Reinforcement Learning for RoboCup-Soccer Keepaway

Peter Stone
Department of Computer Sciences
The University of Texas at Austin
1 University Station C0500
Austin, TX 78712-1188
phone: +1 512 471-9796
fax: +1 512 471-8885
pstone@cs.utexas.edu
http://www.cs.utexas.edu/~pstone

Richard S. Sutton
Department of Computing Science
University of Alberta
2-21 Athabasca Hall
Edmonton, Alberta, Canada T6G 2E8
sutton@cs.ualberta.ca
http://www.cs.ualberta.ca/~sutton/

Gregory Kuhlmann
Department of Computer Sciences
The University of Texas at Austin
1 University Station C0500
Austin, TX 78712-1188
kuhlmann@cs.utexas.edu
http://www.cs.utexas.edu/~kuhlmann

November 27, 2005

Abstract

RoboCup simulated soccer presents many challenges to reinforcement learning methods, including a large state space, hidden and uncertain state, multiple independent agents learning simultaneously, and long and variable delays in the effects of actions. We describe our application of episodic SMDP Sarsa(λ) with linear tile-coding function approximation and variable λ to learning higher-level decisions in a keepaway subtask of RoboCup soccer. In keepaway, one team, "the keepers," tries to keep control of the ball for as long as possible despite the efforts of "the takers." The keepers learn individually when to hold the ball and when to pass to a teammate. Our agents learned policies that significantly outperform a range of benchmark policies. We demonstrate the generality of our approach by applying it to a number of task variations including different field sizes and different numbers of players on each team.

Keywords: multiagent systems, machine learning, multiagent learning, reinforcement learning, robot soccer

1 Introduction

Reinforcement learning (Sutton & Barto, 1998) is a theoretically-grounded machine learning method designed to allow an autonomous agent to maximize its long-term reward via repeated experimentation in and interaction with its environment. Under certain conditions, reinforcement learning is guaranteed to enable the agent to converge to an optimal control policy, and has been empirically demonstrated to do so in a series of relatively simple testbed domains. Despite its appeal, reinforcement learning can be difficult to scale up to larger domains due to the exponential growth of states in the number of state variables (the "curse of dimensionality"). A limited number of successes have been reported in large-scale domains, including backgammon (Tesauro, 1994), elevator control (Crites & Barto, 1996), and helicopter control (Bagnell & Schneider, 2001). This article contributes to the list of reinforcement learning successes, demonstrating that it can apply successfully to a complex multiagent task, namely *keepaway*, a subtask of RoboCup simulated soccer.

RoboCup simulated soccer has been used as the basis for successful international competitions and research challenges (Kitano, Tambe, Stone, Veloso, Coradeschi, Osawa, Matsubara, Noda, & Asada, 1997). As presented in detail by Stone (2000), it is a fully *distributed*, *multiagent* domain with both *teammates* and *adversaries*. There is *hidden state*, meaning that each agent has only a partial world view at any given moment. The agents also have *noisy sensors and actuators*, meaning that they do not perceive the world exactly as it is, nor can they affect the world exactly as intended. In addition, the perception and action cycles are *asynchronous*, prohibiting the traditional AI paradigm of using perceptual input to trigger actions. *Communication* opportunities are limited, and the agents must make their decisions in *real-time*. These italicized domain characteristics combine to make simulated robot soccer a realistic and challenging domain.

In principle, modern reinforcement learning methods are reasonably well suited to meeting the challenges of RoboCup simulated soccer. Reinforcement learning is all about sequential decision making, achieving delayed goals, and handling noise and stochasticity. It is also oriented toward making decisions relatively rapidly rather than relying on extensive deliberation or meta-reasoning. There is a substantial body of reinforcement learning research on multiagent decision making, and soccer is an example of the relatively benign case in which all agents on the same team share the same goal. In this case it is often feasible for each agent to learn independently, sharing only a

common reward signal. The large state space remains a problem, but can, in principle, be handled using function approximation, which we discuss further below. RoboCup soccer is a large and difficult instance of many of the issues which have been addressed in small, isolated cases in previous reinforcement learning research. Despite substantial previous work (e.g., (Andou, 1998; Stone & Veloso, 1999; Uchibe, 1999; Riedmiller, Merke, Meier, Hoffman, Sinner, Thate, & Ehrmann, 2001)), the extent to which modern reinforcement learning methods can meet these challenges remains an open question.

Perhaps the most pressing challenge in RoboCup simulated soccer is the large state space, which requires some kind of general function approximation. Stone and Veloso (1999) and others have applied state aggregation approaches, but these are not well suited to learning complex functions. In addition, the theory of reinforcement learning with function approximation is not yet well understood (e.g., see (Sutton & Barto, 1998; Baird & Moore, 1999; Sutton, McAllester, Singh, & Mansour, 2000)). Perhaps the best understood of current methods is linear Sarsa(λ) (Sutton & Barto, 1998), which we use here. This method is not guaranteed to converge to the optimal policy in all cases, but several lines of evidence suggest that it is near a good solution (Gordon, 2001; Tsitsiklis & Van Roy, 1997; Sutton, 1996) and recent results show that it does indeed converge, as long as the action-selection policy is continuous (Perkins & Precup, 2003). Certainly it has advantages over off-policy methods such as Q-learning (Watkins, 1989), which can be unstable with linear and other kinds of function approximation. An important open question is whether Sarsa's failure to converge is of practical importance or is merely a theoretical curiosity. Only tests on large-state-space applications such as RoboCup soccer will answer this question.

In this article we begin to scale reinforcement learning up to RoboCup simulated soccer. We consider a subtask of soccer involving 5–9 players rather than the full 22. This is the task of *keepaway*, in which one team merely seeks to keep control of the ball for as long as possible. The main contribution of this article is that it considers a problem that is at the limits of what reinforcement learning methods can tractably handle and presents successful results using, mainly, a single approach, namely episodic SMDP Sarsa(λ) with linear tile-coding function approximation and variable λ. Extensive experiments are presented demonstrating the effectiveness of this approach relative to several benchmarks.

The remainder of the article is organized as follows. In the next section we describe keepaway

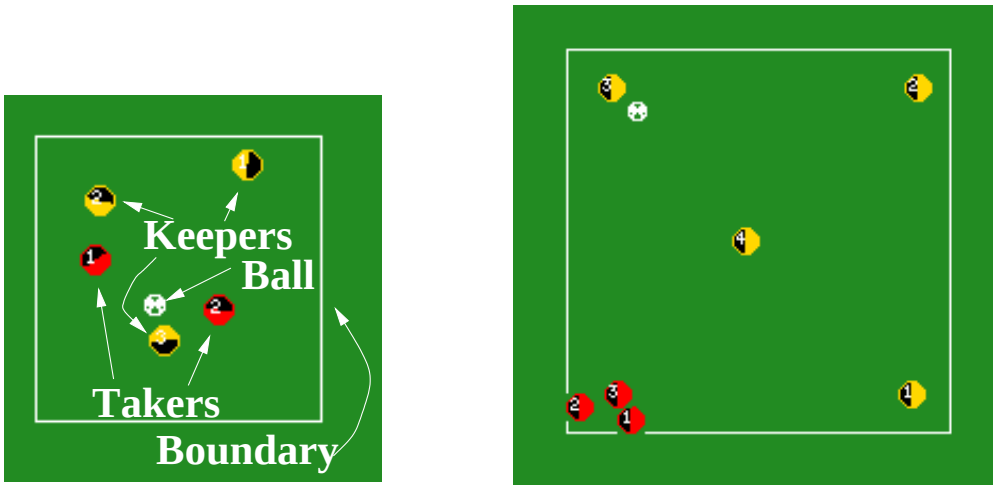


Figure 1: **Left:** A screen shot from the middle of a 3 vs. 2 keepaway episode in a 20m x 20m region. **Right:** A starting configuration for a 4 vs. 3 keepaway episode in a 30m x 30m region.

and how we build on prior work in RoboCup soccer to formulate this problem at an intermediate level above that of the lowest level actions and perceptions. In Section 3 we map this task onto an episodic reinforcement learning framework. In Sections 4 and 5 we describe our learning algorithm in detail and our results respectively. Related work is discussed further in Section 6 and Section 7 concludes.

2 Keepaway Soccer

We consider a subtask of RoboCup soccer, *keepaway*, in which one team, the *keepers*, tries to maintain possession of the ball within a limited region, while the opposing team, the *takers*, attempts to gain possession. Whenever the takers take possession or the ball leaves the region, the *episode* ends and the players are reset for another episode (with the keepers being given possession of the ball again).

Parameters of the task include the size of the region, the number of keepers, and the number of takers. Figure 1 shows screen shots of episodes with 3 keepers and 2 takers (called 3 vs. 2, or 3v2 for short) playing in a 20m x 20m region and 4 vs. 3 in a 30m x 30m region.[1]

All of the work reported in this article uses the standard RoboCup soccer simulator[2] (Noda,

[1] Flash files illustrating the task and are available at `http://www.cs.utexas.edu/~AustinVilla/sim/keepaway/`

[2] Version 8

Matsubara, Hiraki, & Frank, 1998). Agents in the RoboCup simulator receive visual perceptions every 150 *msec* indicating the relative distance and angle to visible objects in the world, such as the ball and other agents. They may execute a parameterized primitive action such as `turn(`*angle*`)`, `dash(`*power*`)`, or `kick(`*power*`,`*angle*`)` every 100 *msec*. Thus the agents must sense and act asynchronously. Random noise is injected into all sensations and actions. Individual agents must be controlled by separate processes, with no inter-agent communication permitted other than via the simulator itself, which enforces communication bandwidth and range constraints. From a learning perspective, these restrictions necessitate that teammates simultaneously learn *independent* control policies: they are not able to communicate their experiences or control policies during the course of learning, nor is any agent able to make decisions for the whole team. Thus multiagent learning methods by which the team shares policy information are not applicable. Full details of the RoboCup simulator are presented by Chen, Foroughi, Heintz, Kapetanakis, Kostiadis, Kummeneje, Noda, Obst, Riley, Steffens, Wang, and Yin (2003).

For the keepaway task, an omniscient coach agent manages the play, ending episodes when a taker gains possession of the ball for a set period of time or when the ball goes outside of the region. At the beginning of each episode, the coach resets the location of the ball and of the players semi-randomly within the region of play as follows. The takers all start in one corner (bottom left). Three randomly chosen keepers are placed one in each of the three remaining corners, and any keepers beyond three are placed in the center of the region. The ball is initially placed next to the keeper in the top left corner. A sample starting configuration with 4 keepers and 3 takers is shown in Figure 1.[3]

Keepaway is a *subproblem* of the complete robot soccer domain. The principal simplifications are that there are fewer players involved; they are playing in a smaller area; and the players are always focused on the same high-level goal—they don't need to balance offensive and defensive considerations. In addition, in this article we focus on learning parts of the keepers' policies when playing against fixed, pre-specified takers. Nevertheless, the skills needed to play keepaway well are also very useful in the full problem of robot soccer. Indeed, ATT-CMUnited-2000—the 3rd-place finishing team in the RoboCup-2000 simulator league—incorporated a successful hand-coded solution to an 11 vs. 11 keepaway task (Stone & McAllester, 2001).

[3]Starting with version 9, we have incorporated support for keepaway into the standard release of the RoboCup soccer simulator.

One advantage of keepaway is that it is more suitable for directly comparing different machine learning methods than is the full robot soccer task. In addition to the reinforcement learning approaches mentioned above, machine learning techniques including genetic programming, neural networks, and decision trees have been incorporated in RoboCup teams (e.g., see (Luke, Hohn, Farris, Jackson, & Hendler, 1998; Andre & Teller, 1999; Stone, 2000)). A frustration with these and other machine learning approaches to RoboCup is that they are all embedded within disparate systems, and often address different subtasks of the full soccer problem. Therefore, they are difficult to compare in any meaningful way. Keepaway is simple enough that it can be successfully learned in its entirety, yet complex enough that straightforward solutions are inadequate. Therefore it is an excellent candidate for a machine learning benchmark problem. We provide all the necessary source code as well as step-by-step tutorials for implementing learning experiments in keepaway at `http://www.cs.utexas.edu/~AustinVilla/sim/keepaway/` .

