In Visser, Ribeiro, Ohashi, Dellaert, editors, *RoboCup-2007: Robot Soccer World Cup XI*, pp. 171--183, Springer Berlin / Heidelberg, 2008.

Model-based Reinforcement Learning in a Complex Domain

Shivaram Kalyanakrishnan, Peter Stone, and Yaxin Liu

Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-0233
{shivaram, pstone, yxliu}@cs.utexas.edu

Abstract. Reinforcement learning is a paradigm under which an agent seeks to improve its policy by making learning updates based on the experiences it gathers through interaction with the environment. *Model-free* algorithms perform updates solely bas ed on observed experiences. By contrast, *model-based* algorithms learn a model of the environment that effectively simulates its dynamics. The model may be used to simulate experiences or to plan into the future, potentially expediting the learning process. This paper presents a model-based reinforcement learning approach for Keepaway, a complex, continuous, stochastic, multiagent subtask of RoboCup simulated soccer. First, we propose the design of an environmental model that is partly learned based on the agent's experiences. This model is then coupled with the reinforcement learning algorithm to learn an action selection policy. We evaluate our method through empirical comparisons with model-free approaches that have been previously applied successfully to this task. Results demonstrate significant gains in the learning speed and asymptotic performance of our method. We also show that the learned model can be used effectively as part of a planning-based approach with a hand-coded policy.

1 Introduction

The reinforcement learning (RL) [12] problem is usually modeled as a Markov Decision Process (MDP) [10], which is of the form (S, A, R, T, γ). S is the set of states in the environment, and A the set of actions available to the agent. $R : S \times A \to \mathbb{R}$ is the reward function for the task: it returns the real number reward provided to the agent for taking an action from a given state. The dynamics of the environment are encapsulated in the transition function $T : S \times A \times S \to [0, 1]$; given a state and action, T returns a probability distribution over next states to which the agent may be transported. A (deterministic) policy $\pi : S \to A$ specifies the action to be taken by the agent from any given state. Every policy π can be associated with an action value function $Q : S \times A \to \mathbb{R}$ that computes the expected long-term discounted reward the agent will accrue by following π after taking some action a from some state S. $\gamma \in [0, 1]$ is a discount factor in the expected long-term reward. The problem is to solve for an *optimal* policy π, i.e., one that maximizes $\max_a Q^\pi(s, a)$ for every state s, defined by:

$$Q^\pi(s, a) = R(s, a) + \gamma \sum_{s' \epsilon S} T(s, a, s') Q^\pi(s', \pi(s')). \quad (1)$$

In most practical settings, the agent must act in the environment to gather *experiences*, using which it can improve its policy. An experience (or *transition*) is of the form (s, a, r, s'), where s is the agent's state, a an action taken from s, r the reward received, and s' the state to which the agent moves. Theoretical guarantees establish that under some conditions, the optimal policy can indeed be learned by making temporal difference updates based on the observed experiences, for instance, through methods like Q-learning [16]. Nonetheless, it is seldom possible in real world tasks to meet the conditions necessary for convergence. Solutions to complex tasks invariably have to adopt an engineering approach and exploit their underlying structure to the extent possible.

In this paper, we explore the potential of model-based methods in scaling RL to complex tasks. Whereas model-free methods like Q-learning interpret the policy directly through the action value function Q, model-based methods seek to decouple Q into its "components" T and R, termed the transition and reward models of the task respectively. By doing so, it becomes possible to use the *model* (T and R together) to *simulate* experiences that can be used to update Q, instead of solely relying on ones gathered from the environment. More specifically, the model can be used to explore parts of the state space that are possibly under-represented in the observed experiences. Hence, simulating experiences using the model can potentially improve the quality of the solution, while achieving economy in sample complexity. A further benefit gained from learning T and R individually is the advantage of separating the dynamics of the environment from the objective of the task at hand, offering the flexibility to share parts of the solution with different tasks in similar environments.

