Using Advice to Transfer Knowledge Acquired in One Reinforcement Learning Task to Another

Lisa Torrey[1], Trevor Walker[1], Jude Shavlik[1], and Richard Maclin[2]

[1] University of Wisconsin, Madison, WI 53706, USA
{ltorrey, twalker, shavlik}@cs.wisc.edu
[2] University of Minnesota, Duluth, MN 55812, USA
rmaclin@d.umn.edu

Abstract. We present a method for transferring knowledge learned in one task to a related task. Our problem solvers employ reinforcement learning to acquire a model for one task. We then transform that learned model into advice for a new task. A human teacher provides a mapping from the old task to the new task to guide this knowledge transfer. Advice is incorporated into our problem solver using a knowledge-based support vector regression method that we previously developed. This advice-taking approach allows the problem solver to refine or even discard the transferred knowledge based on its subsequent experiences. We empirically demonstrate the effectiveness of our approach with two games from the RoboCup soccer simulator: KeepAway and BreakAway. Our results demonstrate that a problem solver learning to play BreakAway using advice extracted from KeepAway outperforms a problem solver learning without the benefit of such advice.

1 Introduction

We propose a novel method to transfer the knowledge gained in one reinforcement learning (RL) task to a related task. Complex RL domains, such as RoboCup soccer [11], can often be divided into several related learnable tasks. *Transfer* is the process of using the knowledge acquired in one task to improve the learning of a related task. For example, the skill of keeping a soccer ball away from opponents can be used to evade players who are defending a goal, making it easier to learn the task of goal scoring.

In this work, we present a method for performing transfer using *advice*. Advice taking is a way to incorporate user guidance into RL and can significantly improve performance in complex domains [6, 7, 9]. In our previous work with advice [9], the user observes the learner performing a task and then provides advice about which actions to prefer in certain situations. In contrast, our method for transfer obtains action preferences automatically from a model learned on a previous task. The user provides a *mapping* that connects the two tasks, allowing this *transfer advice* to be applied in the new task.

We have several reasons for using advice to accomplish this knowledge transfer. It supplies the learner with some prior knowledge of the relative merits of

J. Gama et al. (Eds.): ECML 2005, LNAI 3720, pp. 412–424, 2005.

actions in new situations where old skills might apply. It allows a user who is not familiar with the learning algorithm to guide the knowledge transfer simply by specifying the similarities between two tasks. Also, since advice can be refined or discarded by the learner if it is contradicted by experience, transfer advice should not be harmful in the long run even if the user's guidance is imperfect.

In the RoboCup simulated soccer domain [11], we extract transfer advice from a task called *KeepAway* and apply it to another task called *BreakAway*. The KeepAway game was originally introduced by Stone and Sutton [16]. The game we call BreakAway [9] is the subtask of shooting goals. Both of these games are set on a field with two teams of players, and contain actions for controlling the soccer ball. However, the two games use different field layouts and have different objectives. We provide mapping advice that points out the similarities that do exist, and our algorithm produces transfer advice that captures a reinforcement learner's knowledge about KeepAway. We then give this advice to a new problem solver to allow it to use KeepAway skills wherever they also apply in BreakAway.

The next section describes the RoboCup domain from the RL perspective. The following section gives more detail on our RL implementation and the way that it incorporates advice. After that, we introduce our transfer advice algorithm and demonstrate its potential with some initial results.

2 Reinforcement Learning in RoboCup

Reinforcement learning [17] is a continual learning process in which an agent navigates through an environment trying to earn *rewards*. The environment's state is usually represented by a set of *features*, and the agent executes *actions* that cause the state to change. Typically an agent learns a Q-function, which estimates the best long-term sum of rewards it could receive starting with a specific action from the current state. The agent's *policy*, or procedure for choosing actions, is usually to take the action with the highest Q-value in the current state. After taking the action and receiving some reward, the agent updates its estimate of that Q-value to improve its Q-function.