3 Mapping Keepaway onto Reinforcement Learning

Our keepaway problem maps fairly directly onto the discrete-time, episodic, reinforcement-learning framework. The RoboCup soccer simulator operates in discrete time steps, $t = 0, 1, 2, \ldots$, each representing 100 *msec* of simulated time. When one episode ends (e.g., the ball is lost to the takers), another begins, giving rise to a series of episodes. Each player learns *independently* and may perceive the world differently. For each player, an episode begins when the player is first asked to make a decision and ends when possession of the ball is lost by the keepers.

As a way of incorporating domain knowledge, our learners choose not from the simulator's primitive actions, but from higher level macro-actions based closely on skills used in the CMUnited-99 team.[4] These skills include

HoldBall(): Remain stationary while keeping possession of the ball in a position that is as far away from the opponents as possible.

PassBall(k): Kick the ball directly towards keeper k.

GetOpen(): Move to a position that is free from opponents and open for a pass from the ball's current position (using SPAR (Veloso, Stone, & Bowling, 1999)).

[4]These skills, along with the entire CMUnited-99 team are all fully specified by Stone (2000).

GoToBall(): Intercept a moving ball or move directly towards a stationary ball.

BlockPass(k): Move to a position between the keeper with the ball and keeper k.

All of these skills except PassBall(k) are simple functions from state to a corresponding primitive action; an invocation of one of these normally controls behavior for a single time step. PassBall(k), however, requires an extended sequence of primitive actions, using a series of kicks to position the ball, and then accelerate it in the desired direction (Stone, 2000); a single invocation of PassBall(k) influences behavior for several time steps. Moreover, even the simpler skills may last more than one time step because the player occasionally misses the step following them; the simulator occasionally misses commands; or the player may find itself in a situation requiring it to take a specific action, for instance to self-localize. In these cases there is no new opportunity for decision-making until two or more steps after invoking the skill. To handle such possibilities, it is convenient to treat the problem as a *semi-Markov* decision process, or SMDP (Puterman, 1994; Bradtke & Duff, 1995). An SMDP evolves in a sequence of jumps from the initiation of each SMDP macro-action to its termination one or more time steps later, at which time the next SMDP macro-action is initiated. SMDP macro-actions that consist of a subpolicy and termination condition over an underlying decision process, as here, have been termed *options* (Sutton, Precup, & Singh, 1999). Formally,

> Options consist of three components: a policy $\pi : \mathcal{S} \times \mathcal{A}_p \rightarrow [0,1]$, a termination condition $\beta : \mathcal{S}^+ \rightarrow [0,1]$, and an initiation set $\mathcal{I} \subseteq \mathcal{S}$. An option $(\mathcal{I}, \pi, \beta)$ is available in state s_t if and only if $s_t \in \mathcal{I}$. If the option is taken, then actions are selected according to π until the option terminates stochastically according to β (Sutton et al., 1999).

In this context, $\mathcal{S}$ is the set of primitive states and $\mathcal{A}_p$ is the set of primitive actions in the domain. $\pi(s, a)$ is the probability of selecting primitive action a when in state s and β denotes the probability of terminating the option after seeing a given sequence of states.

From the team perspective, keepaway can be considered a *distributed* SMDP, as each teammate is in charge of a portion of the team's overall decision process. Because the players learn simultaneously without any shared knowledge, in what follows we present the task from an individual's perspective.

Each individual's choices are among macro-actions, not primitive ones, so henceforth we will use the terms *action* and *macro-action* interchangeably, while always distinguising *primitive actions*.

We use the notation $a_i \in \mathcal{A}$ to denote the ith macro-action selected. Thus, several (primitive) time steps may elapse between a_i and a_{i+1}. Similarly, we use $s_{i+1} \in \mathcal{S}$ and $r_{i+1} \in \Re$ for the state and reward following the ith macro-action. From the point of view of an individual, then, an episode consists of a sequence of states, actions, and rewards selected and occurring at the macro-action boundaries:

$$s_0, a_0, r_1, s_1, \ldots, s_i, a_i, r_{i+1}, s_{i+1}, \ldots, r_j, s_j$$

where a_i is chosen based on some, presumably incomplete, perception of s_i, and s_j is the terminal state in which the takers have possession or the ball has gone out of bounds. We wish to reward the keepers for each time step in which they keep possession, so we set the reward r_i to the the number of primitive time steps that elapsed while following action a_{i-1}: $r_i = t_i - t_{i-1}$. Because the task is episodic, no discounting is needed to express the objective: the keepers' goal at each learning step is to choose an action such that the remainder of the episode will be as long as possible, and thus to maximize total reward.

3.1 Keepers

Here we lay out the keepers' policy space in terms of the macro-actions from which they can select. Our experiments investigated learning by the keepers when in possession[5] of the ball. Keepers not in possession of the ball are required to select the Receive action:

Receive: If a teammate possesses the ball, or can get to the ball faster than this keeper can, invoke GetOpen() for one (primitive) time step; otherwise, invoke GoToBall() for one time step. Termination condition: Repeat until a keeper has possession of the ball or the episode ends.

A keeper in possession, on the other hand, is faced with a genuine choice. It may hold the ball, or it may pass to one of its teammates. That is, it chooses a macro-action from {Holdball, PassK_2ThenReceive, PassK_3ThenReceive, ..., PassK_nThenReceive} where the Holdball action simply executes HoldBall() for one step (or more if, for example, the server misses the next step) and the Pass$k$ThenReceive actions involve passes to the other keepers. The keepers are numbered by their closeness to the keeper with the ball: K_1 is the keeper with the ball, K_2 is the closest

[5] "Possession" in the soccer simulator is not well-defined because the ball never occupies the same location as a player. One of our agents considers that it has possession of the ball if the ball is close enough to kick it.

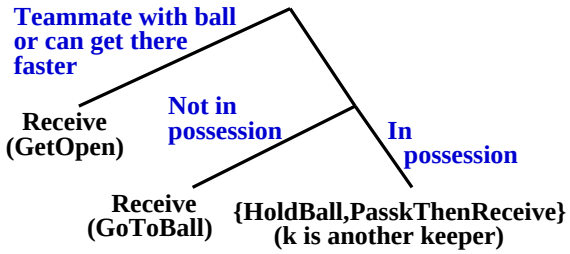


Figure 2: The keepers' policy space. The predicate "teammate with ball or can get there faster" evaluates to true whenever there exists a teammate that is in possession of the ball of that can get to the ball more quickly than the keeper evaluating the predicate. The latter is calculated by a forward-lookahead routine simulating how long it would take each agent to reach the ball if the ball continues along its current trajectory and the keeper moves optimally.

keeper to it, K_3 the next closest, and so on up to K_n, where n is the number of keepers. Each Pass$k$ThenReceive is defined as

Pass$k$ThenReceive: Invoke PassBall(k) to kick the ball toward teammate k. Then behave and terminate as in the Receive action.

The keepers' learning process thus searches a constrained policy space characterized only by the choice of action when in possession of the ball as illustrated in Figure 2. Examples of policies within this space are provided by our benchmark policies:

Random: Choose randomly among the n macro-actions, each with probability $\frac{1}{n}$.

Hold: Always choose HoldBall

Hand-coded: A hand-coded policy that selects from among the n macro-actions based on an intuitive mapping from the same state features that are used for learning (specified next). The hand-coded policy is fully specified in Section 5.3.2.

Note that though at any given moment only one agent has an action choice, behavior is still *distributed* in the sense that each agent can only control a portion of the team's collective behavior. Once passing the ball, subsequent decisions are made by the independently acting teammates until the ball is received back. Additionally, each player experiences the world from a different perspective and, when learning, must learn separate control policies.

We turn now to the representation of state used by the keepers, ultimately for value function approximation as described in the next section. Note that values are only needed on the SMDP

steps, and on these one of the keepers is always in possession of the ball. On these steps the keeper determines a set of state variables, computed based on the positions of: the keepers K_1–K_n, ordered as above; the takers T_1–T_m (m is the number of takers), ordered by increasing distance from K_1; and C, the center of the playing region (see Figure 3 for an example with 3 keepers and 2 takers labeled appropriately). Let $dist(a, b)$ be the distance between a and b and $ang(a, b, c)$ be the angle between a and c with vertex at b. As illustrated in Figure 3, with 3 keepers and 2 takers, we use the following 13 state variables:

- $dist(K_1, C)$; $dist(K_2, C)$; $dist(K_3, C)$;
- $dist(T_1, C)$; $dist(T_2, C)$;
- $dist(K_1, K_2)$; $dist(K_1, K_3)$;
- $dist(K_1, T_1)$; $dist(K_1, T_2)$;
- $\text{Min}(dist(K_2, T_1), dist(K_2, T_2))$;
- $\text{Min}(dist(K_3, T_1), dist(K_3, T_2))$;
- $\text{Min}(ang(K_2, K_1, T_1), ang(K_2, K_1, T_2))$;
- $\text{Min}(ang(K_3, K_1, T_1), ang(K_3, K_1, T_2))$.

This list generalizes naturally to additional keepers and takers, leading to a linear growth in the number of state variables.[6]

3.2 Takers

Although this article focuses on learning by the keepers against fixed, pre-specified takers, we specify the taker behaviors within the same framework for the sake of completeness.

The takers are relatively simple, choosing only macro-actions of minimum duration (one step, or as few as possible given server misses) that exactly mirror low-level skills. When a taker has the ball, it tries to maintain possession by invoking HoldBall() for a step. Otherwise, it chooses an action that invokes one of {GoToBall(), BlockPass(K_2), BlockPass(K_3), ..., BlockPass(K_n)} for one step or as few steps as permitted by the server. In case no keeper has the ball (e.g., during a

[6] More precisely, the growth is linear in the sum of the number of keepers and the number of takers.

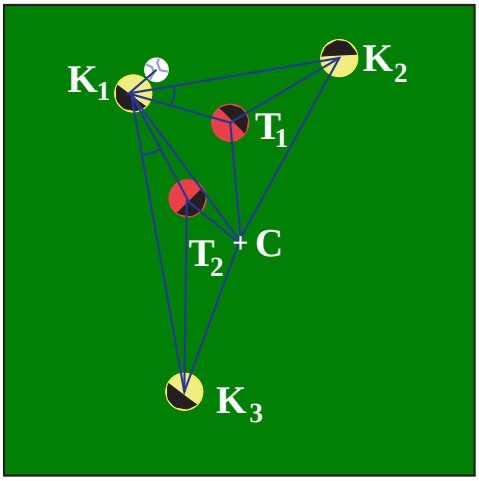


Figure 3: The state variables used for learning with 3 keepers and 2 takers. Keepers and takers are numbered by increasing distance from K_1, the keeper with the ball. The 13 lines and angles show the complete set of state variables.

pass), K_1 is defined here as the keeper predicted to next gain possession of the ball. The takers' policy space is depicted in Figure 4. We define the following three policies as taker benchmarks, characterized by their behavior when not in possession:

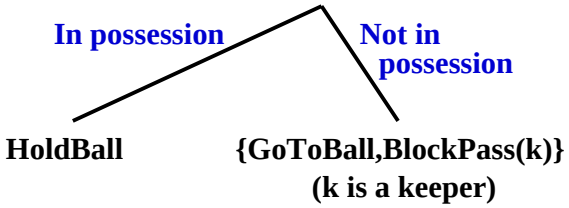


Figure 4: The takers' policy space.

Random-T: Choose randomly from the n macro-actions, each with probability $\frac{1}{n}$.

All-to-ball: Always choose the GoToBall action.

Hand-coded-T:

If no other taker can get to the ball faster than this taker, or this taker is the closest or second closest taker to the ball: choose the GoToBall action;

Else let k be the keeper with the largest angle with vertex at the ball that is clear of takers: choose the BlockPass(k) action.

Note that the All-to-ball and Hand-coded-T policies are equivalent when there are only two takers, since Hand-coded-T specifies that the two closest takers at any given time should go to the ball.

