In Matthew E. Taylor and Karl Tuyls, editors, *Adaptive Agents and Multi-Agent Systems IV*, LNAI, pp. 1-33, Springer-Verlag, 2010.

Abstraction and Generalization in Reinforcement Learning: A Summary and Framework

Marc Ponsen[1], Matthew E. Taylor[2], and Karl Tuyls[1]

[1] Universiteit Maastricht, Maastricht, The Netherlands
{*m.ponsen,k.tuyls*}*@maastrichtuniversity.nl*

[2] The University of Southern California, Los Angeles, CA
taylorm@usc.edu

Abstract. In this paper we survey the basics of reinforcement learning, generalization and abstraction. We start with an introduction to the fundamentals of reinforcement learning and motivate the necessity for generalization and abstraction. Next we summarize the most important techniques available to achieve both generalization and abstraction in reinforcement learning. We discuss basic function approximation techniques and delve into hierarchical, relational and transfer learning. All concepts and techniques are illustrated with examples.

1 Introduction

In this chapter we provide an introduction to the concepts of generalization and abstraction in reinforcement learning (RL). Abstraction is a technique to reduce the complexity of a problem by filtering out irrelevant properties while preserving all the important ones necessary to still be able solve a given problem. Generalization is a technique to apply knowledge previously acquired to unseen circumstances or extend that knowledge beyond the scope of the original problem. Humans show great capability in abstracting and generalizing knowledge in everyday life. RL needs abstraction and generalization as well to deal successfully with contemporary technological challenges, given the huge state and action spaces that characterize real world problems. Recently, abstraction and generalization have received significant attention in the machine learning research community, resulting in a variety of techniques.

We start by introducing the preliminaries of RL itself in Section 2. We will discuss Markov decision processes, policy and value iteration and model-free solution techniques. In Section 3 we define both abstraction and generalization, capturing common features of both found in different definitions in literature, and then describe different operators in a concrete domain, the video-game Wargus. Section 5 gives a concise introduction to function approximation, one of the most commonly used types of methods in RL for generalization and abstraction. Sections 6-8 go into greater detail discussing three classes of techniques used for abstraction and generalization in RL: hierarchical, relational, and transfer learning. In addition to outlining the ideas behind each of these classes of techniques, we present results to assist the reader in understanding how these ideas may be applied in practice, and provide multiple references for additional exposition. Finally, Section 9 concludes.

The goals of this survey are to provide an introduction to, and framework for, discussing abstraction and generalization in RL domains. The article does not provide discussions at an advanced level but merely tries to combine the basics into one coherent structure, such that newcomers to the field easily understand the elementary concepts of abstraction and generalization in RL and have pointers available to more elaborate and detailed expositions in the literature.

2 Reinforcement Learning

This section introduces basic reinforcement learning concepts and notation.

2.1 Markov decision processes

Most RL research is framed as using a Markov decision processes (MDP) [29]. MDPs are sequential decision making problems for fully observable worlds. They are defined by a tuple (s_0, t, S, A, T, R). Starting in an initial state s_0 (or set of states) at each discrete time-step $t = 0, 1, 2, \dots$ an adaptive agent observes an environment state s_t contained in a set of states $S = \{s_1, s_2, \dots, s_n\}$, and executes an action a from a finite set of admissible actions $A = \{a_1, a_2, \dots, a_m\}$. The agent receives an immediate reward $R : S \rightarrow \mathbb{R}$, that assigns a value or reward for being in that state, and moves to a new state s', depending on a probabilistic transition function $T : S \times A \times S \rightarrow [0, 1]$. The probability of reaching state s' after executing action a in state s is denoted as $T(s, a, s')$. For all actions a, and all states s and s', $0 \leq T(s, a, s') \leq 1$ and $\sum_{s' \in S} T(s, a, s') = 1$. An MDP respects the *Markov property*: the future dynamics, transitions and rewards fully depend on the current state: $T(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \dots) = T(s_{t+1}|s_t, a_t)$ and $R(s_{t+1}|s_t, s_{t-1}, \dots) = R(s_{t+1})$. The transition function T and reward function R together are often referred to as the *model* of the environment. The learning task in an MDP is to find a policy $\pi : S \rightarrow A$ for selecting actions with maximal expected (discounted) reward. The quality of a policy is indicated by a *value function* V^π. The value $V^\pi(s)$ specifies the total amount of reward which an agent may expect to accumulate over the future, starting from state s and then following the policy π. Informally, the value function indicates the long-term desirability of states or state-action pairs after taking into account the states that may follow, and the rewards available in those states. In a discounted infinite horizon MDP, the expected cumulative reward (i.e., the value function) is denoted as:

$$V^\pi(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t)|s_0 = s\right] \tag{1}$$

A discount factor $\gamma \in [0, 1\rangle$ may be introduced to ensure that the rewards returned are bounded (finite) values. The variable γ determines the relevance of future rewards in the update. Setting γ to 0 results in a *myopic* update (i.e., only the immediate reward is optimized), whereas values closer to 1 will increase the contribution of future rewards in the update.

The value for a given policy π, expressed by Equation 1, can iteratively be computed by the *Bellman Equation* [3]. One typically starts with an arbitrarily chosen value

function, and at each iteration for each state $s \in S$, the value function is updated based on the immediate reward and the current estimate of V^π:

$$V_{t+1}^\pi(s) = R(s) + \gamma \sum_{s' \in S} T(s, \pi(s), s') V_n^\pi(s') \tag{2}$$

The process of updating state value functions based on current estimates of successor state values is referred to as *bootstrapping*. The depth of successor states considered in the update can be varied, i.e., one can perform a shallow bootstrap where one only looks at immediate successor states or a deep bootstrap where successors of successors are also considered. The value functions of successor states are used to update the value function of the current state. This is called a *backup* operation. Different algorithms use different backup strategies, e.g., sample backups (sample a single successor state) or full backups (sample all successor states).

The solution to an MDP is the *optimal policy*, i.e., the policy that receives the maximum reward. The optimal policy $\pi^*(s)$ is defined such that $V^{\pi^*}(s) \geq V^\pi(s)$ for all $s \in S$ and all policies π. The optimal value function, often abbreviated as V^*following Bellman optimality criterion:

$$V^*(s) = R(s) + \gamma \max_{\alpha \in A} \left[\sum_{s' \in S} T(s, a, s') V^*(s') \right] \tag{3}$$

Solving Equation 3 can be done in an iterative manner, similar to the computation of the value function for a given policy such as expressed in Equation 2. The Bellman optimality criterion is turned into an update rule:

$$V_{t+1}^\pi(s) = R(s) + \gamma \max_{\alpha \in A} \left[\sum_{s' \in S} T(s, a, s') V_n^\pi(s') \right] \tag{4}$$

The optimal action can then be selected as follows:

$$\pi^*(s) = \arg\max_a \left[R(s) + \gamma \sum_{s' \in S} T(s, a, s') V^*(s') \right] \tag{5}$$

Besides learning state-values, one can also define state-action value functions, also called *action-value functions*, or *Q-functions*. Q-functions map state-action pairs to values, $Q : S \times A \rightarrow \mathbb{R}$. They reflect the long term desirability of performing action a in state s, and then performing policy π thereafter. Learning Q-functions is particularly useful when T is unknown. The Q-function is defined as follows:

$$Q^*(s, a) = R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') V^*(s') \tag{6}$$

The optimal policy π^* selects the action which maximizes the optimal action value function $Q^*(s, a)$ for each state $s \in S$:

$$\pi^*(s) = \arg\max_a Q^*(s, a) \tag{7}$$

Algorithm 1: Policy Iteration

```
REQUIRE initialize V(s) and π(s) arbitrarily;
POLICY EVALUATION;
repeat
    Δ = 0;
    foreach s ∈ S do
        v = V(s);
        V(s) = R(s, π(s)) + γ Σ_{s'∈S} T(s, π(s), s') V(s');
        Δ = max(Δ, |v − V(s)|);
    end
until Δ < σ ;
POLICY IMPROVEMENT;
policy-stable = true;
foreach s ∈ S do
    b = π(s);
    π(s) = arg max_a [R(s, a) + γ Σ_{s'∈S} T(s, a, s') V(s')];
    if b ≠ π(s) then policy-stable = false
end
if policy-stable then stop else go to POLICY EVALUATION
```

2.2 Solution techniques

When an environment's model (i.e., transition function T and reward function R) is known, the optimal policy can be computed using a dynamic programming approach, such as in *policy iteration* and *value iteration*. Policy iteration [18] consists of two steps, a policy *evaluation* and policy *improvement* step. It starts with an arbitrary policy and value functions. It then updates the value functions under the given policy (the evaluation step), and uses the new value functions to improve its policy (the improvement step). Each policy is guaranteed to be a strict improvement over the previous one. The algorithm requires an infinite number of iterations to converge, but in practice the algorithm can be stopped when value functions only change by a small amount. A complete description is given in Algorithm 1.

The drawback of policy iteration is that it requires a complete evaluation of the current policy before improvements are made. Another possibility is to make improvements after a single sweep (a single backup of a state). This particular case is called *value iteration* [3]. Value iteration (or greedy iteration) starts with an arbitrary action-value function and for each state it iterates over all actions (unlike policy iteration which only evaluates the action as indicated by the policy) and updates the action-value function. The value iteration backup is identical to the policy evaluation backup except that it requires the maximum to be taken over all actions. Similar to policy iteration, the algorithm can be stopped when the change in policy is within a certain bound. Algorithm 2 gives a complete description of value iteration.

