Learning from Instruction and Experience: Methods for Incorporating Procedural Domain Theories Into Knowledge-Based Neural Networks

Richard Maclin

Technical Report #1285

August 1995

1285

LEARNING FROM INSTRUCTION AND EXPERIENCE: METHODS FOR INCORPORATING PROCEDURAL DOMAIN THEORIES INTO KNOWLEDGE-BASED NEURAL NETWORKS

By

Richard Frank Maclin

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

(COMPUTER SCIENCES)

at the

UNIVERSITY OF WISCONSIN – MADISON

1995

For Russell and Gregory,

who give me renewed hope for the future.

Abstract

This thesis defines and evaluates two systems that allow a teacher to provide instructions to a machine learner. My systems, FSKBANN and RATLE, expand the language that a teacher may use to provide advice to the learner. In particular, my techniques allow a teacher to give partially correct instructions about *procedural* tasks – tasks that are solved as sequences of steps. FSKBANN and RATLE allow a computer to learn both from instruction and from experience. Experiments with these systems on several testbeds demonstrate that they produce learners that successfully use and refine the instructions they are given.

In my initial approach, FSKBANN, the teacher provides instructions as a set of propositional rules organized around one or more finite-state automata (FSAs). FSKBANN maps the knowledge in the rules and FSAs into a recurrent neural network. I used FSKBANN to refine the Chou-Fasman algorithm, a method for solving the secondary-structure prediction problem, a difficult task in molecular biology. FSKBANN produces a refined algorithm that outperforms the original (non-learning) Chou-Fasman algorithm, as well as a standard neural-network approach.

My second system, RATLE, allows a teacher to communicate advice, using statements in a simple programming language, to a connectionist, reinforcement-learning agent. The teacher indicates conditions of the environment and actions the agent should take under those conditions. RATLE allows the teacher to give advice continuously by translating the teacher's statements into additions to the agent's neural network. The RATLE language also includes novel (to the theory-refinement literature) features such as multi-step plans and looping constructs. In experiments with RATLE on two simulated testbeds involving multiple agents, I demonstrate that a RATLE agent receiving advice outperforms both an agent that does not receive advice and an agent that receives instruction, but does not refine it.

My methods provide an appealing approach for learning from both instruction and experience in procedural tasks. This work widens the "information pipeline" between humans and machine learners, without requiring that the human provide absolutely correct information to the learner.

Acknowledgements

Many people had a significant effect on my life during the pursuit of this thesis, and I would like to take the time to acknowledge as many of these people as possible.

I would like to start out by thanking my advisor, Professor Jude Shavlik, who taught me a great deal about the process of doing research. I would also like to thank the professors who served on my defense committee: Charles Dyer, Richard Lehrer, Olvi Mangasarian, and Larry Travis. Your comments on the draft of my dissertation were greatly appreciated.

A number of people read versions of this dissertation and provided useful comments, and I would like to recognize each of these people. Foremost among these is Ernest Colantonio, who waded through the entire thesis, providing detailed comments throughout. I would also like to thank Kevin Cherkauer, Mark Craven, Karen Karavanic, Jami Moss, and Dave Opitz for commenting on various chapters of the dissertation.

I would like to thank a number of people whom I have had discussions with and whose comments greatly shaped my thinking for this work. This group includes Carolyn Allex, Mark Allmen, Kevin Cherkauer, Mark Craven, Ted Faber, Michael Giddings, Rico Gutstein, Haym Hirsch, Jeff Hollingsworth, Jeff Horvath, Karen Karavanic, Ken Koedinger, Rich Lehrer, Gary Lewandowski, Ganesh Mani, Tim Morrison, Dave Opitz, Sam Pottle, Jude Shavlik, Charlie Squires, Jim Stewart, Nick Street, Michael Streibel, Rich Sutton, Scott Swanson, Sebastian Thrun, Geoff Towell, Larry Travis, and Derek Zahn.

My family – my mom, Francine, and my dad, Tom, my grandparents Marguerite and Frank, Evalyn and Russell, my brothers and sister Tom, Cathy, Doug and Keith, my brothers' significant others Kathleen, Jeanine, and Michelle, and my nephews Russell and Gregory provided great support and inspiration to me when I needed it and I would like to thank them all.

I would also like to thank the many people who have become significant parts of my life through the past eight years for keeping me sane and centered. Thanks to Jack Ahrens, the man who defines cynical and the secret owner of Ahrens Cadillac; Carolyn Allex, someone who actually worries more than me; Mark Craven, for ignoring my singing; Ted Faber, who got all my movie quotes; Rico Gutstein, the nicest person I have ever known; Dave Haupert, the best teacher I know; Jeff Hollingsworth (JeffLeft), just for poking the guy who lit the cigarette at the Coliseum; Jeff Horvath (JeffRight), for being from Buffalo; Lynn Jokela, my

Minnesota connection; Karen Karavanic, my virtual cousin; Matt and Sara Koehler, who truly do not mind when someone throws up in the back of their car; Ann Kowalski, a woman of honor; Gary Lewandowski, for his uncompromising approach to life; Walter Ludwig, the smartest person I have ever known; Adam Mlot, my favorite navigator; Sam Pottle, my brewmaster; Sean Selitrenikoff, the sanest man I know; Nick Street, just for listening to me rant; Keith Tookey, for the worst puns ever; Geoff Towell, a consummate researcher; Heidi Vowels, my Oshkosh connection; Gene Zadzilka, ditto on the Buffalo thing; and Mike, Molly, and David Zwilling, my second favorite family (after my own).

Finally, I would like to thank the people whom I have met at PortaBella over the years. Thanks to Jack Ahrens, Stacy Deming, Jennifer Hebeler, Mike McCormick, Nicolle Nelson, Peter Ouimet and John Riggert, Anne Podlich and Bradley Niebres, Karen (the Jammin' lady), and Jill (a cocktail waitress with attitude). If you get the chance, stop by and have a drink (ask for Anne), I recommend:

Sunset in Paradise

$\frac{3}{4}$ large end Myer's rum	*$\frac{1}{2}$ small end triple sec*	*$\frac{3}{4}$ small end lime juice*
1 teaspoon brown sugar	*$\frac{1}{2}$ small end sweet vermouth*	

Blend until smooth, garnish with a cherry, orange slice, and pineapple slice.

