# Learning Team Strategies With Multiple Policy-Sharing Agents: A Soccer Case Study

Technical Report IDSIA-29-97

Rafał Sałustowicz, Marco Wiering, Jürgen Schmidhuber
IDSIA, Corso Elvezia 36, 6900 Lugano, Switzerland
e-mail: {rafal, marco, juergen}@idsia.ch
tel.: +41-91-9919838 fax: +41-91-9919839

January 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 adaptive "probabilistic prototype trees" to synthesize programs that calculate action probabilities from current inputs. Our results show that TD-Q encounters several difficulties in learning appropriate shared EFs. PIPE, however, does not depend on EFs and can find good policies faster and more reliably. This suggests that in multiagent learning scenarios direct search through policy space can offer advantages over EF-based approaches.

**Keywords:** Multiagent Reinforcement Learning, Soccer, TD-Q Learning, Probabilistic Incremental Program Evolution

## 1 Introduction

**Policy-sharing.** Multiagent learning tasks often require several agents to learn to cooperate. In general there may be quite different types of agents specialized to solving particular subtasks. Some cooperation tasks, however, can also be solved by teams of essentially identical agents whose behaviors differ only due to different, situation-specific inputs.

Our case study will be limited to such teams of agents of identical type. Each agent's modifiable policy is given by a variable data structure: for each action $a$ in a given set of possible actions the current policy determines the conditional probability that the agent will execute $a$, given its current input. Each team's members share both action set and adaptive policy. If some multiagent cooperation task indeed can be solved by homogeneous agents then policy-sharing is quite natural as it allows for greatly reducing the number of adaptive free parameters. This tends to reduce the number of required training examples (learning time) and increase generalization performance (Nowlan and Hinton, 1992).

**Challenges of Multiagent Learning.** One challenge is the "partial observability problem" (POP): in general no learner's input will tell the learner everything about its environment (which includes other changing learners). This means that each learner's environment may change in an inherently unpredictable way. Also, in reinforcement learning (RL) scenarios delayed reward/punishment is typically distributed evenly among all members of a successful/failing team of agents. This provokes the "agent credit assignment problem" (ACAP): the problem of identifying those agents that were indeed responsible for the outcome (Weiss, 1996; Versino and Gambardella, 1997).

**Evaluation functions versus search through policy space.** There are two rather obvious classes of candidate algorithms for multiagent RL. The first includes traditional single-agent RL algorithms based on adaptive evaluation functions (EFs) (Watkins, 1989; Bertsekas, 1996). Usually online variants of dynamic programming and function approximators are combined to model EFs mapping input-action pairs to expected discounted future reward. The EFs are then exploited to generate rewarding action sequences.

Methods from the second class search do not require EFs. Their policy space consists of complete algorithms defining agent behaviors, and they search policy space directly. Members of this class are Levin search (Levin, 1973; Levin, 1984; Solomonoff, 1986; Li and Vitányi, 1993; Schmidhuber, 1995; Wiering and Schmidhuber, 1996), Genetic Programming (GP) (Cramer, 1985; Dickmanns et al., 1987; Koza, 1992) and Probabilistic Incremental Program Evolution (PIPE) (Sałustowicz and Schmidhuber, 1997).

**Comparison.** In our case study we compare two learning algorithms, each representative of its class: TD-Q learning with linear neural nets (TD-Q) (Lin, 1993) and Probabilistic Incremental Program Evolution (PIPE) (Sałustowicz and Schmidhuber, 1997). We choose PIPE and TD-Q because both have already been successfully applied to interesting single-agent tasks (TD-Q also because it is very popular). TD-Q selects actions according to linear neural networks trained with the delta rule (Widrow and Hoff, 1960) to map player inputs to evaluations of alternative actions. PIPE is based on probability vector coding of program instructions (Schmidhuber, 1997), Population-Based Incremental Learning (Baluja and Caruana, 1995) and tree coding of programs used in variants of Genetic Programming (GP) (Cramer, 1985; Koza, 1992). PIPE synthesizes programs that calculate action probabilities from inputs. Experiences with programs are stored in "probabilistic prototype trees" that guide program synthesis.

**Soccer.** To come up with a challenging scenario for our multiagent learning case study we decided on a non-trivial soccer simulation. Soccer recently received much attention by various multiagent researchers (Sahota, 1993; Asada et al., 1994; Littman, 1994; Stone and

Veloso, 1995; Matsubara et al., 1996). Most early research focused on physical coordination of soccer playing robots (Sahota, 1993; Asada et al., 1994). There also have been attempts to *learn* low-level cooperation tasks such as pass play (Stone and Veloso, 1995; Matsubara et al., 1996). Recently Stone and Veloso (1996) mentioned that even team strategies might be learnable by TD($\lambda$) or genetic methods.

Published results on learning entire soccer strategies, however, have been limited to extremely reduced scenarios such as Littman's (1994) tiny $5 \times 4$ grid world with two single opponent players. Our comparatively complex case study will involve simulations with varying sets of continuous-valued inputs and actions, simple physical laws to model ball bounces and friction, and up to 11 players (agents) on each team.

In our simulations we will vary the degree of environmental observability by providing more or less informative inputs to the players. Less informative inputs tend to reduce the number of adaptive parameters but increase POP's significance.

**Results Overview.** Our results indicate: as more and more agents are added to the teams, it gets harder and harder for TD-Q to learn appropriate shared EFs, due to ACAP, POP, and problems introduced by unexpected, novel game constellations (outliers). PIPE, however, does not depend on EFs. It learns faster than TD-Q and does not seem to be affected much by ACAP, POP, or outliers. This suggests that currently PIPE-like, EF-independent techniques seem more promising in case of complex multiagent learning scenarios, unless methods for overcoming TD-Q's problems are developed.

**Outline.** Section 2 describes the soccer simulation. Section 3 describes PIPE. Section 4 describes TD-Q. Section 5 reports experimental results. Section 6 discusses our findings.

# 2 Soccer Simulations

Our discrete-time simulations involve two teams. There are either 1, 3 or 11 players per team. Players can move or shoot the ball. Each player's abilities are limited (1) by the built-in power of its pre-wired action primitives and (2) by how informative its inputs are. We conduct two types of simulations. *"Simple"* simulations involve less informative inputs and less sophisticated actions than *"complex"* simulations.

**Field.** We use a two dimensional continuous Cartesian coordinate system. The field's southwest and northeast corners are at positions (0,0) and (4,2) respectively. As in indoor soccer the field is surrounded by impassable walls except for the two goals centered in the east and west walls (see Figure 1-left). Only the ball or a player with ball can enter the goals. Goal width ($y$-extension) is 0.4, goal depth ($x$-extension beyond the field bounds) is 0.01. The east goal's "middle" is denoted $m_{ge} = (x_{ge}, y_g)$ with $x_{ge} = 4.01$ and $y_g = 1.0$ (see Figure 1-right). The west goal's middle is at $m_{gw} = (x_{gw}, y_g)$ with $x_{gw} = -0.01$.