There exist several model-based learning methods, such as Dyna-Q [38, 51] and R-Max [6], but we will not go into much detail here because we are most interested in domains where the model is assumed to be both unknown and too complex to easily

Algorithm 2: Value Iteration

```
REQUIRE initialize V(s) arbitrarily;
repeat
    Δ = 0;
    foreach s ∈ S do
        v = V(s);
        foreach a ∈ A(s) do
            Q(s,a) = R(s,a) + γ Σ_{s'∈S} T(s,a,s')V(s')
        end
        V(s) = max_a Q(s,a);
        Δ = max(Δ, |v − V(s)|);
    end
until Δ < σ ;
```

learn. When the model of the environment is unknown, as it usually is, we can use RL as a viable alternative. RL does not depend on a model but rather collects samples from the environment to estimate the environment's model. Therefore, the crucial distinction between model-free and model-based methods is that the first samples future states whereas the second does a full sweep of successor states. Through exploration the reinforcement learner gathers data (i.e., rewards and future states) and uses this to learn a policy. An important issue that occurs is the exploration and exploitation dilemma, i.e., when to cease exploration and to start exploiting acquired knowledge. Various exploration and exploitation strategies exist, such as ϵ-greedy and Boltzmann exploration. For a thorough overview, we refer interested readers elsewhere [52, 39]. Temporal difference learning methods such as Q-learning [50] and SARSA [33] are model-free solution methods. The algorithms are described in detail in [39]. The update rule for one of the most popular algorithms, *one-step* Q-learning is:

$$Q(a,s) \rightarrow (1-\alpha)Q(a,s) + \alpha \left[R(s,a) + \gamma \max_{a'} Q(a',s') \right] \tag{8}$$

where α is the step-size parameter, and γ the discount-rate. This algorithm is proven to converge to an optimal policy in the limit (under reasonable conditions). Unfortunately, for many complex, real-world problems, solving the MDP is impractical and complexity must be reduced in order to keep learning tractable.

In the following sections we will discuss several ways to reduce the search space, so that learning with RL is still possible in more challenging domains (i.e., domains with large or infinite state spaces).

3 Abstraction and Generalization

In order to make RL feasible in complex domains, abstraction or generalization operators are often applied to make the problem tractable. We describe these operators in the current section and then give concrete examples in the following section.

Abstraction and generalization are important concepts in artificial intelligence (AI). Some claim that the ability to abstract and generalize is the essence of human intelligence [7] and that finding good representations is the primary challenge in designing intelligent systems. However, systems that learn and discover useful representations automatically are scarce. Instead, this problem is often tackled by the human designer.

A consistent definition of abstraction in the AI literature is not available: typically the definitions are tailored to specific subfields of AI, e.g, planning and problem solving [17], theorem proving [16], knowledge representation (e.g., spatial and temporal reasoning), machine learning, and computer vision [53]. The general principle underlying all these definitions is that an abstraction operation maps a representation of a problem onto a new representation so as to simplify reasoning while preserving useful properties. One only considers what is relevant and ignores many less important details for solving a particular task. Readers interested in a survey of state abstraction techniques in MDPs, as well an initial attempt to unify them, are referred elsewhere [24].

In problem solving and theorem proving, abstraction may be associated with a transformation of the problem representation that allows a theorem to be proved (or a problem to be solved) more easily with reduced computational complexity. This form of abstraction first abstracts a goal, proves or solves the abstracted goal, and then uses the structure of this abstracted proof to help construct the proof of the original goal. This method relies on the assumption that the structure of the abstracted proof is similar to the structure of the original goal. Another form of abstraction, as used in knowledge representation, machine learning, and computer vision, focuses more on the conceptualization of a domain, i.e., finding appropriate concepts or features of a domain. In this paper we will adopt the following definition for abstraction:

Definition 1 (Abstraction). *An abstraction operation changes the representation of an object by hiding or removing less critical details while preserving desirable properties. By definition, this implies loss of information.*

This definition is rather general and covers several different abstraction operations. In this paper we will adopt Zucker's taxonomy [53] to further categorize the different abstraction types. These abstraction operations are defined and explained with the help of a concrete example in the next section.

For generalization we employ the following definition:

Definition 2 (Generalization). *A generalization operation defines similarities between objects. This operation does not affect the object's representation. By definition, this implies no loss of information.*

For example, we may hypothesize that all rectangles are similar in some way. A strict definition of generalization states that all rectangles are a subset of its generalized hypothesis (e.g., all rectangles have 4 sides), but typically in machine learning, hypothesis are approximated and allow errors. For example, when stating that all rectangles have equal length sides, it is possible that some rectangles are outside of the hypothesis space (namely, all non square rectangles). Therefore, a weaker definition of generalization states that we have good evidence that all rectangles behave in a similar way. The generalization power measures the quality of the hypothesis on future examples.

We will next describe abstraction and generalization opportunities for RL in a concrete example, namely for learning a policy for a virtual agent in the computer game of Wargus.

4 An Illustrative Example

Fig. 1. A complex learning task

$$w = \begin{bmatrix} agent & structure & forest & rock & sand \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ ... & ... & ... & ... & ... \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Fig. 2. A naive state representation, where rows represent the observations of states (in this case, a pixel) and the columns represent the features used to describe the world

One example application that can benefit from reinforcement learning is computer games. Figure 1 is a screen shot of the computer game Wargus. In this figure we see an agent that is surrounded by bushes and buildings. This agent's responsibility (a peasant in the game) is a typical resource gathering task: it must travel to the goldmine (situated in the top right corner) and gather gold. We will tackle this learning task within the RL framework. The action space will contain the actions for moving in all directions. We assume that the transition function is unknown due to the complex and dynamic nature of the game environment. We define the agent's reward signal to be as follows: a small negative reward for each step and a positive (or zero) reward when completing the task. The difficult part is finding an appropriate state representation for this task. The state complexity in our world can be expressed by m^n, where n represents the number of grid cells and m the number of objects in the world. The state complexity is thus exponential in the dimension of the world and polynomial in the number of objects. A naive state representation (see Figure 2) for our example application would be to consider the smallest particle of this world (in this case a pixel in this 2-dimensional computer game world with dimension 500×500) to be a single grid cell, and then assuming that each

grid cell can be part of any of five different objects (which is already a simplification). For example, in Figure 2 the first row indicates that the first pixel is part of a forest object. When using this representation, learning a policy that directs the agent to the goldmine would be infeasible, due to the large state space, requiring the value function to contain 25000^5 distinct values. For any complex computer game, when modeling the world described as above, none of the standard RL approaches will converge to a decent policy in a reasonable amount of time. Rather than devising new update rules for RL, a more promising approach is to find more compact task representations (i.e., make the problem space simpler) and generalize over similar states. In other words, we need to apply appropriate abstractions and generalizations. We will next describe five different abstraction operations as defined by Zucker [53] that can scale down the problem complexity. We will apply these five techniques consecutively to our challenging problem to reduce complexity.

4.1 Domain Reduction

Domain Reduction is an abstraction operation that reduces part of the domain (i.e., content or instances) by grouping content together. Content refers to the observations of states (i.e., the row vectors in our world matrix). Before we evaluated each single pixel, so that our world matrix contained 25000 state observations (one for each grid cell in our world). The matrix in Figure 2 is a reformulation of the image in Figure 1: it applies a different notation for the same object without losing any information. We can reduce the content, i.e., reduce the number of state observations by making sets of grid cells indistinguishable. In our example we can choose to group neighboring pixels together to form a larger prototype grid cell. As a result, the world is divided in larger grid cells, as illustrated in Figure 3. An observation in our world matrix now covers several pixels, and therefore attribute values are real-valued percentages (averages over the covered pixels) rather than booleans (see Figure 4). The number of pixels grouped together to form a grid cell can be increased, but a coarser view of the world necessitates information loss. The tradeoff between information loss and the quality of the learned policy can be tuned, depending on task requirements.

4.2 Domain Hiding

Domain hiding is an abstraction operation that hides part of the domain, focusing on relevant content or objects in the domain. This is one of the most common form of abstraction. As mentioned before, content refers to the state observations (row vectors in our matrix). Rather than reducing the number of state observations (by grouping them), domain hiding simply ignores less relevant state observations. For example, in our task we want the agent to learn a policy that directs it to the gold mine. Therefore, we are not necessarily interested in some parts of the world, and we hide these state observations. We take the world that was the result of domain reduction as our input and apply domain hiding. The result is shown in Figure 5. In our matrix representation, a domain hiding operation can be performed by deleting observations, whereas domain reduction averages observations together.

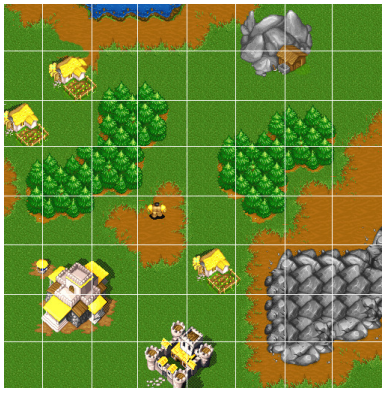

Fig. 3. Domain reduction

$$w = \begin{bmatrix} agent & structure & forest & rock & sand \\ 0 & 0 & .7 & .3 & 0 \\ 0 & .3 & .3 & 0 & .4 \\ ... & ... & ... & ... & ... \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & .9 & .1 \end{bmatrix}$$

Fig. 4. State representation of world after domain reduction

Fig. 5. Domain hiding

$$w = \begin{bmatrix} agent & structure & forest & rock & sand \\ 0 & 0 & .7 & .3 & 0 \\ 0 & .3 & .3 & 0 & .4 \\ ... & ... & ... & ... & ... \\ \cancel{0} & \cancel{1} & \cancel{0} & \cancel{0} & \cancel{0} \\ \cancel{0} & \cancel{0} & \cancel{0} & \cancel{.9} & \cancel{.1} \end{bmatrix}$$

Fig. 6. State representation of world after domain hiding

4.3 Co-Domain Hiding

Co-domain hiding is an abstraction operation that hides part of the co-domain (i.e., description) of an object by selectively paying attention to subsets of useful features in a given task. With the co-domain, we refer to the features of our world. In the state matrix, this is represented by the column vectors. Co-domain hiding ignores columns that are not relevant for the task. For example, in Figure 7, the sand feature is removed from the description since it is believed this feature does not contribute to an improvement for the agent's policy. Notice that the **sand** in Figure 7 and the sand column in Figure 8's matrix have been removed.

Fig. 7. Co-domain hiding

$$w = \begin{array}{c} \begin{matrix} agent & structure & forest & rock & sand \end{matrix} \\ \begin{bmatrix} 0 & 0 & .7 & .3 & 0 \\ 0 & .3 & .3 & 0 & .4 \\ ... & ... & ... & ... & ... \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & .9 & .1 \end{bmatrix} \end{array}$$

Fig. 8. State representation of world after co-domain hiding

4.4 Co-Domain Reduction

Co-domain reduction is an abstraction operation that reduces part of the co-domain (i.e., description) by making sets of attribute values indistinguishable. This implies reducing the range of values an attribute may take. In Figure 8 we see attribute values ranging from 0 to 1. We can apply abstractions by reducing the range of attribute values.