This research was partially supported by a grant from the University of Wisconsin Graduate School, by the National Science Foundation under grant NSF IRI 90-02413, and by the Office of Naval Research under grants N00014-90-J-1941 and N00014-93-1-0998.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Imagine trying to teach a student a complex task like driving a car. As the teacher, you might take an approach of first giving the student some instruction about the task, and then letting the student experiment with the task, while you periodically give the student feedback about his[1] performance. For example, you might first explain how the car works, what the lines on roads mean, what the lights at intersections indicate, etc. Then you would take the student out to practice driving, telling the student when he makes mistakes and explaining what to do instead. This natural approach to teaching can be very efficient for both the student and the teacher. The ability of the student to learn from experience frees the teacher from having to provide a complete set of instructions. Also, the ability of the student to learn means that the teacher's instructions need not be perfect in order for them to be useful to the student. For the student, the instructions of the teacher give him a head start on learning the task – he does not need to induce, from scratch, the knowledge being provided by the teacher.

In the artificial-intelligence field of machine learning, we refer to the technique of combining teacher instruction with a student's learning from experience, as *learning from theory and data.* The "theory" portion of this phrase refers to the instructions provided by the teacher. We generally refer to the knowledge about a task provided by a teacher as a *domain theory.* A domain theory can be represented in any convenient formalism, though most domain theories in machine learning take the form of rules. "Learning from data" refers to the learning the student does by accumulating experiences, including feedback from the teacher, while actually performing the task. A computer that is learning from theory and data generally starts by incorporating a domain theory provided by a teacher. The learner then obtains a set of samples of how the task should work, and the learner uses these samples to *refine* the domain theory so that it produces the correct solution for these samples. Thus, these techniques are often called methods for *refining domain theories.*

[1]In order to help distinguish the teacher from the student, I will refer to the teacher as feminine and the student as masculine; no subtext is implied by this choice.

Table 1: A partial plan for making a left turn while driving.

```
1.  Set the left turn indicator.
2.  Wait for the light to turn green.
3.  Wait until there is no oncoming traffic.
4.  Turn the wheel left.
5.  ...
```

In machine learning, techniques for refining domain theories have been widely studied (Fu, 1989; Ginsberg, 1988; Maclin & Shavlik, 1993; Ourston & Mooney, 1990; Pazzani & Kibler, 1992; Thrun & Mitchell, 1993; Shavlik & Towell, 1989) and have proven to be effective on a number of different tasks. In this thesis I extend these techniques to a largely unexplored area – refining *procedural* domain theories. A procedural domain theory is a domain theory for a task that is solved as a sequence of steps rather than all at once. For example, a procedural domain theory for the driving task might have the multi-step rule for making a left turn shown in Table 1. The rule is procedural because each step follows in a sequence – the next step is executed in the *context* of the previous steps having already been executed. It is this contextual aspect which makes this type of task difficult for standard techniques for refining domain theories, and it is this aspect which my work addresses.

Most algorithms for refining domain theories assume that the task being learned consists of a set of input vectors (i.e., feature values describing the task) and corresponding output vectors (i.e., the value to be computed from the inputs), where each input-output pair is independent of the other input-output pairs. It is difficult, however, to represent contextual tasks this way. For example, in the driving task the input vector might include a 2D-image of the current scene from the car plus readings from the car gauges (see Figure 1). The output vector would then need to represent the action(s) the learner needs to do to achieve the current goal, but this can be extremely difficult. For example, we might have an output vector with one slot for each step the learner needs to take:

SetTurnIndicator			SetSteeringWheel			Wait For Green	
Left	Right	None	Left	Right	Centered		...
X			X			X	

but of course this loses the sequential aspect of the task, since no ordering is indicated for the actions (e.g., it might be disastrous to turn the wheel left and accelerate before waiting

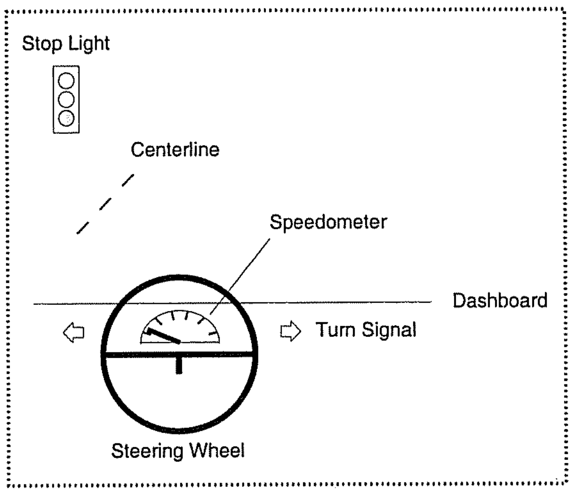


Figure 1: Sample scene showing information that might be available while driving.

for the green light). This aspect could perhaps be partially addressed by changing the representation to include order information:

SetTurnIndicator			SetSteeringWheel			Wait For Green	...
Left	Right	None	Left	Right	Centered		
1	0	0	4	0	0	2	

where non-zero numbers indicate the ordering for the actions, but even this does not really solve the problem, since what happens if an action must be executed multiple times as part of the plan? This problem could be fixed (at least in part), but each of these fixes makes the output vector more and more complex, making the overall learning task increasingly difficult.

Another problem with having a task defined by a single input and output vector pair is how to specify solutions for tasks that change during the execution of the solution. For example, imagine that the driver initially sees the situation shown in Figure 1 and begins the plan to turn left when a pedestrian appears (see Figure 2). In this case we would hope that the learner would alter the current plan and wait until the pedestrian is out of the way, but if the learner's input description is not a complete description of the world, it will be difficult to anticipate all the possible situations that can come up (Schoppers, 1994).