In the learning task of M-on-N KeepAway (see Figure 1), the objective of the M reinforcement learners called *keepers* is to keep the ball away from N (usually $M-1$) hand-coded players called *takers*. The game ends when an opponent takes the ball or when the ball goes out of bounds. A keeper needs to make an action choice only when it has possession of the ball; it may choose to hold the ball, pass to its closest teammate, or pass to its furthest teammate. It does not have movement options.

Our KeepAway state representation is the one designed by Stone and Sutton [16], which consists of features that capture simple geometric properties of the learner's field perspective, such as distances to other players and angles formed by trios of players. They describe relative distances from the learner rather than absolute locations. In our implementation, the three learners share a single Q-function learned from the combination of their experiences, and they all receive a +1 reward for each time step their team keeps the ball.

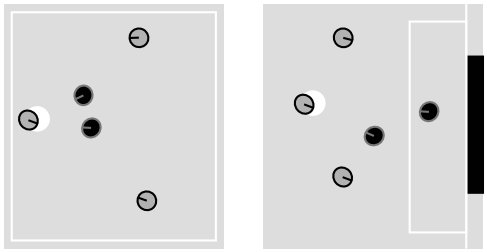

Fig. 1. Samples of 3-on-2 KeepAway and BreakAway games. The ball is the white circle, and is held by a keeper on the left (a light circle with a dark border) and an attacker on the right. The two opponents (dark circles with light borders) are both takers on the left, but on the right, one is a defender and the other a goalie.

In M-on-N BreakAway (see Figure 1), the objective of the M reinforcement learners called *attackers* is to score a goal against $N-1$ hand-coded *defenders* and a hand-coded *goalie*. The game ends when they succeed, when an opponent takes the ball, when the ball goes out of bounds, or after a time limit of 10 seconds. When an attacker has possession of the ball, it has a learnable action choice: to move with the ball, to pass the ball to its closest or furthest teammate, or to shoot the ball at the goal. We limit movement to four choices: forward towards the center of the goal, away from the goal, and clockwise or counterclockwise along a circle centered at the goal. The shoot action directs the ball at the center, right side, or left side of the goal, whichever is least blocked by the goalie.

Our BreakAway state representation also consists of features measuring important distances and angles, many of which are similar to KeepAway features. The new features include distances and angles involving the goal, and the time left in the game. The learners share a single model, and they all receive a +2 reward for a goal, 0 for a failed shot, and -1 for the other game endings.

In our RL method, the learners approximate the Q-function by solving a linear optimization problem. Following Stone and Sutton's approach [16], we use *tile coding* to include some non-linear features in this problem, which allows the model to express more complex functions. Tile coding discretizes each numeric feature into several overlapping tilings, each containing a set of discrete tiles. Each tile is represented by a Boolean feature that is true when the numeric value falls into the tile interval and false otherwise. Through this process, we add 64 Boolean features to the state space for every numeric feature. We have found, as did Stone and Sutton, that this addition to the state space significantly improves learning for RoboCup.

Neither of these games is trivial, especially since the soccer simulator incorporates noise into players' sensors and actions. BreakAway is the more difficult of the two, because it contains only one positive reward that is rarely received by chance (the goalie can easily block random shots). Learners in BreakAway also have more actions to choose from and a larger state space to navigate.

3 Background: Support Vector Regression

Our learners employ a type of RL called SARSA with a one-step look-ahead to estimate Q-values [17]. Our implementation uses support vector regression (SVR). We use a linear optimization method proposed by Mangasarian et al. [10] and extended in Maclin et al. [8, 9] to compute a model that approximates the Q-function. We train the learner in batches: it uses the most recent model to play 100 games, and then updates the model using these new training examples to get a better Q-function approximation.

The main structure in a learned model is a weight vector w, which has one weight for each feature in the feature vector x. Each action has its own weight vector and offset term b, and the expected Q-value of taking that action from the state described by x is $wx + b$. Our learners take the action that scores the highest with probability $(1 - \epsilon)$, and take a sub-optimal exploratory action with probability ϵ, where ϵ typically is a small number between 0.01 and 0.05.

