# On Learning Soccer Strategies

Rafał Sałustowicz, Marco Wiering, Jürgen Schmidhuber

IDSIA, Corso Elvezia 36, 6900 Lugano, Switzerland
e-mail: {rafal, marco, juergen}@idsia.ch

In W. Gerstner, A. Germond, M. Hasler, and J.-D. Nicoud, editors, Proceedings of the Seventh International Conference on Artificial Neural Networks (ICANN'97), volume 1327 of Lecture Notes in Computer Science, pages 769–774. Springer-Verlag Berlin Heidelberg, 1997.

**Abstract.** We use simulated soccer to study multiagent learning. Each team's players (agents) share action set and policy but may behave differently due to position-dependent inputs. All agents making up a team are rewarded or punished collectively in case of goals. We conduct simulations with varying team sizes, and compare two learning algorithms: TD-Q learning with linear neural networks (TD-Q) and Probabilistic Incremental Program Evolution (PIPE). TD-Q is based on evaluation functions (EFs) mapping input/action pairs to expected reward, while PIPE searches policy space directly. PIPE uses an adaptive probability distribution to synthesize programs that calculate action probabilities from current inputs. Our results show that TD-Q has difficulties to learn appropriate shared EFs. PIPE, however, does not depend on EFs and finds good policies faster and more reliably.

## 1 Introduction

Soccer recently received much attention by various multiagent researchers. There have been attempts to learn low-level cooperation tasks such as pass play. Published results on learning entire soccer strategies, however, have been limited to extremely reduced scenarios (e.g., two single opponent players in a $5 \times 4$ grid world [5]). Our comparatively complex case study will involve simulations with continuous-valued inputs, simple physical laws to model ball bounces and friction, and up to 11 players (agents) on each team.

**Evaluation functions versus search through policy space.** There are two rather obvious classes of candidate algorithms for multiagent reinforcement learning (RL). The first includes traditional single-agent RL algorithms based on adaptive evaluation functions (EFs) [1]. Usually online variants of dynamic programming and function approximators are combined to model EFs mapping input-action pairs to expected discounted future reward. Methods from the second class do not require EFs. Their policy space consists of complete algorithms defining agent behaviors, and they search policy space directly. Well-known members of this class are Levin search [3], Genetic Programming, e.g. [2], and Probabilistic Incremental Program Evolution [6].

**Comparison.** In our case study we compare two learning algorithms, each representative of its class: TD-Q learning with linear neural nets (TD-Q) [4] and Probabilistic Incremental Program Evolution (PIPE) [6]. TD-Q selects actions according to linear neural networks trained with the delta rule to map player inputs to evaluations of alternative actions. PIPE uses a probability distribution to synthesize programs that calculate action probabilities from inputs. The probability distribution is then adapted using an evolutionary approach.

## 2 Soccer Simulation

Our discrete-time simulations involve two teams. There are either 1 or 11 players per team. Players can move with and without the ball or shoot it. As in indoor soccer the field is surrounded by impassable walls except for the two goals centered in the east and west walls. The ball slows down due to friction (after having been shot) and bounces off walls obeying the law of equal reflection angles. Players are "solid". If a player, coming from a certain angle, attempts to traverse a wall then it "glides" on it, loosing only that component of its speed which corresponds to the movement direction hampered by the wall. Collisions of players cause them to bounce back to their positions at the previous time step. If one of them had the ball then the ball changes owners. There are fixed initial positions for all players and the ball (see Figure 1). A game lasts from

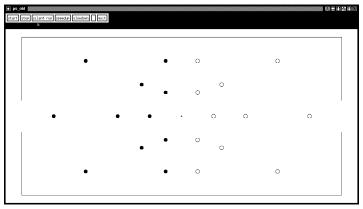

**Fig. 1.** 22 players and ball in initial positions. Players of a 1 player team are those furthest in the back (goalkeepers).

time $t = 0$ to time $t_{end}$.

**Action Framework/Cycles.** At each discrete time step $0 \leq t < t_{end}$ each player executes a "cycle". A cycle consists of: (1) an attempt to get the ball, if it is close enough, (2) input computation, (3) action selection and execution, and (4) another attempt to get the ball, if it is close enough. Once all players have executed a cycle we move the ball. If a team scores or $t = t_{end}$ then all players and ball are reset to their initial positions.