A more appealing approach is to treat a sequential task as a series of sub-tasks. In this approach each input vector represents the current description of the task, and the output vector is the next step to be taken. Figure 3 shows a sample of how this works for the driving task. In the initial environment (shown on the left in Figure 3), the driver decides to turn

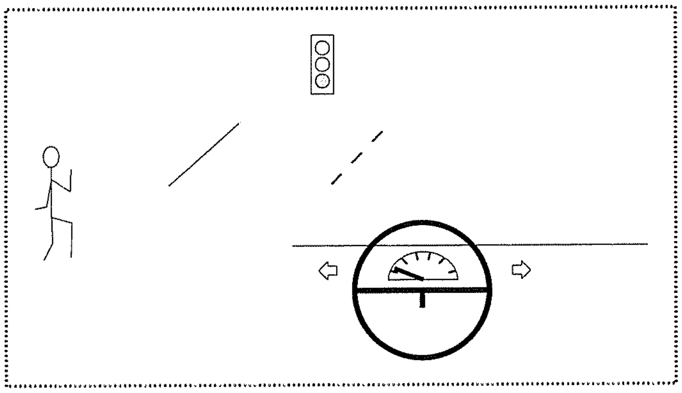


Figure 2: Solving a task that requires a sequence of actions can be difficult if unexpected things can happen – like a pedestrian suddenly appearing.

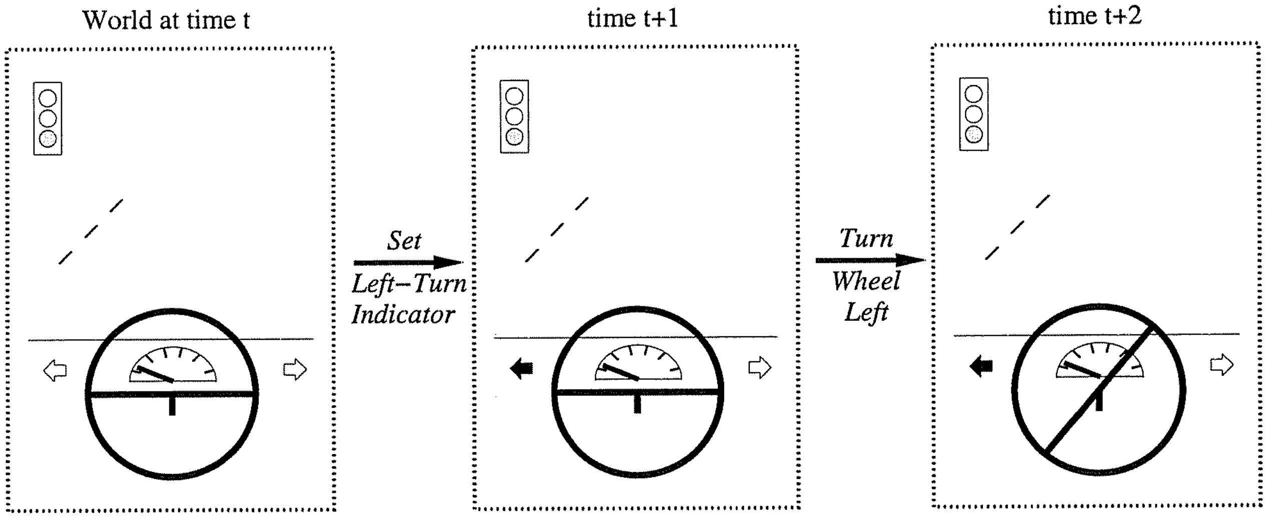


Figure 3: One solution to the problem of representing a series of actions is to represent a task as a series of sub-tasks, with one step per sub-task.

left, and starts this process by setting the left-turn indicator. The driver then continues by turning the wheel left in the resulting environment, until the actions needed to execute the left turn have each been performed. This is essentially the approach on which I will focus in this thesis.

Given that I want to apply a learning-from-theory-and-data approach, and that I am focusing on tasks solved as sequences of steps, the first issue to address is how to form instructions to the computer learner. In one sense, little work needs to be done as long as the teacher understands that her instructions, rather than attempting to solve the whole task, should concern the appropriate next output given the *current* input data. Thus, existing approaches already work, since the teacher could simply treat each input-response pair as a separate task. But this makes it impossible for the teacher to specify certain types of seemingly natural instructions for solving these types of tasks. For example, the teacher cannot give instructions about solving tasks in which the next step taken depends on the step taken previously. This thesis focuses on extending an existing technique for refining domain theories to a richer instruction language that allows these types of instructions.

In my preliminary approach to broadening the instruction language for refining procedural domain theories, I allowed instructions that "remember" information from previous problem-solving steps. In order to do this I introduced the idea of allowing a teacher to give instructions in the form of a finite-state automaton. The *state* of the automaton acts as a memory for information that the teacher thinks is important. At each problem-solving step the student determines a new state given the current input and then retains that state for the next problem-solving step. Thus the student does not have to solve each step as an independent task, but could make use of information determined in prior steps. In the driving task, the state could be used as a memory for information that is not always observable. For example, the learner could use the state to remember the current speed limit. In that case the state would be updated every time the learner observed a speed limit sign that changed the current speed limit.

After experimenting with my preliminary approach, I turned to the more ambitious problem of allowing a teacher to instruct a reinforcement learner. A reinforcement learner learns to select an action, given the current input, that will now (or in the future) cause the learner to receive positive reinforcements (i.e., rewards) while avoiding negative reinforcements (i.e., penalties). A reinforcement learner therefore naturally fits my general approach, since such a learner views a task as a sequence of environment and action pairs.

In Chapter 5, I present the language I developed that allows a teacher to instruct a reinforcement learner, and discuss the types of instruction allowed in that language. One natural type of instruction that I allow the teacher gives the learner plans similar to the one shown in Table 1. This plan suggests a *sequence* of steps to achieve a goal given the current input. Implementing this plan in an approach where each step is treated as a separate sub-task is tedious. The teacher must specify how the environment looks before the first step in the sequence. Then the teacher must specify how the environment looks before the second step, etc. Instead, I allow the teacher to specify this type of instruction as a sequence of related steps with one starting environment, and develop a mechanism that allows the student to incorporate this plan as a sequence.

Most techniques for refining domain theories accept instruction only before the refining process begins. In my approach for instructing a reinforcement learner I removed this restriction. I set up my mechanism so that instructions provided by the teacher result in *additions* to the student's current knowledge that may be made at any time during the student's learning process. This changes the process of instructing the student to a continual interaction. The teacher can watch the performance of the student, and when the teacher feels it is appropriate, she can provide instructions. The teacher can thus address problems of the student's behavior that she may not have anticipated initially. The student spends his time learning by exploring the environment, but periodically receives instructions from the teacher. After receiving instructions, the student returns to exploration, where he can use his experiences both to induce new "rules" as well as refine the instructions given by the teacher. This process can thus be repeated as often as the teacher cares to give instructions.