To compute the weight vector for an action, we find the subset of training examples in which that action was taken and place those feature vectors into rows of a data matrix A. Using the previous model and the actual rewards received during those training steps, we compute new Q-value estimates and place them into an output vector y. The optimal weight vector is then described by

$$Aw + be = y \tag{1}$$

where e denotes a vector of ones (we omit this for simplicity from now on).

In practice, we prefer to have non-zero weights for only a few important features in order to keep the model simple and avoid overfitting the training examples. We therefore introduce *slack* variables s that allow inaccuracies on each example, and a penalty parameter C for trading off these inaccuracies with the complexity of the solution. The resulting minimization problem is

$$\begin{array}{ll} \min\limits_{(w,b,s)} & ||w||_1 + \nu|b| + C||s||_1 \\ s.t. & -s \leq Aw + b - y \leq s. \end{array} \tag{2}$$

where $|\cdot|$ denotes an absolute value, $||\cdot||_1$ denotes a sum of absolute values, and ν is a penalty on the offset term. By solving this problem, we can produce a weight vector w for each action that compromises between accuracy and simplicity.

In Mangasarian et al.'s [10] Knowledge Based Kernel Regression (KBKR) method, advice can be given in the form of a rule about a single action. This rule creates new constraints on the problem solution, in addition to the constraints from the training data. Recently, we introduced an extension to KBKR called Preference-KBKR [9], which allows advice about pairs of actions in the form

$$Bx \leq d \Longrightarrow Q_p(x) - Q_n(x) \geq \beta, \tag{3}$$

which can be read as:

> If the current state satisfies $(Bx \leq d)$, the Q-value of the preferred action p should exceed that of the non-preferred action n by at least β.

For example, consider giving the advice that shooting is better than moving ahead when the distance to the goal is at most 10. The vector B would have one row with a 1 in the column for the "distance to goal" feature and zeros elsewhere. The vector d would contain only the value 10, and β could be set to any small positive number.

Just as we allowed some inaccuracy on the training examples, we allow advice to be followed only partially. To do so, we introduce slack variables z and ζ and penalty parameters μ_1 and μ_2 for trading off the impact of the advice on the solution with the impact of the training examples.

The new minimization problem addresses all the actions together so that it can apply constraints to their relative values. Multiple pieces of preference advice can be incorporated, each with its own B, d, p, n, and β. We use the CPLEX commercial software program to solve the resulting linear program:

$$\begin{aligned}
&\min_{(w_a, b_a, s_a, z_i, \zeta_i \geq 0, u_i \geq 0)} \\
&\sum_{a=1}^{m} (||w_a||_1 + \nu |b_a| + C||s_a||_1) + \sum_{i=1}^{k} (\mu_1 ||z_i||_1 + \mu_2 \zeta_i) \\
&\quad \text{s.t.} \quad \text{for each action } a \in \{1, \ldots, m\} : \\
&\qquad\qquad -s_a \leq A_a w_a + b_a - y_a \leq s_a \\
&\qquad\quad \text{for each piece of advice } i \in \{1, \ldots, k\} : \\
&\qquad\qquad -z_i \leq w_p - w_n + B_i^T u_i \leq z_i \\
&\qquad\qquad -d^T u_i + \zeta_i \geq \beta_i - b_p + b_n.
\end{aligned} \tag{4}$$

4 Transfer Advice

In order to transfer knowledge gained on one task to a related task, we automatically extract advice that tells the learner to prefer some actions over others in new situations based on its experience in the old task. This *transfer advice* is incorporated into the learning procedure for the new task as explained in the previous section. Figure 2 summarizes the overall process.