This can be achieved by applying some threshold function (injective mapping). For example, if a certain cell is covered with an object by more than 50 percent, in our new world representation this cell is now covered completely with this object, whereas objects that cover less than 50 percent are abstracted away from the matrix representation (see Figure 10). Effectively, we transform real numbers (i.e., percentages of objects in a grid cell) to boolean values, just as we saw in Figure 2, but now the boolean values do not correspond to pixels, but to composite grid cells.

4.5 Domain Aggregation

Domain Aggregation is an abstraction operation that aggregates (combines) parts of the domain (i.e., content). Content (or objects) are grouped together and form a new object with its own unique properties and parameters. In our example, we can choose to group objects together that obstruct the agent such as forests, structures or rocks. We group these objects together to form a complete new object, namely an obstacle (see Figure 11).

4.6 Generalization

A generalization operation is different from an abstraction operation in that it does not change an object's representation and, therefore, does not lose any information. Instead it claims generalities between objects, leaving the original objects untouched. In our example we can make a generalized hypothesis that forests and rocks are equivalent

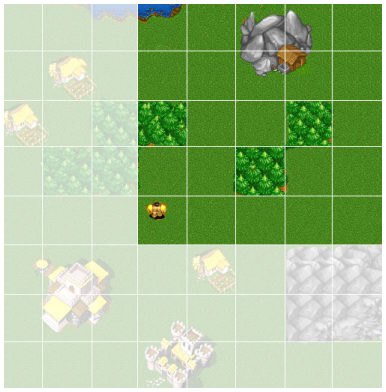

Fig. 9. Co-domain reduction

$$w = \begin{bmatrix} agent & structure & forest & rock & sand \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ ... & ... & ... & ... & ... \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Fig. 10. State representation of world after co-domain reduction

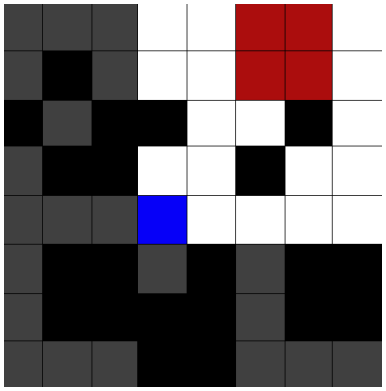

Fig. 11. Domain Aggregation

$$w = \begin{bmatrix} agent & obstacle \\ 0 & 1 \\ 1 & 0 \\ ... & ... \\ 0 & 0 \\ 0 & 1 \end{bmatrix}$$

Fig. 12. State representation of world after domain aggregation

in that they obstruct the agent from moving there. This is illustrated in Figure 13: our generalized hypothesis claims that the light-grey parts of the world are equivalent (i.e., trees and rocks combined), and similarly for the dark-grey parts of the world (i.e., grass and sand combined). The effect is (roughly) similar to the effect of an aggregation operation in Figure 11. However, it is possible that at some point our generalization that forests and rock are equivalent proves to be faulty. Say, the agent has learned to chop trees down so it can move through forest locations. In the case of generalization, we can simply remove or reformulate our hypothesis and return to the original world, whereas with abstraction the original information is lost and we can not turn back to the original world. In our example (see Figure 14), it is unclear whether an obstacle used to be part of a forest or rock. We threw away that information during our abstraction process. Therefore, we claim that generalization is more flexible and less conclusive than abstraction.

Fig. 13. Generalization

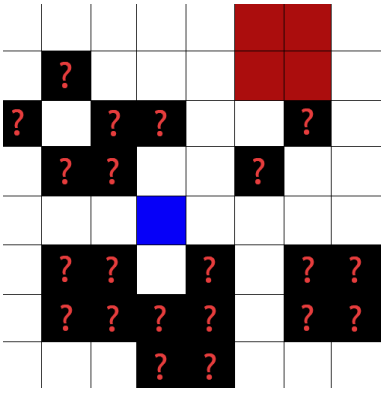

Fig. 14. Difference between abstraction and generalization: with an abstraction operation, information is lost.

5 Function Approximation

The previous section introduced many different generalization and abstraction operators. In this section, we discuss a commonly used approach, where information gathered by an agent is used to tune a mathematical function that represents the agent's gathered knowledge.

In tasks with small and discrete state spaces, the functions V, Q, and π can be represented in a table, such as discussed in the previous section. However, as the state space grows using a table becomes impractical (or impossible if the state space is continuous). In such situations, some sort of *function approximator* is necessary, which allow the agent to use data to estimate previously unobserved (s, a) pairs.

How to best choose which function approximator to use, or how to set its parameters, is currently an open question. Although some work in RL [11, 24, 25] has taken a more systematic approaches to *state abstractions* (also called *structural abstractions*), the majority of current research relies on humans to help bias a learning agent by carefully selecting a function approximator with parameters appropriate for a given task. In the remainder of this section we discuss three popular function approximators: Cerebellar Model Arithmetic Computers (CMACs), neural networks, and instance-based approximation.

The first two methods, CMACs and neural networks, may be considered both approximation and generalization operators. Rather than saving the data gathered in the world, the agent tunes its function approximator and discards data, losing some information (abstraction), but it is then able to calculate the value of the function for values that have not been experienced (generalization). Many methods for instance-based approximation also discard data, but some do not; while all instance-based function approximators are generalizers, not all are abstractors.

Cerebellar Model Arithmetic Computers CMACs [1] take arbitrary groups of continuous state variables and lay infinite, axis-parallel tilings over them (see Figure 15(a)). This allows discretization of continuous state space into tiles while maintaining the capability to generalize via multiple overlapping tilings. Increasing the tile widths allows better generalization; increasing the number of tilings allows more accurate representations of smaller details. The number of tiles and the width of the tilings are generally handcoded: this sets the center, c_i, of each tile and dictates which state values will activate which tiles. The function approximation is trained by changing how much each tile contributes to the output of the function approximator. Thus, the output from the CMAC is the computed sum:

$$f(x) = \sum_i w_i f_i(x) \tag{9}$$

but only tiles which are activated by the current state features contribute to the sum:

$$f_i(x) = \begin{cases} 1, \text{ if tile } i \text{ is activated} \\ 0, \text{ otherwise} \end{cases}$$

Weights in a CMAC are typically initialized to zero and are changed over time via learning.

Artificial Neural Networks The neural network function approximator similarly allows a learner to approximate the action-value function, given a set of continuous, real valued, state variables. Although neural networks have been shown to be difficult to train in certain situations on relatively simple RL problems [5, 30], they have had notable successes on some RL tasks [9, 46]. Each input to the neural network is set to the value of a state variable and each output corresponds to an action. Activations of the output nodes correspond to Q-values (see Figure 15(b) for a diagram).

When used to approximate an action-value function, neural networks often use non-recurrent feedforward networks. Each node in the input layer is given the value of a different state variable and each output node corresponds is the the calculated Q-value for a different action. The number of inputs and outputs are thus determined by the task's specification, but the number of hidden nodes is specified by the agent's designer. Note that by accepting multiple inputs the neural network can determine its output by considering multiple state variables in conjunction (as opposed to a CMAC consisting of a separate 1-dimensional tiling for each state variable). Nodes often have either sigmoid or linear transfer functions. Weights for connections in the network are typically initialized to random values near zero. The networks are often trained using backpropagation, where the error signal to modify weights is generated by the learning algorithm, as with the other function approximators.

Instance-based approximation CMACs and neural networks aim to represent a complex function with a relatively small set of parameters that can be changed over time. In contrast, instance-based approximation stores *instances* experienced by the agent (i.e., $\langle s, a, r, s' \rangle$) to predict the underlying structure of the environment. Specifically, this approximation method can be used by model-learning methods (c.f., [19, 20]), which learn

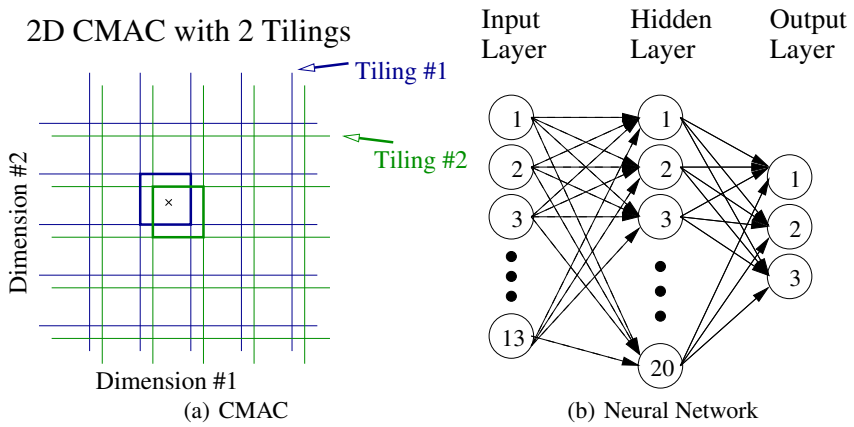

Fig. 15. CMAC's value, shown in (a), is computed by summing the weights, w_i, from multiple activated tiles (outlined above with thicker lines). State variables are used to determine which tile is activated in each of the different tilings. The diagram in (b) shows an artificial feedforward 13-20-3 network, suggesting how Q-values for three actions can be calculated from 13 state variables.

to approximate T and R by observing the agent's experience when interacting with an environment.

Consider the case where an agent is acting in a discrete environment with a small state space. The agent could record every instance that it experienced in a table. If the transition function were deterministic, as soon as the agent observed every possible (s, a) pair, it could calculate the optimal policy. If the transition function were instead stochastic, the agent would need to take multiple samples for every (s, a) pair. Given a sufficient number of samples, as determined by the variance in the resulting r and s', the agent could again directly calculate the optimal policy via dynamic programming.

When used to approximate T and R for tasks with continuous state spaces, using instances for function approximation becomes significantly more difficult. In a stochastic task the agent is unlikely to ever visit the same state twice, with the possible exception of a start state, and thus approximation is critical. Furthermore, since one can never gather "enough" samples for every (s, a) pair, such methods generally need to determine which instances are necessary to store so that the memory requirements are not unbounded. Creating efficient instance-based function approximators, and their associated learning algorithms, are topics of ongoing research in RL.

Now that the basic concepts of abstraction, and generalization have been introduced in the context of RL, the next section describes our own contribution in the field of abstraction and generalization in RL through action abstraction via hierarchical RL techniques. Later sections will then discuss work on generalization using relational reinforcement learning and transfer learning.

