Advice Taking and Transfer Learning: Naturally Inspired Extensions to Reinforcement Learning

Lisa Torrey, Trevor Walker, Richard Maclin*, Jude Shavlik
Computer Sciences Department
University of Wisconsin-Madison
*Computer Sciences Department
University of Minnesota-Duluth

Abstract

Reinforcement learning (RL) is a machine learning technique with strong links to natural learning. However, it shares several "unnatural" limitations with many other successful machine learning algorithms. RL agents are not typically able to take advice or to adjust to new situations beyond the specific problem they are asked to learn. Due to limitations like these, RL remains slower and less adaptable than natural learning. Our recent work focuses on extending RL to include the naturally inspired abilities of *advice taking* and *transfer learning*. Through experiments in the RoboCup domain, we show that doing so can make RL faster and more adaptable.

Introduction

In reinforcement learning (RL), an agent navigates through an environment trying to earn rewards. While many machine learning tasks require learning a single decision model, an RL task involves revising a decision model over many episodes. The challege for the agent is to handle delayed rewards; that is, to learn to choose actions that may be locally sub-optimal in order to maximize later rewards. The RL framework can be viewed as a formalization of the trial-and-error process that natural learners often use.

However, RL diverges from natural learning in that agents typically begin without any information about their environment or rewards. While natural learners often have a teacher or previous experience to apply, agents in model-free RL typically are *tabula rasa*. Therefore RL often requires substantial amounts of random exploration in the early stages of learning, and it can require long training times in complex domains.

Our recent work focuses on providing RL with two benefits that natural learners often have: the ability to take advice from a teacher, and the ability to transfer knowledge from previous experiences. We refer to these abilities as *advice taking* and *transfer learning*. Through experiments in the complex domain of RoboCup soccer, we show that they can speed up RL significantly.

Appears in the 2008 AAAI Fall Symposium on Naturally Inspired AI.

Reinforcement Learning in RoboCup

We begin with a brief overview of reinforcement learning (Sutton and Barto 1998), particularly in the context of our experimental domain of RoboCup soccer (Noda et al. 1998).

An RL agent navigates through an environment trying to earn rewards. The environment's state is described by a set of features, and the agent takes actions to cause the state to change. In one common form called *Q*-learning (Watkins 1989), the agent learns a *Q*-function to estimate the value of taking an action from a state. An agent's *policy* is typically to take the action with the highest *Q*-value in the current state, except for occasional exploratory actions. After taking the action and receiving some reward (possibly zero), the agent updates its *Q*-value estimates for the current state.

Stone and Sutton (2001) introduced RoboCup as an RL domain that is challenging because of its large, continuous state space and non-deterministic action effects. Since the full game of soccer is quite complex, they developed a smaller RL task called KeepAway and did some feature engineering to make it feasible. Using a similar feature design, other researchers have developed other small tasks, such as MoveDownfield and BreakAway (see Figure 1).

In M-on-N KeepAway, the objective of the M keepers is to keep the ball away from the N takers, by taking *pass* and *hold* actions and receiving a +1 reward for each time step they keep the ball. In M-on-N MoveDownfield, the objective of the M attackers is to move across a line on the opposing team's side of the field, by taking *move* and *pass* actions and receiving symmetrical positive and negative rewards for horizontal movement forward and backward. In M-on-N BreakAway, the objective of the M attackers is to

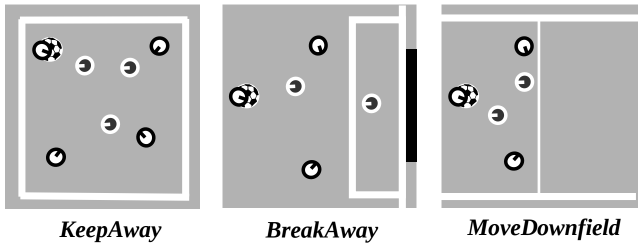


Figure 1: Snapshots of RoboCup soccer tasks.

Table 1: The features that describe a BreakAway state.

distBetween(a0, Player)
distBetween(a0, GoalPart)
distBetween(Attacker, goalCenter)
distBetween(Attacker, ClosestDefender)
distBetween(Attacker, goalie)
angleDefinedBy(topRight, goalCenter, a0)
angleDefinedBy(GoalPart, a0, goalie)
angleDefinedBy(Attacker, a0, ClosestDefender)
angleDefinedBy(Attacker, a0, goalie)
timeLeft

score a goal against $N-1$ defenders and a goalie, by taking *move*, *pass*, and *shoot* actions and receiving a +1 reward for scoring.

RoboCup tasks are inherently multi-agent games, but a standard simplification is to have only one learning agent. This agent controls the attacker currently in possession of the ball, switching its "consciousness" between attackers as the ball is passed. Attackers without the ball follow simple hand-coded policies that position them to receive passes. Opponents also follow hand-coded policies.