The takers' state variables are similar to those of the keepers. As before, C is the center of the region. T_1 is the taker that is computing the state variables, and T_2–T_m are the other takers ordered by increasing distance from K_1. $K_i$mid is the midpoint of the line segment connecting K_1 and K_i for $i \in [2, n]$ and where the K_i are ordered based on increasing distance of $K_i$mid from T_1. That is, $\forall i, j$ s.t. $2 \leq i < j$, $dist(T_1, K_i\text{mid}) \leq dist(T_1, K_j\text{mid})$. With 3 keepers and 3 takers, we used the following 18 state variables:

- $dist(K_1, C)$; $dist(K_2, C)$; $dist(K_3, C)$;
- $dist(T_1, C)$; $dist(T_2, C)$; $dist(T_3, C)$;
- $dist(K_1, K_2)$; $dist(K_1, K_3)$
- $dist(K_1, T_1)$; $dist(K_1, T_2)$; $dist(K_1, T_3)$;
- $dist(T_1, K_2\text{mid})$; $dist(T_1, K_3\text{mid})$;
- $\text{Min}(dist(K_2\text{mid}, T_2), dist(K_2\text{mid}, T_3))$;
- $\text{Min}(dist(K_3\text{mid}, T_2), dist(K_3\text{mid}, T_3))$;
- $\text{Min}(ang(K_2, K_1, T_2), ang(K_2, K_1, T_3))$;
- $\text{Min}(ang(K_3, K_1, T_2), ang(K_3, K_1, T_3))$;
- number of takers closer to the ball than T_1.

Once again, this list generalizes naturally to different numbers of keepers and takers.

4 Reinforcement Learning Algorithm

We use the SMDP version of the Sarsa(λ) algorithm with linear tile-coding function approximation (also known as CMACs) and replacing eligibility traces (see (Albus, 1981; Rummery & Niranjan, 1994; Sutton & Barto, 1998)). Each player learns simultaneously and independently from its own actions and its own perception of the state. Note that as a result, the value of a player's decision

depends in general on the current quality of its teammates control policies, which themselves change over time.

In the remainder of this section we introduce Sarsa(λ) (Section 4.1) and CMACs (Section 4.2) before presenting the full details of our learning algorithm in Section 4.3.

4.1 Sarsa(λ)

Sarsa(λ) is an on-policy learning method, meaning that the learning procedure estimates $Q(s, a)$, the value of executing action a from state s, subject to the current policy being executed by the agent. Meanwhile, the agent continually updates the policy according to the changing estimates of $Q(s, a)$.

In its basic form, Sarsa(λ) is defined as follows ((Sutton & Barto, 1998), Section 7.5):

Initialize $Q(s, a)$ arbitrarily and $e(s, a) = 0$ for all s, a.
Repeat (for each episode):
    Initialize s
    Choose a from s using policy derived from Q
    Repeat (for each step of episode):
        Take action a, observe reward r, s'
        Choose a' from s' using policy derived from Q
        $\delta \leftarrow r + \gamma Q(s', a') - Q(s, a)$
        $e(s, a) \leftarrow e(s, a) + 1$
        For all s, a:
            $Q(s, a) \leftarrow Q(s, a) + \alpha\delta e(s, a)$
            $e(s, a) \leftarrow \gamma\lambda e(s, a)$
        $s \leftarrow s'; a \leftarrow a'$;
    until s is terminal

Here, α is a learning rate parameter and γ is a discount factor governing the weight placed on future, as opposed to immediate, rewards.[7] The values in $e(s, a)$, known as *eligibility traces*, store the credit that past action choices should receive for current rewards; the parameter λ governs how much credit is delivered back to them. A typical policy derived from Q, and the one we use in this

[7]Sutton and Barto (1998) also present a version of Sarsa(λ) with function approximation (Section 8.4). We refrain from presenting their version with function approximation here for the sake of simplicity. But our notation is fully consistent with that presentation and our detailed algorithm is based on the same. Additional parameters introduced by function approximation are $\mathcal{F}_a$ and $\vec{\theta}$, both of which are introduced in Section 4.2.

article, is an ϵ-greedy policy in which a random action is selected with probability ϵ, and otherwise, the action with maximum Q-value $Q(s, a)$ from state s is selected.

In our application, one complication is that most descriptions of Sarsa(λ), including the above, assume the agent has control and occasionally calls the environment to obtain the next state and reward, whereas here the RoboCup simulator retains control and occasionally presents state perceptions and action choices to the agent. This alternate orientation requires a different perspective on the standard algorithm. We need to specify three routines: 1) `RLstartEpisode`, to be run by the player on the first time step in each episode in which it chooses a macro-action, 2) `RLstep`, to be run on each SMDP step, and 3) `RLendEpisode`, to be run when an episode ends. These three routines are presented in detail in Section 4.3.

4.2 Function Approximation

The basic Sarsa(λ) algorithm assumes that each action can be tried in each state infinitely often so as to fully and accurately populate the table of Q-values. A key challenge for applying RL in environments with large state spaces is to be able to generalize the state representation so as to make learning work in practice despite a relatively sparse sample of the state space. In particular, in keepaway we cannot expect the agent to directly experience all possible sets of values of the variables depicted in Figure 3. Rather, the agent needs to learn, based on limited experiences, how to act in new situations. To do so, the table of Q-values must be approximated using some representation with fewer free variables than there are states, a technique commonly known as *function approximation*.

Many different function approximators exist and have been used successfully ((Sutton & Barto, 1998), Section 8). Here we use general tile coding software to specify how the feature sets, $\mathcal{F}_a$, are used for learning. Tile coding allows us to take arbitrary groups of continuous state variables and lay infinite, axis-parallel tilings over them (e.g., see Figure 5). The tiles containing the current state in each tiling together make up a feature set $\mathcal{F}_a$, with each action a indexing the tilings differently. The tilings are formally infinite in extent, but in our case, all the state variables are in fact bounded. Nevertheless, the number of possible tiles is extremely large, only a relatively few of which are ever visited (in our case about 10,000). Thus the primary memory vector, $\vec{\theta}$, and the eligibility trace vector $\vec{e}$ have only this many nonzero elements. Using open-addressed hash-coding,

only these nonzero elements need be stored.

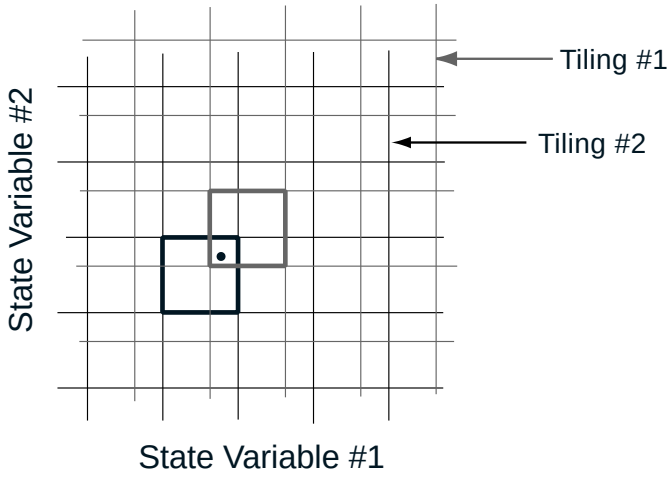


Figure 5: Our tile-coding feature sets were formed from multiple overlapping tilings of the state variables. Here we show two grid-tilings overlaid over the space formed by two state variables. (In this article we primarily considered one-dimensional tilings.) Any state, such as that shown by the dot, is in exactly one tile of each tiling. Tile coding, also known as CMACs, has been widely used in conjunction with reinforcement learning systems (e.g., (Watkins, 1989; Lin & Kim, 1991; Dean et al., 1992)).

An advantage of tile coding is that it allows us ultimately to learn weights associated with discrete, binary features, thus eliminating issues of scaling among features of different types. The most straightforward way to get binary features is to break the state space into discrete bins. However, doing so can lead to over-generalization based on the fact that points in the same bin are required to have the same value and under-generalization due to the fact that points in different bins, no matter how close, have unrelated values. By overlaying multiple tilings it is possible to achieve quick generalization while maintaining the ability to learn fine distinctions. See Figure 6 for an illustrative example.

In our experiments we used primarily single-dimensional tilings, i.e., simple stripes or intervals along each state variable individually. For each variable, 32 tilings were overlaid, each offset from the others by 1/32 of a tile width. In each tiling, the current state is in exactly one tile. The set of all these "active" tiles, one per tiling and 32 per state variable, is what makes up the $\mathcal{F}_a$ vectors. In the 3v2 case, there are 416 tiles in each $\mathcal{F}_a$ because there are thirteen state variables making thirteen single-variable groups, or $13 * 32 = 416$ total. For each state variable, we specified the width of its tiles based on the width of generalization that we desired. For example, distances were

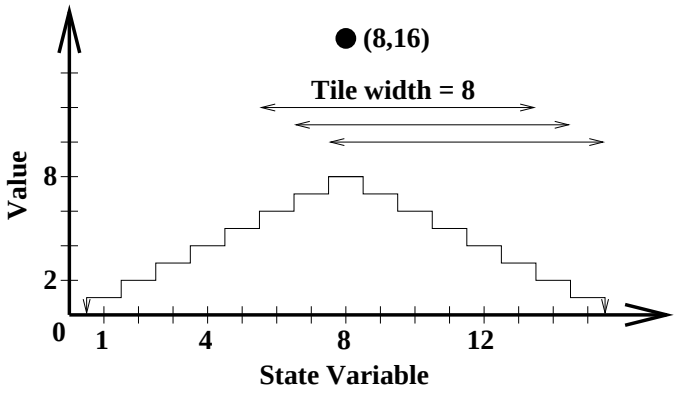


Figure 6: The generalization from a single training example point when using CMACs with learning rate $\alpha = .5$, and 8 overlapping tilings with width 8. The graph shows the predicted value (action value) as a function of a single state variable. The training example, shown as the point (8,16), indicates that when the state variable was set to 8, a reward of 16 was received. Prior to this example, all points were initialized to a value of 0. The tiles that contain the example point from 3 different tilings are shown in the figure. The points that are in all of the same tiles as the example point (those between 7.5 and 8.5) generalize the full amount ($16 * .5 = 8$). Those that share fewer tiles in common generalize proportionately less, out to the points that are almost a full tile-width away (those between 14.5 and 15.5) which only share one tile in common with 8 and therefore only generalize to $\frac{1}{8}$ of $8 = 1$. Carrying this example further by placing additional training examples, one can see that it is possible to generalize broadly and quickly, but also to learn fine distinctions, for example if there are many training examples near (8,16) and many others near (7,0). In that case a near step function will be learned eventually.

given widths of about 3.0 meters, whereas angles were given widths of about 10.0 degrees.

The choice here of state variables, widths, groupings, and so on, was done manually. Just as we as people have to select the state variables, we also have to determine how they are represented to the learning algorithm. A long-term goal of machine learning is to automate representational selections, but to date this is not possible even in supervised learning. Here we seek only to make the experimentation with a variety of representations relatively easy for us to do. The specific choices described here were made after some experimentation with learning by the keepers in a policy-evaluation scenario (Stone, Sutton, & Singh, 2001). Our complete representation scheme is illustrated in Figure 7.

4.3 Algorithmic Detail

In this section, we present the full details of our approach as well as the parameter values we chose, and how we arrived at them. Figure 8 shows pseudocode for the three top-level subroutines,

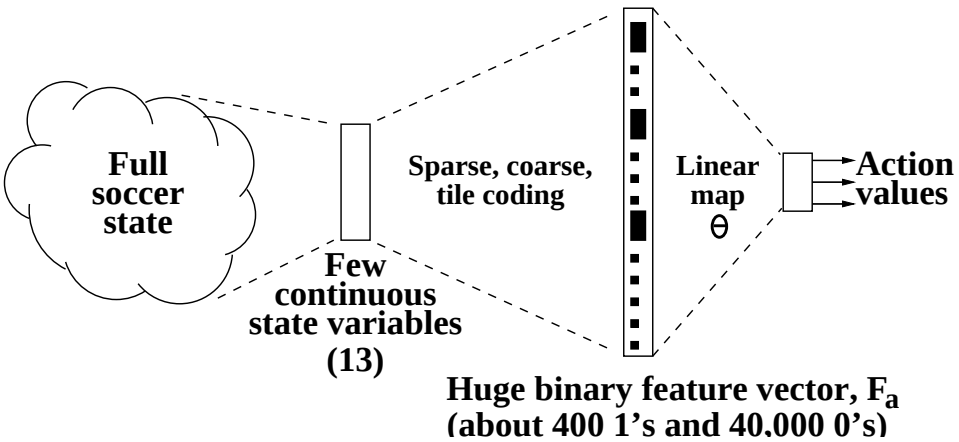


Figure 7: A pictorial summary of the complete representation scheme. The full soccer state is mapped to a few continuous state variables, which are then tiled into binary features, and ultimately combined to get action values.

