Relational Transfer in Reinforcement Learning

Lisa Torrey (Thesis)

Technical Report #1657

May 2009

RELATIONAL TRANSFER IN REINFORCEMENT LEARNING

by

Lisa Torrey

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

May 2009

ACKNOWLEDGMENTS

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

Special thanks to my advisor, Jude Shavlik, to my collaborators Trevor Walker and Rich Maclin, to my committee members, and to the Machine Learning Group at the University of Wisconsin-Madison.

We offer a public distribution of our RoboCup players and server code. They are available at http://www.biostat.wisc.edu/∼ml-group/RoboCup.

DISCARD THIS PAGE

TABLE OF CONTENTS

Page

DISCARD THIS PAGE

LIST OF TABLES

Table Page

DISCARD THIS PAGE

LIST OF FIGURES

Figure Page

ABSTRACT

Transfer learning 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 us master the new task more quickly.

This principle can be applied to machine learning as well. Machine learning often addresses single learning tasks in isolation. Even though multiple related tasks may exist in a domain, many algorithms for machine learning have no way to utilize those relationships. Algorithms that allow successful transfer from one task (the source) to another task (the target) are necessary steps towards making machine learning as adaptable as human learning.

This thesis investigates transfer methods for reinforcement learning (RL), where an agent takes series of actions in an environment. RL often requires substantial amounts of nearly random exploration, particularly in the early stages of learning. The ability to transfer knowledge from previous tasks can therefore be an important asset for RL agents. Transfer from related source tasks can improve the low initial performance that is common in challenging target tasks.

I focus on transferring relational knowledge that guides action choices. Relational knowledge typically uses first-order logic to express information about relationships among objects. First-order logic, unlike propositional logic, can use variables that generalize over classes of objects. This greater generalization makes first-order logic more effective for transfer.

This thesis contributes six transfer algorithms in three categories: advice-based transfer, macro transfer, and MLN transfer. Advice-based transfer uses source-task knowledge to provide advice for a target-task learner, which can follow, refine, or ignore the advice according to its value.

Macro-transfer and MLN-transfer methods use source-task experience to demonstrate good behavior for a target-task learner.

I evaluate these transfer algorithms experimentally in the complex reinforcement-learning domain of RoboCup simulated soccer. All of my algorithms provide empirical benefits compared to non-transfer approaches, either by increasing initial performance or by enabling faster learning in the target task.

Chapter 1

Introduction

Transfer learning 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 us master the new task more quickly [27].

Human transfer learning is often studied in the context of education, where transfer could be seen as the ultimate goal. The hierarchical curricular structure of schools is based upon the belief that learning tasks can share common stimulus-response elements [99]. The focus on abstract problem-solving methods is based on the idea that learning tasks can share general underlying principles [9, 30].

Another area of human activity in which transfer learning is often studied is that of language, or more precisely, multilingualism. Knowledge of one language can affect learning in a second language, and vice versa [61, 111]. In both of these areas, transfer can be a powerful method of facilitating human learning.

This principle can be applied to *machine learning* as well. Machine learning often addresses single learning tasks in isolation. Even though multiple related tasks may exist in a domain, many algorithms for machine learning have no way to utilize those relationships. Algorithms that allow successful transfer are steps towards making machine learning as adaptable as human learning.

In human learning, the goal of transfer research is typically to determine what conditions facilitate transfer. Educators hope to activate and maximize the pre-existing mechanisms for transfer learning in students' brains. In machine learning, however, the goal of transfer research is to design effective mechanisms for transfer. This thesis contributes several transfer mechanisms for one type of machine learning.

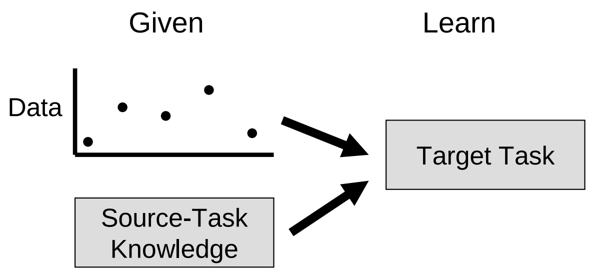

Figure 1.1: Transfer learning is machine learning with an additional source of information apart from the standard training data: knowledge from one or more related tasks.

1.1 Thesis Topic

Transfer in machine learning can be illustrated by Figure 1.1. A typical machine-learning problem can be characterized as:

GIVEN	Training data for task T
DO	Learn task T

A typical transfer-learning problem can be characterized as:

GIVEN	Training data for task T AND knowledge from related task(s) S
DO	Learn task T

Here S represents one or more *source tasks* that were previously learned, and T represents a new and related *target task*. The goal of transfer is to improve learning in a target task using knowledge acquired in source tasks.

Transfer is desirable in many types of machine learning. My work focuses on transfer in *reinforcement learning* (RL), where an agent takes series of actions in an environment [87]. RL often requires substantial amounts of nearly random exploration, particularly in the early stages of learning. The ability to transfer knowledge from previous tasks can therefore be an important asset for RL agents. Transfer can reduce the long initial period of low performance that is common in challenging tasks.

Many types of knowledge can be transferred between RL tasks. My work focuses on transferring *relational knowledge* that guides action choices. Relational knowledge typically uses *first-order logic* to express information about relationships between objects [68]. First-order logic, unlike propositional logic, can use variables that generalize over classes of objects. This greater generalization makes first-order logic more effective for transfer.

1.2 Thesis Statement

This thesis investigates the following claims:

> Transfer learning can improve learning in reinforcement learning tasks. Source-task knowledge can guide learners' action choices in related target tasks to produce better performance than random exploration. This guidance can allow learners to perform better while learning the target task than they would otherwise. Relational learning can provide effective and interpretable transfer knowledge for reinforcement learners.

1.3 Thesis Contributions

This thesis presents research on *relational transfer in reinforcement learning*. It contributes six major RL transfer algorithms, which are listed in Table 1.1.

The material is organized as follows. Background information needed for later chapters is in Chapter 2. Chapter 3 gives a survey of transfer learning on a wide scale, covering both inductive learning and reinforcement learning. Chapters 4, 5, and 6 present my original research. Conclusions and suggestions for future research are in Chapter 7. The Appendices contain additional information set aside from the chapters for the purposes of readability.

The first chapter of original research, Chapter 4, presents two algorithms for *advice-based transfer*. They express source-task knowledge as advice for the target-task learner, which uses an advice-taking RL algorithm. This approach can produce faster learning in the target task.

The second chapter of original research, Chapter 5, presents two algorithms for *macro-operator transfer*. They express source-task knowledge with relational finite-state machines, which the target-task learner uses to demonstrate good behavior. This approach can produce high initial performance in the target task.

Table 1.1: A list of major algorithms contributed by this thesis.

Transfer Algorithm 1:	Policy Transfer via Advice	Table 4.1
Transfer Algorithm 2:	Skill Transfer via Advice	Table 4.3
Transfer Algorithm 3:	Single-Macro Transfer via Demonstration	Table 5.1
Transfer Algorithm 4:	Multiple-Macro Transfer via Demonstration	Table 5.7
Transfer Algorithm 5:	MLN Q-Function Transfer	Table 6.1
Transfer Algorithm 6:	MLN Policy Transfer	Table 6.5

The third chapter of original research, Chapter 6, presents two algorithms for *transfer via Markov Logic Networks*. They express source-task knowledge with a statistical-relational model, which the target-task learner uses to evaluate actions. This approach also can produce high initial performance in the target task. I also show that Markov Logic Networks can improve performance in the source task from which they are learned.

I evaluate all of these transfer algorithms experimentally in a complex reinforcement-learning domain: RoboCup simulated soccer [59]. RoboCup is a much more complex domain than many typical testbeds for RL, which include maze worlds, games, and simple control problems such as balancing poles and accelerating cars up hills. Stone and Sutton [81] introduced RoboCup as a challenging RL domain due to its large, continuous state space and nondeterministic action effects. This complexity also makes it a challenging domain for transfer, which is important for realistic evaluation of my proposed methods.

Chapter 2

Background

This chapter provides background information that lays the framework for the rest of the dissertation. Section 2.1 reviews reinforcement learning (RL) and introduces the RL domain and the RL algorithm I use in experiments. Section 2.2 explains the way I report results and the statistical methods I use to compare algorithms. Section 2.3 reviews inductive logic programming, a method of learning relational concepts that I use extensively for relational transfer. Section 2.4 reviews Markov Logic Networks, which I use in several transfer approaches as a more detailed way to represent relational concepts.

2.1 Reinforcement Learning

In reinforcement learning [87], an agent operates in an 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 immediate reward.

A typical RL agent behaves according to the diagram in Figure 2.1. 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.

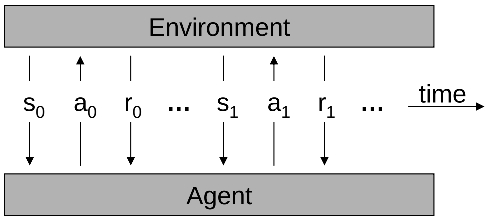

Figure 2.1: A reinforcement learning agent interacts with its environment: it receives information about its state (s), chooses an action to take (a), receives a reward (r) and a new state, learns from that information, and so on.

Formally, a reinforcement learning domain has two underlying functions that determine immediate rewards and 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:

$$V_\pi(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... \tag{2.1}$$

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 ($\epsilon << 1$), 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 appropriate RL technique is *Q*-learning [110], 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 highest-valued action, $argmax_a Q(s_t, a)$, at each step.

RL agents in deterministic worlds can begin with an inaccurate *Q*-function and recursively update it after each step:

$$Q(s_t, a_t) \longleftarrow r_t + \gamma \; max_a \; Q(s_{t+1}, a) \tag{2.2}$$

Thus the current estimate of a *Q*-value on the right is used to produce a new estimate on the left. Under certain conditions, *Q*-learning is guaranteed to converge to an accurate *Q*-function [110]. Even when these conditions are violated, the method can still produce successful learning in practice.

In the SARSA variant of *Q*-learning, the new estimate uses the actual a_{t+1} instead of the a with the highest *Q*-value; this takes exploration steps into account during updates. In non-deterministic worlds, a learning rate $\alpha \in (0, 1]$ is used to form a weighted average between the old estimate and the new one; this allows the *Q*-function to converge despite non-deterministic effects. With these two changes, the update equation becomes:

$$Q(s_t, a_t) \longleftarrow (1 - \alpha) \; Q(s_t, a_t) + \alpha \; (r_t + \gamma \; Q(s_{t+1}, a_{t+1})) \tag{2.3}$$

While these equations give update rules that look just one step ahead, to s_{t+1}, it is also possible to perform updates over multiple steps. In temporal-difference learning [86], 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 due to continuous-valued features, 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, which can speed up learning.

2.1.1 Implementing RL with Support Vector Regression

For experiments in this thesis, I use a form of *Q*-learning called SARSA(λ), which is the SARSA form of temporal-difference learning. The algorithm I use is RL via support-vector regression [48] (RL-SVR). It represents the state with a set of numeric features and approximates the *Q*-function for each action with a weighted linear sum of those features. It finds the feature weights by solving 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 algorithms make incremental updates to the *Q*-functions after each step the agent takes. However, completely re-solving the above optimization problem after each data point would be too computationally intensive. Instead, agents perform batches of 25 full episodes at a time and re-solve the optimization problem after each batch.

Formally, for each action, the RL-SVR algorithm 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. Agents follow the ϵ-greedy exploration method, taking the highest-valued action in all but a small set of random steps.

To compute the weight vector for an action, the RL-SVR algorithm finds the subset of training examples in which that action was taken and places those feature vectors into rows of a data matrix A. When A becomes too large for efficient solving, it begins to discard episodes randomly; 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, it computes *Q*-value estimates and places them into an output vector y. The optimal weight vector is then described by Equation 2.4.

$$Aw + b\overrightarrow{e} = y \tag{2.4}$$

Here $\overrightarrow{e}$ denotes a vector of ones. The 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 chosen randomly. The purpose of including the exploration examples is to ensure that bad moves are not forgotten. When there are not enough exploration examples, the RL-SVR algorithm creates synthetic ones by randomly choosing exploitation steps and using the current model to score random actions for those steps.

In practice, it is preferable 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. The RL-SVR algorithm therefore includes *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

$$\begin{aligned} \min_{(w,b,s)} \quad & ||w||_1 + \nu|b| + C||s||_1 \\ s.t. \quad & -s \leq Aw + b\overrightarrow{e} - y \leq s. \end{aligned} \tag{2.5}$$

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, the RL-SVR algorithm produces a weight vector w for each action that compromises between accuracy and simplicity. The tradeoff parameter C decays exponentially over time so that solutions may be more complex later in the learning process.

Several other parameters 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 process.

2.1.2 RoboCup: A Challenging Reinforcement Learning Domain

One motivating domain for transfer in reinforcement learning is RoboCup simulated soccer. The RoboCup project [59] 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 [81]

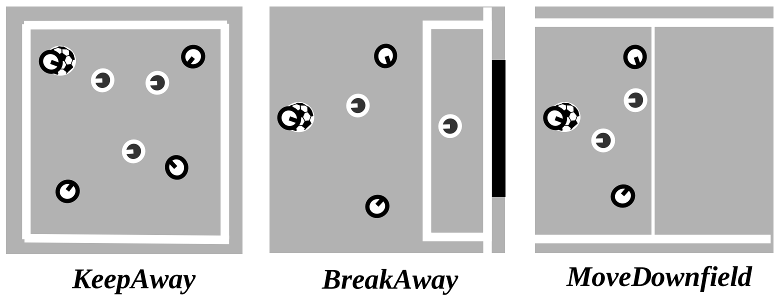

Figure 2.2: Snapshots of RoboCup soccer tasks. In KeepAway, one team passes the ball to prevent the other team from taking possession of it. In BreakAway, one team attempts to score a goal against another team. In Movedownfield, one team attempts to maneuver across a line while another team attempts to take possession of the ball.

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 2.2). 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 model built with data combined from all the players on its team.

The first RoboCup task is M-on-N KeepAway [81], 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. I have also developed a version of KeepAway in which move actions are allowed [103], but I use the standard version in this thesis.

The KeepAway state representation was designed by Stone and Sutton [81]. Appendix A lists all the features and actions. The keepers are ordered by their current distance to the learner *k0*, as are the takers.

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.

The MoveDownfield features and actions are also listed in Appendix A. The attackers are ordered by their current 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.

The BreakAway features and actions are also listed in Appendix A. The attackers are ordered by their current distance to the learner *a0*, as are the non-goalie defenders.

Stone and Sutton [81] found that learning in KeepAway is very difficult with only the continuous features listed. They propose *tiling* to overcome this difficulty. Tiling discretizes each feature into intervals, 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. I follow this approach and add to the feature space 32 tiles per continuous feature in all of the RoboCup tasks.

Some parameters in RL-SVR, including many that decay exponentially, need to be set appropriately for the domain. I use the following settings for RoboCup tasks. The temporal-difference parameter λ = exp($-age/100$) where the age of an episode is the number of episodes the learner trained on before that episode. The learning rate α has an initial value of 0.5 and a half-life of 1000 episodes, and the exploration rate ϵ has an initial value of 0.025 and a half-life of 2500 episodes.

The offset penalty $\nu = 100$, and the complexity penalty C has an initial value of 1000 with a half-life of 2500 games. I tuned these values for RL-SVR, and I use the tuned settings for the transfer algorithms in this thesis.

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.

2.2 Learning Curves and Statistical Comparisons

The performance of a reinforcement learner is typically illustrated with a *learning curve*. A reasonable learning curve displays increased performance (on the y-axis) as training progresses (on the x-axis). In this thesis, the figures for experimental results contain several learning curves: one for RL-SVR, and one for each transfer algorithm being compared.

I use a consistent methodology to display these curves. Because the RoboCup domain has high variance across RL runs, each curve is an average of 25 separate runs. Furthermore, because RoboCup games have high variance across batches within a run, each point on a curve is an average over the last ten batches (250 games). These two kinds of averaging smooth the curves to facilitate visual and statistical comparisons.

For transfer experiments, there is an additional source of variance: the source run used for transfer. To account for differences across source runs, I use five independent source-task runs, and from each of these, I do five target-task runs, producing the total of 25 runs. RL-SVR curves are simply 25 independent runs.

For all experiments, the x-axis shows the number of training games in the target task, which starts at 0 and ends at 3000 for RoboCup games; 3000 games is just long enough for all the

RoboCup games to reach an asymptote. The y-axis shows an appropriate measure of performance in the target task, which depends on the game. In BreakAway, it is the probability that the agents will score a goal in a game. In MoveDownfield, it is the average net distance traveled towards the right edge during a game. In KeepAway, it is the average length of a game before the opponents take possession of the ball.

I do not show any information about source-task learning when reporting experimental results for transfer algorithms. Source tasks are always learned with RL-SVR for 3000 games. I re-use source runs for all transfer algorithms; for example, in all experiments that involve transfer from 2-on-1 BreakAway, I use the same five 2-on-1 BreakAway source runs.

The learning-curve figures can give visual insight into the question of whether one transfer algorithm is better than another. Qualitatively speaking, if curve A is above curve B, then algorithm A is better. However, there are some cases where curve A is above curve B at the beginning but below it further down the x-axis. Qualitative conclusions may still be possible in cases like these if one has a preference for earlier or later performance. If not, a single performance measure is useful to make a quantitative comparison between algorithms. Even if curve A is consistently above curve B, a quantitative comparison is needed to evaluate whether the difference is statistically significant.

The measure I use to express the total performance of a run is the area under its curve, which is approximated by a sum of columns:

$$area = \sum_{i=1}^{120} 25 \times y_i \tag{2.6}$$

where each column height y_i is the y-value at batch i, the column width 25 is the constant size of each batch, and there are 120 batches for a total of 3000 games.

To determine whether the area for runs in group A is significantly different from the area for runs in group B, I use a randomization test [13] (see Table 2.1). The test starts by calculating the actual *t*-statistic for the group of runs in A versus the group of runs in B. Then it shuffles all the runs together, chooses two new groups from them randomly with replacement, and measures a new *t*-statistic. It repeats this randomization step many times (I use $r = 100,000$) to produce a list

Table 2.1: A randomization test to judge whether one group of numbers is significantly higher than another, based on Cohen [13].

```
Input: array of numbers $A = (a_1, a_2, ..., a_n)$   // In my experiments, these are areas under curves in group A
       array of numbers $B = (b_1, b_2, ..., b_n)$   // In my experiments, these are areas under curves in group B
Let $\bar{A}$ = average($a_1, a_2, ..., a_n$) and $\sigma_A^2$ = variance($a_1, a_2, ..., a_n$)
Let $\bar{B}$ = average($b_1, b_2, ..., b_n$) and $\sigma_B^2$ = variance($b_1, b_2, ..., b_n$)
Let $t = \frac{\bar{A}-\bar{B}}{\sqrt{\frac{2}{n}}\sqrt{\frac{\sigma_A^2+\sigma_B^2}{2}}}$        // The actual $t$-statistic
Let $T = \emptyset$                  // The eventual set of $r$ randomized $t$-statistics
For $i = 0$ to $r$
    Shuffle $A$ and $B$ randomly into $A_i$ and $B_i$
    Calculate $\bar{A}_i$, $\bar{B}_i$, $\sigma_{A_i}^2$, $\sigma_{B_i}^2$
    Calculate $t_i$ with the $t$-statistic equation above
    $T \leftarrow T \cup t_i$
If $t < 0$ let $p$ be the fraction of $t_i \in T$ more negative than $t$
If $t > 0$ let $p$ be the fraction of $t_i \in T$ more positive than $t$
If $p < 0.05$ then the difference between $A$ and $B$ is significant
```

of randomized *t*-statistics. Finally it estimates a *p*-value: the proportion of randomized *t*-statistics that have larger magnitude than the actual one. This is illustrated in Figure 2.3.

The idea behind this test is as follows. The *t*-statistic is a measure of how different two groups are. If A and B are truly different, mixing them randomly will not maintain that difference, and the great majority of the randomized *t*-statistics will have lower magnitude than the actual one. On the other hand, if A and B are not very different, mixing them will have less of an effect, and the proportion of randomized *t*-statistics that exceed the actual one will be higher. As the *p*-value becomes higher, one becomes less confident that A and B are significantly different. Convention dictates that one can conclude the difference between A and B is significant when $p < 0.05$.

To determine how different the areas for A and B are, I use another randomization test (see Table 2.2) that calculates a confidence interval for the difference between the areas [13]. This test randomly resamples runs from A with replacement, and likewise B, calculates the average areas for the new groups, and finds the new difference. It repeats this randomization step many times (I use $r = 100,000$) to produce a list of resampled differences. Finally it estimates a 95%

Table 2.2: A randomization test to estimate a 95% confidence interval for the difference between two groups of numbers, based on Cohen [13].

```
Input: array of numbers $A = (a_1, a_2, ..., a_n)$   // In my experiments, these are areas under curves in group A
       array of numbers $B = (b_1, b_2, ..., b_n)$   // In my experiments, these are areas under curves in group B
Let $D = \emptyset$        // The eventual set of $r$ resampled differences $\bar{A} - \bar{B}$
For $i = 0$ to $r$
    Resample $A$ randomly with replacement to get $A_i$
    Resample $B$ randomly with replacement to get $B_i$
    Let $\bar{A}_i$ = average($a_1, a_2, ..., a_n$)
    Let $\bar{B}_i$ = average($b_1, b_2, ..., b_n$)
    Let $d_i = \bar{A}_i - \bar{B}_i$
    $D \leftarrow D \cup d_i$
Sort $D$ in increasing order
Let $lower$ be the $2.5^{th}$ percentile of $D$
Let $upper$ be the $97.5^{th}$ percentile of $D$
Return interval $[lower, upper]$
```

confidence interval: the lower bound is the 2.5^{th} percentile of the list, and the upper bound is the 97.5^{th} percentile. This is also illustrated in Figure 2.3.

While a confidence interval is not particularly meaningful in isolation, it can be useful to compare intervals across different experiments. For example, if the interval between algorithm A and RL-SVR is $[5, 10]$ and the interval between algorithm B and RL-SVR is $[50, 100]$, then the *practical* difference provided by B is greater than that provided by A, even if they both provide a *statistical* difference.

For each comparison between an algorithm A and an algorithm B that I make in this dissertation, I report the *p*-value and the 95% confidence interval for area(A) - area(B). If $p < 0.05$, I indicate that the difference between the algorithms is statistically significant, and I note which algorithm won the comparison.

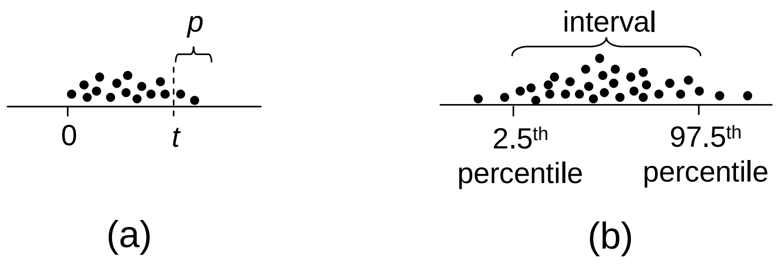

Figure 2.3: Illustrations of the statistical tests in Tables 2.1 and 2.2. (a) The dots represent randomized *t*-statistics for algorithms A and B, and the *p*-value captures how many are more extreme than the actual *t*. (b) The dots represent resampled differences between areas under curves A and B, and the 95% confidence interval encompasses 95% of these values.

2.3 Inductive Logic Programming

Inductive logic programming (ILP) is a technique for learning classifiers in first-order logic [68]. Most of my transfer algorithms use ILP to extract knowledge from the source task. To make these algorithms understandable, this section provides a brief overview of ILP.

2.3.1 What ILP Learns

An ILP algorithm learns a set of first-order clauses, which must usually be definite clauses. A definite clause has a *head*, which is a literal that evaluates to *true* or *false* based on a *body*, which is a conjunction of other literals. Literals 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 separate the literals in the body, denoting conjunction.

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 the ILP algorithm needs to learn a clause with the head *isBottomOfStack(Obj)* that is true when *Obj* = *desk* but false when *Obj* $\in$ {*floor, chair, book*}. Given those positive and negative examples, it might 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. ILP is one of few classification algorithms that use this powerful and natural type of reasoning. Most machine learning algorithms use the equivalent of *propositional* logic, which does not allow variables.

In many domains, the correct 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.

2.3.2 How ILP Learns

There are several types of algorithms for producing a set of first-order clauses. This section focuses on the Aleph system [80], which I use in my algorithms.

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, optionally removes the positive examples covered by that rule, and repeats the process on the remaining examples.

The search for a rule is essentially a search for a good set of literals. Literals may either be grounded, like *stackedOn(chair, floor)*, or variablized, like *stackedOn(Obj, floor)*. Legal literals must be defined for Aleph before the search, and are of course task-specific. Aleph also takes a parameter limiting the maximum clause length, which impacts the running time of the search. In my experiments, I use a maximum clause length of seven literals, which I found to produce reasonable running times given the RoboCup dataset sizes and to allow sufficiently expressive rules.

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 literals 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 subset of the literals 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 literals one by one to maximize a scoring function (see Figure 2.4).

A second Aleph procedure that I also use is *randomized rapid restart* [113]. 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 to maximize a scoring function. It performs M local moves for each of K random restarts. This method often finds better candidate clauses than the heuristic search does.

The rule-scoring function I use is the F measure, which is based on the two basic measures 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 measure combines the two in a harmonic mean:

$$F(\beta) = \frac{(1+\beta^2) \times Precision \times Recall}{\beta^2 \times Precision + Recall}$$

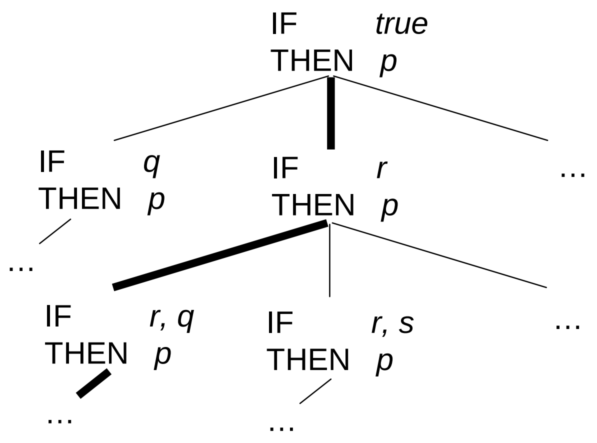

Figure 2.4: An illustration of a top-down ILP search for a clause to express the concept p using candidate literals $q, r, s, ...$. The body of the clause starts empty. The first step considers each literal and chooses the best (here r) to add. It then considers adding another and chooses the best (here q), and so on. Literals are shown here without arguments for simplicity, but in a real ILP search literals might have both grounded and variable arguments.

By default I use $\beta = 1$, in which precision and recall are weighted equally. However, in some algorithms I see fit to weight them unequally; $\beta > 1$ puts more weight on recall, and $0 < \beta < 1$ puts more weight on precision.

Aleph produces a theory for each concept, but I do not use these theories directly in my algorithms. Instead, I use a system called Gleaner [35] to create an ensemble of clauses. Gleaner divides the recall range into intervals $[0, 0.1], [0.1, 0.2]$, etc; it examines the clauses that Aleph encounters during its search and saves those with the highest precision in each recall interval. This method produces a greater diversity of potential clauses than the Aleph theory does.

When a single rule is needed, I use only the best clause that Gleaner saves. When multiple rules are needed, I select a final ruleset from the Gleaner clauses that attempts to maximize an overall F measure. This produces a higher-quality ruleset for my purposes, and the procedure is further described in later sections.

2.4 Markov Logic Networks

The Markov Logic Network (MLN) is a model developed by Richardson and Domingos [70] that combines first-order logic and probability. It expresses concepts with first-order rules, as ILP does, but unlike ILP it puts weights on the rules to indicate how important they are. While ILP rulesets can only predict a concept to be true or false, an MLN can estimate the probability that a concept is true, by comparing the total weight of satisfied rules to the total weight of violated rules. This type of probabilistic logic therefore conveys more information than pure logic. It is also less brittle, since world states that violate some rules are not impossible, just less probable.

Formally, a Markov Logic Network is a set of first-order logic formulas F, with associated real-valued weights W, that provides a template for a Markov network. The network contains a binary node for each possible grounding of each predicate of each formula in F, with groundings determined by a set of constants C. Edges exist between nodes if they appear together in a possible grounding of a formula. Thus the graph contains a clique for each possible grounding of each formula in F.

The classic example from Richardson and Domingos [70] follows. Suppose the formulas are:

$$\forall y \text{ Smokes}(y) \Rightarrow \text{Cancer}(y)$$
$$\forall y, z \text{ Friends}(y, z) \Rightarrow (\text{Smokes}(y) \Leftrightarrow \text{Smokes}(z))$$

These rules assert that smoking leads to cancer and that friends have similar smoking habits. These are both good examples of MLN formulas because they are often true, but not always; thus they will have finite weights (not shown). Given constants *Anna* and *Bob* that may be substituted for the variables y and z, this MLN produces the ground Markov network in Figure 2.5. (Note that the convention for capitalization is opposite here from in ILP; variables here are lower-case and constants are upper-case.)

Let X represent all the nodes in this example, and let $X = x$ indicate that among the possible worlds (the true/false settings of those nodes), x is the actual one. The probability distribution represented by the Markov network is:

$$P(X = x) = \frac{1}{Z} \, exp \sum_{i \in F} w_i n_i(x) \tag{2.7}$$

Here Z is a normalizing constant, w_i is the weight of formula $i \in F$, and $n_i(x)$ is the number of true groundings of formula i in the world x. Based on this equation, one can calculate the

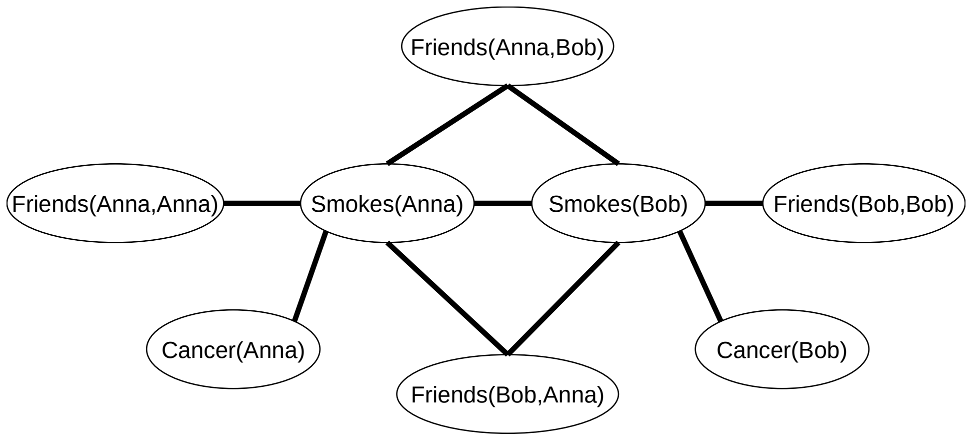

Figure 2.5: The ground Markov network produced by the MLN described in this section. This example and this image come from Richardson and Domingos [70]. Each clique in this network has a weight (not shown) derived from the formula weights.

probability of any node in the network given *evidence* about the truth values of some other nodes. This network inference problem is typically solved by an approximate-inference algorithm called MC-SAT [21] because solving it exactly is usually computationally intractable. However, in my experiments, the arrangement of the evidence makes an exact solution feasible, which I explain later.

Given a set of positive and negative examples of worlds, the formula weights can be learned rather than specified manually. There are several algorithms for weight learning; the current state-of-the-art is a method called *preconditioned scaled conjugate gradient* [46]. This is the default algorithm in the Alchemy software package [40], which I use for my experiments. Alchemy also provides an algorithm for structure learning (i.e. learning the formulas), which I do not use since I already have methods for learning rulesets via ILP.

Chapter 3

Survey of Research on Transfer Learning

This chapter provides an introduction to the goals, settings, and challenges of transfer learning. It surveys current research in this area, giving an overview of the state of the art and outlining the open problems. The survey covers transfer in both inductive learning and reinforcement learning, and discusses the issues of negative transfer and task mapping in depth. There are no original research contributions in this chapter, but my categorization of transfer methods is novel. This chapter is based on published work [101].

3.1 Transfer in General

The transfer of knowledge from one task to another is a desirable property in machine learning. Our ability as humans to transfer knowledge allows us to learn new tasks quickly by taking advantage of relationships between tasks. While many machine-learning algorithms learn each new task from scratch, there are also *transfer-learning* algorithms that can improve learning in a *target task* using knowledge from a previously learned *source task*.

A typical machine-learning problem can be characterized as:

GIVEN	Training data for task T
DO	Learn task T

A typical transfer-learning problem can be characterized as:

GIVEN	Training data for task T AND knowledge from related task(s) S
DO	Learn task T

Here S is one or more source tasks $(S_1, S_2, ...)$. This broad definition of transfer learning allows any type of machine-learning algorithm for learning the target task T. It also allows for the knowledge from the source tasks S to take any form. Figure 1.1 has already illustrated this problem formulation.

Transfer methods tend to be highly dependent on the algorithm used to learn the target task, and many transfer algorithms are simply extensions of traditional learning algorithms. Others are entirely new algorithms based on enabling transfer learning.

Some work in transfer learning is in the context of inductive learning, and involves extending well-known classification and inference algorithms such as neural networks, Bayesian networks, and Markov Logic Networks. Another major area is transfer in the context of reinforcement learning, which involves extending algorithms such as Q-learning and policy search. This chapter surveys these areas separately in Sections 3.2 and 3.3.