The set of features describing the environment in each task mainly consists of distances and angles between players and objects. For example, Table 1 shows the BreakAway features. They are represented in logical notation for future convenience, though our RL algorithm uses the grounded versions of these predicates in a fixed-length feature vector. Capitalized atoms indicate typed variables, while constants and predicates are uncapitalized. The attackers (labeled *a0*, *a1*, etc.) are ordered by their distance to the agent in possession of the ball (*a0*), as are the non-goalie defenders (*d0*, *d1*, etc.).

Our RL implementation uses a $SARSA(\lambda)$ variant of *Q*-learning (Sutton 1988) and employs a support vector machine for function approximation (Maclin et al. 2005b). Because this method is more appropriate for batch learning than for incremental learning, we relearn the *Q*-function after every batch of 25 games.

We represent the state with a set of numeric features and approximate the *Q*-function for each action with a weighted linear sum of those features, learned via support-vector regression. To find the feature weights, we solve a linear optimization problem, minimizing the following quantity:

$$\text{ModelSize} + C \times \text{DataMisfit}$$

Here *ModelSize* is the sum of the absolute values of the feature weights, and *DataMisfit* is the disagreement between the learned function's outputs and the training-example outputs (i.e., the sum of the absolute values of the differences for all examples). The numeric parameter C specifies the relative importance of minimizing disagreement with the data versus finding a simple model.

Formally, for each action the agent finds an optimal weight vector w that has one numeric weight for each feature in the feature vector x. The expected *Q*-value of taking an action from the state described by vector x is $wx + b$, where b is a scalar offset. Our learners use the ϵ-greedy exploration method (Sutton and Barto 1998).

To learn 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 current model and the actual rewards received in the examples, we compute *Q*-value estimates and place them into an output vector y. The optimal weight vector is then described by Equation 1.

$$Aw + b\overrightarrow{e} = y \tag{1}$$

where $\overrightarrow{e}$ 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. Furthermore, an exact linear solution may not exist for any given training set. We therefore introduce *slack* variables s that allow inaccuracies on some examples, and a penalty parameter C for trading off these inaccuracies with the complexity of the solution. The resulting minimization problem is

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

where $|\cdot|$ denotes an absolute value, $||\cdot||_1$ denotes the one-norm (a sum of absolute values), and the scalar 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. This is the algorithm that we refer to throughout the paper as "standard RL" and that we extend to perform advice taking and transfer.

Figure 2 shows the learning curve in 3-on-2 BreakAway using this algorithm. It measures the probability that the agents will score a goal as they train on more games. The curve is an average of 25 runs and each point is smoothed over the previous 500 games to account for the high variance in the RoboCup domain.

The RL agents have an asymptotic performance of scoring in around 50% of their games. Given that their hand-coded opponents begin with a much higher skill level, scoring in

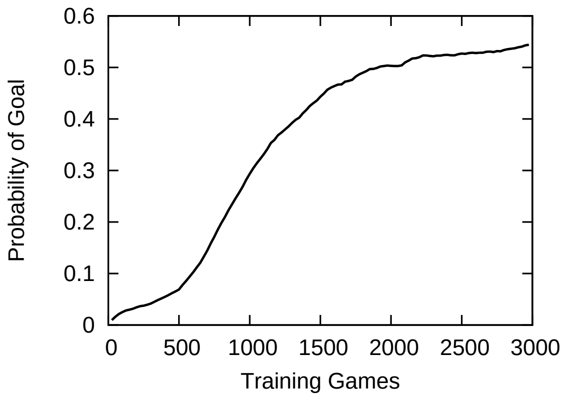


Figure 2: Probability of scoring a goal in 3-on-2 BreakAway as the RL algorithm trains.

50% of the games is a relatively good rate. A random policy, despite the engineered actions, only scores about 1% of the time.

It takes the RL agents about 2000 episodes to reach this asymptote, which is much slower than the human learning rate might be in a real-world version of this task. However, the agents have several disadvantages compared to humans. A human playing soccer knows the objective of the game in advance, while the agents have to learn it indirectly through rewards. The agents' only rewards come upon scoring, while humans can derive intermediate feedback by reasoning; humans can also generalize that feedback across a wider range of situations than agents typically can. Humans can apply logic and planning to the problem, while RL has only trial-and-error and statistical analysis, which requires a certain amount of data to produce an accurate function approximation. The agents also lack any previous experience or outside instruction in the task, which humans usually have.

Despite all these disadvantages, reinforcement learning is often eventually successful in learning a good policy. However, it can take a long time to do so. In this paper, we address two of the disadvantages and propose methods for speeding up RL by incorporating abilities that humans typically have.

Advice Taking

When humans learn complex tasks, we often have instructions to follow or teachers to provide guidance. This allows us to avoid the blind exploration that typically occurs at the beginning of reinforcement learning. In this section, we describe an RL algorithm that accepts guidance from an outside source. We refer to this guidance as *advice*.