Model-based methods have been applied successfully in the past to several challenging problems. In domains such as game-playing, a partial or complete model of the environment is sometimes available, but determining the action selection policy can still be challenging owing to factors like the intractability of searching through the state space [14, 15]. On the other hand, for many real-world domains, learning the environmental model is itself a substantial undertaking. In past efforts involving learning the model [2, 9], the environment is typically a physical system that is sampled at some regular frequency, and the actions are control signals perturbing the state of the system. By contrast, in Keepaway, the domain we consider for our experiments, the actions are abstract, high-level skills, which last for extended, variable durations of time. Keepaway is a large-scale, complex, multiagent task involving both teammates and adversaries, which are part of the environment being modeled. The approach we follow is to partially learn the model for this task, and partially describe it using simple rules. This necessarily approximate model is then used in our Model-based Policy Improvement (MBPI) algorithm to examine if it can still help expedite learning.

The remainder of the paper is organized as follows. Section 2 describes the Keepaway task, and Section 3 presents our design of a model for this task. Section 4 provides details of the model-based RL algorithm. In Section 5 we present experimental results evaluating our method, providing comparisons with other algorithms that have been applied to Keepaway. Section 6 discusses related work, and Section 7 concludes.

2 Keepaway Task Description

Keepaway [11] is a subtask of simulated RoboCup soccer [8] played between a team of m *keepers* and a team of n *takers* inside a rectangular region. The objective of the keepers is to maintain possession of the ball (have it close enough to be kicked), while the takers try to steal it. The task is episodic – each episode starts with the ball in possession of one of the keepers, and ends when some taker gets the ball or it goes outside the region of play. The version of Keepaway we consider for our experiments involves 3 keepers and 2 takers ($3v2$) inside a $20m \times 20m$ region, as depicted in Figure 1. We proceed to describe how Keepaway is framed as a reinforcement learning problem, outlining the challenges it poses.

A complete state description in Keepaway would include the positions and velocities of the players and the ball, the players' body and neck angles, their stamina levels, and so on. However, we find that their positions alone convey most of the information required for the purpose of learning. Since the players and ball may occupy any position inside the region of play, the state space is continuous. Furthermore, the players are provided noisy sensations of state.

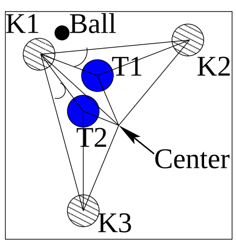

Figure 1. $3v2$ Keepaway. K_1, K_2, and K_3 are keepers. T_1 and T_2 are takers.

The keepers are the learning agents: the task each keeper has to learn is which action to take when it gets possession of the ball. This being the case, it becomes necessary to define the concept of a state only when some keeper has possession. In each state, the keeper closest to the ball is denoted K_1; the other keepers are denoted $K_i, i = 2, 3, 4, \ldots, m$, K_i being the i-th *closest* keeper to K_1. Similarly, the takers are denoted $T_i, i = 1, 2, 3, \ldots, n$, T_i being the i-th *closest* taker to K_1. K_1 is the keeper that must choose an action to execute. The behaviors of the takers and keepers who are without possession are fixed: the takers try to intercept the ball, while $K_2, \ldots, K_m$ attempt to move to positions to which a pass from K_1 is likely to succeed.

Figure 1 illustrates the indexing of keepers and takers, also marking out distances and angles among the players and the center of the field. These serve as *abstract* features derived from the players' positions, which are used as inputs to the function approximator representing the action value function. We refer the reader to Stone, Sutton and Kuhlmann [11] for a detailed description of these abstract state features. Notice that there are 13 for $3v2$ Keepaway.

The actions that are available to K_1, when it has possession of the ball, are **HoldBall**, by which it keeps the ball with itself, and **PassBall**(i), $i = 2, 3, 4, \ldots, m$, which is a direct pass to the K_i. While it is convenient to treat **HoldBall** and **PassBall**(i) as actions, they are really high-level skills or *options* [13] implemented through a series of low-level actions like **Turn** and **Kick**. Passes can last a variable number of simulator cycles; so the task is effectively a Semi-Markov Decision Process [4]. The transition dynamics of the extended high-level actions, which are necessarily stochastic because of the keepers' noisy actuators, thus become susceptible to even greater irregularity. Also, the dynamics are not smooth, as some actions can lead to terminal states.

The reward provided for taking an action from a state is simply the number of cycles elapsed until the next state is reached. Since the task is episodic, no discounting is required. Maximizing expected long-term reward corresponds to maximizing the expected overall duration of the episode, also called the *hold time*. **HoldBall()** typically lasts 1-2 cycles; **PassBall(i)** can last between 4 and 12 cycles, depending on the distance the pass has to travel. A cycle of simulation lasts 100 milliseconds in real time. In $3v2$ Keepaway, a random policy that chooses uniformly among the actions (**HoldBall**, **PassBall(2)**, **PassBall(3)**) registers a hold time of about 4.7 seconds.