The goal of transfer learning is to improve learning in the target task by leveraging knowledge from the source task. There are three common measures by which transfer might improve learning. 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 3.1 illustrates these three measures.

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. Section 3.4 discusses approaches for avoiding negative transfer.

When an agent applies knowledge from one task in another, it is often necessary to map the characteristics of one task onto those of the other to specify correspondences. In much of the work on transfer learning, a human provides this *mapping*, but some work investigates ways to perform mapping automatically. Section 3.5 discusses work in this area.

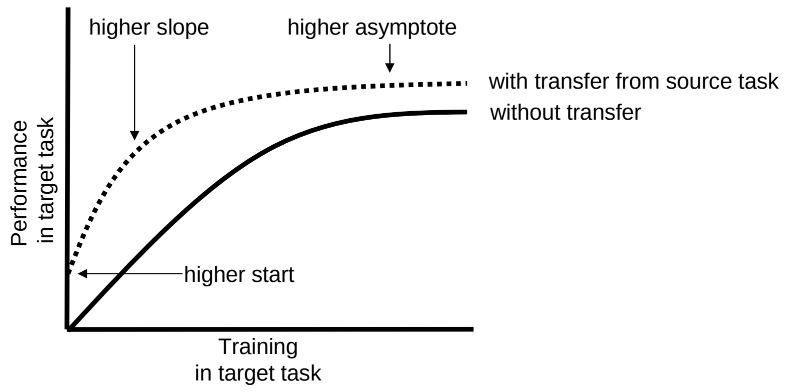

Figure 3.1: A standard learning curve displays increased performance as training progresses. With transfer, the curve may start higher, increase faster, or reach a higher asymptote. Any of these properties could be desired outcomes of transfer learning.

Finally, a small set of theoretical studies about transfer learning is presented in Section 3.6. This work addresses problems like defining task relatedness and setting bounds on transfer performance and efficiency.

I make a distinction between transfer learning and *multi-task learning* [11], in which several tasks are learned simultaneously (see Figure 3.2). Multi-task learning is closely related to transfer, but it does not involve designated source and target tasks; instead the learning agent receives information about several tasks at once. In transfer learning, the agent knows nothing about a target task (or even that there will be a target task) when it learns a source task. This is my own definition, not a universally accepted one, but it is a useful distinction because multi-task learning approaches are not always applicable to transfer learning.

3.2 Transfer in Inductive Learning

In an inductive learning task, the objective is to induce a predictive model from a set of training examples [57]. Often the goal is classification, i.e. assigning class labels to examples. Examples of classification systems are artificial neural networks [73] and symbolic rule-learners [68]. Another type of inductive learning involves modeling probability distributions over interrelated variables, usually with graphical models. Examples of these systems are Bayesian networks [38] and Markov Logic Networks [70].

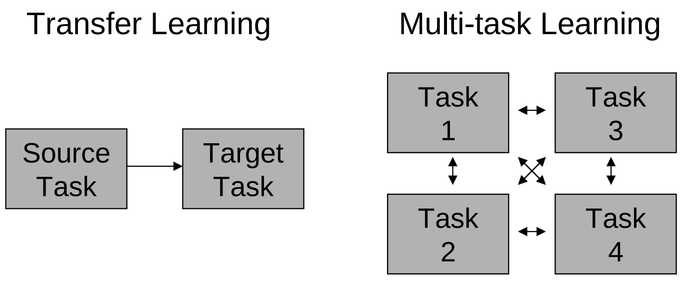

Figure 3.2: As I define transfer learning, the information flows in one direction only, from the source task to the target task. In multi-task learning, information can flow freely among all tasks.

The predictive model constructed by an inductive learning algorithm should make accurate predictions not just on the training examples, but also on future examples that come from the same distribution. In order to produce a model with this generalization capability, a learning algorithm must have an *inductive bias* [57] – a set of assumptions about the training data and the function that produced it.

The bias of an algorithm determines the *hypothesis space* of possible models that it considers. For example, the hypothesis space of the Naive Bayes model is limited by the assumption that example characteristics are conditionally independent given the class of an example. The bias also determines the algorithm's search process through the hypothesis space, which controls the order in which hypotheses are considered. For example, rule-learning algorithms typically construct rules one constraint at a time, which reflects the assumption that constraints contribute significantly to example coverage by themselves rather than in pairs or more.

Transfer in inductive learning typically works by allowing source-task knowledge to affect the target task's inductive bias. It is usually concerned with improving the speed with which a model is learned, or with improving its generalization capability. The next subsection discusses inductive transfer in general, and the following ones elaborate on three specific and popular settings.

There is some related work that is not discussed here because it specifically addresses multi-task learning. For example, Niculescu-Mizil and Caruana [58] learn Bayesian networks simultaneously for multiple related tasks by biasing learning toward similar structures for each task. While this is clearly related to transfer learning, it is not directly applicable to the scenario in which a target task is encountered after one or more source tasks have already been learned.

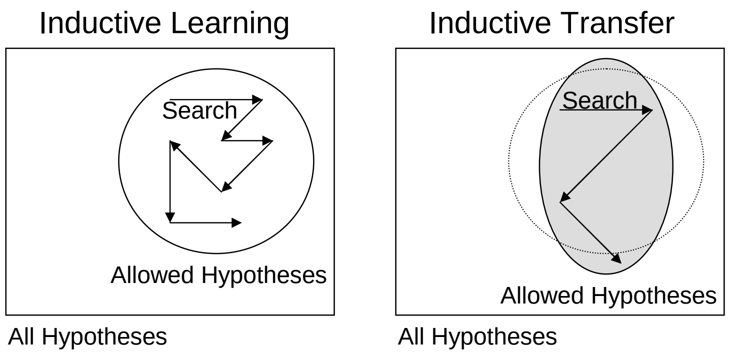

Figure 3.3: Inductive learning can be viewed as a directed search through a hypothesis space [57]. Inductive transfer uses source-task knowledge to adjust the inductive bias, which could involve changing the hypothesis space or the search steps.

3.2.1 Inductive Transfer

In *inductive transfer* methods, the target-task inductive bias is chosen or adjusted based on the source-task knowledge (see Figure 3.3). The way this is done varies depending on which inductive learning algorithm is used to learn the source and target tasks. Some transfer methods narrow the hypothesis space, limiting the possible models, or remove search steps from consideration. Other methods broaden the space, allowing the search to discover more complex models, or add new search steps.

Baxter [5] frames the transfer problem as that of choosing one hypothesis space from a family of spaces. By solving a set of related source tasks in each hypothesis space of the family and determining which one produces the best overall generalization error, he selects the most promising space in the family for a target task.

Thrun and Mitchell [100] look at solving Boolean classification tasks in a lifelong-learning framework, where an agent encounters a collection of related problems over its lifetime. They learn each new task with a neural network, but they enhance the standard gradient-descent algorithm with slope information acquired from previous tasks. This speeds up the search for network parameters in a target task and biases it towards the parameters for previous tasks.

Silver at al. [77] use a single neural network to learn multiple tasks, using context inputs to specify which task an example belongs to. This causes knowledge transfer between tasks through shared parameters in the network. They show that this approach is effective for neural-network learners but not for some other learners, such as decision trees and suppor-vector machines.

Mihalkova and Mooney [56] perform transfer between Markov Logic Network models. Given a learned MLN for a source task, they learn an MLN for a related target task by starting with the source-task one and diagnosing each formula, adjusting ones that are too general or too specific in the target domain. The hypothesis space for the target task is therefore defined in relation to the source-task MLN by the operators that generalize or specify formulas.

Hlynsson [36] phrases transfer learning in classification as a minimum description length problem given source-task hypotheses and target-task data. That is, the chosen hypothesis for a new task can use hypotheses for old tasks but stipulate exceptions for some data points in the new task. This method aims for a tradeoff between accuracy and compactness in the new hypothesis.

Ben-David and Schuller [7] propose a transformation framework to determine how related two Boolean classification tasks are. They define two tasks as related with respect to a class of transformations if they are equivalent under that class– that is, if a series of transformations can make one task look exactly like the other.

3.2.2 Bayesian Transfer

One common scenario for inductive transfer is in Bayesian learning methods. Bayesian learning involves modeling probability distributions and taking advantage of conditional independence among variables to simplify the model. An additional aspect that Bayesian models often have is a *prior distribution*, which describes the assumptions one can make about a domain before seeing any training data. Given the data, a Bayesian model makes predictions by combining it with the prior distribution to produce a *posterior distribution*. A strong prior can significantly affect these results (see Figure 3.4). This serves as a natural way for Bayesian learning methods to incorporate prior knowledge – in the case of transfer learning, source-task knowledge.

Marx et al. [53] use a Bayesian transfer method for tasks solved by a logistic regression classifier. The usual prior for this classifier is a Gaussian distribution with a mean and variance set through cross-validation. To perform transfer, they instead estimate the mean and variance by

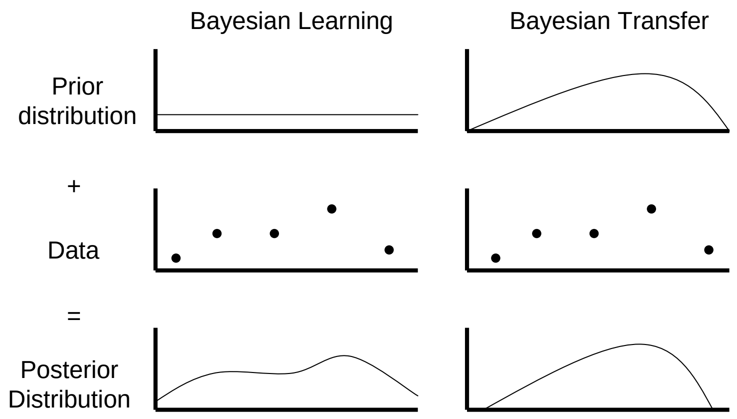

Figure 3.4: Bayesian learning uses a prior distribution to smooth the estimates from training data. Bayesian transfer may provide a more informative prior from source-task knowledge.

averaging over several source tasks. Raina et al. [69] use a similar approach for multi-class classification by learning a multivariate Gaussian prior from several source tasks.

Dai et al. [15] apply a Bayesian transfer method to a Naive Bayes classifier. They set the initial probability parameters based on a single source task, and revise them using target-task data. They also provide some theoretical bounds on the prediction error and convergence rate of their algorithm.

3.2.3 Hierarchical Transfer

Another popular setting for transfer in inductive learning is *hierarchical transfer*. In this setting, solutions to simple tasks are combined or provided as tools to produce a solution to a more complex task (see Figure 3.5). This can involve many tasks of varying complexity, rather than just a single source and target. The target task might use entire source-task solutions as parts of its own, or it might use them in a more subtle way to improve learning.

Sutton and McCallum [84] begin with a sequential approach where the prediction for each task is used as a feature when learning the next task. They then proceed to turn the problem into a multi-task learning problem by combining all the models and applying them jointly, which brings their method outside our definition of transfer learning, but the initial sequential approach is an example of hierarchical transfer.

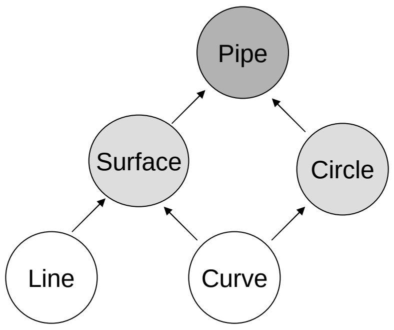

Figure 3.5: An example of a concept hierarchy that could be used for hierarchical transfer, in which solutions from simple tasks are used to help learn a solution to a more complex task. Here the simple tasks involve recognizing lines and curves in images, and the more complex tasks involve recognizing surfaces, circles, and finally pipe shapes.

Stracuzzi [82] looks at the problem of choosing relevant source-task Boolean concepts from a knowledge base to use while learning more complex concepts. He learns rules to express concepts from a stream of examples, allowing existing concepts to be used if they help to classify the examples, and adds and removes dependencies between concepts in the knowledge base.

Taylor et al. [93] propose a transfer hierarchy that orders tasks by difficulty, so that an agent can learn them in sequence via inductive transfer. By putting tasks in order of increasing difficulty, they aim to make transfer more effective. This approach may be more applicable to the multi-task learning scenario, since by our definition of transfer learning the agent may not be able to choose the order in which it learns tasks, but it could be applied to help choose from an existing set of source tasks.

3.2.4 Transfer with Missing Data or Class Labels

Inductive transfer can be viewed not only as a way to improve learning in a standard supervised-learning task, but also as a way to offset the difficulties posed by tasks that involve semi-supervised learning [115] or small datasets. That is, if there are small amounts of data or class labels for a task, treating it as a target task and performing inductive transfer from a related source task can lead to more accurate models. These approaches therefore use source-task data to enhance target-task data, despite the fact that the two datasets are assumed to come from different probability distributions.

The Bayesian transfer methods of Dai et al. [15] and Raina et al. [69] are intended to compensate for small amounts of target-task data. One of the benefits of Bayesian learning is the stability that a prior distribution can provide in the absence of large datasets. By estimating a prior from related source tasks, these approaches can reduce the overfitting that would tend to occur with limited data.

Dai et al. [16] address transfer learning in a boosting algorithm using large amounts of data from a previous task to supplement small amounts of new data. Boosting is a technique for learning several weak classifiers and combining them to form a stronger classifier [31]. After each classifier is learned, the examples are reweighted so that later classifiers focus more on examples the previous ones misclassified. Dai et al. extend this principle by also weighting source-task examples according to their similarity to target-task examples. This allows the algorithm to leverage source-task data that is applicable to the target task while paying less attention to data that appears less useful.

Shi et al. [76] look at transfer learning in unsupervised and semi-supervised settings. They assume that a reasonably-sized dataset exists in the target task, but it is largely unlabeled due to the expense of having an expert assign labels. To address this problem they propose an active learning approach, in which the target-task learner requests labels for examples only when necessary. They construct a classifier with labeled examples, including mostly source-task ones, and estimate the confidence with which this classifer can label the unknown examples. When the confidence is too low, they request an expert label.

3.3 Transfer in Reinforcement Learning

There are several categories of reinforcement learning algorithms, and transfer learning approaches vary between these categories. Some types of methods are only applicable when the agent knows its environment model (the reward function and the state transition function). In this case, dynamic programming can solve directly for the optimal policy without requiring any interaction with the environment. In most RL problems, however, the model is unknown. *Model-learning* approaches use interaction with the environment to build an approximation of the true model. *Model-free* approaches learn to act without ever explicitly modeling the environment.

Temporal-difference methods [86] operate by maintaining and iteratively updating *value functions* to predict the rewards earned by actions. They begin with an inaccurate function and update it based on interaction with the environment, propagating reward information back along action sequences. One popular method is *Q*-learning [110], which involves learning a function $Q(s, a)$ that estimates the 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 corresponding to $argmax_a Q(s_t, a)$. When there are small finite numbers of states and actions, the *Q*-function can be represented explicitly as a table. In domains that have large or infinite state spaces, a function approximator such as a neural network or support-vector machine can be used to represent the *Q*-function.

Policy-search methods, instead of maintaining a function upon which a policy is based, maintain and update a policy directly. They begin with an inaccurate policy and update it based on interaction with the environment. Heuristic search and optimization through gradient descent are among the approaches that can be used in policy search.

Transfer in RL is typically concerned with speeding up the learning process, since RL agents can spend many episodes doing random exploration before acquiring a reasonable *Q*-function. I divide RL transfer into five categories that represent progressively larger changes to existing RL algorithms. The subsections below describe those categories and present examples from published research.

3.3.1 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 (see Figure 3.6). Compared to the random or zero setting that RL algorithms usually use at first, these *starting-point methods* can begin the RL process at a point

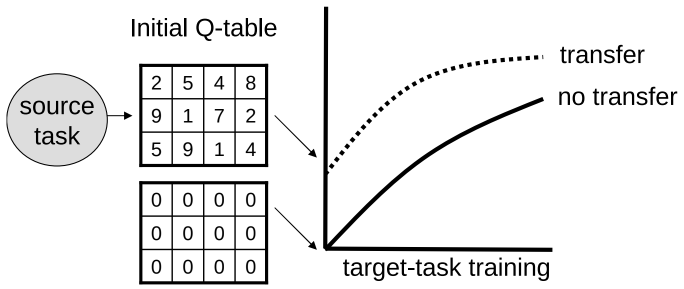

Figure 3.6: Starting-point methods for RL transfer set the initial solution based on the source task, starting at a higher performance level than the typical initial solution would. In this example, a Q-function table is initialized to a source-task table, and the target-task performance begins at a level that is only reached after some training when beginning with a typical all-zero table.

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. [96] 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 [89] 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. [44] 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.

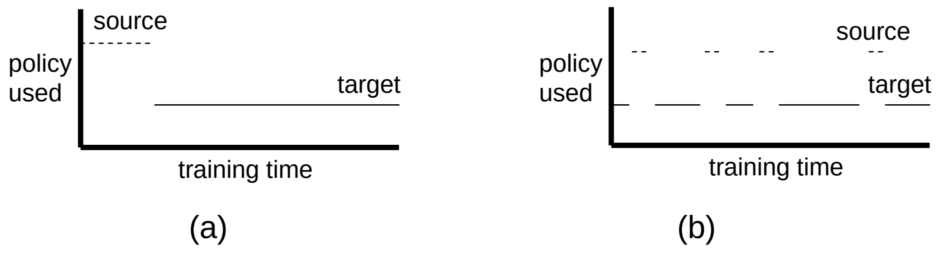

Figure 3.7: Imitation methods for RL transfer follow the source-task policy during some steps of the target task. The imitation steps may all occur at the beginning of the target task (a), or they may be interspersed with steps that follow the developing target-task policy (b).

Moving away from temporal-difference RL, starting-point methods can take even more forms. In a model-learning Bayesian RL algorithm, Wilson et al. [112] 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. [97] transfer a population of policies from a source task to serve as the initial population for a target task.

3.3.2 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 (see Figure 3.7).

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 [51] 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 [29] 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.

I developed an imitation method called *demonstration*, in which the target-task agent follows a source-task policy for a fixed number of initial episodes, and then reverts to normal RL [104]. In the early steps of the target task, the current policy can be so ill-formed that exploiting it is no different than exploring randomly. This approach aims to avoid that initial uncertainty and to generate enough data to create a reasonable target-task policy by the time the demonstration period ends. My original research in Chapters 5 and 6 uses the demonstration method.

3.3.3 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 (see Figure 3.8).

An early approach of this type is to compose several source-task solutions to form a target-task solution, as is done by Singh [78]. 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. [54] 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 [18] in the target task. This uses transfer to remove the burden of designing a task hierarchy.

Other approaches operate within the framework of *options*, which is a term for temporally-extended actions in RL [65]. 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 (see Figure 3.8).

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. [14] do. They learn a relational decision tree to represent the

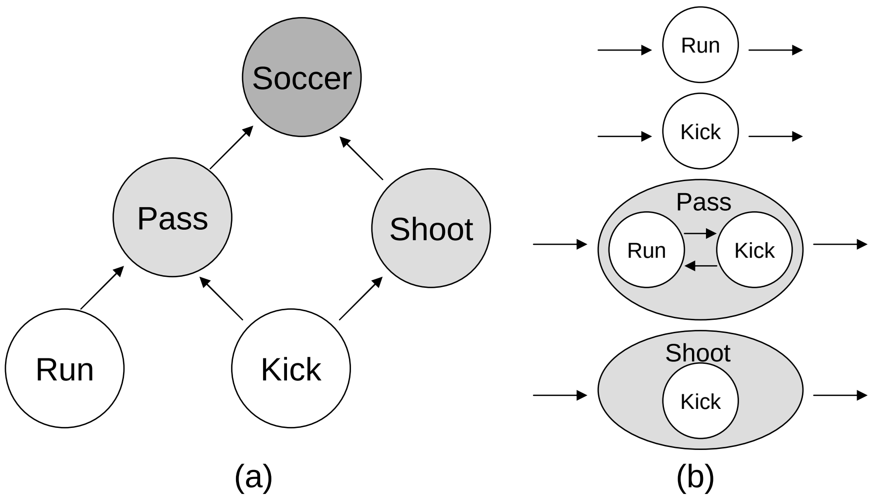

Figure 3.8: (a) An example of a task hierarchy that could be used to train agents to play soccer via hierarchical RL. Lower-level abilities like kicking a ball and running are needed for higher-level abilities like passing and shooting, which could then be combined to learn to play soccer. (b) The mid-level abilities represented as options alongside the low-level actions.

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 [3] do this by finding frequently-visited states in the source task to serve as ending conditions for options.

3.3.4 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. [107] 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 [94] 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 [75] 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 [41] 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.

3.3.5 New RL Algorithms

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.

Price and Boutilier [67] 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 calculation 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. [74] 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. [91] 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.

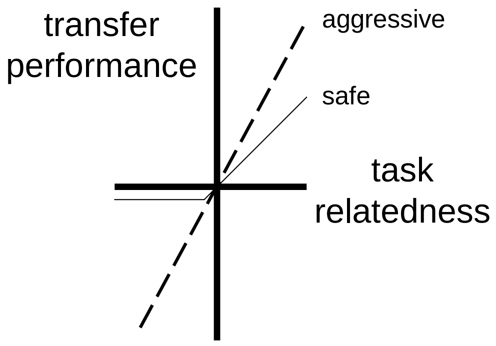

Figure 3.9: A representation of how the degree of relatedness between the source and target tasks translates to target-task performance when conducting transfer from the source task. With aggressive approaches, there can be higher benefits at high degrees of relatedness, but there can also be negative transfer at low levels. Safer approaches may limit negative transfer at the lower end, but may also have fewer benefits at the higher end.

My original research in Chapter 4 uses an algorithm for advice taking in RL that also falls into this category.

3.4 Avoiding Negative Transfer

Given a target task, the effectiveness of any transfer method depends on the source task and how it is related to the target. If the relationship is strong and the transfer method can take advantage of it, the performance in the target task can significantly improve through transfer. However, if the source task is not sufficiently related or if the relationship is not well leveraged by the transfer method, the performance with many approaches may not only fail to improve – it may actually decrease. This section examines work on preventing transfer from negatively affecting performance.

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. Conversely, approaches that transfer aggressively and produce large positive-transfer effects often have less protection against negative transfer (see Figure 3.9).

For example, consider the imitation methods for RL transfer. On one end of the range, an agent imitates a source-task policy only during infrequent exploration steps, and on the other end it demonstrates the source-task policy for a fixed number of initial episodes. The exploration method is very cautious and therefore unlikely to produce negative transfer, but it is also unlikely to produce large initial performance increases. The demonstration method is very aggressive; if the source-task policy is a poor one for the target task, following it blindly will produce negative transfer. However, when the source-task solution is a decent one for the target task, it can produce some of the largest initial performance improvements of any method.

3.4.1 Rejecting Bad Information

One way of approaching negative transfer is to attempt to recognize and reject harmful source-task knowledge while learning the target task. The goal in this approach is to minimize the impact of bad information, so that the transfer performance is at least no worse than learning the target task without transfer. At the extreme end, an agent might disregard the transferred knowledge completely, but some methods also allow it to selectively reject parts and keep other parts.

Option-based transfer in reinforcement learning (e.g. Croonenborghs et al. [14]) is an example of an approach that naturally incorporates the ability to reject bad information. Since options are treated as additional actions, the agent can choose to use them or not to use them; in Q-learning, for example, agents learn Q-values for options just as for native actions. If an option regularly produces poor performance, its Q-values will degrade and the agent will choose it less frequently. However, if an option regularly leads to good results, its Q-values will grow and the agent will choose it more often. Option-based transfer can therefore provide a good balance between achieving positive transfer and avoiding negative transfer.

The advice-taking algorithm that I use for algorithms in Chapter 4 is an approach that incorporates the ability to reject bad information. It is based on the RL-SVR algorithm, which approximates the Q-function with a support-vector machine, and it includes advice from the source task as a soft constraint. Since the Q-function trades off between matching the agent's experience and matching the advice, the agent can learn to disregard advice that disagrees with its experience.

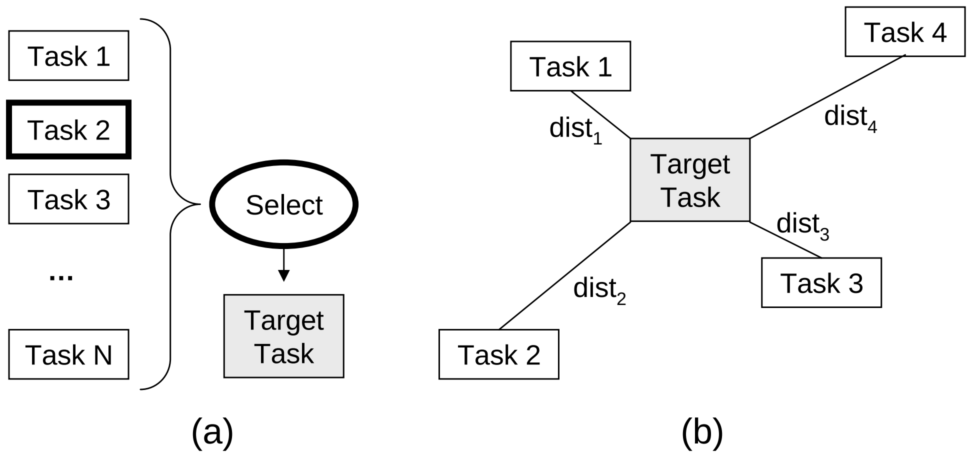

Figure 3.10: (a) One way to avoid negative transfer is to choose a good source task from which to transfer. In this example, Task 2 is selected as being the most related. (b) Another way to avoid negative transfer is to model the way source tasks are related to the target task and combine knowledge from them with those relationships in mind.

Rosenstein et al. [71] present an approach for detecting negative transfer in naive Bayes classification tasks. They learn a hyperprior for both the source and target tasks, and the variance of this hyperprior is proportional to the dissimilarity between the tasks. It may be possible to use a method like this to decide whether to transfer at all, by setting an acceptable threshold of similarity.

3.4.2 Choosing a Source Task

There are more possibilities for avoiding negative transfer if there exists not just one source task, but a set of candidate source tasks. In this case the problem becomes choosing the best source task (see Figure 3.10). Transfer methods without much protection against negative transfer may still be effective in this scenario, as long as the best source task is at least a decent match.

An example of this approach is the previously-mentioned transfer hierarchy of Taylor et al. [93], who order tasks by difficulty. Appropriate source tasks are usually less difficult than the target task, but not so much simpler that they contain little information. Given a task ordering, it may be possible to locate the position of the target task in the hierarchy and select a source task that is only moderately less difficult.

Talvitie and Singh [88] use a straightforward method of selecting a previous Markov decision process to transfer. They run each candidate MDP in the target task for a fixed length of time and order them by their performance. Then they select the best one and continue with it, only proceeding down the list if the current MDP begins performing significantly worse than it originally appeared. This trial-and-error approach, though it may be costly in the aggregate number of training episodes needed, is simple and widely applicable.

Kuhlmann and Stone [42] look at finding similar tasks when each task is specified in a formal language. They construct a graph to represent the elements and rules of a task. This allows them to find identical tasks by checking for graph isomorphism, and by creating minor variants of a target-task graph, they can also search for similar tasks. If they find an isomorphic match, they conduct value-function transfer.

Eaton and DesJardins [25] propose choosing from among candidate solutions to a source task rather than from among candidate source tasks. Their setting is multi-resolution learning, where a classification task is solved by an ensemble of models that vary in complexity. Low-resolution models are simple and coarse, while higher-resolution models are more complex and detailed. They reason that high-resolution models are less transferrable between tasks, and select a resolution below which to share models with a target task.

3.4.3 Modeling Task Similarity

Given multiple candidate source tasks, it may be beneficial to use several or all of them rather than to choose just one (see Figure 3.10). Some approaches discussed in this chapter do this naively, without evaluating how the source tasks are related to the target. However, there are some approaches that explicitly model relationships between tasks and include this information in the transfer method. This can lead to better use of source-task knowledge and decrease the risk of negative transfer.

Carroll and Seppi [10] develop several similarity measures for reinforcement learning tasks, comparing policies, value functions, and rewards. These are only measurable while the target task is being learned, so their practical use in transfer scenarios is limited. However, they make the relevant point that task similarity is intimately linked with a particular transfer method, and cannot be evaluated independently.

Eaton et al. [26] construct a graph in which nodes represent source tasks and weighted arcs represent a transferability metric. Given a new inductive learning task, they estimate parameters by fitting the task into the graph and learning a function that translates graph locations to task parameters. This method not only models the relationships between tasks explicitly, but also gives an algorithm for the informed use of several source tasks in transfer learning.

Gao et al. [32] propose that task similarity can be a local measure rather than a global measure. They estimate similarities in the neighborhood of each test example individually. These local consistency estimates become weights for the source tasks, and are used in a weighted ensemble to classify the test example.

Ruckert and Kramer [72] look at inductive transfer via kernel methods. They learn a meta-kernel that serves as a similarity function between tasks. Given this and a set of kernels that perform well in source tasks, they perform numerical optimization to construct a kernel for a target task. This approach determines the inductive bias in the target task (the kernel) by combining information from several source tasks whose relationships to the target are known.

Zhang et al. [114] model task relatedness through shared latent variables. Each task in their model includes some task-specific variables and some shared variables that provide common structure.

3.5 Automatically Mapping Tasks

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 learner understands. In some cases, the representations of the tasks are assumed to be identical, or at least one is a subset of the other. Otherwise, a *mapping* is needed to translate between task representations (see Figure 3.11).

Many transfer approaches do not address the mapping problem directly and require that a human provide this information, including the algorithms in this thesis. However, there are some

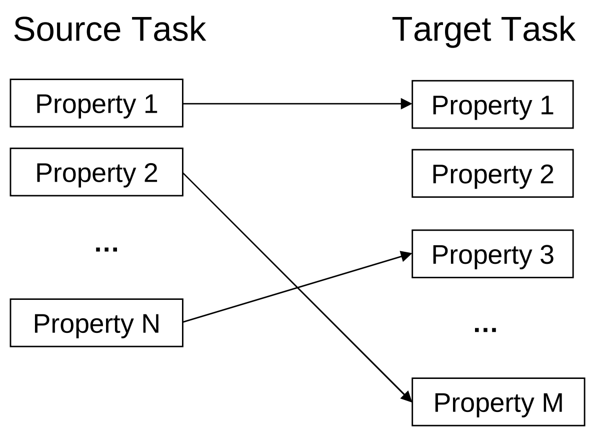

Figure 3.11: A mapping generally translates source-task properties into target-task properties. The numbers of properties may not be equal in the two tasks, and the mapping may not be one-to-one. Properties include entries in a feature vector, objects in a relational world, RL actions, etc.

transfer approaches that do address the mapping problem. This section discusses some of this work.

3.5.1 Equalizing Task Representations

For some transfer scenarios, it may be possible to avoid the mapping problem altogether by ensuring that the source and target tasks have the same representation. If the language of the source-task knowledge is identical to (or a subset of) the language of the target task, it can be applied directly with no translation. Sometimes a domain can be constructed so that this occurs naturally, or a common representation that equalizes the tasks can be found.

Relational learning is useful for creating domains that naturally produce common task representations. First-order logic represents objects in a domain with symbolic variables, which can allow abstraction that the more typical propositional feature vector cannot. Driessens et al. [24] show how relational reinforcement learning can simplify transfer in RL.

Another framework for constructing a domain relationally is that of Konidaris and Barto [41], who express knowledge in two different spaces. In *agent space* the representation is constant across tasks, while in *problem space* it is task-dependent. They transfer agent-space knowledge only because its common representation makes it straightforward to transfer.

Pan et al. [62, 63] take a mathematical approach to finding a common representation for two separate classification tasks. They use kernel methods to find a low-dimensional feature space where the distributions of source and target data are similar, and project a source-task model into this smaller space for transfer.

3.5.2 Trying Multiple Mappings

One straightforward way of solving the mapping problem is to generate several possible mappings and allow the target-task agent to try them all. The candidate mappings can be an exhaustive set, or they can be limited by constraints on what elements are permissible matches for other elements. Exhaustive sets may be computationally infeasible for large domains.

Taylor et al. [92] perform an exhaustive search of possible mappings in RL transfer. They evaluate each candidate using a small number of episodes in the target task, and select the best one to continue learning. Mihalkova et al. [55] limit their search for mappings in MLN transfer, requiring that mapped predicates have matching arity and argument types. Under those constraints, they conduct an exhaustive search to find the best mapping between networks.

Soni and Singh [79] not only limit the candidate mappings by considering object types, but also avoid a separate evaluation of each mapping by using options in RL transfer. They generate a set of possible mappings by connecting target-task objects to all source-task objects of the matching type. With each mapping, they create an option from a source MDP. The options framework gives an inherent way to compare multiple mappings while learning a target MDP without requiring extra trial periods.

3.5.3 Mapping by Analogy

If the task representations must differ, and the scenario calls for choosing one mapping rather than trying multiple candidates, then there are some methods that construct a mapping by analogy. These methods examine the characteristics of the source and target tasks and find elements that correspond. For example, in reinforcement learning, actions that correspond produce similar rewards and state changes, and objects that correspond are affected similarly by actions.

Analogical structure mapping [28] is a generic procedure based on cognitive theories of analogy that finds corresponding elements. It assigns scores to local matches and searches for a global match that maximizes the scores; permissible matches and scoring functions are domain-dependent. Several transfer approaches use this framework solve the mapping problem. Klenk and Forbus [39] apply it to solve physics problems that are written in a predicate-calculus language by retrieving and forming analogies from worked solutions written in the same language. Liu and Stone [45] apply it in reinforcement learning to find matching features and actions between tasks.

