Machine Learning, , – (1998, in press)

Learning Team Strategies: Soccer Case Studies

RAFAŁ P. SALUSTOWICZ rafal@idsia.ch
MARCO A. WIERING marco@idsia.ch
JÜRGEN SCHMIDHUBER juergen@idsia.ch
IDSIA, Corso Elvezia 36, 6900 Lugano, Switzerland

Editors: Michael Huhns and Gerhard Weiss

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 several learning algorithms: TD-Q learning with linear neural networks (TD-Q), Probabilistic Incremental Program Evolution (PIPE), and a PIPE version that learns by coevolution (CO-PIPE). TD-Q is based on learning evaluation functions (EFs) mapping input/action pairs to expected reward. PIPE and CO-PIPE search policy space directly. They use adaptive probability distributions to synthesize programs that calculate action probabilities from current inputs. Our results show that linear TD-Q encounters several difficulties in learning appropriate shared EFs. PIPE and CO-PIPE, however, do not depend on EFs and find good policies faster and more reliably. This suggests that in some multiagent learning scenarios direct search in policy space can offer advantages over EF-based approaches.

Keywords: Multiagent Reinforcement Learning, Soccer, TD-Q Learning, Evaluation Functions, Probabilistic Incremental Program Evolution, Coevolution.

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 in 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 in a given set of possible actions the current policy determines the conditional probability that the agent will execute this action, 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, e.g., (Nowlan & 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 multiagent reinforcement learning (RL) scenarios delayed reward/punishment is typically given to an entire 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; Crites & Barto, 1996; Versino & Gambardella, 1997).

Evaluation Functions versus Search through Policy Space. There are two rather obvious classes of candidate algorithms for learning shared policies in multiagent RL. Class I includes traditional singleagent RL algorithms based on adaptive evaluation functions (EFs) (Watkins, 1989; Bertsekas & Tsitsiklis, 1996). Usually online variants of dynamic programming and function approximators are combined to learn EFs mapping input-action pairs to expected discounted future reward. The EFs are then exploited to generate rewarding action sequences.

Methods from class II 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 & Vitányi, 1993; Wiering & Schmidhuber, 1996; Schmidhuber, 1997a), Genetic Programming (Cramer, 1985; Dickmanns et al., 1987; Koza, 1992) and Probabilistic Incremental Program Evolution (PIPE, Sałustowicz & Schmidhuber, 1997).

Comparison. In our case study we compare two learning algorithms, each representative of its class: TD-Q learning (Lin, 1993; Peng & Williams, 1996; Wiering & Schmidhuber, 1997) with linear neural networks (TD-Q) and Probabilistic Incremental Program Evolution (PIPE, Sałustowicz & Schmidhuber, 1997). We also report results for a PIPE variant based on coevolution (CO-PIPE, Sałustowicz et al., 1997). We chose TD-Q learning and PIPE because both methods have already been successfully applied to interesting singleagent tasks (Lin, 1993; Sałustowicz & Schmidhuber, 1997) (another reason for choosing TD learning (Sutton, 1988) is its popularity due to a successful application to backgammon (Tesauro, 1994)). Linear TD-Q selects actions according to linear neural networks trained with the delta rule (Widrow & Hoff, 1960) to map player inputs to evaluations of alternative actions. We use linear networks to keep simulation time comparable to that of PIPE and CO-PIPE — more complex approximators would require significantly more computational resources. PIPE and CO-PIPE are based on probability vector coding of program instructions (Schmidhuber, 1997b), Population-Based Incremental Learning (Baluja, 1994; Baluja & Caruana, 1995) and tree coding of programs used in variants of Genetic Programming (Cramer, 1985; Koza, 1992). They synthesize programs that calculate action probabilities from inputs. Experiences with programs are stored in adaptive probability distributions over all possible programs. The probability distributions then 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 & Veloso, 1996a; 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 at *learning* low-level cooperation tasks such as pass play (Stone & Veloso, 1996a; Matsubara et al., 1996; Nadella & Sen,

1996). Recently Stone & Veloso (1996b) mentioned that even team strategies might be learnable by TD(λ) or genetic methods.

Published results on learning entire soccer strategies, however, have been limited to extremely reduced scenarios such as Littman's (1994) tiny 5×4 grid world with two single opponent players[1]. 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. We will include certain results reported in (Sałustowicz et al., 1997a,b).

Results Overview. Our results indicate: linear TD-Q has severe problems in learning and keeping appropriate shared EFs. It learns relatively slowly, and once it achieves fairly good performance it tends to break down. This effect becomes more pronounced as team size increases. PIPE and CO-PIPE learn faster than linear TD-Q and continuously increase their performance. This suggests that PIPE-like, EF-independent techniques can easily be applied to complex multiagent learning scenarios with policy-sharing agents, while more sophisticated and time consuming EF-based approaches may be necessary to overcome TD-Q's current problems.

Outline. Section 2 describes the soccer simulation. Section 3 describes PIPE and CO-PIPE. Section 4 describes TD-Q. Section 5 reports experimental results. Section 6 concludes.

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.

Soccer 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 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 $0.8 < y_b < 1.2 \wedge (x_b < 0 \vee x_b > 4.0)$.