In our experiments, we use the same version of $3v2$ Keepaway as used by Kalyanakrishnan and Stone [6], but with one minor change. In their version, K_1 executes **HoldBall** through a series of kicks close to its body that take it away from the direction of the takers. In our implementation, K_1 simply stops the ball once it is kick-able, and subsequently leaves it untouched. We find that this helps our model-based approach by simplifying the transition dynamics. Interestingly, informal testing reveals that it also leads to better performance with the model-free methods successfully applied earlier [6, 11]. We compare all these algorithms using our version of **HoldBall**.

3 Learning the Model

In this section, we describe our design of a model for Keepaway. The precise requirements of the model are that given state s and action a, it predict a distribution over next states s', as well as the reward r for the transition. Since the actions are disparate, high-level skills, we maintain separate models for each action. Figure 2 lays out the schematic design. Though Keepaway is indeed a stochastic domain, we adopt the simple approach of approximating its dynamics using a *deterministic* model, i.e., the model returns a unique next state s' instead of a distribution over next states. Since some transitions can lead to terminal states, we employ a separate predictor to compute t, a boolean value indicating whether a given transition is terminal. Likewise, a separate predictor computes the real-valued transition reward r.

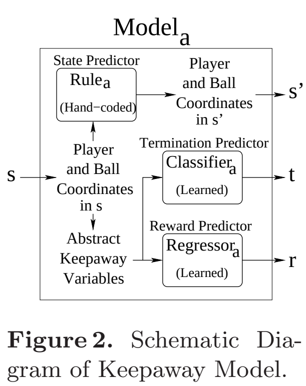

Figure 2. Schematic Diagram of Keepaway Model.

Our main objective is not building an accurate model in itself, but rather to evaluate the advantages of using a model in conjunction with the RL algorithm. We find it sufficient for this purpose to specify parts of the model using intuitive, hand-coded rules, but nonetheless, necessary to derive other parts of it by applying machine learning. As shown in Figure 2, the next state s' is computed by applying a simple rule to the current state s. The rule simply assumes the players do not change their positions between s and s'. In case of the **HoldBall** action, the ball's position in s' is predicted to be the same as K_1's, and if the action is **PassBall(i)**, the ball is predicted to occupy the same position as K_i. Figure 3 illustrates through an example from $3v2$ Keepaway how the next state prediction is made for a given state and action.

In our model, the termination and reward predictors are trained through supervised learning using the observed experiences. The reward predictor for each action is a single-layer neural net with 10 hidden nodes. Its inputs are the abstract state features derived from the Keepaway state (see Section 2), over which effective generalization is possible. The output is a real-valued prediction of the reward. The termination predictor is a single-layer neural net with 5 hidden nodes. It takes the same inputs as the reward predictor, but computes a boolean-valued output instead. In $3v2$ Keepaway, only roughly 10% of all transitions are terminal; nonetheless, we increase the weight of terminal transitions in the training distribution to present each termination predictor an equal number of terminal and non-terminal transitions.

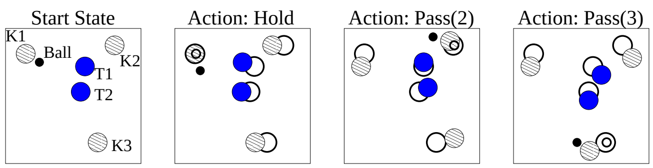

Figure 3. At left is a start state. The subsequent figures show true (shaded) and predicted (outlined) next states reached after each action is taken from the start state.

The operation of the model is summarized as follows. Given a set of training experiences $D = \{(s, a, r, s')\}$, we fit a model as $M = learnModel(D)$, the learning restricted to the termination and reward predictors for each of the actions. Subsequently, M can be used to make predictions; given state s and action a, the predictions are of the form $s' = M.predictNextState(s, a)$, $t = M.predictTermination(s, a)$, and $r = M.predictReward(s, a)$. In Section 4, we explain how the model is employed under the Model-based Policy Iteration (MBPI) algorithm.

