Incorporating Advice into Agents that Learn from Reinforcements

Richard Maclin **Jude W. Shavlik**

Computer Sciences Department
1210 West Dayton Street
Madison, WI 53706

Phone: 608-263-0475
Email: {maclin,shavlik}@cs.wisc.edu

Abstract

Learning from reinforcements is a promising approach for creating intelligent agents. However, reinforcement learning usually requires a large number of training episodes. We present a system called RATLE that addresses this shortcoming by allowing a connectionist Q-learner to accept advice given, at any time and in a natural manner, by an external observer. In RATLE, the advice-giver watches the learner and occasionally makes suggestions, expressed as instructions in a simple programming language. Based on techniques from knowledge-based neural networks, RATLE inserts these programs directly into the agent's utility function. Subsequent reinforcement learning further integrates and refines the advice. We present empirical evidence that shows our approach leads to statistically-significant gains in expected reward. Importantly, the advice improves the expected reward regardless of the stage of training at which it is given.

A shorter version of this paper appears in the *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, AAAI Press.

Incorporating Advice into Agents that Learn from Reinforcements

1 Introduction

A successful and increasingly popular method for creating intelligent agents is to have them learn from reinforcements (Barto et al., 1990; Lin, 1992; Mahadevan & Connell, 1992; Tesauro, 1992). However, these approaches suffer from their need for large numbers of training episodes. While several approaches for speeding up reinforcement learning have been proposed (they are reviewed later), a largely unexplored approach is to design a learner that can also accept advice from an external observer. We present and evaluate an approach, called RATLE (**R**einforcement and **A**dvice-**T**aking **L**earning **E**nvironment), for creating advice-taking learners.

To illustrate the general idea of advice-taking, imagine that you are watching an agent learning to play some video game. Assume you notice that frequently the agent loses because it goes into a "box canyon" in search of food and then gets trapped by its opponents. One would like to give the learner advice such as "do not go into box canyons when opponents are in sight." Importantly, the external observer should be able to provide its advice in some quasi-natural language, using terms about the specific task domain. In addition, the advice-giver should be oblivious to the details of whichever internal representation and learning algorithm the agent is using.

Recognition of the value of advice-taking has a long history in AI. The general idea of an agent accepting advice was first proposed about 35 years ago by McCarthy (1958). Over a decade ago, Mostow (1982) developed a program that accepted and "operationalized" high-level advice about how to better play the card game Hearts. More recently Gordon and Subramanian (1994) created a system that deductively compiles high-level advice into concrete actions, which are then refined using genetic algorithms. However, the problem of making use of general advice has been largely neglected.

In the next two sections, we present a framework for using advice with reinforcement learners, and our instantiation of this framework. The subsequent section presents experiments that investigate the value of our approach. Finally, we list possible extensions to RATLE, further describe its relation to other research, and present some conclusions that can be drawn from our research.

2 A General Framework for Advice-Taking

In this paper we focus on a method that allows an agent employing reinforcement learning (RL) to make use of advice, though our technique applies to many advice-taking tasks. We begin by outlining a framework for advice-taking developed by Hayes-Roth, Klahr, and Mostow (1981)[1], and discuss how our system fits into their framework. In the following section we present specific details of our implemented system, RATLE.

Step 1. Request the advice.

To begin the process of advice-taking, a decision must be made that advice is needed. Often, approaches to advice-taking focus on having the learner ask for advice when it is needs help. Rather than having the learner request advice, RATLE allows the external observer to provide advice whenever the observer feels it is appropriate. There are two reasons for this: (i) it places less of a burden on the observer; and (ii) it is an open question how to create the best mechanism for having an agent recognize (and express) its need for advice. In the specific area of providing advice to RL agents, other work (Clouse & Utgoff, 1992; Whitehead, 1991) has focused on having the observer assess the actions chosen by the agent. However, this can lead to a large amount of interaction and requires that the learner induce the generality of the advice.

Step 2. Convert the advice to an internal representation.

Once the observer has created a piece of advice, the agent must try to understand the advice. Due to the complexities of natural language processing, we require that the external observer express its advice using a simple programming language and a list of acceptable task-specific terms. RATLE then parses the advice, using traditional methods from programming-language compilers.

Step 3. Convert the advice into a usable form.

After the advice has been parsed, RATLE transforms the general advice into terms that can be directly understood by the agent. Using techniques from *knowledge compilation* (Dietterich, 1991), a learner can convert ("operationalize") high-level advice into a (usually larger) collection of directly interpretable statements (see Gordon and Subramanian, 1994). In many task domains, the advice-giver may wish to use natural, but imprecise, terms such as "near" and "many." A compiler for such terms will be needed for each general environment since the terms needed to describe an

[1]See also pg. 345–349 of Cohen and Feigenbaum (1982).

problem will be problem-dependent. In RATLE, we make use of a method similar to Berenji and Khedkar's (1992) for compiling fuzzy-logic terms into neural networks.

Step 4. Integrate the reformulated advice into the agent's current knowledge base.

In this work our agent employs a connectionist approach to RL. To incorporate the observer's advice, the agent's neural network must be updated. RATLE uses ideas from *knowledge-based neural networks* (Fu, 1989; Omlin & Giles, 1992; Towell et al., 1990) to directly install the advice into the agent. In one approach to knowledge-based neural networks, KBANN (Towell et al., 1990; Towell & Shavlik, in press), a set of propositional rules are re-represented as a neural network. KBANN converts a ruleset into a network by mapping the "target concepts" of the ruleset to output units and creating hidden units that represent the intermediate conclusions. In RATLE we extend the KBANN approach to accommodate a programming language we have developed for representing advice.

Step 5. Judge the value of the advice.

The final step of the advice-taking process is to evaluate the advice. One can also envision that in some circumstances – such as a game-learner that can play against itself (Tesauro, 1992) or when an agent builds an internal world model (Sutton, 1991) – it would be straightforward to empirically evaluate the new advice. In RATLE we view the process of evaluating advice as having two parts: (i) the evaluation from the point of view of the agent, who must decide if the advice is useful, and (ii) the evaluation from the point of view of the observer, who must decide if the advice had the desired effect on the behavior of the agent.

The agent performs the process of evaluating the advice by further reinforcement learning. Once the advice is incorporated into the agent, the agent returns to exploring its environment making using of and updating its knowledge. This process allows the agent to evaluate and refine the advice. To allow the observer to evaluate the advice we let the observer watch the performance of the system after the advice has been inserted. This may lead to further advice – thus starting the cycle over.

Summary

Figure 1 shows a diagram of our system and its interaction with the observer and the agent. Note that the process is a cycle: the observer develops advice based on the agent's behavior, RATLE translates the advice and inserts it into the agent, and the agent then tests the advice – which may result in behavior changes that cause the cycle to start over. In the following section we discuss specific details of our current implementation of this approach.

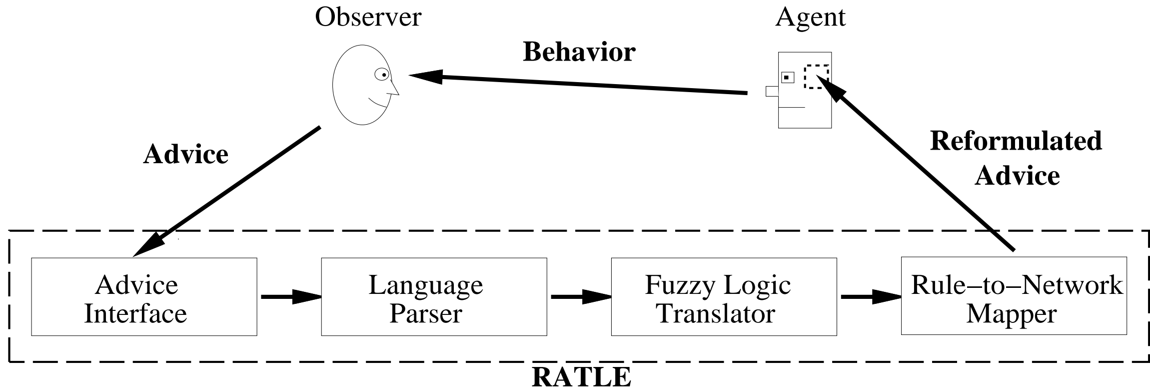

Figure 1: Interaction of the RATLE approach for advice-taking with the reinforcement-learning agent and the external observer.