There is a body of related work on advice taking in RL, some of which we describe briefly here. Maclin and Shavlik (1996) accept rules for action selection and incorporate them into a neural-network *Q*-function model. Driessens and Dzeroski (2002) use behavior traces from an expert to learn a partial initial *Q*-function for relational RL. Kuhlmann et al. (2004) accept rules that give a small boost to the *Q*-values of actions, which is also one method used for transfer in Taylor and Stone (2007).

We view advice as a set of soft constraints on the *Q*-function of an RL agent. For example, here is a possible advice rule for passing in RoboCup:

IF	an opponent is near me AND
	a teammate is open
THEN	*pass* is the best action

In the IF portion of the rule, there are two conditions describing the agent's environment: an opponent is getting close and there is an unblocked path to a teammate. The THEN portion gives a constraint saying that *pass* should have the highest *Q*-value when the environment matches these conditions.

Our advice-taking RL algorithm incorporates advice into our support-vector function-approximation method. Advice creates new constraints on the problem solution, in addition to the usual constraints that encourage fitting the data in the episodes played so far. This method allows for imperfect advice because it balances between fitting the training data and following the advice. We believe this is important for advice-taking in RL, since advice is likely to come from human users who may not have perfect domain knowledge.

Formally, our advice takes the following form:

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

This can be read as:

If the current state x satisfies $Bx \leq d$, then 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 action p is better than action n when the value of feature 5 is at most 10. The vector B would have one row with a 1 in the column for feature 5 and zeros elsewhere. The vector d would contain only the value 10, and β could be set to some small positive number.

Just as we allowed some inaccuracy on the training examples in Equation 2, we allow advice to be followed only partially. To do so, we introduce slack variables as well as a penalty parameter μ for trading off the impact of the advice with the impact of the training examples.

The new optimization problem solves the *Q*-functions for all the actions simultaneously 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, which makes it possible to advise taking a particular action by stating that it is preferred over all the other actions. We use the CPLEX commercial software to minimize:

$$\text{ModelSize} + C \times \text{DataMisfit} + \mu \times \text{AdviceMisfit}$$

See Maclin et al. (Maclin et al. 2005a) for further details on how advice is converted into constraints in the linear program.

Advice can speed up RL significantly. To demonstrate this, we provide the following simple advice to 3-on-2 BreakAway learners:

IF	distBetween(a0, GoalPart) < 10
AND	angleDefinedBy(GoalPart, a0, goalie) > 40
THEN	prefer shoot(GoalPart) over all actions

Figure 3 shows the results of this experiment. With advice, BreakAway learners perform significantly better in the early stages of learning, and they maintain an advantage until they reach their performance asymptote. Based on unpaired *t*-tests between points on the curves at the 95% confidence level, scoring is more probable with advice until about 1750 training games.

Transfer Learning

Knowledge transfer is an inherent aspect of human learning. When humans learn to perform a task, we rarely start from scratch. Instead, we recall relevant knowledge from previous learning experiences and apply that knowledge to help

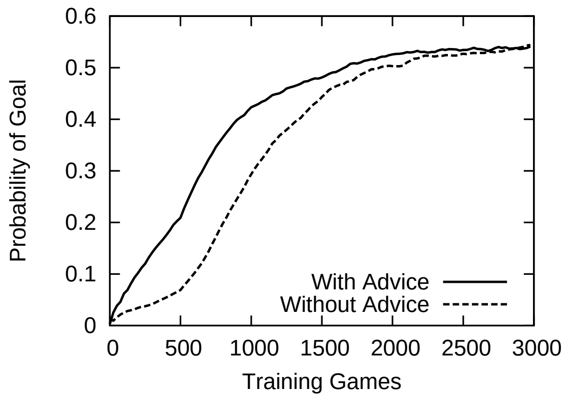


Figure 3: Probability of scoring a goal in 3-on-2 BreakAway with and without shooting advice.

us master the new task more quickly. In this section, we describe several algorithms for transferring knowledge from a *source task* to a *target task* in reinforcement learning.

There is a body of related work on transfer learning in RL, some of which we describe briefly here. Taylor, Stone and Liu (2005) begin performing the target task using the source-task value functions, after performing a suitable mapping between features and actions in the tasks. Fernandez and Veloso (2006) follow source-task policies during the exploration steps of normal RL in the target task. Croonenborghs, Driessens and Bruynooghe (2007) learn multi-step action sequences called options from the source task, and add these as possible actions in the target task.

This section discusses three algorithms from our recent work: skill transfer, macro transfer, and Markov Logic Network transfer. The common property of these algorithms is that they are *relational*. Relational methods generalize across objects in a domain and may use first-order logic to describe knowledge. This can allow them to produce better generalization to new tasks because they capture concepts about logical variables rather than constants. We also view first-order logic as a naturally inspired tool for transfer, since humans often perform this type of reasoning.