Table 1 lists the prediction errors of the models for the three actions in $3v2$ Keepaway. The entries are averages from 5 independent runs – in each run a model is learned based on transitions from 50 episodes (an episode typically comprises 10-20 transitions) during which the keepers follow a random policy. This model is then tested for 1000 episodes, again following random action selection. For the purpose of computing prediction errors in the positions of players, we use the same ordering $(K_1, K_2, K_3, T_1, T_2)$ in s' as seen in s. Thus, if keeper K_A is K_1 in s, and has passed the ball to K_B, K_A is still considered K_1 in s' while computing the error. Of course, the real ordering of s' is used while computing abstract features for s'.

We carried out $3v2$ Keepaway inside a $20m \times 20m$ region; distances among players are typically 5-15m. Notice that for the **HoldBall** action, the prediction errors for all the players' positions are less than $1.0m$; this is because the action itself typically lasts only 1-2 cycles, during which the players do not move very far. The errors are much higher for the pass actions, and indeed higher for **PassBall(3)** than **PassBall(2)** because of the longer distance the pass has to travel. The reward predictions errors are quite small for **HoldBall**, and within about 2 cycles for the pass actions. For the **PassBall(i)** actions, the misclassification probabilities of terminal and non-terminal transitions are comparable. The high error in classifying terminal **Hold** actions arises because of insufficient training data: only a very small fraction of **HoldBall** actions terminate while following a random policy. We recognize that there is scope to improve the accuracy of the model; in particular, the accuracy of the state predictor (see Section 5). Nonetheless, the measure we seek to evaluate in this paper is not the accuracy of the model itself, but the performance achieved by the RL algorithm employing the model. The algorithm is described in the next section.

Action	Position						Terminal	Non-terminal	Reward
	K_1	K_2	K_3	T_1	T_2	Ball			
HoldBall	0.63	0.89	0.91	0.81	0.96	0.64	0.93	0.004	0.33
PassBall(2)	3.62	3.88	4.03	2.85	2.89	3.74	0.16	0.13	2.07
PassBall(3)	4.03	3.78	4.78	2.85	2.92	4.98	0.17	0.12	1.96

Table 1. Errors in the positions are root mean squared values of the distance (in meters) between true and predicted positions. Terminal and Non-terminal errors are the fractions of terminal and non-terminal actions misclassified. Reward errors are root mean squared values of the difference (number of cycles) between true and predicted values.

4 Using the Model

The central idea underlying our Model-based Policy Improvement (MBPI) algorithm is to use the gathered experiences to learn a model of the environment, and then use this model extensively to simulate transitions based on which the action value function is updated. The model and the learned policy are improved iteratively, as we describe in Algorithm 1.

We begin with some initial Q function (line 1). A policy is interpreted from Q through the *selectAction*() function (line 9), which can implement, for instance, ϵ-greedy action selection. A batch of experiences D is collected by following this policy for some fixed number e of episodes (lines 6-14). Once the experiences are collected, they are used to learn a model M of the environment (line 16). Q is now updated using transitions that are simulated using M (lines 18-36). This is accomplished by generating trajectories of depth *depth* using M, beginning with some random start state (line 19) and following an action selection policy specified by the function *selectActionSimulate*() (line 23). Once Q is updated, it is used to generate the next batch of experiences; a new model is learned and the process continues until Q converges. MBPI is similar to Lin's experience replay algorithm [7], applied to Keepaway by Kalyanakrishnan and Stone [6], which differs from it in the following manner: in experience replay, no explicit model is learned, and the policy improvement occurs through (depth 1) updates solely involving the experiences stored in D.

In all our experiments, we have fixed the values of parameters and choices for subroutines through informal experimentation. The function approximation scheme we use for representing Q is the same used by Stone *et al.* [11] and Kalyanakrishnan and Stone [6] – a separate CMAC [1] for each action, taking as input the 13 abstract state features computed from the state. Each CMAC employs 32 one-dimensional tilings along each feature, the tile widths being $3.0m$ for features corresponding to distances, and $10°$ for those corresponding to angles. The *selectAction*() function implements ϵ-greedy action selection, with $\epsilon = 0.01$. We fix the number of experiences in each batch, e, to 50. By setting all CMAC weights to zero in Q_0, the initial action value function, the policy followed for generating the first batch of experiences is random.