**Ball/Scoring.** The ball is a circle with variable center coordinates $c_b = (x_b, y_b)$, variable direction $\vec{o}_b$ and fixed radius $r_b = 0.01$. Its speed at time $t$ is denoted $v_b(t)$. After having been shot the ball's initial speed is $v_b^{init}$ (max. 0.12 units per time step). Each following time step the ball slows down due to friction: $v_b(t+1) = v_b(t) - 0.005$ until $v_b(t) = 0$ or it is picked up by a player (see below). The ball bounces off walls obeying

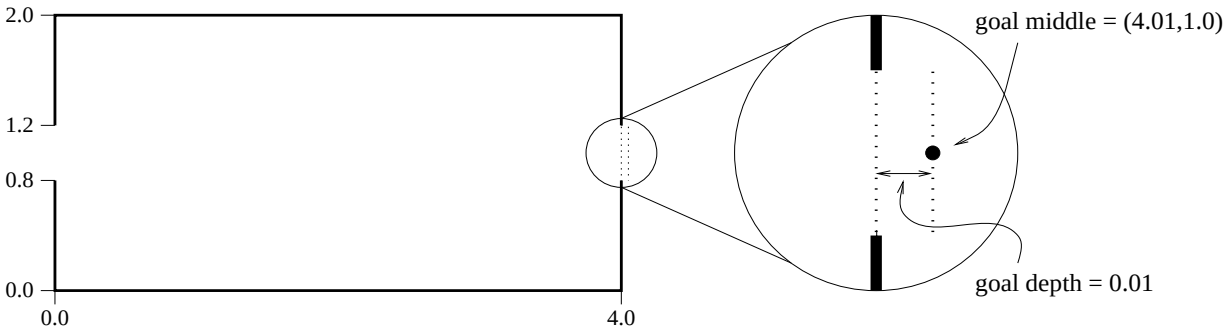


Figure 1: *Left: Field. Right: Depth and "middle" $m_{ge}$ of east goal (enlarged).*

the law of equal reflection angles as depicted in Figure 2. Bouncing causes an additional slow-down: $v_b(t+1) = v_b(t) - 0.005 - 0.01$. A goal is scored whenever $0.8 < y_b < 1.2 \wedge (x_b < 0 \vee x_b > 4.0)$.

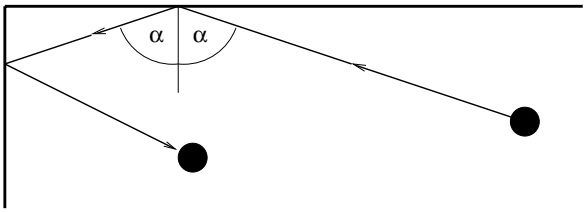


Figure 2: *Ball "reflected" by wall.*

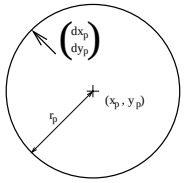


Figure 3: *Player: center $c_p = (x_p, y_p)$, radius $r_p$ and orientation $\vec{o}_p = \binom{dx_p}{dy_p}$.*

**Players.** We vary team size. The number of players per team is denoted $Z$. $Z$ can be 1, 3 or 11. There are two teams $T_{east} = \{pe_1, pe_2, \ldots, pe_Z\}$ and $T_{west} = \{pw_1, pw_2, \ldots, pw_Z\}$. Each consists of $Z$ homogeneous players $pe_j$ and $pw_j$ $(1 \leq j \leq Z)$ respectively. At a given time step each player $p \in T_{east} \cup T_{west}$ is represented by a circle with variable center $c_p = (x_p, y_p)$, fixed radius $r_p = 0.025$ and variable orientation $\vec{o}_p = \binom{dx_p}{dy_p}$ (see Figure 3). Players are "solid". If player $p$, 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. Players $p_i$ and $p_j$ collide if $dist(c_{p_i}, c_{p_j}) < r_{p_i} \vee dist(c_{p_i}, c_{p_j}) < r_{p_j}$, where $dist(c_i, c_j)$ denotes Euclidean distance between points $c_i$ and $c_j$. Collisions cause both players to bounce back to their positions at the previous time step. If one of them owned the ball then the ball changes owners (see below).

**Initial Set-up.** A game lasts from time $t = 0$ to time $t_{end}$. There are fixed initial positions for all players and the ball (see Figure 4).

Initial orientations are $\vec{o}_{pe} = \binom{-1}{0}$ $\forall pe \in T_{east}$ and $\vec{o}_{pw} = \binom{1}{0}$ $\forall pw \in T_{east}$.

**Action Framework/Cycles.** Until one of the teams scores, at each discrete time step $0 \leq t < t_{end}$ each player executes a "cycle" (the temporal order of the $2 \cdot Z$ cycles is chosen randomly). A cycle consists of: (1) attempted ball collection, (2) input computation, (3)

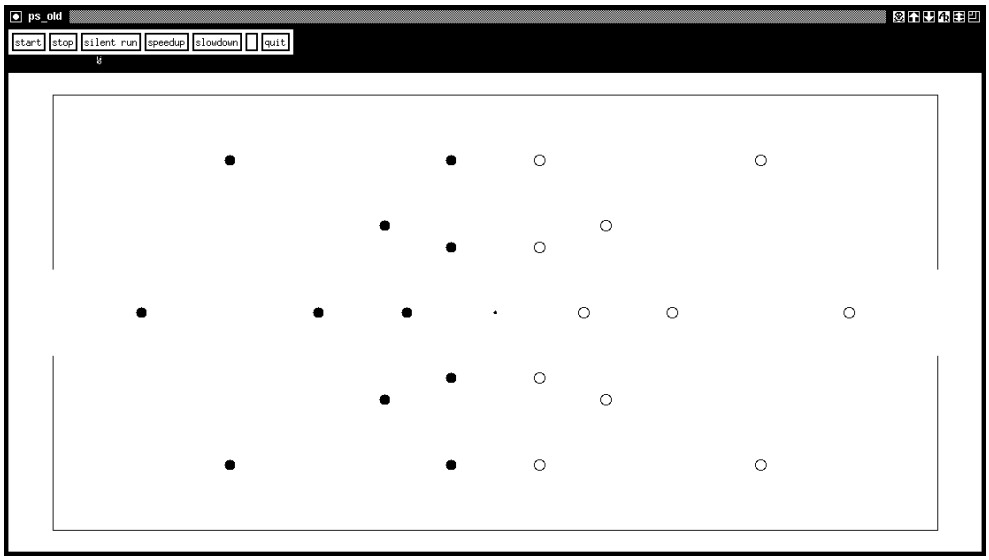


Figure 4: *22 players and ball in initial positions. Players of a 1 or 3 player team are those furthest in the back (defenders and/or goalkeeper).*

action selection, (4) action execution and (5) attempted ball collection. Once all $2 \cdot Z$ cycles have been executed we move the ball if $v_b > 0$. If a team scores or $t = t_{end}$ then all players and ball are reset to their initial positions.

***(1) Attempted Ball Collection.*** A player $p$ successfully collects ball $b$ if its radius $r_p \leq dist(c_p, c_b)$. We then set $c_b := c_p, v_b := 0$. Now the ball will move with $p$ and can be shot by $p$.

***(2) Input Computation.*** In *simple* simulations Player $p$'s input at a given time is a *simple* input vector $\vec{i}_s(p,t)$. In *complex* simulations it's a *complex* input vector $\vec{i}_c(p,t)$.

Simple vector $\vec{i}_s(p,t)$ has 14 components: (1) Three boolean inputs (coded with 1=true and -1=false) that tell whether player $p$/a team member/an opponent has the ball. (2) Polar coordinates (distance, angle) of both goals and the ball with respect to pole $c_p$ and polar axis $\vec{o}_p$ (player-centered coordinate systems). (3) Polar coordinates of both goals relative to ball-centered coordinate system with pole $c_b$ and polar axis $\vec{o}_b$ — if $v_b = 0$, then $\vec{o}_b = \vec{0}$ and the angle towards both goals is defined as 0. (4) Ball speed. Note that these inputs do not provide a lot of information about the environment (partial observability).

The 56-dimensional complex vector $\vec{i}_c(p,t)$ is a concatenation of $\vec{i}_s(p,t)$ and 21 $c_p/\vec{o}_p$-based polar coordinates of all other players ordered by (a) teams and (b) ascending distances to $p$.

TD-Q's and PIPE's input representation of distance $d$ (angle $\alpha$) is $\frac{5-d}{5}$ ($e^{-20 \cdot \alpha^2}$). This helps TD-Q since it makes close distances and small angles appear more important to

TD-Q's neural nets.