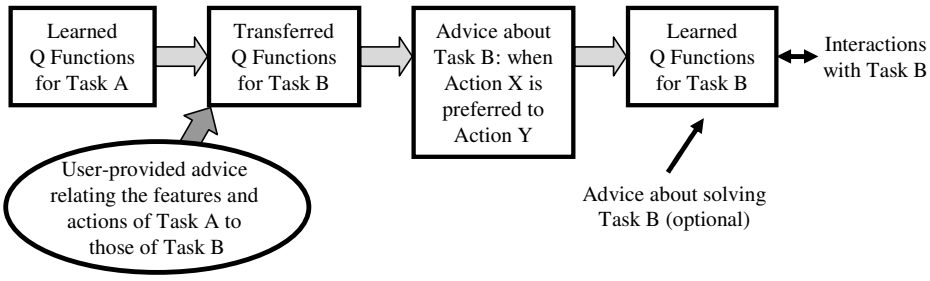

Fig. 2. Transferring knowledge using advice. Advice may also have been given when learning Task A.

As an example, suppose we learned in KeepAway that when a taker was near, passing to the nearest teammate was better than any other action. Our algorithm might transfer this knowledge to BreakAway by generating advice that when a defender is near, passing to the nearest teammate is better than any other action.

This kind of advice is not the only way, or the most obvious way, to transfer knowledge. One alternative would be to translate the actions and features of the old task into actions and features of the new task, and then apply the old Q-function directly to the new task, hoping that it would provide a good starting point for learning. However, if the new task has a different reward structure, these estimates might be uninformative. Simply transferring the Q-function from KeepAway would give a BreakAway learner inaccurate initial estimates.

Instead of transferring Q-values, our method transfers a partial policy that covers some regions of the feature space. By telling the learner to *prefer* some actions over others in those regions, we give *relative* constraints on Q-values instead of specifying them absolutely. This approach is more robust to differences in the tasks' reward structures.

The only input our method requires from a human teacher is a *mapping* that translates features and actions in the old task to features and actions in the new task. For example, we might map KeepAway features involving the nearest taker to BreakAway features involving the nearest defender, and the KeepAway action $HoldBall$ to the BreakAway action $MoveAhead$.

Using this mapping, our algorithm evaluates a state from the perspective of the old task. If one old action would have been better than all the others in this situation, it gives transfer advice that recommends taking the corresponding action in the new task. Table 1 gives the general form of the transfer advice algorithm, and Table 2 gives a simple but concrete example. Note that this direct translation of a learned model into advice is possible because we represent both models and advice as linear expressions of features.

There are a few complications to this basic procedure. For example, sometimes an old feature has no logical analogue in the new task. In these cases, the user may map the old feature f to a constant value instead of a new feature f'. For example, the takers in 3-on-2 KeepAway do not have corresponding defenders in 2-on-1 BreakAway, since the goalie behaves differently from a defender. The user could set the features describing distances to takers to their maximum

Table 1. The basic algorithm to create transfer advice. We set the constant Δ to 1 in our experiments. See Table 2 for more details.

GIVEN
A learned model of Task A AND
A mapping from Task A to Task B
DO
for each $a \in Actions(TaskA)$ generate advice:
IF for each $b \in Actions(TaskA), b \neq a: \quad Q'_a - Q'_b \geq \Delta$
THEN PREFER a' TO ALL b' in Task B

Table 2. A simple demonstration of extracting transfer advice. The actions in the old task are a, b, and c, and the corresponding actions in the new task are a', b', and c'. The learned model for the old task is a set of linear Q-value expressions with weights w and features f, and these are translated into advice that uses the corresponding new task features f'.

