

# CMAC Models Learn to Play Soccer

Proceedings of the 8th International Conference on Artificial Neural Networks  
(ICANN'98), L. Niklasson and M. Bodén and T. Ziemke (eds.), Springer-Verlag,  
London, pages 443-448. 1998.

Marco Wiering, Rafał Śaustowicz, Jürgen Schmidhuber  
IDSIA  
Lugano, Switzerland

## Abstract

Traditional reinforcement learning methods require a function approximator (FA) for learning value functions in large or continuous state spaces. We describe a novel combination of CMAC-based FAs and adaptive world models (WMs) estimating transition probabilities and rewards. Simple variants are tested in multiagent soccer environments where they outperform the evolutionary method PIPE which performed best in previous comparisons.

## 1 Introduction

Most existing reinforcement learning (RL) methods are based on function approximators (FAs) learning value functions (VFs) which map state/action pairs to the expected outcome (reinforcement) of a trial [8, 10]. In non-Markovian, multiagent environments, learning value functions is hard. This makes evolutionary methods a promising alternative. For instance, in previous work on learning soccer strategies [7] we found that Probabilistic Incremental Program Evolution (PIPE) [5], a novel evolutionary approach to searching program space, outperforms  $Q(\lambda)$  [4, 8, 10] combined with FAs based on linear neural networks or neural gas [6]. PIPE was able to isolate important features and combine them in programs with low algorithmic complexity. This motivates our present approach: VF-based RL should also profit from (a) feature selection, (b) existence of low-complexity solutions, and (c) incremental search for more complex solutions where simple ones do not work.

**World models.** *Direct* RL methods [8, 10] do not require a world model (WM). They use temporal differences (TD) [8] for training FAs to learn a VF from simulated trajectories through state/action space. *Indirect* RL, however, learns a WM [3] estimating the reward function and the transition probabilities between states, then uses dynamic programming [2, 3] for computing the VF. This can significantly speed up learning in discrete state/action spaces [3].

For continuous spaces, WMs are most effectively combined with *local* FAs consisting of many small, localized parts. While learning accurate WMs in high-dimensional, continuous, partially observable environments is hard, it is possible to learn useful but incomplete models instead.

**CMAC models.** We will present a novel combination of CMACs with world models. CMACs [1] use filters mapping inputs to a set of activated cells. Each cell has a Q-value for each action. The Q-values of currently active cells are averaged to compute overall Q-values required for action selection. Previous work combined CMACs with Q-learning [10] and  $Q(\lambda)$  methods [9]. We combine CMACs with WMs and learn an independent model for each filter. These WMs are then used by a version of prioritized sweeping (PS) [3] for computing the Q-functions. Later we will see that CMAC models can quickly learn to play a good soccer game and to surpass PIPE’s performance.

**Outline.** Section 2 describes our soccer environment. Section 3 presents our CMAC-based FAs and describes how they are combined with model-based learning. Section 4 describes experimental results. Section 5 concludes.

## 2 Soccer Simulations

Our discrete-time simulations (see [7] for details) involve two teams. There are 1 or 3 players per team. We use a two-dimensional continuous Cartesian coordinate system for the field. 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).

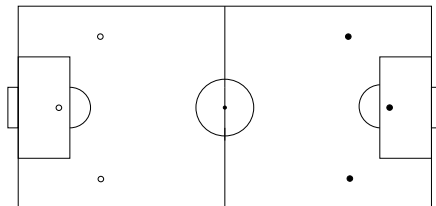


Figure 1: *Players and ball (center) in initial positions. Players of a 1 player team are those furthest in the back.*

**Players/Ball.** Players are represented by solid circles. A player whose circle intersects the ball can pick it up and own it. The ball can be moved or shot by the player who owns it. When shot, the speed of the ball decreases over time due to friction. Players collide when their circles intersect. This causes both players to bounce back to their positions at the previous time step. If one of them has owned the ball then the ball will change owners. Player actions are:  $\{go\_forward, turn\_to\_ball, turn\_to\_goal, shoot\}$ .

**Action framework.** A game lasts from time  $t = 0$  to time  $t_{end} = 5000$ . The temporal order in which players execute their moves during each time step is chosen randomly. We use policy-sharing for selecting actions: all players share the same Q-functions or PIPE-programs. Once all players have selected a move, the ball moves according to its speed and direction. If a team scores or  $t = t_{end}$  then all players and ball will be reset to their initial positions.

