Transfer Learning via Advice Taking

Lisa Torrey, Jude Shavlik, Trevor Walker and Richard Maclin

Abstract The goal of transfer learning is to speed up learning in a new task by transferring knowledge from one or more related source tasks. We describe a transfer method in which a reinforcement learner analyzes its experience in the source task and learns rules to use as advice in the target task. The rules, which are learned via inductive logic programming, describe the conditions under which an action is successful in the source task. The advice-taking algorithm used in the target task allows a reinforcement learner to benefit from rules even if they are imperfect. A human-provided mapping describes the alignment between the source and target tasks, and may also include advice about the differences between them. Using three tasks in the RoboCup simulated soccer domain, we demonstrate that this transfer method can speed up reinforcement learning substantially.

1 Introduction

Machine learning tasks are often addressed independently, under the implicit assumption that each new task has no exploitable relation to the tasks that came before. *Transfer learning* is a machine learning paradigm that rejects this assumption

Lisa Torrey
University of Wisconsin, Madison WI 53706, USA e-mail: ltorrey@cs.wisc.edu

Jude Shavlik
University of Wisconsin, Madison WI 53706, USA e-mail: shavlik@cs.wisc.edu

Trevor Walker
University of Wisconsin, Madison WI 53706, USA e-mail: twalker@cs.wisc.edu

Richard Maclin
University of Minnesota, Duluth, MN 55812, USA e-mail: rmaclin@gmail.com

Appears in *Recent Advances in Machine Learning*, dedicated to the memory of Ryszard S. Michalski, published in the Springer Studies in Computational Intelligence, edited by J. Koronacki, S. Wirzchon, Z. Ras and J. Kacprzyk, 2009.

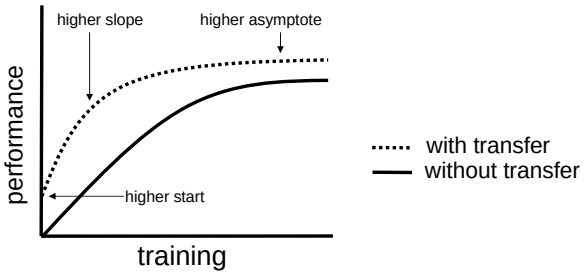

Fig. 1 Three ways in which transfer might improve reinforcement learning.

and uses known relationships between tasks to improve learning. The goal of transfer is to improve learning in a *target task* by transferring knowledge from a related *source task*.

One context in which transfer learning can be particularly useful is *reinforcement learning* (RL), where an agent learns to take actions in an environment to receive rewards [26]. Complex RL tasks can require very long training times. However, when learning a new task in the same domain as previously learned tasks, there are opportunities for reducing the training times through transfer.

There are three common measures by which transfer might improve learning in RL. First is the initial performance achievable in the target task using only the transferred knowledge, before any further learning is done, compared to the initial performance of an ignorant agent. Second is the amount of time it takes to fully learn the target task given the transferred knowledge compared to the amount of time to learn it from scratch. Third is the final performance level achievable in the target task compared to the final level without transfer. Figure 1 illustrates these three measures.

Our transfer method learns *skills* from a source task that may be useful in a target task. Skills are rules in first-order logic that describe when an action should be successful. For example, suppose an RL soccer player has learned, in a source task, to pass to its teammates in a way that keeps the ball from falling into the opponents' possession. In the target task, suppose it must learn to work with teammates to score goals against opponents. If this player could remember its passing skills from the source task, it should master the target task more quickly.

Even when RL tasks have shared actions, transfer between them is a difficult problem because differences in reward structures create differences in the results of actions. For example, the passing skill in the source task above is incomplete for the target task – in the target, unlike the source, passing needs to cause progress toward the goal in addition to maintaining ball possession. This indicates that RL agents using transferred information must continue to learn, filling in gaps left by transfer. Since transfer might also produce partially irrelevant or incorrect skills, RL agents must also be able to modify or ignore transferred information that is imperfect. Our transfer method allows this by applying skills as *advice*, with a learning algorithm that treats rules as soft constraints.

We require a human observer to provide a *mapping* between the source and target task. A mapping describes the structural similarities between the tasks, such as correspondences between player objects in the example above. It might also include simple advice that reflects the differences between the tasks. In our example, additional advice like "prefer passing toward the goal" and "shoot when close to the goal" would be helpful.

Our chapter's presence in this memorial volume is due to the way that our work touches on several topics of interest to Professor Ryszard Michalski. He contributed significantly to the area of rule learning in first-order logic [14], which we use to learn skills for transfer. He also did important work involving expert advice [2], which has connections to our advice-taking methods, and analogical learning [15], which is closely related to transfer learning.

The rest of the chapter is organized as follows. Section 2 provides background information on RL: an overview, and a description of our standard RL and advice-taking RL implementations. Section 3 presents RoboCup simulated soccer and explains how we learn tasks in the domain with RL. Section 4 provides background information on inductive logic programming, which is the machine-learning technique we use to learn skills. Section 5 then describes our transfer method, with experimental results in Section 6. Section 7 surveys some related work, and Section 8 reflects on some interesting issues that our work raises.

2 Background on Reinforcement Learning

A reinforcement learning agent operates in a episodic sequential-control environment. It senses the *state* of the environment and performs *actions* that change the state and also trigger *rewards*. Its objective is to learn a policy for acting in order to maximize its cumulative reward during an episode. This involves solving a temporal credit-assignment problem, since an entire sequence of actions may be responsible for a single reward received at the end of the sequence.

A typical RL agent behaves according to the diagram in Figure 2. At time step t, it observes the current state s_t and consults its current policy π to choose an action, $\pi(s_t) = a_t$. After taking the action, it receives a reward r_t and observes the new state s_{t+1}, and it uses that information to update its policy before repeating the cycle. Often RL consists of a sequence of *episodes*, which end whenever the agent reaches one of a set of ending states (e.g. the end of a game).

Formally, a reinforcement learning domain has two underlying functions that determine immediate rewards and the state transitions. The reward function $r(s,a)$ gives the reward for taking action a in state s, and the transition function $\delta(s,a)$ gives the next state the agent enters after taking action a in state s. If these functions are known, the optimal policy $\pi^\star$ can be calculated directly by maximizing the *value function* at every state. The value function $V_\pi(s)$ gives the discounted cumulative reward achieved by policy π starting in state s (see Equation 1).

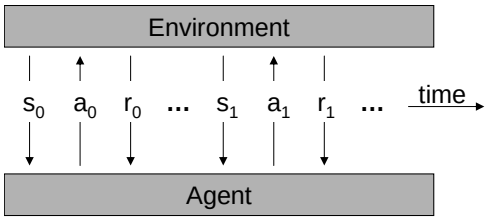

Fig. 2 A reinforcement learning agent interacts with its environment: it receives information about its state (s), chooses an action to take (a), receives a reward (r), and then repeats.

$$V_\pi(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... \tag{1}$$

The discount factor $\gamma \in [0,1]$. Setting $\gamma < 1$ gives later rewards less impact on the value function than earlier rewards, which may be desirable for tasks without fixed lengths.