OLD TASK MODEL:	ADVICE FORMAT:
$Q_a = w_{a1} * f_1 + w_{a2} * f_2 + w_{a0}$	IF $Q'_a - Q'_b \geq \Delta$
$Q_b = w_{b1} * f_1 + w_{b0}$	AND $Q'_a - Q'_c \geq \Delta$
$Q_c = w_{c2} * f_2 + w_{c0}$	THEN prefer a' to b' and c'
USER-PROVIDED MAPPING:	FULL ADVICE EXPRESSION:
$(a, b, c) \longrightarrow (a', b', c')$	IF $(w_{a1} - w_{b1}) * f'_1 + w_{a2} * f'_2 + w_{a0} - w_{b0} \geq \Delta$
$(f_1, f_2) \longrightarrow (f'_1, f'_2)$	AND $w_{a1} * f'_1 + (w_{a2} - w_{c2}) * f'_2 + w_{a0} - w_{c0} \geq \Delta$
TRANSLATED EXPRESSIONS:	THEN prefer a' to b' and c'
$Q'_a = w_{a1} * f'_1 + w_{a2} * f'_2 + w_{a0}$	
$Q'_b = w_{b1} * f'_1 + w_{b0}$	
$Q'_c = w_{c2} * f'_2 + w_{c0}$	

value, implying that the nonexistent defenders are too far away to affect the learner's actions. Another example is the feature describing a player's distance to the center of the KeepAway field. The user could set that feature to the average value of its range. We use these constant mappings in the experiments reported later. To handle a feature in the new task that has no logical analogue in the old task, we simply leave the new feature out of the mapping.

We create one advice rule for each old action that has an analogue, indicating when the analogue looks like the best choice based on experience in the old task. The other new actions (such as $MoveLeft$ in BreakAway) must be learned independently by the agent. To handle old actions that have no analogues, we also simply leave the old action out of the mapping.

Since we added tile features from the tile encoding of numeric features, we also need to map the tiles of each KeepAway feature to tiles of a BreakAway feature. We automatically map tiles to maximize the amount of the BreakAway feature range that they share. This method does not require mapped features to have identical ranges, since that would severely restrict the mapping.

Remapping is a further capability that allows the learner to apply old knowledge in multiple ways. For example, an attacker can clearly use KeepAway skills to evade defenders on the BreakAway field; with a little cleverness, it might be able to use those same skills to shoot. Suppose the learner imagines a teammate is standing inside the goal. Then, a decision on whether to pass to that teammate corresponds to a decision on whether to shoot. We could do this by mapping all the features involving the teammate to features involving the goal. However, if this were the *only* mapping, it would prevent the learner from considering actually passing to the real teammate. To let the learner consider both actions, we create two sets of advice, using first one mapping and then the other. We included such a remapping in our experiments.

Situation-dependent mappings allow old knowledge to be used differently in different areas of the new feature space. For example, we might only want to map a KeepAway action to the BreakAway action *shoot* when the learner is close to the goal, because we know that soccer players should only shoot over short distances. We could do this by providing one mapping that applies when the learner is near the goal and a different one that applies when the learner is far from the goal. We did not use this situation-dependent mapping in our experiments; we expected that the KeepAway passing skills would already prevent attackers from shooting from too far away.

5 RoboCup Transfer Advice

The two RoboCup tasks we explore have significant differences that make transfer a non-trivial problem. In KeepAway, learners should make the game last as long as possible, but in BreakAway, they should end the game quickly by scoring a goal. Learners in KeepAway cannot choose to move, but learners in BreakAway can. KeepAway takers will always move towards the ball, but the BreakAway goalie will not. There are also different numbers of players.

However, there are also some useful similarities between the tasks. Many of the features map directly, and some of the actions are identical. On a conceptual level, the BreakAway attackers must play KeepAway while trying to score.

We designed the default mappings shown in Table 3 to take advantage of these similarities for transfer from 3-on-2 KeepAway to 2-on-1 BreakAway. The BreakAway features and actions that have no KeepAway analogues do not appear in these tables. The KeepAway features that have no BreakAway analogues are mapped to constants within their ranges.

We then used the remapping capability to apply KeepAway skills to shooting. The two remappings in Table 4 advise the learner to imagine that its nearest teammate is standing first in the left side of the goal, and then in the right side. If a pass to that teammate would have been the best action, the advice will recommend shooting. There was no BreakAway feature to describe the distance from the goalie to a goal section, so we used a constant value in its place.