6 Hierarchical Reinforcement Learning

There exist many extensions to standard RL that make use of the abstraction and generalization operators mentioned in Section 4. One such method is hierarchical reinforcement learning (HRL), which essentially aggregates actions. HRL is an intuitive and promising approach to scale up RL to more complex problems. In HRL, a complex task is decomposed into a set of simpler subtasks that can be solved independently. Each subtask in the hierarchy is modeled as a single MDP and allows appropriate state, action and reward abstractions to augment learning compared to a flat representation of the problem. Additionally, learning in a hierarchical setting can facilitate generalization: knowledge learned in a subtask can be transferred to other subtasks. HRL relies on the theory of Semi-Markov decision processes (SMDPs) [40]. SMDPs differ from MDPs in that actions in SMDPs can last multiple time steps. Therefore, in SMDPs actions can either be primitive actions (taking exactly 1 time-step) or temporally extended actions. While the idea of applying HRL in complex domains such as computer games is appealing, with the exception of [2], there are few studies that examining this issue.

We adopted a HRL method similar to Hierarchical Semi-Markov Q-learning (HSMQ) described in [12]. HSMQ learns policies simultaneously for all non-primitive subtasks in the hierarchy, i.e., $Q(p, s, a)$ values are learned to denote the expected total reward of performing task p starting in state s, executing action a, and then following the optimal policy thereafter. Subtasks in HSMQ include termination predicates. These partition the state space S into a set of active states and terminal states. Subtasks can only be invoked in states in which they are active, and subtasks terminate when the state transitions from an active to a terminal state. We added to the HSMQ algorithm described in [12] a pseudo-reward function [12] for each subtask. The pseudo-rewards tell how desirable each of the terminal states are for this subtask. Algorithm 3 outlines our HSMQ-inspired algorithm. Q-values for primitive subtasks are updated with the one-step Q-learning update rule, while the Q-values for non-primitive subtasks are updated based on the reward $R(s, a)$, collected during execution of the subtask and a pseudo reward $\hat{R}$.

6.1 Reactive Navigation Task

We applied our HSMQ algorithm to the game of Wargus. Inspired by the resource gathering task, we created a world wherein a peasant has to learn to navigate to some location on the map while avoiding enemy contact (in the game of Wargus, peasants have no means for defending themselves against enemy soldiers). More precisely, our simplified game consists of a fully observable world that is 32×32 grid cells and includes two units: a peasant (the adaptive agent) and an enemy soldier. The adaptive agent has to move to a certain goal location. Once the agent reaches its goal, a new goal is set at random. The enemy soldier randomly patrols the map and will shoot at the peasant if it is in firing range. The scenario continues for a fixed time period or until the peasant is destroyed by the enemy soldier.

Relevant properties for our task are the locations of the peasant, soldier, and goal. All three objects can be positioned in any of the 1024 locations. A propositional format of the state space describes each state as a feature vector with attributes for each

Algorithm 3: Modified version of the HSMQ algorithm: The update rule for non-primitive subtasks (line 13) differs from the original implementation

1 **Function** HSMQ(state s,subtask p) returns float;
2 Let *Totalreward* $= 0$;
3 **while** *(p is not terminated)* **do**
4 Choose action $a = \Pi(s)$;
5 **if** a *is primitive* **then**
6 Execute a, observe one-step reward $R(s, a)$ and result state s';
7 **else if** a *is non-primitive subtask* **then**
8 $R(s, a) :=$ HSMQ(s, a) , which invokes subtask a and returns the total reward received while a executed
9 *Totalreward* $=$ *Totalreward* $+ R(s, a)$;
10 **if** a *is primitive* **then**
11 $Q(p, a, s) \rightarrow (1 - \alpha)Q(p, a, s) + \alpha \left[R(s, a) + \gamma \max_{a'} Q(p, a', s') \right]$;
12 **else if** a *is non-primitive subtask* **then**
13 $Q(p, a, s) \rightarrow (1 - \alpha)Q(p, a, s) + \alpha \left[R(s, a) + \hat{R} \right]$;
14 **end**
15 **return** *Totalreward*;

possible property of the environment, which amounts to 2^{30} different states. As such, a tabular representation of the value functions is too large to be feasible. Additionally, such encoding prevents any opportunity for generalization for the learning algorithm, e.g., when a policy is learned to move to a specific grid cell, the policy can not be reused to move to another. A deictic state representation identifies objects relative to the agent. This reduces state space complexity and facilitates generalization. This is a first step towards a fully relational representation, such as the one covered in Section 7. We will discuss the state features used for this task in Section 6.2.

The proposed task is complex for several reasons. First, the state space without any abstractions is enormous. Second, the game state is also modified by an enemy unit. (The enemy executes a random move on each timestep unless the peasant is in sight, in which case it moves toward the peasant.) Furthermore, each new task instance is generated randomly (i.e., random goal and enemy patrol behavior), so that the peasant has to learn a policy that generalizes over unseen task instances.

6.2 Solving the Reactive Navigation Task

We compare two different ways to solve the reactive navigation task, namely using flat RL and HRL. For a **flat representation** of our task, the deictic state representation can be defined as the Cartesian-product of the following four features: `Distance(enemy,s)`, `Distance(goal,s)`, `DirectionTo(enemy,s)`, and `DirectionTo (goal,s)`. The function `Distance` returns a number between 1 and 8 or a string indicating that the object is more than 8 steps away in state s, while `DirectionTo` returns the relative direction to a given object in state s. Using 8 possible values for the `DirectionTo` function, namely the eight compass directions, and 9 possible values for the `Distance`

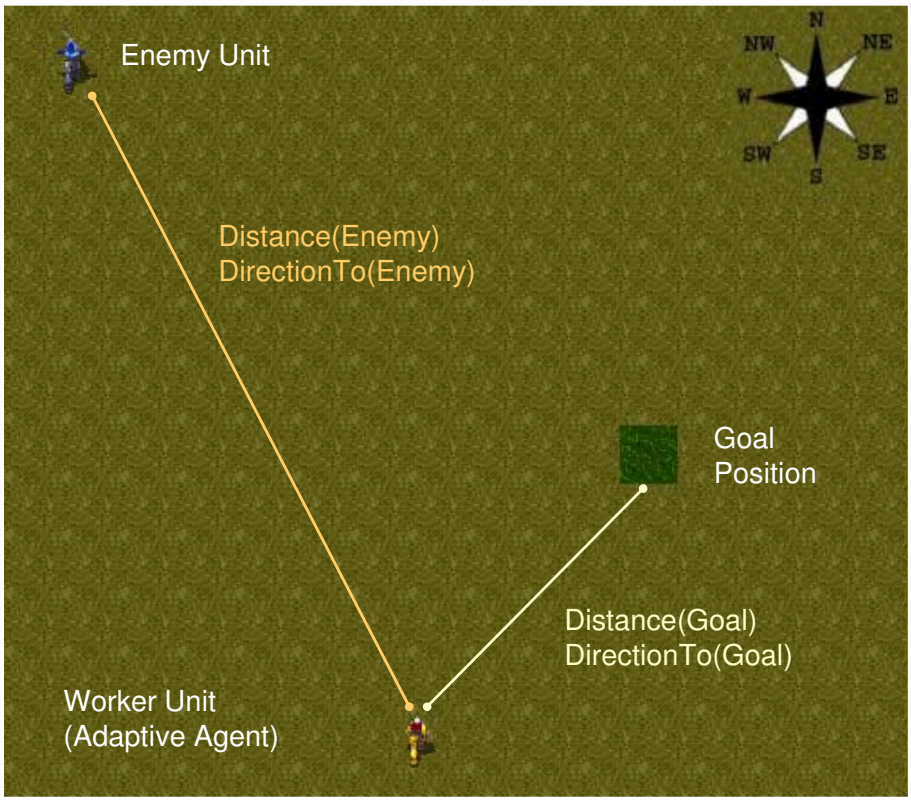

Fig. 16. This figure shows a screenshot of the reactive navigation task in the Wargus game. In this example, the peasant is situated at the bottom. Its task is to move to a goal position (the dark spot right to the center) and avoid the enemy soldier (situated in the upper left corner) that is randomly patrolling the map.

function, the total state space is drastically reduced from 2^{30} to only 5184 states. The size of the action space is 8, containing actions for moving in each of the eight compass directions. The scalar reward signal $R(s, a)$ in the flat representation should reflect the relative success of achieving the two concurrent sub-goals (i.e., moving towards the goal while avoiding the enemy). The environment returns a $+10$ reward whenever the agent is located on a goal location. In contrast, a reward of -10 is returned when the agent is being fired at by the enemy unit, which occurs when the agent is in firing range of the enemy (i.e., within 5 steps). Each primitive action always yields a reward of -1. An immediate concern is that both sub-goals are often in competition. We can certainly consider situations where different actions are optimal for the two sub-goals, although the agent can only take one action at a time. An apparent solution to handle these two concurrent sub-goals is applying a hierarchical representation, which we discuss next.