Algorithm 1 Model-based Policy Improvement

```
Q ← Q_0. //Initialize action value function.
D ← ∅. //Initialize memory of experiences.
//Improve Q iteratively.
repeat
  // Experience Generation
  for e episodes do
    s ← startStateFromEnvironment().
    repeat
      a ← selectAction(Q).
      r ← rewardFromEnvironment().
      s' ← nextStateFromEnvironment().
      D ← D ∪ (s, a, r, s').
    until s' is terminal.
  end for
  // Model Learning
  M ← learnModel(D).
  // Policy Improvement
  for n iterations do
    s ← randomStartStateFromSimulator().
    d ← 0.
    //Simulate trajectories of depth depth.
    repeat
      a ← selectActionSimulate(Q, s).
      r ← M.predictReward(s, a).
      t ← M.predictTermination(s, a).
      // Update Q based on simulated transitions.
      if t then
        Q(s, a) ← Q(s, a) + α(r − Q(s, a)).
      else
        s' ← M.predictNextState(s, a).
        Q(s, a) ← Q(s, a) + α(r + γ max_a' Q(s', a') − Q(s, a)).
        s ← s'.
      end if
      d ← d + 1.
    until t = true or d = depth.
  end for
until Q has converged.
```

The set of experiences D used in every iteration to learn the model comprises *all* the past experiences collected thus far; we find that this yields better performance than obtained by only keeping the most recent batch (or some recent window) of experiences in D. The termination and reward neural networks for each action are trained using supervised learning. For the termination predictors, $200,000$ backprop updates are made with a learning rate of 0.0001, picking terminal and non-terminal transitions in D with equal likelihood. $20,000$ backprop updates using randomly chosen experiences from D are made in the case of the reward predictors, with a learning rate of 0.0005. While making learning updates to the function approximator representing Q using experiences simulated by the learned model M, we fix the number of (Q-learning) *updates* to $30,000$, each made with a learning rate of 0.025. We find that doing so offers more stability than fixing the number of iterations n, under which the actual number of updates would depend on the size of D. The start states of the trajectories are randomly chosen start states from the transitions in D, and $selectActionSimulate()$ implements random action selection.

5 Experimental Results and Discussion

In this section, we present the results of our experiments on $3v2$ Keepaway. Figure 4(a) shows the performance of our model-based policy iteration algorithm, using $depth = 1$ while simulating trajectories (MBPI-1). It is compared with experience replay (ER), which achieves the best asymptotic performance on $3v2$ Keepaway among the batch methods considered by Kalyanakrishnan and Stone [6], and simple on-line learning (OL) [11], where a single Q-learning update is made after every transition. We find ER to achieve its best performance by making $30,000$ Q-learning updates during the policy improvement phase, with learning rate 0.025. Interestingly, the same values were found the best for MBPI-1. For OL, we used a learning rate of 0.125, the same used by Stone *et al.* [11] in their Sarsa-based OL implementation. Figure 4(a) shows that MBPI outperforms ER and OL right from the beginning, and also betters their asymptotic performance (Figure 4(b) shows OL continuing until $20,000$ episodes). At 200 episodes, MBPI-1 registers a higher hold time than ER and OL with p-value at most $p < 5 \times 10^{-9}$, under a single-tailed t-test. The best performance achieved by MBPI-1 (11.62 seconds, 450 episodes) exceeds those of ER (9.24 seconds, 250 episodes) and OL (9.48 seconds, $11,000$ episodes) with p-value at most $p < 10^{-13}$.