3 RATLE - The Details

In this section we further describe RATLE, our system for reinforcement learners that can accept advice. To better understand RATLE, we first present an outline of connectionist Q-learning (Sutton, 1988; Watkins, 1989), the form of reinforcement learning we use in our implementation, and KBANN, a technique for incorporating knowledge in the form of rules into a network. We then discuss our extensions to these techniques by showing how we implement each of the steps presented in the previous section.

Background

Figure 2 shows the general structure of a reinforcement learner, augmented (in bold), with an observer that provides advice. In RL, the learner senses the current world state, chooses an action to execute, and occasionally receives rewards and punishments. Based on these reinforcements from the environment, the task of the learner is to improve its action-choosing module such that it increases the total amount of reinforcement it receives. In our augmentation, an observer watches the learner and periodically provides advice, which RATLE incorporates into the action-choosing module of the RL agent.

In Q-learning (Watkins, 1989) the action-choosing module is a *utility function* that maps states and actions to a numeric value. The utility value of a particular state and action is the predicted future (discounted) reward that will be achieved if that action is taken by the agent in that state. It is easy to see that given a perfect version of this function, the optimal plan is to simply choose, in each state that is reached, the action with the largest utility.

To learn a utility function, a Q-learner starts out with a randomly chosen utility function and explores its environment. As the agent explores, it continually makes

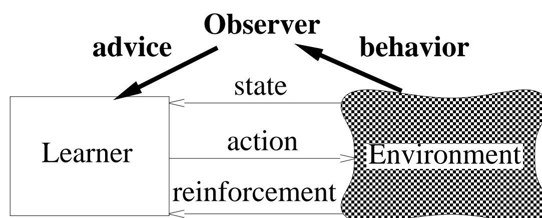

Figure 2: Reinforcement learning with an external advisor.

predictions about the reward it expects and then updates its utility function by comparing the reward it actually receives to its prediction. In *connectionist* Q-learning, the utility function is implemented as a neural network, whose inputs describe the current state and whose outputs are the utility of each action.

To map the knowledge represented in the advice provided by the user, we extended the KBANN algorithm. KBANN is a method for incorporating knowledge in the form of simple propositional rules into a neural network. In a KBANN network, the units of the network represent Boolean concepts. A concept is assumed to be true if the unit representing the concept is highly active (near 1) and false if the unit is inactive (near 0). To represent the meaning of a set of rules, KBANN connects units with highly-weighted links and sets unit biases (thresholds) in such a manner that the (non-input) units emulate AND or OR gates, as appropriate. Figure 3 shows an example of this process for a set of simple propositional rules.

In RATLE we use a programming language to specify advice instead of Prolog-like rules. In order to map this more complex language, we make use of hidden units that record state information about the neural network. These units are recurrent and record the activation of a hidden unit from the previous activation of the network (they "remember" the previous activation value). In the section below on mapping program constructs into a neural networks we discuss how these units are used.

In the following subsections we give specific details on our implementation of the advice-taking strategy we discussed in the last section. In our implementation, we assume that we are giving advice to an agent performing connectionist Q-learning.

Step 1. Request the advice.

To give advice users simply interrupt the agent's execution and type in their advice. RATLE then begins the process of transforming the advice so that it may be added to the agent.

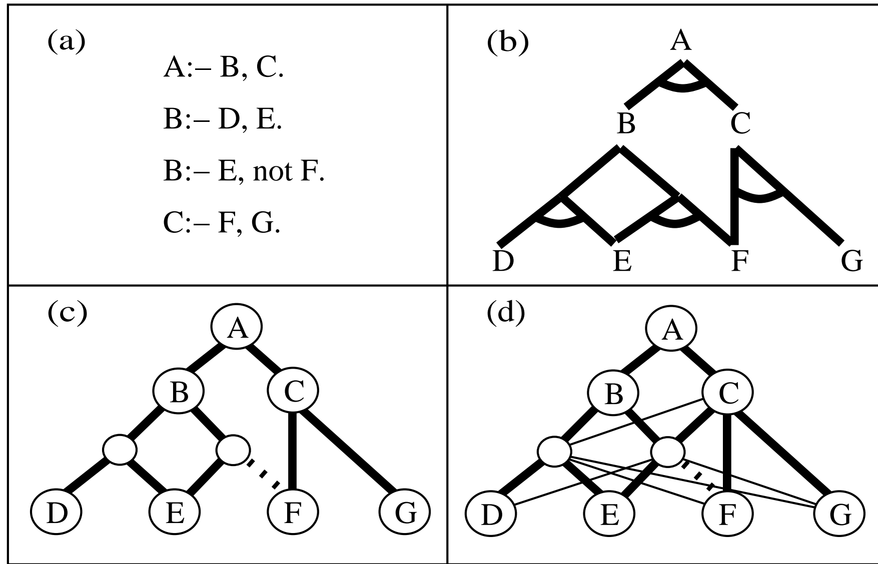

Figure 3: Sample of the KBANN (Towell et al., 1990) algorithm: (a) propositional rule set; (b) the dependencies of the rules; (c) each proposition is represented as a unit (units are also added for other vertices of the dependency graph); and (d) low-weighted links are added between layers as a basis for future learning (an antecedent can be added to a rule by increasing one of these weights).

Step 2. Convert the advice to an internal representation.

We built the parser for RATLE using the standard Unix compiler tools *lex* and *yacc*. A complete grammar for RATLE's advice language is shown in Table 1. Our advice-taking language has three main programming constructs: IF-THEN rules, WHILE loops, and REPEAT loops.

The IF-THEN constructs in the language actually serve two purposes. An IF-THEN can be used to specify that a particular action should be taken in a particular situation. One can also be used to create a new intermediate term. In this case the conclusion of the IF-THEN rule is not an action, but the name of the new intermediate term. This allows the user to build a new set of descriptive terms based on the original terms. For example, the user may want to create an intermediate term *NotLarge* that is true if an object is *Small* or *Medium*. Using an IF-THEN rule the user could create this term, and *NotLarge* could then be used as an antecedent in future rules.

In order to specify antecedents and consequents for the IF-THEN and looping constructs the user must be able to refer to conditions and actions in the specific environment for which they are offering advice. The set of possible actions and the input sensors are defined when creating the initial problem description. The user

Table 1: The grammar used for parsing RATLE's advice language.

A piece of advice may be a single construct or multiple constructs.

$$\begin{array}{lll} rules & \leftarrow & rule \\ & & \mid rules \; ; \; rule \end{array}$$

The grammar has three main constructs: IF-THENs, WHILEs, and REPEATs.

$$\begin{array}{lll} rule & \leftarrow & \text{IF } ante \text{ THEN } act \; else \text{ END} \\ & & \mid \text{WHILE } ante \text{ DO } act \; postact \text{ END} \\ & & \mid pre \text{ REPEAT } act \text{ UNTIL } ante \; postact \text{ END} \end{array}$$

$$\begin{array}{lll} else & \leftarrow & \varepsilon \mid \text{ELSE } act \\ postact & \leftarrow & \varepsilon \mid \text{THEN } act \\ pre & \leftarrow & \varepsilon \mid \text{WHEN } ante \end{array}$$

A MULTIACTION construct specifies a *series* of actions to perform.

$$\begin{array}{lll} act & \leftarrow & cons \mid \text{MULTIACTION } clist \text{ END} \\ clist & \leftarrow & cons \mid cons \; clist \\ cons & \leftarrow & \text{Term_Name} \mid (\; corlst \;) \\ corlst & \leftarrow & \text{Term_Name} \mid \text{Term_Name} \vee corlst \end{array}$$

Antecedents are logical combinations of terms and fuzzy conditionals.

$$\begin{array}{lll} ante & \leftarrow & \text{Term_Name} \\ & & \mid (\; ante \;) \\ & & \mid \neg \; ante \\ & & \mid ante \wedge ante \\ & & \mid ante \vee ante \\ & & \mid \text{Quantifier_Name Object_Name IS } desc \end{array}$$

The descriptor of the fuzzy conditional is a logical combination of fuzzy terms.

$$\begin{array}{lll} desc & \leftarrow & \text{Descriptor_Name} \\ & & \mid \neg \; desc \\ & & \mid \{ \; dlist \; \} \\ & & \mid (\; dexpr \;) \\ dlist & \leftarrow & \text{Descriptor_Name} \mid \text{Descriptor_Name} \; , \; dlist \\ dexpr & \leftarrow & desc \\ & & \mid dexpr \wedge dexpr \\ & & \mid dexpr \vee dexpr \end{array}$$

Table 2: An example of a fuzzy condition in our advice language.