`RLstartEpisode`, `RLstep`, and `RLendEpisode`.

4.3.1 `RLstartEpisode`

`RLstartEpisode` is run by each player on the first time step in each episode in which it chooses a macro-action. In line 1, we iterate over all actions available in the current state. For each action, a, and for each tiling of each state variable, we find the set of tiles, $\mathcal{F}_a$, activated in the current state (line 2). Next, in line 3, the action value for action a in the current state is calculated as the sum of the weights of the tiles in $\mathcal{F}_a$. We then choose an action from the set of available macro-actions by following an ϵ-greedy policy (line 4). The chosen action is stored in $LastAction$ and the current time is stored in $LastActionTime$ (lines 4–5). In line 6, the eligibility trace vector is cleared. Finally, in lines 7–8, the eligibility traces for each active tile of the selected action are set to 1, allowing the weights of these tiles to receive learning updates in the following step.

4.3.2 `RLstep`

`RLstep` is run on each SMDP step (and so only when some keeper has the ball). First, in line 9, the reward for the previous SMDP step is computed as the number of time steps since the macro-action began execution. Second, in line 10, we begin to calculate the error in our action value estimates by computing the difference between r, the reward we received, and $Q_{LastAction}$, the expected return of the previous SMDP step. Next, in lines 11–13, we find the active tiles and use their weights

RLstartEpisode:

For all $a \in \mathcal{A}^s$:
  $\mathcal{F}_a \leftarrow$ set of tiles for a, s
  $Q_a \leftarrow \sum_{i \in \mathcal{F}_a} \theta(i)$
$LastAction \leftarrow \begin{cases} \arg\max_a Q_a & \text{w/prob. } 1-\epsilon \\ \text{random action} & \text{w/prob. } \epsilon \end{cases}$
$LastActionTime \leftarrow CurrentTime$
$\vec{e} = \vec{0}$
For all $i \in \mathcal{F}_{LastAction}$:
  $e(i) \leftarrow 1$

RLstep:

$r \leftarrow CurrentTime - LastActionTime$
$\delta \leftarrow r - Q_{LastAction}$
For all $a \in \mathcal{A}^s$:
  $\mathcal{F}_a \leftarrow$ set of tiles for a, s
  $Q_a \leftarrow \sum_{i \in \mathcal{F}_a} \theta(i)$
$LastAction \leftarrow \begin{cases} \arg\max_a Q_a & \text{w/prob. } 1-\epsilon \\ \text{random action} & \text{w/prob. } \epsilon \end{cases}$
$LastActionTime \leftarrow CurrentTime$
$\delta \leftarrow \delta + Q_{LastAction}$
$\vec{\theta} \leftarrow \vec{\theta} + \alpha\, \delta\, \vec{e}$
$Q_{LastAction} \leftarrow \sum_{i \in \mathcal{F}_{LastAction}} \theta(i)$
$\vec{e} \leftarrow \lambda \vec{e}$
If player acting in state s:
  For all $a \in \mathcal{A}^s$ s.t. $a \neq LastAction$:
    For all $i \in \mathcal{F}_a$:
      $e(i) \leftarrow 0$
  For all $i \in \mathcal{F}_{LastAction}$:
    $e(i) \leftarrow 1$

RLendEpisode:

$r \leftarrow CurrentTime - LastActionTime$
$\delta \leftarrow r - Q_{LastAction}$
$\vec{\theta} \leftarrow \vec{\theta} + \alpha\, \delta\, \vec{e}$

Figure 8: The three main routines of our Sarsa(λ) implementation presented for a keeper. A taker has the sign of the reward, r, reversed. The set of macro-actions available, $\mathcal{A}^s \subseteq \mathcal{A}$, depends on the current state, s. For example, the keepers not in possession of the ball must select the Receive action, whereas the keeper with the ball chooses from among HoldBall and Pass$k$ThenReceive.

to compute the action values for each action in the current state. In lines 14–15, the next action is selected as in `RLstartEpisode`. In line 16, we finish our calculation of the error that began on line 10. Here, we add the new $Q_{LastAction}$, the expected return of choosing action $LastAction$ in the current state. Next, in line 17, we adjust the weights by the learning factor α times our error estimate δ, for tiles with non-zero eligibility traces. Because the weights have changed, in line 18, we must recalculate $Q_{LastAction}$. In line 19, the eligibility traces are decayed. Note that the traces decay only on SMDP time steps. In effect, we are using variable λ (Sutton & Barto, 1998) and setting $\lambda = 1$ for the missing time steps. In lines 20–25, the traces for the chosen action are set to 1, and the traces for the remaining available actions are cleared. Note that we do not clear the traces for actions that are not in $\mathcal{A}^s$ because they don't apply in this state. This scheme, known as *replacing traces*, is one reasonable way to handle eligibility traces for SMDPs.

4.3.3 `RLendEpisode`

`RLendEpisode` is run when an episode ends. First, in line 26, we calculate the reward for the last SMDP step. Next, in line 27, we calculate the error δ. There is no need to add the expected return of the current state since this value is defined to be 0 for terminating states. In line 28, we perform the final weight update for this episode.

4.3.4 Computational Considerations and Parameter Values

The primary memory vector $\vec{\theta}$ and the eligibility trace vector $\vec{e}$ are both of large dimension (e.g., thousands of dimensions for 3v2 keepaway), whereas the feature sets $\mathcal{F}_a$ are relatively small (e.g., 416 elements for 3v2 keepaway). The steps of greatest computational expense are those in which $\vec{\theta}$ and $\vec{e}$ are updated. By keeping track of the few nonzero components of $\vec{e}$, however, this expense can be kept to a small multiple of the size of the $\mathcal{F}_a$ (i.e., of 416). The initial value for $\vec{\theta}$ was $\vec{0}$.

For the results described in this article we used the following values of the scalar parameters: $\alpha = 0.125$, $\epsilon = 0.01$, and $\lambda = 0$. In previous work (Stone et al., 2001), we experimented systematically with a range of values for the step-size parameter. We varied α over negative powers of 2 and observed the classical inverted-U pattern, indicating that values of α both too close to 0 and too close to 1 lead to slower learning than do intermediate values. In our case, we observed the fastest learning at a value of about $\alpha = 2^{-3} = 0.125$, which we use here. We also experimented informally

with ϵ and λ. The value $\epsilon = 0.01$ appears sufficiently exploratory without significantly affecting final performance. The effect of varying λ appears not to be large (i.e. results are not sensitive to varying λ), so in this article we treat the simplest case of $\lambda = 0$. The only exception is on SMDP steps, for which we set $\lambda = 1$.

Since many of these parameters have been chosen as a result of brief, informal experimentation, we make no claims that they are the optimal values. Indeed, our overall methodology throughout this research has been to find *good* parameter values and algorithmic components (e.g. representation, function approximator, etc.) as quickly as possible and to move on, rather than fixating on any individual portion of the problem and insisting on finding the *best* values and/or components. This methodology has allowed us proceed relatively quickly towards our goal of finding effective solutions for large-scale problems. In this article we report all of the values that we used and how we reached them. However, there may very well be room for further optimizations at any number of levels.

5 Empirical Results

In this section we report our empirical results in the keepaway domain. In previous work (Stone et al., 2001), we first learned a value function for the case in which the agents all used fixed, hand-coded policies. Based on these experiments, we chose the representation described in Section 4 that we then used for policy learning experiments, but with the simplifications of full, noise-free vision for the agents (Stone & Sutton, 2001). Here we extend those results by reporting performance with the full set of sensory challenges presented by the RoboCup simulator.

Section 5.1 summarizes our previously-reported initial results. We then outline a series of follow-up questions in Section 5.2 and address them empirically in Section 5.3.

5.1 Initial Results

In the RoboCup soccer simulator, agents typically have limited and noisy sensors: each player can see objects within a 90^o view cone, and the precision of an object's sensed location degrades with distance. However, to simplify the task, we initially removed these restrictions. The learners were given 360^o of noiseless vision to ensure that they would always have complete and accurate knowledge of the world state. Here we begin by summarizing these initial results. The remainder

of this section examines, among other things, the extent to which these simplifications were useful and necessary.

Using the setup described in Section 4, we were able to show an increase in average episode duration over time when keepers learned against hand-coded takers. We compared our results with a Random policy that chooses from among its macro-actions with uniform probability, an Always Hold policy that invokes the HoldBall() macro-action in every cycle, and a hand-coded policy that uses a decision tree for pass evaluation. Experiments were conducted on several different field sizes. In each case, the keepers were able to learn policies that outperform all of the benchmarks. Most of our experiments matched 3 keepers against 2 takers. However, we also showed that our results extended to the 4 vs. 3 scenario.

Our initial results focused on learning by the keepers in 3v2 keepaway in a 20x20 region. For the opponents (takers) we used the Hand-coded-T policy (note that with just 2 takers, this policy is identical to All-to-ball). To benchmark the performance of the learned keepers, we first ran the three benchmark keeper policies, Random, Always Hold, and Hand-coded,[8] as laid out in Section 3.1. Average episode lengths for these three policies were 5.5, 4.8, and 5.6 seconds respectively. Figure 9 shows histograms of the lengths of the episodes generated by these policies.

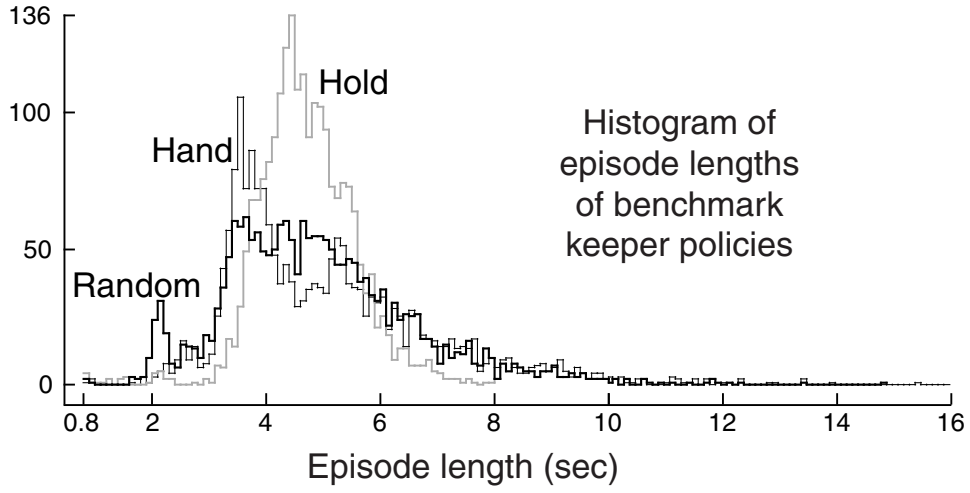


Figure 9: Histograms of episode lengths for the 3 benchmark keeper policies in 3v2 keepaway in a 20x20 region.

We then ran a series of eleven runs with learning by the keepers against the Hand-coded-T

[8]The hand-coded policy used in the initial experiments, as described fully by Stone and Sutton (2001), was slightly simpler than the one specified and used in the main experiments in this article.

takers. Figure 10 shows learning curves for these runs. A sample learned behavior, along with the benchmark behaviors and several of the other behaviors reported in this article, can be viewed at `http://www.cs.utexas.edu/~AustinVilla/sim/keepaway/`. The y-axis is the average time that the keepers are able to keep the ball from the takers (average episode length); the x-axis is training time (simulated time $\approx$ real time). The performance levels of the benchmark keeper policies are shown as horizontal lines.