***(3) Action Selection.*** See Sections 3 and 4.

***(4) Action Execution.*** Depending on the simulation type, player $p$ may execute either *simple* actions from action set $ASET_S$ or *complex* actions from action set $ASET_C$. $ASET_S$ contains:

- *go_forward:* move player $p$ 0.025 units in its current direction $\vec{o}_p$ if without ball and $0.8 \cdot 0.025$ units otherwise.
- *turn_to_ball:* change direction $\vec{o}_p$ of player $p$ such that $\vec{o}_p := \binom{x_b - x_p}{y_b - y_p}$
- *turn_to_goal:* change direction $\vec{o}_p$ of player $p$ such that $\vec{o}_p := \binom{x_{ge} - x_p}{y_g - y_p}$, if $p \in T_{west}$ and $\vec{o}_p := \binom{x_{gw} - x_p}{y_g - y_p}$, if $p \in T_{east}$.
- *shoot:* If $p$ does not own the ball then do nothing. Otherwise, to allow for imperfect, noisy shots, execute $turn(\alpha_{noise})$ which sets $\vec{o}_p := \binom{cos(\alpha_{noise}) \cdot dx_p - sin(\alpha_{noise}) \cdot dy_p}{sin(\alpha_{noise}) \cdot dx_p + cos(\alpha_{noise}) \cdot dy_p}$, where $\alpha_{noise}$ is picked uniformly random from $-0.0277 \cdot \pi \leq \alpha_{noise} \leq 0.0277 \cdot \pi$. Then shoot ball in direction $\vec{o}_b := \vec{o}_p$. Initial ball speed is $v_b^{init} = 0.12$. Noise makes long shots less precise than close passes.

Complex actions in $ASET_C$ are parameterized. They allow for pre-wired cooperation but also increase action space. Parameter $\alpha$ stands for an angle, $P/O$ stands for some teammate player's/opponent's index from $\{1..Z-1\}/\{1..Z\}$. Indices $P$ and $O$ are sorted by distances to the player currently executing an action, where closer teammate players/opponents have lower indices. For TD-Q $\alpha$ is either picked from $s_1 = \{0, \frac{\pi}{4}, \frac{\pi}{2}, -\frac{\pi}{4}, -\frac{\pi}{2}\}$ or from $s_2 = \{0, \frac{2}{5}\pi, \frac{4}{5}\pi, -\frac{2}{5}\pi, -\frac{4}{5}\pi\}$. PIPE uses continuous angles. Player $p$ may execute the following complex actions from $ASET_C$:

- *goto_ball($\alpha$):* If $p$ owns ball do nothing. Otherwise execute *turn_to_ball*, then $turn(\alpha)$ (TD-Q: $\alpha \in s_1$) and finally *go_forward*,
- *goto_goal($\alpha$):* First execute *turn_to_goal*, then $turn(\alpha)$ (TD-Q: $\alpha \in s_1$) and finally *go_forward*.
- *goto_own_goal($\alpha$):* First execute *turn($\beta$)* such that $\vec{o}_p := \binom{x_{gw} - x_p}{y_g - y_p}$ (if $p \in T_{west}$) or $\vec{o}_p := \binom{x_{ge} - x_p}{y_g - y_p}$ (if $p \in T_{east}$); then $turn(\alpha)$ (TD-Q: $\alpha \in s_1$); finally *go_forward*.
- *goto_player(P,$\alpha$):* First execute *turn($\beta$)* such that $\vec{o}_p := \binom{x_P - x_p}{y_P - y_p}$, then $turn(\alpha)$ (TD-Q: $\alpha \in s_2$) and finally *go_forward*. Here $(P, p \in T_{east} \vee P, p \in T_{west}) \wedge P \neq p$.
- *goto_opponent(O,$\alpha$):* First execute *turn($\beta$)* such that $\vec{o}_p := \binom{x_O - x_p}{y_O - y_p}$, then $turn(\alpha)$ (TD-Q: $\alpha \in s_2$) and finally *go_forward*. Here $(p \in T_{east} \wedge O \in T_{west}) \vee (p \in T_{west} \wedge O \in T_{east})$.

- *pass_to_player(P):* First execute *turn(β)* such that $\vec{o}_p := \binom{x_P - x_p}{y_P - y_p}$, then *shoot*. Here $P, p \in T_{east} \vee P, p \in T_{west}$. Initial ball speed is set to $v_b^{init} = 0.005 + \sqrt{2 \cdot 0.005 \cdot dist(c_p, c_P)}$. This ensures that the ball will arrive at $c_P$ at a slow speed. If $v_b^{init} > 0.12$ then $v_b^{init} := 0.12$.

- *shoot_to_goal:* First execute *turn_to_goal*, then *shoot*, where initial ball speed is set to $v_b^{init} = 0.005 + \sqrt{2 \cdot 0.005 \cdot dist(c_p, m_g)}$, where $m_g = m_{ge}$ if $p \in T_{west}$ and $m_g = m_{gw}$ if $p \in T_{east}$. If $v_b^{init} > 0.12$ then $v_b^{init} := 0.12$.

# 3 Probabilistic Incremental Program Evolution (PIPE)

In some of our simulations we use Probabilistic Incremental Program Evolution (PIPE) to synthesize programs which, given player $p$'s input vector $\vec{i}(p,t)$, select actions from *ASET*. In simple simulations we set $ASET := ASET_S$ and $\vec{i}(p,t) := \vec{i}_s(p,t)$. In complex simulations we set $ASET := ASET_C$ and $\vec{i}(p,t) := \vec{i}_c(p,t)$. We use PIPE as described in (Sałustowicz and Schmidhuber, 1997), except for "elitist learning" which we omit due to high environmental stochasticity.

A PIPE alternative for searching program space would be genetic programming (GP) (Cramer, 1985; Dickmanns et al., 1987; Koza, 1992). We chose PIPE over GP because it compared favorably with Koza's GP variant in previous experiments (Sałustowicz and Schmidhuber, 1997).

**Action Selection.** Action selection depends on 5 (8) variables when simple (complex) actions are used: $g \in I\!R$, $A_i \in I\!R$, $\forall i \in ASET$. 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.

## 3.1 Basic Data Structures and Procedures

**Programs.** In simple simulations a main program PROGRAM consists of a program $\text{PROG}^g$ which computes the "greediness" parameter $g$ and 4 "action programs" $\text{PROG}^i$ ($i \in ASET_S$). In complex simulations we need $\text{PROG}^g$, 7 action programs $\text{PROG}^i$ ($i \in ASET_C$), programs $\text{PROG}^{i\alpha}$ for each angle parameter, programs $\text{PROG}^{iP}$ for each player parameter and programs $\text{PROG}^{iO}$ for each opponent parameter (for actions using these parameters). The result of applying PROG to data $x$ is denoted PROG($x$). Given $\vec{i}(p,t)$, $\text{PROG}^i(\vec{i}(p,t))$ returns $A_i$ and $g := |\text{PROG}^g(\vec{i}(p,t))|$. An action $i \in ASET$ is then selected according to (1). In the case of complex actions programs $\text{PROG}^{i\alpha}(\vec{i}(p,t))$, $\text{PROG}^{iP}(\vec{i}(p,t))$ and $\text{PROG}^{iO}(\vec{i}(p,t))$ return values for all parameters of action $i$: $\alpha := \text{PROG}^{i\alpha}(\vec{i}(p,t))$, $P := 1 + (|round(\text{PROG}^{iP}(\vec{i}(p,t))) \mid mod\ (Z-1))$, $O := 1 + (|round(\text{PROG}^{iO}(\vec{i}(p,t))) \mid mod\ Z)$. Recall that $Z$ is the number of players per team.