Form	*quantifier*	*object*	IS/ARE	*descriptor*
Example	Few	Enemies	ARE	Near

makes antecedents for the rules and looping constructs out of logical combinations of the input terms plus any intermediate terms the user creates. To make the language easier to use we also allow the observer to state fuzzy conditions, which we believe provide a natural way to articulate imprecise advice. The particular fuzzy terms used are domain dependent and must be created as part of defining the initial environment. The process of translating a fuzzy condition into a corresponding neural network construct is discussed in the next step.

Step 3. Convert the advice into a usable form.

Step 2's parsing of the language converts the advice into an internal data structure. As we shall see in Step 4, most of the concepts that can be expressed in our grammar can be directly translated into additions to a neural network, but the fuzzy conditions are somewhat different in that we must first determine the combination of input units needed to match fuzzy condition. In this section we describe how to map fuzzy conditions of the form shown in Table 2. We believe that fuzzy conditions of this form apply to environments where the input sensors measure quantities of objects and where the objects measured by a sensor share other properties (e.g. an input sensor counts how many objects of type A are to the North – being North is the property these objects share). For other environments, other types of fuzzy conditions may be more applicable, thus necessitating extension of this notation.

Our method for performing the process of mapping a fuzzy condition to a corresponding neural network construct is actually a two-part process; first we determine the input units to which a fuzzy condition corresponds, then we map the fuzzy condition to the neural network. In this discussion we treat this as a single process, though the second part of the process, mapping the fuzzy condition into the neural network, would more correctly fit into our discussion of Step 4.

Our method for representing fuzzy conditions is based on the method proposed by Berenji and Khedkar (1992). To better understand this process, consider the example in Table 2. In our fuzzy conditions *quantifiers* are fuzzy terms describing the number of objects needed for a match (e.g. A, No, Many, Few); *objects* are labels for different things that may occur in the environment (e.g. Obstacle, Enemy, Food); and *descriptors* are fuzzy terms that describe properties objects may have (e.g. Near, North, Flat). The condition shown in Table 2 refers to objects named "Enemies" and uses the fuzzy terms "Few" and "Near," which must be defined by the user. Assume that "Few" is defined as being two or three objects, and that "Near" means a distance

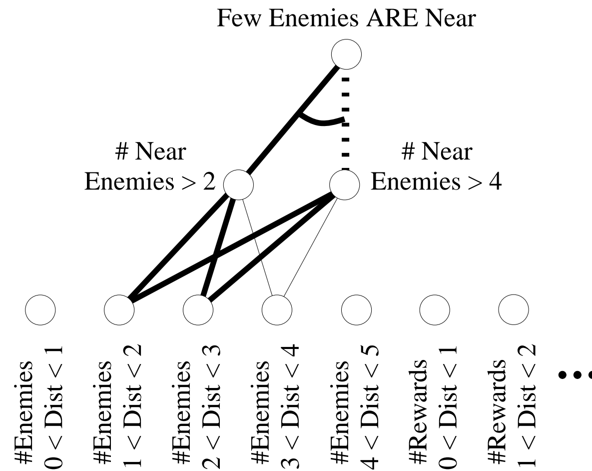

Figure 4: An example of how RATLE converts a fuzzy condition to a set of weights, biases, and hidden units. This example has input sensors where there are at least two types of objects in the world ("Enemies" and "Rewards") and the input sensors count the number of Enemy and Reward objects that have the property of falling within a certain distance from the agent. The input units that represent Enemy objects and match the fuzzy term "Near" are connected to hidden units that count if there are at least two "Near Enemies" and at least four "Near Enemies," respectively. The unit representing "Few Enemies ARE Near" is then the conjunct of two "Near Enemies" and not four "Near Enemies."

between 1.0m and 3.5m.

The goal of RATLE when mapping a fuzzy condition is to create a hidden unit that is highly active when the input matches the fuzzy condition, and is inactive otherwise. In order to do this, RATLE first determines which input units match the fuzzy terms. Then RATLE determines a set of weights and biases for the new hidden unit that calculates when the fuzzy terms are met (drawing on ideas from KBANN and Berenji and Khedkar's work).

Figure 4 shows the set of hidden units and weights that are added to a network to produce a hidden unit representing the condition "Few Enemies ARE Near." The first part of RATLE's process for making these additions is to determine the set of input units that are relevant for the fuzzy condition. RATLE first restricts the input units to those that represent objects of the type indicated in the antecedent – in this case "Enemies." RATLE then determines how well each of these input units matches the descriptor. If an input unit completely matches a description (an input unit representing enemies at a distance of 1.0m to 2.0m would completely match "Near"), RATLE assigns it a match strength of 1.0. For input units that only partially match

a descriptor RATLE makes an assumption of independent distribution[2] and assigns a match strength equal to the fraction of the range of the input unit that is overlapped by the descriptor. For example, since we have defined "Near" as distances between 1.0m and 3.5m, the input unit representing Enemies at a distance of 3.0m to 4.0m matches only partially. RATLE assigns this unit a match strength of 0.5 since the region from 3.0m to 3.5m out of the input unit's total region of 3.0m to 4.0m matches.

Once RATLE determines the set of input units that match the object type and descriptor, it examines the quantifier portion of the antecedent. The quantifier specifies how many of the appropriate type of object are needed to achieve a match. If the quantifier is a range, RATLE actually creates three hidden units: one that tests if the number of matching objects are enough to exceed the low end of the range; one that tests if the number of objects exceeds the high end of the range; and one that tests if the first two units produced are active (true) and inactive (false), respectively. So, in our example we get a hidden unit that tests if there are at least two Near Enemies, one that tests if there are at least four Near Enemies, and one that tests if the first is true and the second false (i.e., there are at least two Near enemies, but not four or more). For quantifiers that are open ended (e.g. "Many" means more than three, etc.), we need only create a single hidden unit.

To set the weights and bias for the unit labeled "# Near Enemies > 2," RATLE uses a technique similar to the standard KBANN technique generalized for the fact that we are counting a number of objects, and that we may be counting over several hidden units. In our example, this means we create weights from the input units representing Enemies at a distance of 1.0m to 2.0m, Enemies at a distance of 2.0m to 3.0m, and Enemies at a distance of 3.0m to 4.0m, such that the net input to the hidden unit will exceed the bias if the count of Enemies is at least two across all of these inputs. Once the weight is set for a particular input unit, RATLE multiplies the weight by the match strength of the input unit (calculated previously) to represent that only part of some inputs should be counted. The weights and bias of the unit representing four Near Enemies are calculated similarly, while those for the unit representing the conjunct of these two units are set using the standard KBANN process.

The result of this process is a hidden unit that is active when the condition "Few Enemies ARE Near" is true, and false otherwise. Note that the weights and biases of the hidden units created for the fuzzy antecedent can be changed by learning in the world, and thus the fuzzy antecedent can be refined by learning.

[2]This means that we assume the objects counted by an input sensor are evenly distributed across the input sensor's range of values. For example, if an input sensor measures three Enemy objects that are at a distance of 1m to 2m from the agent, we assume the actual distances to the objects are may fall anywhere in the range from 1m to 2m with equal probability.

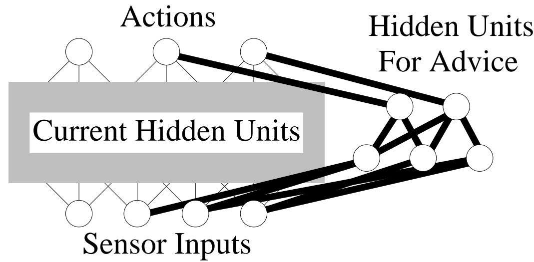

Figure 5: RATLE adds advice to the neural network by adding hidden units that correspond to the advice.

Step 4. Integrate the reformulated advice into the agent's current knowledge base.