Using a high-performing 3-on-2 KeepAway model that was trained without any advice [8], we applied these mappings and our transfer advice algorithm to create advice for 2-on-1 BreakAway. The process produced five advice items, each capturing one of the five KeepAway "skills": the three original actions and the two remapped pass actions. For example, this is the form of the advice that shows how to apply the KeepAway "hold ball" skill to BreakAway:

$$
\begin{aligned}
\text{IF} \quad & Q'_{HoldBall} - Q'_{PassFar} \geq \Delta \text{ AND} \\
& Q'_{HoldBall} - Q'_{PassNear\_remap1} \geq \Delta \text{ AND} \\
& Q'_{HoldBall} - Q'_{PassNear} \geq \Delta \text{ AND} \\
& Q'_{HoldBall} - Q'_{PassNear\_remap2} \geq \Delta \\
\text{THEN PREFER } & MoveAhead \text{ TO } PassNear \text{ AND } Shoot
\end{aligned}
$$

Table 3. Default action and feature mappings from KeepAway to BreakAway. *Near* and *Far* refer to teammates' distances from the learner. The learner is L, the keepers are K's, the takers are T's, and the attackers are A's.

KeepAway	*BreakAway*
PassNear	PassNear
HoldBall	MoveAhead
dist(L, near K)	distance(L, near A)
other player distances	MAX_RANGE
min angle(near K, L, any T)	MID_RANGE
min angle(near K, L, any T)	MID_RANGE
distances to field center	MID_RANGE

Table 4. Action and feature remappings that apply KeepAway skills to shooting

Remapping 1: *PassNear* to *Shoot*	
KeepAway	*BreakAway*
dist(L, near K)	dist(L, goal left)
min dist(near K, any T)	MID_RANGE
min angle(near K, L, any T)	angle(goal left, L, goalie)
Remapping 2: *PassNear* to *Shoot*	
KeepAway	*BreakAway*
dist(L, near K)	dist(L, goal right)
min dist(near K, any T)	MID_RANGE
min angle(near K, L, any T)	angle(goal right, L, goalie)

Essentially, this advice says that if holding the ball would be safer than passing towards a teammate or towards the goal, then moving forward with the ball is probably safer than passing or shooting.

6 Empirical Results

The linear program presented in Equation 4 contains parameters ν, C, μ_1, and μ_2. The first two we set by tuning on KeepAway games without using any advice. This led to C being set to $2500 * (0.1 + 0.9(1 - e^{\#GamesPlayed/10{,}000}))/\#features$ and ν to 100. We exponentially increase C because as RL progresses the estimates of Q (which are infinite sums) become more accurate. We next chose μ_1 and μ_2 by running some tuning games using transfer advice, selecting $\mu_1 = 0.01$ and $\mu_2 = 1.0$. We decay these initial μ values by $e^{(-\#GamesPlayed/2{,}500)}$, since we expect that transfer advice will be less valuable as the amount of expe-

rience in the new domain increases. We only tried a small number of possible settings for each parameter, and after choosing our parameter values, we trained on a fresh set of games – we report the results on this final set of games.

Figure 3 shows the results of our transfer experiments. Training runs with and without transfer advice are compared. The curves were generated by batch training after every 100 games and averaging over 10 different runs; each data point shown is smoothed over the previous 1000 games (the results from the first 100 games are not included in these averages except at the first point on the x axis, where Games Played = 100, since no learning took place until after the first 100 games).

The curve on the left shows the average reinforcement per game that learners earn as they train, since that is the quantity that the learners attempt to maximize. The curve on the right shows a more intuitive measure of performance: the probability that the learners will score a goal as a function of the number of games played.