1.1 Thesis Statement

The key aspect of any algorithm for refining domain theories is the type of domain theory it is able to refine. (Recall that a domain theory is the set of knowledge that a teacher wants to communicate to a student.) The limitations on the types of knowledge a teacher can communicate are determined by the "language" the teacher uses to specify the domain theory – what constructs the teacher may use in creating the domain theory. Thus, the key aspect of my thesis is what types of instruction my systems are able to use and refine:

> ***Thesis:*** *Many interesting tasks are best solved as sequences of related steps. A powerful approach for creating computer problem solvers is to develop machine learners that learn from both instruction by a teacher and direct experience with the task at hand. Such an approach works by refining the instructions, called a* domain theory, *provided by the teacher. The learner refines the domain theory with a set of samples that show how the task is supposed to be solved. In order to apply this approach to tasks defined as sequences of steps, we need to define a language for instruction that includes constructs that capture the sequential aspect of problem solutions. Examples of such language features include constructs for remembering information during problem solving and for representing sequences of problem-solving steps. We are constrained to selecting language features that*

produce knowledge that can be refined *by inductive learning. A technique incorporating such features will demonstrate the benefits of a "learning from theory and data" approach for domains that were not previously amenable to this approach* – procedural *domains.*

Let us examine the claims of this thesis. The first claim, that many interesting tasks are best solved as sequences of steps, I will assert without proof – a casual perusal of any text on algorithms or planning should indicate that many interesting tasks are framed as needing procedural solutions. As demonstrated in my discussion above, a natural way to solve such problems is with a sequence of steps, since the defining characteristic of such problems is their sequential nature.

The power and usefulness of the machine learning approach of refining domain theories has been discussed previously (Maclin & Shavlik, 1993; Ourston & Mooney, 1990; Pazzani & Kibler, 1992; Thrun & Mitchell, 1993; Towell & Shavlik, 1994). The key to combining instruction with experience is that the teacher must be able to communicate the knowledge she considers important to the learner. If we accept the premise that the tasks in which we are interested are best viewed as sequences of steps, it is natural to assume that a language for instructing such a system would need to provide the ability to articulate sequential information. Thus I will focus on the last claim, that such an approach, one that refines *procedural* domain theories, will yield benefits similar to those shown for learning from theory and data approaches on non-sequential tasks.

1.2 Contributions of This Thesis

To assess the main claim of my thesis, I will present two techniques that refine procedural domain theories, and will also present experimental results that demonstrate the effectiveness of these methods. Both of my approaches for refining procedural domain theories use backpropagation (Rumelhart et al., 1986) on neural networks as their basic learning method and build on work in knowledge-based neural networks (Towell et al., 1990).

My preliminary technique (FSKBANN) focuses on refining domain theories represented as finite-state automata (Maclin & Shavlik, 1993). Table 2 shows the general input and output behavior of this system. It uses a finite-state domain theory and some training examples to refine the provided domain theory.

To implement the process in Table 2, I show how to extend an existing algorithm for

Table 2: Overview of the behavior of my system (FSKBANN) for refining finite-state domain theories.

Given:	An *imperfect* finite-state domain theory
	and
	A set of sample solutions
Produce:	A refined finite-state domain theory

refining domain theories to finite-state domain theories. First, I explain how to transform a finite-state automaton into a set of rules that incorporate contextual information. Next, I present a technique to translate these "state-based" rules into a corresponding neural network. To test my system, I present experiments employing this technique to refine the Chou-Fasman (1978) algorithm, a method for predicting protein secondary structure, showing that the resulting refined algorithm significantly outperforms the original algorithm.

In my second approach I develop a method that lets a teacher provide instructions to a reinforcement learner (Maclin & Shavlik, 1996). In this approach I use a learner that performs connectionist Q-learning (Lin, 1992; Sutton, 1988; Watkins, 1989). My work allows the reinforcement learner to take instructions in the form of programming-language constructs in a simple, yet expressive, language that I have developed. This language allows the teacher to make statements about single actions, as well as sequences of actions, to be taken in given environments. The computer learner translates these instructions into additions to its current neural network. Since the learner represents the knowledge as *additions* to its network, it is able to continuously accept more advice, rather than being limited to receiving advice only at the beginning.

My technique adds a third step to the reinforcement learner's traditional sense-react loop. At the start of each iteration of the loop, the student first checks if advice from the teacher is available, and if it is, the student stops to incorporate the advice. The student then returns to exploring the environment, using any future experience to evaluate the advice. The teacher returns to observing the student's behavior, providing further advice if it is warranted. Figure 4 outlines the interaction of the teacher and student in this system. To test this system, I present experiments for two simulated domains, one similar to video

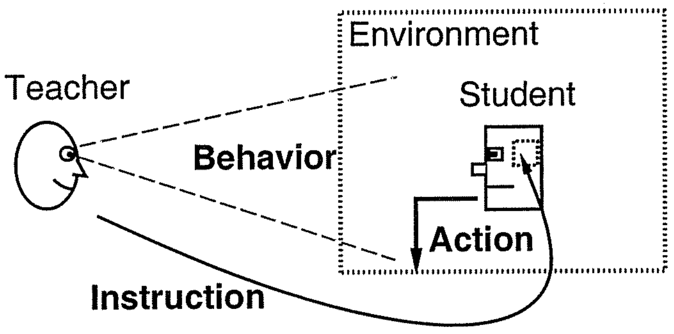


Figure 4: Interaction of the teacher and the student in my system for instructing a reinforcement learner. The process is a continuous loop – the student will continue to explore and learn from experience in its environment until it receives instructions from the teacher; it then stops, incorporates the instructions, and then returns to learning from experience. The teacher watches the behavior of the student until she decides to give the student instruction. After giving instruction the teacher returns to observing the behavior of the student to formulate more instructions.

games used by other researchers (Agre & Chapman, 1987; Lin, 1992), and a second domain for playing soccer.