After RATLE has reformulated any fuzzy conditions, it proceeds to map all of the other advice into the agent's neural-network utility function. To do this, we made three extensions to the standard KBANN algorithm: (i) advice can contain multi-step plans; (ii) it can contain loops; and (iii) it can refer to previously defined terms. For each of our extensions, incorporating advice involves adding hidden units representing the advice to the existing neural network as shown in Figure 5. Note that the inputs and outputs to the network remain unchanged; the advice only changes how the function from states to the utility of actions is calculated.

Table 3 shows some sample advice one might provide to an agent learning to play a video game. We will use these samples to illustrate the process of integrating advice into a neural network. The left column contains the advice as input to our programming language, the center column shows the advice in English, and the right column illustrates the advice.

As an example of a multi-step plan, consider the first entry in Table 3. Figure 6 shows the network additions that represent this advice. RATLE first creates a hidden unit (labeled A) that represents the conjunction of (i) an enemy being near and west and (ii) an obstacle being adjacent and north. It then connects this unit to the action $MoveEast$, which is an existing output unit (recall that the agent's utility function maps states to values of actions); this constitutes the first step of the two-step plan. RATLE also connects unit A to a newly-added hidden unit called $State1$ that records when unit A was active in the previous state. It next connects $State1$ to a new input unit called $State1_{-1}$. This *recurrent* unit becomes active ("true") when $State1$ was active for the previous input (we need a recurrent unit to implement multi-step plans). Finally, it constructs a unit (labeled B) that is active when $State1_{-1}$ is true and the previous action was an eastward move (the input includes the previous action taken in addition to the current sensor values). When active, unit B suggests moving north – the second step of the plan.

Table 3: Samples of advice in our advice language.

Advice	English Version	Pictorial Version
IF An Enemy IS (Near ∧ West) ∧ An Obstacle IS (Near ∧ North) THEN MULTIACTION MoveEast MoveNorth END END;	If an enemy is near and west and an obstacle is adjacent and north, hide behind the obstacle.	

WHEN Surrounded ∧
OKtoPushEast ∧
An Enemy IS Near
REPEAT
MULTIACTION
PushEast
MoveEast
END
UNTIL ¬ OKtoPushEast ∨
¬ Surrounded
END;
	When the agent is surrounded, pushing east is possible, and an enemy is near, then keep pushing (moving the obstacle out of the way) and moving east until there is nothing more to push or the agent is no longer surrounded.	

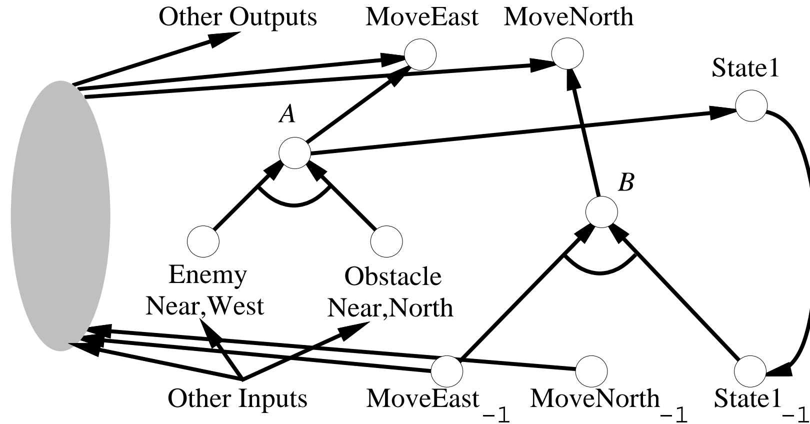

Figure 6: RATLE's translation of the first piece of advice. The large ellipse at left represents the original hidden units. Arcs show units and weights that are set to make a conjunctive unit. We also add, as is typical in knowledge-based networks, zero-weighted links (not shown) to other parts of the current network. These links support subsequent refinement.

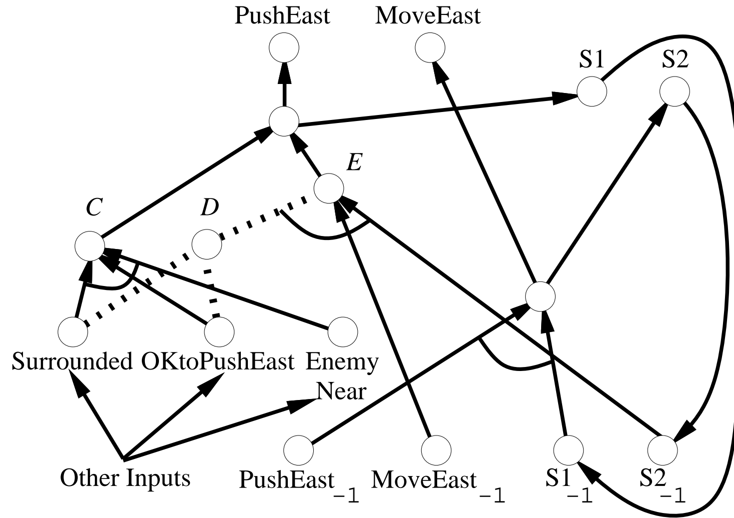

Figure 7: RATLE's translation of the second piece of advice. Dotted lines show negative weights. As with other translations, the units shown are added to the existing network (not shown).

RATLE assigns a high weight[3] to the arcs coming out of units A and B. This means that when either unit is active, the total weighted input to the corresponding output unit will be increased, thereby increasing the utility value for that action. Note, however, that this does not guarantee that the appropriate action will be chosen when units A or B are active. Also, notice that during subsequent training the weight (and thus the definition) of a piece of advice may be substantially altered.

The second piece of advice in Table 3 also contains a multi-step plan, but this time it is embedded in a REPEAT. Figure 7 shows RATLE's additions to the network for this advice. The key to translating this construct is that there are two ways to invoke the two-step plan. The plan executes when the WHEN condition is true (unit C) and also when the plan was just run and the UNTIL condition is false. Unit D is active when the UNTIL condition is met, while unit E is active when the UNTIL is unsatisfied and the agent's two previous actions were pushing and then moving east.

A final issue for RATLE is dealing with advice that involves previously defined terms. This frequently occurs, since advice generally indicates new situations in which to perform existing actions. For new definitions of agent *actions* RATLE adds a strongly weighted link from the new concept to the output unit representing that action. This is done so that in the situations where the advice is active the utility of the action will then be higher than the utility of any other action (thus the action

[3]Through empirical investigation we chose a value of 2.0 for these weights. A topic of our future research is to more intelligently select this value.

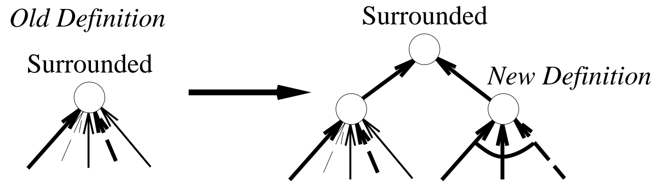

Figure 8: Incorporating the definition of a term that already exists.

will be chosen). Figure 8 shows how we address this issue for terms other than agent actions.[4] We add a new definition of an existing term by first creating the representation of the added definition and making a copy of the unit representing the existing definition. We create a new unit, which becomes the term's new definition, representing the disjunction of the old and new definitions. This process is analogous to how KBANN processes multiple rules with the same consequent. We do not use this process with output units because we do not necessarily want output units activity to be near 1 (max utility) since overpredictions of utility can cause problems for connectionist Q-learning (Thrun, 1994).

Step 5. Judge the value of the advice.

Once RATLE inserts the advice into the RL agent, the agent returns to exploring its environment. In this way the agent can evaluate the advice empirically and refine the advice based on its further experience. This is a key step in our process because RATLE cannot determine the optimal weights to use for the new piece of advice; instead we use RL to adjust the weights towards optimal based on experience.

We also allow the observer to judge the value of their advice by letting the observer watch the agent perform subsequent tasks. This allows the observer to make further suggestions to the agent. We do not currently allow the observer to directly change its advice, although it might be useful for the user to be able to retract bad advice or provide refinements to advice. We instead rely on RL to wash out the effects of bad advice.

4 Experimental Study

We next empirically judge the value of using our system, RATLE, for providing advice to an RL agent.

[4]We tested a number of mechanisms for adding new definitions of existing terms. We chose two separate approaches because the approach used on the non-output units resulted in a significant drop in agent performance after advice was inserted.