All programs PROG$^i$, PROG$^{i\alpha}$, PROG$^{iP}$, PROG$^{iO}$ are generated according to *probabilistic prototype trees* $PPT^i$, $PPT^{i\alpha}$, $PPT^{iP}$, $PPT^{iO}$ respectively. In what follows we explain how a program PROG $\in$ {PROG$^i$, PROG$^{i\alpha}$, PROG$^{iP}$, PROG$^{iO}$} is generated from the corresponding probabilistic prototype tree $PPT \in \{PPT^i, PPT^{i\alpha}, PPT^{iP}, PPT^{iO}\}$.

**Program Instructions.** A program PROG contains instructions from a function set $F = \{f_1, f_2, \ldots, f_k\}$ with $k$ functions and a terminal set $T = \{t_1, t_2, \ldots, t_l\}$ with $l$ terminals. We use $F = \{+, -, *, \%, sin, cos, exp, rlog\}$ and $T = \{\vec{i}(p,t)_1, \ldots, \vec{i}(p,t)_v, R\}$, where % denotes protected division ($\forall y, z \in \mathbb{R}, z \neq 0$: $y\%z = y/z$ and $y\%0 = 1$), $rlog$ denotes protected logarithm ($\forall y \in \mathbb{R}, y \neq 0$: $rlog(y)$=log(abs($y$)) and $rlog(0) = 0$), $\vec{i}(p,t)_i$ $1 \leq i \leq v$ denotes component $i$ of a vector $\vec{i}(p,t)$ with $v$ components and $R$ represents a generic random constant $\in$ [0;1) (see also "ephemeral random constant" (Koza, 1992)).

**Program Representation.** Programs are encoded in $n$-ary trees, with $n$ being the maximal number of function arguments. Each argument is calculated by a subtree. The trees are parsed depth first from left to right.

**Probabilistic Prototype Tree.** A probabilistic prototype tree ($PPT$) is generally a *complete* $n$-ary tree. At each node $N_{d,w}$ it contains a random constant $R_{d,w}$ and a variable probability vector $\vec{P}_{d,w}$, where $d \geq 0$ denotes the node's depth (root node has $d = 0$) and $w$ defines the node's horizontal position when tree nodes with equal depth are read from left to right ($0 \leq w < n^d$). The probability vectors $\vec{P}_{d,w}$ have $l + k$ components. Each component $P_{d,w}(I)$ denotes the probability of choosing instruction $I \in F \cup T$ at $N_{d,w}$. We maintain $\sum_{I \in F \cup T} P_{d,w}(I) = 1$.

**Program Generation.** To generate a program PROG from $PPT$, an instruction $I \in F \cup T$ is selected with probability $P_{d,w}(I)$ for each accessed node $N_{d,w}$ of $PPT$. This instruction is denoted $I_{d,w}$. Nodes are accessed in a depth first way, starting at the root node $N_{0,0}$, and traversing $PPT$ from left to right. Once $I_{d,w} \in F$ is selected, a subtree is created for each argument of $I_{d,w}$. If $I_{d,w} = R$, then an instance of $R$, called $V_{d,w}(R)$, replaces $R$ in PROG. If $P_{d,w}(R)$ exceeds a threshold $T_R$, then $V_{d,w}(R) = R_{d,w}$. Otherwise $V_{d,w}(R)$ is randomly generated.

**Tree Shaping.** To reduce memory requirements we incrementally grow and prune $PPT$s.

***Growing.*** Initially the $PPT$ contains only the root node. Nodes are created "on demand" whenever $I_{d,w} \in F$ is selected and the subtree for an argument of $I_{d,w}$ is missing.

***Pruning.*** We prune subtrees of a $PPT$ attached to nodes which contain at least one probability vector component above a threshold $T_P$. In case of functions we prune only subtrees that are *not* required as function arguments. Pruning also tends to discard old probability distributions that are irrelevant by now.

## 3.2 Learning

PIPE attempts to find better and better programs. Program quality is measured by a scalar, real-valued "fitness value". PIPE guides its search to promising search space areas by incrementally building on previous solutions. It generates successive program populations according to the underlying probabilistic prototype trees ($PPT$s) and stores in those

trees the knowledge gained from evaluating the programs. In what follows we show how the *PPT*s are adapted to synthesize better and better programs.

***PPT* Initialization.** For all *PPT*'s, each *PPT* node $N_{d,w}$ requires an initial random constant $R_{d,w}$ and an initial probability $P_{d,w}(I)$ for each instruction $I \in F \cup T$. We pick $R_{d,w}$ uniformly random from the interval [0;1). To initialize instruction probabilities we use a constant probability $P_T$ for selecting an instruction from $T$ and $(1 - P_T)$ for selecting an instruction from $F$. $\vec{P}_{d,w}$ is then initialized as follows:

$$P_{d,w}(I) := \frac{P_T}{l}, \quad \forall I : I \in T \qquad \text{and} \qquad P_{d,w}(I) := \tfrac{1-P_T}{k}, \quad \forall I : I \in F \tag{2}$$

**Generation-Based Learning.** PIPE learns in successive generations, each comprising 5 distinct phases: (1) creation of program population, (2) population evaluation, (3) learning from population, (4) mutation of prototype trees and (5) prototype tree pruning.

***(1) Creation of Program Population.*** A population of programs $\text{PROGRAM}_j$ ($0 < j \leq PS$; $PS$ is population size) is generated using the prototype trees as described in Section 3.1. All *PPT*s are grown "on demand".

***(2) Population Evaluation.*** Each program $\text{PROGRAM}_j$ of the current population is evaluated and assigned a non-negative "fitness value" $FIT(\text{PROGRAM}_j)$. To evaluate a program we play one entire soccer game. We define *FIT(*$\text{PROGRAM}_j$*) = 100 - number of goals scored by learner + number of goals scored by opponent.* The offset 100 is sufficient to ensure a positive score difference needed by the learning algorithm (see below). If $FIT(\text{PROGRAM}_j) < FIT(\text{PROGRAM}_i)$, then program $\text{PROGRAM}_j$ is said to embody a better solution than program $\text{PROGRAM}_i$. Among programs with equal fitness we prefer shorter ones (Occam's razor), as measured by number of nodes. We define $b$ to be the index of the *best* program of the current generation and preserve the best program found so far in $\text{PROGRAM}^{el}$ (elitist).

***(3) Learning from Population.*** Prototype tree probabilities are modified such that the probabilities $P(\text{PROG}_b^{part})$ of creating each $\text{PROG}_b^{part} \in \text{PROGRAM}_b$ increase, where $part = \{i, i\alpha, iP, iO\}$. Our experiments indicate that it is beneficial to increase $P(\text{PROG}_b^{part})$ *regardless* of $\text{PROG}_b^{part}$'s length. To compute $P(\text{PROG}_b^{part})$ we look at all $PPT^{part}$ nodes $N_{d,w}^{part}$ used to generate $\text{PROG}_b^{part}$:

$$P(\text{PROG}_b^{part}) = \prod_{d,w:N_{d,w}^{part} \text{ used to generate } \text{PROG}_b^{part}} P_{d,w}(I_{d,w}(\text{PROG}_b^{part})), \tag{3}$$

where $I_{d,w}(\text{PROG}_b^{part})$ denotes the instruction of program $\text{PROG}_b^{part}$ at node position $d, w$. Then we calculate a target probability $P_{TARGET}^{part}$ for each $\text{PROG}_b^{part}$:

$$P_{TARGET}^{part} = P(\text{PROG}_b^{part}) + (1 - P(\text{PROG}_b^{part})) \cdot lr \cdot \frac{\varepsilon + FIT(\text{PROGRAM}^{el})}{\varepsilon + FIT(\text{PROGRAM}_b)}. \tag{4}$$

Here $lr$ is a constant learning rate and $\varepsilon$ a user defined constant. The fraction $\frac{\varepsilon+FIT(\text{PROGRAM}^{el})}{\varepsilon+FIT(\text{PROGRAM}_b)}$ enables *fitness dependent learning (fdl).* We learn more from programs with higher quality

(lower fitness) than from programs with lower quality (higher fitness). Constant $\varepsilon$ determines the degree of *fdl*'s influence. If $\forall$ $FIT(\textsc{Program}^{el})$: $\varepsilon << FIT(\textsc{Program}^{el})$, then PIPE can use small population sizes, as generations containing only low-quality individuals do not affect the $PPT$ much. Even learning with *only one* program per generation is then possible.