Players. There are two teams consisting of Z homogeneous players $T_{east} = \{pe_1, pe_2, \ldots, pe_Z\}$ and $T_{west} = \{pw_1, pw_2, \ldots, pw_Z\}$. We vary team size: Z can

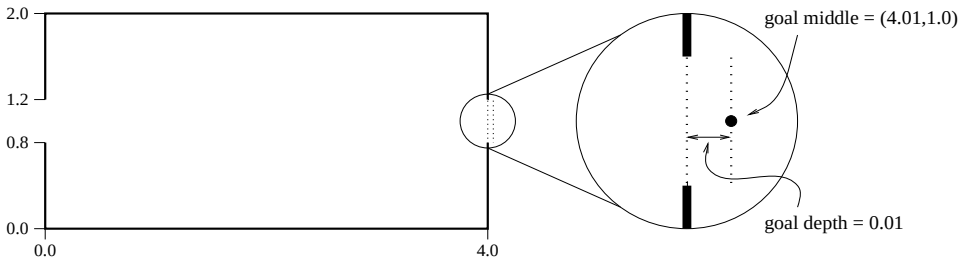

Figure 1. Left: Soccer field. *Right:* Depth and "middle" m_{ge} of east goal (enlarged).

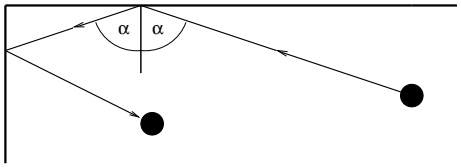

Figure 2. Ball "reflected" by wall.

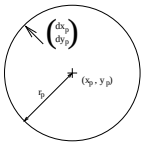

Figure 3. Player: center $c_p = (x_p, y_p)$, radius r_p and orientation $\vec{o}_p = \binom{dx_p}{dy_p}$.

be 1, 3 or 11. 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 $dist(c_{p_i}, c_{p_j}) < r_p$, 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 has owned the ball then the ball will change 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 $\vec{o}_p = \binom{-1}{0}$ $\forall p \in T_{east}$ and $\vec{o}_p = \binom{1}{0}$ $\forall p \in T_{west}$.

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) 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 is a *complex* input vector $\vec{i}_c(p,t)$.

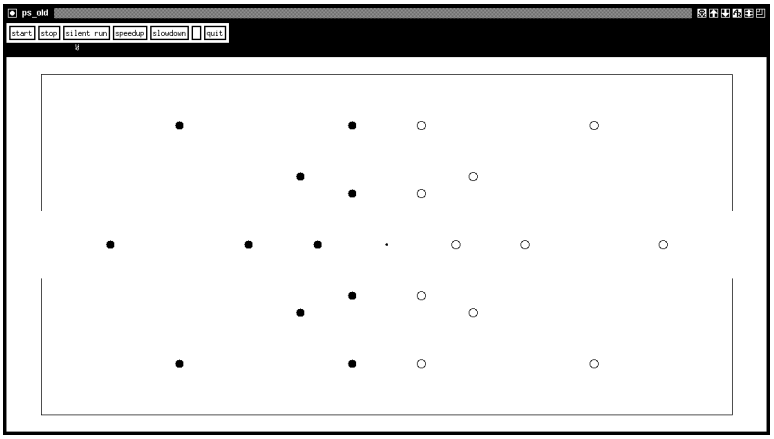

Figure 4. 22 players and ball in initial positions. Players of a 1 or 3 player team are those furthest in the back (defenders and/or goalkeeper).

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 system). (3) Polar coordinates of both goals relative to a 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 are not sufficient to make the environment fully observable — e.g, there is no information about positions of other players.

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. The environment still remains partially observable, however, since the player orientations and changing behaviors are not included in the inputs.

TD-Q's, PIPE's, and CO-PIPE's input representation of distance d (angle α) 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 linear networks.

(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 α_{noise} is picked uniformly random from $-5^\circ \leq \alpha_{noise} \leq 5^\circ$. 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 α 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 α 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(α):* 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(α):* First execute *turn_to_goal*, then $turn(\alpha)$ (TD-Q: $\alpha \in s_1$) and finally *go_forward*.

- *goto_own_goal(α):* First execute *turn(β)* 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,α):* First execute *turn(β)* 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,α):* First execute *turn(β)* 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 $v_b^{init} = 0.005 + \sqrt{2 \cdot 0.005 \cdot dist(c_p, c_P)}$. If $v_b^{init} > 0.12$ then $v_b^{init} := 0.12$. This ensures that the ball will arrive at c_P at a slow speed, if the distance to the player is not larger than 1.5 ("maximal shooting distance").

- *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)

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 & 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 & Schmidhuber, 1997).