There are also some approaches that rely more on statistical analysis than on logical reasoning to find matching elements. Taylor and Stone [95] learn mappings for RL tasks by running a small number of target-task episodes and then training classifiers to characterize actions and objects. If a classifier trained for one action predicts the results of another action well, then those actions are mapped; likewise, if a classifier trained for one object predicts the behavior of another object well, those objects are mapped. Wang and Mahadevan [108] translate datasets to low-dimensional feature spaces using dimensionality reduction, and then perform a statistical shaping technique called Procrustes analysis to align the feature spaces.

3.6 Theory of Transfer Learning

Most transfer methods are evaluated experimentally rather than theoretically, including the algorithms in this thesis. The small amount of work on theoretical evaluation of transfer algorithms focuses on highly restricted transfer scenarios where the relationships between the source and target tasks are mathematically well-defined. For more complex tasks, such as reinforcement-learning tasks, current research does not contain any theoretical analyses.

Baxter [4] uses the Probably Approximately Correct (PAC) framework to give bounds on the number of source tasks and examples to learn an inductive bias for transfer between classification tasks. The PAC framework ensures that with high probability, the learner will find an inductive bias that produces a solution with low error in the target task. He also evaluates Bayesian transfer in classification tasks, showing the inverse relationship between the amount of target-task data needed and the number of source tasks and examples.

Baxter [5] also shows that transfer learning can improve generalization in classification tasks. He derives bounds on the generalization capability of a target-task solution given the number of source tasks and examples in each task.

While most of the work cited in this chapter involves experimental evaluation, some also includes some theoretical analysis. For example, Dai et al. [16] provide some theoretical bounds in their work on transfer via boosting. They derive bounds on the prediction error and convergence rate of their algorithm.

Abernethy et al. [1] provide some theoretical results in the context of sequential problem-solving with expert advice. Their goal is to learn a forecaster for sequential problems of multiple tasks that performs well compared to the best m experts, where the experts are assumed to perform similarly on related tasks. They prove an exponential bound on the exact solution to this problem, and they propose a probabilistic algorithm to solve it approximately.

Obozinski et al. [60] provide some theoretical results in the context of feature selection in multitask learning. They use an optimization problem to assign weights to each feature in each task. They show that an exact solution is computationally difficult, and propose a probabilistic method to solve it approximately.

Ben-David and Schuller [7], instead of defining task relatedness through statistical distributions as Baxter does, propose a transformation framework to determine how related two Boolean classification tasks are. They define two tasks as related with respect to a class of transformations if they are equivalent under that class; that is, if a series of transformations can make one task look exactly like the other. They provide conditions under which learning related tasks concurrently requires fewer examples than single-task learning.

Ando and Zhang [2] provide theoretical justification for using multiple source tasks in transfer learning. They learn predictors for multiple related source tasks and analyze them to find common structures, which they then use in a target task. They prove that using more source tasks leads to better estimation of the shared hypothesis space for the tasks.

3.7 Recap of Contributions

At this point, I recap my contributions to the field of transfer learning, and situate them within the categories provided by this chapter. I focus on transfer in reinforcement learning, so my algorithms belong primarily to Section 3.3.

My transfer algorithms 3, 4, 5, and 6 are imitation methods, belonging to Section 3.3.2. They involve representing the source-task policy in a useful way and then applying it in the target task via demonstration: the target-task learner follows the source-task policy for a fixed number of initial episodes, and then continues learning with normal RL. Chapters 5 and 6 present these algorithms, which differ only in their representation of the source-task policy.

My transfer algorithms 1 and 2 use an RL algorithm that inherently enables transfer, belonging to Section 3.3.5. This new algorithm performs advice-taking, accepting source-task knowledge in the form of logical rules and combining this advice with target-task experience. Chapter 4 presents these algorithms, which only differ in the way they construct advice from the source task.

The techniques from Chapter 4 are also relevant to Section 3.4 because they contain protection against negative transfer. By treating advice as a soft constraint, they are able to reject information that disagrees with target-task experience. These algorithms therefore also belong in part to Section 3.4.1.

Chapter 4

Advice-Based Transfer Methods

Advice is a set of instructions about a task solution that may not be complete or perfectly correct. Advice-taking algorithms allow advice to be followed, refined, or ignored according to its value. Traditionally, advice comes from humans.

I use advice in a novel way to perform transfer: by using source-task knowledge to provide advice for a target task. In an advice-taking algorithm, this advice is followed if it leads to positive transfer, but refined or ignored if it leads to negative transfer.

The transfer methods in this section are essentially different ways of producing transfer advice from source-task knowledge. Some of my transfer methods also allow human-provided advice in addition to transfer advice, providing a natural and powerful way for humans to contribute to the transfer process.

This chapter introduces advice taking and explains how it is implemented within the RL-SVR algorithm, and then presents two advice-based transfer methods for reinforcement learning. One of these is the only algorithm in this thesis that does not use first-order logic, and the other is the first of several relational methods. This chapter is based on published work [49, 50, 103, 105, 106].

4.1 Advice Taking

Advice taking can be viewed as learning with prior knowledge that may not be complete or perfectly correct. It focuses on taking advantage of the useful aspects of prior knowledge without depending on it completely. Research in this area typically assumes the knowledge comes from humans, though in my work it can also be extracted from a source task by an algorithm.

Algorithms for advice taking exist for both reinforcement learning and inductive learning. Pazzani and Kibler [64] describe a rule-learning system that takes existing background knowledge into account. Maclin and Shavlik [47] accept rules for action selection and incorporate them into a neural-network *Q*-function model using the knowledge-based neural network framework. Driessens and Dzeroski [23] use behavior traces from an expert to learn a partial initial *Q*-function for relational RL. Kuhlmann et al. [43] accept rules that give a small boost to the *Q*-values of recommended actions.

The advice-taking algorithm I use in this thesis is called Knowledge-Based Kernel Regression (KBKR). It was designed by Mangasarian et al. [52], and colleagues and I applied it to the RL-SVR algorithm [48].

In KBKR, advice is treated as a set of soft constraints on the *Q*-function. For example, here is some advice about passing in soccer:

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

In this example, there are two conditions describing the state of the agent's environment: an opponent is nearby and an unblocked path to a teammate exists. 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.

An agent can follow advice, only follow it approximately (which is like refining it), or ignore it altogether. This is accomplished by extending the RL-SVR optimization problem from Equation 2.5 to include terms for the advice. Each advice rule creates a new constraint on the *Q*-function solution in addition to the constraints from the training data.

In particular, since I use a version of KBKR called Preference-KBKR [49], my advice rules give conditions under which one action is preferred over another action. Advice therefore can be expressed using two arrays B and d that define the conditions on the state x as follows:

$$Bx \leq d \Longrightarrow Q_p(x) - Q_n(x) \geq \beta, \tag{4.1}$$

This can be read as:

> If the current state satisfies the inequality $Bx \leq d$, 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 the soccer rule above. Suppose that “an opponent is near me” is true when a distance feature $f_1 \leq 5$, and that “a teammate is open” is true when an angle feature $f_4 \geq 30$, and that there are five features in the environment. In this case, Equation 4.1 would become Equation 4.2 below. Note that $(f_4 \geq 30)$ is equivalent to $(-f_4 \leq -30)$).

$$\begin{pmatrix} 1 \\ 0 \\ 0 \\ -1 \\ 0 \end{pmatrix}^T \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \end{pmatrix} \leq \begin{pmatrix} 5 \\ 0 \\ 0 \\ -30 \\ 0 \end{pmatrix} \Longrightarrow Q_{pass}(x) - Q_{hold}(x) \geq \beta, \tag{4.2}$$

The KBKR system is designed to work with a standard RL algorithm that uses a fixed-length feature vector. As such, it only accepts rules in propositional logic, not in first-order logic as one of the algorithms in this chapter generates. Thus first-order rules must currently be grounded before being provided to KBKR.

There is also a variant of Preference-KBKR called ExtenKBKR [50] that incorporates advice in a way that allows for faster problem-solving. I will not present this variant in detail here, but I do use it for transfer when there is more advice than Preference-KBKR can efficiently handle.

4.2 Policy Transfer via Advice

Policy transfer via advice taking, my Transfer Algorithm 1, is a transfer method that advises following the source-task policy in the target task. That is, it advises the target-task learner to give the highest *Q*-value to the action that a source-task agent would take in a comparable situation, without specifying exactly what the *Q*-values of the actions should be. This section is based on published work [106].

Policy transfer assumes that a human provides a mapping of features and actions between tasks, so that the *Q*-functions for the source task can be translated to the target task. One possible transfer method would be simply to use these translated *Q*-functions in the target task. However, if the target task has a different reward function than the source task, their *Q*-functions could be quite different even if the actions they recommend are similar. For example, in both KeepAway and BreakAway it is often good to pass the ball if an opponent is too close, but *Q*-values in BreakAway are in the range [0,1] while *Q*-values in KeepAway are in a larger range [0, *maxGameLength*]. Instead of applying potentially inappropriate *Q*-values directly in the target task, policy transfer compares the *Q*-functions for pairs of actions to produce advice rules saying when one action should be preferred over the other.

Table 4.1 gives the algorithm for policy transfer. It requires as input the action and feature spaces of the tasks, a mapping between these spaces, the final source-task *Q*-function for each action, and average observed values for source-task features. It consists of three steps: translating the source-task *Q*-functions into terms usable in the target task, generating advice that compares each pair of translated *Q*-functions and advises preferring the higher-valued action, and learning the target task with this advice as described above in Section 4.1. The sections below describe the first two steps in more detail.

4.2.1 Translating Q-Functions

The first step of the policy-transfer algorithm in Table 4.1 begins with a set of source-task *Q*-functions and ends with a set of translated *Q*-functions that would be usable in the target task. These are needed in order to generate advice that is meaningful for the target task.

The key to this translation is the human-provided mapping ($M_{actions}$, $M_{features}$). First consider the simplest possible type of mapping, in which each source-task action is mapped to one and only one target-task action, and each source-task feature is mapped to one and only one target-task feature. This can be expressed as:

For each $a \in A_s$ there exists a unique $a' \in A_t$ such that $M_{actions}(a) = a'$
For each $f \in F_s$ there exists a unique $f' \in F_t$ such that $M_{features}(f|*) = f'$

Table 4.1: Transfer Algorithm 1: Policy Transfer via Advice

```
INPUT REQUIRED
Source-task actions $A_s = (a_1, a_2, ...)$ and features $F_s = (f_1, f_2, ...)$
Target-task actions $A_t = (b_1, b_2, ...)$ and features $F_t = (g_1, g_2, ...)$
Mappings $M_{actions} : A_s \to A_t$ and $M_{features} : F_s|A_t \to F_t$
Final source-task Q-function for each action $a$: $Q_a = w_1 f_1 + w_2 f_2 + ...$
Average observed value $\bar{f}$ for each source-task feature $f \in F_s$

TRANSLATE Q-FUNCTIONS FROM SOURCE TO TARGET
Let $T = \emptyset$                          // This will be the set of translated Q-functions
For each action $a \in A_s$
    For each action $b$ such that $M_{actions}(a) = b$ // Build a translated Q-function for $b$
        Start with $Q_b = 0$
        For each feature $f_i \in F_s$
            If exists $g_j$ such that $M_{features}(b, f_i) = (b, g_j)$
                Set $Q_b \leftarrow Q_b + w_i g_j$          // Here there is a matching feature
            Else
                Set $Q_b \leftarrow Q_b + w_i \bar{f}_i$          // Here there is no matching feature
        Set $T \leftarrow T \cup Q_b$

GENERATE ADVICE
Let $V = \emptyset$                                // This will be the set of advice rules
For each pair of translated Q-functions $(Q_{b_i}, Q_{b_j}) \in T$
    Let advice rule $R$ be: IF $Q_{b_i} - Q_{b_j} \geq \Delta$ THEN prefer $b_i$ to $b_j$
    Set $V \leftarrow V \cup R$

LEARN TARGET TASK
For all episodes: Perform RL via ExtenKBKR using advice rules $V$
```

The $*$ indicates that the feature mappings hold for all target-task actions; soon I will discuss the use of this argument to allow more complex mappings. For now, with this simple mapping, the translation step is a straightforward substitution of features. For example, assume these are the source-task *Q*-functions:

$$\begin{aligned} Q_{a_1} &= w_1 f_1 + w_2 f_2 \\ Q_{a_2} &= w_3 f_3 + w_4 f_4 + w_5 f_5 \end{aligned}$$

The translated *Q*-functions would then be:

$$\begin{aligned} Q_{a_1'} &= w_1 f_1' + w_2 f_2' \\ Q_{a_2'} &= w_3 f_3' + w_4 f_4' + w_5 f_5' \end{aligned}$$

However, the mappings need not be one-to-one. $M_{actions}(a)$ may have no values, or it may have several values. In the former case, policy transfer creates no translated *Q*-function for a, because no corresponding action exists in the target task. In the latter case, policy transfer creates multiple translated *Q*-functions for a', a'', and so on, because multiple corresponding actions exist in the target task. Furthermore, each of these functions may use different feature substitutions. This is the reason that $M_{features}$ is a function of both a feature and an action; it allows features to be mapped differently under different action mappings.

For example, consider transferring from 2-on-1 BreakAway, which has a single pass action *pass_to_a1*, to 3-on-2 BreakAway, which has two pass actions *pass_to_a1* and *pass_to_a2*. If *pass_to_a1* could be mapped only once, it could only provide information about one of the two target-task pass actions. By allowing multiple mappings, policy transfer can create translated *Q*-functions for both target-task pass actions. For *pass_to_a1*, it should use obvious feature mappings like *distBetween(a0,a1)* $\rightarrow$ *distBetween(a0,a1)*. For *pass_to_a2*, it should instead use feature mappings like *distBetween(a0,a1)* $\rightarrow$ *distBetween(a0,a2)*.

In some cases, a source-task feature f may have no corresponding f'. For example, there is a KeepAway feature distBetween(k0, fieldCenter) that has no meaningful correspondence with any feature in BreakAway. In this case, there is no appropriate substitution for f in the translated *Q*-functions. Simply dropping terms involving this feature could change the *Q*-functions significantly, since they are linear sums. Instead, policy transfer substitutes the average value of f as observed in the source task, which better maintains the integrity of the *Q*-function.

The algorithm does require that $M_{actions}$ not map multiple source-task actions to the same target-task action, because this would produce multiple translated *Q*-functions for a single target-task action, making the *Q*-values for that action ill-defined.

4.2.2 Generating Advice

The second step of the policy-transfer algorithm in Table 4.1 starts with the translated *Q*-functions and generates advice. For each pair of target-task actions (b_i, b_j), it forms an advice rule that states when to prefer b_i to b_j. Table 4.1 shows the high-level rule format:

IF $Q_{b_i} - Q_{b_j} \geq \Delta$
THEN prefer b_i to b_j

Suppose the translated *Q*-functions are as in the previous section:

$$Q_{a'_1} = w_1 f'_1 + w_2 f'_2$$
$$Q_{a'_2} = w_3 f'_3 + w_4 f'_4 + w_5 f'_5$$

Given these, the final advice format would be:

IF $(w_1 f'_1 + w_2 f'_2) - (w_3 f'_3 + w_4 f'_4 + w_5 f'_5) \geq \Delta$
THEN prefer a'_1 to a'_2

This advice says to prefer b_i to b_j in states where the translated *Q*-functions give a significantly higher value to b_i. Essentially, this means preferring actions that a source-task agent would take if it found itself in the target task. I set Δ to approximately 1% of the target task's Q-value range so that the advice does not apply when values are very similar.

4.2.3 Experimental Results for Policy Transfer

To test the policy-transfer approach, I perform transfer between several RoboCup tasks in both *close transfer* and *distant transfer* scenarios. Close transfer is between closely related tasks, such as KeepAway with different team sizes. I perform close-transfer experiments for all three RoboCup

Table 4.2: Results of statistical tests on the differences between areas under the curve in policy transfer vs. RL-SVR for several source/target pairs. For $p < 0.05$, the difference is significant and the winning algorithm is shown; otherwise, the difference is not statistically significant.

Source	*Target*	*p-value*	*Significance (Winner)*	*95% interval*
2-on-1 BreakAway	3-on-2 BreakAway	0.001	Yes (Policy transfer)	[38, 160]
3-on-2 MoveDownfield	3-on-2 BreakAway	0.402	No	[-53, 70]
3-on-2 KeepAway	3-on-2 BreakAway	0.030	Yes (Policy transfer)	[1, 121]
3-on-2 MoveDownfield	4-on-3 MoveDownfield	0.020	Yes (Policy transfer)	[118, 2510]
3-on-2 KeepAway	4-on-3 KeepAway	0.004	Yes (Policy transfer)	[90, 510]

tasks. Distant transfer is between less similar tasks, such as KeepAway and BreakAway. I perform distant-transfer experiments from the easier RoboCup tasks, KeepAway and MoveDownfield, to the more difficult task of BreakAway. The mappings I use for these scenarios are documented in Appendix B.

Figures 4.1, 4.2, and 4.3 show the performance of policy transfer compared to RL-SVR. These results show that policy transfer can have a small overall positive impact in both close and distant transfer scenarios. Policy-transfer curves converge with RL-SVR by the end of the learning curve.

The statistical analysis in Table 4.2 indicates that the difference between policy transfer and RL-SVR is significant in most cases. However, the figures show that the difference is too small to be practically significant.

Policy transfer produces a large amount of complex advice – so much that it needs to use ExtenKBKR [50], the variant of Preference-KBKR that handles high advice volumes. Furthermore, this method relies on the low-level, task-specific *Q*-functions to perform transfer. Other methods in this thesis instead transfer relational knowledge, leading to larger performance gains.

4.3 Skill Transfer via Advice

Skill transfer via advice, my Transfer Algorithm 2, is a transfer method that provides advice about when to take certain actions. It is designed to capture general knowledge from the source task and filter out specific knowledge. Instead of transferring an entire policy, this method only

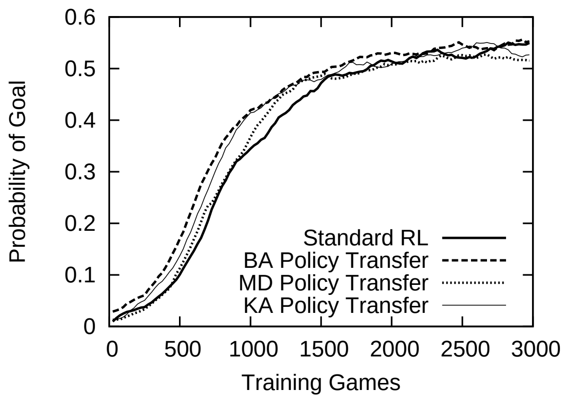

Figure 4.1: Probability of scoring a goal in 3-on-2 BreakAway with RL-SVR and policy transfer from 2-on-1 BreakAway (BA), 3-on-2 MoveDownfield (MD) and 3-on-2 KeepAway (KA).

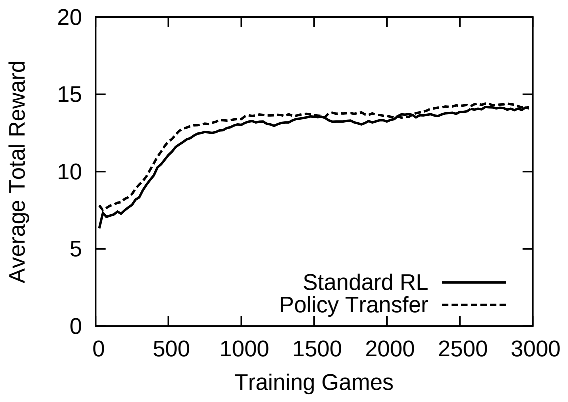

Figure 4.2: Average total reward in 4-on-3 MoveDownfield with RL-SVR and policy transfer from 3-on-2 MoveDownfield.

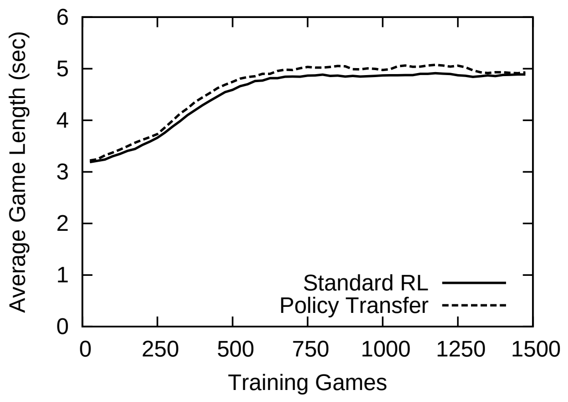

Figure 4.3: Average game length in 4-on-3 KeepAway with RL-SVR and policy transfer from 3-on-2 KeepAway.

transfers advice about the *skills* that the source and target tasks have in common. This section is based on published work [103, 105].

Skills are rules in first-order logic that describe good conditions under which to take an action. I use first-order logic for these rules because variables allow them to be more general than propositional rules. 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. These common skill elements can transfer better to new tasks.

Table 4.3 gives the algorithm for skill transfer. It requires as input the states encountered during source-task learning, the final source-task *Q*-function for each action, a mapping between objects in the two tasks, a list of skills that should be transferred, the desired outcomes of actions, and (optionally) an extra set of human-provided advice. It consists of three steps: learning skills, translating them into advice usable in the target task, and learning the target task with this advice as described above in Section 4.1. The sections below describe the first two steps in more detail.

4.3.1 Learning Skills

The first step of the skill-transfer algorithm in Table 4.3 learns skills that describe when one action is preferable to other actions based on source-task examples. Skills may represent both grounded actions, like *pass(a1)*, and variablized actions, like *pass(Teammate)*.

To learn skills, the algorithm requires positive and negative examples of action preferences. In some source-task states no action is strongly preferred, either because multiple actions are good or because no actions are good. The algorithm prevents these states from being used as examples because of their ambiguity.

In positive examples for action a, of course a must be the action taken. The algorithm also requires that a was strongly preferred, by enforcing reasonable *Q*-value thresholds as shown in Table 4.3. Furthermore, it requires that the outcome of action a was the desired outcome, as defined in the source task. For example, the desired outcome of *pass(a1)* in BreakAway is that the player $a1$ gains possession of the ball.

Table 4.3: Transfer Algorithm 2: Skill Transfer via Advice

```
INPUT REQUIRED
 States $S$ from the source-task learning process
 Actions $A$ for which skills should be transferred and the final $Q_a$ for each $a \in A$
 The desired source-task outcome $desired(a)$ for each action $a \in A$
 Mapping $M$ from source-task objects to target-task objects
 Optional human-provided advice rules $H$

LEARN SKILLS
 Let $K = \emptyset$                         // This will be the set of rules describing skills
 For each action $a \in A$
   Let $P_a = \emptyset$                           // These will be positive examples for $a$
   Let $N_a = \emptyset$                           // These will be negative examples for $a$
   Let $avg(a)$ be the average value of $Q_a(s)$ over all states in $s \in S$
   Let $tenth(a)$ be the tenth percentile of $Q_a(s)$ over all states in $s \in S$
   For each state $s \in S$
     If the action $b$ taken in $s$ is equal to $a$
       If the outcome of $b$ was $desired(a)$
         If $Q_a(s) > tenth(a)$ and $\forall c \neq a \;\; Q_a(s) > 1.05 \times Q_c(s)$
           Set $P_a \leftarrow P_a \cup s$                              // Positive example
         Else reject $s$ for ambiguity
       Else set $N_a \leftarrow N_a \cup s$                        // Negative example type 1
     Else if $Q_b(s) > tenth(b)$ and $Q_a(s) < 0.95 \times Q_b(s)$ and $Q_a(s) < avg(a)$
       Set $N_a \leftarrow N_a \cup s$                             // Negative example type 2
     Else reject $s$ for ambiguity
   Learn rules with Aleph and Gleaner to distinguish $P_a$ from $N_a$
   Let $R$ be the rule with highest $F(1)$ score
   Set $K \leftarrow K \cup R$

TRANSLATE SKILLS
 For each skill $R \in K$                                                   // Map objects
   For each source-task object $o_s$ appearing in $R$
     Replace $o_s$ with the mapped target-task object $M(o_s)$
 For each human-provided advice rule $I \in H$          // Add human-provided advice
   If a rule $R \in K$ represents the same skill as $I$
     Add the literals in $I$ to the rule $R$
   Else set $K \leftarrow K \cup I$
 Let $V = \emptyset$                                // This will be the set of advice rules
 For each skill $R \in K$
   Let $a$ be the action represented by $R$ and let $L$ be the literals in the body of $R$
   For each target-task action $b \neq a$
     Let advice rule $T$ be: IF $L$ THEN prefer $a$ to $b$
     Set $V \leftarrow V \cup T$

LEARN TARGET TASK
 For all episodes: Perform RL via Pref-KBKR using advice rules $V$
```

Negative examples for action a have two types. The first are states in which a was the action taken but the desired outcome did not occur (e.g., an opponent gained possession of the ball instead of $a1$). The second are states in which another action b was taken and was strongly preferred over a, again using reasonable *Q*-value thresholds as shown in Table 4.3.

The skill-transfer algorithm uses Aleph [80] to search for first-order logical rules that approximately separate positive examples from negative examples. It uses Gleaner [35] to record a variety of rules encountered during the search, and in the end selects one rule with the highest $F(1)$ score. These methods are described in Section 2.3, with only the necessary additional details provided here.

Each legal literal for this search specifies a constraint on a source-task feature. In order to ensure that the rules are applicable in the target task, the literals may only give constraints on features that also exist in the target task. For example, the MoveDownfield task has a feature distToRightEdge(Attacker). This feature can be used in rules for transfer from 3-on-2 MoveDownfield to 4-on-3 MoveDownfield, but not for transfer from MoveDownfield to BreakAway, since BreakAway has no such feature.

Note that for real-valued, continuous features, the possible constraints must be limited to a finite set through discretization. I approach this problem by allowing constraints that say a feature is greater than or less than one of a set of threshold values. For RoboCup tasks, I set thresholds for distance features at every 3 yards and thresholds for angle features at every 5 degrees. Thus, for example, possible literals involving distToRightEdge(Attacker) are:

distToRightEdge(Attacker) ≤ 3
distToRightEdge(a0) ≥ 6
distToRightEdge(a1) ≤ 9
distToRightEdge(a2) ≥ 12
...

Here is an example of skill learning in practice. For transfer from 3-on-2 MoveDownfield to 4-on-3 MoveDownfield, the skill-transfer algorithm learns the following rule for the *pass(Teammate)* skill:

```
IF      distBetween(a0, Teammate) $\geq 15$
AND     distBetween(a0, Teammate) $\leq 27$
AND     distToRightEdge(Teammate) $\leq 10$
AND     angleDefinedBy(Teammate, a0, minAngleDefender(Teammate)) $\geq 24$
AND     distBetween(a0, Opponent) $\geq 4$
THEN    pass(Teammate)
```

This rule states that passing to a teammate is preferable if (1) the teammate is between 15 and 27 yards away, (2) the teammate is within 10 yards of the finish line, (3) the teammate has an open passing angle of at least 24 degrees, and (4) no opponent is closer than 4 yards away.

4.3.2 Translating Skills and Adding Human-Provided Advice

The second step of the skill-transfer algorithm in Table 4.3 translates skills into advice for the target task. Just as in policy transfer, this requires a human-provided mapping M. However, the nature of the mapping is different than in policy transfer. Since only shared actions are transferred and only shared features are used to describe skills, the only elements that require mapping are logical objects, such as the player objects in RoboCup. For example, *k0* in KeepAway should map to *a0* in BreakAway, and a variable representing *Keeper* objects should map to a variable representing *Attacker* objects.

Thus a human teacher provides a mapping to facilitate the translation of source-task skills for use in the target task. However, note that there is important information that such a mapping lacks: knowledge about new skills in the target task. It also omits knowledge about how shared skills might differ between the tasks. A human teacher might have these types of knowledge, and the advice-taking system provides a way to accommodate it. The skill-transfer method therefore accepts optional human-provided advice to be combined with the skills it transfers automatically.

For example, consider transferring from KeepAway to BreakAway, where the only shared skill is *pass(Teammate)*. As learned in KeepAway, this skill will make no distinction between passing toward the goal and away from the goal. Since the new objective is to score goals, players should prefer passing toward the goal. A human could provide this guidance by adding an additional constraint to the *pass(Teammate)* skill:

distBetween(a0, goal) - distBetween(Teammate, goal) ≥ 1

Even more importantly, there are several actions in this transfer scenario that are new in the target task, such as *shoot* and *moveAhead*. A human could write simple rules to approximate these:

```
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
```

Given a combined set of learned and provided skills, the algorithm in Table 4.3 forms final advice rules. Each skill generates advice rules saying when to prefer one action over all the others.

4.3.3 Experimental Results for Skill Transfer

To test the skill-transfer approach, I perform transfer between several RoboCup tasks. I present results from the same close and distant transfer scenarios as for policy transfer in Section 4.2.3. The mappings I use for these scenarios are documented in Appendix C.

I use appropriate subsets of the user-advice examples in Section 4.3.2 for all of these experiments. That is, from KeepAway to BreakAway I use all of it, from MoveDownfield to BreakAway I use only the parts advising *shoot*, and for close-transfer experiments I use none.

Figures 4.4, 4.5, and 4.6 show the performance of skill transfer compared to RL-SVR and policy transfer. These graphs show that skill transfer can have a substantial positive impact in both close-transfer and distant-transfer scenarios.

The statistical analysis in Table 4.4 indicates that the difference between skill transfer and policy transfer is significant in some cases. The figures show that this difference also has more practical significance.

Skill transfer has several advantages over policy transfer. It produces fewer items of advice, allowing the use of standard Preference-KBKR instead of ExtenKBKR, the high-volume version. The advice it transfers is human-readable, and therefore also allows users to edit and add to it with ease. Finally, it produces significantly higher performance gains than policy transfer. This result is

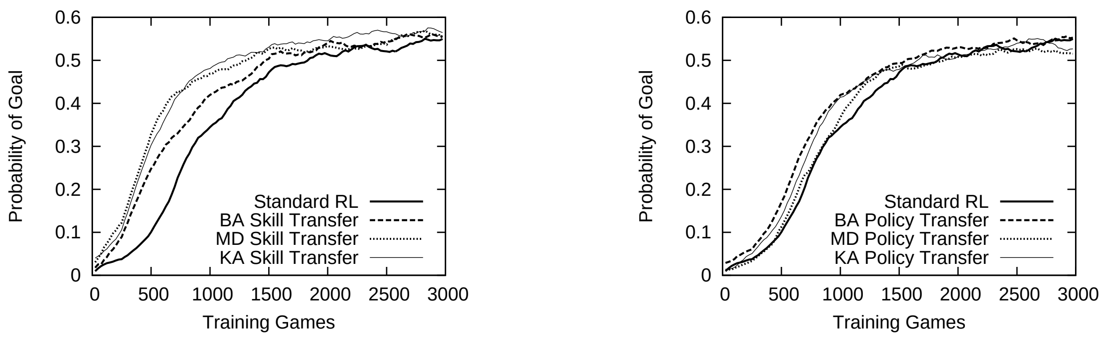

Figure 4.4: Left: Probability of scoring a goal 3-on-2 BreakAway with RL-SVR and skill transfer from 2-on-1 BreakAway (BA), 3-on-2 MoveDownfield (MD) and 3-on-2 KeepAway (KA). Right: Figure 4.1 reproduced to show the corresponding policy-transfer results.

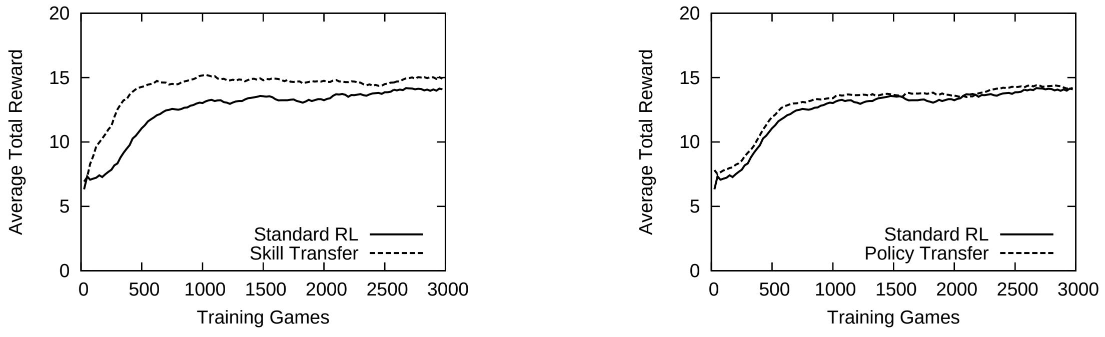

Figure 4.5: Left: Average total reward in 4-on-3 MoveDownfield with RL-SVR and skill transfer from 3-on-2 MoveDownfield. Right: Figure 4.2 reproduced to show the corresponding policy-transfer results.

