

## Creating Advice-Taking Reinforcement Learners

RICHARD MACLIN AND JUDE W. SHAVLIK      maclin@cs.wisc.edu and shavlik@cs.wisc.edu  
*Computer Sciences Dept., University of Wisconsin, 1210 W. Dayton St., Madison, WI 53706*

**Editor:** Leslie Pack Kaelbling

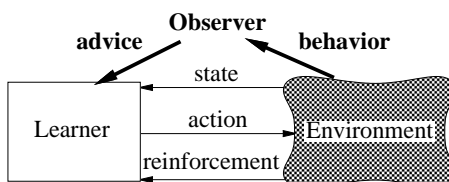
**Abstract.** Learning from reinforcements is a promising approach for creating intelligent agents. However, reinforcement learning usually requires a large number of training episodes. We present and evaluate a design that addresses this shortcoming by allowing a connectionist Q-learner to accept advice given, at any time and in a natural manner, by an external observer. In our approach, the advice-giver watches the learner and occasionally makes suggestions, expressed as instructions in a simple imperative programming language. Based on techniques from knowledge-based neural networks, we insert these programs directly into the agent's utility function. Subsequent reinforcement learning further integrates and refines the advice. We present empirical evidence that investigates several aspects of our approach and show that, given good advice, a learner can achieve statistically significant gains in expected reward. A second experiment shows that advice improves the expected reward regardless of the stage of training at which it is given, while another study demonstrates that subsequent advice can result in further gains in reward. Finally, we present experimental results that indicate our method is more powerful than a naive technique for making use of advice.

**Keywords:** Reinforcement learning, advice-giving, neural networks, Q-learning, learning from instruction, theory refinement, knowledge-based neural networks, adaptive agents

### 1. Introduction

A successful and increasingly popular method for creating intelligent agents is to have them learn from reinforcements (Barto, Sutton, & Watkins, 1990; Lin, 1992; Mahadevan & Connell, 1992; Tesauro, 1992; Watkins, 1989). However, these approaches suffer from their need for large numbers of training episodes. Several methods for speeding up reinforcement learning have been proposed; one promising approach is to design a learner that can also accept *advice* from an external observer (Clouse & Utgoff, 1992; Gordon & Subramanian, 1994; Lin, 1992; Maclin & Shavlik, 1994). Figure 1 shows the general structure of a reinforcement learner, augmented (in bold) with an observer that provides advice. We present and evaluate a connectionist approach in which agents learn from both experience and instruction. Our approach produces agents that significantly outperform agents that only learn from reinforcements.

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 broad advice such as “do not go into box canyons when opponents are in sight.” This approach is more appealing than the current alternative: repeatedly place the learner in similar



*Figure 1.* In basic reinforcement learning the learner receives a description of the current environment (the state), selects an action to choose, and receives a reinforcement as a consequence of selecting that action. We augment this with a process that allows an observer to watch the learner and suggest advice based on the learner's behavior.

circumstances and expect it to learn this advice from direct experience, while not forgetting what it previously learned.

Recognition of the value of advice-taking has a long history in AI. The general idea of a program accepting advice was first proposed nearly 40 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. Recently, after a decade-long lull, there has been a growing amount of research on advice-taking (Gordon & Subramanian, 1994; Huffman & Laird, 1993; Maclin & Shavlik, 1994; Noelle & Cottrell, 1994). For example, Gordon and Subramanian (1994) created a system that deductively compiles high-level advice into concrete actions, which are then refined using genetic algorithms.

Several characteristics of our approach to providing advice are particularly interesting. One, we allow the advisor to provide instruction in a quasi-natural language using terms about the specific task domain; the advisor does not have to be aware of the internal representations and algorithms used by the learner in order to provide useful advice. Two, the advice need not be precisely specified; vague terms such as “big,” “near,” and “old” are acceptable. Three, the learner does not follow the advice blindly; rather, the learner judges the usefulness of the advice and is capable of altering the advice based on subsequent experience.

In Section 2 we present a framework for using advice with reinforcement learners, and in Section 3 we outline an implemented system that instantiates this framework. The fourth section describes experiments that investigate the value of our approach. Finally, we discuss possible extensions to our research, relate our work to other research, and present some conclusions.

## 2. A General Framework for Advice-Taking

In this section we present our design for a reinforcement learning (RL) advice-taker, following the five-step framework for advice-taking developed by Hayes-Roth, Klahr, and Mostow (1981). In Section 3 we present specific details of our implemented system, named RATLE, which concretizes the design described below.

**Step 1. Request/receive 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 needs help (Clouse & Utgoff, 1992; Whitehead, 1991). Rather than having the learner request advice, we allow the external observer to provide advice whenever the observer feels it is appropriate. There are two reasons to allow the observer to determine when advice is needed: (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 general advice. Other RL methods (Clouse & Utgoff, 1992; Whitehead, 1991) focus on having the observer provide information about the action to take in a specific state. However, this can require a lot of interaction between the human advisor and computer learner, and also means that the learner must induce the generality of the advice.

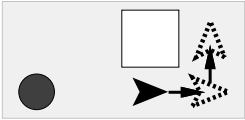
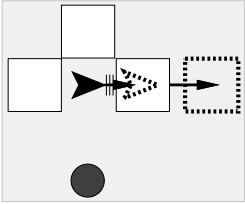
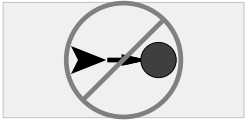
**Step 2. Convert the advice into an internal representation.** Once the observer has created a piece of advice, the agent must try to understand the advice. Due to the complexities of natural language processing, we require that the external observer express its advice using a simple programming language and a list of task-specific terms. We then parse the advice, using traditional methods from the programming-languages literature (Levine, Mason, & Brown, 1992).

Table 1 shows some sample advice that the observer could provide to an agent learning to play a video game. The left column contains the advice as expressed in our programming language, the center column shows the advice in English, and the right column illustrates the advice. (In Section 3 we use these samples to illustrate our algorithm for integrating advice.)

**Step 3. Convert the advice into a usable form.** After the advice has been parsed, the system transforms the general advice into terms that it can directly understand. Using techniques from *knowledge compilation* (Dietterich, 1991), a learner can convert (“operationalize”) high-level advice into a (usually larger) collection of directly interpretable statements (Gordon & Subramanian, 1994; Kaelbling & Rosenschein, 1990; Nilsson, 1994). We only address a limited form of operationalization, namely the concretization of imprecise terms such as “near” and “many.” Terms such as these allow the advice-giver to provide natural, yet partially vague, instructions, and eliminate the need for the advisor to fully understand the learner’s sensors.

**Step 4. Integrate the reformulated advice into the agent’s knowledge base.** In this work we employ a connectionist approach to RL (Anderson, 1987; Barto, Sutton, & Anderson, 1983; Lin, 1992). Hence, to incorporate the observer’s advice, the agent’s neural network must be updated. We use ideas from *knowledge-based neural networks* (Fu, 1989; Omlin & Giles, 1992; Shavlik & Towell, 1989) to directly install the advice into the agent. In one approach to knowledge-based neural networks, KBANN (Towell, Shavlik, & Noordewier, 1990; Towell & Shavlik, 1994), a set of propositional rules is 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

Table 1. Samples of advice in our advice language (left column).

Advice	English Version	Pictorial Version
<pre> IF An Enemy IS (Near <math>\wedge</math> West) <math>\wedge</math>   An Obstacle IS (Near <math>\wedge</math> North) THEN   MULTIACTION     MoveEast     MoveNorth   END END;</pre>	<p>If an enemy is near and west and an obstacle is near and north, hide behind the obstacle.</p>	
<pre> WHEN Surrounded <math>\wedge</math>   OKtoPushEast <math>\wedge</math>   An Enemy IS Near REPEAT   MULTIACTION     PushEast     MoveEast   END UNTIL <math>\neg</math> OKtoPushEast <math>\vee</math>   <math>\neg</math> Surrounded END;</pre>	<p>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.</p>	
<pre> IF An Enemy IS (Near <math>\wedge</math> East) THEN   DO_NOT MoveEast END;</pre>	<p>Do not move toward a nearby enemy.</p>	

(for details, see Section 3). We extend the KBANN method to accommodate our advice-giving language.

Figure 2 illustrates our basic approach for adding advice into the reinforcement learner's action-choosing network. This network computes a function from sensations to the utility of actions. Incorporating advice involves adding to the existing neural network new hidden units that represent the advice.