Action Selection. Action selection depends on 5 (8) variables when simple (complex) actions are used: the "greediness" parameter $g \in I\!R$, and 4 (7) "action values" $A_a \in I\!R$, $\forall a \in ASET$. Action $a \in ASET$ is selected with probability P_{A_a} according to the Boltzmann-Gibbs distribution at temperature $\frac{1}{g}$:

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

All A_a 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 PROG^g which computes the greediness parameter g and 4 "action programs" PROG^a ($a \in ASET_S$). In complex simulations we need PROG^g, 7 action programs PROG^a ($a \in ASET_C$), programs $\text{PROG}^{a\alpha}$ for each angle parameter, programs PROG^{aP} for each player parameter and programs PROG^{aO} for each opponent parameter (for actions using these parameters). The result of applying PROG to data x is denoted PROG(x). Given $\vec{i}(p,t)$, $\text{PROG}^a(\vec{i}(p,t))$ returns A_a and $g := |\text{PROG}^g(\vec{i}(p,t))|$. An action $a \in ASET$ is then selected according to the Boltzmann-Gibbs rule — see Assignment (1). In the case of complex actions programs $\text{PROG}^{a\alpha}$, PROG^{aP} and PROG^{aO} return values for all parameters of action a: $\alpha := \text{PROG}^{a\alpha}(\vec{i}(p,t))$, $P := 1 + (|round(\text{PROG}^{aP}(\vec{i}(p,t)))| \ mod \ (Z-1))$, $O := 1 + (|round(\text{PROG}^{aO}(\vec{i}(p,t)))| \ mod \ Z)$. Recall that Z is the number of players per team.

All programs PROG^a, $\text{PROG}^{a\alpha}$, PROG^{aP}, and PROG^{aO} are generated according to distinct *probabilistic prototype trees* PPT^a, $PPT^{a\alpha}$, PPT^{aP}, and PPT^{aO}, respectively. The *PPTs* contain adaptive probability distributions over all programs that can be constructed from a predefined instruction set. In what follows we will explain how programs and probabilistic prototype trees are represented and how a program $\text{PROG} \in \{\text{PROG}^a, \text{PROG}^{a\alpha}, \text{PROG}^{aP}, \text{PROG}^{aO}\}$ is generated from the corresponding probabilistic prototype tree $PPT \in \{PPT^a, PPT^{a\alpha}, PPT^{aP}, PPT^{aO}\}$. A more detailed description can be found in (Sałustowicz & Schmidhuber, 1997).

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 $rlog(0) = 0$), $\vec{i}(p,t)_j$ $1 \leq j \leq v$ denotes component j of a vector $\vec{i}(p,t)$ with v components and R represents the *generic random constant* $\in$ [0;1).

Generic Random Constants. A generic random constant (compare also "ephemeral random constant" (Koza, 1992)) is a zero argument function (a terminal). When accessed during program creation, it is either instantiated to a random value from a predefined, problem-dependent set of constants (here: [0;1)) or a value previously stored in the *PPT* (see below).

Program Representation. Programs are encoded in n-ary trees, with n being the maximal number of function arguments. Each nonleaf node encodes a function from F and each leaf node a terminal from T. The number of subtrees each node has corresponds to the number of arguments of its function. 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 as $I_{d,w}$. 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 generated uniformly random from the interval [0;1).

Tree Shaping. A *complete PPT* is infinite. A "large" *PPT* is memory intensive. To reduce memory requirements we incrementally grow and prune *PPT*s.

Growing. Initially a *PPT* contains only the root node (node initialization is described in Section 3.2). Further 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 *PPT* subtrees attached to nodes that 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.

3.2. Learning

We first describe how the *PPT*s are initialized. Then we explain how PIPE guides its search to promising search space areas by incrementally building on previous solutions.

***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 user-defined 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:

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

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 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 PROGRAM_j of the current population is evaluated and assigned a scalar, non-negative "fitness value" $FIT(\text{PROGRAM}_j)$, reflecting the program's performance. To evaluate a program we play one entire soccer game. We define $FIT(\text{PROGRAM}_j)$ = *100 - number of goals scored by* PROGRAM_j *+ 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 PROGRAM_j is said to embody a better solution than program PROGRAM_i. Among programs with equal fitness we prefer shorter ones (Occam's razor), as measured by number of nodes.

(3) Learning from Population. We define b to be the index of the *best* program of the current generation and preserve the best fitness found so far in $FIT(\text{PROGRAM}^{el})$ (fitness of the elitist). 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* $\in \{a, a\alpha, aP, aO\}$. To compute $P(\text{PROG}_b^{part})$ we look at all PPT^{part} nodes $N_{d,w}^{part}$ used to generate PROG_b^{part}:

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

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

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

Here lr is a constant learning rate and ε a positive user-defined constant. The fraction $\frac{\varepsilon + FIT(\text{PROGRAM}^{el})}{\varepsilon + FIT(\text{PROGRAM}_b)}$ implements *fitness dependent learning (fdl).* We take larger steps towards programs with higher quality (lower fitness) than towards programs with lower quality (higher fitness). Constant ε determines the degree of

fdl's influence. If $\forall$ *FIT*(PROGRAM^{el}): $\varepsilon \ll$ *FIT*(PROGRAM^{el}), then PIPE can use small population sizes, as generations containing only low-quality individuals (with *FIT*(PROGRAM$_b$) $\gg$ *FIT*(PROGRAM^{el})) do not affect the *PPT*s much.

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

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

Here c^{lr} is a constant influencing the number of iterations. The smaller c^{lr} the higher the approximation precision of P^{part}_{TARGET} and the number of required iterations. We use $c^{lr} = 0.1$, which turned out to be a good compromise between precision and speed.