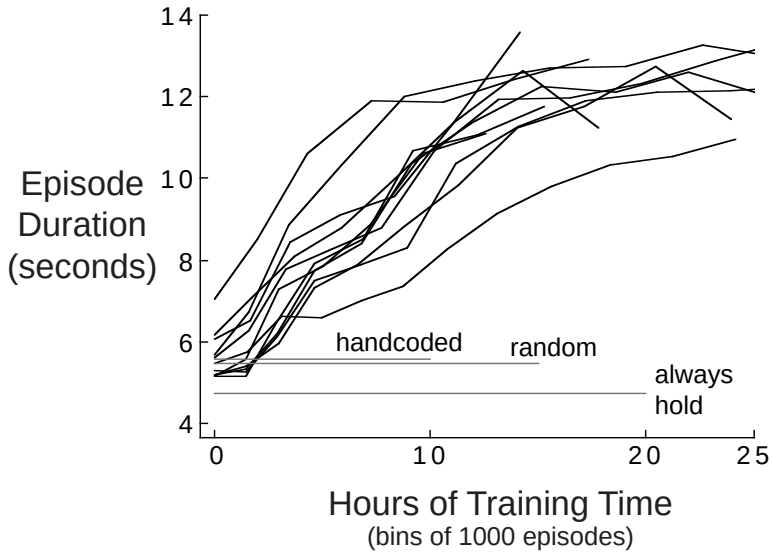


Figure 10: Multiple successful runs under identical characteristics: 3v2 keepaway in a 20x20 region against hand-coded takers.

This data shows that we were able to learn policies that were much better than any of the benchmarks. All learning runs quickly found a much better policy than any of the benchmark policies, including the hand-coded policy. A better policy was often found even in the first data point, representing the first 1000 episodes of learning. Qualitatively, the keepers appear to quickly learn roughly how long to hold the ball, and then gradually learn fine distinctions regarding when and to which teammate to pass. Due to the complex policy representation (thousands of weights), it is difficult to characterize objectively the extent to which the independent learners *specialize* or learn different, perhaps complementary, policies. Qualitatively, they seem to learn similar policies, perhaps in part as a result of the fact that they rotate randomly through the various start positions at the beginnings of the episodes, as described in Section 2.

It is important to note that the x-axis in Figure 10 represents real time. That is, when the

episodes last 10 seconds on average, the simulator executes just 360 episodes per hour. Therefore the players are learning to outperform the benchmark policies in hundreds of episodes and reaching their peak performance in just thousands of episodes. Especially given the enormity of the state space, we view these results as representing fast, successful learning on the part of the Sarsa(λ) algorithm.

We also applied our learning algorithm to learn policies for a slightly larger task, 4 vs. 3 keepaway. Figure 11 shows that the keepers learned a policy that outperform all of our benchmarks in 4v3 keepaway in a 30x30 region. In this case, the learning curves still appear to be rising after 40 hours: more time may be needed to realize the full potential of learning.

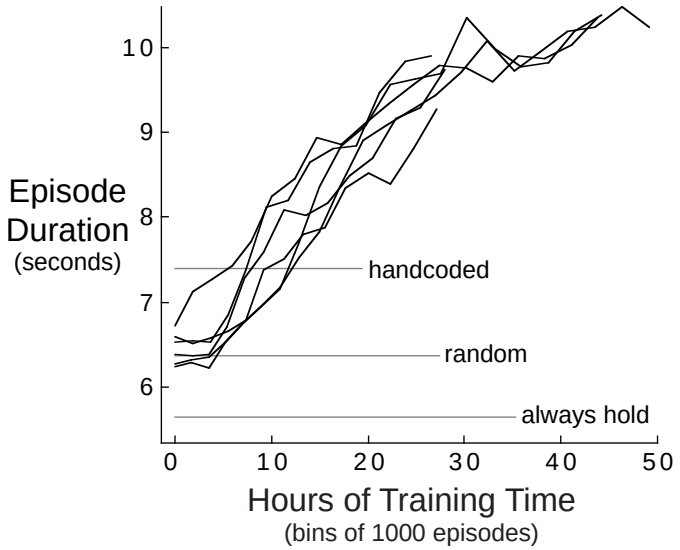


Figure 11: Multiple successful runs under identical characteristics: 4v3 keepaway in a 30x30 region against hand-coded takers.

5.2 Follow-up Questions

The initial results in Section 5.1 represent the main result of this work. They demonstrate the power and robustness of distributed SMDP Sarsa(λ) with linear tile-coding function approximation and variable λ. However, as is not uncommon, these positive results lead us to many additional questions, some of which we consider here. In particular:

1. **Does the learning approach described above continue to work if the agents are limited to noisy, narrowed vision?**

Our initial complete, noiseless vision simplification was convenient for two reasons. First, the learning agents had no uncertainty about the state of the world, effectively changing the world from partially observable to fully observable. Second, as a result, the agents did not need to incorporate any information-gathering actions into their policies, thus simplifying them considerably. However, for these techniques to scale up to more difficult problems such as RoboCup soccer, agents must be able to learn using sensory information that is often incomplete and noisy.

2. **How does a learned policy perform in comparison to a hand-coded policy that has been manually tuned?**

Our initial results compared learned policies to a hand-coded policy that was not tuned at all. This policy was able to perform only slightly better than **Random**. Also, it used a previously learned decision tree for pass evaluation, so it was not 100% "hand-coded" (Stone & Sutton, 2001). Although the need for manual tuning of parameters is precisely what we try to avoid by using machine learning, to assess properly the value of learning, it is important to compare the performance of a learned policy to that of a benchmark that has been carefully thought out. In particular, we seek to discover here whether learning as opposed to hand-coding policies *(i)* leads to a superior solution; *(ii)* saves effort but produces a similarly effective solution; or *(iii)* trades off manual effort against performance.

3. **How robust are these methods to differing field sizes?**

The cost of manually tuning a hand-coded policy can generally be tolerated for single problems. However, having to retune the policy every time the problem specification is slightly changed can become quite cumbersome. A major advantage of learning is that, typically, adapting to variations in the learning task requires little or no modification to the algorithm. Reinforcement learning becomes an especially valuable tool for RoboCup soccer if it can han-

dle domain alterations more robustly than hand-coded solutions.

4. **How dependent are the results on the state representation?**

   The choice of input representation can have a dramatic effect on the performance and computation time of a machine learning solution. For this reason, the representation is typically chosen with great care. However, it is often difficult to detect and avoid redundant and irrelevant information. Ideally, the learning algorithm would be able to detect the relevance of its state variables on its own.

5. **How dependent are the results on using Sarsa rather than Q-learning?**

   Throughout this article, we use a variant of the Sarsa reinforcement learning algorithm, an on-policy learning method that learns values based on the current actual policy. The off-policy variant, Q-learning, is similar except that it learns values based on the control policy that always selects the action with the (currently) highest action value. While Q-learning is provably convergent to the optimal policy under restrictive conditions (Watkins, 1989), it can be unstable with linear and other kinds of function approximation. Nonetheless, Q-learning has proven to be robust at least in some large partially observable domains, such as elevator scheduling (Crites & Barto, 1996). Thus, it is important to understand the extent to which our successful results can be attributed to using Sarsa rather than Q-learning.

6. **How well do the results scale to larger problems?**

   The overall goal of this line of research is to develop reinforcement learning techniques that will scale to 11 vs. 11 soccer on a full-sized field. However, previous results in the keepaway domain have typically included just 3 keepers and at most 2 takers. The largest learned keepaway solution that we know of is in the 4 vs. 3 scenario presented in Section 5.1. This

research examines whether current methods can scale up beyond that.

7. **Is the source of the difficulty the learning task itself, or the fact that multiple agents are learning simultaneously?**

   Keepaway is a multiagent task in which all of the agents learn simultaneously and independently. On the surface, it is unclear whether the learning challenge stems mainly from the fact that the agents are learning to interact with one another, or mainly from the difficulty of the task itself. For example, perhaps it is just as hard for an individual agent to learn to collaborate with previously trained experts as it is for the agent to learn simultaneously with other learners. Or, on the other hand, perhaps the fewer agents that are learning, and the more that are pre-trained, the quicker the learning happens. We explore this question experimentally.

5.3 Detailed Studies

This section addresses each of the questions listed in Section 5.2 with focused experiments in the keepaway domain.

5.3.1 Limited Vision

Limited vision introduces two challenges with respect to the complete vision setup of Section 5.1 and Stone and Sutton (2001). First, without complete knowledge of the world state, agents must model the uncertainty in their knowledge and make the appropriate decisions based on those uncertainties. Second, the agents must occasionally take explicit information-gathering actions to increase their confidence in the world state, thus complicating their action policies.

To keep track of the uncertainty in its world state, a player stores a confidence value along with each state variable. When the player receives sensory information, it updates its world state and sets its confidence in those values to 1.0. Each time step in which the player receives no new information about a variable, the variable's confidence is multiplied by a decay rate (0.99 in our experiments). When the confidence falls below a threshold (0.5), the value is no longer considered reliable.

We assume that if the keeper with the ball does not have reliable information about the position of its teammates, then it would almost certainly do more harm than good by trying to make a blind pass. Therefore, in this situation, we force the keeper to perform a safe default action while it is gathering enough sensory information to compute the values of all of its state variables. A keeper's default action is to invoke HoldBall() and turn its neck to try to locate its teammates and the opponents. This action persists until the keeper knows the positions of itself and all of its teammates so as to be able to make an informed decision. These information-gathering time steps are not treated as SMDP steps: no action choices are made.

Using this method, we attempted to reproduce the results reported in Figure 10 but without the simplification of unrestricted vision. In Figure 10, keepers were able to learn policies with average episode durations of around 15 seconds. However, learning with noisy, narrowed vision is a more difficult problem than learning with complete knowledge of the state. With incomplete information about the ball position, the ball becomes more difficult to intercept; and with incomplete information about teammate positions, passes become more difficult to execute. For this reason, we expected our learners to hold the ball for less than 15 seconds. However, these same difficulties impact the benchmark policies as well. So the salient question is whether or not learning is still able to outperform the benchmarks.

We ran a series of 6 independent learning trials in which the keepers learned while playing against the hand-coded takers. In each run, the keepers gradually improved their performance before leveling off after about 25 hours of simulator time.[9] The learning curves are shown in Figure 12. We plotted all 6 trials to give a sense of the variance.

The keepers start learning from a random policy that is able to maintain possession for about 6 seconds per episode on average. After 25 hours (simulator time) of training, they are able hold the ball an average of 9.6s to 11.1s. Note that as in Section 5.1, one hour of simulator time represents just 600 10-second episodes. Thus the learning occurs over just a few thousand episodes. All of the learning runs were able to outperform the **Always Hold** and **Random** benchmark policies which had average possession times of 4.9 and 6.1 seconds, respectively. The learned policies also

[9] By default, each cycle in the RoboCup simulator lasts 100ms. "Simulator time" is simply the number of cycles multiplied by 100ms. For these experiments, however, we exploited the simulator's "synchronous mode" to speed up the experiments by 5 or 6 times. Because cycle durations are inconsistent in this mode, we report all times in simulator time rather than actual time.

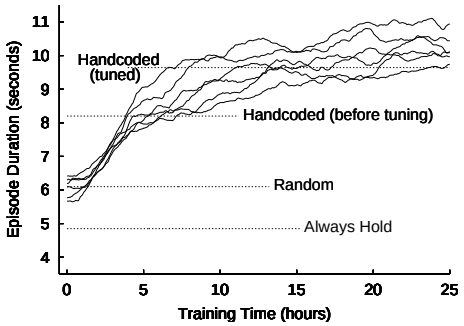


Figure 12: Learning curves for 3 keepers playing against 2 takers on a $20m \times 20m$ field along with several benchmarks.

outperform our **Hand-coded** policy which we describe in detail in the next section.

5.3.2 Comparison to Hand-coded

In addition to the **Always Hold** and **Random** benchmark policies described previously, we compared our learners to a **Hand-coded** policy. [10] In this policy, the keeper in possession, K_1 assigns a score, V_i, to each of its teammates based on how "open" they are. The degree to which the player is open is calculated as a linear combination of the teammate's distance to its nearest opponent, and the angle between the teammate, K_1, and the opponent closest to the passing line. The relative importance of these two features are weighted by the coefficient α. If the most open teammate has a score above the threshold, β, then K_1 will pass to this player. Otherwise, K_1 will invoke HoldBall() for one cycle.