**Step 5. Judge the value of the advice.** The final step of the advice-taking process is to evaluate the advice. We view this process from two perspectives: (i) the learner's, who must decide if the advice is useful; and (ii) the advisor's, who must decide if the advice had the desired effect on the behavior of the learner. Our learner evaluates advice by continued operation in its environment; the feedback provided by the environment offers a crude measure of the advice's quality. (One can also envision that in some circumstances – such as a game-learner that can

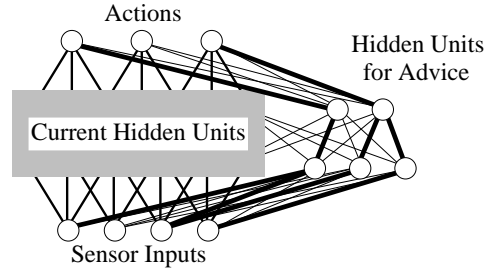


Figure 2. Adding advice to the RL agent's neural network by creating new hidden units that represent the advice. The thick links on the right capture the semantics of the advice. The added thin links initially have near-zero weight; during subsequent backpropagation training the magnitude of their weights can change, thereby refining the original advice. Details and an example appear in Section 3.

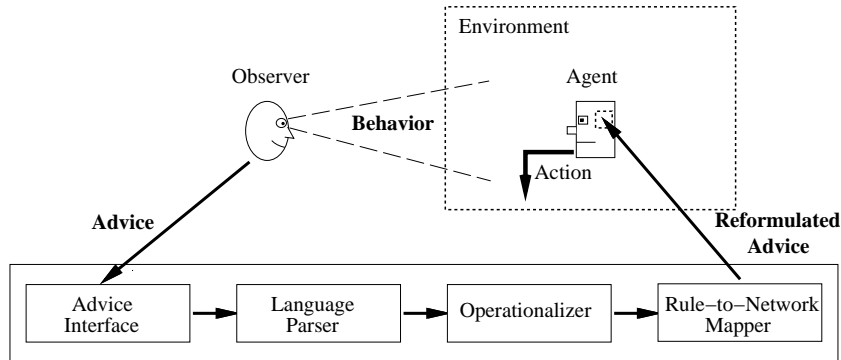


Figure 3. Interaction of the observer, agent, and our advice-taking system. The process is a cycle: the observer watches the agent's behavior to determine what advice to give, the advice-taking system processes the advice and inserts it into the agent, which changes the agent's behavior, thus possibly causing the observer to provide more advice. The agent operates as a normal Q-learning agent when not presented with advice.

play against itself (Tesauro, 1992) or when an agent builds an internal world model (Sutton, 1991) – it would be possible to quickly estimate whether the advice improves performance.) The advisor judges the value of his or her advice similarly (i.e., by watching the learner's post-advice behavior). This may lead to the advisor giving further advice – thereby restarting the advice-taking process.

### 3. The RATLE System

Figure 3 summarizes the approach we discussed in the previous section. We implemented the RATLE (**R**einforcement and **A**dvice-**T**aking **L**earning **E**nvironment)

system as a mechanism for evaluating this framework. In order to explain RATLE, we first review connectionist Q-learning (Sutton, 1988; Watkins, 1989), the form of reinforcement learning that we use in our implementation, and then KBANN (Towell & Shavlik, 1994), a technique for incorporating knowledge in the form of rules into a neural network. We then discuss our extensions to these techniques by showing how we implement each of the five steps described in the previous section.

### *Background – Connectionist Q-Learning*

In standard 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 uses a *utility function* that maps states and actions to a numeric value (the utility). 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 and the agent acts optimally afterwards. 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 typically starts out with a randomly chosen utility function and stochastically explores its environment. As the agent explores, it continually makes 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.

The main difference between our approach and standard connectionist Q-learning is that our agent continually checks for pending advice, and if so, incorporates that advice into its utility function. Table 2 shows the main loop of an agent employing connectionist Q-learning, augmented (in italics) by our process for using advice. The resulting composite system we refer to as RATLE.

### *Background – Knowledge-Based Neural Networks*

In order for us to make use of the advice provided by the observer, we must incorporate this advice into the agent's neural-network utility function. To do so, we extend the KBANN algorithm (Towell & Shavlik, 1994). 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 ac-

Table 2. Steps of the RATLE algorithm. Our additions to the standard connectionist Q-learning loop are Step 6 and the subroutine *IncorporateAdvice* (all shown in italics). We follow Lin's (1992) method exactly for action selection and Q-function updating (Steps 2 and 5). When estimating the performance of a network ("testing"), the action with the highest utility is chosen in Step 2 and no updating is done in Step 5.

Agent's Main Loop	<i>Incorporate Advice</i>
1. Read sensors.	6a. <i>Parse advice.</i>
2. Stochastically choose an action, where the probability of selecting an action is proportional to the log of its predicted utility (i.e., its current Q value). Retain the predicted utility of the action selected.	6b. <i>Operationalize any fuzzy terms.</i>
3. Perform selected action.	6c. <i>Translate advice into network components.</i>
4. Measure reinforcement, if any.	6d. <i>Insert translated advice directly into RL agent's neural-network based utility function.</i>
5. Update utility function – use the current state, the current Q-function, and the actual reinforcement to obtain a new estimate of the expected utility; use the difference between the new estimate of utility and the previous estimate as the error signal to propagate through the neural network.	6e. <i>Return.</i>
6. <i>Advice pending? If so, call IncorporateAdvice.</i>	
7. Go to 1.	

tive (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 4 shows an example of this process for a set of simple propositional rules.

In RATLE, we use an imperative programming language, instead of propositional rules, to specify advice. In order to map this more complex language, we make use of hidden units that record state information. These units are recurrent and record the activation of a hidden unit from the previous activation of the network (i.e., they "remember" the previous activation value). We discuss how these units are used below.

#### *Implementing the Five-Step Framework*

In the remainder of this section we describe how we implemented the advice-taking strategy presented in the last section. Several worked examples are included.

**Step 1. Request/receive the advice.** To give advice, the observer simply interrupts the agent's execution and types his or her advice. Advice must be expressed in the language defined by the grammar in Appendix B.

**Step 2. Convert the advice into an internal representation.** We built RATLE's advice parser using the standard Unix compiler tools *lex* and *yacc* (Levine et al., 1992).

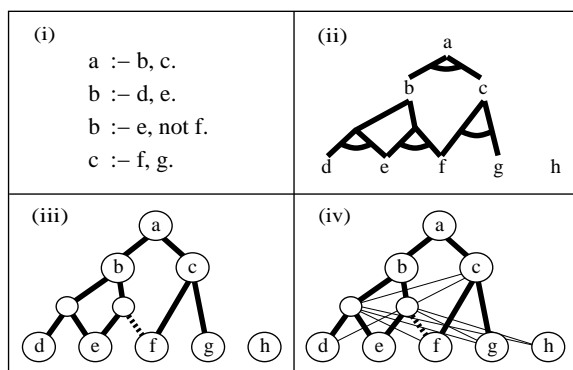


Figure 4. Sample of the KBANN algorithm: (i) a propositional rule set in Prolog notation; (ii) the rules viewed as an AND-OR dependency graph; (iii) each proposition is represented as a unit (extra units are also added to represent disjunctive definitions, e.g.,  $b$ ), and their weights and biases are set so that they implement AND or OR gates, e.g., the weights  $b \rightarrow a$  and  $c \rightarrow a$  are set to 4 and  $a$ 's bias (threshold) to 6 (the bias of an OR node is 2); (iv) low-weighted links are added between layers as a basis for future learning (e.g., an antecedent can be added to a rule by increasing one of these weights).

Our advice-taking language has two main programming constructs: IF-THEN rules and loops (both WHILE and REPEAT). The loop constructs also have optional forms that allow the teacher to specify more complex loops (e.g., the REPEAT may have an entry condition). Each of these constructs may specify either a single action or, via the MULTIACTION construct, a “plan” containing a sequence of actions. The observer may also specify that an action should *not* be taken as a consequent (as opposed to specifying an action to take). Examples of advice in our language appear in Table 1 and in Appendix A.

The IF-THEN constructs actually serve two purposes. An IF-THEN can be used to specify that a particular action should be taken in a particular situation. It 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 instead is the keyword INFER followed by the name of the new intermediate term. This allows the observer to create descriptive terms based on the sensed features. For example, the advisor may want to define an intermediate term *NotLarge* that is true if an object is *Small* or *Medium*, and then use the derived term *NotLarge* in subsequent advice.

In order to specify the preconditions of the IF-THEN and looping constructs, the advisor lists logical combinations of conditions (basic “sensors” and any derived features). To make the language easier to use, we also allow the observer to state “fuzzy” conditions (Zadeh, 1965), which we believe provide a natural way to articulate imprecise advice.