To do relational learning in these transfer methods, we use inductive logic programming (ILP), and specifically the Aleph system (Srinivasan 2001). Aleph selects an example, builds the most specific clause that entails the example, and searches for generalizations of this clause that cover other examples while maximizing a provided scoring function.

Scoring functions typically involve *precision* and *recall*. The precision of a rule is the fraction of examples it calls positive that are truly positive, and the recall is the fraction of truly positive examples that it correctly calls positive. The scoring function we use is

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall}$$

because we consider both precision and recall to be important. We use both a heuristic and randomized search algorithm to find potential rules.

Skill Transfer

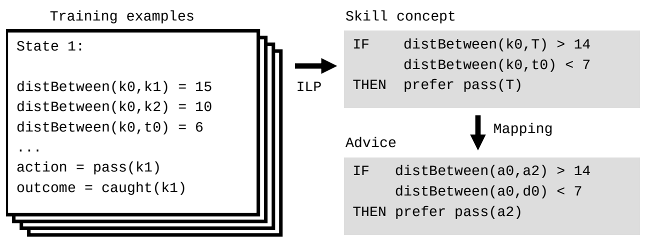
Skill transfer (Torrey et al. 2006b; 2006a) is a method that gives advice about skills the source and target tasks have in common. We use ILP to learn first-order logical rules describing when actions have high *Q*-values in the source task. Then we use our advice-taking algorithm, as described earlier, to apply these rules as soft constraints on when to take actions in the target task. The difference between skill transfer and advice taking is that the advice comes from automated analysis of a source task rather than from a human.

For example, Figure 4 shows the process of transferring the skill *pass(Teammate)* from KeepAway to BreakAway. We assume the user provides a mapping between logical objects in the source and target tasks (e.g., *k0* in KeepAway maps to *a0* in BreakAway) and we only allow the ILP system to use feature predicates that exist in both tasks during its search for rules.

To make the search space finite, it is necessary to replace continuous features (like distances and angles) with finite sets of discrete features. Our system finds the 25 thresholds with the highest information gain and allows the intervals above and below those thresholds to appear as constraints in rules. At the end of the ILP search, it chooses one rule per skill with the highest F1 score to transfer.

Because some skills might be new in the target task, and because we are already using the advice-taking mechanism, we also allow human-provided advice about new actions in this method. For example, when transferring from KeepAway or MoveDownfield to BreakAway, we learn the *pass(Teammate)* skill with ILP but we manually provide advice for *shoot(GoalPart)*.

Skill transfer can speed up RL significantly. To demonstrate this, we learn 3-on-2 BreakAway with transfer from three source tasks: 2-on-1 BreakAway, 3-on-2 MoveDownfield, and 3-on-2 KeepAway. Figure 5 shows the results of these experiments. With transfer learning, BreakAway learners perform significantly better in the early stages of learning, and they maintain an advantage until they reach their performance asymptote. Based on unpaired *t*-tests between points on the curves at the 95% confidence level, scoring is more probable with skill transfer from 2-on-1 BreakAway until about 1500 training games, from 3-on-2 MoveDownfield until about 1750 training games, and from 3-on-2 KeepAway until about 2750 training games.



```
State 1:

distBetween(k0,k1) = 15
distBetween(k0,k2) = 10
distBetween(k0,t0) = 6
...
action = pass(k1)
outcome = caught(k1)
```

```
IF    distBetween(k0,T) > 14
      distBetween(k0,t0) < 7
THEN  prefer pass(T)
```

```
IF   distBetween(a0,a2) > 14
     distBetween(a0,d0) < 7
THEN prefer pass(a2)
```

Figure 4: An example of the skill transfer process for the pass(Teammate) skill when KeepAway is the source task and BreakAway is the target task. This uses the same capitalization conventions as in Table 1.

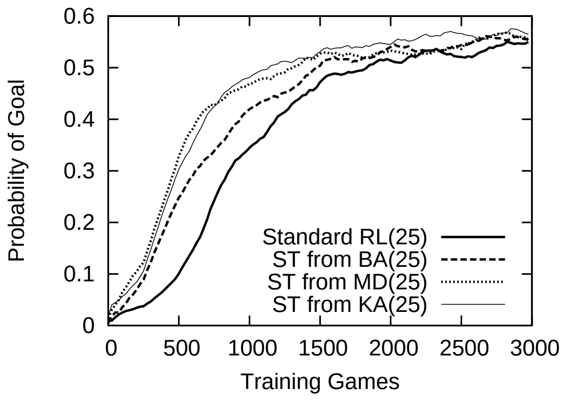


Figure 5: Probability of scoring a goal while training in 3-on-2 BreakAway, with standard RL and with skill transfer (ST) from 2-on-1 BreakAway (BA), 3-on-2 MoveDownfield (MD) and 3-on-2 KeepAway (KA).