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

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

$$P^{part}_{M_p} = \frac{P_M}{(l+k) \cdot \sqrt{|\text{PROG}^{part}_b|}},$$

where the user-defined parameter P_M defines the overall mutation probability and $|\text{PROG}^{part}_b|$ denotes the number of nodes in program PROG^{part}_b. Selected probability vector components are then mutated as follows:

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

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

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

3.3. Coevolution

We also use PIPE to coevolve programs. Each population consists of only two programs with mutually dependent performance. Coevolutionary PIPE (CO-PIPE) works just like PIPE, except that: (1) To evaluate both programs of a population (PROGRAM$_1$ and PROGRAM$_2$) we let them play against each other: *FIT*(PROGRAM$_i$) = *100 - number of goals scored by* PROGRAM$_i$ *+ number of goals scored by* PROGRAM$_j$, where $i, j \in \{1, 2\}$ and $i \neq j$. (2) The next generation consists of the winner and a new program generated according to the adapted *PPT*. (3) Among programs with equal fitness and length we prefer former winners. (4) We do not use *fitness dependent learning*, as the fitness function changes over time.

4. TD-Q Learning

One of the most widely known and promising EF-based approaches to reinforcement learning is TD-Q learning (Sutton, 1988; Watkins, 1989; Peng & Williams, 1996; Wiering & Schmidhuber, 1997). We use an offline TD(λ) Q-variant (Lin, 1993). For efficiency reasons our TD-Q version uses linear neural networks (networks with hidden units require too much simulation time). To implement policy-sharing we use the same networks for all players of a team. 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. 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 network for each of the four actions $\{a_1,\ldots,a_4\}$. To select an action $a(p,t)$ at time t 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

$$Q(\vec{i}(p,t),a_k) := \vec{w}^k \cdot \vec{i}(p,t) + b^k \tag{2}$$

where $\vec{w}^k$ is the weight vector for action network k and b^k is its bias strength. Once all Q-values have been calculated, a single action $a(p,t)$ is chosen according to the Boltzmann-Gibbs rule — see Assignment (1). Unlike PIPE, which evolves the greediness parameter, TD-Q needs an *a priori* value for g.

Complex Action Selection. Since complex actions may have 0, 1, or 2 parameters we use a natural, modular, tree-based architecture. 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 "angle son networks" $N^{a_k}_{\alpha_1},\ldots,N^{a_k}_{\alpha_5}$ are then used for selecting the angle parameter. Similarly, player and opponent parameters are selected using "player son networks" and "opponent son networks", respectively. For instance, if an action contains both player and angle parameters, then there are "son networks" for player-parameters and "son networks" for angle parameters. The complete tree contains 64 linear networks.

After computing the seven "abstract" complex action Q-values according to Equation (2), one of the seven is selected according to the Boltzmann-Gibbs rule — see Assignment (1). If the selected action requires parameters we use Equation (2) to compute the Q-values of all required parameters and select a value for each parameter according to the Boltzmann-Gibbs rule.

TD-Q Learning. For both simple and complex simulations we use an offline TD(λ) Q-variant similar to Lin's (1993). Each game consists of separate trials. At trial start we set time-pointer t to current game time t^c. We increment t after each cycle. The trial stops once one of the teams scores or the game is over. Denote the final time-pointer by t^*. 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

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

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 $R(t^*)$ 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

$$H(p) := \{\{\vec{i}(p,t^1), a(p,t^1), V(\vec{i}(p,t^1))\}, \ldots, \{\vec{i}(p,t^*), a(p,t^*), V(\vec{i}(p,t^*))\}\}.$$

Here $V(\vec{i}(p,t)) := Max_k\{Q(\vec{i}(p,t), a_k)\}$, and t^1 denotes the start of the history list: $t^1 := t^c$, if $t^* < H_{max}$, and $t^1 := t^* - H_{max} + 1$ otherwise.

After each trial we calculate examples using offline TD-Q learning. For each player history list $H(p)$, we compute desired Q-values $Q^{new}(p,t)$ for selecting action $a(p,t)$, given $\vec{i}(p,t)$ $(t = t^1, \ldots, t^*)$ as follows:

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

λ determines future experiences' degree of influence.

To evaluate the selected complex action parameters we store them in history lists as well. Their evaluations are updated on the Q^{new}-values of their (parent) "abstract" complex actions — Q-values of selected action parameters are not used for updates of other previously selected action parameters (or selected actions).

Once all players have created TD-Q training examples, we train the selected networks to minimize their TD-Q errors. All player history-lists are processed by dovetailing 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, and so on. The networks are trained using the delta-rule (Widrow & 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 $\vec{i}_s(p,t)$ and simple actions from $ASET_S$. During complex simulations we use complex input vectors $\vec{i}_c(p,t)$ and complex actions from $ASET_C$. In simple simulations we compare TD-Q's, PIPE's and CO-PIPE's behavior as we vary team size. In complex simulations we study the 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. For a statistical evaluation we perform 10 independent runs for each combination of simulation type, learning algorithm and team size.

5.1. Simple Simulations

We play 3300 games of length $t_{end} = 5000$ for team sizes 1, 3 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 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.