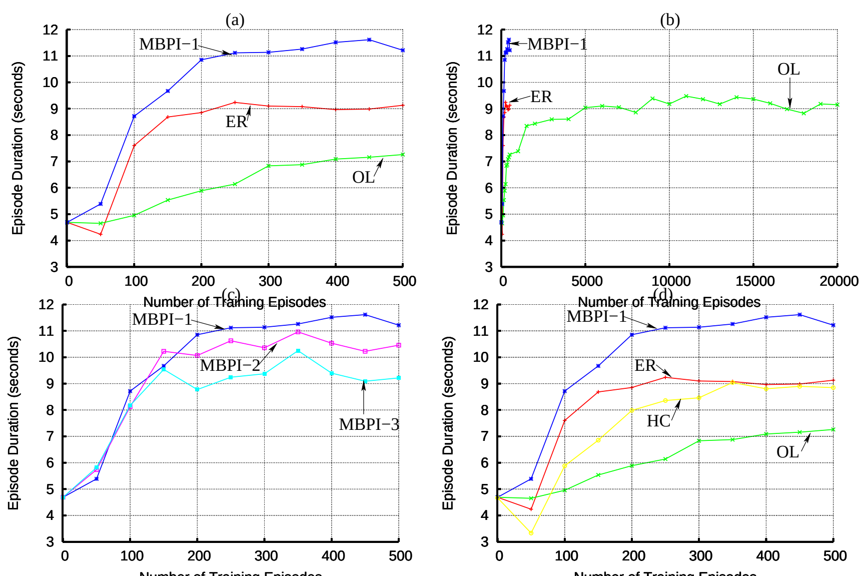

Figure 4. The graphs show on the x axis the number of training episodes; on the y axis the hold time achieved by some policy. The reported hold time is the average over 200 episodes in which the policy is frozen and executed. For the first 500 episodes of training, we evaluate the policies at intervals of 50 episodes. The algorithms making batch updates do so every 50 episodes; they are evaluated immediately *after* the update. Each curve is an average of at least 25 independent runs.

The main reason MBPI-1 and ER achieve an order of magnitude gain in sample complexity over OL is that they make more efficient use of the collected experiences through batch updates. The updates made by ER are strictly based on observed experiences, which reflect the true dynamics of the environment. Further, only states that are reachable by following the policy used while generating the experiences get backed up. In contrast, MBPI explores more parts of the state space by following trajectories randomly generated using the model. Experiences generated by the (approximate) model are likely to be somewhat inaccurate, and the states visited along the simulated trajectories may not be reached in practice. But despite the inaccuracy, the exploration can potentially result in the discovery of desirable states and thus improve the policy.

Lin [7] compares ER with *relaxation planning*, a model-based approach in a discrete, grid-world domain. In case the agents have to learn the model, then the model-based approach performs *worse* than ER; however, when the agents are provided a perfect model to begin with, the algorithms have comparable performances. In our experiments with $3v2$ Keepaway, MBPI-1, under which the model both has to be learned and used, consistently outperforms ER. We conjecture that for domains with small state spaces, the observed transitions may themselves be sufficiently representative of the dynamics of the domain, but as the size of the problem increases, this may cease to be the case, and the extrapolation afforded by the model may prove beneficial. In this paper, it is our intention to compare MBPI and ER on Keepaway by studying them in isolation, but in principle, it is possible to combine them by making policy updates from both observed and simulated transitions. More specifically, it may be possible to offset the noise introduced by an incorrect model by making sufficient updates based on true experiences. Section 6 discusses Sutton and Barto's Dyna-Q algorithm [12], which takes a related approach.

Figure 4(c) shows the effect of increasing the depth of the simulated trajectories in our model-based algorithm. It seems plausible that deeper trajectories will enhance the exploration of the state space, boosting performance. On the other hand, due to the noise in the model predictions, simulated transitions are likely to deviate more from the true transitions deeper in the trajectory. In our experiments, we notice that with increasing depth (MBPI-2 and MBPI-3), the performance of the model-based approach *degrades* progressively. To diminish the adverse effect of noise deep in the trajectories, we decay the learning rates for updates made deeper down, still keeping the sum of the learning rates along each trajectory constant (at 0.025) so that the comparisons among the experiments remain fair. Despite using a sharp decay factor (0.01), the performance of MBPI-2 and MBPI-3, as seen in Figure 4(c), fall significantly short of MBPI-1's. Nonetheless, MBPI-2 (10.96 seconds, 350 episodes) still achieves higher performance than ER (9.24 seconds, 250 episodes) and OL (9.48 seconds, $11,000$ episodes), with p-values at most $p < 2 \times 10^{-3}$.

Surely, a major reason for the loss in performance when exploring deeper is the approximation in our model. Since the dynamics of Keepaway are stochastic, a deterministic model is bound to be inaccurate. Further, the function approximation scheme used in the model may not be sufficiently expressive. Past efforts in modeling physical systems have focused on learning precise models, and indeed modeling environmental noise as well [9]. It is a promising avenue for future research to develop a more accurate model for Keepaway, and examine if it can be used to plan deeper into the future. Nevertheless, the performance gain offered by MBPI-1 is evidence that model-based approaches can be viable even with an approximate model, on a task that is itself continuous and stochastic.