Macro Transfer

Macro transfer (Torrey et al. 2007) is a method that transfers action plans composed of several skills in sequence. We use ILP to learn relational macros and apply them in the target task by *demonstration*; agents follow the macros directly for an initial period before reverting to standard RL.

A relational macro is a finite-state machine (Gill 1962), which models the behavior of a system in the form of a directed graph. The nodes of the graph represent states of the system, and in our case they represent internal states of the agent in which different policies apply.

The policy of a node can be to take a single action, such as *move(ahead)* or *shoot(goalLeft)*, or to choose from a class of actions, such as *pass(Teammate)*. In the latter case a node has first-order logical clauses to decide which grounded action to choose. A finite-state machine begins in a start node and has conditions for transitioning between nodes. In a relational macro, these conditions are also sets of first-order logical clauses.

Figure 6 shows a sample macro. When executing this macro, a KeepAway agent begins in the initial node on the left. The only action it can choose in this node is *hold*. It remains there, taking the default self-transition, until the condition *isClose(Opponent)* becomes true for some opponent player. Then it transitions to the second node, where it evaluates the *pass(Teammate)* rule to choose an action. If the rule is true for just one teammate player, it passes to that teammate; if several teammates qualify, it randomly chooses between them; if no teammate qualifies, it abandons the macro and reverts to using the *Q*-function to choose actions. Assuming it does not abandon the macro, once another teammate receives the ball it becomes the learning agent and remains in the *pass* node if an opponent is close or transitions back to the *hold* node otherwise.

Figure 6 is a simplification in one respect: each transition and node in a macro can have an entire set of rules, rather than just one rule. This allows us to represent disjunctive conditions. When more than one grounded action or transition is possible (when multiple rules match), the agent obeys the rule that has the highest score. The score of a rule is the estimated probability that following it will lead to a successful game, as estimated from the source-task data.

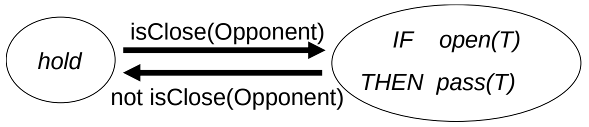


Figure 6: A possible strategy for the RoboCup game KeepAway, in which the RL agent in possession of the soccer ball must execute a series of *hold* or *pass* actions to prevent its opponents from getting the ball. The rules inside nodes show how to choose actions. The labels on arcs show the conditions for taking transitions. Each node has an implied self-transition that applies by default if no exiting arc applies. If no action can be chosen in a node, the macro is abandoned.

Since a macro consists of a set of nodes along with rulesets for transitions and action choices, the simplest algorithm for learning a macro might be to have Aleph learn both the structure and the rulesets simultaneously. However, this would be a very large search space. To make the search more feasible, we separate it into two phases: first we learn the structure, and then we learn each ruleset independently.

In the structure-learning phase, the objective is to find a sequence of actions that distinguishes successful games from unsuccessful games. We use Aleph to find the action sequence with the highest F1 score. For example, a BreakAway source task might produce the sequence in Figure 7.

In the ruleset-learning phase, the objective is to describe when transitions and actions should be taken within the macro structure. These decisions are made based on the RL state features. We use Aleph to search for rules, and we store all the clauses that Aleph encounters during the search that classify the training data with at least 50% accuracy.

Instead of selecting a single best clause as we did for structure learning, we select a set of rules for each transition and each action. We wish to have one strategy (i.e. one finite-state machine), but there may be multiple reasons for making internal choices. To select rules we use a greedy approach: we sort the rules by decreasing precision and walk through the list, adding rules to the final ruleset if they increase the set's recall and do not decrease its F1 score.

We assign each rule a score that may be used to decide which rule to obey if multiple rules match while executing the macro. The score is an estimate of the probability that following the rule will lead to a successful game. We determine this estimate by collecting training-set games that followed the rule and calculating the fraction of these that ended successfully. Since BreakAway has *Q*-values ranging from zero to one, we simply estimate *Q*-values by the rule score (otherwise we could multiply the probability by an appropriate scaling factor to fit a larger *Q*-value range).

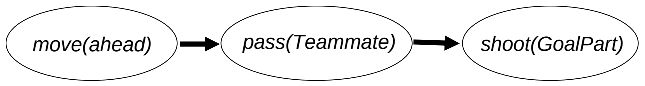


Figure 7: A macro structure that could be learned for BreakAway in the structure-learning phase of macro transfer.

A relational macro describes a strategy that was successful in the source task. There are several ways we could use this information to improve learning in a related target task. One possibility is to treat it as advice, as we did in skill transfer, putting soft constraints on the *Q*-learner that influence its solution. The benefit of this approach is its robustness to error: if the source-task knowledge is less appropriate to the target task than the user expected, the target-task agent can learn to disregard the soft constraints, avoiding negative transfer effects.