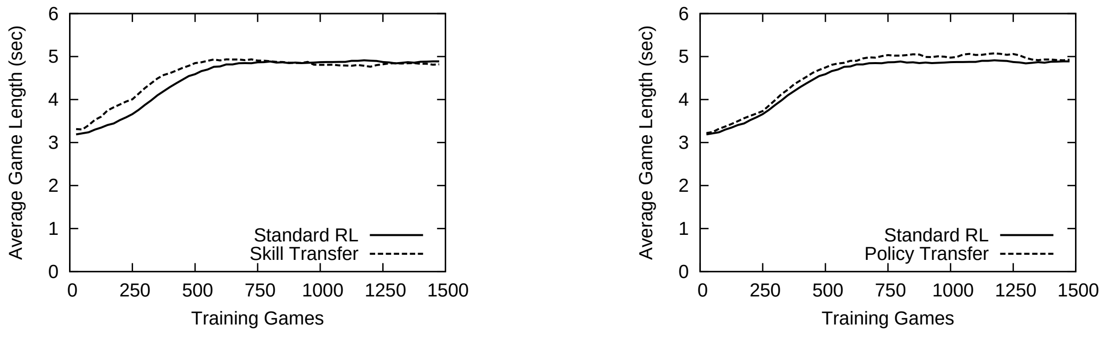

Figure 4.6: Left: Average game length in 4-on-3 KeepAway with RL-SVR and skill transfer from 3-on-2 KeepAway. Right: Figure 4.3 reproduced to show the corresponding policy-transfer results.

Table 4.4: Results of statistical tests on the differences between areas under the curve in skill transfer vs. policy transfer for several source/target pairs. For $p < 0.05$, the difference is significant and the winning algorithm is shown; otherwise, the difference is not statistically significant.

Source	*Target*	*p-value*	*Significance (Winner)*	*95% interval*
2-on-1 BreakAway	3-on-2 BreakAway	0.134	No	[-25, 91]
3-on-2 MoveDownfield	3-on-2 BreakAway	< 0.001	Yes (Skill transfer)	[155, 258]
3-on-2 KeepAway	3-on-2 BreakAway	< 0.001	Yes (Skill transfer)	[128, 225]
3-on-2 MoveDownfield	4-on-3 MoveDownfield	< 0.001	Yes (Skill transfer)	[2250, 5230]
3-on-2 KeepAway	4-on-3 KeepAway	0.118	No	[-421, 95]

the primary basis for my claim that relational transfer is desirable for RL. The rest of the transfer methods in this thesis are also relational.

4.4 Testing the Boundaries of Skill Transfer

In this section I consider the impact of two factors on the effectiveness of skill transfer: quality of learning in the source task, and quality of human-provided advice.

So far I have performed transfer without paying any attention to the source-task learning curve. However, it might be interesting to know if some types of source-task learning curves produce better or worse transfer. If one had a choice of source runs to choose from, one could then choose a run with high expected target-task performance.

Figure 4.7 plots the normalized average area under the curve in the target task with skill transfer against the normalized area under the curve in the source task from which transfer was performed. The correlation coefficient is 0.21, which indicates a small correlation between source-task and target-task area. Therefore it may be only slightly helpful to choose source runs with higher area under the curve.

The second factor I consider is how the quality of human-provided advice affects skill transfer. So far I have simply given reasonable, non-optimized advice for new skills in skill-transfer experiments. Now I investigate the results produced using reasonable variants that another person could easily have chosen instead.

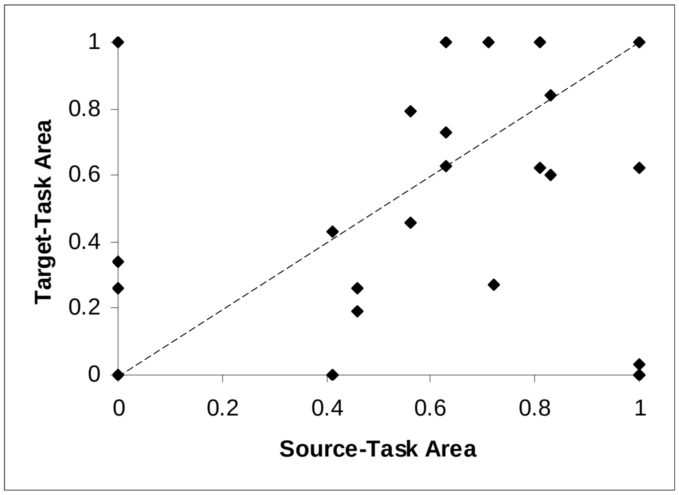

Figure 4.7: A plot showing how area under the learning curve in the target task correlates with area under the learning curve in the source task. The dashed line shows a 45-degree angle for visual reference.

To look at one example, I examine skill transfer from KeepAway to BreakAway with several variants on the original human-provided advice. Variant 1 encourages the players to shoot more often by increasing the maximum distance and decreasing the minimum angle:

IF	distBetween(a0, GoalPart) < 15
AND	angleDefinedBy(GoalPart, a0, goalie) > 35
THEN	prefer shoot(GoalPart) over all actions
IF	distBetween(a0, goalCenter) > 15
THEN	prefer moveAhead over moveAway and shoot
Add to pass(Teammate):	diffGoalDistance(Teammate, Value), Value ≥ 1

Variant 2 prevents the players from shooting as often by decreasing the maximum distance and increasing the minimum angle:

IF	distBetween(a0, GoalPart) < 5
AND	angleDefinedBy(GoalPart, a0, goalie) > 45
THEN	prefer shoot(GoalPart) over all actions
IF	distBetween(a0, goalCenter) > 5
THEN	prefer moveAhead over moveAway and shoot
Add to pass(Teammate):	diffGoalDistance(Teammate, Value), Value ≥ 1

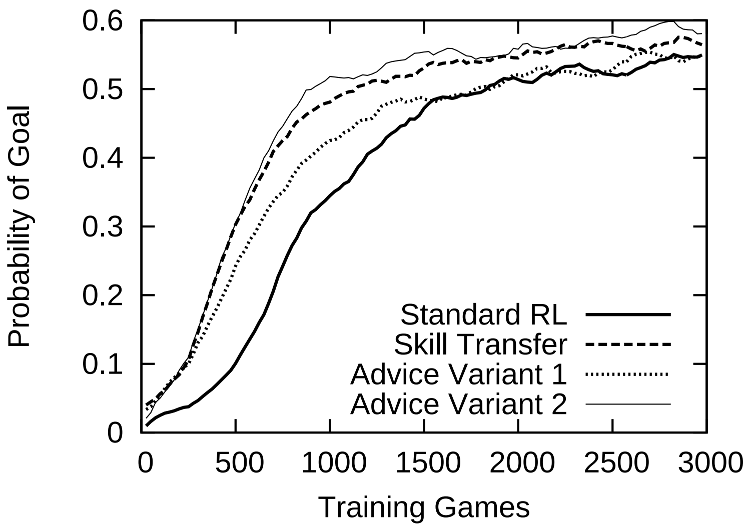

Figure 4.8: Probability of scoring a goal 3-on-2 BreakAway, with RL-SVR and with skill transfer from 3-on-2 KeepAway, using the original human-provided advice and using two variants described above.

Figure 4.8 indicates that these changes in the human-provided advice do affect the performance of skill transfer, but that reasonable variants still do well. This robustness means that users need not worry about providing perfect advice in order for the skill-transfer method to work. Furthermore, even approximate advice can significantly improve the performance.

One might also wonder about the effect of having no human-provided advice at all. To look at one example, I examine transfer from MoveDownfield to BreakAway without any extra advice. Figure 4.9 shows that skill transfer still performs significantly better than RL-SVR even without human-provided advice, although the gain is smaller. The addition of human-provided advice produces another significant gain. This means that while it is not necessary to provide extra advice in order for the skill-transfer method to work, doing so can be worthwhile.

4.5 Summary of Advice-Based Transfer

Transfer via advice can produce performance gains in reinforcement learning. In my experiments, giving a few pieces of relational advice produced better performance than giving many

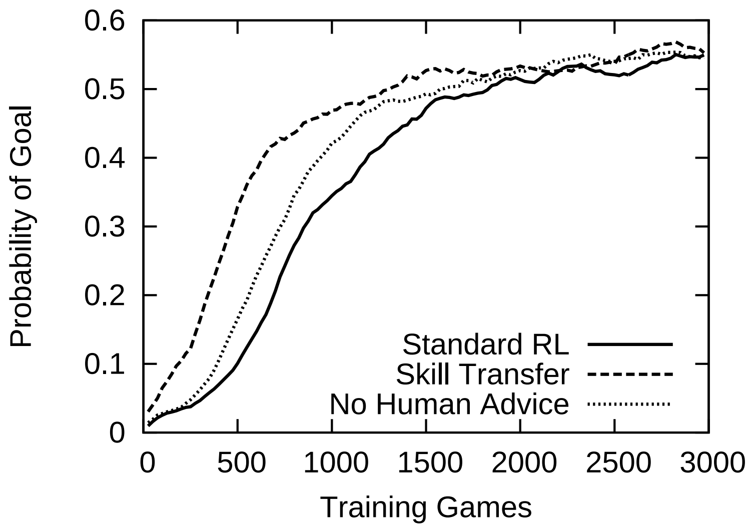

Figure 4.9: Probability of scoring a goal in 3-on-2 BreakAway, with RL-SVR and with skill transfer from 3-on-2 MoveDownfield, with and without human-provided advice.

pieces of propositional advice. I also found simple human-provided advice to produce additional benefits.

Advice-based transfer has both advantages and disadvantages compared to other transfer methods described in Chapter 3. The most notable advantage is its built-in protection against negative transfer, which allows it to be applied to a wide range of transfer scenarios. The most notable disadvantage is that the performance gains do not appear immediately and are relatively modest. Later methods in this thesis will sacrifice protection against negative transfer in order to produce larger performance gains in certain transfer scenarios.

The methods in this chapter are limited to transferring information about individual decisions in a task. That is, they provide knowledge about the relative values of actions at single moments in time. The next chapter presents methods that go a step further and transfer information about *sequences* of decisions.

Chapter 5

Macro-Operator Transfer

Reinforcement learning is often described as solving a Markov decision process [6]. Typically it is a first-order decision process, meaning that decisions are made with respect to the current environment, without reference to previous actions or previous environments. However, if one observes successful RL agents after they have learned tasks like BreakAway, one can see common action sequences that they have learned to execute. These sequences are emergent behavior; the agents do not intentionally execute patterns of actions, but in many tasks patterns spontaneously appear.

This observation begs the question of whether transferred knowledge could contain more structure than the rules in Chapter 4. Perhaps transferring information about action sequences rather than about individual actions could produce faster success in a target task. The transfer of structured knowledge could be a powerful method, though it would come with a price: it would be less applicable to distant-transfer scenarios, since action sequences are likely to be common only among closely related tasks.

In the planning field, action sequences are known as *macro-operators* [8, 12]. Learning action sequences is an example of *structured learning*, a branch of machine learning in which the goal is prediction of complex structured outputs rather than numeric values. The structures represent sets of output variables that have mutual dependencies or constraints. Other examples of structured learning include learning parse trees for natural-language utterances [37] and learning alignments for gene or protein sequences [90].

I use macro-operators to perform transfer by providing action-sequence knowledge from a source task to a target task. Because I have already established the value of relational knowledge by comparing skill transfer to policy transfer in Chapter 4, I design these structures to capture relational information. I refer to them in shorthand as *relational macros* or just *macros*.

A relational macro describes a strategy that is successful in the source task. There are several ways I could use this information to improve learning in a related target task. One possibility is to treat it as advice, as in Chapter 4, or as an option with its own Q-value, as in some methods reviewed in Chapter 3. The primary benefit of these approaches is their protection against negative transfer.

Instead, I introduce a method called *demonstration*, in which the target-task agent executes the transferred strategy for an initial period before continuing with RL-SVR. The benefit of this approach is that it can achieve good target-task performance very quickly. The disadvantage is that it is more aggressive, carrying more risk for negative transfer. However, if the tasks are closely related, the potential benefits can outweigh that risk.

This chapter introduces relational macros and the demonstration method, and then presents two methods for macro transfer. One method transfers a single macro, and the other transfers multiple macros. This chapter is based in part on published work [104].

5.1 Relational Macro-Operators and Demonstration

This section describes the structure of a relational macro and its use, via demonstration, in the target task. The following sections describe the problem of learning a macro from source-task data.

The purpose of a macro is to serve as a decision-control mechanism for a demonstration of good behavior. Therefore, macros are based on finite-state machines [34]. A finite-state machine (FSM) models a control process in the form of a directed graph. The nodes of the graph represent states of the system, and in this 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 (in this case, which *Teammate*

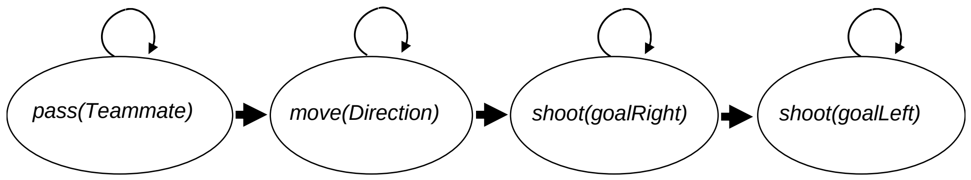

Figure 5.1: A macro for the RoboCup game BreakAway, in which the agent attempts to score a goal. Each node and arc has an attached ruleset (not shown) for choosing actions and deciding transitions.

argument). An FSM 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.

Formally, a macro consists of:

- A linear set of nodes $N = (n_1, n_2, ...)$ with associated rulesets $R_{action}(n_i)$
- Arcs between adjacent nodes with associated rulesets $R_{trans}(n_i, n_{i+1})$
- Arcs that form self-loops for each node

The nodes form a linear structure, as shown in Figure 5.1. An agent executing a macro begins in node n_1, which in the figure is the *pass(Teammate)* node. In the first state of an episode, it uses the ruleset $R_{action}(n_1)$ to choose an action. In the second state of the episode, it uses the ruleset $R_{trans}(n_1, n_2)$ to decide whether to transition to node n_2 or default to the self-arc to remain in n_1; then it uses the appropriate R_{action} ruleset to choose an action. This continues until either the episode ends or an R_{action} ruleset fails to choose an action, in which case the agent abandons the macro and uses its current *Q*-function to choose actions instead.

Since a ruleset for a node or arc can contain an arbitrary number of rules, there are disjunctive conditions for decisions in a macro. When multiple rules match, the agent obeys the rule that has the highest score. In my design, the score of a rule is the probability that following it will lead to a successful game, as estimated from the source-task data.

Target-task agents use macros this way for an initial *demonstration period*. The length of this period is a task-dependent design decision. For RoboCup tasks, I found 100 games to be an appropriate length.

During the demonstration, reinforcement learning does occur, with normal updates to the *Q*-functions. In RL-SVR, this means re-learning *Q*-functions as usual after each batch of 25 games.

However, the *Q*-functions only start being used after the demonstration period ends, at which point the target-task agent reverts to standard RL-SVR. If transfer is effective, its performance level at that point is much higher than it would have been after 100 episodes of nearly random exploration.

5.2 Single-Macro Transfer via Demonstration

Single-macro transfer via demonstration, my Transfer Algorithm 3, is a transfer method that learns one relational macro from a source task, and allows target-task agents to follow the macro for an initial demonstration period. This allows the target-task agents to avoid the slow process of random exploration that traditionally occurs at the beginning of RL. This section is based on published work [104].

The macro-transfer algorithm uses Aleph [80] and Gleaner [35] to construct a relational macro from source-task data. Furthermore, it separates the problem of learning a macro into several independent ILP problems. One is to learn the structure (the node sequence), while others learn rulesets to govern transitions between nodes and action-argument choices.

This is not the only possible approach; some or all of these problems could be combined, requiring a joint solution. However, doing so would significantly increase the run time required for ILP to produce accurate solutions. For this practical reason, I treat each decision problem independently.

Table 5.1 gives the algorithm for single-macro transfer. It requires as input the games played during source-task learning, a definition of which games are "good" and which are "bad," and the length of the demonstration period. It consists of four steps: learning the structure, learning rulesets to govern transitions between nodes, learning rulesets to govern actions taken in variablized nodes, and learning the target task as described above in Section 5.1. The sections below describe the first three steps in more detail, and they are also illustrated in Figure 5.2.

5.2.1 Single-Macro Structure Learning

The first step of the macro-transfer algorithm in Table 5.1 is the structure-learning phase. The objective is to find a sequence of actions that distinguishes successful games from unsuccessful

Table 5.1: Transfer Algorithm 3: Single-Macro Transfer via Demonstration

```
INPUT REQUIRED
 Games $G$ from the source-task learning process
 A definition of high-reward and low-reward games in the source task
 A demonstration-period length $D$

LEARN STRUCTURE
 Learn rules $R$ according to Table 5.2
 Let $L$ be the set of Gleaner rules with the maximal number of literals
 Let $S$ be the rule in $L$ with the highest $F(1)$ score // This is the chosen macro structure
 Let $Macro = \emptyset$                         // This will consist of nodes and rulesets
 For each literal in $S$ representing $(Action, ActionArg)$
     Create a node $n$ of type Action(ActionArg)
     Add node $n$ to $Macro$

LEARN TRANSITIONS
 For each adjacent pair of nodes $n_i, n_{i+1}$ of $Macro$
     Learn rules $T$ according to Table 5.3
     Select rules $R_{trans}$ from $T$ according to Table 5.5
     Attach $R_{trans}$ to $Macro$ as the transition ruleset for nodes $n_i, n_{i+1}$

LEARN ACTIONS
 For each node $n$ of $Macro$ that represents a variablized Action(ActionArg)
     Learn rules $U$ according to Table 5.4
     Select rules $R_{action}$ from $U$ according to Table 5.5
     Attach $R_{action}$ to $Macro$ as the action ruleset for node $n$

LEARN TARGET TASK
 For $D$ episodes: Perform RL but use $Macro$ to choose actions
 For remaining episodes: Perform RL normally
```

Table 5.2: Algorithm for learning one or more macro structures, given games G from the source task.

```
Let $P = \emptyset$                 // These will be the positive examples
Let $N = \emptyset$                 // These will be the negative examples
For each game $g \in G$
    If $g$ is a high-reward game in the source task
        Set $P \leftarrow P \cup g$      // Positive examples are good games
    Else if $g$ is a low-reward game in the source task
        Set $N \leftarrow N \cup g$      // Negative examples are bad games
Learn rules $R$ with Aleph and Gleaner to distinguish $P$ from $N$
Return $R$
```

Table 5.3: Algorithm for learning rules to control the transition between a pair of nodes n_i, n_{i+1} in a macro $Macro$, given games G from the source task. Data for this task can come from any game that contains the sequence represented by $Macro$ or some prefix or suffix of that sequence. For example, in a macro with nodes (n_1, n_2, n_3, n_4), prefixes include games matching (n_1, n_2) and suffixes include games matching (n_2, n_3, n_4).

```
Let $P = \emptyset$                                            // These will be the positive examples
Let $N = \emptyset$                                            // These will be the negative examples
For each game $g \in G$ that matches $Macro$ or a prefix or suffix
  If $g$ is a high-reward game in the source task
    Let $x$ be the state in $g$ after the transition $n_i \rightarrow n_{i+1}$
    Set $P \leftarrow P \cup x$                                // A state that transitions is positive
    Let $Y$ be any states in $g$ after the transition $n_{i-1} \rightarrow n_i$ but before the transition $n_i \rightarrow n_{i+1}$
      Set $N \leftarrow N \cup Y$                              // States that loop are negative
  Else if $g$ is a low-reward game in the source task
    Let $z$ be the state in $g$ after the transition $n_i \rightarrow n_{i+1}$
    If $z$ ends the game
      Set $N \leftarrow N \cup z$                              // A state that ends the game early is negative
  Learn rules $R$ with Aleph and Gleaner to distinguish $P$ from $N$
  Return $R$
```

Table 5.4: Algorithm for learning rules to control the action choice in a node n of a macro that represents the action $Action$ with the variable argument $ActionArg$, given games G from the source task. Data for this task can come from any game, and most comes from those that contain the sequence represented by $Macro$ or some prefix or suffix of that sequence. For example, in a macro with nodes (n_1, n_2, n_3, n_4), prefixes include games matching (n_1, n_2) and suffixes include games matching (n_2, n_3, n_4).

```
Let $P = \emptyset$                                            // These will be the positive examples
Let $N = \emptyset$                                            // These will be the negative examples
For each game $g \in G$
  If $g$ is a high-reward game in the source task and $g$ matches $Macro$ or a prefix or suffix
    Let $X$ be the $ActionArg$ choices taken for any states in node $n$ of $g$
    Set $P \leftarrow P \cup X$                                // An action taken in a good game is positive
    Let $Y$ be the $ActionArg$ choices *not* taken for any states in node $n$ of $g$
    Set $N \leftarrow N \cup Y$                                // An action not taken in a good game is negative
  Else if $g$ is a low-reward game in the source task and $g$ ends with $Action$
    Let $z$ be the $ActionArg$ choice at the end of $g$
    Set $N \leftarrow N \cup z$                                // An action that ends the game early is negative
  Learn rules $R$ with Aleph and Gleaner to distinguish $P$ from $N$
  Return $R$
```

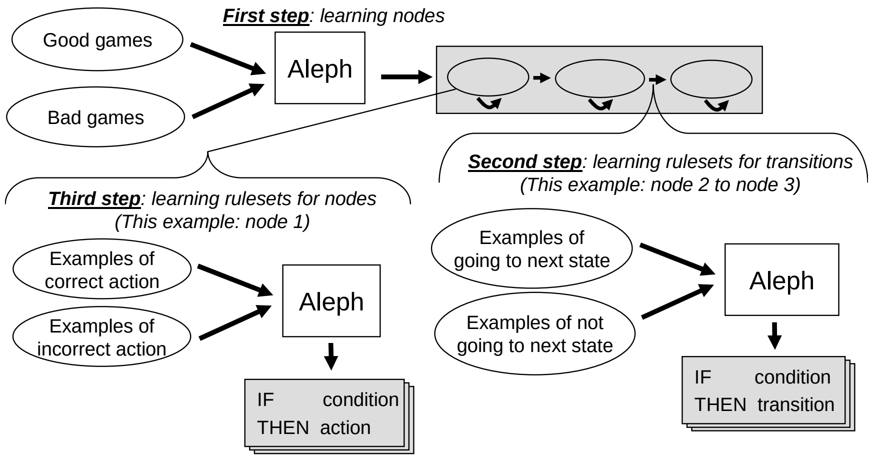
First step: learning nodes
Good games
Bad games
Aleph
Second step: learning rulesets for transitions
(This example: node 2 to node 3)
Third step: learning rulesets for nodes
(This example: node 1)
Examples of correct action
Examples of incorrect action
Aleph
IF condition
THEN action
Examples of going to next state
Examples of not going to next state
Aleph
IF condition
THEN transition

Figure 5.2: A visual depiction of the first three steps of the single-macro transfer algorithm in Table 5.1.

games in the source task. The sequence captures only an action pattern, ignoring properties of states that control transitions between nodes; these are added in later steps. It performs this learning task using all the games from the source-task learning process.

The details of the ILP task for this step are as follows. Let the literal *actionTaken(G,* S_1*, A, R,* S_2*)* denote that action A with argument R was taken in game G at step S_1 and repeated until step S_2. The algorithm asks Aleph to construct a clause *macroSequence(G)* with a body that contains a combination of only these literals. The first literal may introduce two new variables, S_1 and S_2, but the rest must use an existing variable for S_1 while introducing another new variable S_2. In this way Aleph finds a connected sequence of actions that translates directly to a linear node structure.

The algorithm provides Aleph with sets of positive and negative examples, where positives are games with high overall reward and negatives are games with low overall reward. In BreakAway, for example, this is a straightforward separation of scoring and non-scoring games. For tasks with more continuous rewards, it requires finding upper and lower percentiles on the overall reward acquired during a game.

The algorithm uses Gleaner to save the best clauses that Aleph encounters during its search that have a minimum of 50% precision and 25% recall. These requirements are high enough to ensure that the structure it chooses is valuable and not unusual, but low enough to allow it

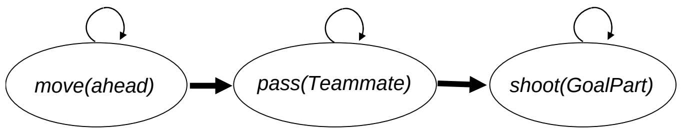

Figure 5.3: The node structure corresponding to the sample macro clause below.

to consider many structures. After this stage, it will improve the precision of the chosen macro through ruleset-learning.

Given the Gleaner output, the algorithm needs to choose a single structure. The problem here is that the structures with the highest $F(1)$ scores tend to be rather short; they classify the source-task games well, but they do not provide an entire strategy for a target-task game. The objective here is not just to discriminate between source-task games, but to generate a good strategy for future games. Since the scores will be improved in the ruleset-learning stage, the algorithm compromises here and chooses the longest macro structure that meets the Gleaner requirements. If there are multiple longest structures, it takes the one with the highest $F(1)$ score.

To make the structure-learning task more concrete, suppose that the scoring BreakAway games consistently look like these examples:

Game 1: move(ahead), pass(a1), shoot(goalRight)
Game 2: move(ahead), move(ahead), pass(a2), shoot(goalLeft)

Assuming that the non-scoring games have different patterns than the examples above do, Aleph might learn the following clause to characterize a scoring game:

IF	actionTaken(Game, StateA, move, ahead, StateB)
AND	actionTaken(Game, StateB, pass, Teammate, StateC)
AND	actionTaken(Game, StateC, shoot, GoalPart, gameEnd)
THEN	macroSequence(Game)

The macro structure corresponding to this sequence is shown in Figure 5.3. The policy in the first node will be to take a single action, *move(ahead)*. In the second node the policy will be to consider multiple *pass* actions, and in the third node the policy will be to consider multiple *shoot* actions. The conditions for choosing an action, and for taking transitions between nodes, are learned next.

5.2.2 Single-Macro Ruleset Learning

The second and third steps of the macro-transfer algorithm in Table 5.1 are for ruleset learning. The objective is to learn control rules for navigating within the macro (i.e. when to transition between nodes, and which argument to choose in a variablized node). Each control decision is based on the RL agent's environment when it reaches that point in the macro.

The details of the ILP tasks for this step are as follows. The legal literals are the same ones described for skill transfer in Section 4.3.1. To describe the conditions on state S under which a transition should be taken, Aleph must construct a clause *transition(S)* with a body that contains a combination of only these literals. To describe the conditions under which an action argument should be chosen, Aleph must construct a clause *action(S, Action, ActionArgs)* with a similarly constituted body.

Aleph may learn some action rules in which the action arguments are grounded, as well as rules in which the action arguments remain variablized: *action(S, move, ahead)* is an example of the former, while *action(S, pass, Teammate)* is an example of the latter. In the case of the *move* action in BreakAway the action arguments in a rule are always grounded, since the original state features do not include useful references to move directions. Note that it is still possible to have a state *move(Direction)* for taking multiple move actions, but the rules for choosing a grounded move action will use only grounded arguments. Rules for *pass* and *shoot* may use either grounded or variable arguments.

The algorithm provides Aleph with sets of positive and negative examples as described in Table 5.1. They are selected from both low-reward games and high-reward games. I found that requiring the complete macro structure to appear in games led to data scarcity in the final nodes, and ruled out many games that appear to contain useful data for those nodes. To address this issue, I allow games that include prefixes and suffixes of the macro sequence to contribute examples as well. For example, in a macro with nodes (n_1, n_2, n_3, n_4), prefixes include games matching (n_1, n_2) and suffixes include games matching (n_2, n_3, n_4).

To make the example selection criteria more concrete, consider the sample macro structure in Figure 5.3. Figure 5.4 illustrates some hypothetical examples for the argument choice in the *pass*

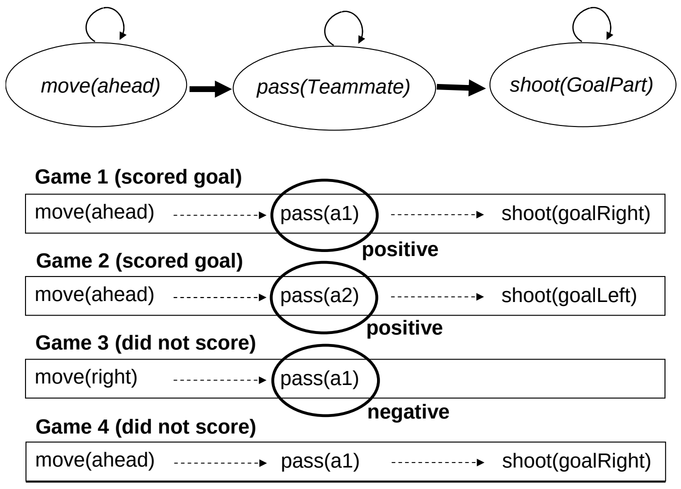

Figure 5.4: Training examples (states circled) for *pass(Teammate)* rules in the second node of the pictured macro. The pass states in Games 1 and 2 are positive examples. The pass state in Game 3 is a negative example; the pass action led directly to a negative game outcome. The pass state in Game 4 is ambiguous because another step may have been responsible for the bad outcome; the algorithm does not use states like these.

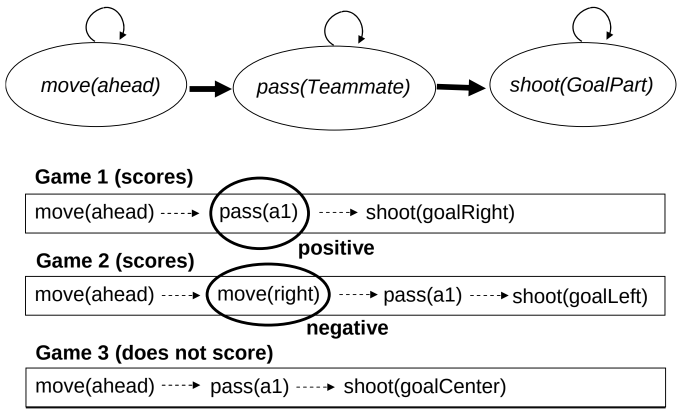

Figure 5.5: Training examples (states circled) for the transition from *move* to *pass* in the pictured macro. The pass state in Game 1 is a positive example. The move state in Game 2 is a negative example; the game follows the macro but remains in the *move* node in the state circled. The pass state in Game 3 is ambiguous because another step may have been responsible for the bad outcome; the algorithm does not use states like these.

node. Figure 5.5 illustrates some hypothetical examples for the transition from the *move* node to the *pass* node.

Again, the algorithm uses Gleaner to save the best clauses that Aleph encounters during its search that have a minimum of 50% precision and 10% recall. These requirements are high enough to ensure that the rules it considers are valuable and representative, but low enough to allow it to consider many rules. Instead of selecting a single clause as it did in the previous phase, it constructs a final ruleset by selecting from the Gleaner clauses. This final ruleset should have a high F score, meaning that it has both good precision and good recall.

I found that it is more important to weight for recall when constructing rulesets, so I use the F(10) score here. Furthermore, since it would be expensive to find the highest-scoring subset exactly, I use a greedy procedure for maximizing the F(10) score. The algorithm sorts the rules by decreasing precision and walks through the list, adding rules to the final ruleset if they increase its F(10) score. This procedure is summarized in Table 5.5.

Table 5.5 also shows how the algorithm scores rules. Each rule has an associated score that is 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. The agent determines this estimate by collecting training-set games that followed the rule and calculating the fraction of these that ended successfully.

5.2.3 Experimental Results for Single-Macro Transfer

To test the single-macro transfer approach, I learn macros from data acquired while training 2-on-1 BreakAway and transfer them to both 3-on-2 and 4-on-3 BreakAway. I use the same source runs of 2-on-1 BreakAway as in our advice-based transfer experiments. Note that I do not test the distant-transfer scenarios from that chapter, since the macro-transfer method is not designed for distant transfer. The mappings I use for these scenarios are documented in Appendix C.

The macros learned from the five source runs had similar structures. Three of them were essentially identical, and their structure is the one in Figure 5.1, with between 30 and 130 rules in each ruleset (not shown). In the other two, the first two nodes are replaced by two grounded *move*

Table 5.5: Algorithm for selecting and scoring the final ruleset for one transition or action. Rules are added to the final set if they increase the overall $F(10)$ measure.

```
Let $S$ = Gleaner rules sorted by decreasing precision on the training set
Let $T = \emptyset$                                      // This will be the final ruleset
For each rule $r \in S$                                              // Select rules
    Let $U$ = $T \cup \{r\}$
    If the $F(10)$ of $U >$ the $F(10)$ of $T$
    Then set $T \leftarrow U$
For each rule $r \in T$                                               // Score rules
    Let $G = \emptyset$                         // These will be the games that followed $r$
    For each source-task game $g$
        If any state $s$ in $g$ matches the literals in $r$ and takes the action $r$ recommends
            Set $G \leftarrow G \cup g$
    Let $H$ = the subset of games in $G$ that are high-reward
    Set $score(r) = \frac{|H|+10}{|G|+20}$              // This is an $m$-estimate of $\frac{|H|}{|G|}$ with $m = 10$
Return $T$
```

nodes. The ordering of *shoot(goalRight)* and *shoot(goalLeft)* also varied, as would be expected in the symmetrical BreakAway domain.

The presence of two *shoot* nodes may seem counterintuitive, but it appears that the RL agent uses the first shot as a feint to lure the goalie in one direction, counting on a teammate to intercept the shot before it reaches the goal. When it does, the teammate in possession of the ball becomes the learning agent and performs the second shot, which is actually intended to score. Thus the first shot is really a type of pass. This tendency of agents to use actions in creative ways is a positive aspect of RL, but it can make human interpretation of policies difficult.

Figures 5.6 and 5.7 show the performance of single-macro transfer in 3-on-2 and 4-on-3 BreakAway compared to RL-SVR and skill transfer. These results show that macro transfer produces qualitatively different behavior in the target task than advice-based transfer. Rather than giving a gradual and long-lasting performance increase, macro transfer gives a large, immediate advantage at the beginning. Both methods converge with RL-SVR by the end of the learning curve, with macro transfer converging sooner than skill transfer.