We also designed a simple but good team *GO* by hand. *GO* consists of players which move towards the ball as long as they do not have it, and shoot it at the opponent's goal otherwise. If we let *GO* play against *BRO* for twenty 5000 time step games then *GO* wins with on average 417 to 0 goals for team size 1, 481 to 0 goals for team size 3, and 367 to 3 goals for team size 11. Note that *GO* implements a non-cooperative (singleagent) strategy. Small *GO* teams perform extremely well — larger *GO* teams with many interacting agents, however, do not (see team size 11).

PIPE and CO-PIPE Set-ups. Parameters for all PIPE and CO-PIPE runs are: P_T=0.8, $\varepsilon = 1$, lr=0.2, P_M=0.1, mr=0.2, T_R=0.3, T_P=0.999999. For PIPE we use a population size of PS=10, while for CO-PIPE we use PS=2, as mentioned in Section 3.3. 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. After a coarse search through parameter space we used the following parameters for all TD-Q runs: γ=0.99, lr_n=0.0001, λ=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 differences achieved during all test phases. Figure 5 shows results for PIPE, CO-PIPE, and TD-Q. It plots goals scored by learner and opponent (*BRO*) against number of games used for learning. Larger teams score more frequently because some of their players start out closer to the ball and the opponent's goal.

PIPE learns fastest and always finds quickly an appropriate policy *regardless* of team size. Its score differences continually increase. CO-PIPE performs worse than PIPE, but is still able to find good policies with respect to *BRO*. Note, however, that CO-PIPE's task is more difficult than PIPE's or TD-Q's. It never "sees" *BRO* during training and therefore has no reason for optimizing its strategy against it. Stochastic fluctuations in CO-PIPE's performance tend to level out with increasing team size.

TD-Q also improves, but in a less spectacular way. It always learns more slowly than PIPE and CO-PIPE. It tends to increase score differences until it scores roughly twice as many goals as in the beginning (when actions are still random). Then, however, the score differences start declining. There are several reasons for TD-Q's slowness and breakdown: (1) POP makes learning appropriate EFs diffi-

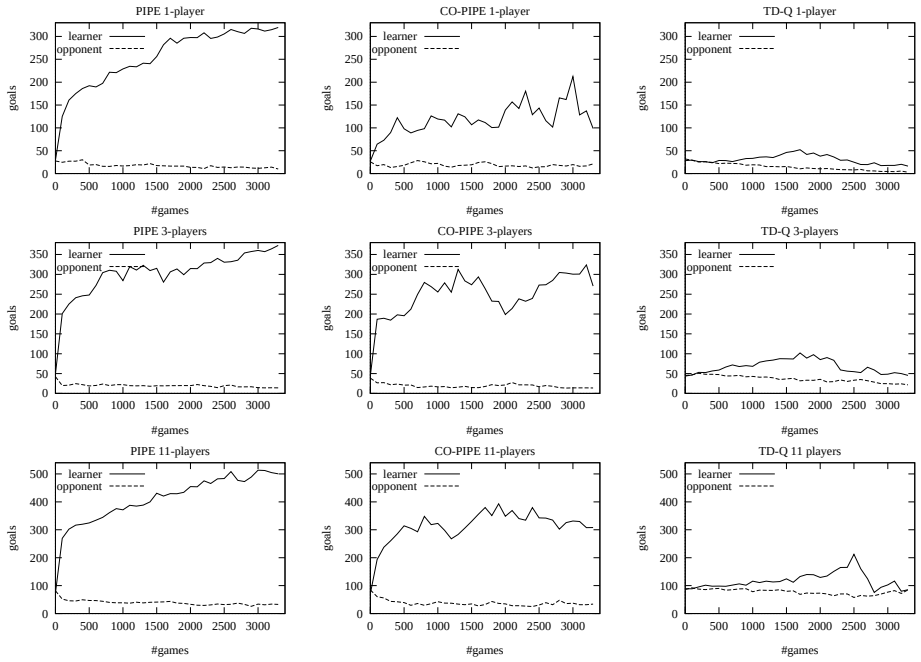

Figure 5. Average number of goals scored during all test phases, for team sizes 1, 3, 11.

cult. (2) The linear neural networks cannot keep useful EFs but tend to unlearn them instead. (3) Unlike PIPE, linear TD-Q suffers from ACAP: it needs to assign proper credit to individual player actions but fails to pick out the truly useful ones. Below we will describe a deeper investigation of a catastrophic performance breakdown in the 11 player TD-Q run.

For each learning algorithm and *GO*, Table 1 lists results against *BRO* (averages over ten runs).

Table 1. Results of PIPE, CO-PIPE, TD-Q, and *GO* playing against *BRO*.

team size		*GO*	PIPE	CO-PIPE	TD-Q
1	**max. score difference**	**417**	**310**	**192**	**42**
	av. goals ± st.d.	417±6	320±42	212±97	52±14
	av. *BRO* goals ± st.d.	0±0	10±7	20±10	10±3
	achieved after games	n.a.	3300	3000	1700
3	**max. score difference**	**481**	**359**	**310**	**70**
	av. goals ± st.d.	481±8	373±86	324±62	102±14
	av. *BRO* goals ± st.d.	0±1	14±6	14±11	32±8
	achieved after games	n.a.	3300	3200	1700
11	**max. score difference**	**364**	**481**	**357**	**154**
	av. goals ± st.d.	367±18	512±129	393±53	212±84
	av. *BRO* goals ± st.d.	3±1	31±23	36±27	58±23
	achieved after games	n.a.	3100	1900	2500

The hand-made *GO* team outperforms (in terms of score difference) any of the 1 and 3 player teams. In the 11 player case, however, it plays worse than PIPE, while CO-PIPE's performance is comparable. This indicates that: (1) Successful singleagent strategies may not suit larger teams. (2) Useful strategies for large teams are learnable by direct policy search.