1.3 Overview of This Thesis

In this chapter I have presented the motivation for my thesis and outlined the major issues I plan to explore in the remainder of this thesis. Chapter 2 presents background material necessary to understanding my work. In Chapter 3, I present my approach to refining domain theories expressed using finite-state automata. An overview of my method for allowing a teacher to instruct a reinforcement learner appears in Chapter 4. Chapter 5 defines my language that allows a teacher to instruct a reinforcement learner, and in the following chapter I give details on how the student maps constructs in this language to changes to the function being learned. In Chapter 7, I present experiments that verify this technique on two simulated domains. Next, I discuss related work, and finally I present conclusions and possible future directions for my work.

1.4 Summary

In summary, my work extends the applicability of machine learning by broadening the interaction between the human teacher and the machine learner. Rather than limiting communication to a set of training examples, possibly augmented with some inference rules, I allow the teacher to also tell the learner what to remember between steps and allow the teacher to provide sequences of actions to be performed under certain circumstances. Importantly, the teacher can provide this rich information at any time while the learner is improving himself. Based on her observations of the learner, the teacher is likely to produce *useful* instruction, something that is less likely if the teacher is required to provide all of her instructions before learning commences. Finally, the teacher's instructions need not be perfectly correct – the learner can refine them based on subsequent experiences.

Chapter 2

Background

In this chapter I will present an overview of the four areas of research that my work builds upon. The first area is neural networks, which I use as my means of *empirical learning* – learning a concept from samples of the concept. I chose neural networks because they have been shown to be a powerful inductive learning technique for a number of different types of problems (Atlas et al., 1990; Fisher & McKusick, 1989; Shavlik et al., 1991). I also chose neural networks because they allow me to build upon a particular algorithm for refining domain theories called KBANN (Towell et al., 1990; Towell, 1991; Towell & Shavlik, 1994), which I will outline in the second section. KBANN is an algorithm for translating propositional rules into neural networks, and has proved effective on a number of domains. The third area I will present is simple recurrent neural networks (Elman, 1990; Jordan, 1989), the specific type of neural network I will use in order to deal with the contextual information in procedural domain theories. Finally, I will describe reinforcement learning (Sutton, 1988), the task I investigate in Chapters 4-7.

2.1 Artificial Neural Networks

Neural networks are mathematical constructs based loosely on observations of the behavior of human brain cells. A neural network is composed of a set of units, corresponding to cells in the brain, and connections (or links) between those units. Each link has associated with it a weight that reflects the strength of the connection between the units. Associated with each unit is a net input value and an activation value. The net input value represents the total input impinging on the unit. One common way to calculate the net input value for a unit i is to sum the value of the activation times the weight for each of the units j connected to unit i:

$$NetInput_i = \sum_{j \in LinkedTo(i)} weight_{j \rightarrow i} \times activation_j \qquad (1)$$

where $weight_{j \to i}$ is the weight on the link from unit j to unit i, and $LinkedTo(i)$ are the units that have links *to* unit i. To this sum we add a term called the bias, which may be thought of as a weight on a link from a unit whose activation is always one. The reason for using a bias is discussed below. This gives us the equation:

$$NetInput_i = \left(\sum_{j \in LinkedTo(i)} weight_{j \to i} \times activation_j \right) + bias_i \qquad (2)$$

The activation value of unit i is calculated as a function of the net input to the unit. One common activation function is the sigmoid function:

$$activation_i = \frac{1}{1 + e^{-NetInput_i}} \qquad (3)$$

which can be thought of as a smoothed step function – at large positive net input values the activation for unit i will be near one and at large negative net input values the activation will be near zero. The advantage of a *smoothed* step function is that the derivative can be taken of this function, which means a simple learning rule exists. The bias term of Equation 2 can be thought of as a threshold for the smoothed step function. A large negative bias term acts as a high threshold, since the total net input from the connected units is offset by the large negative bias value.

A description of the units and the connections between units is referred to as the *architecture* of a neural network. Neural networks generally have a set of units that are labeled the input units – the activation values of these units are set prior to activating the network. After setting the activation values of the input units, the network then proceeds to calculate the activation values for the other units in the network; this is called *activating* the network. Neural networks generally produce a set of output values which represent the function the network is supposed to calculate. For example, we could create a network with two Boolean input units. If the network was supposed to calculate the AND of these two units, we would have one output unit whose value would be one when both input values are one and zero otherwise.

One commonly used network architecture (see Figure 5) divides the units of the network into three distinct groups: a layer of input units, zero or more layers of hidden units, and a layer of output units. In this type of architecture links are unidirectional and are only allowed from units in lower layers to units in higher layers (this type of network is often called a *feedforward* neural network). This feedforward aspect makes it easy to calculate the

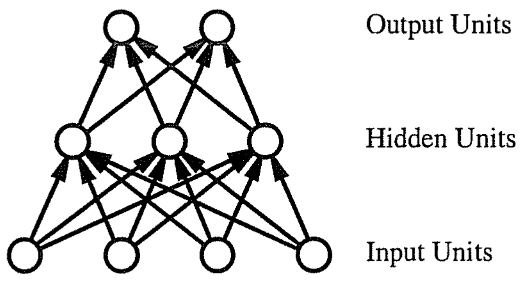


Figure 5: A standard feedforward neural network with one layer of input units, one layer of hidden units, and one layer of output units. Note that input units have links only out of them and output units have links only into them. In future neural-network diagrams, I will leave out the arrow heads on links to reduce diagram clutter. Unless a link is specifically marked, the reader may assume that it is a unidirectional link from the unit lower in the diagram to the unit higher in the diagram.

activation value of units since there are, by definition, no cycles in the connectivity graph; hence, the activation values can be calculated in a single pass.

The key question in neural networks is how to "teach" a network a particular function. We teach a neural network by *training* it on a set of examples of the input-output behavior we desire the network to reproduce. Table 3 shows the backpropagation (Rumelhart et al., 1986) method for training a network on an example. The error signal for a unit depends on the *cost* function used in learning. A basic principal of cost functions is that as the predicted

Table 3: The backpropagation (Rumelhart et al., 1986) algorithm for training a neural network on a labelled example.

---

1. Set the input activations to the inputs for the example.
2. Activate the network to determine the current predicted outputs.
3. Calculate an *Error* signal for the output units using the target outputs from the example and the predicted outputs from the current network (based on the cost function).
4. *Backpropagate* the error signals through the network; this involves determining the appropriate error signals for the non-output units.
5. Change the weights and biases in the network to reduce the cost.