The reason for this difference is that skill transfer provides a constant subtle influence on the target-task solution, while macro transfer provides an entire temporary solution that is reasonably

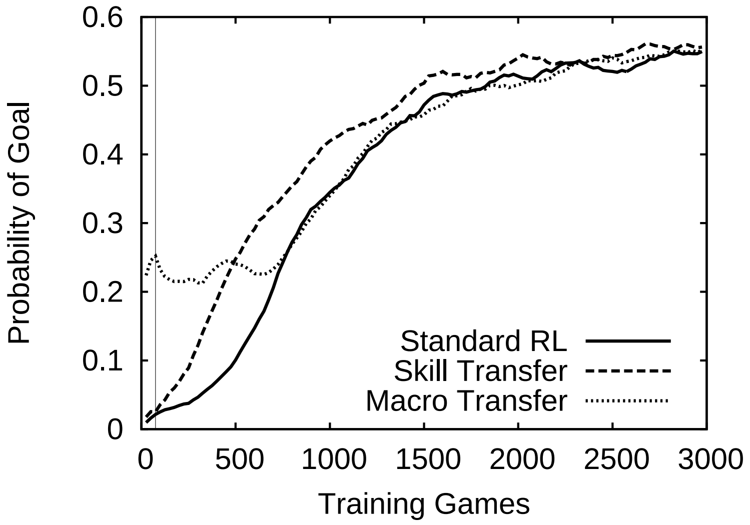

Figure 5.6: Probability of scoring a goal in 3-on-2 BreakAway with RL-SVR, skill transfer from 2-on-1 BreakAway, and single-macro transfer from 2-on-1 BreakAway. The thin vertical line marks the end of the demonstration period.

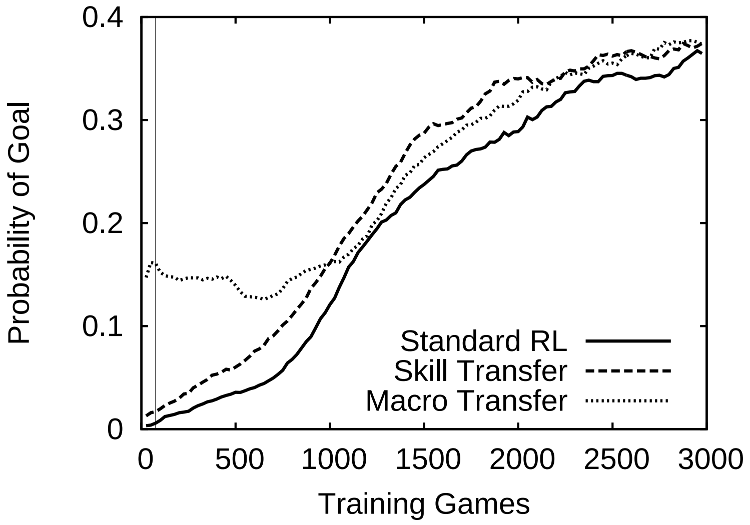

Figure 5.7: Probability of scoring a goal in 4-on-3 BreakAway with RL-SVR, skill transfer from 2-on-1 BreakAway, and single-macro transfer from 2-on-1 BreakAway. The thin vertical line marks the end of the demonstration period.

Table 5.6: Results of statistical tests on the differences between areas under the curve in single-macro transfer vs. skill transfer for several source/target pairs. For $p < 0.05$, the difference is significant and the winning algorithm is shown; otherwise, the difference is not statistically significant.

Source	*Target*	*p-value*	*Significance (Winner)*	*95% CI*
2-on-1 BreakAway	3-on-2 BreakAway	0.168	No	[-88, 31]
2-on-1 BreakAway	4-on-3 BreakAway	0.090	No	[-12, 126]

(but not perfectly) good for the target task. In order to reach the asymptotic solution, the target-task agent must explore its new environment, and it takes some time to explore beyond the immediate neighborhood of the macro due to the bias that the macro provides. Thus the macro is both a help and a hindrance; its bias is quite helpful for early performance, but in a sense it is a local maximum that must be unlearned in order to reach the true asymptote in the target task.

The statistical analysis in Tables 5.6 indicates that the difference between single-macro transfer and skill transfer is not statistically significant in the global sense, though local differences are visually evident. Whether one method is better than the other depends on how much priority is placed on early performance.

It is also interesting to evaluate the performance of a 2-on-1 BreakAway macro when used in 2-on-1 BreakAway, in what I refer to as *self-transfer*. This experiment provides some insight on how well a single macro describes successful behavior in the source task. On average, the macros in my experiments score in 32% of episodes in 2-on-1 BreakAway episodes. In comparison, a random policy scores in less than 1% of episodes, and a fully learned 2-on-1 BreakAway policy scores in 56% of episodes. A single macro therefore captures a large portion of the successful behavior, but does not describe it completely.

5.3 Multiple-Macro Transfer via Demonstration

Multiple-macro transfer via demonstration, my Transfer Algorithm 4, is a transfer method that learns multiple macros from a source task, and allows target-task agents to use them for an initial demonstration period. It is similar to single-macro transfer, except that instead of learning one overall strategy from the source task, it learns several smaller interacting strategies.

My motivation for developing this variant of the macro transfer method is that single macros do not fully capture the source-task behavior. One reason may be that they have a strict linear form describing a single plan, which may be brittle in complex, non-deterministic RL domains. Multiple macros can provide information on what to do when the initial strategy must be abandoned. This ability has particular value for transfer, since the environment in a target task is likely to change in unexpected ways, violating the source-task assumptions built into the macro.

In the multiple-macro method, an agent can abandon its current strategy at any point and switch to a new one. A potential advantage of this structure is greater adaptability. A potential disadvantage is more frequent abandonment of macros, which could prevent steady progress towards game objectives.

Table 5.7 gives the algorithm for multiple-macro transfer. It requires the same input as single-macro transfer and consists of similar steps. The sections below describe the first three steps in more detail.

5.3.1 Multiple-Macro Structure and Ruleset Learning

The first step of the macro-transfer algorithm in Table 5.7 is the structure-learning phase. It performs the same ILP structure search as for single-macro transfer (see Section 5.2.1). However, instead of choosing just one structure, it now includes all the action sequences saved by Gleaner. It still requires these sequences to have a minimum of 50% precision, but it decreases the recall requirement to 10% to allow for more structures. The set of sequences should vary in length and in quality at this stage. Overall, the result looks like the diagram in Figure 5.8.

The second and third steps of the macro-transfer algorithm in Table 5.7 are for ruleset learning. The objective is the same as before: to learn control rules for navigating within each macro based on the RL agent's environment when it reaches that point.

However, the control decisions needed are now slightly changed. In single-macro transfer, agents automatically entered the macro at the start of each game. Since there are now multiple

Table 5.7: Transfer Algorithm 4: Multiple-Macro Transfer via Demonstration

```
INPUT REQUIRED
Games $G$ from the source-task learning process
A definition of high-reward and low-reward games in the source task
A demonstration-period length $D$

LEARN STRUCTURE
Learn rules $R$ according to Table 5.2
Let $Macros = \emptyset$                                  // This will be a list of macros
For each rule $S \in R$                         // Each $S$ represents one macro structure
    Create an empty macro $Macro = \emptyset$ // This will be a set of nodes and rulesets
    For each literal in $S$ representing $(Action, ActionArg)$
        Create a node $n$ of type *Action(ActionArg)*
        Add node $n$ to $Macro$
    Set $Macros \leftarrow Macros \cup Macro$

LEARN ENTRIES
For each macro $M \in Macros$
    For each node $n$ of $M$
        Learn rules $T$ according to Table 5.8
        Select rules $R_{entry}$ from $T$ according to Table 5.5
        Attach $R_{entry}$ to $M$ as the entry ruleset for node $n$

LEARN LOOPS
For each macro $M \in Macros$
    For each node $n$ of $M$
        Learn rules $U$ according to Table 5.8
        Select rules $R_{loop}$ from $U$ according to Table 5.5
        Attach $R_{loop}$ to $M$ as the loop ruleset for node $n$

LEARN TARGET TASK
For $D$ episodes: Perform RL but use $Macros$ to choose actions
For remaining episodes: Perform RL normally
```

Table 5.8: Algorithm for learning rules to control the entry arc of a node n of a macro, given games G from the source task. Data for this task can come from any game, and most comes from those that contain the sequence represented by $Macro$ or some prefix of that sequence. For example, in a macro with nodes (n_1, n_2, n_3, n_4), prefixes include games matching (n_1, n_2).

```
Let $P = \emptyset$                                  // These will be the positive examples
Let $N = \emptyset$                                  // These will be the negative examples
For each game $g$ that is high-reward in the source task and matches $S$ or a prefix of $S$
   Let $x$ be the first state in node $n$ of $g$
   Set $P \leftarrow P \cup x$                                  // States that enter are positive
   If $n$ is the first node of $M$
      Let $Y$ be any states in $g$ that enter other macros
      Set $N \leftarrow N \cup Y$                 // States that could enter but do not are negative
   Else
      Let $z$ be any state in $g$ that abandons the macro after node $n - 1$
      Set $N \leftarrow N \cup z$                    // States that abandon the macro are negative
Learn rules $R$ with Aleph and Gleaner to distinguish $P$ from $N$
Return $R$
```

Table 5.9: Algorithm for learning rules to control the loop arc of a node n of a macro, given games G from the source task. Data for this task can come from any game, and most comes from those that contain the sequence represented by $Macro$ or some prefix of that sequence. For example, in a macro with nodes (n_1, n_2, n_3, n_4), prefixes include games matching (n_1, n_2).

```
Let $P = \emptyset$                                  // These will be the positive examples
Let $N = \emptyset$                                  // These will be the negative examples
For each game $g$ that is high-reward in the source task and matches $S$ or a prefix of $S$
   Let $X$ be any states in $g$ that remain in node $n$ after entering it in a previous state
   Set $P \leftarrow P \cup X$                                  // States that loop are positive
   Let $y$ be any state in $g$ that abandons the macro after node $n$
   Set $N \leftarrow N \cup y$                    // States that abandon the macro are negative
Learn rules $R$ with Aleph and Gleaner to distinguish $P$ from $N$
Return $R$
```

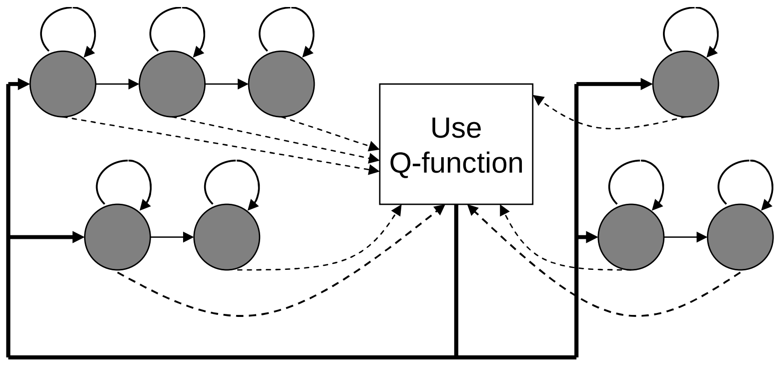

Figure 5.8: A structural diagram of the knowledge captured by the multiple-macro transfer method. An agent starts in a default node using the *Q*-function, and it can choose to enter any macro. At any point in that macro, it can choose to abandon and return to the default node. There it can choose to enter a new macro, or if none seem appropriate, it can fall back on the *Q*-function to choose an action.

structures, agents need to evaluate conditions for entering a structure, so they can determine the most appropriate macro for their current situation. Furthermore, at each node, agents need to decide explicitly whether to continue, loop, or abandon. These decisions are made in the following order:

1. Should I enter the next node or not?
2. If not, should I loop in the current node or abandon this macro?

This changes my original definition of a macro in Section 5.1. Now, a macro consists of:

- A linear set of nodes $N = (n_1, n_2, ...)$
- Arcs for entering each node from the previous one with associated rulesets $R_{entry}(n_i)$
- Arcs that form self-loops for each node with associated rulesets $R_{loop}(n_i)$

The ruleset $R_{entry}(n_i)$ is used to answer the first question above, and the ruleset $R_{loop}(n_i)$ is used to answer the second question. Note that there is no separate ruleset for choosing actions within nodes; the R_{entry} and R_{loop} rulesets now incorporate that decision.

The details of the ILP tasks for this step are as follows. To describe the conditions on state S under which the agent should enter a node, Aleph must construct a clause *enter(S)* for a single-action node or *enter(S, ActionArgs)* for a variablized-action node. To describe the conditions under which the agent should loop in a node, Aleph must construct a clause *loop(S)* for a single-action node or *loop(S, ActionArgs)* for a variablized-action node. As before, in variablized *pass* and *shoot* nodes, some rules leave *ActionArgs* variablized while other rules apply only to specific arguments.

All the examples in this algorithm are states from high-reward games. This is another difference from single-macro transfer, which also gleans some negative examples from low-reward games. It would also be a reasonable design decision to include such examples here. However, I found that doing so complicates the algorithm without providing noticeable benefits, since the high-reward games provide enough negative examples already to learn good rulesets.

For an entry ruleset, the positive examples are states that entered the node, and the negative examples are states that looped in the previous node or abandoned the macro after the previous node. An exception is the first node in the macro, in which negative examples are states that entered a different macro. For a loop ruleset, the positive examples are states that looped in the node, and the negative examples are states that abandoned after the node. The algorithm uses the same procedures as before for selecting final rulesets and scoring rules.

5.3.2 Experimental Results for Multiple-Macro Transfer

To test the multiple-macro transfer approach, I learn sets of macros from the same 2-on-1 BreakAway source runs as for single macros, and transfer them to 3-on-2 and 4-on-3 BreakAway. Each source run produced a similar set of 8 to 10 macros, ranging in length from one node to four nodes. Figure 5.9 shows the actual set for one of the source runs.

Figures 5.10 and 5.11 show the performance of multiple-macro transfer in 3-on-2 and 4-on-3 BreakAway compared to RL-SVR, skill transfer, and single-macro transfer. These results show that multiple-macro transfer produces a learning curve similar in shape to single-macro transfer, but with an overall improvement in some cases. The improvement is not drastic, which indicates that the BreakAway task may not have a large number of necessary macros, but clearly allowing more than just one macro can be useful.

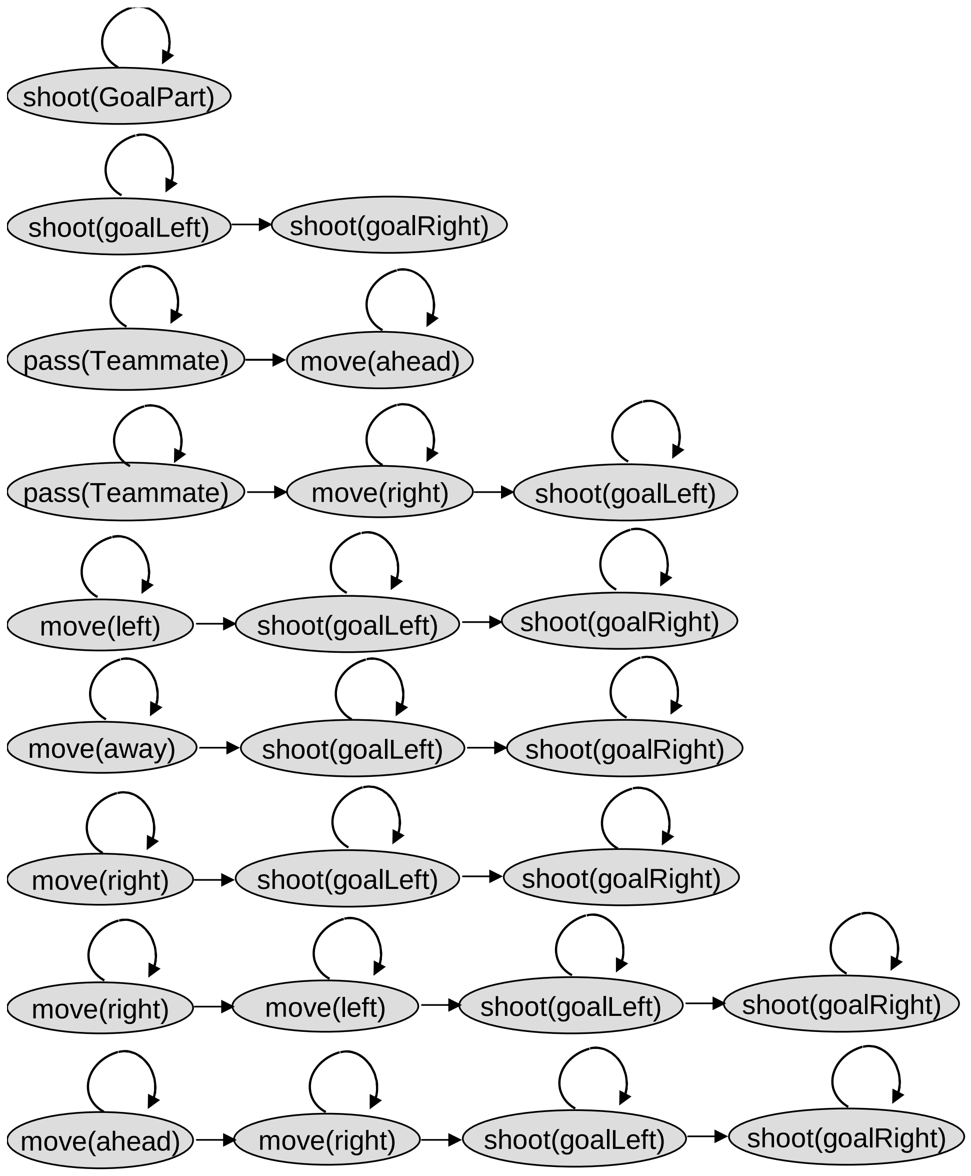

Figure 5.9: The actual list of macros learned for one source run in my experiments with multiple-macro transfer. Only the nodes, and not the rulesets, are shown.

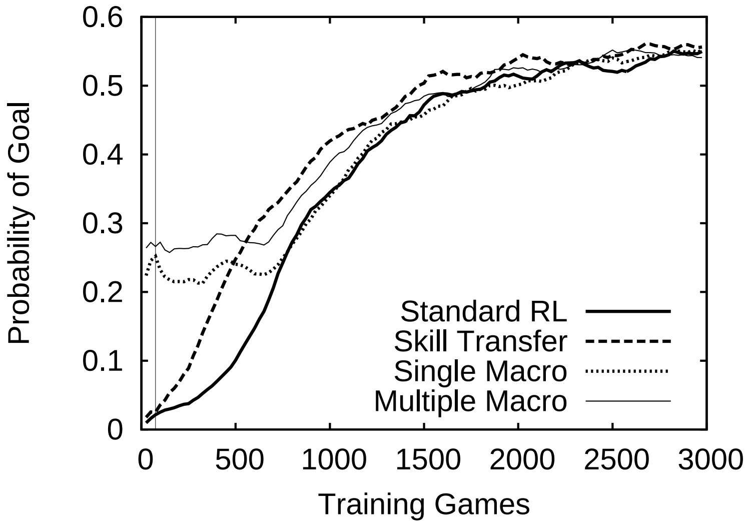

Figure 5.10: Probability of scoring a goal in 3-on-2 BreakAway with RL-SVR, skill transfer from 2-on-1 BreakAway, single-macro transfer from 2-on-1 BreakAway, and multiple-macro transfer from 2-on-1 BreakAway. The thin vertical line marks the end of the demonstration period.

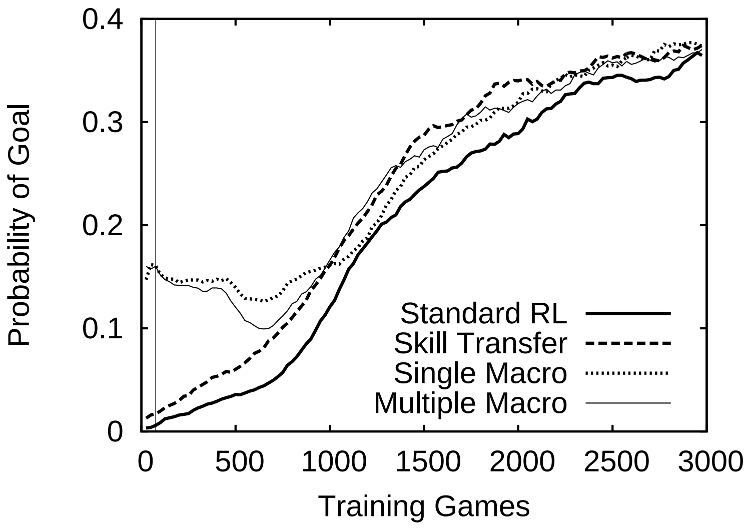

Figure 5.11: Probability of scoring a goal in 4-on-3 BreakAway with RL-SVR, skill transfer from 2-on-1 BreakAway, single-macro transfer from 2-on-1 BreakAway, and multiple-macro transfer from 2-on-1 BreakAway. The thin vertical line marks the end of the demonstration period.

Table 5.10: Results of statistical tests on the differences between areas under the curve in multiple-macro transfer vs. single-macro transfer for several source/target pairs. For $p < 0.05$, the difference is significant and the winning algorithm is shown; otherwise, the difference is not statistically significant.

Source	*Target*	*p-value*	*Significance (Winner)*	*95% CI*
2-on-1 BreakAway	3-on-2 BreakAway	0.004	Yes (Multiple)	[21, 116]
2-on-1 BreakAway	4-on-3 BreakAway	0.452	No	[-56, 48]

The statistical analysis in Table 5.10 indicates that the difference between multiple-macro transfer and single-macro transfer is significant in 3-on-2 BreakAway, though not in 4-on-3 BreakAway. Multiple macros may provide fewer benefits as the source and target tasks grow further apart.

In a 2-on-1 BreakAway self-transfer test, multiple macros score in 43% of episodes, compared to 32% for single macros and 56% at the asymptote of the learning curve. Multiple macros therefore capture more of the successful behavior than single macros do, though they still do not describe it completely.

5.4 Testing the Boundaries of Macro Transfer

One shortcoming of relational macros may be their ad-hoc method of making decisions. Recall that when multiple rules match, they simply follow the single rule with the highest score. Thus they are essentially making decisions based on one rule in the set. An agent may be able to make better decisions by taking information from all the rules into account, instead of just the highest-scoring one. This section describes further experiments with macro transfer along these lines.

When an agent reaches a decision point in a macro, its current environment will likely satisfy some of the rules in the relevant ruleset and not satisfy others. I now wish to employ a classifier that uses all of that information to make the final decision, and preferably one that handles relational features. My solution is to use a Markov Logic Network (MLN), a statistical-relational model described in Section 2.4, which interprets first-order statements as soft constraints with weights [70].

I describe and test this method in the context of single macros for simplicity. For each ruleset in the macro, I use the rules as formulas for an MLN. However, rather than expressing them

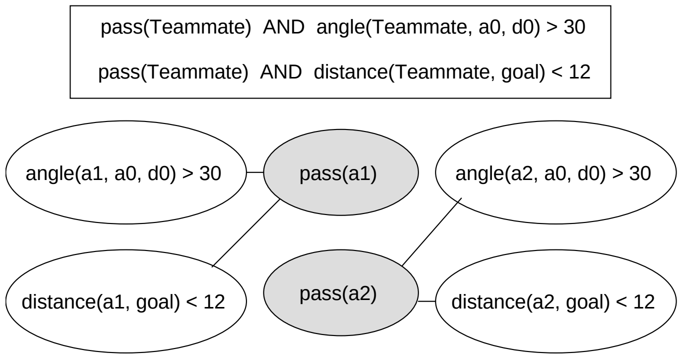

Figure 5.12: A small ruleset (with rule weights not shown) and the ground Markov network it would produce. Each grounded literal becomes a node in the MLN. When literals appear together in clauses, their nodes are linked. Groups of linked nodes form cliques, which have potential functions (not shown) describing their joint probabilities. To use this MLN to choose between *pass(a1)* and *pass(a2)*, an agent infers their conditional probabilities and takes the higher one.

as implications, I express them as conjuncts, since this is recommended by MLN experts [20]. Figure 5.12 shows a simple hypothetical ruleset in this conjunct form instead of the usual IF...THEN form. It also shows what the ground Markov network would look like in 3-on-2 BreakAway.

I learn formula weights with the Alchemy MLN software [40], using the same data for positive and negative examples as the original algorithm did when learning the rulesets. Each ruleset is therefore replaced by an MLN. Here again, weight learning across multiple rulesets from different arcs in a macro could be treated as a combined problem requiring a joint solution. However, to maintain consistency and to avoid large problem sizes, I continue to treat each ruleset independently.

As described in Figure 5.12, a target-task agent uses an MLN like this to make decisions by inferring the conditional probabilities of the nodes representing its choices and taking the choice whose node has highest probability. Since the MLN captures the distribution of source-task choices at one point in a macro, this means taking the action the source-task policy would most likely take.

Inference is normally accomplished by an approximation algorithm. However, the particular structure of my MLNs allows for efficient exact inference. I now describe this calculation, using the example of choosing an action in Figure 5.12.

Recall that each MLN formula $f_i \in F$, with weight w_i, has a number $n_i(x)$ of true groundings in each possible world x, and that the probability that the world x is the correct one is, from Equation 2.7 in Section 2.4:

$$P(X = x) = \frac{1}{Z}\, exp \sum_{i \in F} w_i n_i(x) \tag{5.1}$$

The nodes for *pass(a1)* and *pass(a2)* in Figure 5.12 are conditionally independent. Intuitively, the reason is that these decision nodes are only connected to evidence nodes whose truth values are known, so they do not affect each other's values. For a formal proof, see Appendix D. The result of this conditional independence is that when calculating the probability for *pass(a1)*, only two worlds need to be considered: $X = 1$ where *pass(a1)* is true, and $X = 0$ where *pass(a1)* is false.

Since the formulas are in conjunct form, the only true groundings of formulas for *pass(a1)* are those in which *pass(a1)* is true. This means that $n_i(0) = 0$, and that:

$$P(X = 0) = \frac{1}{Z}\, exp \sum_{i \in F} w_i n_i(0) = \frac{1}{Z}\, exp(0) = \frac{1}{Z} \tag{5.2}$$

Recall that Z is just a normalizing factor so that the probabilities of both worlds sum to 1:

$$P(X = 0) + P(X = 1) = 1 \tag{5.3}$$

$$\frac{1}{Z} + \frac{1}{Z}\, exp \sum_{i \in F} w_i n_i(1) = 1 \tag{5.4}$$

The value of Z is therefore:

$$Z = exp \sum_{i \in F} w_i n_i(1) + 1 \tag{5.5}$$

Substituting Z back into Equation 5.1 gives:

$$P(X=1) = \frac{exp \sum_{i \in F} w_i n_i(1)}{1 + \ exp \sum_{i \in F} w_i n_i(1)} \tag{5.6}$$

This is the solution for the conditional probability of the *pass(a1)* node. Note that it is a logistic function, which confirms the point made by Domingos and Richardson [22] that logistic regression can be considered a special case of Markov Logic Networks.

To evaluate the use of MLNs for decision-making in macros, I take the macros learned from 2-on-1 BreakAway in single-macro transfer and construct MLNs for them as described above. I then transfer these enhanced macros to 3-on-2 and 4-on-3 BreakAway.

I found it important to tune one Alchemy parameter in this approach. This parameter, called *priorStdDev* in Alchemy, governs the width of the Gaussian prior distribution of each formula weight (centered at zero). I found that the default setting of 2 was too high for some runs, allowing formula weight magnitudes to grow too large, and producing overfitting (the untuned MLNs perform much worse on a tuning set than on the training data). I addressed this by choosing from values of $\{2, 1, 0.5, 0.1, 0.05, 0.01\}$ with a validation set of source-task data. The appropriate setting was 0.05 for most runs.

Figures 5.13 and 5.14 show the performance of this method in 3-on-2 and 4-on-3 BreakAway compared to RL-SVR and the original single-macro transfer approach. These results show that at best, this method produces no significant difference in performance, and at worst, as the distance between the source and target task grows, it actually decreases performance.

The statistical analysis in Table 5.11 indicates that the difference between MLN-enhanced macros and regular macros is not statistically significant for 3-on-2 BreakAway, though it is for 4-on-3 BreakAway. This method may become more harmful as the source and target tasks grow further apart.

In a 2-on-1 BreakAway self-transfer test, MLN-enhanced macros score in 43% of episodes, compared to 32% for single macros, 43% for multiple macros, and 56% at the asymptote of the learning curve. The addition of MLNs for decision-making therefore does improve the description

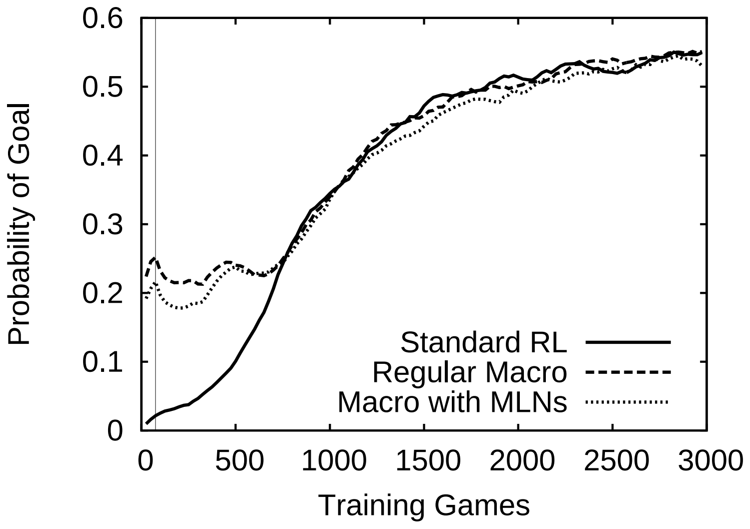

Figure 5.13: Probability of scoring a goal in 3-on-2 BreakAway with RL-SVR, single-macro transfer from 2-on-1 BreakAway, and single-macro transfer with MLNs from 2-on-1 BreakAway. The thin vertical line marks the end of the demonstration period.

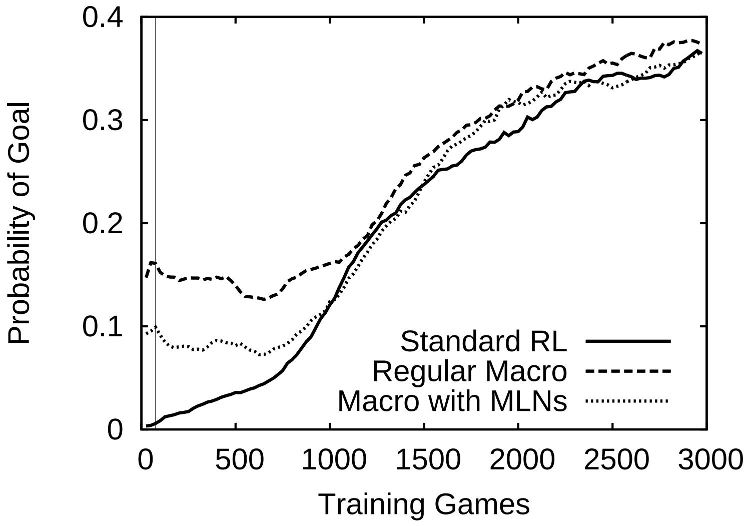

Figure 5.14: Probability of scoring a goal in 4-on-3 BreakAway with RL-SVR, single-macro transfer from 2-on-1 BreakAway, and single-macro transfer with MLNs from 2-on-1 BreakAway. The thin vertical line marks the end of the demonstration period.

Table 5.11: Results of statistical tests on the differences between areas under the curve in single-macro transfer with MLNs vs. regular single-macro transfer for several source/target pairs. For $p < 0.05$, the difference is significant and the winning algorithm is shown; otherwise, the difference is not statistically significant.

Source	*Target*	*p-value*	*Significance (Winner)*	*95% CI*
2-on-1 BreakAway	3-on-2 BreakAway	0.068	No	[-92, 10]
2-on-1 BreakAway	4-on-3 BreakAway	0.002	Yes (Regular)	[29, 163]

of successful source-task behavior. However, this source-task improvement does not carry over into target-task improvement the way that it did for multiple macros.

The reason for this result appears to be that the MLNs improve classification accuracy, compared to the original rulesets, primarily by decreasing the number of false positives. In many scenarios, this would be a desirable change. However, in the specific case of macro decision-making, the result is that the MLNs decide to take fewer transitions and consider fewer action arguments. This produces over-cautious behavior that misses opportunities, particularly in transfer scenarios when the environment does not look exactly as it would in the source task.

Macros perform better when they base their decisions on important similarities between source and target environments, rather than making a complete comparison. Taking all the rules into account puts too much focus on the differences between the tasks and does not allow the macro to take advantage of the similarities. Within macro transfer, the seemingly ad-hoc method of following the highest-scoring rule is superior in my experiments to the theoretically more principled MLN method.

This result is an example of a phenomenon that is related to overfitting. The traditional sense of overfitting is that modeling training data too closely causes a learner to treat spurious patterns as important, which means the model does not generalize well even to data drawn from the same distribution. If this traditional kind of overfitting occurred in transfer, then the source-task knowledge would produce low performance in self-transfer. However, source-task knowledge can produce high performance in self-transfer while failing to generalize well to target tasks, especially as they grow more distant from the source. I call this phenomenon *overspecialization*.

5.5 Summary of Macro Transfer

Since standard RL has to act mostly randomly in the early steps of a task, a good macro strategy can provide a large immediate advantage. The performance level of the demonstrated strategy is unlikely to be as high as the target-task agent can achieve with further training, unless the two tasks have identical strategies. However, through further learning the agent can improve its performance from the level of the demonstration up to its asymptote.