How do PIPE and CO-PIPE solve the task? Comparing best programs of several successive generations revealed that both algorithms are able to: (1) quickly identify the inputs that are relevant for selecting actions, and (2) find programs that compute useful action probabilities given the selected inputs. PIPE's and CO-PIPE's ability to set the greediness parameter helps to control exploration as it makes action selection more or less stochastic depending on the inputs.

TD-Q's Instability Problems. Some linear TD-Q runs led to good performance. This implies clusters (or niches) in weight vector space that contain good solutions. TD-Q's dynamics, however, do not always lead towards such niches. Furthermore, sudden performance breakdowns hint at a lack of stability of good solutions. To understand TD-Q's problems in the 11 player case we saved a "good" network just before breakdown (after 2300 games). Ten times we continued training it for 25 games, testing it every game by playing twenty test games. To achieve more pronounced score differences we set the greedy parameter g to 90 — this leads to more deterministic behavior than the value 42 used before saving the network.

Figure 6(left) plots the average number of goals scored by TD-Q and *BRO* during all test games against the number of training games. Although initial performance is quite good (the score difference is 418 goals), the network fails to keep it up. To analyze a particular breakdown we focus on a single run. Figure 6(middle) shows the number of goals scored during all test phases, Figure 6(right) the relative frequencies of selected actions.

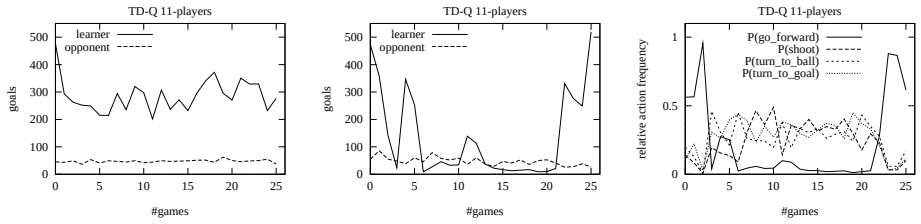

Figure 6. *BRO* against TD-Q starting out with a well-trained linear TD-Q network: performance breakdown study for 11 players. *Left:* average numbers of goals (means of 10 runs). *Middle:* plot for a single run. *Right:* relative frequencies of actions selected by TD-Q during the single run.

Figure 6(middle) shows a performance breakdown occuring within just a few games. It is accompanied by dramatic policy changes displayed in Figure 6(right). Analyzing the learning dynamics we found the following reason for the instability: Since TD-Q's linear networks learn a global EF approximation, they compute an average expected reward for all game constellations. This makes the Q-values of many actions quite similar: their weight vectors differ only slightly in size and direction. Hence small updates can create large policy changes. This becomes more

likely with increasing g, which enhances the effects of small Q-value differences (a large g, however, is necessary to obtain more deterministic policies).

TD-Q does adapt some weight vectors more than others, however. The weight vector of the most frequent action of some rewarding (unrewarding) trial will grow (shrink) most. For example, according to Figures 6(middle) and 6(right), the action *go_forward* is selected most frequently by the initially good policy. Two more games, however, cause the behavior to be dominated by this action. This results in worse performance and provokes a strong correction in the third game. This suddenly makes the action unlikely even in game constellations where it should be selected, and leads to even worse performance. (Note that in this particular case after 25 games the policy performs well again — niches can occasionally be rediscovered.)

For 11 player teams the effect of update steps on the policy is 11-fold (as compared to 1 player teams) and the instability is much more pronounced. We could not get rid of the instability 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.

A promising remedy may be to use action frequency dependent learning rates. This will make update steps for all actions almost equal such that successes/failures will not easily lead to dominating actions. Another remedy may be the pocket algorithm (Gallant, 1993) (which stores the best EF found so far) or the more complex success-story algorithm (Wiering & Schmidhuber, 1996; Schmidhuber et al., 1997a; Schmidhuber et al., 1997b) that backtracks once the reward per time interval decreases.

Yet another promising remedy may be to use local (but more time-consuming) function approximators such as CMACS (Albus, 1975; Sutton, 1996). CMACS-based TD-Q learners will not break down as easily, although our preliminary experiments indicate that in case of simple input vectors they do suffer from the POP. Preliminary CMACS/TD-Q experiments with complex inputs and simple actions, however, already led to promising results.

5.2. Complex Simulations

From now on we focus on team size 11. One run with complex actions and more informative inputs consists of 1200 games, each lasting for $t_{end} = 5000$ time steps. Again we train PIPE and TD-Q against the "biased random opponent" *BRO*, while CO-PIPE learns through coevolution. Every 100 games we test current performance by playing 20 test games (no learning) against *BRO* and summing the score results.