**Input.** At any given time a player’s input vector  $\vec{x}$  consists of 16 (1 player) or 24 (3 players) features: (1) Three boolean inputs that tell whether the player/a team member/opponent team has the ball. (2) Polar coordinates (distance, angle) of both goals and the ball with respect to the player’s orientation and position. (3) Polar coordinates of both goals relative to the ball’s orientation and position. (4) Ball speed. (5) Polar coordinates of all other players w.r.t. the player ordered by (a) teams and (b) distances to the player.

### 3 CMAC Models

CMACs [1] use multiple filters to extract multiple characteristic input features. Each filter consists of several cells with associated Q-values. Applying the filters yields a set of activated cells (a discrete distributed representation of the input). Their Q-values are averaged to compute the overall Q-value.

**General remarks on filter design.** In principle the filters may yield arbitrary divisions of the state-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 [9]. Although hashing techniques may help to overcome storage problems, we do not believe that the random grouping is natural. We prefer hyperslices which group inputs by using *subsets* of all input-dimensions.

**Soccer filter design.** 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. Multiple filters are applied to the same input to allow for smoother generalization. For certain tasks with low-complexity solutions, this architecture will generalize well and training time will be short.

**Partitioning the input space.** Inputs representing Boolean values, distances (or speeds), and angles, are split in various ways: (1) Filters associated with *Boolean* inputs just return the input. (2) *Distance* or *ball-speed* inputs are rescaled to values between 0 and 1. Then the filters partition the input into  $n_c$  equal quanta. (3) *Angle* inputs are partitioned in  $n_c$  equal quanta in a circular (and thus natural) way — the angles  $359^\circ$  and  $0^\circ$  are grouped to the same cell.

**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

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

where  $Q_k$  is the Q-function of filter  $k$ . After computing the Q-values of all actions we select the action with maximal Q-value.

**Learning with WMs.** We introduce a novel combination of model-based RL and CMACs. Learning accurate models for complex tasks is hard. Instead we use a set of independent models to estimate the dynamics of the activated

cell of a specific 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) to the estimated models. For online learning DP is computationally very expensive, however, and some sort of efficient update-step management should be performed instead. This is done by a method similar to prioritized sweeping (PS) [3] which 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 [2]:

$$Q_f(c_i, a) := \sum_j P_f(c_j|c_i, a)(\gamma V_f(c_j) + R_f(c_i, a, c_j))$$

where  $V_f(c_i) := \max_a Q_f(c_i, a)$  and  $\gamma$  is the discount factor. PS uses a parameter to set the maximum number of updates per time step and a cutoff parameter  $\epsilon$  so that small updates are not made. After each player action we update all filter models and use PS to compute the new Q-functions. Note that PS can use different numbers of updates for different filters.

**Non-pessimistic value functions.** There is no straightforward way of combining experiences of different players in policy-sharing multiagent teams. For instance, an agent may expect certain actions to be bad due to previous unlucky experiences of another agent. To overcome this problem we compute non-pessimistic value functions: we decrease the probability of the worst transition from each cell/action to the lowest bound of its 95% confidence interval and renormalize the other probabilities. Then we use PS with the new probabilities.

**Multiple restarts.** The method sometimes may get stuck with continually losing policies (also observed with our previous simulations based on linear networks and neural gas). We could not overcome this problem by adding standard exploration techniques. Instead we reset Q-function and WM once the team has not scored for 5 games but the opponent scored during the most recent game.

## 4 Experiments

We compare the CMAC model to PIPE [5], a novel evolutionary program search method which outperformed Q( $\lambda$ )-learning combined with various FAs in previous comparisons [6, 7].

**Task.** We train and test the learners against handmade programs of different strengths. The programs are mixtures of a program which randomly executes actions and a 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 mixture programs, called  $Opponent(P_r)$ , use the random program with probability  $P_r \in \{0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1\}$ .

**CMAC model set-up.** We play a total of 200 games. 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. We use 2 filters per input (total of 32 or 48 filters) and set the number of cells  $n_c := 20$ , Q-values are initially zero. PS uses  $\epsilon := 0.01$  and a maximum of 1000 updates per time step.

**PIPE set-up.** For PIPE we play a total of 1000 games. 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 previous experiments [7].

**Results.** We plot number of points (2 for scoring more goals than the opponent during the 20 test games) against number of games in Figure 2.