The macro is both a help and a hindrance; its bias is quite helpful for early performance, but in a sense it is a local maximum that must be unlearned in order to reach the true asymptote in the target task. This is one reason that my demonstration approach stops using transferred knowledge abruptly instead of smoothly decaying its use, which would also be a reasonable design choice. Extending the use of the transferred knowledge might only delay the unlearning and relearning process in the target task.

Transferring multiple macros rather than a single macro provides a more flexible strategy, and can produce better performance. Macros perform well with the simple method of following the highest-scoring rule, and more sophisticated methods of decision-making that improve classification accuracy actually decrease performance due to decreased flexibility and overly cautious behavior. This is an example of overspecialization to a source task, which can hurt performance in a target task.

Demonstration has both advantages and disadvantages compared to advice taking. The most notable disadvantage is its lack of protection against negative transfer, which makes it appropriate only for close-transfer scenarios. A more subtle disadvantage is the time that it takes to adjust the macro strategy appropriately to the target task. However, if there is limited time and the target task cannot be trained to its asymptote, then the immediate advantage that macros can provide may be quite valuable in comparison to advice-taking methods.

Although I found that using Markov Logic Networks in conjunction with macros was not desirable, the MLN remains a powerful relational model in its own right that could be used as a structure for knowledge transfer. The next chapter presents methods for transfer via MLNs.

Chapter 6

Transfer via Markov Logic Networks

Statistical relational learning (SRL) is a type of machine learning designed to operate in domains that have both uncertainty and rich relational structure [33]. 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.

An SRL model could therefore capture relational information about a task in a more flexible way than the ILP-based techniques I have discussed so far. One of the realistic aspects of the RoboCup domain is its non-determinism, and ILP techniques, which only express that concepts are true or false, do not capture non-determinism. SRL techniques can express that concepts are true with certain probabilities, which makes them potentially a more powerful type of model for transfer.

One recent and popular SRL formalism is the Markov Logic Network (MLN), described in Section 2.4, which interprets first-order statements as soft constraints with weights [70]. Where skill transfer captured single actions, and macro transfer captured common sequences of actions, MLNs can capture an entire Q-function or policy. Like macro transfer, my MLN transfer methods are appropriate only for close-transfer scenarios because they sacrifice protection against negative transfer in order to produce large initial benefits.

This chapter presents two methods for performing transfer with MLNs. One method expresses the source-task Q-function with an MLN, and the other expresses the source-task policy. This chapter is based in part on published work [102].

6.1 MLN Relational Q-Function Transfer

MLN relational Q-function transfer, my Transfer Algorithm 5, is a transfer method that learns an MLN to express the source-task Q-function relationally, and allows target-task agents to use it for an initial demonstration period. Like the macro-transfer methods, this allows the target-task agents to avoid the slow process of random exploration that traditionally occurs at the beginning of RL. This section is based on published work [102].

An MLN Q-function is potentially more expressive than a macro, because it is not limited to one action sequence. Furthermore, due to its relational nature, it provides better generalization to new tasks than a propositional value-function transfer method could.

This method uses an MLN to define a probability distribution for the Q-value of an action, conditioned on the state features. It chooses a source-task batch and uses its training data to learn an MLN Q-function for transfer. The choice of which source-task batch has an impact, as I will discuss.

In this scenario, an MLN formula describes some characteristic of the RL agent's environment 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, one formula in an MLN representing the Q-function for BreakAway could look like the following:

```
IF     distBetween(a0, GoalPart) > 10
AND    angleDefinedBy(GoalPart, a0, goalie) < 30
THEN   levelOfQvalue(move(ahead), high)
```

The MLN could contain multiple formulas like this for each action. After learning weights for the formulas from source-task data, one could use this MLN to infer, given a target-task state, whether action Q-values are most likely to be high, medium, or low.

Note that Q-values in RoboCup are continuous rather than discrete, so I do not actually learn rules classifying them as high, medium, or low. Instead, the algorithm discretizes the continuous Q-values into bins that serve a similar purpose.

Table 6.1: Transfer Algorithm 5: MLN Relational *Q*-Function Transfer

```
INPUT REQUIRED
 A set of batches $B = (b_1, b_2, ...)$ to consider for transfer
 The Q-function $Q^b$ for each batch $b \in B$
 The set of games $G(b)$ that trained the Q-function for each batch $b \in B$
 A parameter $\epsilon$ determining distance between bins
 A demonstration-period length $D$
 A validation-run length $V$

CREATE Q-VALUE BINS                          // This is a hierarchical clustering procedure
 For each batch $b \in B$
     For each source-task action $a$
         Determine $bins(b, a)$ for action $a$ in batch $b$ using Table 6.2
         (Provide inputs $G(b)$ and $\epsilon$)

LEARN FORMULAS                               // This accomplishes MLN structure learning
 For each batch $b \in B$
     For each source-task action $a$
         For each $bin \in bins(b, a)$
             Let $P = \emptyset$                          // These will be the positive examples
             Let $N = \emptyset$                          // These will be the negative examples
             For each state $s$ in a game $g \in G(b)$
                 If $s$ used action $a$ and $Q_a^b(s)$ falls into $bin$
                     Set $P \leftarrow P \cup g$          // Examples that fall into the bin are positive
                 Else if $s$ used action $a$ and $Q_a^b(s)$ does not fall into $bin$
                     Set $N \leftarrow N \cup g$          // Examples that fall outside the bin are negative
             Learn rules with Aleph and Gleaner to distinguish $P$ from $N$
             Let $M(b, a, bin)$ be the ruleset chosen by the algorithm in Table 5.5
         Let $M(b, a)$ be the union of $M(b, a, bin)$ for all bins

LEARN FORMULA WEIGHTS
 For each batch $b \in B$
     For each source-task action $a$
         Learn MLN weights $W(b, a)$ for the formulas $M(b, a)$ using Alchemy
         Define $MLN(b, a)$ as $(M(b, a), W(b, a))$
     Define $MLN(b)$ as the set of MLNs $MLN(b, a)$

CHOOSE A BATCH                     // Do a validation run in the source task to pick the best batch
 For each batch $b \in B$
     For $V$ episodes: Use $MLN(b)$ as shown in Table 6.3 to choose actions in a new source-task run
     Let $score(b)$ be the average score in this validation run
 Choose the highest-scoring $b^* \in B = argmax_b\ score(b)$

LEARN TARGET TASK
 For $D$ episodes: Perform RL but use $MLN(b^*)$ to choose actions as shown in Table 6.3
 For remaining episodes: Perform RL normally
```

Table 6.1 gives the algorithm for MLN Q-function transfer. It requires as input a set of batches from which to attempt transfer, the Q-function learned after each batch and the games used to train it, a parameter determining the number of bins, the length of the demonstration period, and the length of a validation run that is used to choose a batch. It consists of five steps: creating Q-value bins, learning formulas for the MLNs, learning weights for the formulas, using a validation run to choose the best batch, and learning the target task via demonstration. The sections below describe these steps in more detail.

6.1.1 Learning an MLN Q-function from a Source Task

The first step of the MLN Q-function transfer algorithm in Table 6.1 is to divide the Q-values for an action into bins, according to the procedure in Table 6.2. The training example Q-values could have any arbitrary distribution, so it uses a hierarchical clustering algorithm to find good bins. Initially every training example is its own cluster, and it repeatedly joins 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.

Table 6.2: Algorithm for dividing the Q-values of an action a into bins, given training data from games G and a parameter ϵ determining distance between bins.

```
For each state $i$ in a game $g \in G$ that takes action $a$
   Create cluster $c_i$ containing only the $Q$-value of example $i$
Let $C$ = sorted list of $c_i$ for all $i$
Let $m$ = min distance between two adjacent $c_x, c_y \in C$
While $m < \epsilon$                         // Join clusters until too far apart
   Join clusters $c_x$ and $c_y$ into $c_{xy}$
   $C \leftarrow C \cup c_{xy} - \{c_x, c_y\}$
   $m \leftarrow$ min distance between two new adjacent $c'_x, c'_y \in C$
Let $B = \emptyset$                          // These will be the bins for action $a$
For each final cluster $c \in C$        // Center one bin on each cluster
   Let bin $b$ have midpoint $\bar{c}$, the average of values in $c$
   Set the boundaries of $b$ at adjacent midpoints or $Q$-value limits
   Set $B \leftarrow B \cup b$
Return $B$
```

The value of ϵ should be domain-dependent. For BreakAway, which has *Q*-values ranging from approximately 0 to 1, I 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. I 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 second step of the MLN *Q*-function transfer algorithm in Table 6.1 performs structure-learning for the MLN. The MLN formulas are rules that assign training examples into bins. The algorithm creates these rulesets with the same ILP techniques described in previous chapters. Some examples of bins learned for *pass* in 2-on-1 BreakAway, and of rules learned for those bins, are:

IF	distBetween(a0, GoalPart) ≥ 42
AND	distBetween(a0, Teammate) ≥ 39
THEN	pass(Teammate) has a Q-value in the interval [0, 0.11]

IF	angleDefinedBy(topRightCorner, goalCenter, a0) ≤ 60
AND	angleDefinedBy(topRightCorner, goalCenter, a0) ≥ 55
AND	angleDefinedBy(goalLeft, a0, goalie) ≥ 20
AND	angleDefinedBy(goalCenter, a0, goalie) ≤ 30
THEN	pass(Teammate) has a Q-value in the interval [0.11, 0.27]

IF	distBetween(Teammate, goalCenter) ≤ 9
AND	angleDefinedBy(topRightCorner, goalCenter, a0) ≤ 85
THEN	pass(Teammate) has a Q-value in the interval [0.27, 0.43]

The third step of the algorithm learns weights for the formulas using Alchemy's conjugate gradient-descent algorithm, as described in Section 2.4. The fourth step of the algorithm selects the best batch from among the set of candidates. I found that the results can vary widely depending on the source-task batch from which the algorithm transfers. It selects a good batch using a validation set of source-task data.

6.1.2 Applying an MLN Q-function in a Target Task

The final step of the MLN *Q*-function transfer algorithm in Table 6.1 is to learn the target task with a demonstration approach. During the demonstration period, the target-task learner queries

Table 6.3: Algorithm for estimating the Q-value of action a in target-task state s using the MLN Q-function. This is a weighted sum of bin expected values, where the expected value of a bin is estimated from the training data for that bin.

```
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 $b$
    Collect training examples $T$ for which $Q_a$ falls into bin $b$
    Let $E[Q_a|b]$ be the average of $Q_a(t)$ for all $t \in T$
Return $Q_a(s)$ = $\sum_b(p_b * E[Q_a|b])$
```

the MLN to determine the estimated *Q*-value of each action, and it takes the highest-valued action. Meanwhile, it learns *Q*-functions after each batch, and after the demonstration ends, it begins using those *Q*-functions.

The algorithm in Table 6.3 shows how to estimate a *Q*-value for an action in a new state using an MLN *Q*-function. It begins by performing inference in the MLN to estimate the probability, for each action and bin, that *levelOfQvalue(action, bin)* is true. As when I used MLNs within macros, it can do exact inference because the only non-evidence nodes are the query nodes (see Section 5.4).

For each action a, the algorithm infers the probability p_b that the *Q*-value falls into each bin b. It then uses 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, estimated as the average *Q*-value of the training data in that bin.

The probability distribution that an MLN provides over the *Q*-value of an action could look like one of the examples in Figure 6.1. This approach has links to Bayesian reinforcement learning [83], which also learns distributions over values rather than single values. By explicitly representing uncertainty over *Q*-values through distributions, Bayesian RL can balance exploitation and exploration in principled ways rather than using the ϵ-greedy heuristic. I have not included Bayesian

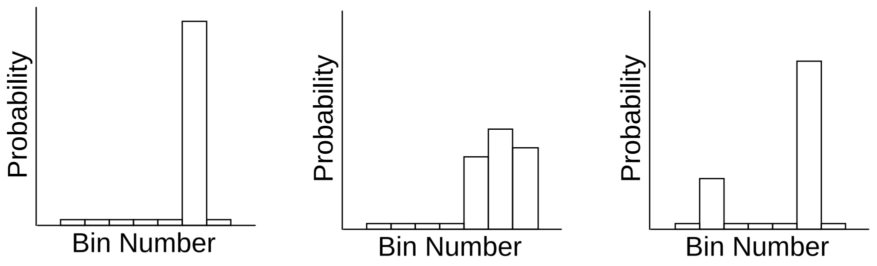

Figure 6.1: Examples of probability distributions over *Q*-value of an action that an MLN *Q*-function might produce. On the left, the MLN has high confidence that the *Q*-value falls into a certain bin, and the action will get a high *Q*-value. In the center, the MLN is undecided between several neighboring bins, and the action will still get a high *Q*-value. On the right, there is a high likelihood of a high bin but also a non-negligible likelihood of a low bin, and the action will get a lower *Q*-value.

exploration in MLN *Q*-function transfer because the short demonstration period makes it unlikely to produce noticeable differences. However, I discuss potential uses of an MLN *Q*-function for Bayesian exploration in the future-work section.

6.1.3 Experimental Results for MLN Q-function Transfer

To test MLN *Q*-function transfer, I learn MLNs from the same 2-on-1 BreakAway source tasks as in previous chapters and transfer them to 3-on-2 and 4-on-3 BreakAway. I focus on close-transfer scenarios for the same reasons as in macro transfer. The mappings I use for these scenarios are documented in Appendix C. As in Section 5.4, I use a validation set of source-task data to tune the Alchemy parameter *priorStdDev*.

Figures 6.2 and 6.3 show the performance of MLN *Q*-function transfer in 3-on-2 and 4-on-3 BreakAway compared to RL-SVR and multiple-macro transfer. These results show that MLN *Q*-function transfer is less effective than multiple-macro transfer, which was the best macro-transfer method.

The statistical analysis in Table 6.4 indicates that the difference between MLN *Q*-function transfer and multiple-macro transfer is statistically significant.

In a 2-on-1 BreakAway self-transfer test, MLN *Q*-functions score in 59% of episodes, compared to 43% for multiple macros and 56% at the asymptote of the learning curve. This indicates

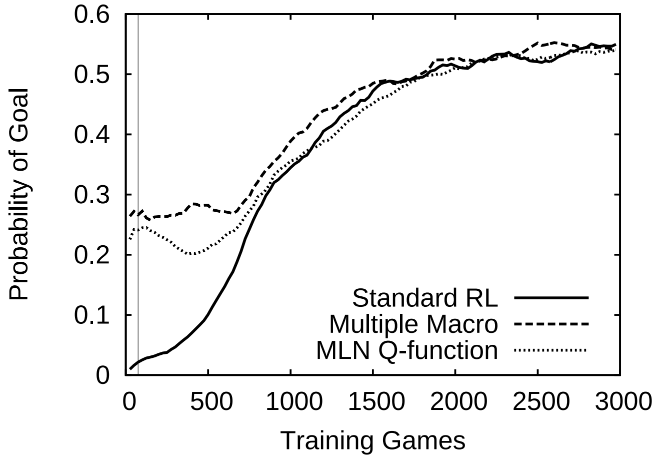

Figure 6.2: Probability of scoring a goal in 3-on-2 BreakAway with RL-SVR, macro transfer from 2-on-1 BreakAway, and MLN Q-function transfer from 2-on-1 BreakAway. The thin vertical line marks the end of the demonstration period.

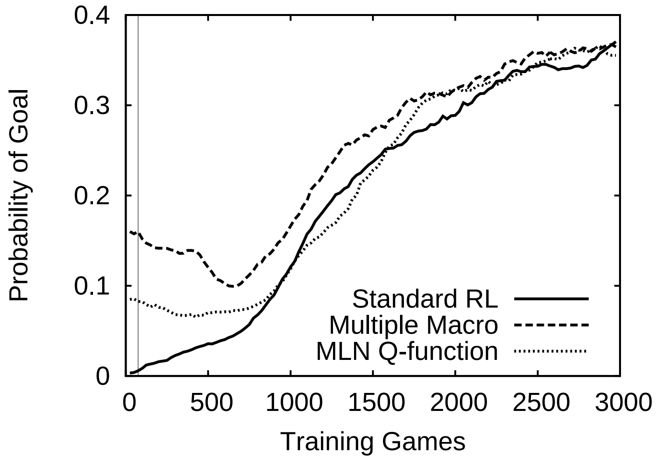

Figure 6.3: Probability of scoring a goal in 4-on-3 BreakAway with RL-SVR, macro transfer from 2-on-1 BreakAway, and MLN Q-function transfer from 2-on-1 BreakAway. The thin vertical line marks the end of the demonstration period.

Table 6.4: Results of statistical tests on the differences between areas under the curve in MLN *Q*-function transfer vs. multiple-macro transfer for several source/target pairs. For $p < 0.05$, the difference is significant and the winning algorithm is shown; otherwise, the difference is not statistically significant.

Source	*Target*	*p-value*	*Significance (Winner)*	*95% CI*
2-on-1 BreakAway	3-on-2 BreakAway	0.004	Yes (Macro)	[24, 132]
2-on-1 BreakAway	4-on-3 BreakAway	0.001	Yes (Macro)	[41, 154]

that MLN *Q*-functions capture the source-task behavior more thoroughly than multiple macros do. However, this source-task improvement does not carry over into target-task improvement.

One reason that MLN *Q*-functions are less effective than multiple macros may be that they transfer information about *Q*-values rather than about policy. As I discussed in Section 4.2 in the context of advice-based policy transfer, it is likely that the *Q*-values in the two tasks are different even in cases where their action choices would be the same. Applying that argument here suggests a different method of MLN transfer, which I address in the next section.

6.2 MLN Relational Policy Transfer

MLN relational policy transfer, my Transfer Algorithm 6, is a method that learns an MLN to express the source-task policy, and allows target-task agents to use it for an initial demonstration period. This approach is closely related to MLN *Q*-function transfer, but it has the potential to transfer more effectively by focusing on policy rather than *Q*-values.

Instead of needing to create bins for continuous *Q*-values, MLN policy transfer learns an MLN that simply predicts the best action to take. This may be more directly comparable to macro transfer. It is also simpler than MLN *Q*-function transfer in that it does not need to choose a batch from which to transfer, which was a significant tuning step in the previous method.

Table 6.5 gives the algorithm for MLN *Q*-function transfer. It requires the same input as the macro-transfer algorithms. It consists of three steps: learning formulas for the MLN, learning weights for the formulas, and learning the target task via demonstration. The section below describes these steps in more detail.

Table 6.5: Transfer Algorithm 6: MLN Relational Policy Transfer

```
INPUT REQUIRED
 Games $G$ from the source-task learning process
 A definition of high-reward and low-reward games in the source task
 The demonstration-period length $D$

LEARN FORMULAS
 Let $G$ be the set of high-reward source-task games
 For each source-task action $a$
     Let $P = \emptyset$                                   // These will be the positive examples
     Let $N = \emptyset$                                   // These will be the negative examples
     For each state $s$ in a game $g \in G$
         If $g$ is a high-reward game in the source task and $s$ used action $a$
             Set $P \leftarrow P \cup s$                   // States that use the action are positive
         Else if $g$ is a high-reward game in the source task and $s$ used action $b \neq a$
             Set $N \leftarrow N \cup s$          // States that use a different action are negative
     Learn rules with Aleph and Gleaner to distinguish $P$ from $N$
     Let $M$ be the ruleset chosen by the algorithm in Table 5.5

LEARN FORMULA WEIGHTS
 Learn MLN weights $W$ for the formulas $M$ using Alchemy
 Define $MLN$ by $(M, W)$

LEARN TARGET TASK
 For $D$ episodes: Perform RL but choose the highest-probability action according to $MLN$
 For remaining episodes: Perform RL normally
```

6.2.1 Learning and Using an MLN Policy

The first and second steps of the MLN policy-transfer algorithm in Table 6.5 perform structure-learning and weight-learning for the MLN. However, the formulas simply predict when an action is the best action to take, rather than predicting a *Q*-value bin for an action as they do in MLN *Q*-function transfer.

Examples for this learning task come from high-reward games throughout the learning curve; as in multiple-macro transfer, it would be possible to glean additional negative examples from low-reward games, but I found it unnecessary. The positive examples for an action are states in high-reward games in which that action was taken. The negative examples are states in high-reward games in which a different action was taken.

Note that this ILP task is a simplified version of the skill-transfer ILP task in Section 4.3. It produces similar rules, though instead of just one per action, there are many. Some examples of rules learned for *pass* in 2-on-1 BreakAway are:

```
IF     angleDefinedBy(topRightCorner, goalCenter, a0) $\leq 70$
AND    timeLeft $\geq 98$
AND    distBetween(a0, Teammate) $\geq 3$
THEN   pass(Teammate)

IF     distBetween(a0, GoalPart) $\geq 36$
AND    distBetween(a0, Teammate) $\geq 12$
AND    timeLeft $\geq 91$
AND    angleDefinedBy(topRightCorner, goalCenter, a0) $\leq 80$
THEN   pass(Teammate)

IF     distBetween(a0, GoalPart) $\geq 27$
AND    angleDefinedBy(topRightCorner, goalCenter, a0) $\leq 75$
AND    distBetween(a0, Teammate) $\geq 9$
AND    angleDefinedBy(Teammate, a0, goalie) $\geq 25$
THEN   pass(Teammate)
```

The rulesets produced by my usual search and selection procedures become formulas in the MLN policy. Weights for the formulas are learned as before, using Alchemy's conjugate-gradient descent algorithm.

The final step of the MLN policy-transfer algorithm in Table 6.5 learns the target task with a demonstration approach. During the demonstration period, the target-task learner queries the MLN to determine the probability that each action is best, and it takes the highest-probability action. Meanwhile, it learns *Q*-functions after each batch, and after the demonstration ends, it begins using those *Q*-functions.

6.2.2 Experimental Results for MLN Policy Transfer

To test MLN policy transfer, I learn MLNs from the same 2-on-1 BreakAway source tasks as in MLN *Q*-function transfer and transfer them to 3-on-2 and 4-on-3 BreakAway. As in Section 5.4, I use a validation set of source-task data to tune the Alchemy parameter *priorStdDev*.

Figures 6.4 and 6.5 show the performance of MLN policy transfer in 3-on-2 and 4-on-3 BreakAway compared to RL-SVR, multiple-macro transfer, and MLN *Q*-function transfer. These results show that transferring an MLN policy is more effective than transferring an MLN *Q*-function, and is comparable to multiple-macro transfer. The statistical analysis in Table 6.6 indicates that the area under the curve for MLN policy transfer is significantly higher than for MLN *Q*-function transfer.

In a 2-on-1 BreakAway self-transfer test, MLN policies score in 65% of episodes, compared to 43% for multiple macros, 59% for MLN *Q*-functions, and 56% at the asymptote of the learning curve. This indicates that an MLN policy is a substantial improvement over the source-task policy from which it was learned, which is an interesting achievement beyond the context of transfer learning. As I will discuss later, it suggests that MLN policies could be used to improve reinforcement learning.

As hypothesized, transferring policy information rather than *Q*-value information does lead to better MLN transfer. However, despite the fact that MLN policies clearly capture more source-task knowledge than multiple macros do, they both lead to similar transfer results. This does not appear to be overspecialization, because the closer source-task fit is not hurting performance in the target task. However, it does indicate that transfer can have diminishing returns. After a point, transferring more accurate detail about the source-task solution no longer helps in the target task because the tasks are different.

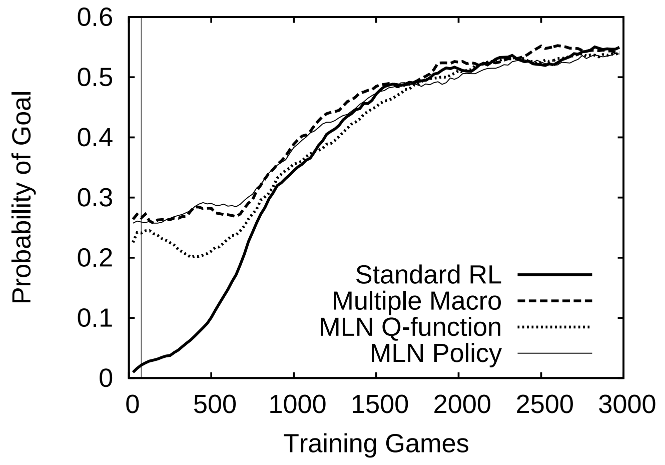

Figure 6.4: Probability of scoring a goal in 3-on-2 BreakAway with RL-SVR, multiple-macro transfer from 2-on-1 BreakAway, MLN *Q*-function transfer from 2-on-1 BreakAway, and MLN policy transfer from 2-on-1 BreakAway. The thin vertical line marks the end of the demonstration period.

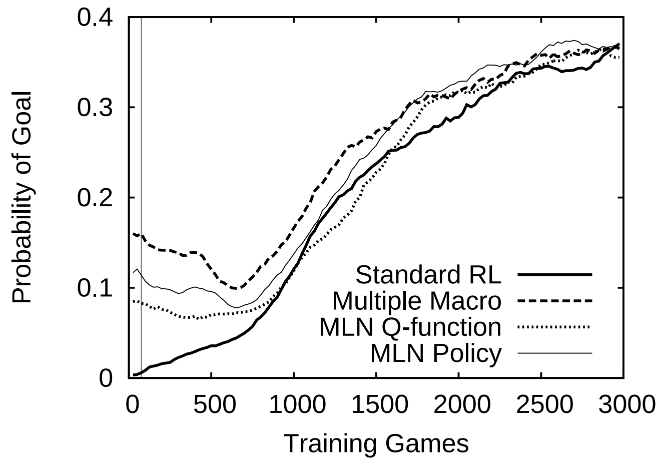

Figure 6.5: Probability of scoring a goal in 4-on-3 BreakAway with RL-SVR, multiple-macro transfer from 2-on-1 BreakAway, MLN *Q*-function transfer from 2-on-1 BreakAway, and MLN policy transfer from 2-on-1 BreakAway. The thin vertical line marks the end of the demonstration period.

Table 6.6: Results of statistical tests on the differences between areas under the curve in MLN policy transfer vs. MLN Q-function transfer for several source/target pairs. For $p < 0.05$, the difference is significant and the winning algorithm is shown; otherwise, the difference is not statistically significant.

Source	*Target*	*p-value*	*Significance (Winner)*	*95% CI*
2-on-1 BreakAway	3-on-2 BreakAway	0.023	Yes (Policy)	[4, 105]
2-on-1 BreakAway	4-on-3 BreakAway	0.029	Yes (Policy)	[1, 119]

6.3 Testing the Boundaries of MLN Policy Transfer

In this section I evaluate two aspects of MLN policy transfer. First, I examine the benefit of using an MLN at all, when the ILP rulesets alone could provide a policy. Second, I test whether it is useful to include action-sequence information in MLN policy rules, which in a sense enhances MLNs with a property of macros, just as the previous chapter attempted to enhance macros with MLNs.

An MLN policy could be viewed as a macro that contains just one node in which every action is possible. As in a normal macro node, there is a ruleset for each possible action. In a macro, I found that allowing the best satisfied rule to choose the action performed better than using an MLN to choose an action based on all the rules. It is therefore a reasonable question whether an MLN is necessary at all in MLN policy transfer; perhaps using the rulesets directly would perform as well or better.

To answer this question, I score each rule in the MLN policy rulesets in the same way that I score rules in macros, according to Table 5.5. At each step in the target task, I have my agents check all the rules, and instead of consulting an MLN to determine actions, they simply take the action recommended by the highest-scoring satisfied rule.

Figures 6.6 and 6.7 show the performance of this approach in 3-on-2 and 4-on-3 BreakAway, compared with RL-SVR and regular MLN policy transfer. These results show that MLN policy transfer does outperform ruleset policy transfer. The statistical analysis in Table 6.7 indicates that

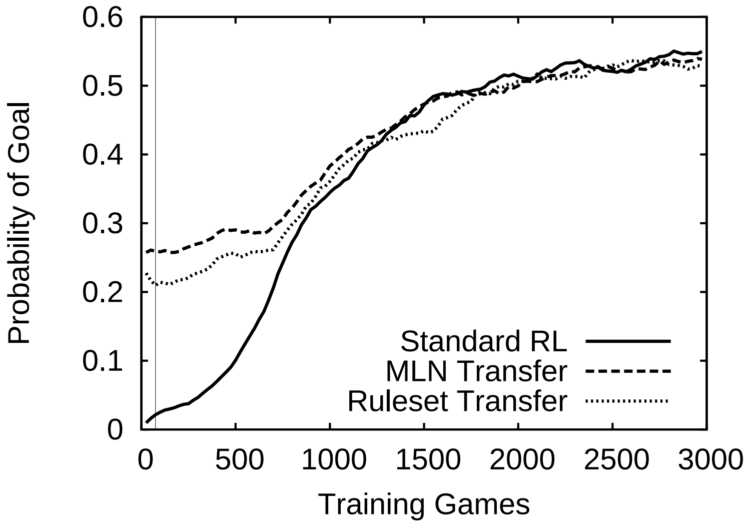

Figure 6.6: Probability of scoring a goal in 3-on-2 BreakAway with RL-SVR, regular MLN policy transfer from 2-on-1 BreakAway, and ruleset MLN policy transfer from 2-on-1 BreakAway. The thin vertical line marks the end of the demonstration period.

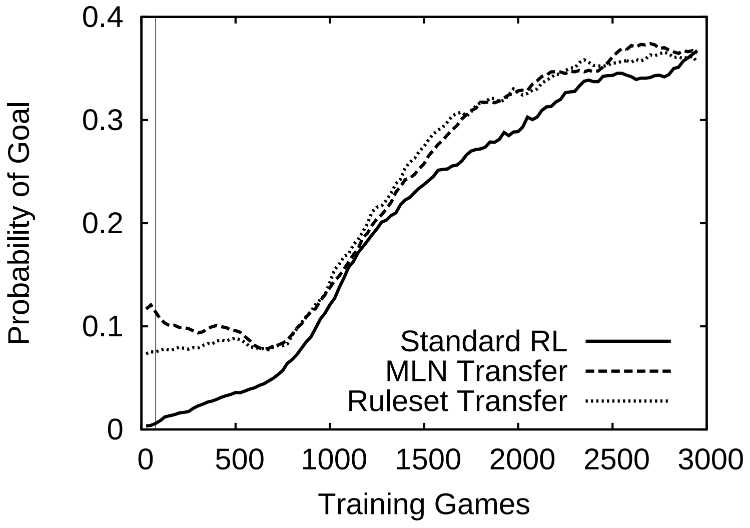

Figure 6.7: Probability of scoring a goal in 4-on-3 BreakAway with RL-SVR, regular MLN policy transfer from 2-on-1 BreakAway, and ruleset MLN policy transfer from 2-on-1 BreakAway. The thin vertical line marks the end of the demonstration period.

Table 6.7: Results of statistical tests on the differences between areas under the curve in ruleset policy transfer vs. regular MLN policy transfer for several source/target pairs. For $p < 0.05$, the difference is significant and the winning algorithm is shown; otherwise, the difference is not statistically significant.

Source	*Target*	*p-value*	*Significance (Winner)*	*95% CI*
2-on-1 BreakAway	3-on-2 BreakAway	0.014	Yes (Regular)	[8, 94]
2-on-1 BreakAway	4-on-3 BreakAway	0.382	No	[-56, 40]

the area under the curve decreases slightly in 3-on-2 BreakAway and remains equivalent in 4-on-3. In a 2-on-1 BreakAway self-transfer test, the ruleset policy scores in 53% of the episodes, compared to 65% for the regular MLN policy.

I conclude that MLNs do provide an additional benefit over ILP in the scenario of relational policy transfer, in both the source and target task. The reason they are beneficial in this context but not in macros is that there is no structure here for them to hinder. In both contexts, MLNs produce higher classification accuracy compared to their rulesets, primarily by decreasing the number of false positives. In macros, this caused too much caution in progressing through the strategy, but here all actions are possible from the same "node," so that interference does not occur.

This experiment also provides insight on the use of advice in a demonstration setting. These rulesets contain clauses that are comparable to the advice in the skill-transfer method of Chapter 4; the only difference is that there are more of them. Skills applied via demonstration would produce a target-task learning curve of similar shape. However, if I used only one clause per skill as I did in skill transfer, the initial performance would be lower due to lower coverage.

MLN policy transfer assumes the Markov property, in which the action choice depends only on the current environment and is independent of previous environments and actions. However, it need not do so; the MLN formulas for action choices could use such information. I examine the benefit of doing so by adding predicates to the ILP hypothesis space that specify previous actions. I add predicates for one, two, and three steps back in a game. Like macros, this approach allows transfer of both relational information and multi-state reasoning.

In these experiments, Aleph only chose to use the predicate for one step back, and never used the ones for two and three steps. This indicates that it is sometimes informative to know what the immediately previous action was, but beyond that point, action information is not useful. Aleph primarily used action predicates in rules for *move_away* and the shoot actions, both of which are sensible. Moving away can cause agents to go out of bounds, so it makes sense to consider that action only after other actions that bring the players further infield. Two consecutive shoot actions are typical of successful BreakAway games, so it makes sense that rules for shooting make reference to previous shoot actions.

Figures 6.8 and 6.9 show the performance of multi-step MLN policy transfer in 3-on-2 and 4-on-3 BreakAway, compared with RL-SVR and regular MLN policy transfer. These results show that adding action-sequence information does not improve MLN policy transfer. The statistical analysis in Table 6.8 indicates that the area under the curve decreases slightly in 3-on-2 BreakAway and remains equivalent in 4-on-3. In a 2-on-1 BreakAway self-transfer test, the multi-step MLN policy scores in 64% of the episodes, compared to 65% for the regular MLN policy.