Given $P^{part}_{TARGET}$, *all* single node probabilities $P_{d,w}(I_{d,w}(\textsc{Prog}^{part}_b))$ are increased iteratively (in parallel):

REPEAT UNTIL $P(\textsc{Prog}^{part}_b) \geq P^{part}_{TARGET}$ :
$P_{d,w}(I_{d,w}(\textsc{Prog}^{part}_b)) := P_{d,w}(I_{d,w}(\textsc{Prog}^{part}_b)) + c^{lr} \cdot lr \cdot (1 - P_{d,w}(I_{d,w}(\textsc{Prog}^{part}_b)))$

Here $c^{lr}$ is a constant influencing the number of iterations. We use $c^{lr} = 0.1$, which turned out to be a good compromise between precision and speed.

Finally each random constant in $\textsc{Prog}^{part}_b$ is copied to the appropriate node in $PPT^{part}$: if $I_{d,w}(\textsc{Prog}^{part}_b) = R$ then $R^{part}_{d,w} := V^{part}_{d,w}(R)$.

***(4) Mutation of Prototype Trees.*** Mutation is PIPE's major exploration mechanism. Mutation of probabilities in all $PPT$s is guided by the current best solution $\textsc{Program}_b$. We want to explore the area "around" $\textsc{Program}_b$. Probabilities $P^{part}_{d,w}(I)$ stored in all nodes $N^{part}_{d,w}$ that were accessed to generate program $\textsc{Program}_b$ are mutated with a probability $P^{part}_{M_p}$, defined as:

$$P^{part}_{M_p} = \frac{P_M}{(l+k) \cdot \sqrt{|\textsc{Prog}^{part}_b|}}, \tag{5}$$

where $P_M$ is a free parameter setting the overall mutation probability and $|\textsc{Prog}^{part}_b|$ denotes the number of nodes in program $\textsc{Prog}^{part}_b$. The justification of the square root in equation (5) is empirical: we found that larger programs improve faster with a higher mutation rate. Selected probability vector components are mutated as follows:

$$P^{part}_{d,w}(I) := P^{part}_{d,w}(I) + mr \cdot (1 - P^{part}_{d,w}(I)), \tag{6}$$

where $mr$ is the mutation rate, another free parameter. All mutated vectors $\vec{P}^{part}_{d,w}$ are then renormalized.

We see from assignment (6) that small probabilities (close to 0) are subject to stronger mutations than high probabilities. Otherwise mutations would tend to have little effect on the next generation.

***(5) Prototype Tree Pruning.*** At the end of each generation we prune all prototype trees as described in Section 3.1.

# 4 TD-Q Learning

One of the most widely known and promising EF-based approaches to reinforcement learning is TD-Q learning. We use Lin's popular and successful TD($\lambda$) Q-variant (Lin, 1993).

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 $\vec{i}(p,t)$ to action evaluations $Q(\vec{i}(p,t),a_1),\ldots,$Q$(\vec{i}(p,t),a_N)$, where N denotes the number of possible actions. To save free parameters 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.

**Simple action selection.** In simple simulations we use a different 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 $\vec{i}(p,t)$ is

$$Q(\vec{i}(p,t),a_k)=\sum_{i=1}^{i=v} w_i^k \vec{i}(p,t)_i + w_{v+1}^k, \tag{7}$$

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

**Complex action selection.** Since complex actions may have 0, 1, or 2 parameters we use a natural, modular, tree-based architecture that allows for reducing the number of action evaluations. Instead of using continuous angles we use discrete angles (see Section 2). The root node contains networks $N^{a_1},\ldots,N^{a_7}$ for evaluating "abstract" complex actions neglecting the parameters, e.g., *pass_to_player*. Some specific root-network $N^{a_k}$'s "son networks" $N^{a_{k_1}},\ldots,N^{a_{k_{P1(a_k)}}}$ are then used for selecting the first parameter, where $P1(a_k)$ denotes the number of possible discrete values of action $a_k$'s first parameter. Similarly, second parameters are selected using "grandson networks". For instance, if an action contains both player and angle parameters, then there are "son networks" for player-parameters and "grandson networks" for angle parameters. The complete tree contains 159 linear networks.

After computing the 7 "abstract" complex action Q-values according to equation (7), one of the 7 is selected according to the Boltzmann-Gibbs rule (see assignment (1)). If the selected action requires parameters we use equation (7) to compute the Q-values of all possible first parameters and select one according to the Boltzmann-Gibbs rule (see assignment (1)). Similarly we select the second parameter if there is any.

**TD-Q learning.** For both simple and complex simulations we use Lin's TD($\lambda$) Q-variant (Lin, 1993). 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. The trial stops once one of the teams scores or the game is over. Since some player may have scored before it was another player's turn, at trial end some players' time-pointers may differ (by at most 1). Denote player $p$'s final time-pointer by $t^*(p)$. We want the Q-value $Q(\vec{i}(p,t),a_k)$ of selecting action $a_k$ given input $\vec{i}(p,t)$ to approximate

$$Q(\vec{i}(p,t),a_k)\sim\mathcal{E}(\gamma^{t^*(p)-t(p)}R(t^*(p)), \tag{8}$$

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 $H_{max}$. At trial end player $p$'s history list $H(p)$ is
$\{\{\vec{i}(p,t^1(p)), a_{t^1(p)}, V'(\vec{i}(p,t^1(p)))\}, \ldots, \{\vec{i}(p,t^*(p)), a_{t^*(p)}, V'(\vec{i}(p,t^*(p)))\}\}$.
Here $V'(\vec{i}(p,t)) := Max_k\{Q(\vec{i}(p,t^*(p)), a_k)\}$, $t^1(p) := t^c$, if $t^*(p) < H_{max}$, and $t^1(p) := t^*(p) - H_{max} + 1$ otherwise ($t^1(p)$ denotes the start of the history list). To evaluate the selected complex action parameters we store them in the history list as well. Their evaluations are updated just like the Q-values of the "abstract" complex actions, but their Q-values are not used for updates of other previously selected actions (or action parameters).

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 $\vec{i}(p,t)$ $(t = t^1(p), \ldots, t^*(p))$ as follows:

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

$\lambda$ 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 fist entries have been processed we start processing the second entries, etc. The nets are trained using the delta-rule (Widrow and Hoff, 1960) with learning rate $Lr^N$.

# 5 Experiments

We conduct two different types of simulations - simple and complex. During simple simulations we use simple input vectors $\vec{i}_s(t,p)$ and simple actions from $ASET_S$. During complex simulations we use complex input vectors $\vec{i}_c(t,p)$ and complex actions from $ASET_C$. In simple simulations we analyze TD-Q's and PIPE's behavior as we vary team size. In complex simulations we study both algorithms' performances in case of more sophisticated action sets and more informative inputs. Informative inputs are meant to decrease POP's significance. On the other hand, they increase the number of adaptive parameters.

To obtain statistically significant results we perform 10 independent runs for each combination of simulation type, learning algorithm and team size.

## 5.1 Experiments with simple actions

**Experiment 1.** To keep the number of cycles (and training examples) per simulation constant as team size is varied, we play 3300 (1100, 300) games for 1 (3, 11) players. Each game takes $t_{end} = 5000$ time steps. Every 100 games (50 in the 11 player case) 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 simple actions from $ASET_S$. *BRO* is not a bad player due to the initial bias in the action set. For instance, *BRO* greatly prefers shooting at the opponent's goal over shooting at its own. 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, 44.5 to 0.1 goals for team size 3, 108.6 to 0.5 goals for team size 11.