On the other hand, the advice-taking approach is conservative and can be somewhat slow to reach its full effect, even when the source-task knowledge is highly appropriate to the target task. Since a macro is a full strategy rather than isolated skills, we might achieve good target-task performance more quickly by executing the strategy in the target task and using it as a starting point for learning. This *demonstration* method is a more aggressive approach, carrying more risk for negative transfer if the source and target tasks are not similar enough. Still, if the user believes that the tasks are similar, the potential benefits could outweigh that risk.

Our target-task learner therefore begins by simply executing the macro strategy for a set of episodes, instead of exploring randomly as an untrained RL agent would traditionally do. In this demonstration period, we generate (state, *Q*-value) pairs: each time the macro chooses an action because a high-scoring rule matched, we use the rule score to estimate the *Q*-value of the action. The demonstration period lasts for 100 games in our system, and as usual after each batch of 25 games we relearn the *Q*-function.

After 100 games, we continue learning the target task with standard RL. This generates new *Q*-value examples in the standard way, and we combine these with the old macro-generated examples as we continue relearning the *Q*-function after each batch. As the new examples accumulate, we gradually drop the old examples by randomly removing them at the rate that new ones are being added.

Macro transfer can speed up RL significantly as well, but in a different way than skill transfer does. To demonstrate this, we learn 3-on-2 BreakAway with transfer from 2-on-1 BreakAway. Figure 8 shows the results of this experiment. It also includes results for the same experiment with skill transfer and with value-function transfer, which simply uses the final source-task model as the initial target-task model. Macro transfer speeds up RL more dramatically than skill transfer at the beginning of learning, but it loses its advantage sooner as learning proceeds. Based on unpaired *t*-tests between points on the curves at the 95% confidence level, scoring is more probable with macro transfer until about 1000 training games.

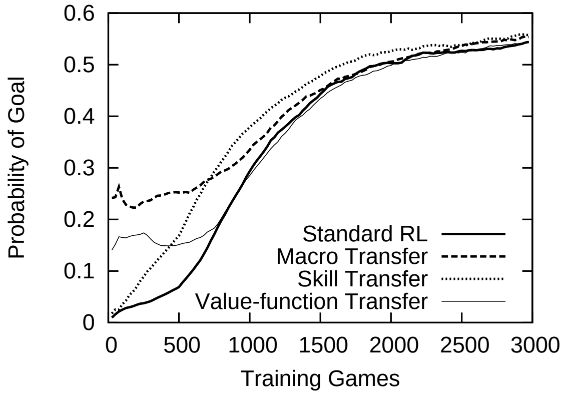


Figure 8: Probability of scoring a goal in 3-on-2 BreakAway, with standard RL and with three transfer approaches that use 2-on-1 BreakAway as the source task.

Markov Logic Network Transfer

Statistical relational learning (SRL) is a type of machine learning designed to operate in domains that have both uncertainty and rich relational structure. It focuses on combining the two powerful paradigms of first-order logic, which generalizes among the objects in a domain, and probability theory, which handles uncertainty. One recent and popular SRL formalism is the Markov Logic Network (MLN), introduced by Richardson and Domingos (2005), which interprets first-order statements as soft constraints with weights. This framework can allow us to express source-task knowledge with relational structure as well as non-deterministic events.

MLN transfer (Torrey et al. 2008) is a method that transfers an MLN describing the source-task *Q*-function. We use both ILP and MLN software to learn the MLN. As in macro transfer, we apply this knowledge in the target task by demonstration.

A Markov network can be viewed as a set of ground predicates with potential functions that define a probability distribution over possible worlds. A Markov Logic Network is a set of first-order logic formulas that can be grounded to form a Markov network. Each formula describes a property that may be present in the world, and has an associated real-valued weight. Worlds become more probable as they satisfy more high-weighted formulas.

Given a set of formulas along with positive and negative examples of worlds, the weights can be learned via gradient descent (Lowd and Domingos 2007). Then, given a set of *evidence* about a world–a list of predicates that are known to be true or false–standard inference in the ground Markov network can determine the probability that the remaining predicates are true or false.

We use an MLN to define a probability distribution for the *Q*-value of an action, conditioned on the state features. In this scenario, a world corresponds to a state in the RL environment, and a formula describes some characteristic that helps determine the *Q*-value of an action in that state. For example, assume that there is a discrete set of *Q*-values that a RoboCup action can have: *high*, *medium*, and *low*. In this simplified case, formulas in an MLN representing the *Q*-function for BreakAway could look like the following example:

```
IF      distBetween(a0, GoalPart) < 10 AND
        angleDefinedBy(GoalPart, a0, goalie) > 40
THEN    levelOfQvalue(shoot(GoalPart)) = high
```

The MLN could contain multiple formulas like this for each action, each with a weight learned via gradient descent from a training set of source-task states in which all the properties and *Q*-values are known. We could then use this MLN to evaluate action *Q*-values in a target-task state: we evaluate which properties are present and absent in the state, give that information as evidence, and infer the probability that each action's *Q*-value is high, medium, or low.