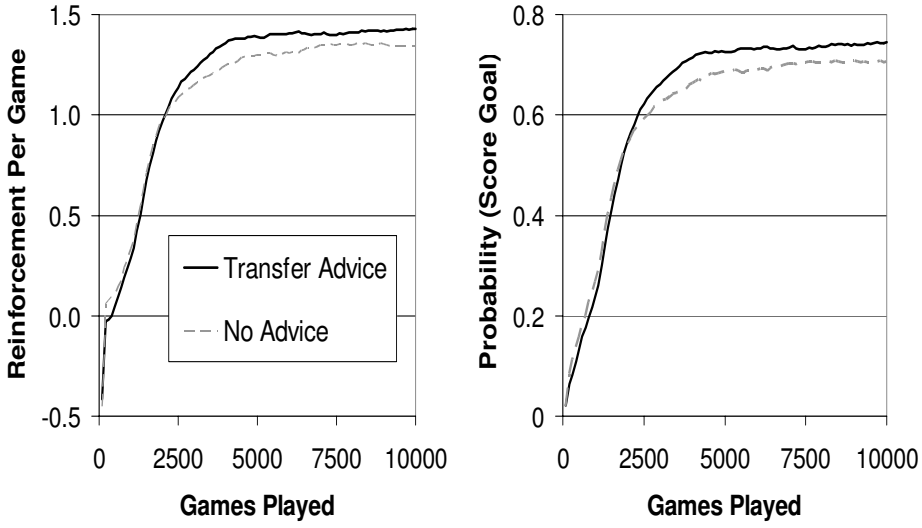

Fig. 3. Performance as a function of BreakAway games played by two different metrics, with a learner using transfer advice compared against a learner using no advice

These results show that the transfer advice gives a reinforcement learner a modest advantage in learning to score goals. Advice was slightly detrimental at first as the learners refined it, but after about 2,500 games it began to improve performance, and in the end it led to a higher asymptotic result. We have obtained qualitively similar results when transferring to 3-on-2 BreakAway, although in this task the probability of scoring a goal is above 0.5.

7 Related Work

A number of researchers have explored methods for providing advice to learning algorithms. Clouse and Utgoff [2] allow a human observer to step in and advise the learner to take a specific action. Lin [6] "replays" teacher sequences to bias a learner towards a teacher's performance. Gordon and Subramanian [3] accept advice in the form IF *condition* THEN *achieve goals* and then use genetic algorithms to adjust it with respect to the data. Maclin and Shavlik [7] also developed an IF-THEN advice language, but incorporated the rules into a neural network for later adjustment. Price and Boutilier [12] designed a method for reinforcement learners to imitate expert agents in the same domain. Andre and Russell [1] describe a language for creating learning agents whose policies are constrained by user commands. Laud and DeJong [5] use reinforcements to shape the learner. Kuhlmann et al. [4] developed a rule-based advice system that increases Q-values by a fixed amount. In recent work [9], we developed the Preference-KBKR method, which allows advice to be specified in the form of action preferences. Our current work differs from these previous advice-taking methods because it extracts advice from a model learned in another task, instead of employing user-designed advice.

Other related work deals with knowledge transfer in machine learning. Some early research focuses on learning a simpler version of a task and applying that knowledge to a more difficult version of the same task. Selfridge et al. [13] call this "directed training" and use it in robotics. Singh [15] addresses transfer of knowledge between sequential decision tasks, where an agent keeps track of useful action subsequences for use in later tasks. Thrun and Mitchell [19] study transfer between problems in a "lifelong learning" framework of many related Boolean classification tasks. Taylor and Stone [18] have investigated copying Q-functions to transfer between KeepAway games of different team sizes. Sherstov and Stone [14] have investigated "action transfer" in RL, which uses transfer to improve learning on tasks with large action spaces.

8 Conclusions and Future Work

We have presented a novel technique of extracting knowledge gained on one task and automatically transferring it to a related task to improve learning. Our experiments demonstrate that the difficult BreakAway task in RoboCup soccer can be learned more effectively using advice transferred from the related KeepAway task.