During learning, the agent must balance between *exploiting* the current policy (acting in areas that it knows to have high rewards) and *exploring* new areas to find higher rewards. A common solution is the ε-greedy method, in which the agent takes random exploratory actions a small fraction of the time ($\varepsilon << 1$), but usually takes the action recommended by the current policy.

Often the reward and transition functions are not known, and therefore the optimal policy cannot be calculated directly. In this situation, one applicable RL technique is *Q*-learning [36], which involves learning a *Q*-function instead of a value function. The *Q*-function, $Q(s,a)$, estimates the discounted cumulative reward starting in state s and taking action a and following the current policy thereafter. Given the optimal *Q*-function, the optimal policy is to take the action $argmax_a Q(s_t,a)$. RL agents in deterministic worlds can begin with an inaccurate *Q*-function and recursively update it after each step according to the rule in Equation 2.

$$Q(s_t,a_t) \longleftarrow r_t + \gamma \, max_a \, Q(s_{t+1},a) \tag{2}$$

In this equation, the current estimate of a *Q*-value on the right is used to produce a new estimate on the left. In the SARSA variant [26], the new estimate uses the actual a_{t+1} instead of the a with the highest *Q*-value in s_{t+1}; this takes the ε-greedy action selections into account. In non-deterministic worlds, a learning rate $\alpha \in (0,1]$ is required to form a weighted average between the old estimate and the new one. Equation 3 shows the SARSA update rule for non-deterministic worlds.

$$Q(s_t,a_t) \longleftarrow (1-\alpha) \, Q(s_t,a_t) + \alpha \, (r_t + \gamma \, Q(s_{t+1},a_{t+1})) \tag{3}$$

While these equations give update rules that look just one step ahead, it is possible to perform updates over multiple steps. In temporal-difference learning [25], agents can combine estimates over multiple lookahead distances.

When there are small finite numbers of states and actions, the *Q*-function can be represented in tabular form. However, some RL domains have states that are described by very large feature spaces, or even infinite ones when continuous-valued

features are present, making a tabular representation infeasible. A solution is to use a function approximator to represent the Q-function (e.g., a neural network). Function approximation has the additional benefit of providing generalization across states; that is, changes to the Q-value of one state affect the Q-values of similar states.

Under certain conditions, Q-learning is guaranteed to converge to an accurate Q-function [37]. Although these conditions are typically violated (by using function approximation, for example) the method can still produce successful learning. For further information on reinforcement learning, there are more detailed introductions by Mitchell [16] and Sutton and Barto [26].

2.1 Performing RL with Support Vector Regression

Our implementation is a form of Q-learning called SARSA(λ), which is the SARSA variant combined with temporal-difference learning. 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 (SVR). 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.

Most Q-learning implementations make incremental updates to the Q-functions after each step the agent takes. However, completely re-solving the SVR optimization problem after each data point would be too computationally intensive. Instead, our agents perform batches of 25 full episodes at a time and re-solve the optimization problem after each batch.

Formally, for each action the agent finds an optimal weight vector w that has one 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.

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. When A becomes too large for efficient solving, we begin to discard episodes randomly such that the probability of discarding an episode increases with the age of the episode. 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 4.

$$Aw + b\overrightarrow{e} = y \tag{4}$$

where $\vec{e}$ denotes a vector of ones (we omit this for simplicity from now on).

Our matrix A contains 75% exploitation examples, in which the action is the one recommended by the current policy, and 25% exploration examples, in which the action is off-policy. We do this so that bad moves are not forgotten, as they could be if we used almost entirely exploitation examples. When there are not enough exploration examples, we create synthetic ones by randomly choosing exploitation steps and using the current model to score unselected actions for those steps.

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. The resulting minimization problem is

$$\begin{array}{l} \min\limits_{(w,b,s)} \; ||w||_1 + \nu|b| + C||s||_1 \\ s.t. \quad -s \leq Aw + b - y \leq s. \end{array} \tag{5}$$

where $|\cdot|$ denotes an absolute value, $||\cdot||_1$ denotes the one-norm (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. We let C decay exponentially over time so that solutions may be more complex later in the learning curve.

Several other parameters in our system also decay exponentially over time: the temporal-difference parameter λ, so that earlier episodes combine more lookahead distances than later ones; the learning rate α, so that earlier episodes tend to produce larger Q-value updates than later ones; and the exploration rate ε, so that agents explore less later in the learning curve.

2.2 Performing Advice Taking in RL

Advice taking is learning with additional knowledge that may be imperfect. It attempts to take advantage of this knowledge to improve learning, but avoids trusting it completely. Advice often comes from humans, but in our work it also comes from automated analysis of successful behavior in a source task.

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

IF an opponent is near me AND
a teammate is open
THEN *pass* has a high Q-value

In this example, there are two conditions describing the state of the agent's environment: an opponent is nearby and there is an unblocked path to a teammate. These form the IF portion of the rule. The THEN portion gives a constraint on the

Q-function that the advice indicates should hold when the environment matches the conditions.

In our advice-taking system, an agent can follow advice, only follow it approximately (which is like refining it), or ignore it altogether. We extend the support-vector regression technique described in Section 2.1 to accomplish this. Recall that Equation 5 describes the optimization problem for learning the weights that determine an action's Q-function. We incorporate advice into this optimization problem using a method called Knowledge-Based Kernel Regression (KBKR), designed by Mangasarian et al. [12] and applied to reinforcement learning by Maclin et al. [8].

An advice rule creates new constraints on the problem solution in addition to the constraints from the training data. In particular, since we use an extension of KBKR called Preference-KBKR [9], our advice rules give conditions under which one action is preferred over another action. Our advice therefore takes the following form:

$$Bx \leq d \Longrightarrow Q_p(x) - Q_n(x) \geq \beta, \tag{6}$$

This can be read as:

> If the current state 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 5, we allow advice to be followed only partially. To do so, we introduce slack variables z and penalty parameters μ for trading off the impact of the advice with the impact of the training examples. Over time, we decay μ so that advice has less impact as the learner gains more experience.

The new optimization problem [9] 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 solve the resulting linear program. We do not show the entire formalization here, but it minimizes the following quantity:

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

We have also developed a variant of Preference-KBKR called ExtenKBKR [10] that incorporates advice in a way that allows for faster problem-solving. We will not present this variant in detail here, but we do use it for transfer when there is more advice than Preference-KBKR can efficiently handle.

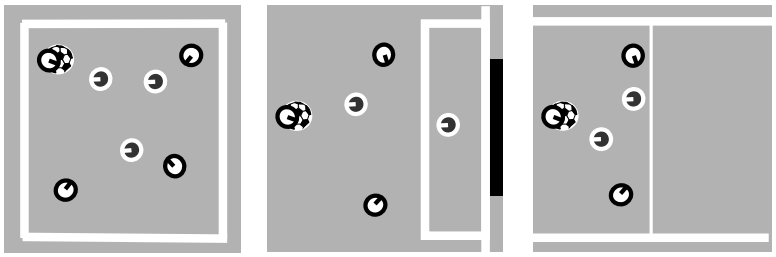

Fig. 3 Snapshots of RoboCup soccer tasks. In KeepAway, the keepers pass the ball around and keep it away from the takers. In BreakAway, the attackers attempt to score a goal against the defenders. In MoveDownfield, the attackers attempt to move the ball toward the defenders' side.