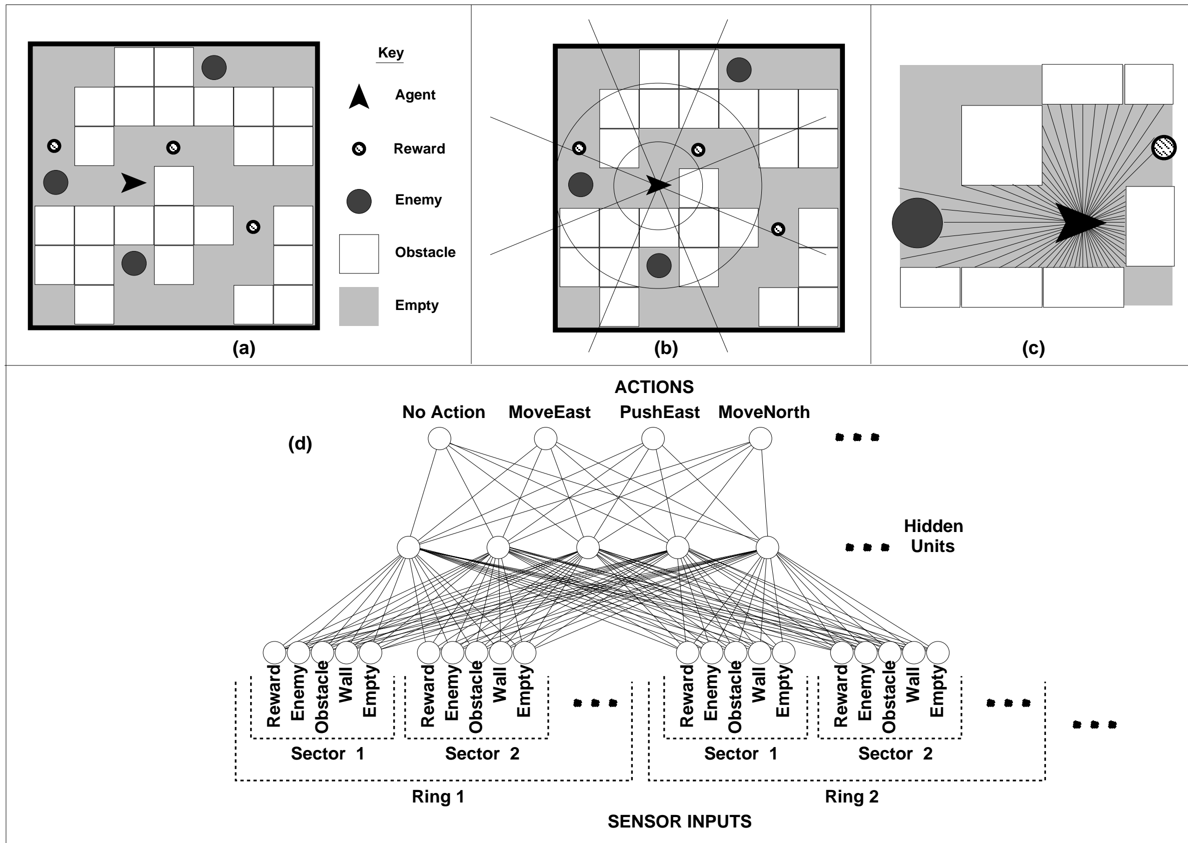

Figure 9: Our sample test environment: (a) sample configuration; (b) sample division of the environment into sectors; (c) the agent measuring the distance to the nearest occluding object along a fixed set of arcs; (d) a neural network that computes the utility of actions.

4.1 Testbed

Figure 9a illustrates our test environment. Our task is similar to those explored by Agre and Chapman (1987) and Lin (1992). The agent can perform nine actions: *moving* and *pushing* in each of the directions East, North, West and South; and *doing nothing*. Pushing moves the obstacles in the environment – when the agent is next to an obstacle and pushes it, the obstacle slides until it encounters another obstacle or the edge of the board. If the obstacle is unable to move (there is an obstacle or wall behind it), the obstacle object disintegrates. Reward objects are collected when the agent touches them and destroyed if an enemy touches them. A moving obstacle will destroy both reward and enemy objects if it touches them (since moving obstacles are much faster than the agent and they move away from the agent's push it is impossible for the obstacle to hit the agent).

The initial mazes are generated randomly using a maze creation program that randomly lays out lines of obstacles and then creates connections between "rooms." The percentage of the total board covered by obstacles is controlled by a parameter, as are the number of enemy and reward objects. The enemy, reward, and agent objects are randomly deposited on the board with the caveat that the enemies are required to be at least a fixed distance (another parameter) away from the agent at the start.

The agent receives reinforcement signals when: (i) an enemy eliminates the agent by touching the agent (–1.0); (ii) the agent collects one of the reward objects (+0.7); and (iii) the agent destroys an enemy by pushing an obstacle into it (+0.9). Note that the agent receives no signal if an action it takes fails. Each enemy moves randomly unless the agent is in sight, in which case it moves toward the agent. Enemies may move off the board (they appear again at a random interval), but the agent is constrained to remain on the board.

We do not assume a global view of the environment, but instead use an agent-centered sensor model. It is based on partitioning the world into a set of sectors around the agent (see Figure 9b). Each sector is defined by a minimum and maximum distance from the agent and a minimum and maximum angle with respect to the direction the agent is facing. The agent calculates the percentage of each sector that is occupied by each type of object – reward, enemy, obstacle, or wall. To calculate the sector occupancy, we assume the agent is able to measure the distance to the nearest occluding object along a fixed set of angles around the agent (see Figure 9c). This means that the agent is only able to represent the objects in direct line-of-sight from the agent (for example, the enemy object to the South of the agent is out of sight). The value of each object in a sector is just the number of distances that fall within that sector that correspond to that type of object divided by the maximum number of distances that could fall within a sector. So for example, given Figure 9b, the agent's percentage for "obstacle" would be high for the sector to its right. These percentages constitute the input to the neural network (see Figure 9d).

4.2 Methodology

We train the agents for a fixed number of *episodes* for each experiment. An episode consists of placing the agent into a randomly generated, initial environment, and then allowing it to explore until it is captured or a threshold of 500 steps is reached. Each of our environments contains a 7x7 grid with approximately 15 obstacles, 3 enemy agents, and 10 rewards. We use three randomly-generated sequences of initial environments as a basis for the training episodes. We train 10 randomly initialized networks on each of the three sequences of environments; hence, we report the averaged results of 30 neural networks. We estimate the average total reinforcement (the average sum

Table 4: Testset results for the baseline and the four different types of advice; each of the gains (over the baseline) in average total reinforcement for the four sets of advice is statistically significant at the $p < 0.01$ level (i.e., with 99% confidence).

Advice Added	Average Total Reinforcement
None (baseline)	1.32
SimpleMoves	1.92
NonLocalMoves	2.01
ElimEnemies	1.87
Surrounded	1.72

of the reinforcements received by the agent)[5] by freezing the network and measuring the average reinforcement on a testset of 100 randomly-generated environments.

We chose parameters for our Q-learning algorithm that are similar to those investigated by Lin (1992). The learning rate for the network is 0.15, with a discount factor of 0.9. To establish a baseline system, we experimented with various numbers of hidden units, settling on 15 since that number resulted in the best average reinforcement for the baseline system.

After choosing an initial network topology, we then spent time acting as a user of RATLE, observing the behavior of the agent at various times. Based on these observations, we wrote several collections of advice. For use in our experiments, we chose four sets of advice (see Appendix), two that use multi-step plans (referred to as *ElimEnemies* and *Surrounded*), and two that do not (*SimpleMoves* and *NonLocalMoves*).

4.3 Results and Discussion

For our first experiment, we evaluate the hypothesis that our system can in fact take advantage of advice. After 1000 episodes of initial learning, we measure the value of (independently) providing each of the four sets of advice to our agent using RATLE. We train the system for 2000 more episodes after adding the advice before measuring the average cumulative reinforcement on the testset. (The baseline is trained for 3000 episodes). Table 4 reports the averaged testset reinforcement; all gains over the baseline system are statistically significant. Note that the gain is higher for the simpler pieces of advice *SimpleMoves* and *NonLocalMoves*, which do not incorporate multi-step plans. This suggests the need for further work on taking complex advice;

[5] We report the average total reinforcement rather than the average discounted reinforcement because this is the standard for the RL community. Graphs of the average *discounted* reward are qualitatively similar to those shown in the next section.