This result indicates that the Markov property is a valid assumption in the BreakAway domain. While action patterns do exist, and the macro approach takes advantage of them, there is apparently enough information in the current state to make action choices independently. A multi-step MLN policy is therefore unnecessary in this domain, though it could be helpful in different domains where the Markov property does not hold.

Table 6.8: Results of statistical tests on the differences between areas under the curve in multi-step MLN policy transfer vs. regular MLN policy transfer for several source/target pairs. For $p < 0.05$, the difference is significant and the winning algorithm is shown; otherwise, the difference is not statistically significant.

Source	*Target*	*p-value*	*Significance (Winner)*	*95% CI*
2-on-1 BreakAway	3-on-2 BreakAway	0.047	Yes (Regular)	[-71, 3]
2-on-1 BreakAway	4-on-3 BreakAway	0.158	No	[-82, 25]

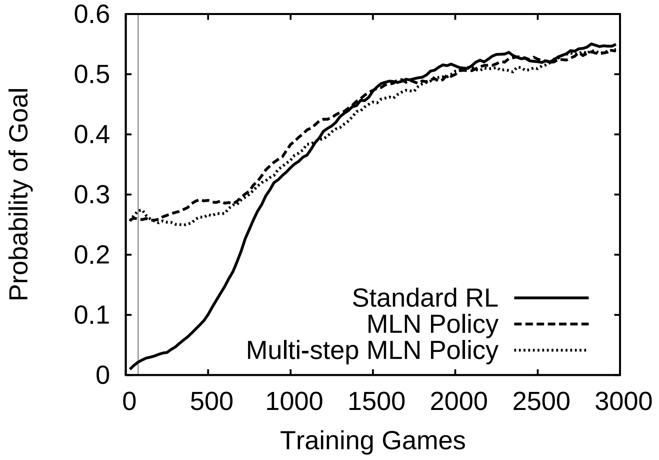

Figure 6.8: Probability of scoring a goal in 3-on-2 BreakAway with RL-SVR, regular MLN policy transfer from 2-on-1 BreakAway, and multi-step MLN policy transfer from 2-on-1 BreakAway. The thin vertical line marks the end of the demonstration period.

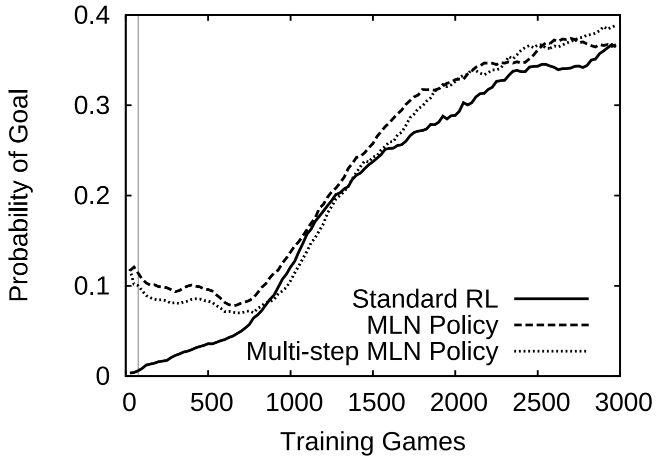

Figure 6.9: Probability of scoring a goal in 4-on-3 BreakAway with RL-SVR, regular MLN policy transfer from 2-on-1 BreakAway, and multi-step MLN policy transfer from 2-on-1 BreakAway. The thin vertical line marks the end of the demonstration period.

6.4 Summary of MLN Transfer

MLN transfer via demonstration can give the learner a significant head start in the target task. As in macro transfer, the performance level of the demonstrated strategy is unlikely to be as high as the target-task agent can achieve with further training, but the learner can improve its performance from the level of the demonstration up to its asymptote. As in macro transfer, the demonstration method lacks protection against negative transfer, which makes it appropriate only for close-transfer scenarios.

Transferring a policy with an MLN is a more natural and effective method than transferring a Q-function. Rulesets expressing a policy can be demonstrated effectively as well, but using an MLN to combine the rulesets provides additional benefits. An MLN captures complete enough information about the source task that adding knowledge about actions previously taken provides no additional benefit.

The advantage that MLN transfer has over related propositional methods is that it makes use of relational information present in the domain. It "lifts" the transferred information to the level of first-order logic, even if the source task was learned at a lower level. This makes the transferred knowledge more general and thus more easily applicable in some target tasks, and may be responsible for the lack of overspecialization seen in these experiments, despite the high level of detail that MLN models capture in the source task.

A potential reason to choose MLN transfer over relational-macro transfer given their comparable performance is that MLNs provide well-defined opportunities for refinement in the target task. Existing work on revision of MLNs, such as that of Mihalkova et al. [55], could be applied to this problem.

In my experiments with MLN transfer, I made an unexpected discovery: an MLN policy can outperform the source-task policy from which it was learned. There are two potential explanations for this surprising result. One is that learning the MLN policy from high-reward episodes across the entire learning curve provides additional training from a wide range of episodes. Another is that the relational nature of an MLN allows it to pool data across related actions and symmetrical features, making it an inherently more powerful model that can get more leverage out of less data. This result suggests that MLN policies could be used outside the context of transfer learning to improve standard RL, which I will discuss as future work.

Chapter 7

Conclusions and Future Work

This thesis presents original research evaluating three types of relational transfer for reinforcement learning: advice-based transfer, macro transfer, and MLN transfer. It contributes the six transfer algorithms listed in Table 1.1. Each algorithm produces a significant improvement over standard RL in experiments in the RoboCup simulated soccer domain.

Advice-based transfer methods use source-task knowledge to provide advice for a target-task learner, which can follow, refine, or ignore the advice according to its value. Relational advice is preferable to propositional advice because it increases generality. Human-provided advice can produce additional benefits. The most notable advantage of advice-based transfer is its built-in protection against negative transfer, which allows it to be applied to a wide range of transfer scenarios. The most notable disadvantage is that the performance gains do not appear immediately and are relatively modest.

Macro-transfer methods use source-task experience to form a macro-operator that demonstrates good behavior for a target-task learner. Since standard RL has to act mostly randomly in the early steps of a task, a good macro strategy can provide a large immediate advantage. Multiple macros may perform better than a single macro in some domains. The most notable disadvantage of macro transfer is its lack of protection against negative transfer. A more subtle disadvantage is the time that it takes to adjust the macro strategy appropriately to the target task. However, if there is limited time and the target task cannot be trained to its asymptote, then the immediate advantage that macros can provide may be worth the risk.

MLN-transfer methods use source-task experience to learn a Markov Logic Network that demonstrates good behavior for a target-task learner. An MLN that represents the source-task policy is an effective vehicle for transfer. MLN transfer produces performance comparable to macro transfer, with similar advantages and disadvantages. A potential reason to prefer MLN transfer is that it provides well-defined opportunities for refinement in the target task.

For the demonstration-based transfer methods, I found it interesting to test the relational model learned from the source task in a new run of the source task itself (self-transfer). Figure 7.1 shows the self-transfer scores of several macro-transfer and MLN-transfer methods together.

At first self-transfer was merely a debugging tool, but it turned out to highlight several interesting issues. One is the discovery that one of my relational models is powerful enough to outperform the source-task policy from which it is learned, which inspired one of the future-work areas later in this chapter. Another is the phenomenon of *overspecialization*, a transfer-learning issue that is related to the general machine-learning issue of overfitting. Source-task knowledge that produces high performance in self-transfer can be overspecialized to a source task, meaning that it does not generalize well to target tasks, especially as they grow more distant from the source.

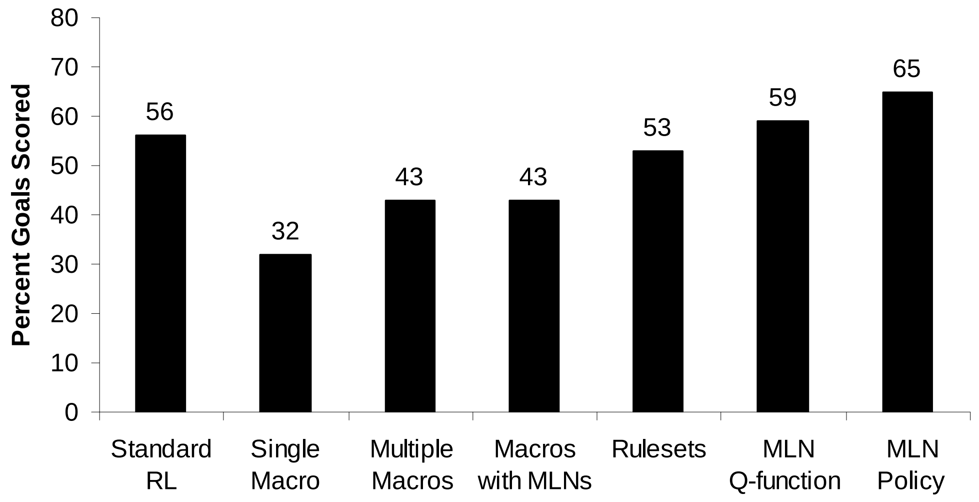

Figure 7.1: Self-transfer results: the percent of games in which a goal is scored in 2-on-1 BreakAway at the asymptote of standard RL, and during demonstration of several relational models learned from 2-on-1 BreakAway.

7.1 Algorithm Comparison

Given a transfer scenario, which algorithm from Table 1.1 should be used? Advice-based methods have the lowest initial performance in the target task, but their learning curves are steepest, and they are robust to negative transfer. Macro-transfer and MLN-transfer methods can have higher initial performance, but in doing so they sacrifice protection against negative transfer.

In close-transfer scenarios, such as the 2-on-1 BreakAway to 3-on-2 BreakAway experiment, a reasonable goal is to provide the best initial performance possible. To achieve this goal, the more aggressive demonstration-based approaches should be favored. Figure 7.2 shows the order in which I would recommend my transfer algorithms in such cases.

In distant-transfer scenarios, such as the KeepAway to BreakAway experiment or with even more distinct source and target tasks, a high initial performance is unlikely to be possible. Instead, a reasonable goal is to shorten the period of initial low performance as much as possible, while avoiding negative transfer. Here the more cautious advice-based approaches should be favored. Figure 7.3 shows the order in which I would recommend my transfer algorithms in such cases.

Some transfer scenarios could fall in between close and distant transfer, where it is less clear which algorithm would perform best. Currently, our own discretion as human facilitators of transfer is the only way to decide such cases. Better prediction of the relative performance of transfer algorithms is an important area of future work in transfer learning. A related important area is increasing the autonomy of transfer algorithms; that is, reducing the reliance on human input for choosing source tasks and providing mappings.

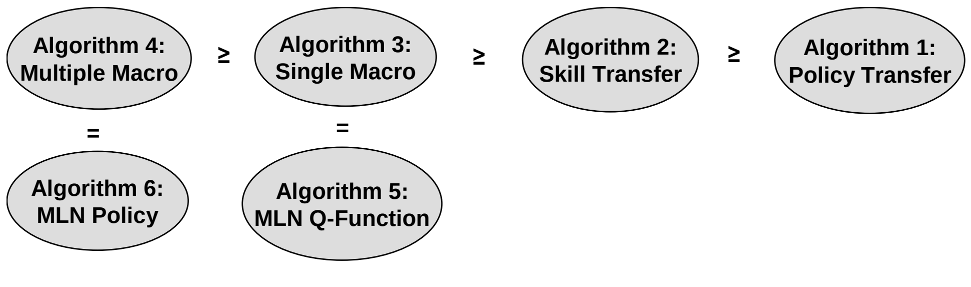
Algorithm 4: Multiple Macro
≥
Algorithm 3: Single Macro
≥
Algorithm 2: Skill Transfer
≥
Algorithm 1: Policy Transfer
=
=
Algorithm 6: MLN Policy
Algorithm 5: MLN Q-Function

Figure 7.2: A recommended ordering of my transfer algorithms in close-transfer scenarios. Demonstration-based algorithms will likely perform better than advice-based ones. Within each type, the more strongly relational and general algorithms will likely perform better.

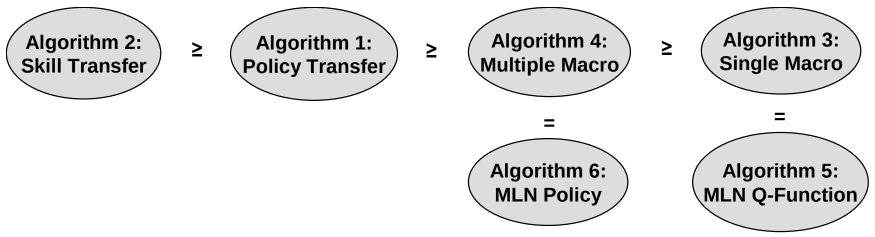
Algorithm 2: Skill Transfer
≥
Algorithm 1: Policy Transfer
≥
Algorithm 4: Multiple Macro
≥
Algorithm 3: Single Macro
=
=
Algorithm 6: MLN Policy
Algorithm 5: MLN Q-Function

Figure 7.3: A recommended ordering of my transfer algorithms in distant-transfer scenarios. Advice-based algorithms will likely perform better than demonstration-based ones. As before, however, within each type the more strongly relational and general algorithms will likely perform better.

7.2 Potential Extensions of My Research

At this point, I pause the reflection on future directions for transfer learning in general to discuss some specific extensions to this thesis that could be addressed by future research.

Transfer from Multiple Source Tasks

My experiments have focused on learning a target task with transferred knowledge from a single source task. An extension of interest might be to combine knowledge from multiple tasks. In some contexts, such as skill transfer and multiple-macro transfer, this could be a straightforward combination of relational knowledge. In other contexts, such as MLN policy transfer, it is less clear how to combine knowlege from multiple sources in the most effective way.

Negative Transfer and Overspecialization

My macro-transfer and MLN-transfer approaches behave similarly because they both use the demonstration method in a target task. In that sense, they could both be considered subtypes of *demonstration-based* transfer methods, which stand in contrast to advice-based methods. Demonstration is an aggressive method that can benefit greatly from the detailed policy knowledge that both macros and MLNs capture. However, it does not provide protection against negative transfer.

Macros and MLNs could be used in different ways that would provide such protection. The options framework [65], in which a macro or MLN would be treated as an alternative action with its own Q-value, is one possibility. Another potential approach would be interleaving demonstration with learning and decreasing the amount of demonstration over time. This might sacrifice some initial performance, but it could increase learning speed as well as providing protection against negative transfer.

Transfer methods that capture detailed knowledge from a source task may be susceptible to *overspecialization*. Protection against negative transfer in the target task can limit the negative effects of overspecialization. However, it could also be addressed more directly earlier in the transfer process, during the extraction of knowledge from the source task. Relational learning is one approach that may help to prevent overspecialization since it produces more general knowledge. Future work in this area could develop other approaches.

Theoretical Results

This thesis evaluates transfer algorithms experimentally. While this is an important form of evaluation, it would also be valuable to determine the theoretical bounds of their performance, as some of the work in Section 3.6 does. Bounds of interest include the following properties of a target-task learner with transfer (as compared to without transfer):

- The initial target-task performance
- The rate of improvement over time
- The number of episodes required to reach the asymptote

To answer questions like these, it will first be necessary to define a precise relationship between a source task and a target task. Furthermore, a rigorous analysis is likely to require a simple *Q*-learning algorithm and a simple domain, just as existing theoretical results for inductive transfer require simple Boolean classification tasks.

Joint Learning and Inference in Macros

In Chapter 5, the algorithms for learning macros perform independent ILP runs for structure and ruleset learning. As I note in that chapter, this is not the only possible approach. Some or all of these problems could be combined, requiring a joint solution.

There are some arguments in favor of a joint method. It strongly favors the development of a coherent, global plan. The value of joint inference for solving interdependent problems has been established in other areas, most notably in natural language processing (NLP). Often NLP tasks are solved by a pipeline of local steps, each one passing output to the next, but recent work has shown that joint inference can improve overall performance. For example, Sutton et al. [85] simultaneously perform the two tasks of tagging parts of speech and chunking noun phrases. Other applications of joint inference in NLP include Dietterich and Bao [19], who jointly infer the topics of emails and documents given the likely relationships revealed by attached documents, and Poon and Domingos [66], who perform joint inference to make unsupervised coreference resolution competitive with supervised.

In the context of macro learning, there are several levels of joint inference that could be considered. At one extreme, a single ILP search could be conducted through a possible space of macro clauses. This is probably infeasible, because the allowed clause length would need to be much larger, and the ILP search would take an overwhelming amount of time. A more practical method would be to still learn a macro structure separately, but then learn rulesets jointly. In MLN-enhanced macros, weights could also be learned jointly.

There is one important difference between NLP problems and the macro problem that affects joint-inference solutions: the element of time. When performing tagging and chunking in a test example, Sutton et al. [85] receive an entire sentence at a time; when executing a macro, a target-task learner receives information one state at a time. Thus decisions at early states cannot depend on decisions at later states. In a sense, joint learning in macros would need to be partially ordered.

Refinement of Transferred Knowledge

The demonstration-based algorithms in this thesis currently use transferred knowledge for only a fixed period of time in the target task. After the demonstration period, the target-task learners abruptly ignore their macros or MLNs and conduct standard RL. While it is certainly worthwhile to do further learning in the target task, it may be possible to do so instead by incrementally revising the source-task knowledge.

In macros, there are three levels of possible revision: the structure, the rulesets, and the rule scores. In MLNs, there are two levels: the formulas and the weights. A reasonable approach would be to make frequent revisions of lower-level knowledge, like weights and rule scores, and make less frequent revisions of higher-level knowledge, like first-order logical rules.

There is potentially a difference between revising rules and relearning them. Performing ILP searches frequently is quite inefficient, though it may be good to do occasionally. However, incremental changes to rules can be made efficiently without using ILP. For example, Mihalkova et al. [55] propose an algorithm for finding clauses that are too general or too specific and considering additions and deletions of literals in those clauses.

The speed of learning in the target task, compared to standard RL, would depend on the effectiveness of these revision strategies. The asymptotic performance, compared to standard RL, would depend on the type of transferred knowledge. A single macro is unlikely to reach asymptotic performance, no matter how well revised, but an MLN policy may.

Even if refinement of transferred knowlege does not produce tangible benefits in target-task learning, it may provide interesting information. Differences between the original knowledge and the refined knowledge could essentially define the differences between the source and target tasks.

Relational Reinforcement Learning

The problem of refining relational knowledge in a target task is closely related to the more general problem of relational reinforcement learning. Domains like RoboCup could benefit from relational RL, which would provide substantial generalization over objects and actions.

At first glance, the most logical candidate to serve as a basis for relational RL appears to be the MLN Q-function. An MLN could be used as a Q-function approximator instead of the more typical propositional model. It could be employed in a batch algorithm similar to RL-SVR, where agents play batches of games and relearn the MLN after every batch.

One interesting difference from RL-SVR is that the MLN Q-function provides a probability distribution over the Q-values of actions, rather than providing just a single value. As I noted previously, this property makes it related to Bayesian reinforcement learning [83]. If one used MLN Q-functions as the basis for relational RL, there would be opportunities for using the distribution information to conduct exploration in a more sophisticated way than the ϵ-greedy method.

However, some preliminary experiments indicate that the MLN *Q*-function may not be a good basis for relational RL due to some limitations in its regression accuracy. Consider a training example s used to learn the MLN for the current batch. Let $Q_a(s)$ be the *Q*-value in the previous batch for action a in state s. Let α be the learning-rate parameter, and let $T(s)$ be the temporal-difference value for state s, which is a function of the rewards received after state s. The training output for state s in the current batch is:

$$Q'_a(s) = (1 - \alpha) \times Q_a(s) + \alpha \times T(s) \tag{7.1}$$

If $Q'_a(s)$ became more accurate over time, as it does in RL-SVR, this would be an effective algorithm. However, it does not appear to do so, and this prevents the algorithm from learning effectively. The reason is that the *Q*-value estimates of an MLN *Q*-function are not precise, because of the binning and weighted-averaging strategy the algorithm uses.

In practice, an MLN typically assigns a high probability to the correct *Q*-value bin and low probabilities to the other bins, as in the example on the left in Figure 6.1. Thus it is performing good classification of examples into *Q*-value bins, and it ranks actions reliably for the first batch. However, since the *Q*-values are weighted averages, they are strongly biased towards the expected value of the correct bin. Rather than ranging across the *Q*-value space as they naturally would, they become essentially fixed near one value per bin. Thus the MLN is not performing good regression for *Q*-values, which prevents effective learning in subsequent batches.

Increasing the number of bins leads to some smoothing of *Q*-values, but it also makes the classification into bins more difficult, and the overall performance does not improve. The number of bins required to achieve high accuracy in *Q*-values cannot be supported by the amount of data available, especially early in the learning curve.

Another inherent modeling issue with bins is that the overall range of possible *Q*-values shrinks with each chunk. With its weighted average calculation, an MLN cannot produce a *Q*-value higher than the expected value of the rightmost bin (or lower than the expected value of the leftmost bin). Thus each model has a smaller possible *Q*-value range than the last, which is not a natural dynamic for *Q*-learning. While scaling could keep the range from shrinking, the correct behavior would actually be for the range to increase by some unknown amount.

Relational *Q*-learning therefore appears to require a model that can perform regression naturally, without losing information via binning. One possibility might be the recently proposed Hybrid MLN [109], which can contain real-valued nodes. Alternatively, moving away from *Q*-learning, relational RL could be approached with a policy-search algorithm based on macros from Section 5.3 or MLN policies from Section 6.2. Regardless of the method, the main challenge to overcome in performing relational RL in a complex domains is the computational cost of repeated stages of symbolic learning.

Bootstrapping Reinforcement Learning and Self-Training

The discovery that an MLN policy can outperform the source-task policy from which it was learned suggests that MLNs could be used to improve reinforcement learning outside the context of transfer. In a bootstrapping approach, one could alternate standard RL and MLN policy demonstration to speed up RL within a single task.

There are two conditions that must hold in order for an approach of this type to be effective. First, an MLN policy must provide benefits near the beginning of the source-task learning process and not only near the asymptote. Second, the source-task learner must be able to take advantage of short demonstrations efficiently.

MLN policies might also be useful in self-training [98], where an agent improves by playing games against a version of itself. Learning an MLN policy from an earlier version, rather than using the earlier version exactly, would give an agent an immediate advantage.

7.3 The Future of Transfer Learning

The challenges discussed in this thesis will remain relevant in future work on transfer learning in general. Humans appear to have natural mechanisms for deciding when to transfer information, for selecting appropriate sources of knowledge, and for determining the appropriate level of abstraction. It is not always clear how to make these decisions for a single machine learning algorithm, much less in general.

One general challenge for future work is to enable transfer between more diverse tasks. Davis and Domingos [17] provide a potential direction for this in their work on MLN transfer. They perform pattern discovery in the source task to find second-order formulas, which represent universal abstract concepts like symmetry and transitivity. When learning an MLN for the target task, they allow the search to use the discovered formulas in addition to the original predicates in the domain. This approach is recognizeable as inductive transfer, but the source-task knowledge is highly abstract, which allows the source and target tasks to differ significantly.

Yet another general challenge is to perform transfer in complex testbeds. Particularly in reinforcement learning, it can become much more difficult to achieve transfer as the source and target tasks become more complex. Since practical applications of reinforcement learning are likely to be highly complex, it is important not to limit research on RL transfer to simple domains. RoboCup is an important step toward realistic domains, but it is not a final step.

Transfer learning has become a sizeable subfield in machine learning. It has ideological benefits, because it is seen as an important aspect of human learning, and also practical benefits, because it can make machine learning more efficient. As computing power increases and researchers apply machine learning to more and more complex problems, abilities like knowledge transfer can only become more desirable.

LIST OF REFERENCES

[1] J. Abernethy, P. Bartlett, and A. Rakhlin. Multitask learning with expert advice. In *Computational Learning Theory*, San Diego, CA, 2007.

[2] R. Ando and T. Zhang. A framework for learning predictive structures from multiple tasks and unlabeled data. *Journal of Machine Learning Research*, 6:1817–1853, 2005.

[3] 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.

[4] J. Baxter. Theoretical models of learning to learn. In S. Thrun and L. Pratt, editors, *Learning to Learn*. Kluwer, 1997.

[5] J. Baxter. A model of inductive bias learning. *Journal of Artificial Intelligence Research*, 12:149–198, 2000.

[6] R. Bellman. A Markovian decision process. *Journal of Mathematics and Mechanics*, 6:679–684, 1957.

[7] S. Ben-David and R. Schuller. Exploiting task relatedness for multiple task learning. In *Computational Learning Theory*, Washington, DC, 2003.

[8] A. Botean, M. Enzenberger, M. Muller, and J. Schaeer. Macro-FF: Improving AI planning with automatically learned macro-operators. *Journal of Artificial Intelligence Research*, 24:581–621, 2005.

[9] A. Brown. Knowing when, where, and how to remember: A problem of metacognition. In R. Glaser, editor, *Advances in Instructional Psychology Vol. 1*, pages 77–166. Lawrence Erlbaum Associates, 1978.

[10] C. Carroll and K. Seppi. Task similarity measures for transfer in reinforcement learning task libraries. In *IEEE International Joint Conference on Neural Networks*, Montreal, Canada, 2005.

[11] R. Caruana. Multitask learning. *Machine Learning*, 28:41–75, 1997.

[12] D. Choi and P. Langley. Learning teleoreactive logic programs from problem solving. In *International Conference on Inductive Logic Programming*, Bonn, Germany, 2005.

[13] P. Cohen. *Empirical Methods for Artificial Intelligence*. MIT Press, 1995.

[14] 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.

[15] W. Dai, G. Xue, Q. Yang, and Y. Yu. Transferring Naive Bayes classifiers for text classification. In *AAAI Conference on Artificial Intelligence*, Vancouver, BC, 2007.

[16] W. Dai, Q. Yang, G. Xue, and Y. Yu. Boosting for transfer learning. In *International Conference on Machine Learning*, Corvallis, OR, 2007.

[17] J. Davis and P. Domingos. Deep transfer via second-order Markov logic. In *AAAI Workshop on Transfer Learning for Complex Tasks*, Chicago, IL, 2008.

[18] T. Dietterich. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000.

[19] T. Dietterich and X. Bao. Integrating multiple learning components through Markov logic. In *AAAI Conference on Artificial Intelligence*, Chicago, IL, 2008.

[20] P. Domingos. Personal communication, 2008.

[21] P. Domingos, S. Kok, H. Poon, M. Richardson, and P. Singla. Unifying logical and statistical AI. In *AAAI Conference on Artificial Intelligence*, Boston, MA, 2006.

[22] P. Domingos and M. Richardson. Markov logic: A unifying framework for statistical relational learning. In *ICML Workshop on Statistical Relational Learning and its Connections to Other Fields*, Banff, Canada, 2004.

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

[24] K. Driessens, J. Ramon, and T. Croonenborghs. Transfer learning for reinforcement learning through goal and policy parametrization. In *ICML Workshop on Structural Knowledge Transfer for Machine Learning*, Pittsburgh, PA, 2006.

[25] E. Eaton and M. DesJardins. Knowledge transfer with a multiresolution ensemble of classifiers. In *ICML Workshop on Structural Knowledge Transfer for Machine Learning*, Pittsburgh, PA, 2006.

[26] E. Eaton, M. DesJardins, and T. Lane. Modeling transfer relationships between learning tasks for improved inductive transfer. In *European Conference on Machine Learning*, Antwerp, Belgium, 2008.

[27] H. Ellis. *The Transfer of Learning*. The Macmillan Company, 1965.

[28] B. Falkenhainer, K. Forbus, and D. Gentner. The structure-mapping engine: Algorithm and examples. *Artificial Intelligence*, 41:1–63, 1989.

[29] 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.

[30] J. Flavell. Metacognitive aspects of problem solving. In L. Resnick, editor, *The Nature of Intelligence*, pages 231–236. Lawrence Erlbaum Associates, 1976.

[31] Y. Freund and R. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55:119–139, 1997.

[32] J. Gao, W. Fan, J. Jiang, and J. Han. Knowledge transfer via multiple model local structure mapping. In *International Conference on Knowledge Discovery and Data Mining*, Las Vegas, NV, 2008.

[33] L. Getoor and B. Taskar. *Introduction to Statistical Relational Learning*. MIT Press, 2007.

[34] A. Gill. *Introduction to the Theory of Finite-State Machines*. McGraw-Hill, 1962.

[35] M. Goadrich, L. Oliphant, and J. Shavlik. Gleaner: Creating ensembles of first-order clauses to improve recall-precision curves. *Machine Learning*, 64:231–262, 2006.

[36] H. Hlynsson. Transfer learning using the minimum description length principle with a decision tree application. Master's thesis, University of Amsterdam, Graduate Programme in Logic, 2007.

[37] H. Daume III. *Practical Structured Learning Techniques for Natural Language Processing*. PhD thesis, University of Southern California, Department of Computer Science, 2006.

[38] F. Jensen. *An Introduction to Neural Networks*. Springer Verlag, 1996.

[39] M. Klenk and K. Forbus. Measuring the level of transfer learning by an AP physics problem-solver. In *AAAI Conference on Artificial Intelligence*, Vancouver, BC, 2007.

[40] S. Kok, P. Singla, M. Richardson, and P. Domingos. The Alchemy system for statistical relational AI. http://alchemy.cs.washington.edu, 2005.

[41] G. Konidaris and A. Barto. Autonomous shaping: Knowledge transfer in reinforcement learning. In *International Conference on Machine Learning*, Pittsburgh, PA, 2006.

[42] G. Kuhlmann and P. Stone. Graph-based domain mapping for transfer learning in general games. In *European Conference on Machine Learning*, Warsaw, Poland, 2007.

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

[44] A. Lazaric, M. Restelli, and A. Bonarini. Transfer of samples in batch reinforcement learning. In *International Conference on Machine Learning*, Helsinki, Finland, 2008.

[45] Y. Liu and P. Stone. Value-function-based transfer for reinforcement learning using structure mapping. In *AAAI Conference on Artificial Intelligence*, Boston, MA, 2006.

[46] D. Lowd and P. Domingos. Efficient weight learning for Markov logic networks. In *Knowledge Discovery in Databases*, Warsaw, Poland, 2007.

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

[48] 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.

[49] 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.

[50] 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.

[51] M. Madden and T. Howley. Transfer of experience between reinforcement learning environments with progressive difficulty. *Artificial Intelligence Review*, 21:375–398, 2004.

[52] O. Mangasarian, J. Shavlik, and E. Wild. Knowledge-based kernel approximation. *Journal of Machine Learning Research*, 5:1127–1141, 2004.

[53] Z. Marx, M. Rosenstein, L. Kaelbling, and T. Dietterich. Transfer learning with an ensemble of background tasks. In *NIPS Workshop on Transfer Learning*, Vancouver, BC, 2005.

[54] 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.

[55] L. Mihalkova, T. Huynh, and R. Mooney. Mapping and revising Markov logic networks for transfer learning. In *AAAI Conference on Artificial Intelligence*, Vancouver, BC, 2007.

[56] L. Mihalkova and R. Mooney. Transfer learning with Markov logic networks. In *ICML Workshop on Structural Knowledge Transfer for Machine Learning*, Pittsburgh, PA, 2006.

[57] T. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[58] A. Niculescu-Mizil and R. Caruana. Inductive transfer for Bayesian network structure learning. In *Conference on AI and Statistics*, San Juan, Puerto Rico, 2007.

[59] 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.

[60] G. Obozinski, B. Taskar, and M. Jordan. Joint covariate selection and joint subspace selection for multiple classification problems. *Statistics and Computing*, to appear, 2009.

[61] T. Odlin. *Language Transfer: Cross-Linguistic Influence in Language Learning*. Cambridge University Press, 1989.

[62] S. Pan, J. Kwok, and Q. Yang. Transfer learning via dimensionality reduction. In *AAAI Conference on Artificial Intelligence*, Chicago, IL, 2008.

[63] S. Pan, I. Tsang, J. Kwok, and Q. Yang. Domain adaptation via transfer component analysis. In *International Joint Conference on Artificial Intelligence*, Pasadena, CA, 2009.

[64] M. Pazzani and D. Kibler. The utility of background knowledge in inductive learning. *Machine Learning*, 9:57–94, 1992.

[65] T. Perkins and D. Precup. Using options for knowledge transfer in reinforcement learning. Technical Report UM-CS-1999-034, University of Massachusetts, Amherst, 1999.

[66] H. Poon and P. Domingos. Joint unsupervised coreference resolution with Markov logic. In *Conference on Empirical Methods in Natural Language Processing*, Honolulu, HI, 2008.

[67] B. Price and C. Boutilier. Implicit imitation in multiagent reinforcement learning. In *International Conference on Machine Learning*, Bled, Slovenia, 1999.

[68] L. De Raedt. *Logical and Relational Learning*. Springer, 2008.

[69] R. Raina, A. Ng, and D. Koller. Constructing informative priors using transfer learning. In *International Conference on Machine Learning*, Pittsburgh, PA, 2006.

[70] M. Richardson and P. Domingos. Markov logic networks. *Machine Learning*, 62:107–136, 2006.

[71] M. Rosenstein, Z. Marx, L. Kaelbling, and T. Dietterich. To transfer or not to transfer. In *NIPS Workshop on Inductive Transfer*, Vancouver, BC, 2005.

[72] U. Ruckert and S. Kramer. Kernel-based inductive transfer. In *European Conference on Machine Learning*, Antwerp, Belgium, 2008.

[73] D. Rumelhart, B. Widrow, and M. Lehr. The basic ideas in neural networks. *Communications of the ACM*, 37:87–92, 1994.

[74] 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.

[75] A. Sherstov and P. Stone. Action-space knowledge transfer in MDPs: Formalism, suboptimality bounds, and algorithms. In *Computational Learning Theory*, Bertinoro, Italy, 2005.