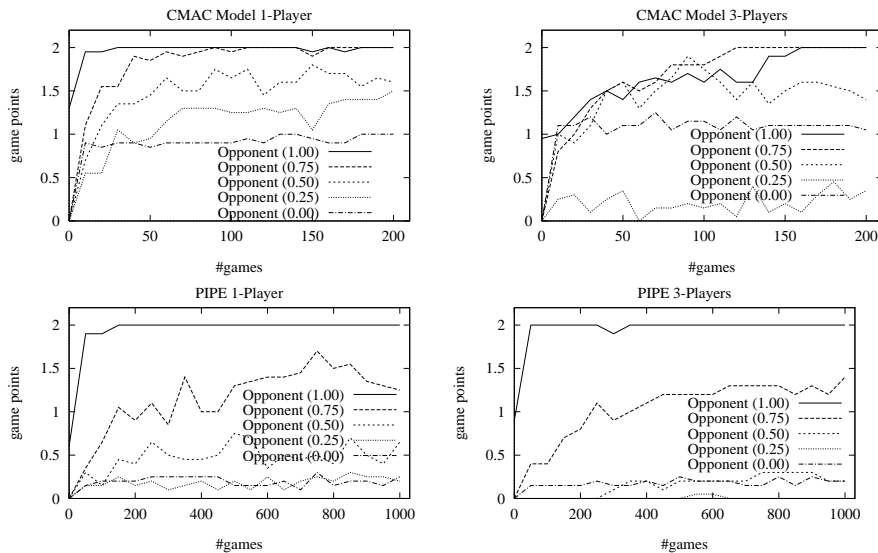


Figure 2: Number of points (means of 20 simulations) during test phases for team sizes 1 and 3. Note the varying x-axis scalings.

**1-Player case.** We observe that our CMAC model wins against almost all training programs. Only against the best 1-player team ( $P_r = 0$ ) it learns to play ties (it always finds a blocking strategy leading to a 0-0 result). PIPE is able to find programs beating the random and 75% random teams, but often does not find programs that win or play ties against the better teams.

**3-Player case.** CMAC model wins against most training opponents, but loses against the best 3-player team (with  $P_r = 0.25$ ). Note that this strategy mixture works better than always using the deterministic program ( $P_r = 0$ )

against which CMAC models play ties or even win. PIPE performs worse — it only wins against the worst opponents.

**Discussion.** Despite treating all features 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 stabilizes good strategies: given sufficient experiences, the policy will hardly change anymore.

## 5 Conclusion

A novel combination of CMACs and world models allows for finding successful soccer strategies with low complexity, and tends to outperform PIPE.

In some environments certain more complex filters grouping multiple context-dependent inputs may be necessary. Instead of handcrafting CMAC filters for the value function, methods learning them from reinforcement will be an interesting topic for future research.

**Acknowledgments.** This work was supported in part by SNF grant 2100-49'144.96 “Long Short-Term Memory”.

## References

- [1] J. S. Albus. A new approach to manipulator control: The cerebellar model articulation controller (CMAC). *Dynamic Systems, Measurement and Control*, 97:220–227, 1975.
- [2] R. Bellman. *Adaptive Control Processes*. Princeton University Press, 1961.
- [3] A. Moore and C. G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning*, 13:103–130, 1993.
- [4] J. Peng and R. J. Williams. Incremental multi-step Q-learning. *Machine Learning*, 22:283–290, 1996.
- [5] R. P. Sałustowicz and J. Schmidhuber. Probabilistic incremental program evolution. *Evolutionary Computation*, 5(2):123–141, 1997.
- [6] R. P. Sałustowicz, M. A. Wiering, and J. Schmidhuber. Evolving soccer strategies. In *Proceedings of the Fourth International Conference on Neural Information Processing (ICONIP'97)*, pages 502–506. Springer-Verlag Singapore, 1997.
- [7] R. P. Sałustowicz, M. A. Wiering, and J. Schmidhuber. Learning team strategies: Soccer case studies. *Machine Learning*, 1998. To appear.
- [8] R. S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

- [9] R. S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, editors, *Advances in Neural Information Processing Systems 8*, pages 1038–1045. MIT Press, Cambridge MA, 1996.
- [10] C. Watkins. *Learning from Delayed Rewards*. PhD thesis, King’s College, Cambridge, 1989.