**Inputs.** Player $p$'s input at a given time $t$ is an input vector $\mathbf{i}(p,t)$. Vector $\mathbf{i}(p,t)$ has 14 components: (1) Three boolean inputs that tell whether the player/a team member/an opponent has the ball. (2) Polar coordinates (distance, angle) of both goals and the ball with respect to a player-centered coordinate system. (3) Polar coordinates of both goals with respect to a ball-centered coordinate system. (4) Ball speed. Note that these inputs make the environment partially observable.

**Actions.** Players may execute actions from action set $ASET$. $ASET$ contains: *go_forward*, *turn_to_ball*, *turn_to_goal* and *shoot*. Shots are noisy and noise makes long shots less precise than close passes. For a detailed description of the soccer simulator see [7].

## 3 Probabilistic Incremental Program Evolution (PIPE)

We use PIPE [6] to synthesize programs which, given player $p$'s input vector $\mathbf{i}(p,t)$, select actions from $ASET$.

**Action Selection.** Action selection depends on 5 variables: $g \in I\!R$, $A_i \in I\!R$, $\forall i \in ASET$. Action $i \in ASET$ is selected with probability $P_{A_i}$ according to the Boltzmann-Gibbs distribution at temperature $\frac{1}{g}$:

$$P_{A_i} := \frac{e^{A_i \cdot g}}{\sum_{\forall j \in ASET} e^{A_j \cdot g}} \qquad \forall i \in ASET \tag{1}$$

All $A_i$ and $g$ are calculated by a program.

**Programs.** A main program $\textsc{Program}$ consists of a program $\textsc{Prog}^g$ which computes the "greediness" parameter $g$ and 4 "action programs" $\textsc{Prog}^i$ ($i \in ASET$). The result of applying $\textsc{Prog}$ to data $x$ is denoted $\textsc{Prog}(x)$. Given $\mathbf{i}(p,t)$, $\textsc{Prog}^i(\mathbf{i}(p,t))$ returns $A_i$ and $g := |\textsc{Prog}^g(\mathbf{i}(p,t))|$. An action $i \in ASET$ is then selected according to (1).

**Program Instructions.** A program $\textsc{Prog}$ contains instructions from a function set $F$ and a terminal set $T$. We use $F = \{+, -, *, \%, sin, cos, exp, rlog\}$ and $T = \{\mathbf{i}(p,t)_1, \ldots, \mathbf{i}(p,t)_v, R\}$, where % denotes protected division ($\forall y, z \in I\!R, z \neq 0$: $y\%z = y/z$ and $y\%0 = 1$), $rlog$ denotes protected logarithm ($\forall y \in I\!R, y \neq 0$: $rlog(y)$=log(abs($y$)) and $rlog(0) = 0$), $\mathbf{i}(p,t)_l$ $1 \leq l \leq v$ denotes component $l$ of a vector $\mathbf{i}(p,t)$ with $v$ components and $R$ represents the *generic random constant* from [0;1).

**PIPE Overview.** PIPE programs are encoded in $n$-ary trees that are parsed depth first from left to right, with $n$ being the maximal number of function arguments. PIPE generates programs according to a probability distribution over all possible programs composable from the instruction set ($F \cup T$). The probability distribution is stored in an underlying *probabilistic prototype tree* ($PPT$). The $PPT$ contains at each node an initial probability for each instruction from $F \cup T$ and a random constant from [0;1). Programs are generated by traversing the $PPT$ depth first starting at the root node. At each node an instruction is picked according to the node's probability distribution. In case the *generic random constant* is picked it is instantiated either to the value stored in the $PPT$ node or a random value from [0;1), depending on the instruction's probability. To adapt $PPT$'s probabilities PIPE generates successive populations of programs. It evaluates each program of a population and assigns it a scalar, non-negative "fitness value", which reflects the program's performance. To evaluate a program we play one entire soccer game. PIPE then adapts $PPT$'s probabilities so that the overall probability of creating the best program of the current population increases. Finally *PPT*'s probabilities are mutated to better explore the search space. All details can be found in [6].

## 4 TD-Q Learning

One of the most widely used EF-based approaches to reinforcement learning is TD-Q learning. We use Lin's successful TD($\lambda$) Q-variant [4]. For efficiency reasons our TD-Q version uses linear neural nets (nets with hidden units require too much simulation time). The goal of the networks is to map the player-specific input $\mathbf{i}(p,t)$ to action evaluations $Q(\mathbf{i}(p,t),a_1),\ldots,$Q$(\mathbf{i}(p,t),a_4)$, where $a_i \in ASET$. We use the same networks for all policy-sharing players. We reward the players equally whenever a goal has been made or the game is over.