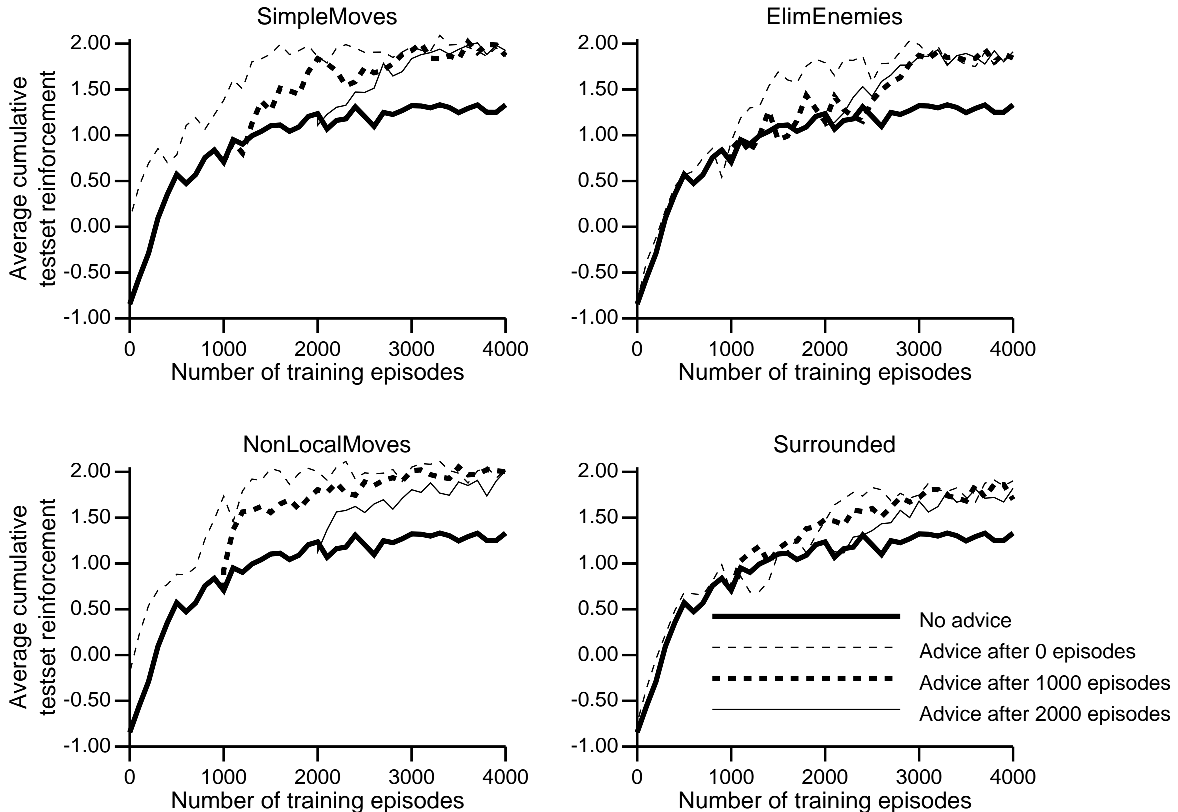

Figure 10: Testset results of four sample pieces of advice.

Table 5: Mean number of enemies captured, rewards collected, and number of actions taken for the experiments summarized in Table 4.

Advice Added	Enemies	Rewards	Survival Time
None (baseline)	0.15	3.09	32.7
SimpleMoves	0.28	3.79	39.6
NonLocalMoves	0.26	3.95	39.1
ElimEnemies	0.44	3.50	38.3
Surrounded	0.30	3.48	46.2

however the multi-step advice may simply be less useful.

In our second experiment we investigate the hypothesis that the observer can beneficially provide advice at any time during training. To test this, we insert the four sets of advice at different points in training (after 0, 1000, and 2000 episodes). Figure 10 contains the results for the four pieces of advice. They indicate the learner does indeed converge to approximately the same expected reward no matter when the advice is presented.

Each of our pieces of advice to the agent addresses specific subtasks: collect-

ing rewards (*SimpleMoves* and *NonLocalMoves*); eliminating enemies (*ElimEnemies*); and avoiding enemies, thus surviving longer (*SimpleMoves*, *NonLocalMoves*, and *Surrounded*). Hence, it is natural to ask how well each piece of advice meets its intent. Table 5 reports statistics on the components of the reward. These statistics show that the pieces of advice do indeed lead to the expected improvements. For example, our advice *ElimEnemies* leads to a much larger number of enemies eliminated than the baseline or any of the other pieces of advice.

5 Future Work

There are a number of experiments we intend to perform to further evaluate RATLE. Our current experiments have only demonstrated the value of giving a single piece of advice. We plan to empirically study the effect of providing multiple pieces of advice at different times during training. It is our hypothesis that given useful advice the agent will show gains for each piece of advice provided.

We also intend to evaluate the use of "replay" in training our agent. Replay involves the periodic retraining of an agent on past states and reinforcements stored by the agent. Replay has been shown to greatly reduce the number of training examples needed to learn a policy function (Lin, 1992).

Our agent currently suffers from the limitation that it must learn the value of actions when facing each of the four directions East, North, West, and South. We plan to examine the use of soft weight-sharing (Nowlan & Hinton, 1992) to foster transfer of learning across the corresponding rules for each direction.

We also plan to evaluate our approach in other domains. One especially interesting area of interest is the training of agents in a cooperative environment, where tasks are performed by multiple agents working together. Such a domain is interesting in that the user may want to specify means for the agents to communicate toward reaching shared goals.

We intend to make a number of further extensions to RATLE using our growing understanding of the problem of giving advice to RL agents. We plan to extend RATLE's language for instructing agents in a number of ways. First, we plan to allow the user to provide advice in the form of warnings, i.e., "do not perform this action in this state." This could be especially useful for tasks where there are large negative reinforcements, as the user could specify actions that lead to these reinforcements, and therefore should be avoided. Second, we plan to add a method for the user to declare and make use of state units, so that users may (if they choose) specify information to remember and use in their plans. Using this mechanism the user could specify information to retain that might be useful for solving long-term problems. Third, we plan to extend the multi-step plan construct so that the user may include conditions which must be checked as the plan is being executed (rather than just conditions that

start the plan).

One important part of the framework for advice-taking that we have thus far spent little time on is the evaluation phase. We intend to introduce mechanisms for users to track their advice in terms of how often and how well it performs, and then to remove the advice if it is proving detrimental. This step might be combined with replay, in that once a hidden unit is removed from a neural network the system may need to do some significant retraining to reset the weights and biases that were indirectly affected by the removed weight. We will also plan to allow users to make refinements to their advice (adding or deleting preconditions, for example).

6 Related Work

Our work on RATLE relates to a number of recent research efforts. This related work can be roughly divided into four groups: (i) providing advice to a problem solver, (ii) giving advice to a problem solver employing reinforcement learning, (iii) developing programming languages for interacting with agents, and (iv) creating knowledge-based neural networks.

Providing advice to a problem solver

An early example of a system that makes use of advice is the ABSTRIPS planning system (Sacerdoti, 1974). In ABSTRIPS, a human user assigns initial "criticalities" to preconditions to cause the planner to focus on making key planning steps. In both ABSTRIPS and RATLE the user is in some way directing the system's search, but the ABSTRIPS mechanism requires the user to have some understanding of the search process, while in RATLE the user need not understand how the system works internally.

In Mostow's (1982) system FOO, general advice is *operationalized* by reformulating the advice into search heuristics. These search heuristics are then applied during problem solving. In FOO the advice is assumed to be correct and the learner has to learn how to convert the general advice into an executable plan based on its knowledge about the domain. Our system is different in that we try to directly incorporate general advice, but we do not provide a sophisticated means of operationalizing advice. Also, we do not assume the advice is correct; instead we use learning to refine and evaluate the advice.