Algorithm 2 Hand-coded Policy (Model M, State s)

```
A_non-terminal ← {a | M.predictTermination(s, a) = false}.
if A_non-terminal = ∅ then
   Return random(HoldBall, PassBall(2), PassBall(3)).
else
   if HoldBall ∈ A_non-terminal then
      Return HoldBall.
   else if PassBall(2) ∈ A_non-terminal then
      Return PassBall(2).
   else
      Return PassBall(3).
   end if
end if
```

In our MBPI algorithm, the learned model is used to update the Q function through which the action selection policy is interpreted. While this conforms with the traditional RL approach of learning the Q function, it is not necessary for putting the model to use. Algorithm 2 lays out a hand-coded policy that uses an available model to select the action to take. In fact, this policy only makes use of the termination predictor of the model, implementing the following intuitive strategy: from any state, choose **HoldBall** if the model predicts it will not terminate the episode. If it is predicted to terminate, try **PassBall(2)** in a similar manner, and then **PassBall(3)**. If all actions are predicted to terminate, simply choose a random one. Note that the hand-coded policy is myopic: it doesn't perform lookahead to take actions that will avoid future bad states. In this way it is handicapped when compared to the learning algorithms.

Figure 4(d) plots the performance of this hand-coded policy. As with MBPI, it begins with a random model that is updated every 50 episodes; however, the policy followed in between the updates is the hand-coded policy. After 350 episodes, this policy registers 9.04 seconds of hold time, which is within 0.5 seconds of the best reached by OL and ER. The purpose of this experiment is not to highlight the performance of the hand-coded policy in itself, but to illustrate that a model can be useful even independent of the action value function. Here, it is necessary to improve the model iteratively, but one can imagine scenarios where a model is available from past experiences or adapted from related tasks. A model-based approach provides the flexibility to re-use parts of the solution in a natural way. It would be promising as part of future research to adapt the model-based approach followed here to interact with similar tasks in the RoboCup soccer domain, for instance, $4v3$ Keepaway [11] and Half Field Offense [5]. Another possible avenue for research is to employ the environmental model as part of a planning algorithm for solving the task.

6 Related Work

In their expository textbook, Sutton and Barto [12] investigate the relationship between model-based RL and planning. They present the Dyna-Q algorithm, in which an environmental model is learned and used to simulate experiences for updating the Q function along with direct updates based on real experiences. Dyna-Q is enhanced by using Prioritized Sweeping, a technique whereby the model-based updates are concentrated around the regions where the Q function is changing rapidly. The main motivation for our work is indeed to extend the qualitative results of model-based approaches like those seen in the simple, relatively small, discrete domains considered by Sutton and Barto to a realistic, high-dimensional, continuous task. Our MBPI algorithm is similar to their Trajectory Sampling method, where model-based updates are based on the *on-policy* distribution of experiences. In our case, an ϵ-greedy policy is used while interacting with the environment, but a random policy is used to generate trajectories for the model-based updates. The complexity of Keepaway and the real-time constraints of the RoboCup simulator force us to make model-based updates *off-line*, whereas it is possible to make such updates on-line in the example domains used to illustrate Dyna-Q and Prioritized Sweeping.

In several past efforts of model-based approaches, learning the model is itself the key issue. Ng *et al.*[9] successfully learn a model for helicopter control. The state space is described by 8 body coordinates, and 4 continuous actions serve as control signals to maneuver the helicopter every 50^{th} of a second. A stochastic model is learned using locally weighted regression; an action policy is derived from this model using the PEGASUS algorithm. $3v2$ Keepaway has a state space of higher dimension (13), with states being more temporally distant. Also, actions are abstract, high-level skills, unlike control signals to the helicopter that perturb its state smoothly. Additionally, in Keepaway, it is actually necessary to iteratively gather experiences based on updated versions of the policy (about 5-6 times using MBPI-1) in order to achieve high performance. The helicopter control policy, on the other hand, is learned based on a single batch of experiences obtained by a human pilot flying the helicopter.