Note that *Q*-values in RoboCup are continuous rather than discrete, so we do not actually learn rules classifying them as high, medium, or low. However, we do discretize the *Q*-values into bins, using hierarchical clustering to find bins that fit the data. Initially every training example is its own cluster, and we repeatedly join clusters whose midpoints are closest until there are no midpoints closer than ϵ apart. The final cluster midpoints serve as the midpoints of the bins.

The value of ϵ is domain-dependent. For BreakAway, which has *Q*-values ranging from approximately 0 to 1, we use $\epsilon = 0.1$. This leads to a maximum of about 11 bins, but there are often less because training examples tend to be distributed unevenly across the range. We experimented with ϵ values ranging from 0.05 to 0.2 and found very minimal differences in the results; the approach appears to be robust to the choice of ϵ within a reasonably wide range.

The MLN transfer process begins by finding these bins and learning rules with Aleph that classify the training examples into the bins. We select the final rulesets for each bin with the same greedy algorithm as for rulesets in macro transfer. We then learn formula weights for these rules using the scaled conjugate-gradient algorithm in the Alchemy MLN implementation (Kok et al. 2005). We typically end up with a few dozen formulas per bin in our experiments.

To use a transferred MLN in a target task, we use the demonstration approach again. Given an MLN *Q*-function, we can estimate the *Q*-value of an action in a target-task state with the algorithm in Table 2. We begin by performing inference in the MLN to estimate the probability, for each action and bin, that *levelOfQvalue(action, bin)* is true. Typically, inference in MLNs is approximate because exact inference is intractable for most networks, but in our case exact inference is possible because there are no missing features and the Markov blanket of a query node contains only known evidence nodes.

For each action a, we infer the probability p_b that the *Q*-value falls into each bin b. We then use these probabilities as weights in a weighted sum to calculate the *Q*-value of a:

$$Q_a(s) = \sum_b (p_b * E[Q_a|b])$$

where $E[Q_a|b]$ is the expected *Q*-value given that b is the correct bin. We estimate this by the *Q*-value of the training example in the bin that is most similar to state s. This method performed slightly better than taking the average *Q*-value of all the training examples in the bin, which would be another reasonable estimate for the expected *Q*-value. The similarity measure between two states is the dot product of two vectors that indicate which of the bin clauses the states satisfy. For each formula, a vector has a $+1$ entry if the state satisfies it or a -1 entry if it does not.

Table 2: Our algorithm for calculating the Q-value of action a in target-task state s using the MLN Q-function.

```
Provide state $s$ to the MLN as evidence
For each bin $b \in [1, 2, ..., n]$
    Infer the probability $p_b$ that $Q_a(s)$ falls into bin $b$
    Find the training example $t$ in bin $b$ most similar to $s$
    Let $E[Q_a|b] = Q_a(t)$
Return $Q_a(s) = \sum_b (p_b * E[Q_a|b])$
```

MLN transfer can also speed up RL significantly at the beginning of learning. To demonstrate this, we learn 3-on-2 BreakAway with transfer from 2-on-1 BreakAway. Figure 9 shows the results of this experiment, compared with macro transfer and value-function transfer. MLN transfer performs comparably to macro transfer. Based on unpaired *t*-tests between points on the curves at the 95% confidence level, scoring is more probable with MLN transfer until about 1000 training games.

Conclusions

Advice taking and transfer learning are naturally inspired abilities that our work incorporates into reinforcement learning. We describe algorithms for doing so and show that they can speed up RL significantly in complex domains like RoboCup. Our algorithms are relational, using first-order logic to describe concepts.

Our advice-taking method allows humans to provide guidance to RL agents with rules about the *Q*-values of actions under specified conditions. The advice need not be complete or perfectly correct, since it is treated as a soft constraint that can be violated if the agent's experience indicates that it is faulty.

Our three transfer methods involve learning relational knowledge about the source-task solution and using it in the target task. The skill-transfer method learns rules about the *Q*-values of actions under specified conditions, and ap-

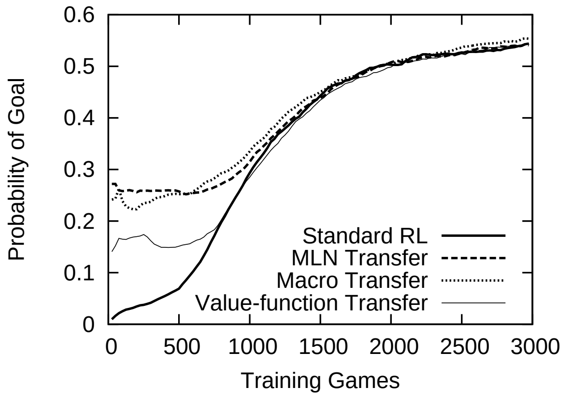


Figure 9: Probability of scoring a goal in 3-on-2 BreakAway, with *Q*-learning and with three transfer approaches that use 2-on-1 BreakAway as the source task.

plies them using our advice-taking algorithm. The macro-transfer and MLN-transfer approaches learn more complex structures and use them to demonstrate useful behavior at the beginning of the target task. All three methods can speed up learning significantly, but they have different strengths. Macro and MLN transfer can provide a higher initial performance than skill transfer, but skill transfer is more robust to differences between the source and target tasks.