More recently, Laird et al. (1990) created an advice-taking system called ROBO-SOAR based on the SOAR architecture (Laird et al., 1987). SOAR uses its knowledge to select operators to achieve goals. If it is unable to select from a set of operators or no operator is available an *impasse* arises. In this case SOAR creates a subgoal and applies itself to the subgoal. In ROBO-SOAR, a user can provide advice to the system during an impasse by suggesting which operators to explore in an attempt to resolve

the impasse. As with FOO and ABSTRIPS, the advice presented is used to guide the learner's search process, while in RATLE we directly incorporate the advice into the learner's knowledge and then refine that knowledge.

Huffman and Laird (1993) developed another SOAR-based system called INSTRUCTO-SOAR that allows an agent to interpret simple imperative statements such as "Pick up the red block" and "Move down." INSTRUCTOSOAR examines these instructions in the context of its current problem solving situation and uses explanation-based learning (Mitchell et al., 1986) to try to generate a general rule based on the advice that may be used in similar situations. RATLE differs from INSTRUCTOSOAR in that we provide a language for entering general advice rather than attempting to generalize specific advice.

Providing advice to a problem solver that uses reinforcement learning

A number of researchers have introduced methods for providing advice to a reinforcement learning agent. Lin (1992) designed a technique for allowing an RL agent to use advice in the form of sequences of teacher's actions. In his system the agent "replays" the teacher actions periodically to bias the agent toward the actions chosen by the teacher. Our approach differs in that RATLE inputs the advice in a general form; also, RATLE directly installs the advice into the learner rather than using the advice as a basis for learning.

Utgoff and Clouse (1991) developed a learner that consults a set of teacher actions if the action it chose resulted in significant error. This system has the advantage that it determines the situations in which it required advice, but is limited in that it may require advice more often than the user is willing to provide it. On the other hand, in RATLE users provide advice whenever they feel they have something to say.

Whitehead (1991) examined an approach similar to both Lin's and Utgoff & Clouse's that can learn both by receiving advice in the form of critiques (a reward indicating whether the chosen action was optimal or not), as well as learning by observing the actions chosen by a teacher. Clouse and Utgoff (1992) created a second system that takes advice in the form of actions suggested by the teacher. Both systems are similar to ours in that they can incorporate advice whenever the teacher chooses to provide it, but unlike RATLE they do not accept broadly applicable advice.

Thrun and Mitchell (1993) investigated a method for allowing RL agents to make use of prior knowledge in the form of neural networks. These neural networks are assumed to have been trained to predict the results of actions. This proves to be effective, but requires previously trained neural networks that are related to the task being addressed.

Gordon and Subramanian (1994) developed a system that is closely related to ours. Their system employs genetic algorithms, an alternate approach for learning from reinforcements. Their agent accepts high-level advice of the form IF *conditions*

THEN ACHIEVE *goal*. It operationalizes these rules using its background knowledge about goal achievement. Our work primarily differs from Gordon and Subramanian's in that RATLE uses connectionist Q-learning instead of genetic algorithms, and in that RATLE's advice language focuses on actions to take rather than goals to achieve. Also, we allow advice to be given at any time during the training process, but as with the Mostow's FOO system, our system does not have the operationalization capability of Gordon and Subramanian's system.

Developing robot-programming languages

A number of researchers have introduced languages for programming robot-like agents (e.g. Brooks, 1990; Chapman, 1991; Gat, 1991; Kaelbling, 1987; Nilsson, 1994). These systems do not generally focus on programming agents that learn to refine their programs. Suppes (1991) investigated how a robot can learn to understand a human's instructions, but he focuses on the problem of understanding natural language, which is beyond the scope of our system. We chose to focus on the more limited task of providing a language for a human to instruct a robot agent that is relatively simple to parse, but still powerful enough for useful and reasonably natural interaction.

Diederich (1989) devised a method that accepts instructions in a symbol form. He uses the instructions to create examples, then trains a neural network with these examples to incorporate the instructions, as opposed to directly installing the instructions.

Siegelman (1994) proposed a technique for converting programs expressed in a general-purpose, high-level language into a type of recurrent neural networks. Her system is especially interesting in that it provides a mechanism for performing arithmetic calculations, but the learning abilities of this system have not yet been empirically demonstrated.

Gruau (1994) developed a compiler that can translate Pascal programs into neural networks. While his approach has so far only been tested on simple programs, his technique may prove applicable to the task of programming agents. Gruau's approach includes two methods for refining the networks he produces: a genetic algorithm and a hill-climber. The main difference between Gruau's system and ours is that the networks we produce can be refined using standard connectionist techniques such as backpropagation, while Gruau's networks require the development of a specific learning algorithm, since they require integer weights (-1,0,1) and incorporate functions that do not have derivatives.

Creating knowledge-based neural networks

As mentioned earlier, RATLE represents an extension of the KBANN system (Towell et al., 1990; Towell & Shavlik, in press). Our work extends knowledge-based neural

networks to a new task and shows that "domain theories" can be supplied incrementally (as opposed to providing the domain theory at the start of the learning task). Our work on RATLE is similar to our earlier work with the FSKBANN system (Maclin & Shavlik, 1993). FSKBANN uses a type of recurrent neural network introduced by Jordan (1989) and Elman (1990) that maintains information from previous activations using the recurrent network links. FSKBANN extends KBANN to deal with *state* units, but it does not create *new* state units.

Similarly, Omlin and Giles (1992) insert prior knowledge about a finite-state automaton into a recurrent neural network. As with FSKBANN, Omlin and Giles do not make use of knowledge provided after training has begun, nor do they study RL tasks.

Lin (1993) has also investigated the idea of having a learner use prior state knowledge. Lin uses an RL agent that has as input not only the current input state, but also some number of the previous input states. The difference between Lin's approach and ours is that we use the advice to determine a portion of the previous information to keep rather than trying to keep all of the previous information; keeping all of the information about previous states makes learning more difficult.

7 Conclusions

We present a system called RATLE that allows a reinforcement-learning agent to take advantage of suggestions provided by an external observer. The observer communicates advice using a simple programming language, one that does not require the observer to have any knowledge of the agent's internal workings. RATLE directly installs the advice into a neural network that represents the agent's utility function, and the agent then refines this knowledge with further learning. Our experiments demonstrate the validity of this advice-taking approach, since each of four types of advice lead to statistically significant gains in expected future reward. Interestingly, our experiments show the gains in expected future reward does not depend on when the observer supplies the advice.

Acknowledgements

This research was partially supported by Office of Naval Research Grant N00014-93-1-0998 and National Science Foundation Grant IRI-9002413. We also wish to thank Carolyn Allex, Mark Craven, Diana Gordon, and Sebastian Thrun for their helpful comments on drafts of this paper.

Appendix – Four Sample Pieces of Advice

The four pieces of advice used in the experiments in Section 4 appear below. Recall that in our testbed the agent has two actions (moving and pushing) that can be executed in any of the four directions (East, North, West, and South). To make it easier for an observer to specify advice that applies in any direction, we defined the special term *dir*. During parsing, *dir* is expanded by replacing each rule containing it with four rules, one for each direction. Similarly we have defined a set of four terms {*ahead, back, side1, side2*}. Any rule using these terms leads to *eight* rules – two for each case where *ahead* is East, North, West and South and *back* is appropriately set. There are two for each case of *ahead* and *back* because *side1* and *side2* can have two sets of values for any value of *ahead* (e.g. if *ahead* is North, *side1* could be East and *side2* West or vice-versa).

SimpleMoves

If An Obstacle is (NextTo ∧ *dir*) Then OkPush*dir* End;
If No Obstacle is (NextTo ∧ *dir*) ∧ No Wall is (NextTo ∧ *dir*)
  Then OkMove*dir* End;
If OkMove*dir* ∧ An Enemy is (Near ∧ ¬ *dir*)
  Then Move*dir* End;
If OkMove*dir* ∧ A Reward is (Near ∧ *dir*) ∧
  No Enemy is (Near ∧ *dir*) Then Move*dir* End;
If OkPush*dir* ∧ An Enemy is (Near ∧ *dir*)
  Then Push*dir* End

Grab rewards next to you; run from enemies next to you; push obstacles at enemies behind obstacles. [This leads to 20 rules.]

