t+1}) :
    1) $e'_t \leftarrow (r_t + \gamma V_k(c_{t+1}) - Q_k(c_t, a_t))$;
    2) $Q_k(c_t, a_t) \leftarrow Q_k(c_t, a_t) + \alpha_n(k, c_t, a)e'_t$;
```

Here $V_k(c) = \max_a Q_k(c, a)$, $\alpha_n(k, c, a)$ is the learning rate for the n^{th} update of FCAP (k, c, a), and e'_t is the temporal difference or TD(0)-error, which tends to decrease over time.

The learning rate $\alpha_n(k, c, a)$ should decrease online, such that it fulfills two conditions for stochastic iterative algorithms [39], [7]. The conditions for the learning rate $\alpha_n(k, c, a)$ are:

(1) $\sum_{n=1}^{\infty} \alpha_n(k, c, a) = \infty$, and

(2) $\sum_{n=1}^{\infty} \alpha_n^2(k, c, a) < \infty$.

Learning rate adaptions for which the conditions are satisfied may be of the form : $\alpha_n = \frac{1}{n^\beta}$, where n is a variable that counts the number of times an FCAP has been updated.

Q(λ)-learning uses eligibility traces $l_t(k, c, a)$ [3], and [32] to allow for updating multiple FCAPs which have occurred in the past. We use the replacing traces algorithm [30]:

$$\begin{array}{lcl}
l_{t+1}(k, c, a) \leftarrow & \gamma\lambda l_t(k, c, a) & \text{if } f_t^k \neq c \\
l_{t+1}(k, c, a) \leftarrow & 1 & \text{if } f_t^k = c \text{ and } a_t = a \\
l_{t+1}(k, c, a) \leftarrow & 0 & \text{if } f_t^k = c \text{ and } a_t \neq a
\end{array}$$

where λ discounts the influence of FCAPs occurring in the distant future relative to immediate FCAPs. After updating the eligibility traces we update the Q-values: $\forall (k, c, a) \;\; do$:

$$Q_k(c, a) \leftarrow Q_k(c, a) + \alpha[e'_t \eta_k^t(c, a) + e_t l_t(k, c, a)]$$

where $\eta_k^t(c, a)$ denotes the indicator function which returns 1 if (k, c, a) occurred at time t, and 0 otherwise ($\alpha = \alpha_n(k, c, a)$). The TD-error e_t of the value function is defined as: $e_t \leftarrow (r_t + \gamma V_k(c_{t+1}) - V_k(c_t))$.

The procedure described here updates all occurred FCAPs at each time step. This is computationally expensive. We actually used a faster method which allows for updating Q-values in time proportional to O($z|A|$), the number of filters times actions [42].

Acknowledgments

We would like to thank Rich Sutton for inspiring suggestions concerning the potential benefits of CMACs. This research was supported by SNF grant 2100-49'144.96 "Long Short-Term Memory". This chapter is a reprint of — M. A. Wiering and R. P. Salustowicz and J. Schmidhuber, (1999), "Reinforcement Learning Soccer Teams with Incomplete World Models", Neural Networks for Robot Learning. Special issue of Autonomous Robots, Vol 7(1), pp. 77-88, 1999. It was reprinted with friendly permission from Kluwer Academic Publishers.

References

[1] Albus, J.S. (1975), "A new approach to manipulator control: The cerebellar model articulation controller (CMAC)," *Dynamic Systems, Measurement and Control*, pp. 220–227.

[2] Baluja, S. and Caruana, R. (1995), "Removing the genetics from the standard genetic algorithm," In A. Prieditis and S. Russell, editors, *Machine Learning: Proceedings of the Twelfth International Conference*, pp. 38–46. Morgan Kaufmann Publishers, San Francisco, CA.

[3] Barto, A.G., Sutton, R.S., and Anderson, C.W. (1983), "Neuronlike adaptive elements that can solve difficult learning control problems," *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13, pp. 834–846.

[4] Baxter, J., Tridgell, A., and Weaver, L. (1997), "Knightcap: A chess program that learns by combining TD(λ) with minimax search," Technical report, Australian National University, Canberra.

[5] Bellman, R. (1961), *Adaptive Control Processes*, Princeton University Press.

[6] Berliner, H. (1977), ”Experiences in evaluation with BKG - a program that plays backgammon,” *Proceedings of IJCAI*, pp. 428–433.

[7] Bertsekas, D.P. and Tsitsiklis, J.N. (1996), *Neuro-dynamic Programming*. Athena Scientific, Belmont, MA.

[8] Chapman, D. and Kaelbling, L.P. (1991), ”Input generalization in delayed reinforcement learning,” *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI)*, Vol. 2, pp. 726–731, Morgan Kaufman.

[9] Cramer, N.L. (1985), “A representation for the adaptive generation of simple sequential programs”, *Proceedings of an International Conference on Genetic Algorithms and Their Applications*, pp. 183–187, Hillsdale NJ.

[10] Dickmanns, D., Schmidhuber, J., and Winklhofer, A. (1986), “Der genetische Algorithmus: Eine Implementierung in Prolog”, Fortgeschrittenenpraktikum, Institut für Informatik, Lehrstuhl Prof. Radig, Technische Universität München.

[11] Holland, J.H. (1975), *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor.

[12] Kaelbling, L.P., (1993), *Learning in Embedded Systems,* MIT Press.

[13] Koza, J.R. (1992), “Genetic evolution and co-evolution of computer programs,” *Artificial Life II*, pp. 313–324, Addison Wesley Publishing Company.

[14] Lin, L-J. (1993), *Reinforcement Learning for Robots Using Neural Networks*. PhD thesis, Carnegie Mellon University, Pittsburgh.