In the **hierarchical representation**, the original task is decomposed into two simpler subtasks that solve a single sub-goal independently (see Figure 17). The *to goal* subtask is responsible for navigation to goal locations. Its state space includes the `Distance(goal,s)` and `DirectionTo(goal,s)` features. The *from enemy* subtask is responsible for evading the enemy unit. Its state space includes the `Distance(enemy,s)` and `DirectionTo(enemy,s)` features. The action spaces for both subtasks include the primitive actions for moving in all compass directions. The two subtasks are hierarchically combined in a higher-level *navigate* task. The state space of this task is represented by the `InRange(goal,s)` and `InRange(enemy,s)` features, and its action space consists of the two subtasks that can be invoked as if they were primitive actions. `InRange` is a function that returns *true* if the distance to an object is 8 or less in state s, and *false* otherwise. Because these new features can be defined in terms of existing features, we are not introducing any additional domain knowledge compared to the flat representation. The *to goal* and *from enemy* subtasks terminate at each state change on the root level, e.g., when the enemy (or goal) transitions from in range to out of range and vice versa. We choose to set the pseudo-rewards for both subtasks to $+100$ whenever the agent completes a subtask and 0 otherwise. The *navigate* task never terminates, but the primitive subtasks always terminate after execution. The state

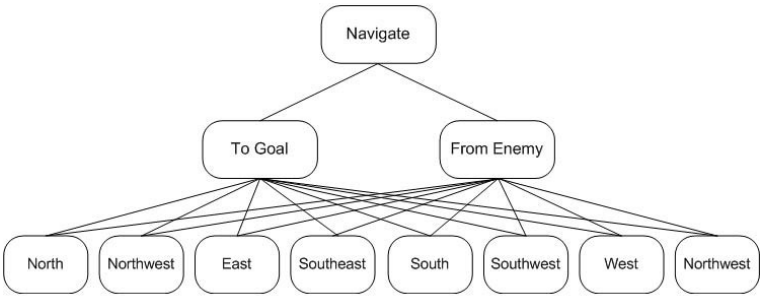

Fig. 17. Hierarchical decomposition of the reactive navigation task

spaces for the two subtasks are of size 72, and four for *navigate*. Therefore, the state space complexity in the hierarchical representation is approximately 35 times less than with the flat representation. Additionally, in the hierarchical setting we are able to split the reward signal, one for each subtask, so they do not interfere. The *to goal* subtask rewards solely moving to the goal (i.e., only process the $+10$ reward when reaching a goal location). Similarly, the *from enemy* subtask only rewards evading the enemy. Based on these two reward signals and the pseudo-rewards, the root *navigate* task is responsible for choosing the most appropriate subtask. For example, suppose that the peasant at a certain time decided to move to the goal and it took the agent 7 steps to reach it. The reward collected while the *to goal* subtask was active is -7 (reward of -1 for all primitive actions) and $+10$ (for reaching the goal location) resulting in a $+3$ total reward. Additionally, a pseudo-reward of $+100$ is received because *to goal* successfully terminated, resulting in a total reward of $+103$ that is propagated to the *navigate* subtask, that is used to update its Q-values. The Q-values for the *to goal* subtask are updated based on the immediate reward and estimated value of the successor state (see equation 8).

6.3 Experimental Results

We evaluated the performance of flat RL and HRL in the reactive navigation task. The step-size and discount-rate parameters were set to 0.2 and 0.7, respectively. These values were determined during initial experiments. We chose to use more exploration for the *to goal* and *from enemy* subtasks compared to *navigate*, since more Q-values must be learned. Therefore, we used Boltzmann action selection with a relatively high (but decaying) temperature for the *to goal* and *from enemy* subtasks and ϵ-greedy action selection at the top level, with ϵ set to 0.1 [39].

A "trial" (when Q-values are adapted) lasted for 30 episodes. An episode terminated when the adaptive agent was destroyed or until a fixed time limit was reached. During training, random instances of the task were generated, i.e., random initial starting locations for the units, random goals and random enemy patrol behavior. After each trial, we empirically tested the current policy on a test set consisting of five fixed task instances that were used throughout the entire experiment. These included fixed starting locations for all objects, fixed goals and fixed enemy patrol behavior. We measured the performance of the policy by counting the number of goals achieved by the adaptive agent (i.e., the number of times the agent successfully reached the goal location before it was destroyed or time ran out) by evaluating the greedy policy. We ended the experiment after 1500 training episodes (50 trials). The experiment was repeated five times and the averaged results are shown in Figure 18.

From this figure we can conclude that while both methods improve with experience, learning with the HRL representation outperforms learning with a flat representation. By using (more human-provided) abstractions, HRL represented the policy more compactly than the flat representation, resulting in faster learning. Furthermore, HRL is more suited to handling concurrent and competing subtasks due to the split reward signal. We expect that even after considerable learning with flat RL, HRL will still achieve a higher overall performance. This experiment shows that when goals have clearly con-

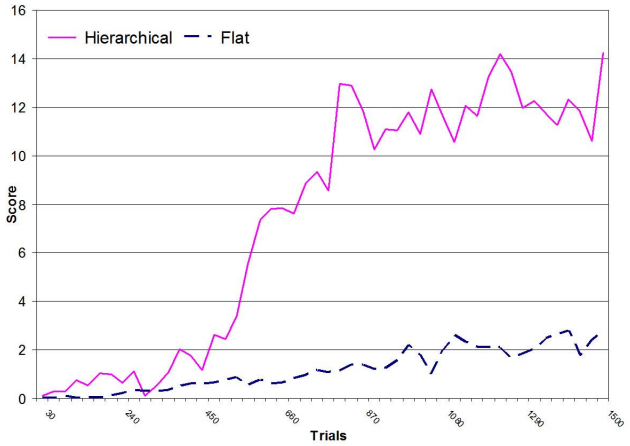

Fig. 18. This figure shows the average performance of Q-learning over 5 experiments in the reactive navigation task for both flat and HRL. The x-axis denotes the number of training trials and the y-axis denotes the average number of goals achieved by the agent for the tasks in the test set.

flicting rewards and the overall task can be logically divided into subtasks, HRL could be successfully applied.

7 Relational Reinforcement Learning

Relational reinforcement learning [14] (RRL) combines the RL setting with relational learning or inductive logic programming [26] (ILP) in order to represent states, actions, and policies using the structures and relations that identify them. These structural representations allow generalization over specific goals, states, and actions. Because relational reinforcement learning algorithms try to solve problems at an abstract level, the solutions will often carry to different instantiations of that abstract problem. For example, resulting policies learned by an RRL system often generalize over domains with varying number of existing objects.

A typical example is the blocks world. A number of blocks with different properties (size, color, etc.) are placed on each other or on the floor. It is assumed that an infinite number of blocks can be put on the floor and that all blocks are neatly stacked onto each other, e.g., a block can only be on one other block. The possible actions consist of moving one clear block (e.g., a block with no other block on top of it) onto another clear block, or onto the floor. It is impossible to represent such world states with a propositional representation without an exponential increase of the number of states. Consider as an example the right-most state in Figure 19. In First-Order Logic (FOL), this state can be represented, presuming this state is called s, by the conjunction

$$\{on(s,c,floor) \wedge clear(s,c) \wedge on(s,d,floor) \wedge on(s,b,d) \wedge on(s,a,b) \wedge \\ clear(s,a) \wedge on(s,e,floor) \wedge clear(s,e)\}.$$

s is reached by executing the move action (indicated by the arrow), noted as *move(r,s,a,b)*, in the previous state (on the left in Figure 19).

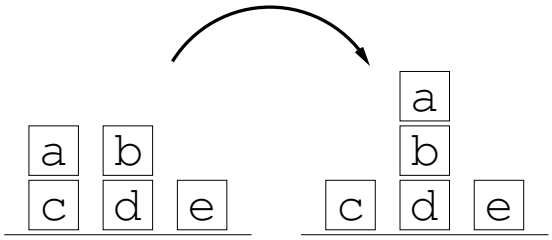

Fig. 19. The blocks world

One of the most important benefits of the relational learning approach is that one can generalize over states, actions, objects, but one is not forced to do so (one can abstract away selectively only these things that are less important). For instance, suppose that all blocks have a size property. One could then say that "there exists a small block which is on a large block" ($\exists B1, B2 : block(B1, small, \_), block(B2, large, \_), on(B1, B2)$). Objects $B1$ and $B2$ are free variables, and can be instantiated by any block in the environment. RRL can generalize over blocks and learn policies for a variable number of objects without necessarily suffering from the "curse of dimensionality" (where the size of the value function increases exponentially with the dimension of the state space).

Although it is a relatively new representation, several approaches to RRL have been proposed during the last few years. One of the first methods developed within RRL was relational Q-learning [14], described in Algorithm 4. Relational Q-learning behaves similarly to standard Q-learning, but is adapted to the RRL representation. In relational reinforcement learning, the representation contains structural or relational information about the environment. Relational Q-learning employs a relational regression algorithm to generalize over the policy space. Learning examples, stored as a tuple $(a, s, Q(s, a))$, are processed by an incremental relational regression algorithm to produce a relational value-function or policy as a result. So far, a number of different relational regression learners have been developed.[1]

In [28], we demonstrated how relational reinforcement learning and multi-agent systems techniques could be combined to plan well in tasks that are complex, multi-state, and dynamic. We used a relational representation of the state space in multi-agent reinforcement learning, as this has many benefits over the propositional one. For instance, it handled large state spaces, used a rich relational language, modeled other agents without a computational explosion (generalizing over agents' policies), and generalized over newly derived knowledge. We investigated the positive effects of relational reinforcement learning applied to the problem of agent communication in multi-agent systems. More precisely, we investigated the learning performance of RRL given some communication constraints. Our results confirm that RRL can be used to adequately deal with large state spaces by generalizing over states, actions and even agent policies.

[1] A thorough discussion and comparison can be found in [13].

Algorithm 4: The Relational Reinforcement Learning Algorithm

REQUIRE initialize $Q(s, a)$ and s_0 arbitrarily;
$e \leftarrow 0$;
repeat {for each episode}
  Examples $\leftarrow \emptyset$;
  $i \leftarrow 0$;
  repeat {for each step $\in$ episode}
    take a for s using policy $\pi(s)$ and observe r and s';
    $Q(a, s) \rightarrow (1 - \alpha)Q(a, s) + \alpha \left[r + \gamma \max_{a'} Q(a', s') \right]$;
    Examples $\leftarrow$ *Examples* $\cup \{a, s, Q(s, a)\}$;
    $i \leftarrow i + 1$;
  until s_i is terminal ;
  Update $\hat{Q}_e$ using *Examples* and a relational regression algorithm to produce $\hat{Q}_{e+1}$;
  $e \leftarrow e + 1$;
until done ;

8 Transfer Learning

This section of the chapter chapter focuses on transfer learning (TL) and its relationship to generalization and abstraction. The interested reader is referred elsewhere [44] for a more complete treatment of transfer in RL.

8.1 Transfer Learning Background

All transfer learning algorithms for reinforcement learning agents use one or more *source tasks* to better learn in a *target task*, relative to learning without the benefit of the source task(s). Transfer techniques assume varying degrees of autonomy and make many different assumptions. For instance, one way TL algorithms differ is in how they allow source and target tasks to differ. Consider the pairs of MDPs represented in Figure 20. The source and target tasks could differ in any portion of the MDP: the transition function, T, the reward function, R, what states exist, what actions the agent can perform, and/or how the agent represents the world (the state is represented here in terms of *state variables* $\langle x_1, x_2, \ldots, x_n \rangle$ in the source task MDP and $\langle x_1, x_2, \ldots, x_m \rangle$ in the target task MDP).