**Action selection.** We use a linear net for each of the four actions $\{a_1,\ldots,$ $a_4\}$. To select an action for player $p$ we first calculate Q-values of all actions. The Q-value of action $a_k$, given input $\mathbf{i}(p,t)$ is

$$Q(\mathbf{i}(p,t),a_k) = \sum_{j=1}^{j=v} w_j^k \mathbf{i}(p,t)_j + w_{v+1}^k, \tag{2}$$

where $\mathbf{w}^k$ is the weight vector for action network $k$, $v$ denotes the number of inputs, and $\mathbf{w}_{v+1}^k$ is the bias strength. Once all Q-values have been calculated, a single action is chosen according to the Boltzmann rule (see assignment (1)).

**TD-Q learning.** Each game consists of separate trials. For each player $p$ there is a variable time-pointer $t(p)$. At trial start we set $t(p)$ to current game time $t^c$. We increment $t(p)$ after each cycle of player $p$. The trial stops once one of the teams scores or the game is over. Denote player $p$'s final time-pointer by $t^*(p)$. To achieve an optimal strategy we want the Q-value $Q(\mathbf{i}(p,t),a_k)$ for selecting action $a_k$ given input $\mathbf{i}(p,t)$ to approximate

$$Q(\mathbf{i}(p,t),a_k) \sim \mathcal{E}(\gamma^{t^*(p)-t(p)} R(t^*(p))), \tag{3}$$

where $\mathcal{E}$ denotes the expectation operator, $0 \leq \gamma \leq 1$ the discount factor which encourages quick goals (or a lasting defense against opponent goals), and $R(t^*(p))$ denotes the reinforcement at trial end (-1 if opponent team scores, 1 if own team scores, 0 otherwise).

To learn these Q-values we monitor player experiences in player-dependent history lists with maximum size $H_{max}$. After each trial we calculate examples using the TD-Q method. For each player history list, we compute desired Q-values $Q^{new}(t)$ for selecting action $a_t$, given $\mathbf{i}(p,t)$ ($t = t^1(p),\ldots,t^*(p)$, where $t^1(p) = Max(1, t^*(p) + 1 - H_{max})$) as follows:

$$\begin{aligned} Q^{new}(t^*(p)) &:= R(t^*(p)); \\ Q^{new}(t) &:= \gamma \cdot [\lambda \cdot Q^{new}(t+1) + (1-\lambda) \cdot Max_k\{Q(\mathbf{i}(p,t),a_k)\}]. \end{aligned}$$

$\lambda$ determines future experiences' degree of influence.

Once all players have created TD-Q training examples, we train the selected nets to minimize their TD-Q errors. All player history-lists are processed as follows: we train the networks starting with the first history list entry of player 1, then we take the first entry of player 2, etc. Once all first entries have been processed we start processing the second entries, etc. The nets are trained using the delta-rule with learning rate $Lr^N$. All details can be found in [7].

## 5 Experiments

We analyze TD-Q's and PIPE's behavior as we vary team size. We perform 10 independent runs for each combination of learning algorithm and team size. We play 3300 games of length $t_{end} = 5000$ for both team sizes (1 and 11). Every 100 games we test current performance by playing 20 test games (no learning) against a "biased random opponent" $BRO$ and summing the score results.

$BRO$ randomly executes actions from $ASET$. $BRO$ is not a bad player due to the initial bias in the action set. If we let $BRO$ play against a non-acting opponent $NO$ (all $NO$ can do is block) for twenty 5000 time step games then $BRO$ wins against $NO$ with on average 71.5 to 0.0 goals for team size 1 and 108.6 to 0.5 goals for team size 11.

**PIPE Set-up.** Parameters for PIPE runs are: $P_T$=0.8, $\varepsilon = 1$, $P_{el} = 0$, $PS$=10, $lr$=0.2, $P_M$=0.1, $mr$=0.2, $T_R$=0.3, $T_P$=0.999999 (see [6] for details). During performance evaluations we test the current best-of-current-population program (except for the first evaluation where we test a random program).

**TD-Q Set-up.** After a thorough parameter search we found the following best parameters for TD-Q runs: $\gamma$=0.99, $Lr^N$=0.0001, $\lambda$=0.9, $H_{max}$=100. All weights are randomly initialized in $[-0.01, 0.01]$. During each run the Boltzmann rule's greediness parameter $g$ is linearly increased from 0 to 60.