[15] Moore, A. and Atkeson, C.G. (1993), “Prioritized sweeping: Reinforcement learning with less data and less time,” *Machine Learning*, Vol.13, pp. 103–130.

[16] Nowlan, S.J. and Hinton, G.E. (1992), “Simplifying neural networks by soft weight sharing,” *Neural Computation*, Vol. 4, pp. 173–193.

[17] Peng, J. and Williams, R.J. (1996), “Incremental multi-step Q-learning,” *Machine Learning*, Vol. 22, pp. 283–290.

[18] Rechenberg, I. (1971), *Evolutionsstrategie - Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. PhD thesis, Published 1973 by Fromman-Holzboog.

[19] Rummery, G.A. and Niranjan, M. (1994), “On-line Q-learning using connectionist sytems,” Technical Report CUED/F-INFENG-TR 166, Cambridge University, UK.

[20] Sałustowicz, R.P. and Schmidhuber, J. (1997). “Probabilistic incremental program evolution,” *Evolutionary Computation*, Vol. 5(2), pp. 123–141.

[21] Sałustowicz, R.P., Wiering, M.A., and Schmidhuber, J. (1997), “Evolving soccer strategies,” *Proceedings of the Fourth International Conference on Neural Information Processing (ICONIP’97)*, pp. 502–506. Springer-Verlag Singapore.

[22] Sałustowicz, R.P., Wiering, M.A., and Schmidhuber, J. (1997), “On learning soccer strategies,” *Proceedings of the Seventh International Conference on Artificial Neural Networks (ICANN’97)*, Vol. 1327 of *Lecture Notes in Computer Science*, pp. 769–774, Springer-Verlag Berlin Heidelberg.

[23] Sałustowicz, R.P., Wiering, M.A., and Schmidhuber, J. (1998), “Learning team strategies: Soccer case studies,” *Machine Learning*, Vol. 33(2/3), pp. 263–282.

[24] Samuel, A.L. (1959), “Some studies in machine learning using the game of checkers”, *IBM Journal on Research and Development*, Vol. 3, pp. 210–229.

[25] Santamaria, J.C., Sutton, R.S., and Ram, A. (1996), “Experiments with reinforcement learning in problems with continuous state and action spaces”, Technical Report COINS 96-088, Georgia Institute of Technology, Atlanta.

[26] Schmidhuber, J. (1994), “On learning how to learn learning strategies,” Technical Report FKI-198-94, Fakultät für Informatik, Technische Universität München, Revised January 1995.

[27] Schmidhuber, J., Zhao, J., and Schraudolph, N. (1997), "Reinforcement learning with self-modifying policies," In S. Thrun and L. Pratt, editors, *Learning to learn*, pp. 293–309. Kluwer.

[28] Schmidhuber, J., Zhao, J., and Wiering, M.A. (1997), "Shifting inductive bias with success-story algorithm, adaptive Levin search, and incremental self-improvement," *Machine Learning*, Vol. 28, pp. 105–130.

[29] Schraudolph, N.N., Dayan, P., and Sejnowski, T.J. (1994), "Temporal difference learning of position evaluation in the game of go," *Advances in Neural Information Processing Systems*, Vol. 6, pp. 817–824. Morgan Kaufmann, San Francisco.

[30] Singh, S.P., and Sutton, R.S. (1996), "Reinforcement learning with replacing elibibility traces", *Machine Learning*, Vol. 22, pp. 123–158.

[31] Stone, P., Veloso, M., and Riley, P. (1999), "CMUnited-98 champion simulator team," *RoboCup-98: Robot Soccer World Cup II*, Vol. 1604 of *Lecture Notes in Artificial Intelligence*, pp. 61–76, Springer-Verlag.

[32] Sutton, R.S. (1988), "Learning to predict by the methods of temporal differences," *Machine Learning*, Vol. 3, pp. 9–44.

[33] Sutton, R.S. (1996), "Generalization in reinforcement learning: Successful examples using sparse coarse coding," *Advances in Neural Information Processing Systems 8*, pp. 1038–1045. MIT Press, Cambridge MA.

[34] Sutton, R.S. and Barto, A.G. (1998) *Reinforcement learning: an introduction*. MIT Press/Bradford Books.

[35] Tesauro, G. (1992), "Practical issues in temporal difference learning," *Advances in Neural Information Processing Systems 4*, pp 259–266, San Mateo, CA: Morgan Kaufmann.

[36] Thrun, S. (1995), "Learning to play the game of chess," *Advances in Neural Information Processing Systems 7*, pp. 1069–1076, San Fransisco, CA: Morgan Kaufmann.

[37] Thrun, S., Fox, D., and Burgard, W. (1998), "A probabilistic approach to concurrent mapping and localization for mobile robots", *Machine Learning*, Vol. 31, pp. 29–53. Also appeared in *Autonomous Robots* Vol. 5, pp. 253–271, 1998 as joint issue.

[38] Watkins, C.J.C.H. (1989), *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, England.

[39] Watkins, C.J.C.H. and Dayan, P. (1992) "Q-learning," *Machine Learning*, Vol. 8, pp. 279–292.

[40] Wiering, M.A. (1999), *Explorations in Efficient Reinforcement Learning*. PhD thesis, University of Amsterdam / IDSIA.

[41] Wiering, M.A. and Schmidhuber, J. (1998), "Efficient model-based exploration," *Proceedings of the Sixth International Conference on Simulation of Adaptive Behavior: From Animals to Animats 6*, pp. 223–228, MIT Press/Bradford Books.

[42] Wiering, M.A. and Schmidhuber, J. (1998), "Fast online Q(λ)", *Machine Learning*, Vol. 33(1), pp. 105–116.