**Step 3. Convert the advice into a usable form.** As will be seen in Step 4, most of the concepts expressible in our grammar can be directly translated into a neural network. The fuzzy conditions, however, require some pre-processing. We



must first “operationalize” them by using the traditional methods of fuzzy logic to create an explicit mathematical expression that determines the fuzzy “truth value” of the condition as a function of the sensor values. We accomplish this representation by applying the method of Berenji and Khedkhar (1992), adapted slightly (Maclin, 1995) to be consistent with KBANN’s mapping algorithm.

Though fuzzy logic is a powerful method that allows humans to express advice using intuitive terms, it has the disadvantage that someone must explicitly define the fuzzy terms in advance. However, the definitions need not be perfectly correct, since we insert our fuzzy conditions into the agent’s neural network and, thus, allow their definitions to be adjusted during subsequent training.

At present, RATLE only accepts fuzzy terms of the form:

*quantifier object IS/ARE descriptor*

where the quantifier is a fuzzy term specifying number (e.g., A, No, Few, Many), the object is the type of object being sensed (e.g., Blocks, Trees, Enemies) and the descriptor is a property of the referenced objects (e.g., Near, Big). For example, a fuzzy condition could be “Many Trees ARE Near.”

Currently we use only sigmoidal membership functions. To operationalize a fuzzy condition, RATLE determines a set of weights and a threshold that implement the given sigmoidal membership function, as a function of the current sensor readings. The exact details depend on the structure of a given domain’s sensors (see Maclin, 1995, for additional details) and have not been a major focus of this research. The result of this process essentially defines a perceptron; hence, operationalized fuzzy conditions can be directly inserted into the agent’s neural network during Step 4.

**Step 4. Integrate the reformulated advice into the agent’s knowledge base.** After RATLE operationalizes any fuzzy conditions, it proceeds to insert all of the advice into the agent’s current neural-network utility function. To do this, we made five extensions to the standard KBANN algorithm: (i) advice can contain multi-step plans, (ii) it can contain loops, (iii) it can refer to previously defined terms, (iv) it may suggest actions to *not* take, and (v) it can involve fuzzy conditions (discussed above). We achieve each of these extensions by following the general approach illustrated earlier in Figure 2.

Consider, as an example of a multi-step plan, the first entry in Table 1. Figure 5 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 near 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<sub>-1</sub>*. This *recurrent* unit becomes active (“true”) when *State1* was active for the previous input (we need recurrent units to implement multi-step plans). Finally, it constructs a unit (labeled *B*) that is active when *State1<sub>-1</sub>* is true and the previous action was an eastward move (the

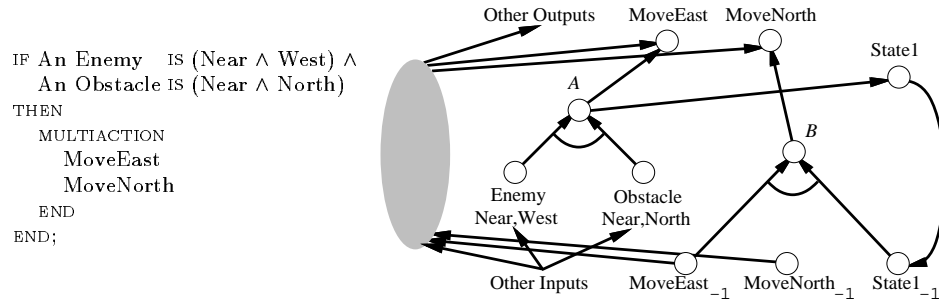


Figure 5. On the left is the first piece of advice from Table 1. On the right is RATLE's translation of this piece of advice. The shaded ellipse represents the original hidden units. Arcs show units and weights that are set to implement a conjunction. RATLE also adds zero-weighted links (not shown here – see Figure 4d) between the new units and other parts of the current network; these links support subsequent refinement.

network's input vector records the previous action taken in addition to the current sensor values). When active, unit *B* suggests moving north – the second step of the plan. (In general, RATLE represents plans of length  $N$  using  $N - 1$  state units.)

RATLE assigns a high weight<sup>1</sup> 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 suggested 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 1 also contains a multi-step plan, but this time it is embedded in a REPEAT. Figure 6 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 if the WHEN condition is true (unit *C*) and also if 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 if the UNTIL is unsatisfied and the agent's two previous actions were pushing and then moving east.

A third 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. There are two types of new definitions: (i) new preconditions of actions, and (ii) new definitions for derived features. We process the two types differently, since the former involve real-valued outputs while the latter are essentially Boolean-valued.

For new preconditions of *actions*, RATLE adds a highly weighted link from the unit representing the definition to the output unit representing the action. This is done so that in the situations where the advice is applicable, the utility of the action will then be higher that it would otherwise be. When the advisor provides a new definition of a derived feature, RATLE operates as shown in Figure 7. It first creates a new hidden unit that represents the new definition, then makes an OR

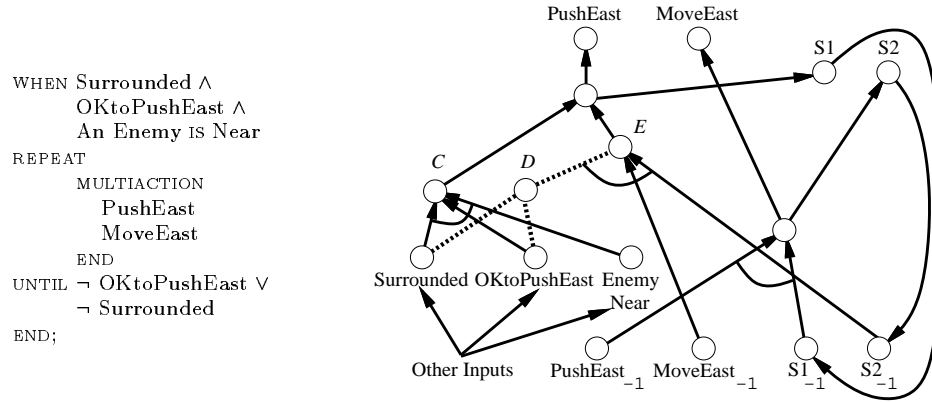


Figure 6. On the left is the second piece of advice from Table 1. On the right is RATLE’s translation of it. Dotted lines indicate negative weights. These new units are added to the existing network (not shown).

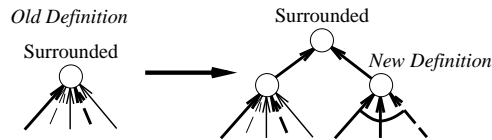


Figure 7. Incorporating the definition of a term that already exists.

node that combines the old and new definitions. This process is analogous to how KBANN processes multiple rules with the same consequent.

A fourth issue is how to deal with advice that suggests *not* doing an action. This is straightforward in our approach, since we connect hidden units to “action” units with a highly weighted link. For example, for the third piece of advice shown in Table 1, RATLE would create a unit representing the fuzzy condition “An Enemy IS (Near and East)” and then connect the resulting unit to the action MoveEast with a negatively weighted link. This would have the effect of lowering MoveEast’s utility when the condition is satisfied (which is the effect we desire). This technique avoids the question of what to do when one piece of advice suggests an action and another prohibits that action. Currently the conflicting pieces of advice (unless refined) cancel each other, but this simple approach may not always be satisfactory.

Maclin (1995) fully describes how each of the constructs in RATLE’s advice language is mapped into a neural-network fragment.

## 4. Experimental Study

We next empirically judge the value of using RATLE to provide advice to an RL agent.