Figure 5.3.2 shows pseudo-code for the **Hand-coded** policy. It has been designed to use only state variables and calculations that are available to the learner. Our initial values, based on educated guesses, for α and β were 3 and 50, respectively. We tuned these values by experimenting with values near our initial guesses, trying α values between 2 and 4.5 and β values between 30 and 100. Altogether, we tried about 30 combinations, eventually finding the best performance at $\alpha = 4$ and $\beta = 90$.

We ran a few thousand episodes of our tuned **Hand-coded** policy and found that it was able to keep the ball for an average of 9.6 seconds per episode. Also, for comparison, we tested our

[10] The creation and tuning of the hand-coded behavior was the result of a good-faith effort on the part of a motivated student to generate the strongest possible policy without the aid of any automated testing.

Hand-coded:

If no taker is within $4m$ (i.e. $dist(K_1, T_1) > 4$) Then

    HoldBall()

Else

    For $i \in [2, n]$

        $V_i \leftarrow$Min$(ang(K_i, K_1, T_1), ang(K_i, K_1, T_2)) +$

            α $*$ Min$(dist(K_i, T_1), dist(K_i, T_2))$

    $I \leftarrow \arg\max_i V_i$

    If $V_I > \beta$ Then

        PassBall(K_I)

    Else

        HoldBall()

Figure 13: The **Hand-coded** policy.

Hand-coded policy before manual tuning (i.e. α set to 3 and β set to 50). This policy was able to hold the ball for an average of 8.2 seconds. From Figure 12 we can see that the keepers are able to learn policies that outperform our initial **Hand-coded** policy and exhibit performance roughly as good as (perhaps slightly better than) the tuned version.

Figure 12 also shows that, with some fine tuning, it is possible to create a fairly simple hand-coded policy that is able to perform almost as well as a learned policy. On the one hand, it is disappointing that learning does not vastly outperform a tuned hand-coded solution as it did the initial hand-coded solution. But on the other hand, it is promising that the learned solution is at least as good as the tuned, hand-coded approach. We examined the **Hand-coded** policy further to find out to what degree its performance is dependent on tuning.

5.3.3 Robustness to Differing Field Sizes

In our preliminary work, we demonstrated that learning is robust to changes in field sizes, albeit under conditions of unrestricted vision (Stone & Sutton, 2001). Here we verify that learning is

still robust to such changes even with the addition of significant state uncertainty, and we also benchmark these results against the robustness of the **Hand-coded** policy to the same changes. Overall, we expect that as the size of the play region gets smaller, the problem gets more difficult and the keepers have a harder time maintaining possession of the ball regardless of policy. Here we compare the **Hand-coded** policy to learned policies on five different field sizes. The average episode durations for both solutions are shown in Table 1. Each value for the learned runs was calculated as an average of six separately learned policies. The standard deviation is reported along with the mean.

As can be seen from the table, the hand-coded policy does better on the easier problems ($30m \times 30m$ and $25m \times 25m$), but the learned policies do better on the more difficult problems.

	Keeper Policy	
Field Size	Hand-coded	Learned ($\pm 1\sigma$)
30×30	**19.8**	18.2 ± 1.1
25×25	**15.4**	14.8 ± 0.3
20×20	9.6	$\mathbf{10.4 \pm 0.4}$
15×15	6.1	$\mathbf{7.4 \pm 0.9}$
10×10	2.7	$\mathbf{3.7 \pm 0.4}$

Table 1: Comparison of average possession times (in simulator seconds) for hand-coded and learned policies on various field sizes.

A possible explanation for this result is that the easier cases of keepaway have more intuitive solutions. Hence, these problems lend themselves to a hand-coded approach. When the field is large, and the takers both charge directly for the ball, the obvious solution is to wait until the takers are fairly close to the ball, then pass the ball to the teammate whose passing lane is not being covered. If this behavior is repeated with the proper timing, the ball can be kept almost indefinitely. The tuned hand-coded policy appears to qualitatively exhibit this behavior on the larger field sizes. However, without any impetus to choose a simpler approach over a more complicated one, learned policies tend to appear more asymmetric and irregular. This lack of bias towards the straightforward solution may lead to suboptimal performance on easier tasks.

In contrast, when the keepers are forced to play in a smaller area, the intuitive solution breaks down. The hand-coded keepers tend to pass too frequently, leading to missed passes. In these more difficult tasks, the learned keepers appear to find "safer" solutions in which the ball is held

for longer periods of time. Learned policies are hard to characterize, but in general, the keeper in possession waits until the takers are about to converge on the ball from both sides. Then, it quickly spins the ball around and makes a pass to one of its two teammates both of which tend to be clumped together in the opposite side of the field. Even if the intended receiver misses the pass, the secondary receiver has a very good chance of reaching it before either taker. This approach leads to fewer missed passes and better overall performance than the hand-coded solution.

5.3.4 Changing the State Representation

A frequent challenge in machine learning is finding the correct state representation. In all of the experiments reported so far, we have used the same state variables, which were chosen without any detailed exploration. Here we explore how sensitive the learning is to the set of state variables used. Ideally, if it is not particularly sensitive to these variables, then we can avoid detailed searches in this part of representation space.[11]

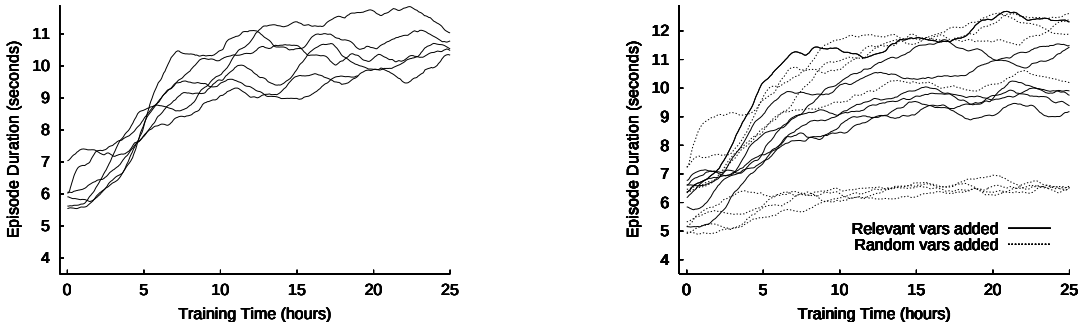
As a starting point, notice that our **Hand-coded** policy uses only a small subset of the 13 state variables mentioned previously. For 3 keepers and 2 takers, the 5 variables are:

- $dist(K_1, T_1)$;
- $\text{Min}(dist(K_2, T_1), dist(K_2, T_2))$;
- $\text{Min}(dist(K_3, T_1), dist(K_3, T_2))$;
- $\text{Min}(ang(K_2, K_1, T_1), ang(K_2, K_1, T_2))$;
- $\text{Min}(ang(K_3, K_1, T_1), ang(K_3, K_1, T_2))$.

Because the **Hand-coded** policy did quite well without using the remaining variables, we wondered if perhaps the unused state variables were not essential for the keepaway task.

To test this theory, we performed a series of learning runs in which the keepers used only the five variables from the hand-coded policy. The takers followed the **Hand-coded-T** policy as before.

[11] As indicated in Figure 7, the state variables are just one part of building the state representation. From the point of view of the learning algorithm, the real features come from the binary feature vector F_a. However, from a practical point of view, we do not usually manipulate these features individually. Rather, we create them from the continuous state variables using CMACs. Thus our experimentation in this section is not directly in feature space. But it does touch on one of the most direct places in which human knowledge is injected in our learning process.



(a) (b)

Figure 14: **(a)** Keepers learning with only the 5 state variables from the **Hand-coded** policy. **(b)** Learning with the original 13 state variables plus an additional two.

Figure 14(a) shows the learning curves for six runs. As is apparent from the graph, the results are very similar to those in Figure 12. Although we found that the keepers were able to achieve better than random performance with as little as one state variable, the five variables used in the hand-coded policy seem to be minimal for peak performance. Notice by comparing Figures 12 and 14(a) that the keepers are able to learn at approximately the same rate whether the nonessential state variables are present or not. The learner seems not to be deterred by the presence of extraneous state variables.

To explore this notion further, we tried adding additional state variables to the original 13. We ran two separate experiments. In the first experiment, we added 2 new angles that appeared relevant but perhaps redundant:

- $ang(K_1, C, K_2)$
- $ang(K_1, C, K_3)$

In the second experiment, we added 2 completely irrelevant variables: each time step, new values were randomly chosen from [-90,90] with uniform probability. We performed several learning runs for both state representations and plotted them all in Figure 14(b).

From the graph, we can see that the learners are not greatly affected by the addition of relevant variables. The learning curves look roughly the same as the ones that used the original 13 state variables (Figure 12). However, the curves corresponding to the additional random variables look somewhat different. The curves can clearly be divided into two groups. In the first group, teams are able to perform about as well as the ones that used the original 13 variables. In the second

group, the agents perform very poorly. It appears that agents in the second group are confused by the irrelevant variables while the agents in the first group are not. This distinction seems to be made in the early stages of learning (before the 1000th episode corresponding to the first data point on the graph). The learning curves that start off low stay low. The ones that start off high continue to ascend.

From these results, we conclude that it is important to choose relevant variables for the state representation. However, it is unnecessary to carefully choose the minimum set of these variables.

5.3.5 Comparison to Q-learning

To understand the extent to which our successful results can be attributed to our choice of reinforcement learning algorithm, we compared our results to those achieved by another popular algorithm, Q-learning. Unlike Sarsa, which uses the same policy for control and updates, Q-learning learns values based on the control policy that always selects the action with the (currently) highest action value. [12]

To implement Q-learning, we must make a subtle, yet important, change to the Sarsa algorithm detailed earlier in Figure 8. In both `RLstartEpisode` and `RLstep`, lines 4 and 14, a macro-action is selected by the ϵ-greedy method to be executed by the agent and to be stored in *LastAction* for future updates. In Q-learning, the agent continues to use this action for control, but now stores the action with the highest action value in *LastAction* for updating. This is accomplished by replacing lines 4 and 14 each with the following two lines:

$$SelectedAction \leftarrow \begin{cases} \arg\max_a Q_a & \text{w/prob. } 1-\epsilon \\ \text{random action} & \text{w/prob. } \epsilon \end{cases}$$

$$LastAction \leftarrow \arg\max_a Q_a$$

In `RLstartEpisode` and `RLstep` the action to be executed is changed from *LastAction* to *SelectedAction*.

We performed five runs of the Q-learning algorithm for 3 keepers and 2 takers in a 20x20 region. The learning curves for these runs are plotted in Figure 15 along with our results using Sarsa under the same conditions. From the graph, we can see that Q-learning takes more time to converge than Sarsa, requiring 40–50 hours of learning time versus 15–20. Also, the policies learned by Q-learning have a higher variability in performance across the runs. This may be attributed to the instability

[12] See Chapters 6 and 7 of (Sutton & Barto, 1998) for a more detailed comparison of Sarsa and Q-learning.

of Q-learning when using function approximation. Finally, the graph shows that, with unlimited learning time, the best policies found by Q-learning perform about as well as those learned by Sarsa in this task.

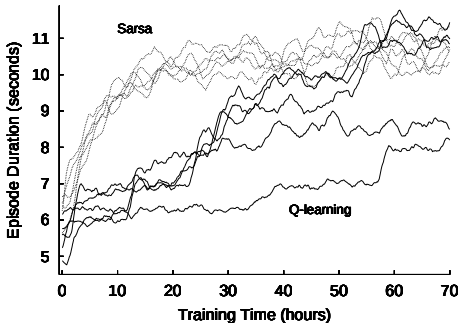


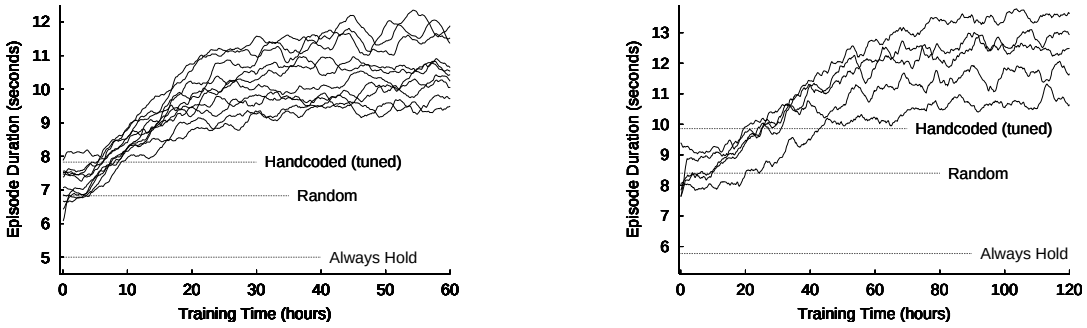
Figure 15: Learning curve comparison for Q-learning and Sarsa(λ).