This work is inspired by the natural abilities for advice taking and transfer learning that we observe in humans. Reinforcement learning can be faster and more adaptable with these abilities. Though we do not know exactly how humans handle advice or transfer, we suspect that representing knowledge with relational structures is part of the human approach.

Many machine learning algorithms might be improved by developing their links to natural learning. Traditionally, machine learning operates only from a set of training examples, and a large number of these are often needed to learn concepts well. Natural learning tends to need fewer examples because it uses other sources of information as well, such as advice and prior experience. These sources may not be perfect or complete, but they provide valuable biases for learning that can speed it up significantly.

Acknowledgements

This research was supported by DARPA grants HR0011-04-1-0007, HR0011-07-C-0060 and FA8650-06-C-7606 as well as United States Naval Research Laboratory grant N00173-06-1-G002.

References

Croonenborghs, T.; Driessens, K.; and Bruynooghe, M. 2007. Learning relational skills for inductive transfer in relational reinforcement learning. In *2007 International Conference on Inductive Logic Programming*.

Driessens, K., and Dzeroski, S. 2002. Integrating experimentation and guidance in relational reinforcement learning. In *2002 International Conference on Machine Learning*.

Fernandez, F., and Veloso, M. 2006. Policy reuse for transfer learning across tasks with different state and action spaces. In *2006 ICML Workshop on Structural Knowledge Transfer for Machine Learning*.

Gill, A. 1962. *Introduction to the Theory of Finite-state Machines*. McGraw-Hill.

Kok, S.; Singla, P.; Richardson, M.; and Domingos, P. 2005. The Alchemy system for statistical relational AI. Technical report, University of Washington.

Kuhlmann, G.; Stone, P.; Mooney, R.; and Shavlik, J. 2004. Guiding a reinforcement learner with natural language advice: Initial results in RoboCup soccer. In *2004 AAAI Workshop on Supervisory Control of Learning and Adaptive Systems*.

Lowd, D., and Domingos, P. 2007. Efficient weight learning for Markov logic networks. In *2007 Conference on Knowledge Discovery in Databases.*

Maclin, R., and Shavlik, J. 1996. Creating advice-taking reinforcement learners. *Machine Learning* 22:251–281.

Maclin, R.; Shavlik, J.; Torrey, L.; Walker, T.; and Wild, E. 2005a. Giving advice about preferred actions to reinforcement learners via knowledge-based kernel regression. In *2005 AAAI Conference on Artificial Intelligence*.

Maclin, R.; Shavlik, J.; Torrey, L.; and Walker, T. 2005b. Knowledge-based support vector regression for reinforcement learning. In *2005 IJCAI Workshop on Reasoning, Representation, and Learning in Computer Games*.

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

Richardson, M., and Domingos, P. 2005. Markov logic networks. *Machine Learning* 62:107–136.

Srinivasan, A. 2001. The Aleph manual (available online).

Stone, P., and Sutton, R. 2001. Scaling reinforcement learning toward RoboCup soccer. In *2001 International Conference on Machine Learning*.

Sutton, R., and Barto, A. 1998. *Reinforcement Learning: An Introduction*. MIT Press.

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

Taylor, M., and Stone, P. 2007. Cross-domain transfer for reinforcement learning. In *2007 International Conference on Machine Learning*.

Taylor, M.; Stone, P.; and Liu, Y. 2005. Value functions for RL-based behavior transfer: A comparative study. In *2005 AAAI Conference on Artificial Intelligence*.

Torrey, L.; Shavlik, J.; Walker, T.; and Maclin, R. 2006a. Relational skill transfer via advice taking. In *2006 ICML Workshop on Structural Knowledge Transfer for Machine Learning*.

Torrey, L.; Shavlik, J.; Walker, T.; and Maclin, R. 2006b. Skill acquisition via transfer learning and advice taking. In *2006 European Conference on Machine Learning*.

Torrey, L.; Shavlik, J.; Walker, T.; and Maclin, R. 2007. Relational macros for transfer in reinforcement learning. In *2007 International Conference on Inductive Logic Programming*.

Torrey, L.; Shavlik, J.; Natarajan, S.; Kuppili, P.; and Walker, T. 2008. Transfer in reinforcement learning via Markov logic networks. In *2008 AAAI Workshop on Transfer Learning for Complex Tasks*.

Watkins, C. 1989. *Learning from delayed rewards*. Ph.D. Dissertation, University of Cambridge.