### 4.1. Testbed

Figure 8a illustrates the Pengo task. We chose Pengo because it has been previously explored in the AI literature (Agre & Chapman, 1987; Lin, 1992). The agent in Pengo 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. A moving obstacle will destroy the food and enemies it hits, and will continue to slide until it encounters another obstacle or the edge of the board. When the obstacle is unable to move (because there is an obstacle or wall behind

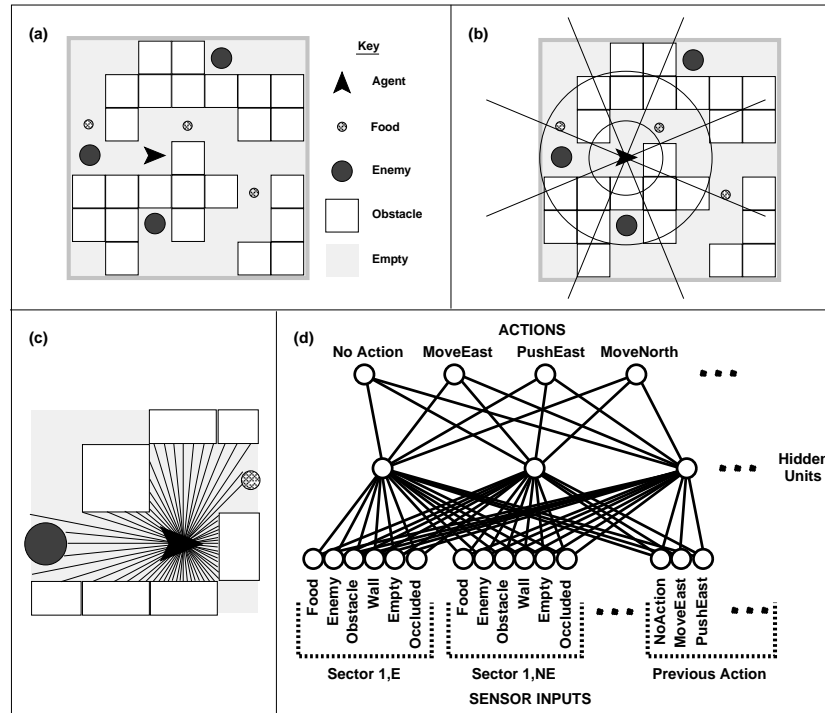


Figure 8. Our sample test environment: (a) sample configuration; (b) sample division of the environment into sectors; (c) distances to the nearest occluding object along a fixed set of arcs (measured from the agent); (d) a neural network that computes the utility of actions.

it), the obstacle disintegrates. Food is collected when touched by the agent or an enemy.

Each enemy follows a fixed policy. It 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 after a random interval), but the agent is constrained to remain on the board. Enemies do not push obstacles.

The initial mazes are generated randomly using a maze-creation program (Maclin, 1995) 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 enemies and food items. The agent, enemies, and food are randomly deposited on the board, with the caveat that the enemies are required to be initially at least a fixed distance 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 food objects ( $+0.7$ ), or (iii) the agent destroys an enemy by pushing an obstacle into it ( $+0.9$ ).

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 8b). 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 – food, 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 8c). This means that the agent is only able to represent the objects in direct line-of-sight from the agent (for example, the enemy to the south of the agent is out of sight). The percentage of each object type in a sector is just the number of sensing arcs that end in that sector by hitting an object of the given type, divided by the maximum number of arcs that could end in the sector. So for example, given Figure 8b, the agent’s percentage for “obstacle” would be high for the sector to the east. The agent also calculates how much of each sector is empty and how much is occluded. These percentages constitute the input to the neural network (see Figure 8d). Note that the agent also receives as input, using a 1-of- $N$  encoding, the action the agent took in the previous state.<sup>2</sup>

## 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. We report our results by training episodes rather than number of training actions because we believe episodes are a more useful measure of “meaningful” training done – an agent having collected all of the food and eliminated all of the enemies could spend a large amount of time in useless wandering (while receiving no reinforcements), thus counting actions might penalize such an agent since it gets to

experience fewer reinforcement situations. In any case, for all of our results the results appear qualitatively similar when graphed by the number of training actions (i.e., the agents all take a similar number of actions per episode during training).

Each of our environments contains a 7x7 grid with approximately 15 obstacles, 3 enemies, and 10 food items. 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 future average total reinforcement (the average sum of the reinforcements received by the agent)<sup>3</sup> by “freezing” the network and measuring the average reinforcement on a testset of 100 randomly generated environments; the same testset is used for all our experiments.

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. We also experimented with giving this system recurrent units (as in the units RATLE adds for multi-step and loop plans), but these units did not lead to improved performance for the baseline system, and, hence, the baseline results are for a system without recurrent links. However, recall that the input vector records the last action taken.

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 A), two that use multi-step plans (referred to as *ElimEnemies* and *Surrounded*), and two that do not (*SimpleMoves* and *NonLocalMoves*).

### 4.3. Results

In our first experiment, we evaluate the hypothesis that our approach can in fact take advantage of advice. After 1000 episodes of initial learning, we judge the value of (independently) providing each of the four sets of advice to our agent using RATLE. We train the agent for 2000 more episodes after giving the advice, then measure its average cumulative reinforcement on the testset. (The baseline is also trained for 3000 episodes). Table 3 reports the averaged testset reinforcement; all gains over the baseline system are statistically significant<sup>4</sup>. 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; however, the multi-step advice may simply be less useful.

Each of our pieces of advice to the agent addresses specific subtasks: collecting food (*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 3. Testset results for the baseline and the four different types of advice. Each of the four gains over the baseline is statistically significant.

Advice Added	Average Total Reinforcement on the Testset
None (baseline)	1.32
<i>SimpleMoves</i>	1.91
<i>NonLocalMoves</i>	2.01
<i>ElimEnemies</i>	1.87
<i>Surrounded</i>	1.72

Table 4. Mean number of enemies captured, food collected, and number of actions taken (survival time) for the experiments summarized in Table 3.

Advice Added	Enemies Captured	Food Collected	Survival Time
None (baseline)	0.15	3.09	32.7
<i>SimpleMoves</i>	0.31	3.74	40.8
<i>NonLocalMoves</i>	0.26	3.95	39.1
<i>ElimEnemies</i>	0.44	3.50	38.3
<i>Surrounded</i>	0.30	3.48	46.2

Table 4 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.

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 9 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.

Our third experiment investigates the hypothesis that subsequent advice will lead to further gains in performance. To test this hypothesis, we supplied each of our four pieces of advice to an agent after 1000 episodes (as in our first experiment), supplied one of the remaining three pieces of advice after another 1000 episodes, and then trained the resulting agent for 2000 more episodes. These results are averaged over 60 neural networks instead of the 30 networks used in the other experiments in order to obtain statistically significant results. Table 5 shows the results for this test.

In all cases, adding a second piece of advice leads to improved performance. However, the resulting gains when adding the second piece of advice are not as large as the original gains over the baseline system. We suspect this occurs due to a combination of factors: (i) there is an upper limit to how well the agents can do – though it is difficult to quantify; (ii) the pieces of advice interact – they may suggest different actions in different situations, and in the process of resolving these conflicts,

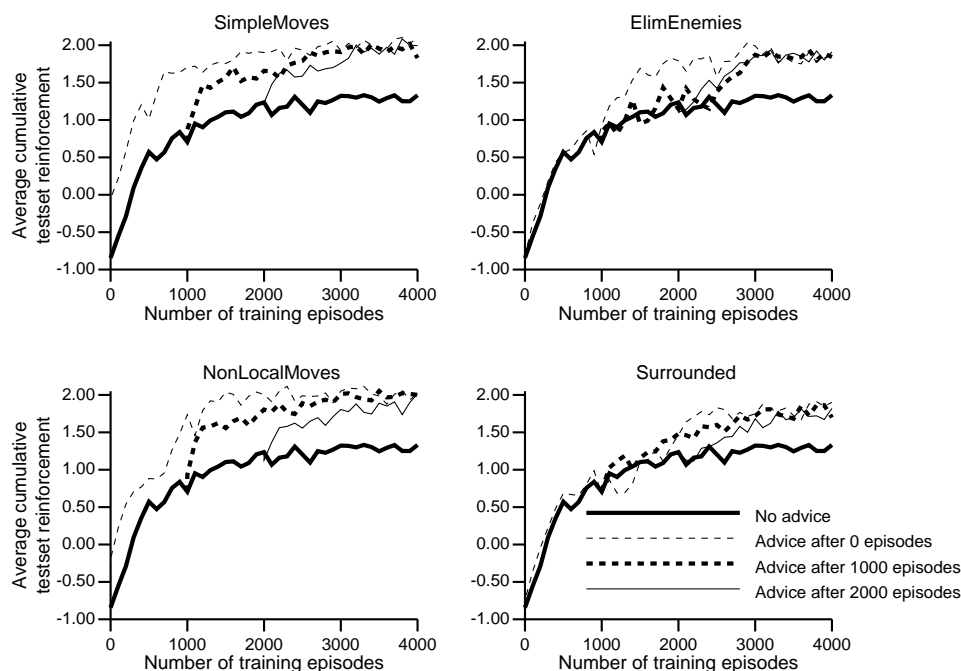


Figure 9. Average total reinforcement for our four sample pieces of advice as a function of amount of training and point of insertion of the advice.