Other approaches involving modeling physical systems include, among others, those of Atkeson and Santamaría [2], and Boone [3]. The former investigate a pendulum swing-up problem with 2 state variables and 1 continuous action; the latter considers the Acrobot problem, having 4 state variables and 3 discrete actions. Apart from having fewer state variables and actions, these physical world tasks have smoother transition dynamics than Keepaway: a key component of our Keepaway model is the termination predictor, which is not required for the pendulum and acrobot tasks. Nonetheless, the main results from these tasks concur with ours – that model-based RL can greatly reduce sample complexity, while improving the quality of the learned solution.

Experience Replay is a model-free batch learning method due to Lin [7], which has been applied to Keepaway by Kalyanakrishnan and Stone [6]. The results in this paper show that our model-based approach registers faster learning and better asymptotic performance than experience replay on Keepaway. We compare and contrast the approaches in Section 5.

7 Conclusion

We examine the viability of using model-based RL for Keepaway, a complex, stochastic, continuous, high-dimensional, multiagent task. The actions in this task are abstract, high-level skills that can last variable periods of time, making it novel from a model-learning perspective. Our model is partially specified through simple rules, partially learned, and then used as a subroutine in the RL algorithm to learn the action selection policy. Empirical results demonstrate that such a model-based RL algorithm can yield significant gains in sample complexity and asymptotic performance when compared to model-free approaches that have been applied to Keepaway successfully in the past. Also, we show that a model can be used effectively with other static policies, lending flexibility to the learned solution. Problems for future work include improving upon our design of the Keepaway model, using it for knowledge transfer among related tasks, and applying it with planning-based approaches.

Acknowledgements

This research was supported in part by NSF CISE Research Infrastructure Grant EIA-0303609, NSF CAREER award IIS-0237699, and DARPA grant HR0011-04-1-0035.

References

1. J. S. Albus. *Brains, Behavior, and Robotics.* BYTE Books, Peterborough, 1981.
2. C. Atkeson and J. Santamaría. A comparison of direct and model-based reinforcement learning. *IEEE International Conference on Robotics and Automation*, 4:3557–3564, April 1997.
3. G. Boone. Efficient reinforcement learning: model-based acrobot control. *IEEE International Conference on Robotics and Automation*, 1:229–234, April 1997.
4. S. J. Bradtke and M. O. Duff. Reinforcement learning methods for continuous-time Markov decision problems. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems*, volume 7, pages 393–400. The MIT Press, 1995.
5. S. Kalyanakrishnan, Y. Liu, and P. Stone. Half field offense in RoboCup soccer: A multiagent reinforcement learning case study. *Proceedings of the RoboCup International Symposium 2006*, June 2006.
6. S. Kalyanakrishnan and P. Stone. Batch reinforcement learning in a complex domain. In *The Sixth International Joint Conference on Autonomous Agents and Multiagent Systems*, May 2007.
7. L.-J. Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8:293–321, 1992.
8. M.Chen, E.Foroughi, F.Heintz, Z.Huang, S.Kapetanakis, K.Kostiadis, J.Kummeneje, I.Noda, O.Obst, P.Riley, T.Steffens, Y.Wang, and X.Yin. Users manual: RoboCup soccer server — for soccer server version 7.07 and later. *The RoboCup Federation*, August 2002.
9. A. Y. Ng, H. J. Kim, M. I. Jordan, and S. Sastry. Autonomous helicopter flight via reinforcement learning. In S. Thrun, L. Saul, and B. Schölkopf, editors, *Advances in Neural Information Processing Systems 16.* MIT Press, Cambridge, MA, 2004.
10. M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming.* John Wiley and Sons, New York, NY, 1994.
11. P. Stone, R. S. Sutton, and G. Kuhlmann. Reinforcement learning for RoboCup-soccer keepaway. *Adaptive Behavior*, 13(3):165–188, 2005.
12. R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, 1998.
13. R. S. Sutton, D. Precup, and S. P. Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1-2):181–211, 1999.
14. G. Tesauro. Practical issues in temporal difference learning. In J. E. Moody, S. J. Hanson, and R. P. Lippmann, editors, *Advances in Neural Information Processing Systems*, volume 4, pages 259–266. Morgan Kaufmann Publishers, Inc., 1992.
15. J. N. Tsitsiklis and B. V. Roy. Feature-based methods for large scale dynamic programming. *Machine Learning*, 22(1-3):59–94, 1996.
16. C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3-4):279–292, 1992.