5.3.6 Scaling to Larger Problems

In addition to our experiments with 3 vs. 2 keepaway with limited vision, we ran a series of trials with larger team sizes to determine how well our techniques scale. First we performed several learning runs with 4 keepers playing against 3 hand-coded takers. We compared these to our three benchmark policies. The results are shown in Figure 16(a). As in the 3 vs. 2 case, the players are able to learn policies that outperform all of the benchmarks. These results again serve to verify that SMDP Sarsa(λ) is able to learn this task even with limited vision.

We also ran a series of experiments with 5 vs. 4 keepaway. The learning curves for these runs along with our three benchmarks are shown in Figure 16(b). Again, the learned policies outperform all benchmarks. As far as the authors are aware, these experiments represent the largest scale keepaway problems that have been successfully learned to date.

From these graphs, we see that the learning time approximately doubles every time we move up in size. In 3 vs. 2, the performance plateaus after roughly (by eyeballing the graphs) 15 hours of training. In 4 vs. 3, it takes about 30 hours to learn. In 5 vs. 4, it takes about 70 hours.



(a) (b)

Figure 16: **(a)** Training 4 Keepers against 3 takers with benchmarks. **(b)** Training 5 Keepers against 4 takers with benchmarks.

5.3.7 Difficulty of Multiagent Learning

A key outstanding question about keepaway is whether it is difficult as an individual learning task, or if the multiagent component of the problem is the largest source of difficulty. To see how the number of agents learning simultaneously affects the overall training time, we ran a series of experiments in which a subset of the keepers learned while its remaining teammates followed a fixed policy learned previously. In each run, 3 keepers played against 2 hand-coded takers.

We began by training all of the keepers together until their learning curves appeared to flatten out. We then fixed two of them, and had the third learn from random initial conditions. Finally, we allowed one keeper to continue to use its learned policy while the other two learned from scratch. We ran each experiment three times. The learning curves for all nine runs are shown in Figure 17.

From the graph we can see that the learning curves for 2 learning agents and 3 learning agents look roughly the same. However, the runs with only 1 player learning peak much sooner. Apparently, having pre-trained teammates allows an agent to learn much faster. However, if more than one keeper is learning, the presence of a pre-trained teammate is not helpful. This result suggests that multiagent learning is an inherently more difficult problem than single agent learning, at least for this task. In the long run, all three configurations' learned policies are roughly equivalent. The number of learning agents does not seem to affect the quality of the policy, only the rate at which the policy is learned.

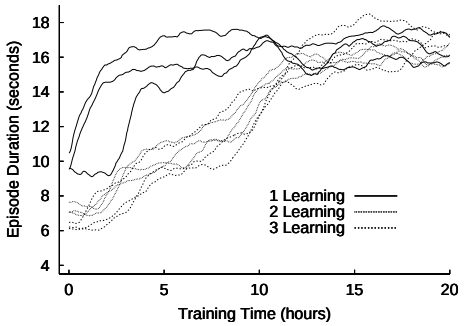


Figure 17: Learning curves for varying number of keepers learning simultaneously.

6 Related Work

Reinforcement learning has been previously applied to robot soccer. Using real robots, Uchibe (1999) used reinforcement learning methods to learn to shoot a ball into a goal while avoiding an opponent. This task differs from keepaway in that there is a well-defined goal state. In more recent research on the same task, they used options with (probabilistic) termination constraints reminiscent of the macro-actions used in this article (Uchibe, Yanase, & Asada, 2001). Also using goal-scoring as the goal state, TPOT-RL (Stone & Veloso, 1999) was successfully used to allow a full team of agents to learn collaborative passing and shooting policies using a Monte Carlo learning approach, as opposed to the TD learning explored in this article. Andou's (1998) "observational reinforcement learning" was used for learning to update players' positions on the field based on where the ball has previously been located.

Perhaps most related to the work reported here, Riedmiller et al. (2001) use reinforcement learning to learn low-level skills ("moves"), such as kicking, ball-interception, and dribbling, as well as a cooperative behavior in which 2 attackers try to score against one or two takers. In contrast to our approach, this work uses the full sensor space as the input representation, with a neural network used as a function approximator. The taker behaviors were always fixed and constant, and no more than 2 attackers learned to cooperate. More recently, this approach was scaled up to 3 attackers against 4 defenders in several different scoring scenarios (Riedmiller, Merke, Hoffmann, Withopf, Nickschas, & Zacharias, 2003). This research is also part of an on-going effort to implement a full soccer team through reinforcement learning techniques.

Distributed reinforcement learning has been explored previously in discrete environments, such

as the pursuit domain (Tan, 1993) and elevator control (Crites & Barto, 1996). The keepaway task differs from both of these applications in that keepaway is continuous, that it is real-time, and that there is noise both in agent actions and in state-transitions.

One of reinforcement learning's previous biggest success stories is Tesauro's TD-Gammon (1994) which achieved a level of performance at least as good as that of the best human players in the game of backgammon. Like soccer keepaway, backgammon is a stochastic domain with a large state space. However, keepaway also includes the challenges of perceptual and actuator noise, distributed learning, and continuous state variables.

There has been considerable recent effort in the field of reinforcement learning on scaling up to larger problems by decomposing the problem hierarchically (e.g., (Dietterich, 2000; Andre & Russell, 2001)) or by exploiting factored representations (e.g., (Boutilier, Dean, & Hanks, 1999; Koller & Parr, 1999; Guestrin, Koller, & Parr, 2001)). Our approach to keepaway is hierarchical in some respect, given that the low-level primitive actions are pre-composed into more abstract actions and temporally extended macro-actions. However, in our case, the primitive actions are not learned: from a learning perspective, we are in a completely "flat" scenario. The main leverage for both factored and hierarchical approaches is that they allow the agent to ignore the parts of its state that are irrelevant to its current decision (Andre & Russell, 2002). Although our state variables can be seen as a factoring of the state space, there are no independencies among the variables such that actions affect only subsets of the state variables. Thus, existing factored approaches are not directly applicable. We believe that it is still a very interesting topic of research to try to scale hierarchical and factored methods to work well tabula rasa in such a complex environment as the one we are considering. However, the empirical results with these methods on large-scale problems have been scarce. By focusing on learning one part of the problem with a flat technique, we have been able to achieve successful results on a very large-scale problem (with respect to the size of the state space), despite the lack of useful factorings of the state space.

Machine learning techniques other than reinforcement learning have also been applied successfully in the RoboCup domain. There have been two attempts to learn the entire simulated RoboCup task via genetic programming (GP) (Luke et al., 1998; Andre & Teller, 1999). While both efforts were initially intended to test the ability of GP to scale to the full, cooperative robot soccer task, the first system ended up evolving over hand-coded low-level behaviors, and the second achieved

some successful individual behaviors but was unable to generate many collaborative team behaviors. Whether this approach can be scaled up to produce more successful teams remains to be seen.

Neural networks and decision trees have been used to address various soccer subtasks such as deciding whether to pass or shoot near the goal (Noda, Matsubara, & Hiraki, 1996), and learning how to shoot and intercept the ball (Marsella, Tambe, Adibi, Al-Onaizan, Kaminka, & Muslea, 2001). A hierarchical paradigm called layered learning was used to combine a ball-interception behavior trained with a back-propagation neural network; a pass-evaluation behavior trained with the C4.5 decision tree training algorithm (Quinlan, 1993); and a pass-decision behavior trained with TPOT-RL (mentioned above) into a single, successful team (Stone, 2000).

Several previous studies have used keepaway soccer as a machine learning testbed. Whiteson and Stone (2003) used neuroevolution to train keepers in the SoccerBots domain (Balch, 2000b). The players were able to learn several conceptually different tasks from basic skills to higher-level reasoning using a hierarchical approach they call "concurrent layered learning." A hand-coded decision tree was used at the highest level. The keepers were evaluated based on the number of completed passes. Hsu and Gustafson (2002) evolved keepers for 3 vs. 1 keepaway in the much simpler and more abstract TeamBots simulator (Balch, 2000a). In this domain, players move around in a coarse grid and execute discrete actions. The takers move twice as quickly as the keepers and the ball moves twice as quickly as the takers. Keepers were trained to minimize the number of turnovers in fixed duration games. It is difficult to compare these approaches to ours because they use different fitness functions and different game dynamics.

More comparable work to ours applied evolutionary algorithms to train 3 keepers against 2 takers in the RoboCup soccer simulator (Pietro, While, & Barone, 2002). Similar to our work, they focused on learning keepers in possession of the ball. The keepers chose from the same high-level behaviors as ours. Also, they used average episode duration to evaluate keeper performance. However, because their high-level behaviors and basic skills were implemented independently from ours, it is difficult to compare the two learning approaches empirically.

In conjunction with the research reported here, we have explored techniques for keepaway in a full 11 vs. 11 scenario played on a full-size field (McAllester & Stone, 2001). The successful hand-coded policies were incorporated into ATT-CMUnited-2000, the 3rd-place finisher in the

RoboCup-2000 simulator competition.

ATT-CMUnited-2000 agents used an action architecture that is motivated by a desire to facilitate reinforcement learning over a larger, more flexible action space than is considered in this article: actions are parameterized such that hundreds of actions are considered at a time (Stone & McAllester, 2001). The agents select actions based on their perceived values and success probabilities. However, the value function is tuned manually.

7 Conclusion

This article presents an application of episodic SMDP Sarsa(λ) with linear tile-coding function approximation and variable λ to a complex, multiagent task in a stochastic, dynamic environment. With remarkably few training episodes, simultaneously learning agents achieve significantly better performance than a range of benchmark policies, including a reasonable hand-coded policy, and comparable performance to a tuned hand-coded policy. Although no known theoretical results guarantee the success of Sarsa(λ) in this domain, in practice it performs quite well.

Taken as a whole, the experiments reported in this article demonstrate the possibility of multiple independent agents learning simultaneously in a complex environment using reinforcement learning after a small number of trials. The main contribution is an empirical possibility result and success story for reinforcement learning.

Our on-going research aims to build upon the research reported in this article in many ways. In the long-term, we aim to scale up to the full RoboCup soccer task and to enable learned behaviors to outperform the best current competition entries, at least in the RoboCup simulation league, and ideally in one of the real robot leagues as well. In parallel, we aim to explore the application of the current techniques as reported here to other large-scale multiagent real-time domains, such as distributed training simulations.

Meanwhile, there are three more immediate, short-term goals along the path to our ultimate aim. First, though we have some moderately successful results at taker learning, we aim at improving the ability of the takers to learn by altering their representation and/or learning parameters. One promising line of inquiry is into the efficacy of alternately training the takers and the keepers against each other so as to improve both types of policies.

Second, while the formulation of keepaway presented in this article includes an enormous state

space, the action space is quite limited. An important direction for future research is to explore whether reinforcement learning techniques can be extended to keepaway with large, discrete, or continuous, parameterized action spaces, perhaps using policy gradient methods (Sutton et al., 2000). For example, the agents could learn where to move when not in possession of the ball or they could learn *direction* in which to pass as opposed to the *player* to which to pass. This latter possibility would enable passes in front of a teammate so that it can move to meet the ball.

Third, when training in a new, but related environment (such as a different field size or a different number of players), one alternative is always to train a completely new behavior from scratch, as we have done throughout this article. However, another alternative is to begin training from a previously learned behavior. We plan to investigate the extent to which behavior transfer of this form is feasible and beneficial using the learning approach presented in this article. Preliminary work in this direction demonstrates the utility of transferring learned behaviors from 3v2 to 4v3 and 5v4 scenarios (Taylor & Stone, 2005).