PIPE and CO-PIPE Set-ups. Parameters for all PIPE and CO-PIPE runs are the same as used in simple simulations.

TD-Q Set-up. Parameters for all TD-Q runs are also the same as used in simple simulations with the exception that lr_n=0.001 (several other parameter values led to worse results).

Results. Figure 7 shows the average number of goals scored by PIPE, CO-PIPE, and TD-Q (learners) in comparison to *BRO* (opponent) during all test phases.

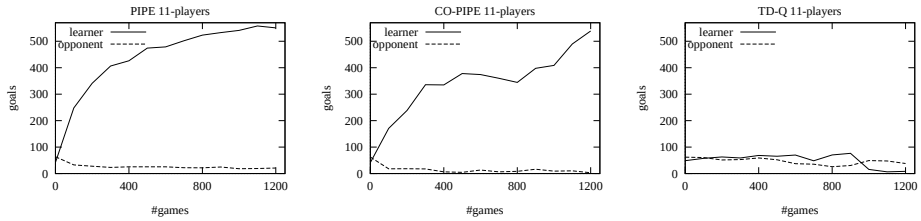

Figure 7. Average number of goals (means of 10 independent runs) for PIPE *(left)*, CO-PIPE *(middle)*, and TD-Q *(right)* vs. *BRO* using complex actions and inputs.

PIPE and CO-PIPE quickly find successful strategies. PIPE's performance steadily increases while CO-PIPE's is slightly more stochastic. In the long run (after 3300 games – not shown), however, both are very similar. Note again, though, that CO-PIPE solves a more difficult task — it is tested against an opponent that it never meets during training.

Linear TD-Q initially does worse than its opponent. It does learn to beat *BRO* by about 50 % but then breaks down completely. Examining all single runs we found that TD-Q's average score results were strongly influenced by a single good run that scored up to 471 goals. Once this run's performance broke down after 1000 games the average declined to 16 goals.

We compare maximal average score differences in Table 2. PIPE and CO-PIPE both achieve score differences that are significantly better than *GO*'s. Linear TD-Q does not.

Table 2. Maximal average score differences against *BRO* for different learning methods and *GO*.

	GO	PIPE	CO-PIPE	TD-Q
max. score difference	**364**	**530**	**536**	**46**
av. goals ± st.d.	367±18	551±215	539±220	76±140
av. *BRO* goals ± st.d.	3±1	21±35	3±4	30±29
achieved after games	n.a.	1200	1200	900

Complex actions embody stronger initial bias and make cooperation easier, while more informative inputs make the POP less severe. In principle, this allows for better soccer strategies. PIPE and CO-PIPE are able to exploit this and perform better than with simple actions (compare Figures 5 and 7 and Tables 1 and 2). Linear TD-Q does not. It still suffers from the problems described in Section 5.1.

6. Conclusion

In a simulated soccer case study with policy-sharing agents we compared direct policy search methods (PIPE and coevolutionary CO-PIPE) and an EF-based one (linear TD-Q). All competed against a biased random opponent (*BRO*). PIPE and CO-PIPE always easily learned to beat this opponent regardless of team size, amount

of information conveyed by the inputs, or complexity of actions. In particular, CO-PIPE outperformed *BRO* without ever meeting it during the training phase. TD-Q achieved performance improvements, too, but its results were less exciting, especially in case of several agents per team, more informative inputs, and more sophisticated actions.

PIPE and CO-PIPE found good strategies by simultaneously: (1) identifying relevant inputs, (2) making action probabilities depend on relevant inputs only, (3) evolving programs that calculate useful conditional action probabilities. Another important aspect is: unlike TD-Q, PIPE and CO-PIPE learn to map inputs to "greediness values" used in the (Boltzmann-Gibbs) exploration rule. This enables them to pick actions more or less stochastically and control their own exploration process.

TD-Q's problems are due to a combination of reasons. **(1)** *Linear networks.* Linear networks have limited expressive power. They seem unable to learn *and* keep appropriate evaluation functions (EFs). Increasing expressive power by adding hidden units (time-consuming!) or using local function approximators such as CMACS (Albus, 1975; Sutton, 1996) (as proposed by Sutton, personal communication, 1997) may significantly improve TD-Q's performance. In fact, initial experiments with CMACS and complex inputs already led to promising results. **(2)** *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 & Barto, 1996). Our soccer scenario's POP, however, seems harder to overcome than POPs of many scenarios studied in previous work. **(3)** *Agent credit assignment problem (ACAP)* (Weiss, 1996; Versino & 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 networks 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. **(4)** *Instability.* 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 which can lead to significant "shifts in policy space" and to "unlearning" of previous knowledge. This may lead to performance breakdowns.

Our multiagent scenario seems complex enough to require more sophisticated and time-consuming EF-based approaches than the one we tried. In principle, however, EFs are not necessary for finding good or optimal policies. Sometimes, particularly in the presence of POPs and ACAPs, it can make more sense to search policy space directly. That is what PIPE and CO-PIPE do. Currently PIPE-like, EF-independent techniques seem to learn faster and be easier applicable to complex multiagent learning scenarios.