Our key idea is that we can view the models learned in an old task as a source of advice for a new task. Since we represent both our learned models and advice as linear expressions of features, all the user needs to do is match the features and actions of the old task to the new one. Since advice-taking systems are robust to imperfections in the advice they receive, the user's guidance need only be approximate.

In future work, we plan to evaluate the sensitivity of our algorithm to errors and omissions in the user's mapping advice. We may also investigate ways to

further automate the process by helping the user design a mapping. We have already begun to adapt the transfer advice process to work with non-linear models.

When a new learning task arises in a domain, it is likely that human experts will be able to provide information about how the new task relates to known tasks. Learning algorithms should be able to exploit this information to extract knowledge from these known tasks. Transfer advice shows potential as an effective and intuitive way to do this. It can increase human ability to interact productively with reinforcement learners, and we believe that such interaction will be important for scaling RL to large problems.

Acknowledgements

The research is partially supported by DARPA grant HR0011-04-1-0007 and United States Naval Research Laboratory grant N00173-04-1-G026. The BreakAway code is available at ftp://ftp.cs.wisc.edu/machine-learning/shavlik-group/robocup/breakaway. We would also like to thank Michael Ferris for his assistance in improving the efficiency of our linear programs.

References

1. D. Andre and S. Russell. Programmable reinforcement learning agents. In *NIPS*, 2001.
2. J. Clouse and P. Utgoff. A teaching method for reinforcement learning. In *Proc. ICML '92*, 1992.
3. D. Gordon and D. Subramanian. A multistrategy learning scheme for agent knowledge acquisition. *Informatica*, 17:331–346, 1994.
4. G. Kuhlmann, P. Stone, R. Mooney, and J. Shavlik. Guiding a reinforcement learner with natural language advice: Initial results in RoboCup soccer. In *AAAI Workshop on Supervisory Control of Learning and Adaptive Systems*, 2004.
5. A. Laud and G. DeJong. Reinforcement learning and shaping: Encouraging intended behaviors. In *ICML*, 2002.
6. L. Lin. Self-improving reactive agents based on reinforcement learning, planning, and teaching. *Machine Learning*, 8:293–321, 1992.
7. R. Maclin and J. Shavlik. Creating advice-taking reinforcement learners. *Machine Learning*, 22:251–281, 1996.
8. R. Maclin, J. Shavlik, L. Torrey, and T. Walker. Knowledge-based support vector regression for reinforcement learning. In *IJCAI Workshop on Reasoning, Representation, and Learning in Computer Games*, 2005.
9. R. Maclin, J. Shavlik, L. Torrey, T. Walker, and E. Wild. Giving advice about preferred actions to reinforcement learners via knowledge-based kernel regression. In *AAAI*, 2005.
10. O. Mangasarian, J. Shavlik, and E. Wild. Knowledge-based kernel approximation. *JMLR*, 5:1127–1141, 2004.
11. I. Noda, H. Matsubara, K. Hiraki, and I. Frank. Soccer server: A tool for research on multiagent systems. *Applied Artificial Intelligence*, 12:233–250, 1998.

12. B. Price and C. Boutilier. Implicit imitation in multiagent reinforcement learning. In *ICML*, 1999.
13. O. Selfridge, R. Sutton, and A. Barto. Training and tracking in robotics. In *IJCAI*, 1985.
14. A. Sherstov and P. Stone. Improving action selection in MDP's via knowledge transfer. In *AAAI*, 2005.
15. S. Singh. Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning*, 8(3-4):323–339, 1992.
16. P. Stone and R. Sutton. Scaling reinforcement learning toward RoboCup soccer. In *ICML*, 2001.
17. R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.
18. M. Taylor and P. Stone. Behavior transfer for value-function-based reinforcement learning. In *4th Int. Joint Conf. on Autonomous Agents and Multiagent Sys.*, 2005.
19. S. Thrun and T. Mitchell. Learning one more thing. In *IJCAI*, 1995.