3 RoboCup: A Challenging Reinforcement Learning Domain

One motivating domain for transfer in reinforcement learning is RoboCup simulated soccer. The RoboCup project [17] has the overall goal of producing robotic soccer teams that compete on the human level, but it also has a software simulator for research purposes. Stone and Sutton [24] introduced RoboCup as an RL domain that is challenging because of its large, continuous state space and nondeterministic action effects.

Since the full game of soccer is quite complex, researchers have developed several smaller games in the RoboCup domain (see Figure 3). These are inherently multi-agent games, but a standard simplification is to have only one agent (the one in possession of the soccer ball) learning at a time using a shared model built with data combined from all the players on its team.

The first RoboCup task we use is *M*-on-*N* KeepAway [24], in which the objective of the *M* reinforcement learners called *keepers* is to keep the ball away from *N* hand-coded players called *takers*. The keeper with the ball may choose either to hold it or to pass it to a teammate. Keepers without the ball follow a hand-coded strategy to receive passes. The game ends when an opponent takes the ball or when the ball goes out of bounds. The learners receive a +1 reward for each time step their team keeps the ball.

Our KeepAway state representation is the one designed by Stone and Sutton [24]. The features are listed in Table 1. The keepers are ordered by their distance to the learner *k0*, as are the takers.

Note that we present these features as predicates in first-order logic. Variables are capitalized and typed (*Player*, *Keeper*, etc.) and constants are uncapitalized. For simplicity we indicate types by variable names, leaving out implied terms like *player(Player)*, *keeper(Keeper)*, etc. Since we are not using fully relational reinforcement learning, the predicates are actually grounded and used as propositional features during learning. However, since we transfer relational information, we represent them in a relational form here.

Table 1 Feature spaces for RoboCup tasks. The functions *minDistTaker(Keeper)* and *minAngleTaker(Keeper)* evaluate to the player objects *t0*, *t1*, etc. that are closest in distance and angle respectively to the given Keeper object. Similarly, the functions *minDistDefender(Attacker)* and *minAngleDefender(Attacker)* evaluate to the player objects *d0*, *d1*, etc.

KeepAway features	
distBetween(k0, Player)	Player $\in$ {k1, k2, ...} $\cup$ {t0, t1, ...}
distBetween(Keeper, minDistTaker(Keeper))	Keeper $\in$ {k1, k2, ...}
angleDefinedBy(Keeper, k0, minAngleTaker(Keeper))	Keeper $\in$ {k1, k2, ...}
distBetween(Player, fieldCenter)	Player $\in$ {k0, k1, ...} $\cup$ {t0, t1, ...}
MoveDownfield features	
distBetween(a0, Player)	Player $\in$ {a1, a2, ...} $\cup$ {d0, d1, ...}
distBetween(Attacker, minDistDefender(Attacker))	Attacker $\in$ {a1, a2, ...}
angleDefinedBy(Attacker, a0, minAngleDefender(Attacker))	Attacker $\in$ {a1, a2, ...}
distToRightEdge(Attacker)	Attacker $\in$ {a0, a1, ...}
timeLeft	
BreakAway features	
distBetween(a0, Player)	Player $\in$ {a1, a2, ...} $\cup$ {d0, d1, ...}
distBetween(Attacker, minDistDefender(Attacker))	Attacker $\in$ {a1, a2, ...}
angleDefinedBy(Attacker, a0, minAngleDefender(Attacker))	Attacker $\in$ {a1, a2, ...}
distBetween(Attacker, goalPart)	Attacker $\in$ {a0, a1, ...}
distBetween(Attacker, goalie)	Attacker $\in$ {a0, a1, ...}
angleDefinedBy(Attacker, a0, goalie)	Attacker $\in$ {a1, a2, ...}
angleDefinedBy(GoalPart, a0, goalie)	GoalPart $\in$ {right, left, center}
angleDefinedBy(topRightCorner, goalCenter, a0)	
timeLeft	

A second RoboCup task is *M*-on-*N* MoveDownfield, where the objective of the *M* reinforcement learners called *attackers* is to move across a line on the opposing team's side of the field while maintaining possession of the ball. The attacker with the ball may choose to pass to a teammate or to move ahead, away, left, or right with respect to the opponent's goal. Attackers without the ball follow a hand-coded strategy to receive passes. The game ends when they cross the line, when an opponent takes the ball, when the ball goes out of bounds, or after a time limit of 25 seconds. The learners receive symmetrical positive and negative rewards for horizontal movement forward and backward.

Our MoveDownfield state representation is the one presented in Torrey et al. [32]. The features are listed in Table 1. The attackers are ordered by their distance to the learner *a0*, as are the defenders.

A third RoboCup task is *M*-on-*N* BreakAway, where the objective of the *M* attackers is to score a goal against $N-1$ hand-coded *defenders* and a hand-coded *goalie*. The attacker with the ball may choose to pass to a teammate, to move ahead,

away, left, or right with respect to the opponent's goal, or to shoot at the left, right, or center part of the goal. Attackers without the ball follow a hand-coded strategy to receive passes. The game ends when they score a goal, when an opponent takes the ball, when the ball goes out of bounds, or after a time limit of 10 seconds. The learners receive a +1 reward if they score a goal, and zero reward otherwise.

Our BreakAway state representation is the one presented in Torrey et al. [33]. The features are listed in Table 1. The attackers are ordered by their distance to the learner *a0*, as are the non-goalie defenders.

Our system discretizes each feature in these tasks into 32 tiles, each of which is associated with a Boolean feature. For example, the tile denoted by *distBetween(a0, a1)*$_{[10,20]}$ takes value 1 when *a1* is between 10 and 20 units away from *a0* and 0 otherwise. Stone and Sutton [24] found tiling to be important for timely learning in RoboCup.

The three RoboCup games have substantial differences in features, actions, and rewards. The goal, goalie, and shoot actions exist in BreakAway but not in the other two tasks. The move actions do not exist in KeepAway but do in the other two tasks. Rewards in KeepAway and MoveDownfield occur for incremental progress, but in BreakAway the reward is more sparse. These differences mean the solutions to the tasks may be quite different. However, some knowledge should clearly be transferable between them, since they share many features and some actions, such as the *pass* action. Furthermore, since these are difficult RL tasks, speeding up learning through transfer would be desirable.

4 Inductive Logic Programming

Inductive logic programming (ILP) is a technique for learning classifiers in first-order logic [16]. Our transfer algorithms uses ILP to extract knowledge from the source task. This section provides a brief introduction to ILP.

4.1 What ILP Learns

An ILP algorithm learns a set of first-order clauses, usually definite clauses. A definite clause has a *head*, which is a predicate that is implied to be true if the conjunction of predicates in the *body* is true. Predicates describe relationships between objects in the world, referring to objects either as constants (lower-case) or variables (upper-case). In Prolog notation, the head and body are separated by the symbol :- denoting implication, and commas denoting *and* separate the predicates in the body, as in the rest of this section.