**Results.** We plot goals scored by learner and opponent against number of games in Figure 2. PIPE's score differences continually increase. It always quickly

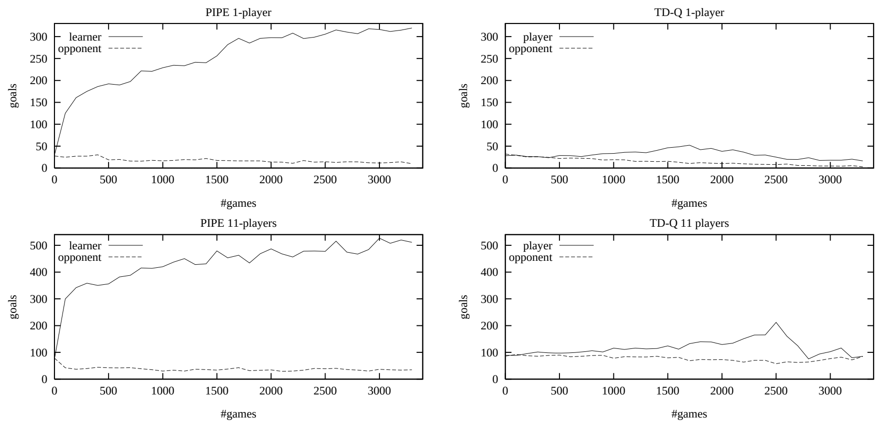

**Fig. 2.** Average number of goals scored during all test phases, for team sizes 1 and 11.

learns an appropriate policy *regardless* of team size. PIPE learns much faster than TD-Q. This is partially due to PIPE's ability to efficiently select the relevant input features for each action. TD-Q's score differences first increase until TD-Q scores roughly twice as many goals as in the beginning (when it was still random). Then, however, performance breaks down. This phenomenon is most pronounced in the 11 player TD-Q run.

**TD-Q's outlier problem.** To understand TD-Q's major performance breakdown in the 11 player case we saved a network just before breakdown (after 2300 games). We then analyzed the network's behaviour with our simulator and discovered the "outlier problem". There are particular game constellations where the opponent has the ball and is close to the goal but somehow fails to score. Instead, the TD-Q team manages to grab the ball and score soon afterwards. How does this affect its EFs? Once the linear nets have learned a good EF, they assign negative evaluations to all actions in such dangerous situations, since most of the times the opponent will indeed score. But once there is an outlier, the nets are trained on completely different values. In single-player teams this is less of a problem. In 11 player teams, however, the effect on the nets is 11-fold. We could not get rid of this problem, neither by (1) bounding error updates nor by (2) lowering learning rates or lambda. Case (2) actually just causes slower learning. Increasing the greediness value tends to help a bit, but does not work well either.

## 6 Discussion

In a simulated soccer case study with policy-sharing agents we compared a direct policy search method (PIPE) and an optimized EF-based one (TD-Q). Both competed against a biased random opponent. PIPE quickly learned to beat this opponent. TD-Q achieved performance improvements, too, but its results were less exciting, especially in case of multiple agents per team. TD-Q's problems were due to: (1) partial observability of the environment, and (2) inability to handle outliers.

### Acknowledgments

Thanks to Jieyu Zhao, Nicol Schraudolph, Luca Gambardella, and Cristina Versino for valuable comments and suggestions.

## References

1. D. P. Bertsekas and J. N. Tsitsiklis. *Neuro-Dynamic Programming.* Athena Scientific, Belmont, MA, 1996.
2. N. L. Cramer. A representation for the adaptive generation of simple sequential programs. In J.J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and Their Applications*, pages 183–187, Hillsdale NJ, 1985. Lawrence Erlbaum Associates.
3. L. A. Levin. Universal sequential search problems. *Problems of Information Transmission*, 9(3):265–266, 1973.
4. L. J. Lin. *Reinforcement Learning for Robots Using Neural Networks.* PhD thesis, Carnegie Mellon University, Pittsburgh, January 1993.

5. M. L. Littman. Markov games as a framework for multi-agent reinforcement learning. In A. Prieditis and S. Russell, editors, *Machine Learning: Proceedings of the Eleventh International Conference*, pages 157–163. Morgan Kaufmann Publishers, San Francisco, CA, 1994.
6. R. P. Sałustowicz and J. Schmidhuber. Probabilistic incremental program evolution. *Evolutionary Computation*, to appear, 1997. See ftp://ftp.idsia.ch/pub/rafal/-PIPE.ps.gz.
7. R. P. Sałustowicz, M. A. Wiering, and J. Schmidhuber. Learning team strategies with multiple policy-sharing agents: A soccer case study. Technical Report IDSIA-29-97, IDSIA, 1997. See ftp://ftp.idsia.ch/pub/rafal/soccer.ps.gz.