The use of transfer in humans has been studied for many years in the psychological literature [34, 47]. More related is sequential transfer between machine learning tasks (c.f., [48]), which allows the learning of higher performing classifiers with less data. For instance, one could imagine using a large training corpus from a newspaper to help learn a classifier such that when the classifier is presented with training examples from a magazine, it can learn with fewer examples than if the newspaper data had not been used. Another common approach is that of learning to perform multiple tasks simultaneously (c.f., [8]). The motivation in this case is that a classifier capable of performing multiple tasks in a single domain will be forced to capture more structure of the domain, performing better than if any one classifier was trained and tested in isolation.

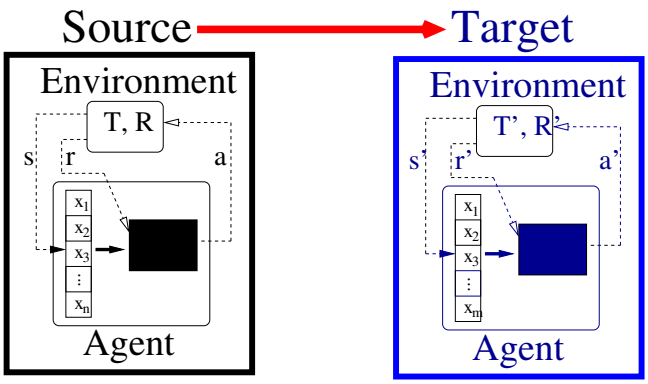

Fig. 20. Simple transfer schematic

8.2 Transfer as Generalization

Generalization may be thought of as the heart of machine learning: given a set of data, how should one perform on novel data? Such questions are prevalent in RL as well. For instance, in a continuous action space, an agent is unlikely to visit a single state more than once and it must therefore constantly generalize its previous data to predict how to act.

Transfer learning may be thought of as a different type of generalization, where the agent must generalize knowledge *across tasks*. This section examines two strategies common to transfer methods. The first strategy is to learn some low-level information in the source task and then use this information to better bias learning in the target task. The second strategy is to learn something that is true for the general domain, regardless of the particular task in question.

Example Domain: Keepaway One popular domain for demonstrating TL in RL tasks is that of *Keepaway* [37], a sub-task of the full 11 vs. 11 simulated soccer, which uses the RoboCup Soccer Server [27] to simulate sensor and actuator noise of physical robots. Typically, n teammates (the *keepers*) attempt to keep control of the ball within a small field area, while $n-1$ teammates (the *takers*) attempt to capture the ball or kick it out of bounds. The keepers typically learn while the takers follow a fixed policy. Only the keeper with the ball may select actions intelligently: all keepers without the ball always executes a `getopen` action, where they attempt to move to an area of the field to receive a pass.

Keepers learn to extend the average length of an episode over time by selecting between executing the `hold` ball action (attempting to maintain possession), or they may `pass` to a teammate. In 3 vs. 2 Keepaway, there are three keepers and two takers, and thus the keeper with the ball has 3 macro[2] actions (hold, pass to closest teammate, and

[2] Note that because the actions may last more than one timestep, Keepaway is technically an SMDP, rather than an MDP, but such distinction is not critical for the purposes of this chapter.

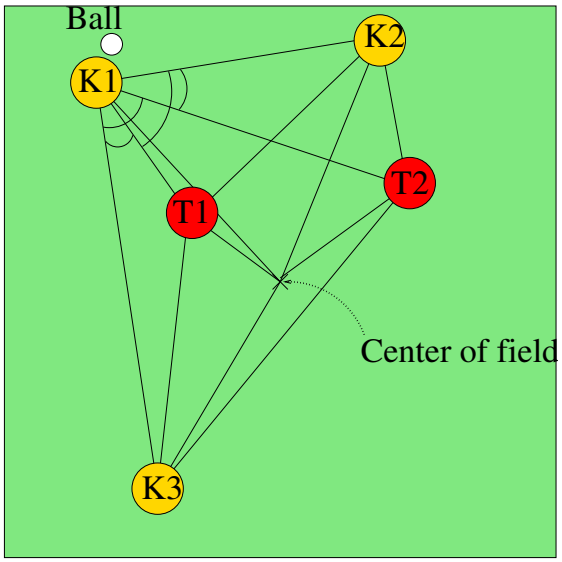

Fig. 21. 3 vs. 2 Keepaway uses multiple distances and angles to represent state, enumerated in Table 1.

pass to second closest teammate). The keeper with the ball represents its state with 13 (rotational invariant) state variables, shown in Figure 21. This is also a deictic representation, similar to the one discussed in Section 6.1, because players are labeled relative to their distance to the ball, rather than labeled by jersey number (or another fixed scheme).

The 4 vs. 3 Keepaway task, which adds an additional keeper and taker, is more difficult to learn. First, there are more actions (the keeper with the ball selects from 4 actions) and keepers have a more complex state representation (due to the extra players, the world is described by 19 state variables). Second, because there are more players on the field, passes must be more frequent. Each pass has a chance of being missed by the intended receiver, and thus each pass brings additional risk that the ball will be lost, ending the episode.

Given these two tasks, there are (at least) two possible goals for transfer. First, one may assume that a set of keepers have trained on the source task. Considering this source task training a sunk cost, the goal is to learn better/faster on the target task by using the source task information, compared to learning the target task while ignoring knowledge from the source task. A second, more difficult, goal is to explicitly account for source task training time. In other words, the second goal of transfer is to learn the source task, transfer some knowledge, and then target task better/faster than if the agents had directly trained only on the target task.

Transferring Low-Level Information: Ignoring novel structure When considering transfer from 3 vs. 2 to 4 vs. 3, one approach would be to learn a Q-value function in the source task and then copy it over into the target task, ignoring the novel state

Description 3 vs. 2 and 4 vs. 3 State Variables

3 vs. 2 state variable	4 vs. 3 state variable
$dist(K_1, C)$	$dist(K_1, C)$
$dist(K_1, K_2)$ $dist(K_1, K_3)$	$dist(K_1, K_2)$ $dist(K_1, K_3)$ $\mathbf{dist(K_1, K_4)}$
$dist(K_1, T_1)$ $dist(K_1, T_2)$	$dist(K_1, T_1)$ $dist(K_1, T_2)$ $\mathbf{dist(K_1, T_3)}$
$dist(K_2, C)$ $dist(K_3, C)$	$dist(K_2, C)$ $dist(K_3, C)$ $\mathbf{dist(K_4, C)}$
$dist(T_1, C)$ $dist(T_2, C)$	$dist(T_1, C)$ $dist(T_2, C)$ $\mathbf{dist(T_3, C)}$
Min($dist(K_2, T_1)$, $dist(K_2, T_2)$) Min($dist(K_3, T_1)$, $dist(K_3, T_2)$)	Min($dist(K_2, T_1)$, $dist(K_2, T_2)$, $\mathbf{dist(K_2, T_3)}$) Min($dist(K_3, T_1)$, $dist(K_3, T_2)$, $\mathbf{dist(K_3, T_3)}$) Min($\mathbf{dist(K_4, T_1)}$, $\mathbf{dist(K_4, T_2)}$, $\mathbf{dist(K_4, T_3)}$)
Min($ang(K_2, K_1, T_1)$, $ang(K_2, K_1, T_2)$) Min($ang(K_3, K_1, T_1)$, $ang(K_3, K_1, T_2)$)	Min($ang(K_2, K_1, T_1)$, $ang(K_2, K_1, T_2)$, $\mathbf{ang(K_2, K_1, T_3)}$) Min($ang(K_3, K_1, T_1)$, $ang(K_3, K_1, T_2)$, $\mathbf{ang(K_3, K_1, T_3)}$) Min($\mathbf{ang(K_4, K_1, T_1)}$, $\mathbf{ang(K_4, K_1, T_2)}$, $\mathbf{ang(K_4, K_1, T_3)}$)

Table 1. This table lists the 13 state variables for 3 vs. 2 Keepaway and the 19 state variables for 4 vs. 3 Keepaway. The distance between a and b is denoted as $dist(a, b)$; the angle made by a, b, and c, where b is the vertex, is denoted by $ang(a, b, c)$; and values not present in 3 vs. 2 are in bold. Relevant points are the center of the field C, keepers K_1-K_4, and takers T_1-T_3, where players are ordered by increasing distance from the ball.

variables and actions. This is similar to the idea of domain hiding, discussed earlier in Section 4.2. For instance, the `hold` action in 4 vs. 3 can be considered the same as the `hold` action in 3 vs. 2. Likewise, the 4 vs. 3 actions `pass` to closest teammate and `pass` to second closest teammate may be considered the same as the 3 vs. 2 actions `pass` to closest teammate and `pass` to second closest teammate. The "novel" 4 vs. 3 action, `pass` to third closest teammate, is ignored. Table 1 shows the state variables from the 3 vs. 2 and 4 vs. 3 tasks, where the novel state variables in the 4 vs. 3 task are in bold.

This is precisely the approach used in *Q-value Reuse* [45]. Results (reproduced in Table 2) show that the Q-values saved after training in the 3 vs. 2 task can be successfully used directly in the 4 vs. 3 task by ignoring the novel state variables and actions. Specifically, the source task action-value function is used as an initial bias in the target task, which is then refined over time with SARSA learning (e.g., Q-values for the novel 4 vs. 3 action are learned, and existing Q-values are refined). Column 2 of the table shows that the source task knowledge can be successfully reused, and column 3 shows that the total training can be successfully reduced via transfer.

Lazaric et al. [23] also focuses on transferring very low-level information by using source task *instances* in a target task. After learning one or more source tasks, some experience is gathered in the target task, which may have a different state space or transition function, but the state variables and actions must remain unchanged. Saved instances (that is, observed $\langle s, a, r, s' \rangle$ tuples) are compared to recorded instances in the target task. Source task instances that are very similar, as judged by their distance and alignment with target task data, are transferred. A batch learning algorithm (*Fitted Q-iteration* [15], which uses instance-based function approximation, then uses both source instances and target instances to achieve a higher total reward (relative to learning without transfer).

Region transfer, introduced in the same chapter [23], calculates the similarity between the target task and different source tasks per sample, rather than per task. Thus, if source tasks have different regions of the state space which are more similar to the target, only those most similar regions can be transferred. In this way, different regions of the target task may reuse data from different source tasks, and regions of the target task that are completely novel will use no source task data. Although this work did not use Keepaway, one possible example would be to have source tasks with different coefficients of friction: in the target task, grass conditions in different sections of the field would dictate which source tasks(s) are most similar and where data should be transferred from.