As an example, consider applying ILP to learn a clause describing when an object in an agent's world is at the bottom of a stack of objects. The world always contains the object *floor*, and may contain any number of additional objects. The

configuration of the world is described by predicates *stackedOn(Obj1, Obj2)*, where *Obj1* and *Obj2* are variables that can be instantiated by the objects, such as:

```
stackedOn(chair, floor).
stackedOn(desk, floor).
stackedOn(book, desk).
```

Suppose we want the ILP algorithm to learn a clause that implies *isBottomOfStack(Obj)* is true when *Obj = desk* but not when *Obj* $\in$ {*floor, chair, book*}. Given those positive and negative examples, it would learn the following clause:

```
isBottomOfStack(Obj) :-
    stackedOn(Obj, floor),
    stackedOn(OtherObj, Obj).
```

That is, an object is at the bottom of the stack if it is on the floor and there exists another object on top of it. On its way to discovering the correct clause, the ILP algorithm would probably evaluate the following clause:

```
isBottomOfStack(Obj) :-
    stackedOn(Obj, floor).
```

This clause correctly classifies 3 of the 4 objects in the world, but incorrectly classifies *chair* as positive. In domains with noise, a partially correct clause like this might be optimal, though in this case the concept can be learned exactly.

Note that the clause must be first-order to describe the concept exactly: it must include the variables *Obj* and *OtherObj*. First-order logic can posit the existence of an object and then refer to properties of that object. Most machine learning algorithms use *propositional* logic, which does not include variables, but ILP is able to use a more powerful and natural type of reasoning.

In many domains, the true concept is disjunctive, meaning that multiple clauses are necessary to describe the concept fully. ILP algorithms therefore typically attempt to learn a set of clauses rather than just one. The entire set of clauses is called a *theory*.

4.2 How ILP Learns

There are several types of algorithms for producing a set of first-order clauses, including Michalski's AQ algorithm [14]. This section focuses on the Aleph system [23], which we use in our experiments.

Aleph constructs a ruleset through *sequential covering*. It performs a search for the rule that best classifies the positive and negative examples (according to a user-specified scoring function), adds that rule to the theory, and then removes the positive examples covered by that rule and repeats the process on the remaining examples.

The default procedure Aleph uses in each iteration is a heuristic search. It randomly chooses a positive example as the *seed* for its search for a single rule. Then

it lists all the predicates in the world that are true for the seed. This list is called the *bottom clause*, and it is typically too specific, since it describes a single example in great detail. Aleph conducts a search to find a more general clause (a variablized subset of the predicates in the bottom clause) that maximizes the scoring function. The search process is top-down, meaning that it begins with an empty rule and adds predicates one by one to greedily maximize a scoring function.

Our rule-scoring function is the F(1) measure, which relies on the concepts of *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 F(1) measure combines the two:

$$F(1) = \frac{2 * Precision * Recall}{Precision + Recall}$$

An alternative Aleph procedure that we also use is a randomized search [34]. This also uses a seed example and generates a bottom clause, but it begins by randomly drawing a legal clause of length N from the bottom clause. It then makes local moves by adding and removing literals from the clause. After M local moves, and possibly K repeats of the entire process, it returns the highest-scoring rule encountered.

5 Skill Transfer in RL via Advice Taking

Our method for transfer in reinforcement learning, called *skill transfer*, begins by analyzing games played by a successful source-task agent. Using the ILP algorithm from Section 4.2, it learns first-order rules that describe *skills*. We define a skill as a rule that describes the circumstances under which an action is likely to be successful [32]. Our method then uses a human-provided mapping between the tasks to translate skills into a form usable in the target task. Finally, it applies the skills as advice in the target task, along with any additional human advice, using the KBKR algorithm from Section 2.2.

Figure 4 shows an example of skill transfer from KeepAway to BreakAway. In this example, KeepAway games provide training examples for the concept "states in which passing to a teammate is a good action," and ILP learns a rule representing the *pass* skill, which is mapped into advice for BreakAway.

We learn first-order rules because they can be more general than propositional rules, since they can contain variables. For example, the rule *pass(Teammate)* is likely to capture the essential elements of the passing skill better than rules for passing to specific teammates. We expect these common skill elements to transfer better to new tasks.

In a first-order representation, corresponding feature and action predicates can ideally be made identical throughout the domain so that there is no need to map them. However, 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).

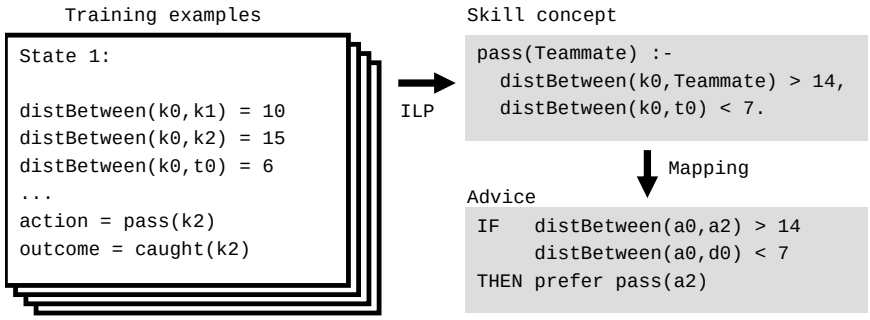

Fig. 4 Example showing how we transfer skills. We provide positive and negative source-task examples of *pass* actions to ILP, which learns a rule describing the *pass* skill, and we apply a mapping to produce target-task advice.

The actions in the two tasks need not have one-to-one correspondences. If an action in the source does not exist in the target, we do not attempt to transfer a skill for it. The feature sets also do not need to have one-to-one correspondences, because the ILP search algorithm can limit its search space to only those feature predicates that are present in the target task. We therefore allow only feature predicates that exist in the target task to appear in advice rules. This forces the algorithm to find skill definitions that are applicable to the target task.

5.1 Learning Skills in a Source Task

For each action, we conduct a search with ILP for the rule with the highest F(1) score. To produce datasets for this search, we examine states from games in the source task and select positive and negative examples. Not all states should be used as training examples; some are not unambiguously positive or negative and should be left out of the datasets. These states can be detected by looking at their *Q*-values, as described below. Figure 5 summarizes the overall process with an example from RoboCup.

In a good positive example, several conditions should be met: the skill is performed, the desired outcome occurs (e.g. a pass reaches its intended recipient), the expected *Q*-value (using the most recent *Q*-function) is above the 10th percentile in the training set and is at least 1.05 times the predicted *Q*-values of all other actions. The purpose of these conditions is to remove ambiguous examples in which several actions may be good or no actions seem good.

There are two types of good negative examples. These conditions describe one type: some other action is performed, that action's *Q*-value is above the 10th percentile in the training set, and the *Q*-value of the skill being learned is at most 0.95 times that *Q*-value and below the 50th percentile in the training set. These conditions also remove ambiguous examples. The second type of good negative example

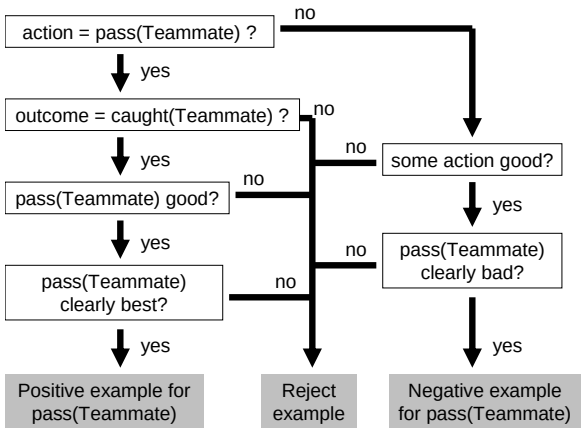

Fig. 5 Example of how our algorithm selects training examples for skills.

includes states in which the skill being learned was taken but the desired outcome did not occur.