---

output values move closer to the actual output values the cost decreases. One common cost function makes cost proportional to the sum of squared errors between the actual output values and the predicted output values:

$$cost \;=\; \frac{1}{2} \sum_{k=1}^{\#output\ units} (target_k \;-\; activation_k)^2 \tag{4}$$

In order to change the network to reduce the cost, we take the derivative of the cost function we have chosen with respect to the free parameters (i.e., the weights and biases). For the output units using the sigmoid activation function, taking the derivative of the cost function, we get an error signal (δ) of:

$$\delta_o \;=\; activation_o\,(1 \;-\; activation_o)\,(target_o \;-\; activation_o) \tag{5}$$

We backpropagate the output error signals recursively to the hidden units to get:

$$\delta_h \;=\; activation_h\,(1 \;-\; activation_h) \sum_{m \in LinkedFrom(h)} \delta_m\ weight_{h \rightarrow m} \tag{6}$$

where $LinkedFrom(h)$ are the units that have connections from unit h. For more details on how these derivatives are calculated see Rumelhart et al. (1986).

Note that we do not immediately try to set the weight to a "correct" value given its error signal. Rather we do *gradient descent* learning[1], where we change the weights by a small amount in the direction indicated by the sign of the error signal and the sign of the weight. One obvious reason to do this is that the partial derivatives do not take into account the interactions among the changes to the weights. Another important consideration is that we want the network to produce a function that is correct for all of the examples. We do this by presenting each of the examples a number of times and making a small change for each example – thus the network will hopefully discover a set of weights that will work for many of the examples. In this way we hope to achieve a network that *generalizes* well – a network that produces the correct output vector even for examples that are not presented during training. If we were to change the weights in the network to get one example right we might

[1]Technically, the learning is only gradient *descent* if we change the weights with respect to the error for all of the patterns. When we change the weights after determining the error for a single pattern (this is referred to as *online learning*), the technique is more properly referred to as gradient-based – since the error direction for a single pattern will not necessarily be in the same direction as the error for all of the patterns collectively.

undo the learning we did to get a previous example right. Finally, we do not want to make the assumption that the outputs of any one example are guaranteed to be correct; therefore we do not want to change the network to overly emphasize any particular example.

So, the basic approach to training a network is to repeatedly present a series of examples of the desired function until the function is "learned." We call the examples used to train the network the *training data.* A key issue is deciding when to stop training. A standard problem with neural networks is *overfitting.* Overfitting occurs when a network starts learning to "fit" the *noise* in the set of examples. Noise in a set of examples occurs when the output values are not perfect. In this case, it may be inefficient for the learning algorithm to completely reproduce the outputs for the training data (see Figure 6). Training a network until it overfits may reduce the generalization done by the network.

The problem of overfitting is ubiquitous in neural networks, and there are a number of approaches to handle this problem. One approach is to introduce a term into the cost function that penalizes a network that is overfitting the data. Such techniques are called *regularization* methods. One standard method for regularization is called weight decay (Hinton, 1986). Weight decay works as its name suggests: at each step each weight is decayed towards zero by a small amount. This approach is equivalent to adding a penalty term to the cost function proportional to the sum of the squared weights in the network. Other approaches to regularization include network pruning (Le Cun et al., 1990) and soft-weight sharing (Nowlan & Hinton, 1992).

Another way to prevent overfitting is to use a *validation set* (Lang et al., 1990). A

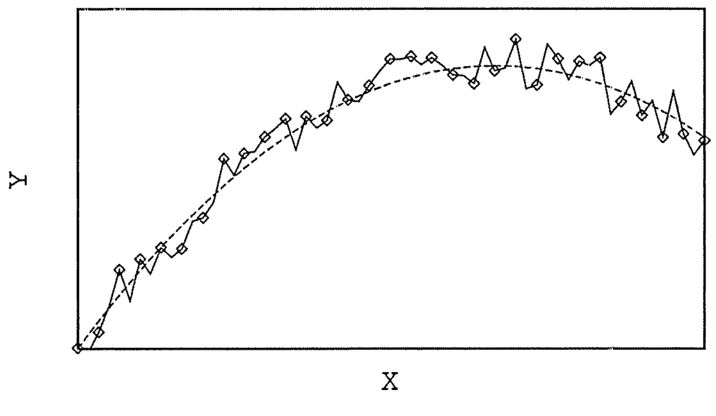


Figure 6: Two possible functions we could fit to a set of points (shown as diamonds). If we assume a certain amount of error in the observed y values associated with the x values, the solid curve is probably a more desirable solution; the dashed line would be a curve that "overfits" the data.

validation set is a subset of the training data that is set aside before training occurs. During training we periodically assess how well the network performs for the validation set and keep the network that does the best. This works on the theory that once the network starts overfitting, the generalization performance of the network will suffer and therefore the performance of the network on the validation set will go down. In this work, I use both weight decay and validation sets to prevent overfitting during my experiments – though not both at the same time. I will indicate which method I use in the description of the methodology for each of the experiments.

A final, and perhaps the most fundamental issue in neural networks, is how to select an appropriate neural-network architecture for a particular problem. The number of input and output units is largely determined by the problem being addressed, but the number of hidden layers, how to connect the units together, and even whether to use a feedforward network at all, are often decided on the basis of intuition. For example, if the resulting network does not have enough units to solve the problem, there is no easy way to determine this – other than having the learner fail to learn a solution. Methods addressing the problem of selecting an appropriate network architecture include techniques for network pruning (Le Cun et al., 1990), genetic search (Opitz & Shavlik, 1993), and cascade correlation (Fahlman & Lebiere, 1990). In this work I rely on the domain theory provided by the teacher to select an appropriate architecture. As will be seen in the next section, the instructions of the teacher result in a corresponding neural-network architecture. Thus, as in KBANN, I operate under the principle that the network architecture I have chosen reflects an expert's knowledge.

2.2 Knowledge-Based Neural Networks

The KBANN (for Knowledge-Based Artificial Neural Networks) algorithm (Towell et al., 1990; Towell, 1991; Towell & Shavlik, 1994) is designed to refine domain theories presented in the form of simple propositional rules (see Figure 7a). KBANN works by creating a neural network that contains the knowledge encoded in the rule set. We can then refine the resulting network using a standard neural-network learning algorithm such as backpropagation (Rumelhart et al., 1986) on a set of examples.