Table 5. Average testset reinforcement for each of the possible pairs of our four sets of advice. The first piece of advice is added after 1000 episodes, the second piece of advice after an additional 1000 episodes, and then trained for 2000 more episodes (total of 4000 training episodes). Shown in parentheses next to the first pieces of advice are the performance results from our first experiment where only a single piece of advice was added. All of the resulting agents show statistically significant gains in performance over the agent with just the first piece of advice.

First Piece of Advice	Second Piece of Advice			
	<i>SimpleMoves</i>	<i>NonLocalMoves</i>	<i>ElimEnemies</i>	<i>Surrounded</i>
<i>SimpleMoves</i> (1.91)	-	2.17	2.10	2.05
<i>NonLocalMoves</i> (2.01)	2.27	-	2.18	2.13
<i>ElimEnemies</i> (1.87)	2.01	2.26	-	2.06
<i>Surrounded</i> (1.72)	2.04	2.11	1.95	-

the agent may use one piece of advice less often; and (iii) the advice pieces are related, so that one piece may cover situations that the other already covers. Also interesting to note is that the order of presentation affects the level of performance achieved in some cases (e.g., presenting *NonLocalMoves* followed by *SimpleMoves* achieves higher performance than *SimpleMoves* followed by *NonLocalMoves*).



Table 6. Average total reinforcement results for the advice *NonLocalMoves* using two forms of replay. The advice is inserted and then the network is trained for 1000 episodes. Replay results are the *best* results achieved on 1000 episodes of training (occurring at 600 episodes for Action Replay and 500 episodes for Sequence Replay). Results for the RATLE approach without replay are also shown; these results are for 1000 training episodes.

Training Method	Average Total Testset Reinforcement
Standard RATLE (no replay)	1.74
Action-Replay Method	1.48
Sequence-Replay Method	1.45

In our fourth experiment we evaluate the usefulness of combining our advice-giving approach with Lin’s “replay” technique (1992). Lin introduced the replay method to make use of “good” sequences of actions provided by a teacher. In replay, the agent trains on the teacher-provided sequences frequently to bias its utility function towards these good actions. Thrun (1994, personal communication) reports that replay can in fact be useful even when the remembered sequences are not teacher-provided sequences – in effect, by training multiple times on each state-action pair the agent is “leveraging” more value out of each example. Hence, our experiment addresses two related questions: (i) does the advice provide any benefit over simply reusing the agent’s experiences multiple times?, and (ii) can our approach benefit from replay, for example, by needing fewer training episodes to achieve a given level of performance? Our hypothesis was that the answer to both questions is “yes.”

To test our hypothesis we implemented two approaches to replay in RATLE and evaluated them using the *NonLocalMoves* advice. In one approach, which we will call *Action Replay*, we simply keep the last  $N$  state-action pairs that the agent encountered, and on each step train with all of the saved state-action pairs in a randomized order. A second approach (similar to Lin’s), which we will call *Sequence Replay*, is more complicated. Here, we keep the last  $N$  *sequences* of actions that ended when the agent received a non-zero reinforcement. Once a sequence completes, we train on all of the saved sequences, again, in a randomized order. To train the network with a sequence, we first train the network on the state where the reinforcement was received, then the state one step before that state, then two steps before that state, etc., on the theory that the states nearest reinforcements best estimate the actual utility of the state. Results for keeping 250 state-action pairs and 250 sequences<sup>5</sup> appear in Table 6; due to time constraints we trained these agents for only 1000 episodes.

Surprisingly, replay did not help our approach. After examining networks during replay training, we hypothesize this occurred because we are using a single network to predict all of the Q values for a state. During training, to determine a target vector for the network, we first calculate the new Q value for the action the agent actually performed. We then activate the network, and set the target output vector for the network to be equal to the actual output vector, except that we use the new

Table 7. Average testset reinforcement using the strawman approach to using advice compared to the RATLE method.

Advice	STRAWMAN	RATLE
<i>SimpleMoves</i>	1.63	1.91
<i>NonLocalMoves</i>	1.46	2.01
<i>ElimEnemies</i>	1.28	1.87
<i>Surrounded</i>	1.21	1.72

prediction for the action taken. For example, assume the agent takes action two (of three actions) and calculates that the Q value for action two should be 0.7. To create a target vector the agent activates the network with the state (assume that the resulting output vector is  $[0.4, 0.5, 0.3]$ ), and then creates a target vector that is the same as the output vector except for the new Q value for the action taken (i.e.,  $[0.4, \mathbf{0.7}, 0.3]$ ). This causes the network to have error at only one output unit (the one associated with the action taken). For replay this is a problem because we will be activating the network for a state a number of times, but only trying to correctly predict one output unit (the other outputs are essentially allowed to take on any value), and since the output units share hidden units, changes made to predict one output unit may affect others. If we repeat this training a number of times, the Q values for other actions in a state may become greatly distorted. Also, if there is unpredictability in the outcomes of actions, it is important to average over the different results; replay focuses on a single outcome. One possible solution to this problem is to use separate networks for each action, but this means the actions will not be able to share concepts learned at the hidden units. We plan to further investigate this topic, since replay intuitively seems to be a valuable technique for reducing the amount of experimentation an RL agent has to perform.

Our final experiment investigates a naive approach for using advice. This simple strawman algorithm follows the observer’s advice when it applies; otherwise it uses a “traditional” connectionist Q-learner to choose its actions. We use this strawman to evaluate if it is important that the agent refine the advice it receives. When measured on the testset, the strawman employs a loop similar to that shown in Table 2. One difference for the strawman’s algorithm is that Step 6 in Table 2’s algorithm is left out. The other difference is that Step 2 (selecting an action) is replaced by the following:

Evaluate the advice to see if it suggests any actions:  
 If any actions are suggested, choose one randomly,  
 Else choose the action that the network predicts has maximum utility.

The performance of this strawman is reported in Table 7. In all cases, RATLE performs better than the strawman; all of these reinforcement gains are statistically significant. In fact, in two of the cases, *ElimEnemies* and *Surrounded*, the resulting method for selecting actions is actually worse than simply using the baseline network (whose average performance is 1.32).

#### 4.4. Discussion

Our experiments demonstrate that: (i) advice can improve the performance of an RL agent; (ii) advice produces the same resulting performance no matter when it is added; (iii) a second piece of advice can produce further gains; and (iv) it is important for the agent to be able to refine the advice it receives. In other experiments (not reported here), we demonstrate that an agent can quickly overcome the effects of “bad” advice (Maclin, 1995). We corroborated our Pengo results using a second testbed (Maclin, 1995). A significant feature of our second testbed is that its agent’s sensors record the complete state of the environment. Thus, the results in our second testbed support the claim that the value of advice in the Pengo testbed is not due solely to the fact that the teacher sees the complete state, while the learner only has line-of-sight sensors (and, hence, is trying to learn a *partially observable* Markov decision process; Monahan, 1982).

One key question arises from our Pengo results: will the baseline system eventually achieve the same level of performance that the advice-taking system achieves? After all, Q-learning converges to the optimal Q function when a Q table is used to represent the function (Watkins & Dayan, 1992). However, a backpropagation-trained network may only converge to a *local* minimum in the weight space defining the Q function. To further answer the performance-in-the-limit question, we will address a more general one – what effect do we expect advice to have on the agent?

When we introduce “good” advice into an agent, we expect it to have one or more of several possible effects. One possible effect of advice is that the advice will change the network’s predictions of some of the Q values to values that are closer to the desired “optimal” values. By reducing the overall error the agent may be able to converge more quickly towards the optimal Q function. A second related effect is that by increasing (and decreasing) certain Q values the advice changes which states are explored by the agent. Here, good advice would cause the agent to explore states that are useful in finding the optimal plan (or ignoring states that are detrimental). Focusing on the states that are important to the optimal solution may lead to the agent converging more quickly to a solution. A third possible effect is that the addition of advice alters the weight space of possible solutions that the learner is exploring. This is because the new weights and hidden units change the set of parameters that the learner was exploring. For example, the advice may construct an intermediate term (represented by a hidden unit) with very large weights, that could not have been found by gradient-descent search. In the resulting altered weight space the learner may be able to explore functions that were unreachable before the advice is added (and these functions may be closer to the optimal).