**PIPE Set-up.** Parameters for all PIPE runs are: $P_T$=0.8, $\varepsilon = 1$, $PS$=10, $lr$=0.2, $P_M$=0.1, $mr$=0.2, $T_R$=0.3, $T_P$=0.999999. During performance evaluations we test the current best-of-generation program (except for the first evaluation where we test a random program).

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

**Results.** We compare average score ratios achieved during all the test phases. If at least one goal occurs then the score ratio is $\frac{learner\ score}{learner\ score\ +\ opponent\ score}$ (0.5 otherwise). Figure 5 summarizes results for PIPE and TD-Q. It plots score ratios against number of player actions. PIPE learns faster than TD-Q. Both algorithms learn slightly better with

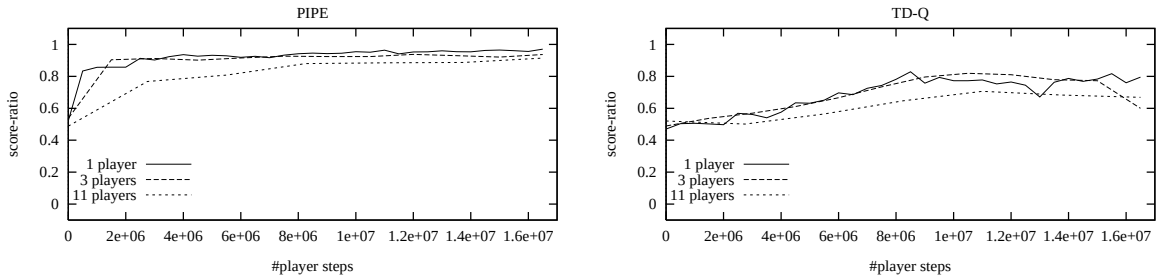


Figure 5: *Average score ratios for PIPE (left) and TD-Q (right) plotted against number of player actions.*

small teams. There are several possible explanations. (1) Learning gets harder as POP and ACAP get worse with increasing team size. (2) Although the number of training examples per run remains constant, increasing team size may lead to less information per training example.

**Experiment 2.** Now we play 3300 games of length $t_{end} = 5000$ for *all* team sizes (1, 3 and 11). Figure 6 summarizes the score ratios. PIPE always quickly learns an appropriate policy *regardless* of team size. In the long run TD-Q learns better with single agents than with multiple agents. The 11 player TD-Q run exhibits an abnormality: the score ratio steadily increases until performance suddenly breaks down (see explanation below).

For all simulations both algorithms start with average score ratios around 0.5.

For team size 1 TD-Q eventually increases the average score ratio to 0.83, PIPE to 0.97. The best ratios ever achieved by both PIPE and TD-Q are 1.0 (not shown). TD-Q's average TD-error decreases constantly from 0.072 to 0.046.

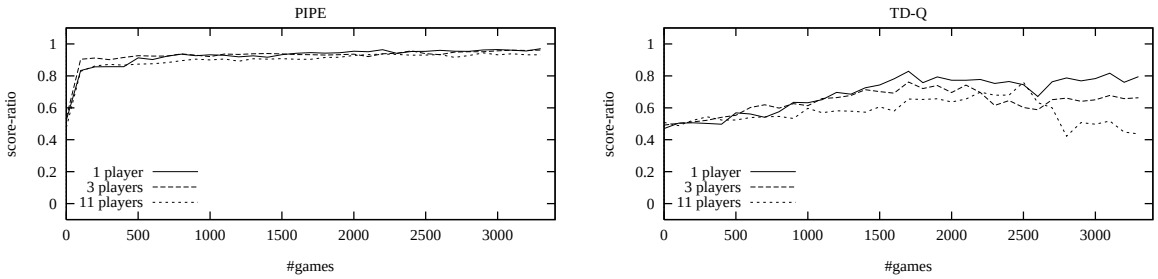


Figure 6: *Average score ratios for PIPE (left) and TD-Q (right) plotted against number of games.*

For team size 3 TD-Q eventually increases the average score ratio to 0.76, PIPE to 0.96. TD-Q's (PIPE's) best ratio of all time is 0.94 (1.0) (not shown). TD-Q's average TD-error decreases from 0.078 to 0.065 (but is at a minimum of 0.064 after 2000 games).

For team size 11 TD-Q eventually increases the average score ratio to 0.76, PIPE to 0.94. TD-Q's (PIPE's) best ratio of all time is 0.91 (1.0) (not shown). TD-Q's average TD-error decreases from 0.080 to 0.067 (but was slightly higher around 2500 games).

To get more insight into what's going on we plot goals scored by learner and opponent against number of games in Figure 7. PIPE's score differences continually increase. TD-Q's 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.

For team size 1 TD-Q initially scores 28.1 goals on average, its opponent 31.9. TD-Q's maximal average score difference is 52.1 - 10.3 = 41.8 goals after 1700 games. Instead of learning to increase the number of own goals, within 3300 games TD-Q learns to reduce the number of opponent goals down to 2.9 (compared to 16.2 own goals). PIPE initially scores 29.4 goals on average, its opponent 27.5. PIPE's maximal average score difference is 319.6 - 10.0 = 309.6 goals after 3300 games.

For team size 3 TD-Q initially scores 43.7 goals on average, its opponent 45. TD-Q's maximal average score difference is 101.8 - 31.7 = 70.1 goals after 1700 games. PIPE initially scores 49.3 goals on average, its opponent 42.1 PIPE's maximal average score difference is 372.8 - 13.9 = 358.9 goals after 3300 games.

For team size 11, both TD-Q curves in Figure 7 diverge initially, but at some point (around 2500 games) performance breaks down again. Initially TD-Q scores 88.9 goals on average, its opponent 86. TD-Q's maximal average score difference is 212.1 - 57.8 = 154.3 goals after 2500 games. PIPE initially scores 73.9 goals on average, its opponent 81.3. PIPE's maximal average score difference is 512.4 - 31.3 = 481.1 goals after 3100 games.

Figure 7 also shows that scoring becomes easier with increasing team size, even for the random system (beginning of learning) and *BRO*. This is partly due to the fact that the larger the team the higher the probability that at least one player is close to the ball.

**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 conducted 5 runs with it, testing it every 50 games. Figure 8 shows the details. Analyzing