KBANN starts by constructing an AND-OR dependency graph from the rules in the domain theory. For example, the rules in Figure 7a would result in the dependency graph shown in Figure 7b. KBANN replaces each proposition with a corresponding network unit and

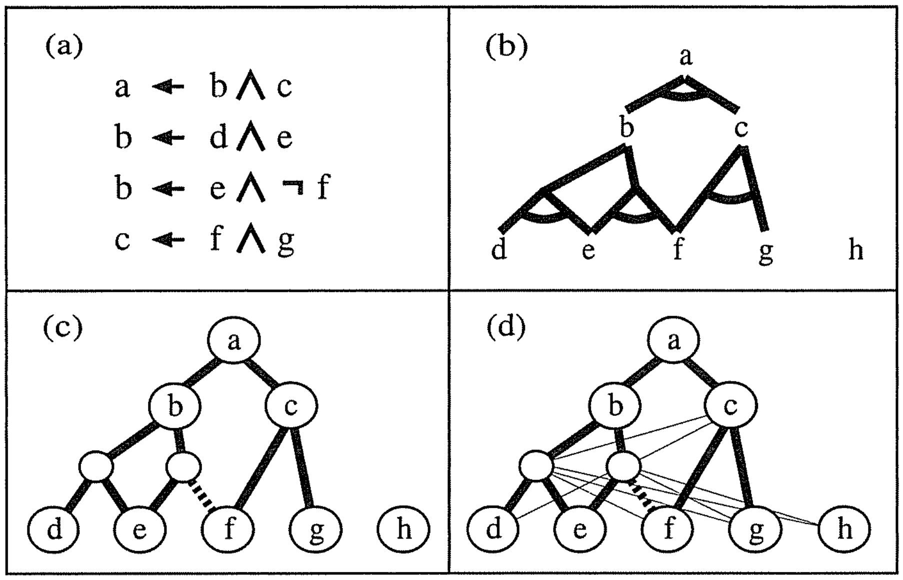


Figure 7: Sample of the KBANN algorithm: (a) a propositional rule set; (b) the rules viewed as an AND-OR dependency graph; (c) each proposition is represented as a unit (extra units are also added to represent disjunctive definitions, e.g., *b*), and their weights and biases are set so that they implement AND or OR gates, e.g, the weights *b*→*a* and *c*→*a* are set to 4 and the bias (threshold) of unit *a* to -6; (d) 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).

adds units where conjunctions are combined into disjunctions (see Figure 7c). 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 (i.e., the net input is highly positive, and therefore the activation value of the unit is near one), and false if the unit is inactive (i.e., the net input is highly negative, and therefore the activation value of the unit is near zero). Thus, to capture the knowledge of the rules, KBANN must set the weights and biases of the network so that the units representing propositions have this Boolean property: the unit must have activation near one when the rule set indicates the proposition is true and zero otherwise.

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. For an AND unit, this is done by setting the weight for each unnegated antecedent to a large positive value (in my work, to a value uniformly in the range [3.9,4.1]), and the weight for negated antecedents to a large negative value (in the range [-4.1,-3.9]). The bias is then set to

$$bias = -4\ (\#unnegated\_antecedents - 0.5) \tag{7}$$

Thus, the unit will have a large positive net input value (around 2.0), if and only if, each of the unnegated antecedents to the unit are true (near one) and all of the negated antecedents are false (near zero); otherwise the unit will have a large negative net input value. For an OR unit the weights are set similarly, but the bias is set to

$$bias = 4\ (\#negated\_antecedents - 0.5) \tag{8}$$

Thus, the unit will have a large negative net input value (around -2.0), if and only if, each of the negated antecedents to the unit are true (near one) and all of the unnegated antecedents are false (near zero); otherwise the unit will have a large positive net input value.

Once the network has been initialized with the knowledge in the rule set KBANN finishes by adding links between units at "adjacent" levels that are not already connected (see Figure 7d). In this work, I calculate the level of a unit bottom up (from the input layer upwards). The links I add are low-weighted links (e.g., links with weights in the range [-0.1,0.1]). These links serve as a basis for refining the domain theory – they allow the network to add an antecedent to a rule where that antecedent was not in the original domain theory.

In this thesis, I expand on the language that KBANN is able to translate. One extension I introduce adds the capability for the network to "remember" values across multiple inputs. I use this capability to retain contextual information during the execution of a task (as discussed in Chapter 1). In order to make this extension, I need a more powerful neural network than the standard feedforward network shown in Figure 5, one that has the ability to "remember" information. To do this I use simple recurrent neural networks, a type of recurrent network, which I present in the next section.

2.3 Simple Recurrent Neural Networks

A simple recurrent neural network (SRN) (Elman, 1990; Elman, 1991; Jordan, 1989), is, as its name implies, a straightforward form of recurrent network. In an SRN, some of the units in the network are connected to input units that "remember" the values of those units from the previous activation of the network. This is done by introducing recurrent links.

These recurrent links connect the units whose values are to be "remembered" to input units that represent the remembered values. These links use a simple copying function as their activation function.

Figure 8 shows an example of an SRN. In this SRN one input value, one hidden value, and one output value are remembered. The input units can be divided into two groups: the input units whose values are set for the current input vector; and those input units, called *context* units, whose values are copies of the values of units from the previous activation of the network.

We activate an SRN for an example by first setting the activation values for the context units; they are set by copying the activation values at the last time step of the units they are connected to. Then we set the remaining input values according to the input vector of the example. Finally we activate the remainder of the network as done with a standard feedforward network. During learning we ignore the recurrent links – no learning is done on these links. This is the main difference between SRNs and other recurrent techniques such as backpropagation through time (Minsky & Papert, 1969; Rumelhart et al., 1986) and recurrent backpropagation (Pineda, 1987). Most recurrent network techniques are more complicated than SRNs because they have to deal with learning across these recurrent links, but they are also more powerful for this same reason.