To make the search space finite, it is necessary to replace continuous features (like distances and angles) with finite sets of discrete features. For example, the rule in Figure 4 contains the Boolean constraint *distBetween(k0,t0)* < 7, derived from the continuous distance feature. Our algorithm finds the 25 thresholds with the highest information gain and allows the intervals above and below those thresholds to appear as constraints in rules. Furthermore, we allow up to 7 constraints in each rule. We found these parameters to produce reasonable running times for RoboCup, but they would need to be adjusted appropriately for other domains.

5.2 Mapping Skills for a Target Task

To convert a skill into transfer advice, we need to apply an object mapping and propositionalize the rule. Propositionalizing is necessary because our KBKR advice-taking algorithm only works with propositional advice. This automated process preserves the meaning of the first-order rules without losing any information, but there are several technical details involved.

First we instantiate skills like *pass(Teammate)* for the target task. For 3-on-2 BreakAway, this would produce two rules, *pass(a1)* and *pass(a2)*. Next we deal with any other conditions in the rule body that contain variables. For example, a rule might have this condition:

$$10 < \text{distBetween(a0, Attacker)} < 20$$

This is effectively a disjunction of conditions: either the distance to *a1* or the distance to *a2* is in the interval $[10, 20]$. Since disjunctions are not part of the advice language, we use tile features to represent them. Recall that each feature range is

divided into Boolean tiles that take the value 1 when the feature value falls into their interval and 0 otherwise. This disjunction is satisfied if at least one of several tiles is active; for 3-on-2 BreakAway this is:

$$\text{distBetween(a0, a1)}_{[10,20]} + \text{distBetween(a0, a2)}_{[10,20]} \geq 1$$

If these exact tile boundaries do not exist in the target task, we add new tile boundaries to the feature space. Thus transfer advice can be expressed exactly even though the target-task feature space is unknown at the time the source task is learned.

It is possible for multiple conditions in a rule to refer to the same variable. For example:

$$\begin{aligned}&\text{distBetween(a0, Attacker)} > 15,\\ &\text{angleDefinedBy(Attacker, a0, ClosestDefender)} > 25\end{aligned}$$

Here the variable *Attacker* represents the same object in both clauses, so the system cannot propositionalize the two clauses separately. Instead, it defines a new predicate that puts simultaneous constraints on both features:

```
newFeature(Attacker, ClosestDefender) :-
  Dist is distBetween(a0, Attacker),
  Ang is angleDefinedBy(Attacker, a0, ClosestDefender),
  Dist > 15, Ang > 25.
```

It then expresses the entire condition using the new feature; for 3-on-2 BreakAway this is:

$$\text{newFeature(a1, d0)} + \text{newFeature(a2, d0)} \geq 1$$

We add these new Boolean features to the target task. Thus skill transfer can actually enhance the feature space of the target task.

Each advice item produced from a skill says to prefer that skill over the other actions shared between the source and target task. We set the preference amount Δ to approximately 1% of the target task's Q-value range.

5.3 Adding Human Advice

Skill transfer produces a small number of simple, interpretable rules. This introduces the possibility of further user input in the transfer process. If users can understand the transfer advice, they may wish to add to it, either further specializing rules or writing their own rules for new, non-transferred skills in the target task. Our skill-transfer method therefore allows optional *user advice*.

For example, the passing skills transferred from KeepAway to BreakAway make no distinction between passing toward the goal and away from the goal. Since the new objective is to score goals, players should clearly prefer passing toward the goal. A user could provide this guidance by instructing the system to add a condition like this to the *pass(Teammate)* skill:

$$\text{distBetween(a0, goal)} - \text{distBetween(Teammate, goal)} \geq 1$$

Even more importantly, there are several actions in this transfer scenario that are new in the target task, such as *shoot* and *moveAhead*. We allow users to write simple rules to approximate skills like these, such as:

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

IF distBetween(a0, goalCenter) $>$ 10
THEN prefer moveAhead over moveAway and the shoot actions

The advice-taking framework is a natural and powerful way for users to provide information not only about the correspondences between tasks, but also about the differences between them.

6 Results

We performed experiments with skill transfer in many scenarios with RoboCup tasks. Some are *close transfer* scenarios, where the tasks are closely related: the target task is the same as the source task except each team has one more player. Others are *distant transfer* scenarios, where the tasks are more distantly related: from KeepAway to BreakAway and from MoveDownfield to BreakAway. With distant transfer we concentrate on moving from easier tasks to harder tasks.

For each task, we use an appropriate measure of performance to plot against the number of training games in a learning curve. In BreakAway, it is the probability that the agents will score a goal in a game. In MoveDownfield, it is the average distance traveled towards the right edge during a game. In KeepAway, it is the average length of a game.

Section 6.1 shows examples of rules our method learned in various source tasks. Section 6.2 shows learning curves in various target tasks with and without skill transfer.

6.1 Skills Learned

From 2-on-1 BreakAway, one rule our method learned for the *shoot* skill is:

shoot(GoalPart) :-
  distBetween(a0, goalCenter) $\geq$ 6,
  angleDefinedBy(GoalPart, a0, goalie) $\geq$ 52,
  distBetween(a0, oppositePart(GoalPart)) $\geq$ 6,
  angleDefinedBy(oppositePart(GoalPart), a0, goalie) $\leq$ 33,
  angleDefinedBy(goalCenter, a0, goalie) $\geq$ 28.

This rule requires a large open shot angle, a minimum distance to the goal, and angle constraints that restrict the goalie's position to a small area.

From 3-on-2 MoveDownfield, one rule our method learned for the *pass* skill is:

pass(Teammate) :-
  distBetween(a0, Teammate) $\geq$ 15,
  distBetween(a0, Teammate) $\leq$ 27,
  angleDefinedBy(Teammate, a0, minAngleDefender(Teammate)) $\geq$ 24,
  distToRightEdge(Teammate) $\leq$ 10,
  distBetween(a0, Opponent) $\geq$ 4.

This rule specifies an acceptable range for the distance to the receiving teammate and a minimum pass angle. It also requires that the teammate be close to the finish line on the field and that an opponent not be close enough to intercept.

From 3-on-2 KeepAway, one rule our method learned for the *pass* skill is:

pass(Teammate) :-
  distBetween(Teammate, fieldCenter) $\geq$ 6,
  distBetween(Teammate, minDistTaker(Teammate)) $\geq$ 8,
  angleDefinedBy(Teammate, a0, minAngleTaker(Teammate)) $\geq$ 41,
  angleDefinedBy(OtherTeammate, a0, minAngleTaker(OtherTeammate)) $\leq$ 23.

This rule specifies a minimum pass angle and an open distance around the receiving teammate. It also requires that the teammate not be too close to the center of the field and gives a maximum pass angle for the alternate teammate.

Some parts of these rules were unexpected, but make sense in hindsight. For example, the shoot rule specifies a minimum distance to the goal rather than a maximum distance. Presumably this is because large shot angles are only available at reasonable distances anyway. This shows the advantages that advice learned through transfer can have over human advice.