Acknowledgments

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

Notes

1. After submission of this paper another recent attempt at learning soccer team strategies in more complex environments was published (Luke et al., 1997).

References

Albus, J. S. (1975). A new approach to manipulator control: The cerebellar model articulation controller (CMAC). *Dynamic Systems, Measurement and Control*, 97:220–227.

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

Baluja, S. (1994). Population-based incremental learning: A method for integrating genetic search based function optimization and competitive learning. Technical Report CMU-CS-94-163, Carnegie Mellon University, Pittsburgh.

Baluja, S. & 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, San Francisco, CA. Morgan Kaufmann Publishers.

Bertsekas, D. P. & Tsitsiklis, J. N. (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*, pages 183–187, Hillsdale NJ. Lawrence Erlbaum Associates.

Crites, R. & 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., & Winklhofer, A. (1987). Der genetische Algorithmus: Eine Implementierung in Prolog. Fortgeschrittenenpraktikum, Institut für Informatik, Lehrstuhl Prof. Radig, Technische Universität München.

Gallant, S. I. (1993). *Neural Network Learning and Expert Systems.* MIT Press, Cambridge MA.

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

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. & Vitányi, P. M. B. (1993). *An Introduction to Kolmogorov Complexity and its Applications.* Springer-Verlag, New York.

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, San Francisco, CA. Morgan Kaufmann Publishers.

Luke, S., Hohn, C., Farris, J., Jackson, G., & Hendler, J. (1997). Co-evolving soccer softbot team coordination with genetic programming. In *Proceedings of the First International Workshop on RoboCup, at the International Joint Conference on Artificial Intelligence (IJCAI-97).*

Matsubara, H., Noda, I., & Hiraki, K. (1996). Learning of cooperative actions in multi-agent systems: a case study of pass play in soccer. In Sen, S., editor, *Working Notes for the AAAI-*

96 Spring Symposium on Adaptation, Coevolution and Learning in Multi-agent Systems, pages 63–67, Menlo Park, CA. AAAI Press.

Nadella, R. & Sen, S. (1996). Correlating internal parameters and external performance: learning soccer agents. In Weiss, G., editor, *Distributed Artificial Intelligence Meets Machine Learning. Learning in Multi-Agent Environments*, pages 137–150. Springer-Verlag, Berlin.

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

Peng, J. & Williams, R. (1996). Incremental multi-step Q-learning. *Machine Learning*, 22:283–290.

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

Sałustowicz, R. P. & Schmidhuber, J. (1997). Probabilistic incremental program evolution. *Evolutionary Computation*, 5(2):123–141.

Sałustowicz, R. P., Wiering, M. A., & Schmidhuber, J. (1997a). Evolving soccer strategies. In *Proceedings of the Fourth International Conference on Neural Information Processing (ICONIP'97)*, pages 502–506, Singapore. Springer-Verlag.

Sałustowicz, R. P., Wiering, M. A., & Schmidhuber, J. (1997b). On learning soccer strategies. In Gerstner, W., Germond, A., Hasler, M., and Nicoud, J.-D., editors, *Proceedings of the Seventh International Conference on Artificial Neural Networks (ICANN'97)*, volume 1327 of *Lecture Notes in Computer Science*, pages 769–774, Berlin Heidelberg. Springer-Verlag.

Schmidhuber, J. (1997a). Discovering neural nets with low Kolmogorov complexity and high generalization capability. *Neural Networks*, 10(5):857–873.

Schmidhuber, J. (1997b). 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., Zhao, J., & Schraudolph, N. (1997a). Reinforcement learning with self-modifying policies. In Thrun, S. and Pratt, L., editors, *Learning to learn*, pages 293–309. Kluwer.

Schmidhuber, J., Zhao, J., & Wiering, M. (1997b). Shifting inductive bias with success-story algorithm, adaptive Levin search, and incremental self-improvement. *Machine Learning*, 28:105–130.

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. & Veloso, M. (1996a). 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 8*, pages 896–902, Cambridge MA. MIT Press.

Stone, P. & Veloso, M. (1996b). A layered approach to learning client behaviors in the robocup soccer server. To appear in Applied Artificial Intelligence (AAI) in 1998.

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44.

Sutton, R. S. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding. In Touretzky, D. S., Mozer, M. C., and Hasselmo, M. E., editors, *Advances in Neural Information Processing Systems 8*, pages 1038–1045, Cambridge MA. MIT Press.

Sutton, R. S. (1997). Personal communication at the Seventh International Conference on Artificial Neural Networks (ICANN'97).

Tesauro, G. (1994). TD-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6(2):215–219.

Versino, C. & Gambardella, L. M. (1997). Learning real team solutions. In Weiss, G., editor, *DAI Meets Machine Learning*, volume 1221 of *Lecture Notes in Artificial Intelligenc*, pages 40–61. Springer-Verlag, Berlin.

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

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 of *Lecture Notes in Artificial Intelligence*, pages 1–21. Springer-Verlag, Berlin Heidelberg.

Widrow, B. & 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. & 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, San Francisco, CA. Morgan Kaufmann Publishers.

Wiering, M. A. & Schmidhuber, J. (1997). Fast online Q(λ). Technical Report IDSIA-21-97, IDSIA, Lugano, Switzerland.