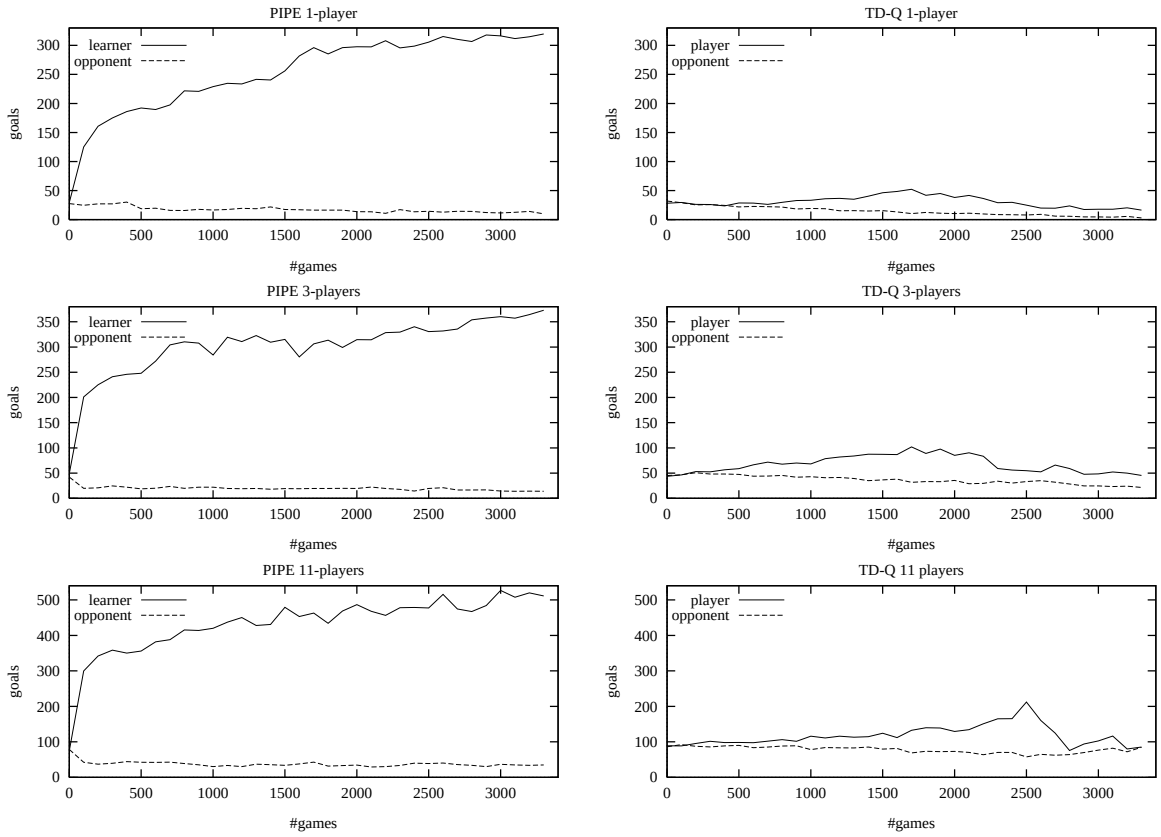


Figure 7: *Average number of goals scored during all test phases, for team sizes 1, 3, 11.*

the runs with our simulator we 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 may not be a big 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, without stifling the effects caused by relatively equal Q-value assignments to actions.

Increasing the greediness value tends to help a bit since this focuses reinforcements on the best actions (although high greediness values do not work well either). Another yet untried option might be to use a pocket algorithm-like method that stores good EFs and backtracks once performance decreases (e.g., the success-story algorithm (Wiering and Schmidhuber, 1996; Schmidhuber and Zhao, 1997)).

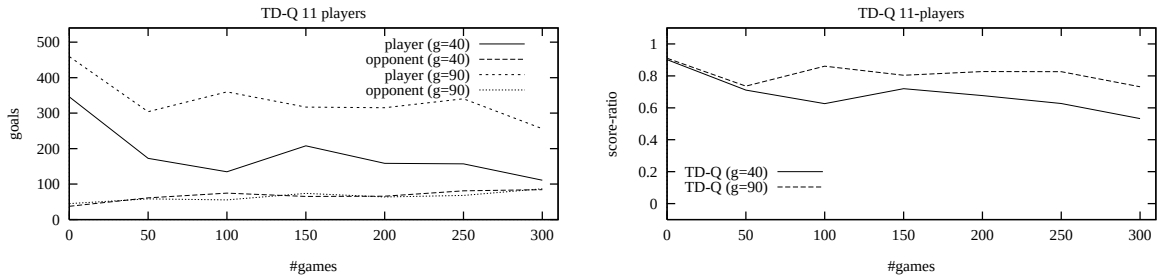


Figure 8: *Performance breakdown study: Left: Average number of goals for simple actions with 11 players starting with already trained, good TD-Q networks with different values for the greediness value g (means of 5 runs). Right: Corresponding score ratios.*

## 5.2 Experiments with complex actions

Now we focus on team size 11. One run with complex actions consists of 250 games, each lasting for $t_{end} = 2000$ time steps. We let both algorithms learn against the "biased random opponent" *BRO*. Every 10 games we test current performance by playing 5 test games (no learning) against *BRO* and summing the score results.

How good is *BRO*? If we let *BRO* play against a non-acting opponent *NO* for five 2000 time step games (all *NO* can do is block), then *BRO* wins against *NO* with on average 14.4 to 0.2. goals (2.84 goals/game; mean of 10 simulations).

**PIPE Set-up.** Parameters for all PIPE runs are the same as used in the experiments with simple actions.

**TD-Q Set-up.** Parameters for TD-Q runs with complex actions are: $\gamma$=0.99, $Lr^N$=0.001, $H_{max}$=100, $\lambda$ is linearly decreased from 1.0 to 0.2. All network weights are randomly initialized in $[-0.01, 0.01]$. During each run the Boltzmann-Gibbs rule's greediness parameter $g$ is linearly increased from 0 to 30.

**Results.** See Figure 9. TD-Q starts out with a ratio of 0.3 and increases this to a maximum of 0.53. PIPE starts out with a ratio of 0.58 and increases this to a maximum of 0.98. PIPE's initial ratio is higher than 0.5, because we test the best program of the first generation, not a random one. TD-Q's (PIPE's) best ratio of all time is 0.88 (1.0) (not shown).

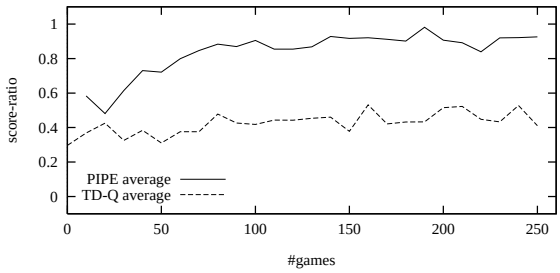


Figure 9: *Average score ratios with 11 players and complex actions.*

Figure 10 shows the average total number of goals scored by learner and opponent

during all test phases.

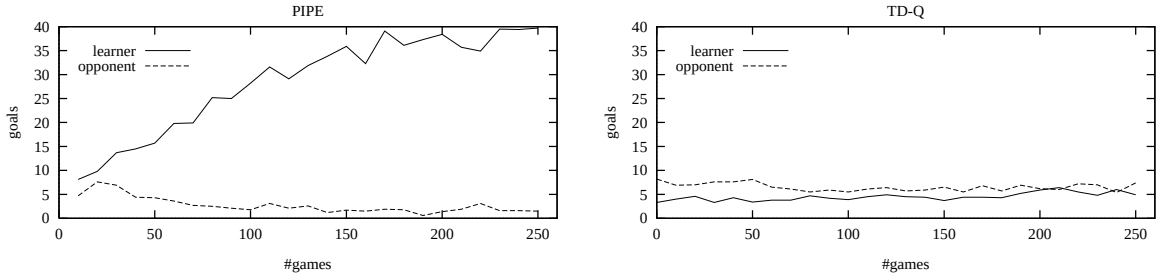


Figure 10: *Average number of goals over 10 independent learning runs for PIPE (left) and TD-Q (right) using complex actions.*

TD-Q initially is worse than its opponent — TD-Q scores 3.3 goals on average, the opponent 8.2 goals. TD-Q wins with a maximal average score difference of only 6.0 - 5.4 = 0.6 goals after 240 games. For the PIPE run both curves diverge clearly. Initially PIPE scores 8.1 goals on average, the opponent 4.7. PIPE wins with a maximal average score difference of 39.7 - 1.5 = 38.2 goals after 250 games.

Complex actions embody stronger initial bias. They allow for cooperation and better optimal strategies. PIPE is able to exploit this. TD-Q is not, although we tried hard to come up with a good TD-Q variant. For instance, to improve TD-Q we tried various locality-enforcing heuristics, such as letting learning rate depend on the distance to the ball, or presenting training examples in different order. This did not work well either though.

In principle, increasing the TD nets' expressive power by adding hidden units might help to store more context information. Since, however, the introduction of hidden units multiplies simulation time by a significant factor, we did not try them.