Transferring Low-Level Information: Mapping novel structure *Inter-task mappings* [45] allow a TL algorithm to explicitly state the relationship between different state variables and actions in the two tasks. For instance, the novel 4 vs. 3 action, `pass` to third closest teammate, may be mapped to the 3 vs. 2 action `pass` to second closest teammate. When such an inter-task mapping is provided (and is correct), transfer may be even more effective than if the novelty is ignored. Such an approach generalizes the source task knowledge to the target task. By generalizing over different target task state variables and actions, the TL method can initialize all target task Q-values to values learned in the source task, biasing learning and resulting in significantly faster learning. The Value Function Transfer method in Table 2 uses inter-task mappings (columns four and five), and outperforms Q-value Reuse, which had ignored novel state variables and actions.

While inter-task mappings are a convenient way to fully specify relationships between MDPs, it is possible that not enough information is known to design a full mapping. For instance, an agent may know that a pair of state variables describe "distance to teammate" and "distance from teammate to marker," but the agent is not told *which* teammate the state variables describe. *Homomorphisms* [31] are a different abstraction that can define transformations between MDPs based on transition and reward dynamics, similar in spirit to inter-task mappings, and have been used successfully for transfer [35]. However, discovering homomorphisms is NP-hard [32]. Work by Soni and Singh [35] supply an agent with a series of possible state transformations (i.e., potential homomorphisms) and an inter-task mapping for all actions. One transformation, X, exists for every possible mapping between target task state variables to source task state variables. The agent learns in the source task as normal. Then the agent must learn the

Transfer Results between 3 vs. 2 and 4 vs. 3 Keepaway

# of 3 vs. 2 Episodes	Q-value Reuse Avg. 4 vs. 3 Time (sim. hours)	Q-value Reuse Avg. Total Time (sim. hours)	Value Function Transfer Avg. 4 vs. 3 Time (sim. hours)	Value Function Transfer Avg. Total Time (sim. hours)
0	30.84	30.84	30.84	30.84
10	28.18	28.21	24.99	25.02
50	28.0	28.13	19.51	19.63
100	26.8	27.06	17.71	17.96
250	24.02	24.69	16.98	**17.65**
500	22.94	24.39	17.74	19.18
1,000	22.21	**24.05**	16.95	19.70
3,000	**17.82**	27.39	**9.12**	18.79

Table 2. Results in columns two and three (reproduced from [45]) show learning 3 vs. 2 for different numbers of episodes and then using the learned 3 vs. 2 CMAC directly while learning 4 vs. 3. Minimum learning times for reaching a preset 4 vs. 3 performance threshold are bold. All times reported are "simulator hours," the number of playing hours simulated, as opposed to wall-clock time. The top row of results shows the time required to learn in the 4 vs. 3 task without transfer. As source task training time increases, the required target task training time decreases. The total training time is minimized with a moderate amount of source task training. The results of using Value Function Transfer (with inter-task mappings) are shown in columns four and five. Q-value Reuse provides a statistically significant benefit relative to no transfer, and Value Function Transfer yields an even higher improvement.

correct transformation: in each target task state s, the agent must choose to randomly explore the target task actions, or choose to take the action suggested by the learned source task policy using one of the existing transformations, X. Q-learning allows the agent to select the best state variable mapping, defined as the one which allows the player to accrue the most reward, as well as learn the action-values for the target task. Later work by Sorg and Singh [36], extend this idea to that of learning "soft" homomorphisms. Rather than a strict surjection, these mappings assign probabilities that a state in a source task is the same as a state in the target task. This added flexibility allows the authors to provide bounds on their algorithm's performance, as well as show that such mappings are indeed learnable.

The MASTER algorithm [42] transfers instances similar to Lazaric [23], except that novel state variables and actions may be explicitly accounted for by using inter-task mappings. MASTER uses an exhaustive search to generate all possible inter-task mappings, and then selects the one that best describes the relationship between the source and target task, learning the best inter-task mapping. Then, data from the source task is mapped to the target task, and learning can continue to refine the source task data. Although this method currently does not scale to Keepaway (due to its reliance on model-learning methods [19] for continuous state variables), but is similar in spirit to the above two methods.

Transferring High-Level Information This section discusses four methods which transfer higher-level information than the previously discussed transfer methods. One

example is an *option*, where a set of actions are composed into a single high-level macro-action that the agent may choose to execute. Thus far, researchers have not quantified "high-level" and "low-level" information well, nor have they made convincing arguments to support the claim that high-level information is likely to generalize to different tasks within a similar domain. However, this claim makes intuitive sense and is an interesting open question.

Section 7 introduced Relational RL (RRL). Recall that the propositional representation allows state to be discussed in terms of objects and their properties. Actions in the RRL framework typically have pre- and post-conditions over objects. When the state changes such that there are more or fewer objects, learned Q-values for actions may be very similar, as the object on which the action acts has not changed. One example of such transfer is by Croonenbourghs et al. [10], where they first learn a source task policy with RRL. This source task policy then generates state-action pairs, which are in turn used to build a relational decision tree. This tree predicts which action would be executed by the policy in a given state. As a final step, the trees produce *relational options*. These options are directly used in the target task with the assumption that the tasks are similar enough that no translation of the relational options is necessary.

Konidaris and Barto [22] also consider transferring options, but do so in a different framework. Instead of an RRL approach, they divide problems into *agent-space* and *problem-space* representations [21]. Agent-space defines an agent's capabilities that remains fixed across all problems (e.g., it represents a robot's physical sensors and actuators); agent-space may be considered a type of domain hiding. The problem-space may change between tasks (e.g., there may be different room configurations in a navigation task). By assuming "pre-specified salient events," such as when a light turns on or an agent unlocks a door, agents may learn options to achieve these events. Options succeed in improving learning in a single task by making it easier for agents to reach such salient events (which are assumed to be relevant for the task being performed). Additionally, the agent may train on a series of tasks, learning options in both agent- and problem-space, and reusing them in subsequent tasks. The authors suggest that agent-space options will likely be more portable than problem-space options, and it is likely that problem-space options will only be useful when the source and target tasks are very similar.

Torrey et al. [49] also consider transferring option-like knowledge. Their method involves learning a *strategy*, represented as a finite-state machine, which can be applied to a target task with different state variables and actions. Strategies are learned in the source task and then remapped to the target task with inter-task mappings. These transferred strategies are then demonstrated to the target task learner. Instead of exploring randomly, the agent is forced to execute any applicable strategy for the first 100 episodes, learning to estimate the value of executing different strategies. After this demonstration phase, agents may then select from all of the MDP's actions. Experiments show that learning first on 2-on-1 BreakAway (similar to 2 vs. 1 Keepaway, but where the "keepers" are trying to score a goal on the "takers"), improves learning in both 3-on-2 Breakaway and 4-on-3 BreakAway.

8.3 Abstraction in Transfer

The majority of the TL methods in the above section focused on the first goal of transfer: showing that source task knowledge can improve target task learning, when the source task is treated as a sunk cost. (An exception is Taylor et al. [45], where they demonstrate that learning a pair or sequence of tasks can be faster than directly learning the target task.) In this section, we discuss abstraction in the context of transfer. Specifically, if the source task is an abstract version of the target task, it may be substantially faster to learn than the target task, but still provide the target task learner with a significant advantage.

Taylor and Stone [43] introduced the notion of *inter-domain* transfer in the context of *Rule Transfer* (similar in spirit to Torrey et al.'s method [49]). The chapter showed two instances where transfer between very different domains was successful. In both cases, the source tasks are discrete and fully observable, and one is deterministic. The target task was 3 vs. 2 Keepaway, which has continuous state variables, is partially observable, and is stochastic (due to sensor and actuator noise).

One of the source tasks is shown in Figure 22. In this task, the player begins at one end of the 25×25 board and the opponent begins at the opposite end of the board. The goal of the player is to reach the opposite end of the board without being touched by the opponent (either condition will end the episode). The player's state is represented by three state variables and it has three actions available. The player may move North (forward) or perform a knight's jump: North + East + East, or North + West + West. Although this task is very different from Keepaway, there are some important similarities. For instance, when the distance between the keeper with the ball and the closest taker is small, the player is likely to lose the ball (and thus end the episode). In Knight's Joust, when the distance between the player and the opponent is small, the episode is also likely to end.

Most significant with respect to transfer is the fact that an agent sees an average of only 600 distinct states over a 50,000 episode learning trial of Knight's Joust. This makes learning a good source task policy relatively easy: learning in the Knight's Joust abstraction takes on the order of a minute, whereas learning in 3 vs. 2 Keepaway takes hours of wall clock time. Using an abstract source task that allows very fast learning makes the task of reducing the total training time must easier: if the goal is to learn 3 vs. 2 Keepaway, it makes sense to spend a minute or two learning an abstract source task because it can save hours of learning in the target task (relative to directly learning 3 vs. 2 Keepaway).

While this result is encouraging, it can be regarded as a proof of concept: both source tasks used by Taylor and Stone [43] were carefully designed to be useful for transfer into the Keepaway task. The idea of constructing sequences of tasks which are fast to learn is very appealing [4, 41], but there are currently no concrete guidelines. This is due, in part, to the absence of a general method for deciding when transfer from a given source task will help learn a target task.

For instance, consider an agent that first learns the game of "Giveaway," where the goal is to loose the ball as fast as possible. If the agent then transfers this knowledge to Keepaway, it will perform worse than if it had ignored its previous knowledge [45]. While this may appear "obvious" to a human, such differences may be opaque to an

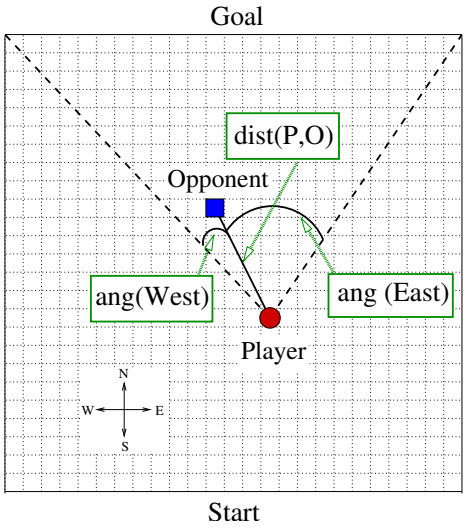

Fig. 22. Knight's Joust: The player attempts to reach the goal end of a a 25×25 grid-world while the opponent attempts to touch the player.

agent and its learning algorithm. In the above example, the source task and target task differ only in reward function: the transition model, state variables, actions, etc. are all identical. Similarly, if an agent learns a specific path out of a maze in a source task and then uses this policy to navigate a target task maze, a malicious designer of the target task may make the source task policy perform arbitrarily poorly. Protecting against such negative transfer is an important open question.