Given these possible effects of good advice, we can conclude that advice can both cause the agent to converge more quickly to a solution, and that advice may cause the agent to find a better solution than it may have otherwise found. For our experiments, we see the effect of speeded convergence in the graphs of Figure 9, where the advice, generally after a small amount of training, leads the agent to

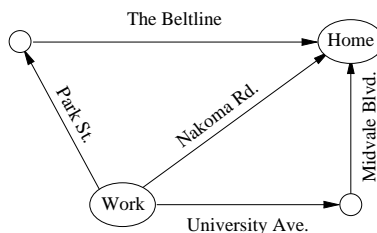


Figure 10. A sample problem where good advice can fail to enhance performance. Assume the goal of the agent is to go from Work to Home, and that the agent will receive a large reward for taking Nakoma Road, and a slightly smaller reward for taking University Avenue followed by Midvale Boulevard. If the agent receives advice that University followed by Midvale is a good plan, the agent, when confronted with the problem of going from Work to Home will likely follow this plan (even during training, since actions are selected proportional to their predicted utility). Thus it may take a long time for the learner to try Nakoma Road often enough to learn that it is a better route. A learner without advice might try both routes equally often and quickly learn the correct utility value for each route.

achieve a high level of performance quickly. These graphs also demonstrate the effect of convergence to a better solution, at least given our fixed amount of training. Note that these effects are a result of what we would call “good” advice. It is possible that “bad” advice could have equally deleterious effects. So, a related question is how do we determine whether advice is “good” or not?

Unfortunately, determining the “goodness” of advice appears to be a fairly tricky problem, since even apparently useful advice can lead to poor performance in certain cases. Consider for example, the simple problem shown in Figure 10. This example demonstrates a case where advice, though providing useful information, could actually cause the agent to take longer to converge to an optimal policy. Basically, the good advice “masks” an even better policy. This example suggests that we may want to rethink the stochastic mechanism that we use to select actions. In any case, it appears that defining the properties of “good” advice is a challenging topic for future work. As a first (and admittedly vague) approximation we would expect advice to be “good” when it causes one of the effects mentioned above: (i) it reduces the overall error in the agent’s predicted Q values, (ii) it causes the agent to pick actions that lead to states that are important in finding a solution, or (iii) it transforms the network so that the agent is able to perform gradient descent to a better solution.

## 5. Future Work

Based on our initial experience, we intend to expand our approach in a number of directions. One important future topic is to evaluate our approach in other domains. In particular, we intend to explore tasks involving multiple agents working in cooperation. Such a domain would be interesting in that an observer could give

advice on how a “group” of agents could solve a task. Another domain of interest is software agents (Riecken, 1994). For example, a human could advise a software agent that looks for “interesting” papers on the World-Wide Web.

We see algorithmic extensions as fitting into the three categories explained below.

### *Broadening the Advice Language*

Our experience with RATLE has led us to consider a number of extensions to our current programming language. Examples include:

- **Prefer action** – let the teacher indicate that one action is “preferred” in a state. Here the advisor would only be helping the learner sort through its options, rather than specifically saying what should be done.
- **Forget advice** – permit the advisor to retract previous advice.
- **Add/Subtract condition from advice** – allow the advisor to add or remove conditions from previous rules, thereby fine-tuning advice.
- **Reward/Punish state** – let the teacher specify “internal” rewards that the agent is to receive in certain states. This type of advice could be used to give the agent a set of internal goals.
- **Remember condition** – permit the advisor to indicate propositions that the learner should remember (e.g., the location of some important site, like a good place to hide, so that it can get back there again). We would implement this using recurrent units that record state information. This remembered information could then be used in future advice.

We also plan to explore mechanisms for specifying multi-user advice when we explore domains with multiple agents.

### *Improving the Algorithmic Details*

At present, our algorithm adds hidden units to the learner’s neural network whenever advice is received. Hence, the network’s size grows monotonically. Although recent evidence (Weigend, 1993) suggests overly large networks are not a problem given that one uses proper training techniques, we plan to evaluate techniques for periodically “cleaning up” and shrinking the learner’s network. We plan to use standard neural-network techniques for removing network links with low saliency (e.g., Le Cun, Denker, & Solla, 1990).

In our current implementation, a plan (i.e., a sequence of actions) can be interrupted if another action has higher utility (see Figure 5). Recall that the advice only increases the utility of the actions in the plan, and that the learner can choose to execute another action if it has higher utility. Once a plan is interrupted, it

cannot be resumed from the point of interruption because the necessary state unit is not active. We intend to evaluate methods that allow plans to be temporarily interrupted; once the higher-utility tasks complete, the interrupted plan will resume. We anticipate that this will involve the use of exponentially decaying state units that record that a plan was being executed recently.

The networks we use in our current implementation have numeric-valued output units (recall that they represent the expected utility of actions). Hence, we need to more thoroughly investigate the setting of the weights between the Boolean-valued advice nodes and the numeric-valued utility nodes, a topic not relevant to the original work with KBANN, since that research only involved Boolean concepts. Currently, advice simply increases the utility of the recommended actions by a fixed amount. Although subsequent training can alter this initial setting, we plan to more intelligently perform this initial setting. For example, we could reset the weights of the network so that the suggested action always has the highest utility in the specified states. This approach would guarantee that the suggested action will have the highest utility, but can be faulty if the action is already considered the best of several bad choices. In this case the alternate approach would simply leave the current network unchanged, since the advised action is already preferred. But the teacher may be saying that the action is not only the best choice, but that the utility of the action is high (i.e., the action is “good” in some sense). Therefore, simply requiring that the action be the “best” choice may not always capture the teacher’s intentions. This approach also requires that RATLE find an appropriate set of weights to insure that the suggested action be selected first (possibly by solving a non-linear program).

In another form of reinforcement learning, the agent predicts the utility of a state rather than the utility of an action in a state (Sutton, 1988); here the learner has a model of how its actions change the world, and determines the action to take by checking the utility of the states that are reachable from the current state. Applying RATLE to this type of reinforcement-learning system would be difficult, since RATLE statements suggest actions to take. In order to map a statement indicating an action, RATLE would first have to determine the set of states that meet the condition of the statement, then calculate the set of states that would result by following the suggested action. RATLE would then increase the utility of the states that follow from the suggested action. Other types of advice would be more straightforward under this approach. For example, if the teacher gave advice about a goal the agent should try to achieve (i.e., as in Gordon and Subramanian’s, 1994, approach), RATLE could determine the set of states corresponding to the goal and simply increase the utility of all of these states.

Finally, our system maintains no statistics that record how often a piece of advice was applicable and how often it was followed. We intend to add such statistics-gatherers and use them to inform the advisor that a given piece of advice was seldom applicable or followed. We also plan to keep a record of the *original* advice and compare its statistics to the *refined* version. Significant differences between the two should cause the learner to inform its advisor that some advice has substantially

changed (we plan to use the rule-extraction techniques described below when we present the refined advice to the advisor).

### *Converting Refined Advice into a Human-Comprehensible Form*

One interesting area of future research is the “extraction” (i.e., conversion to a easily comprehensible form) of learned knowledge from our connectionist utility function. We plan to extend previous work on rule extraction (Craven & Shavlik, 1994; Towell & Shavlik, 1993) to produce rules in RATLE’s language. We also plan to investigate the use of rule extraction as a mechanism for transferring learned knowledge between RL agents operating in the same or similar environments.

## **6. Related Work**

Our work relates to a number of recent research efforts. This related work can be roughly divided into five 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, (iv) creating knowledge-based neural networks, and (v) refining prior domain theories.

### *Providing advice to a problem solver*

An early example of a system that makes use of advice is Mostow’s (1982) `FOO` system, which operationalizes general advice 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 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 reinforcement learning to refine and evaluate the advice.

More recently, Laird, Hucka, Yager, and Tuck (1990) created an advice-taking system called `ROBO-SOAR`. In this system, an observer can provide advice whenever the system is at an impasse by suggesting which operators to explore in an attempt to resolve the impasse. As with `FOO`, the advice presented is used to guide the learner’s reasoning process, while in `RATLE` we directly incorporate the advice into the learner’s knowledge base and then refine that knowledge using subsequent experience. Huffman and Laird (1993) developed the `INSTRUCTO-SOAR` system that allows an agent to interpret simple imperative statements such as “Pick up the red block.” `INSTRUCTO-SOAR` examines these instructions in the context of its current problem solving, and uses `SOAR`’s form of explanation-based learning to generalize the instruction into a rule that can be used in similar situations. `RATLE`

differs from INSTRUCTO-SOAR 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 that uses advice expressed as 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 training examples.

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 requires advice, but is limited in that it may require advice more often than the observer is willing to provide it. In RATLE the advisor provides advice whenever he or she feels 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. However, our system does not have the operationalization capability of Gordon and Subramanian's system.



*Developing robot-programming languages*

Many researchers have introduced languages for programming robot-like agents (Chapman, 1991; Kaelbling, 1987; Nilsson, 1994). These systems do not generally focus on programming agents that learn to refine their programs. Crangle and Suppes (1994) investigated how a robot can understand a human's instructions, expressed in ordinary English. However, they do not address correction, by the learner, of approximately correct advice.