NonLocalMoves

If No Obstacle is (NextTo ∧ *dir*) ∧ No Wall is (NextTo ∧ *dir*)
  Then OkMove*dir* End;
If Many Enemy are (¬ *dir*) ∧ No Enemy is (Near ∧ *dir*) ∧
  OkMove*dir* Then Move*dir* End;
If OkMove*dir* ∧ An Enemy is (*dir* ∧ {Medium ∨ Far}) ∧
  No Enemy is (*dir* ∧ Near) ∧ A Reward is (*dir* ∧ Near)
  Then Move*dir* End

Run away if many enemies in a direction (even if they are not close), and move towards rewards even if there is an enemy in that direction (as long as the enemy is a ways off). [12 rules.]

ElimEnemies

If No Obstacle is (NextTo ∧ *dir*) ∧ No Wall is (NextTo ∧ *dir*)
  Then OkMove*dir* End;
If OkMove*ahead* ∧ An Enemy is (Near ∧ *back*) ∧
  An Obstacle is (NextTo ∧ *side1*) Then
  MultiAction
      Move*ahead* Move*side1* Move*side1*
      Move*back* Push*side2*
  End
End

When an enemy is closely behind you and a convenient obstacle is nearby, spin around the obstacle and push it at the enemy. [12 rules.]

Surrounded

If An Obstacle is (NextTo ∧ *dir*)
  Then OkPush*dir* End;
If An Enemy is (Near ∧ *dir*) ∨ A Wall is (NextTo ∧ *dir*) ∨
  An Obstacle is (NextTo ∧ *dir*) Then Blocked*dir* End;
If BlockedEast ∧ BlockedNorth ∧ BlockedSouth ∧ BlockedWest
  Then Surrounded End;
When Surrounded ∧ OkPush*dir* ∧ An Enemy is Near
  Repeat
    Multiaction Push*dir* Move*dir* End
  Until ¬ OkPush*dir*
End

When surrounded by obstacles and enemies, push obstacles out of the way and move through the holes. [13 rules.]

References

Agre, P. & Chapman, D. (1987). Pengi: An implementation of a theory of activity. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, (pp. 268–272), Seattle, WA.

Barto, A., Sutton, R., & Watkins, C. (1990). Learning and sequential decision making. In Gabriel, M. & Moore, J., editors, *Learning and Computational Neuroscience*. MIT Press, Cambridge, MA.

Berenji, H. & Khedkar, P. (1992). Learning and tuning fuzzy logic controllers through reinforcements. *IEEE Transactions on Neural Networks*, 3:724–740.

Brooks, R. (1990). The behavior language; user's guide. AI Memo 1227, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA.

Chapman, D. (1991). *Vision, instruction, and action.* MIT Press, Cambridge, MA.

Clouse, J. & Utgoff, P. (1992). A teaching method for reinforcement learning. In *Machine Learning: Proceedings of the Ninth International Workshop*, (pp. 92–101), Aberdeen, Scotland.

Cohen, P. & Feigenbaum, E. (1982). *The Handbook of Artificial Intelligence (volume 3).* William Kaufmann, Los Altos, CA.

Diederich, J. (1989). "Learning by instruction" in connectionist systems. In *Machine Learning: Proceedings of the Sixth International Workshop*, (pp. 66–68), Ithaca, NY.

Dietterich, T. (1991). Knowledge compilation: Bridging the gap between specification and implementation. *IEEE Expert*, 6:80–82.

Elman, J. (1990). Finding structure in time. *Cognitive Science*, 14:179–211.

Fu, L. M. (1989). Integration of neural heuristics into knowledge-based inference. *Connection Science*, 1:325–340.

Gat, E. (1991). ALFA: A language for programming reactive robotic control systems. In *IEEE International Conference on Robotics and Automation (volume 2)*, (pp. 1116–1121).

Gordon, D. & Subramanian, D. (1994). A multistrategy learning scheme for agent knowledge acquisition. *Informatica*, 17:331–346.

Gruau, F. (1994). *Neural Network Synthesis using Cellular Encoding and the Genetic Algorithm.* PhD thesis, Ecole Normale Superieure de Lyon, France.

Hayes-Roth, F., Klahr, P., & Mostow, D. J. (1981). Advice-taking and knowledge refinement: An iterative view of skill acquisition. In Anderson, J., editor, *Cognitive Skills and their Acquisition.* Lawrence Erlbaum, Hillsdale, NJ.

Huffman, S. & Laird, J. (1993). Learning procedures from interactive natural language instructions. In *Machine Learning: Proceedings on the Tenth International Conference*, (pp. 143–150), Amherst, MA.

Jordan, M. (1989). Serial order: A parallel, distributed processing approach. In Elman, J. & Rumelhart, D., editors, *Advances in Connectionist Theory: Speech.* Erlbaum, Hillsdale, NJ.

Kaelbling, L. (1987). REX: A symbolic language for the design and parallel implementation of embedded systems. In *Proceedings of the AIAA Conference on Computers in Aerospace*, Wakefield, MA.

Laird, J., Hucka, M., Yager, E., & Tuck, C. (1990). Correcting and extending domain knowledge using outside guidance. In *Machine Learning: Proceedings of the Seventh International Workshop*, (pp. 235–243), Austin, TX.

Laird, J., Newell, A., & Rosenbloom, P. (1987). SOAR: an architecture for general intelligence. *Artificial Intelligence*, 33:1–64.

Lin, L. (1992). Self-improving reactive agents based on reinforcement learning, planning, and teaching. *Machine Learning*, 8:293–321.

Lin, L. (1993). Scaling up reinforcement learning for robot control. In *Machine Learning: Proceedings on the Tenth International Conference*, (pp. 182–189), Amherst, MA.

Maclin, R. & Shavlik, J. (1993). Using knowledge-based neural networks to improve algorithms: Refining the Chou-Fasman algorithm for protein folding. *Machine Learning*, 11:195–215.

Mahadevan, S. & Connell, J. (1992). Automatic programming of behavior-based robots using reinforcement learning. *Artificial Intelligence*, 55:311–365.

McCarthy, J. (1958). Programs with common sense. In *Proceedings of the Symposium on the Mechanization of Thought Processes (volume I)*, (pp. 77–84). (Reprinted in M. Minsky, editor, 1968, *Semantic Information Processing.* Cambridge, MA: MIT Press, 403–409.).

Mitchell, T., Keller, R., & Kedar-Cabelli, S. (1986). Explanation-based generalization: A unifying view. *Machine Learning*, 1:47–80.

Mostow, D. J. (1982). Transforming declarative advice into effective procedures: A heuristic search example. In Michalski, R., Carbonell, J., & Mitchell, T., editors, *Machine Learning: An Artificial Intelligence Approach (volume 1)*. Tioga Press, Palo Alto.

Nilsson, N. (1994). Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research*, 1:139–158.

Nowlan, S. & Hinton, G. (1992). Simplifying neural networks by soft weight-sharing. *Neural Computation*, 4:473–493.

Omlin, C. & Giles, C. (1992). Training second-order recurrent neural networks using hints. In *Machine Learning: Proceedings of the Ninth International Workshop*, (pp. 361–366), Aberdeen, Scotland.

Sacerdoti, E. (1974). Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5:115–135.

Siegelmann, H. (1994). Neural programming language. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, Seattle, WA.

Suppes, P. (1991). *Language for Humans and Robots.* Blackwell, Cambridge, MA.

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

Sutton, R. (1991). Reinforcement learning architectures for animats. In Meyer, J. & Wilson, S., editors, *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior*. MIT Press, Cambridge, MA.

Tesauro, G. (1992). Practical issues in temporal difference learning. *Machine Learning*, 8:257–277.

Thrun, S. (1994). Personal communication.

Thrun, S. & Mitchell, T. (1993). Integrating inductive neural network learning and explanation-based learning. In *Proceedings of the Thirteenth Joint Conference on Artificial Intelligence*, (pp. 930–936), Chambery, France.

Towell, G. & Shavlik, J. (in press). Knowledge-based artificial neural networks. *Artificial Intelligence*.

Towell, G., Shavlik, J., & Noordewier, M. (1990). Refinement of approximate domain theories by knowledge-based neural networks. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, (pp. 861–866), Boston, MA.

Utgoff, P. & Clouse, J. (1991). Two kinds of training information for evaluation function learning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, (pp. 596–600), Anaheim, CA.

Watkins, C. (1989). *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge.

Whitehead, S. (1991). A complexity analysis of cooperative mechanisms in reinforcement learning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, (pp. 607–613), Anaheim, CA.