In conjunction with the research reported in this article, by having incorporated the substrate domain for this research—keepaway—into the publicly available, open-source distribution of the RoboCup soccer simulator and by making player source code and tutorials for implementing keepaway agents available at `http://www.cs.utexas.edu/~AustinVilla/sim/keepaway/`, we hope to encourage additional research on learning in the keepaway domain. We believe that keepaway is a promising benchmark problem for machine learning algorithms (Stone & Sutton, 2002). While Sarsa(λ) has shown promising results, it may not be the final answer.

Acknowledgments

We thank Satinder Singh for his assistance in formulating our reinforcement learning approach to keepaway; Patrick Riley and Manuela Veloso for their participation in the creation of the CMUnited-99 agent behaviors that were used as a basis for our agents; Tom Miller and others at the University of New Hampshire for making available their CMAC code; Anna Koop for editing suggestions on the final draft; and the anonymous reviewers for their helpful comments and suggestions. This research was supported in part by NSF CAREER award IIS-0237699 and DARPA award HR0011-04-1-0035.

References

Albus, J. S. (1981). *Brains, Behavior, and Robotics*. Byte Books, Peterborough, NH.

Andou, T. (1998). Refinement of soccer agents' positions using reinforcement learning. In Kitano, H. (Ed.), *RoboCup-97: Robot Soccer World Cup I*, pp. 373–388. Springer Verlag, Berlin.

Andre, D., & Russell, S. J. (2001). Programmable reinforcement learning agents. In *Advances in Neural Information Processing Systems 13*, pp. 1019–1025.

Andre, D., & Russell, S. J. (2002). State abstraction for programmable reinforcement learning agents. In *Proceedings of the 18th National Conference on Artificial Intelligence*.

Andre, D., & Teller, A. (1999). Evolving team Darwin United. In Asada, M., & Kitano, H. (Eds.), *RoboCup-98: Robot Soccer World Cup II*. Springer Verlag, Berlin.

Bagnell, J. A., & Schneider, J. (2001). Autonomous helicopter control using reinforcement learning policy search methods. In *International Conference on Robotics and Automation*.

Baird, L. C., & Moore, A. W. (1999). Gradient descent for general reinforcement learning. In *Advances in Neural Information Processing Systems*, Vol. 11. The MIT Press.

Balch, T. (2000a). Teambots.. `http://www.teambots.org`.

Balch, T. (2000b). Teambots domain: Soccerbots.. `http://www-2.cs.cmu.edu/~trb/TeamBots/Domains/SoccerBots`.

Boutilier, C., Dean, T., & Hanks, S. (1999). Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, *11*, 1–94.

Bradtke, S. J., & Duff, M. O. (1995). Reinforcement learning methods for continuous-time Markov decision problems. In G. Tesauro, D. Touretzky, T. L. (Ed.), *Advances in Neural Information Processing Systems 7*, pp. 393–400 San Mateo, CA. Morgan Kaufmann.

Chen, M., Foroughi, E., Heintz, F., Kapetanakis, S., Kostiadis, K., Kummeneje, J., Noda, I., Obst, O., Riley, P., Steffens, T., Wang, Y., & Yin, X. (2003). Users manual: RoboCup soccer server manual for soccer server version 7.07 and later.. Available at `http://sourceforge.net/projects/sserver/`.

Crites, R. H., & Barto, A. G. (1996). Improving elevator performance using reinforcement learning. In Touretzky, D. S., Mozer, M. C., & Hasselmo, M. E. (Eds.), *Advances in Neural Information Processing Systems 8* Cambridge, MA. MIT Press.

Dean, T., Basye, K., & Shewchuk, J. (1992). Reinforcement learning for planning and control. In Minton, S. (Ed.), *Machine Learning Methods for Planning and Scheduling*. Morgan Kaufmann.

Dietterich, T. G. (2000). Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of Artificial Intelligence Research*, *13*, 227–303.

Gordon, G. (2001). Reinforcement learning with function approximation converges to a region. In *Advances in Neural Information Processing Systems*, Vol. 13. The MIT Press.

Guestrin, C., Koller, D., & Parr, R. (2001). Multiagent planning with factored mdps. In *Advances in Neural Information Processing Systems 14*, pp. 1523–1530.

Hsu, W. H., & Gustafson, S. M. (2002). Genetic programming and multi-agent layered learning by reinforcements. In *Genetic and Evolutionary Computation Conference* New York,NY.

Kitano, H., Tambe, M., Stone, P., Veloso, M., Coradeschi, S., Osawa, E., Matsubara, H., Noda, I., & Asada, M. (1997). The RoboCup synthetic agent challenge 97. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pp. 24–29 San Francisco, CA. Morgan Kaufmann.

Koller, D., & Parr, R. (1999). Computing factored value functions for policies in structured mdps. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*.

Lin, C.-S., & Kim, H. (1991). CMAC-based adaptive critic self-learning control. In *IEEE Trans. Neural Networks*, Vol. 2, pp. 530–533.

Luke, S., Hohn, C., Farris, J., Jackson, G., & Hendler, J. (1998). Co-evolving soccer softbot team coordination with genetic programming. In Kitano, H. (Ed.), *RoboCup-97: Robot Soccer World Cup I*, pp. 398–411 Berlin. Springer Verlag.

Marsella, S., Tambe, M., Adibi, J., Al-Onaizan, Y., Kaminka, G. A., & Muslea, I. (2001). Experiences acquired in the design of RoboCup teams: a comparison of two fielded teams. *Autonomous Agents and Multi-Agent Systems*, *4*(2), 115–129.

McAllester, D., & Stone, P. (2001). Keeping the ball from CMUnited-99. In Stone, P., Balch, T., & Kraetszchmar, G. (Eds.), *RoboCup-2000: Robot Soccer World Cup IV*. Springer Verlag, Berlin.

Noda, I., Matsubara, H., & Hiraki, K. (1996). Learning cooperative behavior in multi-agent environment: a case study of choice of play-plans in soccer. In *PRICAI'96: Topics in Artificial Intelligence (Proc. of 4th Pacific Rim International Conference on Artificial Intelligence)*, pp. 570–579 Cairns, Australia.

Noda, I., Matsubara, H., Hiraki, K., & Frank, I. (1998). Soccer server: A tool for research on multiagent systems. *Applied Artificial Intelligence*, *12*, 233–250.

Perkins, T. J., & Precup, D. (2003). A convergent form of approximate policy iteration. In Becker, S., Thrun, S., & Obermayer, K. (Eds.), *Advances in Neural Information Processing Systems 16* Cambridge, MA. MIT Press.

Pietro, A. D., While, L., & Barone, L. (2002). Learning in RoboCup keepaway using evolutionary algorithms. In Langdon, W. B., Cantú-Paz, E., Mathias, K., Roy, R., Davis, D., Poli, R., Balakrishnan, K., Honavar, V., Rudolph, G., Wegener, J., Bull, L., Potter, M. A., Schultz, A. C., Miller, J. F., Burke, E., & Jonoska, N. (Eds.), *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 1065–1072 New York. Morgan Kaufmann Publishers.

Puterman, M. L. (1994). *Markov Decision Problems*. Wiley, NY.

Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA.

Riedmiller, M., Merke, A., Meier, D., Hoffman, A., Sinner, A., Thate, O., & Ehrmann, R. (2001). Karlsruhe brainstormers—a reinforcement learning approach to robotic soccer. In Stone, P., Balch, T., & Kraetszchmar, G. (Eds.), *RoboCup-2000: Robot Soccer World Cup IV*. Springer Verlag, Berlin.

Riedmiller, M., Merke, A., Hoffmann, A., Withopf, D., Nickschas, M., & Zacharias, F. (2003). Brainstormers 2002 - team description. In Kaminka, G. A., Lima, P. U., & Rojas, R. (Eds.), *RoboCup-2002: Robot Soccer World Cup VI*. Springer Verlag, Berlin.

Rummery, G. A., & Niranjan, M. (1994). On-line Q-learning using connectionist systems. Tech. rep. CUED/F-INFENG/TR 166, Cambridge University Engineering Department.

Stone, P. (2000). *Layered Learning in Multiagent Systems: A Winning Approach to Robotic Soccer*. MIT Press.

Stone, P., & McAllester, D. (2001). An architecture for action selection in robotic soccer. In *Proceedings of the Fifth International Conference on Autonomous Agents*, pp. 316–323.

Stone, P., & Sutton, R. S. (2001). Scaling reinforcement learning toward RoboCup soccer. In *Proceedings of the Eighteenth International Conference on Machine Learning*, pp. 537–544. Morgan Kaufmann, San Francisco, CA.

Stone, P., & Sutton, R. S. (2002). Keepaway soccer: a machine learning testbed. In Birk, A., Coradeschi, S., & Tadokoro, S. (Eds.), *RoboCup-2001: Robot Soccer World Cup V*, pp. 214–223. Springer Verlag, Berlin.

Stone, P., Sutton, R. S., & Singh, S. (2001). Reinforcement learning for 3 vs. 2 keepaway. In Stone, P., Balch, T., & Kraetszchmar, G. (Eds.), *RoboCup-2000: Robot Soccer World Cup IV*. Springer Verlag, Berlin.

Stone, P., & Veloso, M. (1999). Team-partitioned, opaque-transition reinforcement learning. In Asada, M., & Kitano, H. (Eds.), *RoboCup-98: Robot Soccer World Cup II*. Springer Verlag, Berlin. Also in *Proceedings of the Third International Conference on Autonomous Agents*, 1999.

Sutton, R. S. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding. In Touretzky, D. S., Mozer, M. C., & Hasselmo, M. E. (Eds.), *Advances in Neural Information Processing Systems 8*, pp. 1038–1044 Cambridge, MA. MIT Press.

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA.

Sutton, R., McAllester, D., Singh, S., & Mansour, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems*, Vol. 12, pp. 1057–1063. The MIT Press.

Sutton, R., Precup, D., & Singh, S. (1999). Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, *112*, 181–211.

Tan, M. (1993). Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the Tenth International Conference on Machine Learning*, pp. 330–337.

Taylor, M. E., & Stone, P. (2005). Behavior transfer for value-function-based reinforcement learning. In *The Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*. To appear.

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

Tsitsiklis, J. N., & Van Roy, B. (1997). An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, *42*, 674–690.

Uchibe, E. (1999). *Cooperative Behavior Acquisition by Learning and Evolution in a Multi-Agent Environment for Mobile Robots*. Ph.D. thesis, Osaka University.

Uchibe, E., Yanase, M., & Asada, M. (2001). Evolution for behavior selection accelerated by activation/termination constraints. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 1122–1129.

Veloso, M., Stone, P., & Bowling, M. (1999). Anticipation as a key for collaboration in a team of agents: A case study in robotic soccer. In *Proceedings of SPIE Sensor Fusion and Decentralized Control in Robotic Systems II*, Vol. 3839 Boston.

Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards*. Ph.D. thesis, King's College, Cambridge, UK.

Whiteson, S., & Stone, P. (2003). Concurrent layered learning. In *Second International Joint Conference on Autonomous Agents and Multiagent Systems*.

Peter Stone is an Alfred P. Sloan Research Fellow and Assistant Professor in the Department of Computer Sciences at the University of Texas at Austin. He received his Ph.D. in Computer Science in 1998 from Carnegie Mellon University and his B.S. in Mathematics from the University of Chicago in 1993. From 1999 to 2002 he worked at AT&T Labs. Peter is the author of "Layered Learning in Multiagent Systems: A Winning Approach to Robotic Soccer" (MIT Press, 2000). In 2003, he won a CAREER award from the National Science Foundation. He can be reached at `pstone@cs.utexas.edu`.

Richard S. Sutton is a fellow of the American Association for Artificial Intelligence and co-author of the textbook *Reinforcement Learning: An Introduction* from MIT Press. Before taking his current position as professor of computing science at the University of Alberta, he worked in industry at AT&T and GTE Labs, and in academia at the University of Massachusetts. He received a Ph.D. in computer science from the University of Massachusetts in 1984 and a BA in psychology from Stanford University in 1978. He can be reached at `rich@richsutton.com` or 2-21 Athabasca Hall, University of Alberta, Edmonton, AB, Canada T6G 2E8.

Gregory Kuhlmann is currently a Ph.D. student in the Department of Computer Sciences at The University of Texas at Austin. He received his B.S. in Computer Science and Engineering from U.C.L.A. in 2001. His research interests include machine learning, multiagent systems, and robotics. He can be reached at `kuhlmann@cs.utexas.edu`.