6.2 Learning Curves

Figures 6, 7, and 8 are learning curves from our transfer experiments. One curve in each figure is the average of 25 runs of standard reinforcement learning. The other curves are RL with skill transfer from various source tasks. For each transfer curve we average 5 transfer runs from 5 different source runs, for a total of 25 runs (this way, the results include both source and target variance). Because the variance is high, we smooth the *y*-value at each data point by averaging over the *y*-values of the last 250 games.

These figures show that skill transfer can have a large overall positive impact in both close-transfer and distant-transfer scenarios. The statistical results in Table 2 indicate that in most cases the difference (in area under the curve) is statistically significant.

We use appropriate subsets of the human-advice examples in Section 5.3 for all of our skill-transfer experiments. That is, from KeepAway to BreakAway we use all of it, from MoveDownfield to BreakAway we use only the parts advising *shoot*, and for close-transfer experiments we use none.

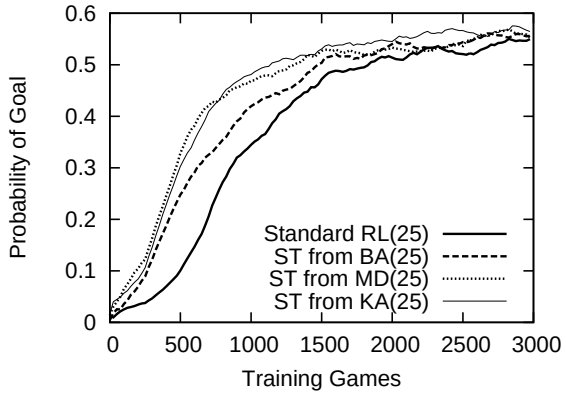

Fig. 6 Probability of scoring a goal while training in 3-on-2 BreakAway with standard RL and skill transfer (ST) from 2-on-1 BreakAway (BA), 3-on-2 MoveDownfield (MD) and 3-on-2 KeepAway (KA).

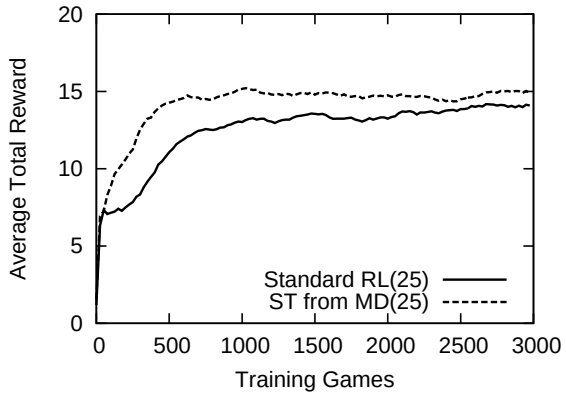

Fig. 7 Average total reward while training in 4-on-3 MoveDownfield with standard RL and skill transfer (ST) from 3-on-2 MoveDownfield (MD).

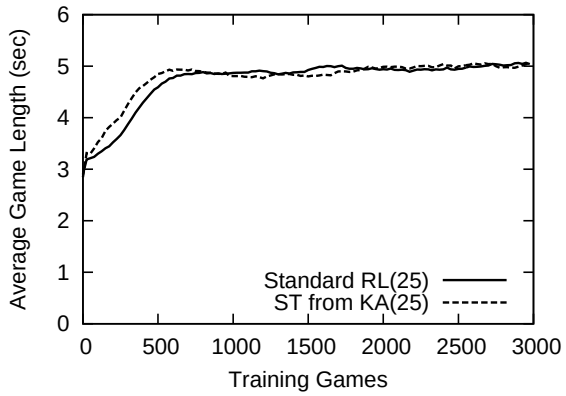

Fig. 8 Average game length while training in 4-on-3 KeepAway with standard RL and skill transfer (ST) from 3-on-2 KeepAway (KA).

Table 2 Statistical results from skill transfer (ST) experiments in BreakAway (BA), MoveDownfield (MD), and KeepAway (KA), comparing area under the curve to standard reinforcement learning (SRL).

Scenario	*Conclusion*	*p-value*	*95% confidence interval*
BA to BA	ST higher with 99% confidence	0.0003	63.75, 203.36
MD to BA	ST higher with 99% confidence	< 0.0001	153.63, 278.02
KA to BA	ST higher with 97% confidence	< 0.0001	176.42, 299.87
MD to MD	ST higher with 98% confidence	< 0.0001	3682.59, 6436.61
KA to KA	ST and SRL equivalent	0.1491	-114.32, 389.20

6.3 Further Experiments with Human Advice

To show the effect of adding human advice, we performed skill transfer without any (Figure 9). In the scenario shown, MoveDownfield to BreakAway, we compare learning curves for skill transfer with and without human advice. Our method still improves learning significantly when it includes no human advice about shooting, though the gain is smaller. The addition of our original human advice produces another significant gain.

To demonstrate that our method can cope with incorrect advice, we also performed skill transfer with intentionally bad human advice (Figure 10). In the scenario shown, KeepAway to BreakAway, we compare learning curves for skill transfer with our original human advice and with its opposite. In the bad advice the inequalities are reversed, so the rules instruct the learner to pass backwards, shoot when far away from the goal and at a narrow angle, and move when close to the goal. Our method no longer improves learning significantly with this bad advice,

but since the KBKR algorithm can learn to ignore it, learning is never impacted negatively.

The robustness indicated by these experiments means that users need not worry about providing perfect advice in order for the skill-transfer method to work. It also means that skill transfer can be applied to reasonably distant tasks, since the source-task skills need not be perfect for the target task. It can be expected that learning with skill transfer will perform no worse than standard reinforcement learning, and it may perform significantly better.

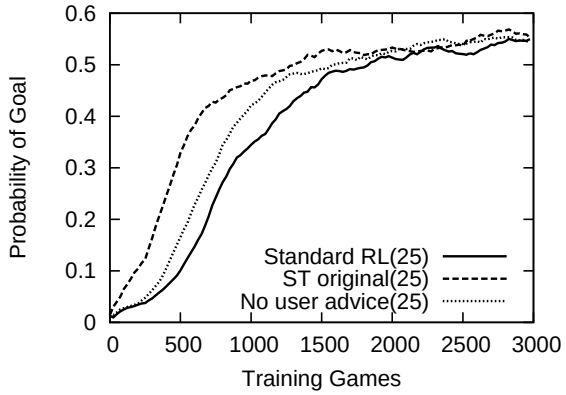

Fig. 9 Probability of scoring a goal while training in 3-on-2 BreakAway with standard RL and skill transfer (ST) from 3-on-2 MoveDownfield, with and without the original human advice.

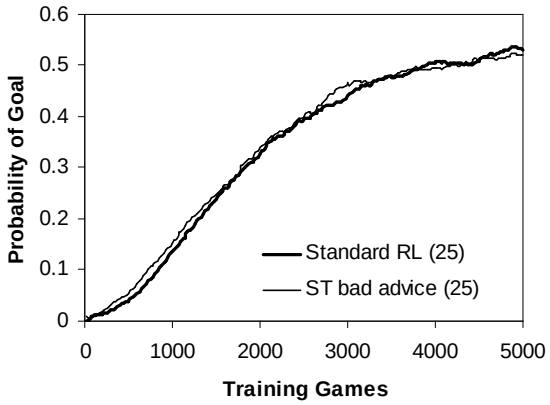

Fig. 10 Probability of scoring a goal while training in 3-on-2 BreakAway with standard RL skill transfer (ST) from 3-on-2 KeepAway that includes intentionally bad human advice.