*Incorporating advice into neural networks*

Noelle and Cottrell (1994) suggest an alternative approach to making use of advice in neural networks. One way their approach differs from ours is that their connectionist model itself performs the process of incorporating advice, which contrasts to our approach where we directly add new "knowledge-based" units to the neural network. Our approach leads to faster assimilation of advice, although theirs is arguably a better psychological model.

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 her system have not yet been empirically demonstrated.

Gruau (1994) developed a compiler that translates 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.

Diederich (1989) devised a method that accepts instructions in a symbolic 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.

Abu-Mostafa (1995) uses an approach similar to Diederich's to encode "hints" in a neural network. A hint is a piece of knowledge provided to the network that indicates some important general aspect for the network to have. For example, a hint might indicate to a network trying to assess people as credit risks that a "monotonicity" principle should hold (i.e., when one person is a good credit risk, then an identical person with a higher salary should also be a good risk). Abu-Mostafa uses these hints to generate examples that will cause the network to have this property, then mixes these examples in with the original training examples. As

with Diederich's work, our work differs from Abu-Mostafa's in that RATLE directly installs the advice into the network.

Suddarth and Holden (1991) investigated another form of "hint" for a neural network. In their approach, a hint is an extra output value for the neural network. For example, a neural network using sigmoidal activation units to try to learn the difficult XOR function might receive a hint in the form of the output value for the OR function. The OR function is useful as a hint because it is simple to learn. The network can use the hidden units it constructs to predict the OR value when learning XOR (i.e., the hint serves to decompose the problem for the network). Suddarth and Holden's work however only deals with hints in the form of useful output signals, and still requires network learning, while RATLE incorporates advice immediately.

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 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, other researchers (Frasconi, Gori, Maggini, & Soda, 1995; Omlin & Giles, 1992) insert prior knowledge about a finite-state automaton into a recurrent neural network. Like our FSKBANN work, this work does 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. He 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 it, thereby focusing learning.

### *Refining prior knowledge*

There has been a growing literature on automated "theory refinement" (Fu, 1989; Ginsberg, 1988; Ourston & Mooney, 1994; Pazzani & Kibler, 1992; Shavlik & Towell, 1989), and it is from this research perspective that our advice-taking work arose. Our new work differs by its novel emphasis on theory refinement in the context of multi-step problem solving in multi-actor worlds, as opposed to refinement of theories for categorization and diagnosis. Here, we view "domain theories" as statements in a procedural programming language, rather than the common view of a domain theory being a collection of declarative Prolog statements. We also address reinforcement learning, rather than learning-from-examples. Finally, unlike previous approaches, we allow domain theories to be provided piecemeal at any time during the training process, as the need becomes apparent to the advisor. In complex tasks it is not desirable to simply restart learning from the beginning whenever one wants to add something to the domain theory.

## 7. Conclusions

We present an approach that allows a connectionist, reinforcement-learning agent to take advantage of instructions provided by an external observer. The observer communicates advice using a simple imperative programming language, one that does not require that the observer have any knowledge of the agent’s internal workings. The reinforcement learner applies techniques from knowledge-based neural networks to directly insert the observer’s advice into the learner’s utility function, thereby speeding up its learning. Importantly, the agent does not accept the advice absolutely nor permanently. Based on subsequent experience, the learner can refine and even discard the advice.

Experiments with our RATLE system demonstrate the validity of this advice-taking approach; each of four types of sample advice lead to statistically significant gains in expected future reward. Interestingly, our experiments show that these gains do not depend on when the observer supplies the advice. Finally, we present results that show our approach is superior to a naive approach for making use of the observer’s advice.

In conclusion, we have proposed an appealing approach for learning from both instruction and experience in dynamic, multi-actor tasks. This work widens the “information pipeline” between humans and machine learners, without requiring that the human provide absolutely correct information to the learner.

## Acknowledgments

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 Alex, Mark Craven, Diana Gordon, Leslie Pack Kaelbling, Sebastian Thrun, and the two anonymous reviewers for their helpful comments on drafts of this paper. A shorter version of this article appeared as Maclin and Shavlik (1994).

## Appendix A

### 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). Appendix A in Maclin (1995) contains the definitions of the fuzzy terms (e.g., Near, Many, An, and East).

#### *SimpleMoves*

<pre> IF An Obstacle IS (NextTo <math>\wedge</math> <i>dir</i>)   THEN INFER OkPush<i>dir</i> END; IF No Obstacle IS (NextTo <math>\wedge</math> <i>dir</i>) <math>\wedge</math>   No Wall IS (NextTo <math>\wedge</math> <i>dir</i>)   THEN INFER OkMoved<i>dir</i> END; IF An Enemy IS (Near <math>\wedge</math> <i>dir</i>)   THEN DO_NOT Moved<i>dir</i> END; IF OkMoved<i>dir</i> <math>\wedge</math> A Food IS (Near <math>\wedge</math> <i>dir</i>) <math>\wedge</math>   No Enemy IS (Near <math>\wedge</math> <i>dir</i>)   THEN Moved<i>dir</i> END; IF OkPush<i>dir</i> <math>\wedge</math> An Enemy IS (Near <math>\wedge</math> <i>dir</i>)   THEN Push<i>dir</i> END </pre>	<p>Grab food next to you; run from enemies next to you; push obstacles at enemies behind obstacles. [This leads to 20 rules.]</p>
---	---

#### *NonLocalMoves*

<pre> IF No Obstacle IS (NextTo <math>\wedge</math> <i>dir</i>) <math>\wedge</math>   No Wall IS (NextTo <math>\wedge</math> <i>dir</i>)   THEN INFER OkMoved<i>dir</i> END; IF OkMoved<i>dir</i> <math>\wedge</math> Many Enemy ARE (<math>\neg</math> <i>dir</i>) <math>\wedge</math>   No Enemy IS (Near <math>\wedge</math> <i>dir</i>)   THEN Moved<i>dir</i> END; IF OkMoved<i>dir</i> <math>\wedge</math> No Enemy IS (<i>dir</i> <math>\wedge</math> Near) <math>\wedge</math>   A Food IS (<i>dir</i> <math>\wedge</math> Near) <math>\wedge</math>   An Enemy IS (<i>dir</i> <math>\wedge</math> {Medium <math>\vee</math> Far})   THEN Moved<i>dir</i> END </pre>	<p>Run away if many enemies in a direction (even if they are not close), and move towards foods even if there is an enemy in that direction (as long as the enemy is a ways off). [12 rules.]</p>
--	---

#### *ElimEnemies*

<pre> IF No Obstacle IS (NextTo <math>\wedge</math> <i>dir</i>) <math>\wedge</math>   No Wall IS (NextTo <math>\wedge</math> <i>dir</i>)   THEN INFER OkMoved<i>dir</i> END; IF OkMoveahead <math>\wedge</math> An Enemy IS (Near <math>\wedge</math> <i>back</i>) <math>\wedge</math>   An Obstacle IS (NextTo <math>\wedge</math> <i>side1</i>) THEN   MULTIACTION     Moveahead     Moveside1     Moveside1     Moveback     Pushside2   END END </pre>	<p>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.]</p>
--	--

*Surrounded*

```

IF An Obstacle IS (NextTo  $\wedge$  dir)
  THEN INFER OkPushdir END;
IF An Enemy IS (Near  $\wedge$  dir)  $\vee$ 
  A Wall IS (NextTo  $\wedge$  dir)  $\vee$ 
  An Obstacle IS (NextTo  $\wedge$  dir)
  THEN INFER Blockeddir END;
IF BlockedEast  $\wedge$  BlockedNorth  $\wedge$ 
  BlockedSouth  $\wedge$  BlockedWest
  THEN INFER Surrounded END;
WHEN Surrounded  $\wedge$  OkPushdir  $\wedge$  An Enemy IS Near
  REPEAT
    MULTIACTION Pushdir Moveddir END
  UNTIL  $\neg$  OkPushdir
END

```

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

**Appendix B****The Grammar for RATLE's Advice Language**

The start nonterminal of the grammar is *rules*. Grammar rules are shown with vertical bars (|) indicating alternate rules for nonterminals (e.g., *rules*, *rules*, and *ante*). Names like IF, THEN, and WHILE are keywords in the advice language. Additional details can be found in Maclin (1995).