9 Conclusions

In this survey we investigated generalization and abstraction in Reinforcement Learning. We provided the fundamentals of RL, introduced definitions of generalization and abstraction, and elaborated on the most common techniques to achieve both. These techniques include function approximation, hierarchical reinforcement learning, relational reinforcement learning and transfer learning. With novel and existing examples from the literature, we illustrated these techniques and provided many references to the literature. Our hope is that this chapter has provided a solid introduction and structure to the concepts of abstraction and generalization in RL, encouraging additional work in this exciting field.

Acknowledgements

The authors would like to thank Scott Alfeld and Jason Tsai for useful comments and suggestions. Marc Ponsen is sponsored by the Interactive Collaborative Information Systems (ICIS) project, supported by the Dutch Ministry of Economic Affairs, grant nr: BSIK03024.

References

1. J. S. Albus. *Brains, Behavior, and Robotics*. Byte Books, Peterborough, NH, 1981.
2. A. Barto and S. Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems: Theory and Application*, 13(4):341–379, 2003.
3. R. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, New Jersey, 1957.
4. Y. Bengio, J. L., R. Collobert, and J. Weston. Curriculum learning. In *Proceedings of the Twenty-Sixth International Conference on Machine Learning*, June 2009.
5. J. A. Boyan and A. W. Moore. Generalization in reinforcement learning: Safely approximating the value function. In G. Tesauro, D. S. Touretzky, and T. K. Leen, editors, *Advances in Neural Information Processing Systems 7*, pages 369–376, Cambridge, MA, 1995. The MIT Press.
6. R. I. Brafman and M. Tennenholtz. R-max - a general polynomial time algorithm for near-optimal reinforcement learning. *Journal of Machine Learning Research*, 3:213–231, 2003.
7. R.A. Brooks. Intelligence without representation. *Artificial Intelligence*, (47):139–159, 1991.
8. R. Caruana. Multitask learning. *Machine Learning*, 28:41–75, 1997.
9. R. H. Crites and A. G. Barto. Improving elevator performance using reinforcement learning. In D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, editors, *Advances in Neural Information Processing Systems 8*, pages 1017–1023, Cambridge, MA, 1996. MIT Press.
10. T. Croonenborghs, K. Driessens, and M. Bruynooghe. Learning relational options for inductive transfer in relational reinforcement learning. In *Proceedings of the Seventeenth Conference on Inductive Logic Programming*, 2007.
11. T. Dean and R. Givan. Model minimization in Markov decision processes. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 106–111, 1997.
12. T. Dietterich. An overview of MAXQ hierarchical reinforcement learning. In *Proceedings of the Symposium on Abstraction, Reformulation and Approximation (SARA-00), Lecture Notes in Computer Science*, volume 1864, pages 26–44, 2000.
13. K. Driessens. *Relational Reinforcement Learning*. PhD thesis, DEPTCW, 2004. http://www.cs.kuleuven.be/publicaties/doctoraten/cw/CW2004_05.abs.html.
14. S. Džeroski, L. De Raedt, and K. Driessens. Relational reinforcement learning. *Machine Learning*, 43:7–52, 2001.
15. D. Ernst, P. Geurts, and L. Wehenkel. Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research*, 6:503–556, 2005.
16. F. Giunchiglia and T. Walsh. A theory of abstraction. *Artificial Intelligence*, 57(2-3):323–389, 1992.
17. R.C. Holte and B.Y. Chouiery. Abstraction and reformulation in ai. *Philosophical transactions of the Royal Society of London*, 358(1435:1):197–204, 2003.
18. R.A. Howard. *Dynamic Programming and Markov Processes*. The MIT press, Cambridge, Massachusetts, 1960.
19. N. K. Jong and P. Stone. Model-based exploration in continuous state spaces. In *The Seventh Symposium on Abstraction, Reformulation, and Approximation*, July 2007.
20. M. Kearns and S. Singh. Near-optimal reinforcement learning in polynomial time. In *Proc. 15th International Conf. on Machine Learning*, pages 260–268. Morgan Kaufmann, San Francisco, CA, 1998.
21. G. Konidaris and A. Barto. Autonomous shaping: Knowledge transfer in reinforcement learning. In *Proceedings of the 23rd International Conference on Machine Learning*, pages 489–496, 2006.

22. G. Konidaris and A. G. Barto. Building portable options: Skill transfer in reinforcement learning. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 895–900, 2007.
23. A. Lazaric, M. Restelli, and A. Bonarini. Transfer of samples in batch reinforcement learning. In *Proceedings of the 25th Annual ICML*, pages 544–551, 2008.
24. L. Li, T. J. Walsh, and M. L. Littman. Towards a unified theory of state abstraction for MDPs. In *Proceedings of the Ninth International Symposium on Artificial Intelligence and Mathematics*, pages 531–539, 2006.
25. S. Mahadevan and M. Maggioni. Proto-value functions: A Laplacian framework for learning representation and control in Markov decision processes. *Journal of Machine Learning Research*, 8:2169–2231, 2007.
26. S. Muggleton and L. De Raedt. Inductive logic programming : Theory and methods. *Journal of Logic Programming*, 19,20:629–679, 1994.
27. I. Noda, H. Matsubara, K. Hiraki, and I. Frank. Soccer server: A tool for research on multi-agent systems. *Applied Artificial Intelligence*, 12:233–250, 1998.
28. M. Ponsen, T. Croonenborghs, J. Ramon K. Tuyls, K. Driessens, J. van den Herik, and E. Postma. Learning with whom to communicate using relational reinforcement learning. In *International Conference on Autonomous Agents and Multi Agent Systems (AAMAS)*, 2009.
29. M. Puterman. *Markov decision processes: Discrete stochastic dynamic programming*. John Wiley and Sons, New York, 1994.
30. L. D. Pyeatt and A. E. Howe. Decision tree function approximation in reinforcement learning. In *Proceedings of the Third International Symposium on Adaptive Systems: Evolutionary Computation & Probabilistic Graphical Models*, pages 70–77, 2001.
31. B. Ravindran and A. Barto. Model minimization in hierarchical reinforcement learning. In *Proceedings of the Fifth Symposium on Abstraction, Reformulation and Approximation*, 2002.
32. B. Ravindran and A. Barto. An algebraic approach to abstraction in reinforcement learning. In *Twelfth Yale Workshop on Adaptive and Learning Systems*, pages 109–114, 2003.
33. G. A. Rummery and M. Niranjan. On-line Q-learning using connectionist systems. Technical report, Cambridge University Engineering Department, 1994.
34. B. F. Skinner. *Science and Human Behavior*. Colliler-Macmillian, 1953.
35. V. Soni and S. Singh. Using homomorphisms to transfer options across continuous reinforcement learning domains. In *Proceedings of the Twenty First National Conference on Artificial Intelligence*, July 2006.
36. J. Sorg and S. Singh. Transfer via soft homomorphisms. In *Proceedings of the Eighth International Conference on Autonomous Agents and Multiagent Systems*, pages 741–748, May 2009.
37. P. Stone, G. Kuhlmann, M. E. Taylor, and Y. Liu. Keepaway soccer: From machine learning testbed to benchmark. In Itsuki Noda, Adam Jacoff, Ansgar Bredenfeld, and Yasutake Takahashi, editors, *RoboCup-2005: Robot Soccer World Cup IX*, volume 4020, pages 93–105. Springer Verlag, Berlin, 2006.
38. R. Sutton. Dyna, an integrated architecture for learning, planning, and reacting. *SIGART Bulletin*, 2:160–163, 1991.
39. R. Sutton and A. Barto. *Reinforcement Learning: an introduction*. MIT Press, Cambridge, MA, USA, 1998.
40. R. Sutton, D. Precup, and S. Singh. Between mdps and semi-mdps: a framework for temporal abstraction in reinforcement learningx,. *Artificial Intelligence*, 112:181–211, 1999.
41. M. E. Taylor. Assisting transfer-enabled machine learning algorithms: Leveraging human knowledge for curriculum design. In *The AAAI 2009 Spring Symposium on Agents that Learn from Human Teachers*, March 2009.

42. M. E. Taylor, Nicholas K. Jong, and P. Stone. Transferring instances for model-based reinforcement learning. In *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD)*, pages 488–505, September 2008.
43. M. E. Taylor and P. Stone. Cross-domain transfer for reinforcement learning. In *Proceedings of the Twenty-Fourth International Conference on Machine Learning*, June 2007.
44. M. E. Taylor and P. Stone. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(1):1633–1685, 2009.
45. M. E. Taylor, P. Stone, and Y. Liu. Transfer learning via inter-task mappings for temporal difference learning. *Journal of Machine Learning Research*, 8(1):2125–2167, 2007.
46. G. Tesauro. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6(2):215–219, 1994.
47. E. Thorndike and R. Woodworth. The influence of improvement in one mental function upon the efficiency of other functions. *Psychological Review*, 8:247–261, 1901.
48. S. Thrun. Is learning the n-th thing any easier than learning the first? In *Advances in Neural Information Processing Systems*, volume 8, pages 640–646, 1996.
49. L. Torrey, J. W. Shavlik, T. Walker, and R. Maclin. Relational macros for transfer in reinforcement learning. In *Proceedings of the Seventeenth Conference on Inductive Logic Programming*, 2007.
50. C. Watkins. *Learning with Delayed Rewards*. PhD thesis, Cambridge University, 1989.
51. G. Weiss. A multiagent variant of dyna-q. In *Proceedings of the 4th International Conference on Multi-Agent Systems (ICMAS-2000)*, pages 461–462, 2000.
52. M. Wiering. *Explorations in Efficient Reinforcement Learning*. PhD thesis, Universiteit van Amsterdam, 1999.
53. J.D. Zucker. A grounded theory of abstraction in artificial intelligence. *Philosophical transactions of the Royal Society of London*, 358(1435:1):293–309, 2003.