7 Related Work

There is a strong body of related work on transfer learning in RL. We divide RL transfer into five broad categories that represent progressively larger changes to existing RL algorithms.

7.1 Starting-point methods

Since all RL methods begin with an initial solution and then update it through experience, one straightforward type of transfer in RL is to set the initial solution in a target task based on knowledge from a source task. Compared to the arbitrary setting that RL algorithms usually use at first, these *starting-point methods* can begin the RL process at a point much closer to a good target-task solution. There are variations on how to use the source-task knowledge to set the initial solution, but in general the RL algorithm in the target task is unchanged.

Taylor et al. [30] use a starting-point method for transfer in temporal-difference RL. To perform transfer, they copy the final value function of the source task and use it as the initial one for the target task. As many transfer approaches do, this requires a mapping of features and actions between the tasks, and they provide a mapping based on their domain knowledge.

Tanaka and Yamamura [27] use a similar approach in temporal-difference learning without function approximation, where value functions are simply represented by tables. This greater simplicity allows them to combine knowledge from several source tasks: they initialize the value table of the target task to the average of tables from several prior tasks. Furthermore, they use the standard deviations from prior tasks to determine priorities between temporal-difference backups.

Approaching temporal-difference RL as a batch problem instead of an incremental one allows for different kinds of starting-point transfer methods. In batch RL, the agent interacts with the environment for more than one step or episode at a time before updating its solution. Lazaric et al. [7] perform transfer in this setting by finding source-task samples that are similar to the target task and adding them to the normal target-task samples in each batch, thus increasing the available data early on. The early solutions are almost entirely based on source-task knowledge, but the impact decreases in later batches as more target-task data becomes available.

Moving away from temporal-difference RL, starting-point methods can take even more forms. In a model-learning Bayesian RL algorithm, Wilson et al. [38] perform transfer by treating the distribution of previous MDPs as a prior for the current MDP. In a policy-search genetic algorithm, Taylor et al. [31] transfer a population of policies from a source task to serve as the initial population for a target task.

7.2 Imitation methods

Another class of RL transfer methods involves applying the source-task policy to choose some actions while learning the target task. While they make no direct changes to the target-task solution the way that starting-point methods do, these *imitation methods* affect the developing solution by producing different function or policy updates. Compared to the random exploration that RL algorithms typically do, decisions based on a source-task policy can lead the agent more quickly to promising areas of the environment. There are variations in how the source-task policy is represented and in how heavily it is used in the target-task RL algorithm.

One method is to follow a source-task policy only during exploration steps of the target task, when the agent would otherwise be taking a random action. Madden and Howley [11] use this approach in tabular *Q*-learning. They represent a source-task policy as a set of rules in propositional logic and choose actions based on those rules during exploration steps.

Fernandez and Veloso [5] instead give the agent a three-way choice between exploiting the current target-task policy, exploiting a past policy, and exploring randomly. They introduce a second parameter, in addition to the ϵ of ϵ-greedy exploration, to determine the probability of making each choice.

7.3 Hierarchical methods

A third class of RL transfer includes *hierarchical methods*. These view the source as a subtask of the target, and use the solution to the source as a building block for learning the target. Methods in this class have strong connections to the area of hierarchical RL, in which a complex task is learned in pieces through division into a hierarchy of subtasks.

An early approach of this type is to compose several source-task solutions to form a target-task solution, as is done by Singh [22]. He addresses a scenario in which complex tasks are temporal concatenations of simple ones, so that a target task can be solved by a composition of several smaller solutions.

Mehta et al. [13] have a transfer method that works directly within the hierarchical RL framework. They learn a task hierarchy by observing successful behavior in a source task, and then use it to apply the MaxQ hierarchical RL algorithm [4] in the target task. This removes the burden of designing a task hierarchy through transfer.

Other approaches operate within the framework of *options*, which is a term for temporally-extended actions in RL [18]. An option typically consists of a starting condition, an ending condition, and an internal policy for choosing lower-level actions. An RL agent treats each option as an additional action along with the original lower-level ones.

In some scenarios it may be useful to have the entire source-task policy as an option in the target task, as Croonenborghs et al. [3] do. They learn a relational decision tree to represent the source-task policy and allow the target-task learner to

execute it as an option. Another possibility is to learn smaller options, either during or after the process of learning the source task, and offer them to the target. Asadi and Huber [1] do this by finding frequently-visited states in the source task to serve as ending conditions for options.

7.4 Alteration methods

The next class of RL transfer methods involves altering the state space, action space, or reward function of the target task based on source-task knowledge. These *alteration methods* have some overlap with option-based transfer, which also changes the action space in the target task, but they include a wide range of other approaches as well.

One way to alter the target-task state space is to simplify it through state abstraction. Walsh et al. [35] do this by aggregating over comparable source-task states. They then use the aggregate states to learn the target task, which reduces the complexity significantly.

There are also approaches that expand the target-task state space instead of reducing it. Taylor and Stone [29] do this by adding a new state variable in the target task. They learn a decision list that represents the source-task policy and use its output as the new state variable.

While option-based transfer methods add to the target-task action space, there is also some work in decreasing the action space. Sherstov and Stone [21] do this by evaluating in the source task which of a large set of actions are most useful. They then consider only a smaller action set in the target task, which decreases the complexity of the value function significantly and also decreases the amount of exploration needed.

Reward shaping is a design technique in RL that aims to speed up learning by providing immediate rewards that are more indicative of cumulative rewards. Usually it requires human effort, as many aspects of RL task design do. Konidaris and Barto [6] do reward shaping automatically through transfer. They learn to predict rewards in the source task and use this information to create a shaped reward function in the target task.

7.5 New RL Algorithms for Transfer

A final class of RL transfer methods consists of entirely new RL algorithms. Rather than making small additions to an existing algorithm or making changes to the target task, these approaches address transfer as an inherent part of RL. They incorporate prior knowledge as an intrinsic part of the algorithm. Our KBKR algorithm falls into this category of methods [32].

Price and Boutilier [19] propose a temporal-difference algorithm in which value functions are influenced by observations of expert agents. They use a variant of the usual value-function update that includes an expert's experience, weighted by the agent's confidence in itself and in the expert. They also perform extra backups at states the expert visits to focus attention on those areas of the state space.

There are several algorithms for case-based RL that accomodate transfer. Sharma et al. [20] propose one in which *Q*-functions are estimated using a Gaussian kernel over stored cases in a library. Cases are added to the library from both the source and target tasks when their distance to their nearest neighbor is above a threshold. Taylor et al. [28] use source-task examples more selectively in their case-based RL algorithm. They use target-task cases to make decisions when there are enough, and only use source-task examples when insufficient target examples exist.