[76] X. Shi, W. Fan, and J. Ren. Actively transfer domain knowledge. In *European Conference on Machine Learning*, Antwerp, Belgium, 2008.

[77] D. Silver, R. Poirier, and D. Currie. Inductive transfer with context-sensitive neural networks. *Machine Learning*, 73:313–336, 2008.

[78] S. Singh. Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning*, 8:323–339, 1992.

[79] V. Soni and S. Singh. Using homomorphisms to transfer options across continuous reinforcement learning domains. In *AAAI Conference on Artificial Intelligence*, Boston, MA, 2006.

[80] A. Srinivasan. The Aleph manual. http://www.comlab.ox.ac.uk/activities/machinelearning/Aleph/aleph.html, 2001.

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

[82] D. Stracuzzi. Memory organization and knowledge transfer. In *ICML Workshop on Structural Knowledge Transfer for Machine Learning*, Pittsburgh, PA, 2006.

[83] M. Strens. A Bayesian framework for reinforcement learning. In *International Conference on Machine Learning*, Stanford University, CA, 2000.

[84] C. Sutton and A. McCallum. Composition of conditional random fields for transfer learning. In *Conference on Empirical Methods in Natural Language Processing*, Vancouver, BC, 2005.

[85] C. Sutton, K. Rohanimanesh, and A. McCallum. Factorized probabilistic models for labeling and segmenting sequence data. In *International Conference on Machine Learning*, 2004.

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

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

[88] E. Talvitie and S. Singh. An experts algorithm for transfer learning. In *International Joint Conference on Artificial Intelligence*, Hyderabad, India, 2007.

[89] F. Tanaka and M. Yamamura. Multitask reinforcement learning on the distribution of MDPs. *Transactions of the Institute of Electrical Engineers of Japan*, 123:1004–1011, 2003.

[90] B. Taskar, V. Chatalbashev, D. Koller, and C. Guestrin. Learning structured prediction models: A large margin approach. In *International Conference on Machine Learning*, 2005.

[91] M. Taylor, N. Jong, and P. Stone. Transferring instances for model-based reinforcement learning. In *European Conference on Machine Learning*, Antwerp, Belgium, 2008.

[92] M. Taylor, G. Kuhlmann, and P. Stone. Autonomous transfer for reinforcement learning. In *Conference on Autonomous Agents and Multi-Agent Systems*, Estoril, Portugal, 2008.

[93] M. Taylor, G. Kuhlmann, and P. Stone. Accelerating search with transferred heuristics. In *ICAPS Workshop on AI Planning and Learning*, Providence, RI, 2007.

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

[95] M. Taylor and P. Stone. Transfer via inter-task mappings in policy search reinforcement learning. In *Conference on Autonomous Agents and Multi-Agent Systems*, Honolulu, HI, 2007.

[96] 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.

[97] 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.

[98] G. Tesauro. Temporal difference learning and TD-gammon. *Communications of the ACM*, 38:58–68, 1995.

[99] E. Thorndike. *Principles of Teaching*. Mason Henry, 1906.

[100] S. Thrun and T. Mitchell. Learning one more thing. In *International Joint Conference on Artificial Intelligence*, Montreal, Quebec, 1995.

[101] L. Torrey and J. Shavlik. Transfer learning. In E. Soria, J. Martin, R. Magdalena, M. Martinez, and A. Serrano, editors, *Handbook of Research on Machine Learning Applications*. IGI Global, 2009.

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

[103] 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.

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

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

[106] 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.

[107] 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.

[108] C. Wang and S. Mahadevan. Manifold alignment using Procrustes analysis. In *International Conference on Machine Learning*, Helsinki, Finland, 2008.

[109] J. Wang and P. Domingos. Hybrid Markov logic networks. In *AAAI Conference on Artificial Intelligence*, Chicago, IL, 2008.

[110] C. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8:279–292, 1992.

[111] U. Weinreich. *Languages in Contact*. Mouton, 1953.

[112] 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.

[113] F. Zelezny, A. Srinivasan, and D. Page. Lattice-search runtime distributions may be heavy-tailed. In *International Conference on Inductive Logic Programming*, Sydney, Australia, 1995.

[114] J. Zhang, Z. Ghahramani, and Y. Yang. Flexible latent variable models for multi-task learning. *Machine Learning*, 73:221–242, 2008.

[115] X. Zhu. Semi-supervised learning literature survey. Technical Report Computer Sciences TR 1530, University of Wisconsin-Madison, 2005.

DISCARD THIS PAGE

NOMENCLATURE

This is a glossary of useful terms. Some terms have specific meanings in the context of this thesis, and may have other significance elsewhere.

Advice: a set of instructions about a task solution that may not be complete or perfectly correct.

Advice-based transfer: my term for a set of transfer algorithms that use source-task knowledge as advice in a target task.

Alchemy: an implementation of Markov Logic Networks by the group of Pedro Domingos at the University of Washington.

Aleph: an implementation of inductive logic programming by Ashwin Srinivasan.

Alteration methods: my term for transfer methods in reinforcement learning that change the state space, action space, or reward function of a target task based on source-task knowledge.

Agent: a term used interchangeably with *learner*.

Batch: a set of episodes in the RL-SVR reinforcement-learning algorithm that are performed sequentially with the same policy; the policy is updated after each batch.

Bayesian reinforcement learning: a type of reinforcement learning that models distributions of values, and to which MLN Q-functions are related.

BreakAway: a reinforcement-learning subtask in the RoboCup simulated soccer domain.

Classification: a mapping from a feature space to a set of labels.

Clause: a disjunction of literals; in this thesis I use Horn clauses, which contain a head implied by a conjunction of non-negated literals.

Close transfer: transfer between closely related tasks.

Demonstration: my term for transfer methods that use a source-task policy for an initial period in a target task.

Distant transfer: transfer between less similar tasks.

ϵ-greedy: a popular method of balancing exploration and exploitation in reinforcement learning, where the learner mostly exploits its current policy but explores randomly in a small fraction (ϵ) of steps.

Episode: a segment of training in reinforcement learning that has a defined beginning and end.

Exploitation: when a reinforcement learner takes the action recommended by its current policy in order to maximize its reward.

Exploration: when a reinforcement learner takes an action not recommended by its current policy in order to discover new information.

ExtenKBKR: a variant of Preference-KBKR that is designed to handle many pieces of advice.

F-measure: a scoring function for clauses that balances recall and precision.

Feature: in this thesis, refers to one of a set of properties that describe a state in a reinforcement learner's environment.

First-order logic: a system for representing logical statements that, unlike propositional logic, allows the use of quantified variables.

Formula: in this thesis, one of a set of statements in first-order logic in a Markov Logic Network.

Gleaner: a system that extracts a wide range of useful clausesencountered during an Aleph search.

Head: a literal implied by other literals in a Horn clause.

Hierarchical learning: any type of learning that involves simpler tasks combined to learn more complex tasks.

Imitation methods: my term for transfer methods in reinforcement learning that apply a source-task policy to choose some actions while learning the target task.

Inductive bias: the set of assumptions that an inductive learner makes about the concept it induces.

Inductive learning: a type of machine learning in which a predictive model is induced from a set of training examples.

Inductive logic programming (ILP): a set of methods for learning classifiers in first-order logic.

KBKR: knowledge-based kernel regression, a method for solving a regression problem that includes advice, upon which my advice-based transfer algorithms depend.

KeepAway: a reinforcement-learning subtask in the RoboCup simulated soccer domain.

Learning curve: a plot reporting the performance of a reinforcement learner over time.

Literal: a statement of a property of the world that may be either true or false.

Macro-operator (macro): a composition of primitive actions into a useful group; in this thesis, a relational macro is a relational finite-state machine that describes a successful action sequence in a task.

Macro transfer: my term for a set of transfer algorithms that learn successful source-task action sequences for use in a target task.

Mapping: a description of the correspondences between source and target tasks in transfer learning or analogical reasoning.

Markov Logic Network (MLN): a model that expresses concepts with first-order clauses but that also indicates probability by putting weights on clauses.

MLN policy transfer: my term for my proposed Transfer Algorithm 6 6.2.

MLN Q-function transfer: my term for my proposed Transfer Algorithm 5 6.1.

MLN transfer: my term for a set of transfer algorithms that learn Markov Logic Networks to express source-task knowledge for use in a target task.

Model-free: a type of reinforcement learning in which the environment is not modeled.

Model-learning: a type of reinforcement learning in which the environment is modeled.

MoveDownfield: a reinforcement-learning subtask in the RoboCup simulated soccer domain.

Multi-task learning: methods in machine learning that learn multiple tasks simultaneously.

Multiple-macro transfer: my term for my proposed Transfer Algorithm 4 5.3.

Negative transfer: a decrease in learning performance in a target task due to transfer learning.

Node: one of a sequence of internal states in a macro.

Option: a high-level action in reinforcement learning that involves several lower-level actions.

Overfitting: when a machine-learning algorithm models its training data too closely, thus treating spurious patterns as important and not generalizing well even to data drawn from the same distribution.

Overspecialization: my term for when a transfer algorithm models a source task too closely, thus not generalizing well to target tasks that differ from the source.

p-value: a statistic that measures how confident one can be that two sets of numbers are significantly different.

Policy: the mechanism by which a reinforcement-learning agent chooses which action to execute next.

Policy-search: a type of reinforcement learning in which a policy is directly and iteratively updated.

Policy transfer: my term for my proposed Transfer Algorithm 1 4.2.

Precision: the fraction of examples a classifier calls positive that are truly positive.

Preference-KBKR: an advice-taking system based on KBKR that accepts advice saying to prefer one action over another.

Propositional logic: a system for representing logical statements that, unlike first-order logic, does not allow the use of quantified variables.

Q*-function:* a function incrementally learned by a reinforcement learner to predict the expected long-term reward after taking an action in a state.

Q*-learning:* a popular algorithm for reinforcement learning that involves learning a Q-function.

Q*-value:* the expected long-term reward for a reinforcement learner after taking an action in a state.

Recall: the fraction of truly positive examples that a classifier correctly calls positive.

Regression: a mapping from a feature space to a real value.

Reinforcement learning (RL): a type of machine learning in which an agent learns through experience to navigate through an environment, choosing actions in order to maximize rewards.

Relational knowledge: information about relationships between objects, expressed in first-order logic.

Relational reinforcement learning: a type of reinforcement learning in which states are expressed in first-order logic rather than in fixed-length feature vectors.

Reward: a real-valued reinforcement received by a reinforcement learner when it takes an action.

RoboCup: a simulated soccer domain that has been adapted for reinforcement learning.

RL-SVR: an algorithm used to implement reinforcement learning via support-vector regression.

SARSA: a variant of Q-learning that takes exploration steps into account during updates.

Self-transfer: transfer learning when the source and target task are the same, often used in this thesis as a way to measure the completeness of transferred knowledge.

Single-macro transfer: my term for my proposed Transfer Algorithm 3 5.2.

Skill: a rule in first-order logic that describe good conditions under which to take an action.

Skill transfer: my term for my proposed Transfer Algorithm 2 4.3.

Source task: a task that a learner has already learned and from which it transfers knowledge.

Starting-point methods: my term for transfer methods in reinforcement learning that set the initial solution in the target task based on knowledge from a source task.

State: in this thesis, refers to one of many possible settings of the features in a reinforcement learner's environment.

Statistical-relational learning: a type of machine learning that combines paradigms of logic and probability.

Support-vector machine: a classification approach that constructs a hyperplane to separate data into classes by maximizing the margin between the training data; also a regression approach that fits a hyperplane to the training data by related methods.

Target task: a task in which learning is improved through knowledge transfer.

Temporal-difference methods: algorithms for reinforcement learning that iteratively update value functions.

Theory: a set of clauses learned by inductive logic programming to describe a concept.

Tiling: discretizing continuous features into intervals and adding these intervals as additional Boolean features to enhance the description of a reinforcement-learning environment.

Transfer learning: methods in machine learning that improve learning in a target task by transferring knowledge from one or more related source tasks.

Value function: a function incrementally learned by a reinforcement learner to predict the value of a state or action.

APPENDIX
A. RoboCup Feature and Action Spaces

This appendix provides information that was omitted from earlier chapters for readability: the features and actions for the RoboCup tasks.

Table A.1 shows the action spaces for KeepAway, MoveDownfield, and BreakAway, the three RoboCup tasks used for experiments in this thesis, and Table A.2 shows the feature spaces.

Player objects are numbered in order of increasing current distance to the player with the ball. The functions *minDistTaker(Keeper)* and *minAngleTaker(Keeper)* evaluate to the player objects *t0*, *t1*, and so on that are currently closest in distance and angle respectively to the given Keeper object. Similarly, the functions *minDistDefender(Attacker)* and *minAngleDefender(Attacker)* each evaluate to one of the player objects *d0*, *d1*, etc.

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

Table A.1: RoboCup task action spaces.

KeepAway actions	
pass(Teammate)	Teammate ∈ {k1, k2, ...}
holdBall	
MoveDownfield actions	
pass(Teammate)	Teammate ∈ {a1, a2, ...}
move(Direction)	Direction ∈ {ahead, away, left, right}
BreakAway actions	
pass(Teammate)	Teammate ∈ {a1, a2, ...}
move(Direction)	Direction ∈ {ahead, away, left, right}
shoot(GoalPart)	GoalPart ∈ {goalRight, goalLeft, goalCenter}

Table A.2: RoboCup task feature spaces.

KeepAway features	
distBetween(k0, Player)	Player ∈ {k1, k2, ...} ∪ {t0, t1, ...}
distBetween(Keeper, minDistTaker(Keeper))	Keeper ∈ {k1, k2, ...}
angleDefinedBy(Keeper, k0, minAngleTaker(Keeper))	Keeper ∈ {k1, k2, ...}
distBetween(Player, fieldCenter)	Player ∈ {k0, k1, ...} ∪ {t0, t1, ...}
MoveDownfield features	
distBetween(a0, Player)	Player ∈ {a1, a2, ...} ∪ {d0, d1, ...}
distBetween(Attacker, minDistDefender(Attacker))	Attacker ∈ {a1, a2, ...}
angleDefinedBy(Attacker, a0, minAngleDefender(Attacker))	Attacker ∈ {a1, a2, ...}
distToRightEdge(Attacker)	Attacker ∈ {a0, a1, ...}
timeLeft	(in tenths of seconds)
BreakAway features	
distBetween(a0, Player)	Player ∈ {a1, a2, ...} ∪ {d0, d1, ...}
distBetween(Attacker, minDistDefender(Attacker))	Attacker ∈ {a1, a2, ...}
angleDefinedBy(Attacker, a0, minAngleDefender(Attacker))	Attacker ∈ {a1, a2, ...}
distBetween(Attacker, GoalPart)	Attacker ∈ {a0, a1, ...}
distBetween(Attacker, goalie)	Attacker ∈ {a0, a1, ...}
angleDefinedBy(Attacker, a0, goalie)	Attacker ∈ {a1, a2, ...}
angleDefinedBy(GoalPart, a0, goalie)	GoalPart ∈ {goalRight, goalLeft, goalCenter}
angleDefinedBy(topRightCorner, goalCenter, a0)	
timeLeft	(in tenths of seconds)

APPENDIX
B. Propositional Mappings

This appendix provides information that was omitted from earlier chapters for readability: the mappings I provide for policy transfer between RoboCup tasks.

Mappings for policy transfer, as introduced in Section 4.2, match up propositional features and actions between the source and target. As described in Section 4.2.1, actions may have zero, one, or multiple mappings. For each mapped source-target action pair, each source-task feature may map to a single target-task feature, or it may map to a constant.

Tables B.1, B.2, and B.3 show the propositional mappings I use for close-transfer scenarios, where the target task is the same as the source task except with different numbers of players. In these scenarios, the target-task action space is the source-task action space plus an additional *pass* action for the new teammate, and the feature space is the source-task feature space plus some additional angles and distances for the new teammate. Thus every source-task action and feature has at least one mapping, and one *pass* action has multiple mappings in order to provide information for all of the target-task actions.

Tables B.4 and B.5 show the propositional mappings I use for distant-transfer scenarios, where the source and target tasks differ. In these scenarios, there are some actions in the source-task action space that do not have mappings, and there are some source-task features that map to constants because they have no matching target-task features. Also of note are features that refer to the furthest opponent ($d1$ or $t1$ in the source tasks), which I map to features that refer to the furthest opponent in the target tasks ($d2$ or $t2$).

Table B.1: Propositional mappings from 2-on-1 BreakAway to 3-on-2 BreakAway, used in policy transfer.

Source: 2-on-1 BreakAway	*Target*: 3-on-2 BreakAway
Actions	*Mapped actions*
move(ahead)	move(ahead)
move(away)	move(away)
move(left)	move(left)
move(right)	move(right)
shoot(goalCenter)	shoot(goalCenter)
shoot(goalLeft)	shoot(goalLeft)
shoot(goalRight)	shoot(goalRight)
pass(a1)	pass(a1)
Features	*Mapped features for the above actions*
distBetween(a0, a1)	distBetween(a0, a1)
distBetween(a0, goalie)	distBetween(a0, goalie)
distBetween(a1, goalie)	distBetween(a1, goalie)
angleDefinedBy(a1, a0, goalie)	angleDefinedBy(a1, a0, goalie)
distBetween(a0, goalLeft)	distBetween(a0, goalLeft)
distBetween(a0, goalRight)	distBetween(a0, goalRight)
distBetween(a0, goalCenter)	distBetween(a0, goalCenter)
angleDefinedBy(goalLeft, a0, goalie)	angleDefinedBy(goalLeft, a0, goalie)
angleDefinedBy(goalRight, a0, goalie)	angleDefinedBy(goalRight, a0, goalie)
angleDefinedBy(goalCenter, a0, goalie)	angleDefinedBy(goalCenter, a0, goalie)
angleDefinedBy(topRightCorner, goalCenter, a0)	angleDefinedBy(topRightCorner, goalCenter, a0)
timeLeft	timeLeft
Actions with additional mappings	*Mapped actions*
pass(a1)	pass(a2)
Features	*Mapped features for the above actions*
distBetween(a0, a1)	distBetween(a0, a2)
distBetween(a0, goalie)	distBetween(a0, goalie)
distBetween(a1, goalie)	distBetween(a2, goalie)
angleDefinedBy(a1, a0, goalie)	angleDefinedBy(a2, a0, goalie)
distBetween(a0, goalLeft)	distBetween(a0, goalLeft)
distBetween(a0, goalRight)	distBetween(a0, goalRight)
distBetween(a0, goalCenter)	distBetween(a0, goalCenter)
angleDefinedBy(goalLeft, a0, goalie)	angleDefinedBy(goalLeft, a0, goalie)
angleDefinedBy(goalRight, a0, goalie)	angleDefinedBy(goalRight, a0, goalie)
angleDefinedBy(goalCenter, a0, goalie)	angleDefinedBy(goalCenter, a0, goalie)
angleDefinedBy(topRightCorner, goalCenter, a0)	angleDefinedBy(topRightCorner, goalCenter, a0)
timeLeft	timeLeft

Table B.2: Propositional mappings from 3-on-2 MoveDownfield to 4-on-3 MoveDownfield, used in policy transfer.

3-on-2 MoveDownfield	*4-on-3 MoveDownfield*
Actions	*Mapped actions*
move(ahead)	move(ahead)
move(away)	move(away)
move(left)	move(left)
move(right)	move(right)
pass(a1)	pass(a1)
pass(a2)	pass(a2)
Features	*Mapped features for the above actions*
distBetween(a0, a1)	distBetween(a0, a1)
distBetween(a0, a2)	distBetween(a0, a2)
distBetween(a0, d0)	distBetween(a0, d0)
distBetween(a0, d1)	distBetween(a0, d2)
distBetween(a1, minDistDefender(a1))	distBetween(a1, minDistDefender(a1))
distBetween(a2, minDistDefender(a2))	distBetween(a2, minDistDefender(a2))
angleDefinedBy(a1, a0, minAngleDefender(a1))	angleDefinedBy(a1, a0, minAngleDefender(a1))
angleDefinedBy(a2, a0, minAngleDefender(a2))	angleDefinedBy(a2, a0, minAngleDefender(a2))
distToRightEdge(a0)	distToRightEdge(a0)
distToRightEdge(a1)	distToRightEdge(a1)
distToRightEdge(a2)	distToRightEdge(a2)
timeLeft	timeLeft
Actions with additional mappings	*Mapped actions*
pass(a2)	pass(a3)
Features	*Mapped features for the above actions*
distBetween(a0, a1)	distBetween(a0, a1)
distBetween(a0, a2)	distBetween(a0, a3)
distBetween(a0, d0)	distBetween(a0, d0)
distBetween(a0, d1)	distBetween(a0, d2)
distBetween(a1, minDistDefender(a1))	distBetween(a1, minDistDefender(a1))
distBetween(a2, minDistDefender(a2))	distBetween(a3, minDistDefender(a3))
angleDefinedBy(a1, a0, minAngleDefender(a1))	angleDefinedBy(a1, a0, minAngleDefender(a1))
angleDefinedBy(a2, a0, minAngleDefender(a2))	angleDefinedBy(a3, a0, minAngleDefender(a3))
distToRightEdge(a0)	distToRightEdge(a0)
distToRightEdge(a1)	distToRightEdge(a1)
distToRightEdge(a2)	distToRightEdge(a3)
timeLeft	timeLeft

Table B.3: Propositional mappings from 3-on-2 KeepAway to 4-on-3 KeepAway, used in policy transfer.

3-on-2 KeepAway	*4-on-3 KeepAway*
Actions	*Mapped actions*
holdBall	holdBall
pass(k1)	pass(k1)
pass(k2)	pass(k2)
Features	*Mapped features for the above actions*
distBetween(k0, k1)	distBetween(k0, k1)
distBetween(k0, k2)	distBetween(k0, k2)
distBetween(k0, t0)	distBetween(k0, t0)
distBetween(k0, t1)	distBetween(k0, t2)
distBetween(k1, minDistKeeper(k1))	distBetween(k1, minDistKeeper(k1))
distBetween(k2, minDistKeeper(k2))	distBetween(k2, minDistKeeper(k2))
angleDefinedBy(k1, k0, minAngleKeeper(k1))	angleDefinedBy(k1, k0, minAngleKeeper(k1))
angleDefinedBy(k2, k0, minAngleKeeper(k2))	angleDefinedBy(k2, k0, minAngleKeeper(k2))
distBetween(k0, fieldCenter)	distBetween(k0, fieldCenter)
distBetween(k1, fieldCenter)	distBetween(k1, fieldCenter)
distBetween(k2, fieldCenter)	distBetween(k2, fieldCenter)
distBetween(t0, fieldCenter)	distBetween(t0, fieldCenter)
distBetween(t1, fieldCenter)	distBetween(t2, fieldCenter)
Actions with additional mappings	*Mapped actions*
pass(k2)	pass(k3)
Features	*Mapped features for the above actions*
distBetween(k0, k1)	distBetween(k0, k1)
distBetween(k0, k2)	distBetween(k0, k3)
distBetween(k0, t0)	distBetween(k0, t0)
distBetween(k0, t1)	distBetween(k0, t2)
distBetween(k1, minDistKeeper(k1))	distBetween(k1, minDistKeeper(k1))
distBetween(k2, minDistKeeper(k2))	distBetween(k3, minDistKeeper(k3))
angleDefinedBy(k1, k0, minAngleKeeper(k1))	angleDefinedBy(k1, k0, minAngleKeeper(k1))
angleDefinedBy(k2, k0, minAngleKeeper(k2))	angleDefinedBy(k3, k0, minAngleKeeper(k3))
distBetween(k0, fieldCenter)	distBetween(k0, fieldCenter)
distBetween(k1, fieldCenter)	distBetween(k1, fieldCenter)
distBetween(k2, fieldCenter)	distBetween(k3, fieldCenter)
distBetween(t0, fieldCenter)	distBetween(t0, fieldCenter)
distBetween(t1, fieldCenter)	distBetween(t2, fieldCenter)

Table B.4: Propositional mappings from 3-on-2 MoveDownfield to 3-on-2 BreakAway, used in policy transfer.

3-on-2 MoveDownfield	*3-on-2 BreakAway*
Actions	*Mapped actions*
move(ahead)	move(ahead)
move(away)	move(away)
move(left)	move(left)
move(right)	move(right)
pass(a1)	pass(a1)
pass(a2)	pass(a2)
Features	*Mapped features for the above actions*
distBetween(a0, a1)	distBetween(a0, a1)
distBetween(a0, a2)	distBetween(a0, a2)
distBetween(a0, d0)	distBetween(a0, d0)
distBetween(a0, d1)	distBetween(a0, d0)
distBetween(a1, minDistDefender(a1))	distBetween(a1, minDistDefender(a1))
distBetween(a2, minDistDefender(a2))	distBetween(a2, minDistDefender(a2))
angleDefinedBy(a1, a0, minAngleDefender(a1))	angleDefinedBy(a1, a0, minAngleDefender(a1))
angleDefinedBy(a2, a0, minAngleDefender(a2))	angleDefinedBy(a2, a0, minAngleDefender(a2))
distToRightEdge(a0)	distBetween(a0, goalCenter)
distToRightEdge(a1)	distBetween(a1, goalCenter)
distToRightEdge(a2)	distBetween(a2, goalCenter)
timeLeft	timeLeft

Table B.5: Propositional mappings from 3-on-2 KeepAway to 3-on-2 BreakAway, used in policy transfer.

3-on-2 KeepAway	*3-on-2 BreakAway*
Actions	*Mapped actions*
holdBall	
pass(k1)	pass(a1)
pass(k2)	pass(a2)
Features	*Mapped features for the above actions*
distBetween(k0, k1)	distBetween(a0, a1)
distBetween(k0, k2)	distBetween(a0, a2)
distBetween(k0, t0)	distBetween(a0, d0)
distBetween(t0, t1)	distBetween(a0, d0)
distBetween(k1, minDistTaker(k1))	distBetween(a1, minDistDefender(a1))
distBetween(k2, minDistTaker(k2))	distBetween(a2, minDistDefender(a2))
angleDefinedBy(k1, k0, minAngleTaker(k1))	angleDefinedBy(a1, a0, minAngleDefender(a1))
angleDefinedBy(k2, k0, minAngleTaker(k2))	angleDefinedBy(a2, a0, minAngleDefender(a2))
distBetween(k0, fieldCenter)	15
distBetween(k1, fieldCenter)	15 // No match in target, so use
distBetween(k2, fieldCenter)	15 // average value in source
distBetween(t0, fieldCenter)	10
distBetween(t1, fieldCenter)	10

APPENDIX
C. Object Mappings

This appendix provides information that was omitted from earlier chapters for readability: the mappings I provide for relational transfer between RoboCup tasks.

Mappings for relational transfer, as described in Section 4.3, match up objects in the source and target tasks. These mappings also apply to macro transfer in Chapter 5 and to MLN transfer in Chapter 6. In RoboCup, the objects are players and goal parts.

For 2-on-1 BreakAway to 3-on-2 BreakAway and 4-on-2 BreakAway, I map all objects to objects of the same name:

a0	⟶	a0
a1	⟶	a1
goalie	⟶	goalie
goalLeft	⟶	goalLeft
goalRight	⟶	goalRight
goalCenter	⟶	goalCenter

For 3-on-2 MoveDownfield to 4-on-3 MoveDownfield, I map most objects to objects of the same name, but the furthest teammate and opponent ($a2$ and $d1$ in the source) have different names in the target task ($a3$ and $d2$):

a0	⟶	a0
a1	⟶	a1
a2	⟶	a3
d0	⟶	d0
d1	⟶	d2
minDistDefender(a1)	⟶	minDistDefender(a1)
minDistDefender(a2)	⟶	minDistDefender(a3)
minAngleDefender(a1)	⟶	minAngleDefender(a1)
minAngleDefender(a2)	⟶	minAngleDefender(a3)

For 3-on-2 KeepAway to 4-on-3 KeepAway, the situation is similar:

k0	$\longrightarrow$	k0
k1	$\longrightarrow$	k1
k2	$\longrightarrow$	k3
t0	$\longrightarrow$	t0
t1	$\longrightarrow$	t2
minDistTaker(k1)	$\longrightarrow$	minDistTaker(k1)
minDistTaker(k2)	$\longrightarrow$	minDistTaker(k3)
minAngleTaker(k1)	$\longrightarrow$	minAngleTaker(k1)
minAngleTaker(k2)	$\longrightarrow$	minAngleTaker(k3)

For 3-on-2 MoveDownfield to 3-on-2 BreakAway, I map most objects to objects of the same name, but I map the second opponent $d1$ to $d0$ instead of the goalie because the goalie behaves quite differently from a mobile defender:

a0	$\longrightarrow$	a0
a1	$\longrightarrow$	a1
a2	$\longrightarrow$	a2
d0	$\longrightarrow$	d0
d1	$\longrightarrow$	d0
minDistDefender(a1)	$\longrightarrow$	minDistDefender(a1)
minDistDefender(a2)	$\longrightarrow$	minDistDefender(a2)
minAngleDefender(a1)	$\longrightarrow$	minAngleDefender(a1)
minAngleDefender(a2)	$\longrightarrow$	minAngleDefender(a2)

For 3-on-2 KeepAway to 3-on-2 BreakAway, the situation is similar:

k0	$\longrightarrow$	a0
k1	$\longrightarrow$	a1
k2	$\longrightarrow$	a2
t0	$\longrightarrow$	d0
t1	$\longrightarrow$	d0
minDistTaker(k1)	$\longrightarrow$	minDistDefender(a1)
minDistTaker(k2)	$\longrightarrow$	minDistDefender(a2)
minAngleTaker(k1)	$\longrightarrow$	minAngleDefender(a1)
minAngleTaker(k2)	$\longrightarrow$	minAngleDefender(a2)

APPENDIX
D. Conditional Independence Proof

This appendix supplies a proof that decision nodes in my MLNs are conditionally independent given evidence at all other nodes. This is important because it allows the larger problem of inferring the probabilities of multiple decision nodes to be split into independent problems. I prove conditional independence for two nodes, specifically the two in Figure 5.12 from the thesis, but it holds true in general for any number of decision nodes that have direct links only to evidence nodes.

Recall that each MLN formula $f_i \in F$, with weight w_i, has a number $n_i(x)$ of true groundings in each possible world x, and that the probability that the world x is the correct one is, from Equation 2.7 in Section 2.4:

$$P(X = x) = \frac{1}{Z}\, exp \sum_{i \in F} w_i n_i(x) \tag{D.1}$$

Consider calculating the probabilities of the *pass(a1)* and *pass(a2)* nodes in Figure 5.12 given evidence at all the other nodes. Let $x = (x_1, x_2)$ represent a possible world, where x_1 represents the truth value for the *pass(a1)* node and x_2 represents the truth value for the *pass(a2)* node. The four possible worlds are $\{(0,0),(0,1),(1,0),(1,1)\}$.

If x_1 and x_2 are conditionally independent, then the following is true:

$$P(X = (x_1, x_2)) = P(X_1 = x_1)P(X_2 = x_2) \tag{D.2}$$

Because the evidence literals are all known, the count $n_i(x)$ can be divided into two separate counts. These are $n_i(x_1)$, the number of true groundings given that *Teammate = a1*, and $n_i(x_2)$, the number of true groundings given that *Teammate = a2*. Inserting these into Equation D.1 gives:

$$P(X = (x_1, x_2)) = \frac{1}{Z}\, exp \sum_{i \in F} w_i [n_i(x_1) + n_i(x_2)] \tag{D.3}$$

$$= \frac{1}{Z}\, exp \sum_{i \in F} w_i n_i(x_1)\; exp \sum_{i \in F} w_i n_i(x_2) \tag{D.4}$$

Because the formulas are conjuncts that include at most one decision node, some of the counts are known to be zero. Note that $n_i(x_1) = 0$ when $x_1 = 0$, since there are no true groundings with ***Teammate = a1*** if $Teammate \neq a1$, and likewise $n_i(x_2) = 0$ when $x_2 = 0$. This means that Equation D.4 can be simplified for the following worlds:

$$P(X = (0, 0)) = \frac{1}{Z}\, exp(0)\; exp(0) = \frac{1}{Z} \tag{D.5}$$

$$P(X = (0, 1)) = \frac{1}{Z}\, exp \sum_{i \in F} w_i n_i(x_2) \tag{D.6}$$

$$P(X = (1, 0)) = \frac{1}{Z}\, exp \sum_{i \in F} w_i n_i(x_1) \tag{D.7}$$

Recall that Z is just a normalizing factor so that the probabilities of both worlds sum to 1:

$$P(X = (0, 0)) + P(X = (0, 1)) + P(X = (1, 0)) + P(X = (1, 1)) = 1 \tag{D.8}$$

By substituting Equations D.4, D.5, D.6, and D.7 into Equation D.8, the value of Z can be calculated as:

$$Z = \left(1 + \; exp \sum_{i \in F} w_i n_i(x_1)\right) \left(1 + \; exp \sum_{i \in F} w_i n_i(x_2)\right) \tag{D.9}$$

Substituting Z back into Equation D.4 gives:

$$P(X = (x_1, x_2)) = \frac{exp \sum_{i \in F} w_i n_i(x_1)}{1 + \; exp \sum_{i \in F} w_i n_i(x_1)} \; \frac{exp \sum_{i \in F} w_i n_i(x_2)}{1 + \; exp \sum_{i \in F} w_i n_i(x_2)} \tag{D.10}$$

Equation D.10 is equivalent to the independence assertion:

$$P(X = (x_1, x_2)) = P(X_1 = x_1)P(X_2 = x_2) \tag{D.11}$$

This completes the proof that the decision nodes are conditionally independent.