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

*rules*  $\leftarrow$  *rule* | *rules* ; *rule*

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

*rule*  $\leftarrow$  IF *ante* THEN *conc* ELSE END  
 | WHILE *ante* DO *act* *postact* END  
 | *pre* REPEAT *act* UNTIL *ante* *postact* END

*else*  $\leftarrow$   $\varepsilon$  | ELSE *act*  
*postact*  $\leftarrow$   $\varepsilon$  | THEN *act*  
*pre*  $\leftarrow$   $\varepsilon$  | WHEN *ante*

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

*conc*  $\leftarrow$  *act* | INFER *Term\_Name* | REMEMBER *Term\_Name*  
*act*  $\leftarrow$  *cons* | MULTIACTION *clist* END  
*clist*  $\leftarrow$  *cons* | *cons* *clist*  
*cons*  $\leftarrow$  *Term\_Name* | DO\_NOT *Term\_Name* | ( *corlst* )  
*corlst*  $\leftarrow$  *Term\_Name* | *Term\_Name*  $\vee$  *corlst*

**Antecedents are logical combinations of terms and fuzzy conditionals.**

*ante*  $\leftarrow$  *Term\_Name* | ( *ante* ) |  $\neg$  *ante*

$$\begin{aligned} & | \textit{ante} \wedge \textit{ante} | \textit{ante} \vee \textit{ante} \\ & | \text{Quantifier\_Name Object\_Name IS } \textit{desc} \end{aligned}$$

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

$$\begin{aligned} \textit{desc} & \leftarrow \text{Descriptor\_Name} | \neg \textit{desc} | \{ \textit{dlist} \} | ( \textit{dexpr} ) \\ \textit{dlist} & \leftarrow \text{Descriptor\_Name} | \text{Descriptor\_Name} , \textit{dlist} \\ \textit{dexpr} & \leftarrow \textit{desc} | \textit{dexpr} \wedge \textit{dexpr} | \textit{dexpr} \vee \textit{dexpr} \end{aligned}$$

## Notes

1. 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. See the discussion in Section 5.
2. The agent needs this information when employing multiple-step plans (see Section 3). We include this information as input for all of the agents used in our experiments so that none will be at a disadvantage.
3. 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.
4. All results reported as statistically significant are significant at the  $p < 0.05$  level (i.e., with 95% confidence).
5. We also experimented with keeping only 100 pairs or sequences; the results using 250 pairs and sequences were better.

## References

- Abu-Mostafa, Y. (1995). Hints. *Neural Computation*, 7, 639–671.
- 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.
- Anderson, C. (1987). Strategy learning with multilayer connectionist representations. In *Proceedings of the Fourth International Workshop on Machine Learning*, pp. 103–114 Irvine, CA.
- Barto, A., Sutton, R., & Anderson, C. (1983). Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, 13, 834–846.
- Barto, A., Sutton, R., & Watkins, C. (1990). Learning and sequential decision making. In Gabriel, M., & Moore, J. (Eds.), *Learning and Computational Neuroscience*, pp. 539–602. 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.
- Chapman, D. (1991). *Vision, Instruction, and Action*. MIT Press, Cambridge, MA.
- Clouse, J., & Utgoff, P. (1992). A teaching method for reinforcement learning. In *Proceedings of the Ninth International Conference on Machine Learning*, pp. 92–101 Aberdeen, Scotland.
- Crangle, C., & Suppes, P. (1994). *Language and Learning for Robots*. CSLI Publications, Stanford, CA.
- Craven, M., & Shavlik, J. (1994). Using sampling and queries to extract rules from trained neural networks. In *Proceedings of the Eleventh International Conference on Machine Learning*, pp. 37–45 New Brunswick, NJ.
- Diederich, J. (1989). “Learning by instruction” in connectionist systems. In *Proceedings of the Sixth International Workshop on Machine Learning*, 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.
- Frasconi, P., Gori, M., Maggini, M., & Soda, G. (1995). Unified integration of explicit knowledge and learning by example in recurrent networks. *IEEE Transactions on Knowledge and Data Engineering*, 7, 340–346.
- Fu, L. M. (1989). Integration of neural heuristics into knowledge-based inference. *Connection Science*, 1, 325–340.
- Ginsberg, A. (1988). *Automatic Refinement of Expert System Knowledge Bases*. Pitman, London.
- 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*. Ph.D. thesis, Ecole Normale Supérieure 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. (Ed.), *Cognitive Skills and their Acquisition*, pp. 231–253. 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.
- 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.
- Kaelbling, L., & Rosenschein, S. (1990). Action and planning in embedded agents. *Robotics and Autonomous Systems*, 6, 35–48.
- Laird, J., Hucka, M., Yager, E., & Tuck, C. (1990). Correcting and extending domain knowledge using outside guidance. In *Proceedings of the Seventh International Conference on Machine Learning*, pp. 235–243 Austin, TX.
- Le Cun, Y., Denker, J., & Solla, S. (1990). Optimal brain damage. In Touretzky, D. (Ed.), *Advances in Neural Information Processing Systems*, Vol. 2, pp. 598–605. Morgan Kaufmann, Palo Alto, CA.
- Levine, J., Mason, T., & Brown, D. (1992). *Lex & yacc*. O'Reilly, Sebastopol, CA.
- 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 *Proceedings of the Tenth International Conference on Machine Learning*, pp. 182–189 Amherst, MA.
- Maclin, R. (1995). *Learning from Instruction and Experience: Methods for Incorporating Procedural Domain Theories into Knowledge-Based Neural Networks*. Ph.D. thesis, Computer Sciences Department, University of Wisconsin, Madison, WI.
- 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.
- Maclin, R., & Shavlik, J. (1994). Incorporating advice into agents that learn from reinforcements. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pp. 694–699 Seattle, WA.
- 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*, Vol. I, pp. 77–84. (Reprinted in M. Minsky, editor, 1968, *Semantic Information Processing*. Cambridge, MA: MIT Press, 403–409.)
- Monahan, G. (1982). A survey of partially observable Markov decision processes: Theory, models, and algorithms. *Management Science*, 28, 1–16.
- Mostow, D. J. (1982). Transforming declarative advice into effective procedures: A heuristic search example. In Michalski, R., Carbonell, J., & Mitchell, T. (Eds.), *Machine Learning: An Artificial Intelligence Approach*, Vol. 1. Tioga Press, Palo Alto.
- Nilsson, N. (1994). Telemo-reactive programs for agent control. *Journal of Artificial Intelligence Research*, 1, 139–158.

- Noelle, D., & Cottrell, G. (1994). Towards instructable connectionist systems. In Sun, R., & Bookman, L. (Eds.), *Computational Architectures Integrating Neural and Symbolic Processes*. Kluwer Academic, Boston.
- Omlin, C., & Giles, C. (1992). Training second-order recurrent neural networks using hints. In *Proceedings of the Ninth International Conference on Machine Learning*, pp. 361–366 Aberdeen, Scotland.
- Ourston, D., & Mooney, R. (1994). Theory refinement combining analytical and empirical methods. *Artificial Intelligence*, 66, 273–309.
- Pazzani, M., & Kibler, D. (1992). The utility of knowledge in inductive learning. *Machine Learning*, 9, 57–94.
- Riecken, D. (1994). Special issue on intelligent agents. *Communications of the ACM*, 37(7).
- Shavlik, J., & Towell, G. (1989). An approach to combining explanation-based and neural learning algorithms. *Connection Science*, 1, 233–255.
- Siegelmann, H. (1994). Neural programming language. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pp. 877–882 Seattle, WA.
- Suddarth, S., & Holden, A. (1991). Symbolic-neural systems and the use of hints for developing complex systems. *International Journal of Man-Machine Studies*, 35, 291–311.
- 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. (Eds.), *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior*, pp. 288–296. MIT Press, Cambridge, MA.
- Tesauro, G. (1992). Practical issues in temporal difference learning. *Machine Learning*, 8, 257–277.
- Thrun, S., & Mitchell, T. (1993). Integrating inductive neural network learning and explanation-based learning. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pp. 930–936 Chambery, France.
- Towell, G., & Shavlik, J. (1993). Extracting refined rules from knowledge-based neural networks. *Machine Learning*, 13, 71–101.
- Towell, G., & Shavlik, J. (1994). Knowledge-based artificial neural networks. *Artificial Intelligence*, 70, 119–165.
- 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*. Ph.D. thesis, King's College, Cambridge.
- Watkins, C., & Dayan, P. (1992). Q-learning. *Machine Learning*, 8, 279–292.
- Weigend, A. (1993). On overfitting and the effective number of hidden units. In *Proceedings of the 1993 Connectionist Models Summer School*, pp. 335–342 San Mateo, CA. Morgan Kaufmann.
- 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.
- Zadeh, L. (1965). Fuzzy sets. *Information and Control*, 8, 338–353.