# 6 Discussion

In a simulated soccer case study with policy-sharing agents we compared a direct policy search method (PIPE) and an EF-based one (TD-Q). Both competed with a biased random opponent. PIPE easily learned to beat this opponent. TD-Q achieved performance improvement, too, but its results were less exciting, especially in case of multiple agents per team.

TD-Q's problems are due to a combination of several reasons. **(1)** *Partial observability.* Q-learning assumes that the environment is fully observable; otherwise it is not guaranteed to work. Still, Q-learning variants already have been successfully applied to partially observable environments, e.g., (Crites and Barto, 1996). The POPs in our soccer simulations, however, seem too severe for the linear networks. **(2)** *Too many trainable parameters* (variance in the "bias-variance dilemma" (Geman et al., 1992) too high — more training games are needed). **(3)** *Agent credit assignment problem (ACAP)* (Weiss, 1996; Versino and Gambardella, 1997): how much did some agent contribute to team performance? ACAP

is particularly difficult in the case of multiagent soccer. For instance, a particular agent may do something truly useful and score. Then all the other agents will receive reward, too. Now the TD nets will have to learn an evaluation function (EF) mapping input-action pairs to expected discounted rewards based on experiences with player actions that have little or nothing to do with the final reward signal. This problem is actually independent of whether policies are shared or not. Player-dependent history lists also do not contribute much to solving ACAP (see next issue). **(4)** *Outliers.* Using player-dependent history lists, each player learns to evaluate actions given inputs by computing updates based on its own TD-return signal. The players collectively update their shared EF to model outliers (novel game situations). Collective updates, however, can lead to significant "shifts in policy-space" and to "unlearning" of previous knowledge. This may lead to performance breakdowns, and makes it hard to learn correct EFs.

Our multiagent scenario seems complex enough to prevent standard EF learning techniques from working efficiently. In principle, however, EFs are not necessary to find good or optimal policies. Sometimes, particularly in the presence of POPs, it makes more sense to search policy space directly instead of spending too much time on fine-tuning EFs (Wiering and Schmidhuber, 1996). That's what PIPE does. Currently PIPE-like, EF-independent techniques seem more promising for complex multiagent learning scenarios, unless methods for overcoming TD-Q's problems are developed.

An interesting aspect of PIPE is: unlike TD-Q it can learn to map inputs to "greediness values" used in the (Boltzmann-Gibbs) exploration rule. This enables PIPE to pick actions more or less stochastically, thus controlling its own exploration process.

# References


Asada, M., Uchibe, E., Noda, S., Tawaratsumida, S., and Hosoda, K. (1994). A vision-based reinforcement learning for coordination of soccer playing behaviors. In *Proc. of AAAI-94 Workshop on AI and A-life and Entertainment*, pages 16–21.

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

Bertsekas, D. P. (1996). *Neuro-Dynamic Programming.* Athena Scientific, Belmont, MA.

Cramer, N. L. (1985). A representation for the adaptive generation of simple sequential programs. In Grefenstette, J., editor, *Proceedings of an International Conference on Genetic Algorithms and Their Applications*, Hillsdale NJ. Lawrence Erlbaum Associates.

Crites, R. and Barto, A. (1996). Improving elevator performance using reinforcement learning. In Touretzky, D., Mozer, M., and Hasselmo, M., editors, *Advances in Neural Information Processing Systems 8*, pages 1017–1023, Cambridge MA. MIT Press.

Dickmanns, D., Schmidhuber, J. H., and Winklhofer, A. (1987). Der genetische Algorithmus: Eine Implementierung in Prolog. Fortgeschrittenenpraktikum, Institut für Informatik, Lehrstuhl Prof. Radig, Technische Universität München.

Geman, S., Bienenstock, E., and Doursat, R. (1992). Neural networks and the bias/variance dilemma. *Neural Computation*, 4:1–58.

Koza, J. R. (1992). *Genetic Programming – On the Programming of Computers by Means of Natural Selection*. MIT Press.

Levin, L. A. (1973). Universal sequential search problems. *Problems of Information Transmission*, 9(3):265–266.

Levin, L. A. (1984). Randomness conservation inequalities: Information and independence in mathematical theories. *Information and Control*, 61:15–37.

Li, M. and Vitányi, P. M. B. (1993). *An Introduction to Kolmogorov Complexity and its Applications*. Springer.

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

Littman, M. L. (1994). Markov games as a framework for multi-agent reinforcement learning. In Prieditis, A. and Russell, S., editors, *Machine Learning: Proceedings of the Eleventh International Conference*, pages 157–163. Morgan Kaufmann Publishers, San Francisco, CA.

Matsubara, H., Noda, I., and Hiraki, K. (1996). Learning of cooperative actions in multi-agent systems: a case study of pass play in soccer. In *AAAI-96 Spring Symposium on Adaptation, Coevolution and Learning in Multi-agent Systems*, pages 63–67.

Nowlan, S. J. and Hinton, G. E. (1992). Simplifying neural networks by soft weight sharing. *Neural Computation*, 4:173–193.

Sahota, M. (1993). Real-time intelligent behaviour in dynamic environments: Soccer-playing robots. Master's thesis, University of British Columbia.

Sałustowicz, R. P. and Schmidhuber, J. (1997). Probabilistic incremental program evolution. *Evolutionary Computation*, to appear. See ftp://ftp.idsia.ch/pub/rafal/-PIPE.ps.gz.

Schmidhuber, J. (1995). Discovering solutions with low Kolmogorov complexity and high generalization capability. In Prieditis, A. and Russell, S., editors, *Machine Learning: Proceedings of the Twelfth International Conference*, pages 488–496. Morgan Kaufmann Publishers, San Francisco, CA.

Schmidhuber, J. (1997). A general method for incremental self-improvement and multi-agent learning in unrestricted environments. In Yao, X., editor, *Evolutionary Computation: Theory and Applications*. Scientific Publ. Co., Singapore. In press.

Schmidhuber, J. and Zhao, J. (1997). The success-story algorithm for multi-agent reinforcement learning. In Weiss, G., editor, *Distributed Artificial Intelligence meets Machine Learning*. Springer. To appear.

Solomonoff, R. (1986). An application of algorithmic probability to problems in artificial intelligence. In Kanal, L. N. and Lemmer, J. F., editors, *Uncertainty in Artificial Intelligence*, pages 473–491. Elsevier Science Publishers.

Stone, P. and Veloso, M. (1995). Beating a defender in robotic soccer: Memory-based learning of a continuous function. In Tesauro, G., Touretzky, D. S., and Leen, T. K., editors, *Advances in Neural Information Processing Systems 7*. MIT Press, Cambridge MA.

Stone, P. and Veloso, M. (1996). A layered approach to learning client behaviors in the robocup soccer server. Submitted to Applied Artificial Intelligence (AAI) in August 1996.

Versino, C. and Gambardella, L. M. (1997). Learning real team solutions. In Weiss, G., editor, *DAI Meets Machine Learning*, Lecture Notes in Artificial Intelligence. Springer-Verlag. In press.

Watkins, C. (1989). *Learning from Delayed Rewards*. PhD thesis, King's College.

Weiss, G. (1996). Adaptation and learning in multi-agent systems: Some remarks and a bibliography. In Weiss, G. and Sen, S., editors, *Adaptation and Learning in Multi-Agent Systems*, volume 1042, pages 1–21. Springer-Verlag, Lecture Notes in Artificial Intelligence.

Widrow, B. and Hoff, M. E. (1960). Adaptive switching circuits. *1960 IRE WESCON Convention Record*, 4:96–104. New York: IRE. Reprinted in Anderson and Rosenfeld [1988].

Wiering, M. A. and Schmidhuber, J. (1996). Solving POMDPs with Levin search and EIRA. In Saitta, L., editor, *Machine Learning: Proceedings of the Thirteenth International Conference*, pages 534–542. Morgan Kaufmann Publishers, San Francisco, CA.