8 Conclusion

We have described a method for transferring knowledge in reinforcement learning that learns logical rules to represent skills and uses them as advice for a new task. This approach can provide significant benefits in target tasks, as evidenced by our results in a complex RL domain. Our work has connections to Professor Michalski's interests in rule learning, advice, and analogical reasoning. As Michalski did, we emphasize the value of logic as a means of representing knowledge. We believe that first-order logic is a powerful mechanism for transfer.

An inherent aspect of transfer learning is recognizing the correspondences between tasks. Knowledge from one task can only be applied to another if it is expressed in a way that the target-task agent understands. If the task representations are not identical, a *mapping* is needed to translate between task representations. We assume a human-provided mapping so far, but learning a mapping is also an interesting task.

If a transfer method actually decreases performance, then *negative transfer* has occurred. One of the major challenges in developing transfer methods is to produce positive transfer between appropriately related tasks while avoiding negative transfer between tasks that are less related. Ideally, a transfer method would produce positive transfer between appropriately related tasks while avoiding negative transfer when the tasks are not a good match. In practice, these goals are difficult to achieve simultaneously. Approaches that have safeguards to avoid negative transfer often produce a smaller effect from positive transfer due to their caution.

Another challenge that we have encountered in RL transfer learning is that differences in reward structures between the source and target task make it difficult to transfer even shared actions. Changing the game objective or adding a new action changes the meaning of a shared skill. This means that it is important to continue learning in the target task and to avoid relying on source-task skills too much. We have also addressed this issue through human guidance, by allowing additional advice that points out differences between tasks.

9 Acknowledgements

This chapter was written while the authors were partially supported by DARPA grants HR0011-07-C-0060 and FA8650-06-C-7606.

References

1. M. Asadi and M. Huber. Effective control knowledge transfer through learning skill and representation hierarchies. In *International Joint Conference on Artificial Intelligence*, Hyderabad, India, 2007.
2. E. Bloedorn, R. Michalski, and J. Wnek. Multistrategy constructive induction: AQ17-MCI. In *International Workshop on Multistrategy Learning*, 1993.
3. T. Croonenborghs, K. Driessens, and M. Bruynooghe. Learning relational skills for inductive transfer in relational reinforcement learning. In *International Conference on Inductive Logic Programming*, Corvallis, OR, 2007.
4. T. Dietterich. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000.
5. F. Fernandez and M. Veloso. Probabilistic policy reuse in a reinforcement learning agent. In *Conference on Autonomous Agents and Multi-Agent Systems*, Hakodate, Japan, 2006.
6. G. Konidaris and A. Barto. Autonomous shaping: Knowledge transfer in reinforcement learning. In *International Conference on Machine Learning*, Pittsburgh, PA, 2006.
7. A. Lazaric, M. Restelli, and A. Bonarini. Transfer of samples in batch reinforcement learning. In *International Conference on Machine Learning*, Helsinki, Finland, 2008.
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*, Edinburgh, Scotland, 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 Conference on Artificial Intelligence*, Pittsburgh, PA, 2005.
10. R. Maclin, J. Shavlik, T. Walker, and L. Torrey. A simple and effective method for incorporating advice into kernel methods. In *AAAI Conference on Artificial Intelligence*, Boston, MA, 2006.
11. M. Madden and T. Howley. Transfer of experience between reinforcement learning environments with progressive difficulty. *Artificial Intelligence Review*, 21:375–398, 2004.
12. O. Mangasarian, J. Shavlik, and E. Wild. Knowledge-based kernel approximation. *Journal of Machine Learning Research*, 5:1127–1141, 2004.
13. N. Mehta, S. Ray, P. Tadepalli, and T. Dietterich. Automatic discovery and transfer of MAXQ hierarchies. In *International Conference on Machine Learning*, Helsinki, Finland, 2008.
14. R. Michalski. A theory and methodology of inductive learning. *Artificial Intelligence*, 20(2):111–161, 1983.
15. R. Michalski. Toward a unified theory of learning: Multistrategy task-adaptive learning. In B.G. Buchanan and D.C. Wilkins, editors, *Readings in Knowledge Acquisition and Learning: Automating the Construction and Improvement of Expert Systems*. Morgan Kaufmann, 1993.
16. T. Mitchell. *Machine Learning*. McGraw-Hill, 1997.
17. 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.
18. T. Perkins and D. Precup. Using options for knowledge transfer in reinforcement learning. Technical Report UM-CS-1999-034, University of Massachusetts, Amherst, 1999.
19. B. Price and C. Boutilier. Implicit imitation in multiagent reinforcement learning. In *International Conference on Machine Learning*, Bled, Slovenia, 1999.

20. M. Sharma, M. Holmes, J. Santamaria, A. Irani, C. Isbell, and A. Ram. Transfer learning in real-time strategy games using hybrid CBR/RL. In *International Joint Conference on Artificial Intelligence*, Hyderabad, India, 2007.
21. A. Sherstov and P. Stone. Action-space knowledge transfer in MDPs: Formalism, suboptimality bounds, and algorithms. In *Conference on Learning Theory*, Bertinoro, Italy, 2005.
22. S. Singh. Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning*, 8(3-4):323–339, 1992.
23. A. Srinivasan. The Aleph manual, 2001.
24. P. Stone and R. Sutton. Scaling reinforcement learning toward RoboCup soccer. In *International Conference on Machine Learning*, Williamstown, MA, 2001.
25. R. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.
26. R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.
27. F. Tanaka and M. Yamamura. Multitask reinforcement learning on the distribution of MDPs. *Transactions of the Institute of Electrical Engineers of Japan*, 123(5):1004–1011, 2003.
28. M. Taylor, N. Jong, and P. Stone. Transferring instances for model-based reinforcement learning. In *European Conference on Machine Learning*, Antwerp, Belgium, 2008.
29. M. Taylor and P. Stone. Cross-domain transfer for reinforcement learning. In *International Conference on Machine Learning*, Corvallis, OR, 2007.
30. M. Taylor, P. Stone, and Y. Liu. Value functions for RL-based behavior transfer: A comparative study. In *AAAI Conference on Artificial Intelligence*, Pittsburgh, PA, 2005.
31. M. Taylor, S. Whiteson, and P. Stone. Transfer learning for policy search methods. In *ICML Workshop on Structural Knowledge Transfer for Machine Learning*, Pittsburgh, PA, 2006.
32. L. Torrey, J. Shavlik, T. Walker, and R. Maclin. Skill acquisition via transfer learning and advice taking. In *European Conference on Machine Learning*, Berlin, Germany, 2006.
33. L. Torrey, T. Walker, J. Shavlik, and R. Maclin. Using advice to transfer knowledge acquired in one reinforcement learning task to another. In *European Conference on Machine Learning*, Porto, Portugal, 2005.
34. F. Železný, A. Srinivasan, and D. Page.
35. T. Walsh, L. Li, and M. Littman. Transferring state abstractions between MDPs. In *ICML Workshop on Structural Knowledge Transfer for Machine Learning*, Pittsburgh, PA, 2006.
36. C. Watkins. *Learning from delayed rewards*. PhD thesis, University of Cambridge, 1989.
37. C. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8:279–292, 1992.
38. A. Wilson, A. Fern, S. Ray, and P. Tadepalli. Multi-task reinforcement learning: A hierarchical Bayesian approach. In *International Conference on Machine Learning*, Corvallis, OR, 2007.