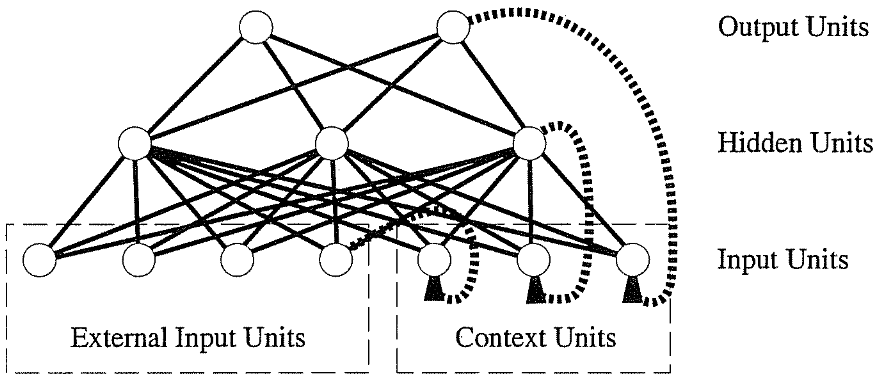


Figure 8: A sample simple recurrent neural network (SRN). The recurrent links are shown as dashed lines. These links are used to create context units, whose activation values are copies of the activation values of the units to which they are connected. The activation value of a context unit is the activation value of the unit to which it is connected from the previous activation of the network – these units "remember" previous activation values. In this SRN the rightmost non-context unit from each layer is remembered, although any possible combination of the units could be remembered.

In the work, I use the instructions the teacher presents to select what contextual information I retain via context units. The key questions I address are (1) how does this contextual information get represented in the instruction languages my learner understands? and (2) how does this information get translated into appropriate network units, including context units?

2.4 Reinforcement Learning

Reinforcement learning (RL) is the focus of the second portion of this thesis, and therefore I will spend some time outlining this area. A number of early AI systems made use of reinforcement learning techniques, work such as Samuel's checker-playing program (Samuel, 1959) and Holland's bucket-brigade algorithm (Holland, 1986). In standard RL (see Figure 9), the reinforcement learner (usually called the agent) senses the current state of the world, chooses an action to apply to the world, and occasionally receives rewards and punishments based on its actions and the states it sees. Figure 10 shows an example of a reinforcement learning problem. Here the states are the agent's location in a 2D maze, the actions are moves to adjacent states, and the reinforcements favor the agent finding the goal as quickly as possible. Based on the reinforcements from the environment, the task of the agent is to discover a *policy* that indicates the "best" action to take in each state (see Figure 11).

In this work, I employ a particular type of RL called Q-learning. In Q-learning (Watkins, 1989) the policy is implemented by an action-choosing module that employs a *utility function* that maps states and actions to a numeric value (the utility). The utility function in Q-learning is the Q-function which maps pairs of states (S) and actions (A) to the predicted future (discounted) reward $Q(S, A)$ that will be achieved if action A is taken by the agent in state S and the agent acts optimally afterwards. It is easy to see that given a perfect

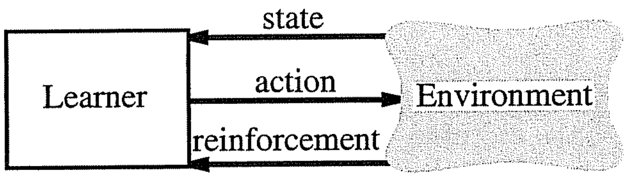


Figure 9: A reinforcement learning agent interacts with the environment in three ways: it receives a description of the state from the environment, selects an action that changes the environment, and then receives a reinforcement signal based on the action it chose.

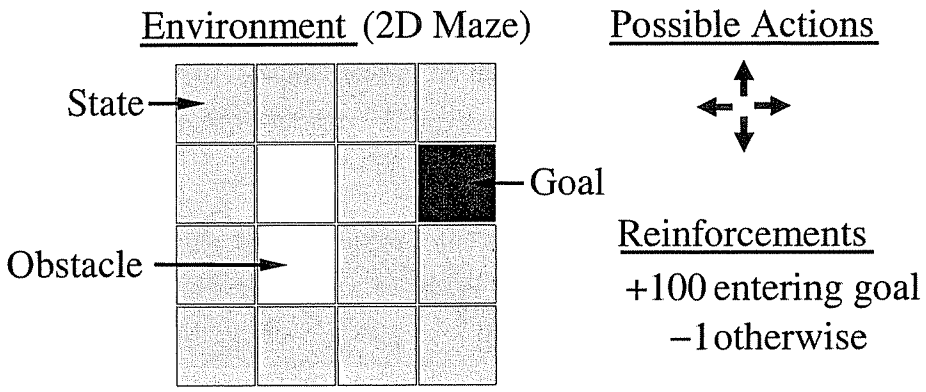


Figure 10: A sample reinforcement learning problem, with the set of possible states, the actions that the agent may take to change (move around in) the environment and the reinforcement signals.

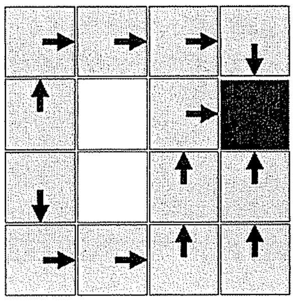


Figure 11: An optimal policy function for the problem shown in Figure 10. An optimal policy function should achieve the maximum future discounted reward.

version of this function, the optimal policy is to simply choose, in each state that is reached, the action with the largest utility.

Note that the utility function predicts the *discounted* future reward that will result from an action:

$$discounted\ future\ reward\ =\ \sum_{t=1}^{\infty} \lambda^{t-1}\ reward_t \tag{9}$$

where $reward_t$ is the reinforcement received at time t and λ is the discount value. Generally we use a λ value between zero and one. This is done for two reasons. First, setting $\lambda = 0$ would mean all future rewards are ignored, while choosing the $\lambda = 1$ would mean that all solutions that eventually reach the same goal would be equivalent, even if one path was arbitrarily longer. Choosing a value between zero and one causes the agent to seek solutions that will produce future rewards, but the discounting causes the agent to seek to achieve these solutions sooner rather than later. A second reason for choosing a value between zero

Table 4: The processing loop of an RL agent. Once the policy function is learned, we forgo step 5 and in step 2 simply choose the action with the highest utility.

1. Read sensors (description of state).
2. Stochastically choose an action, where the probability of selecting an action is proportional to the logarithm of its predicted utility (i.e., its current Q value). Retain the predicted utility of the action selected.
3. Perform selected action.
4. Measure reinforcement, if any.
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 for taking the previous action in the previous state; use the difference between the new estimate of utility and the previous estimate to update the predicted Q-value for the previous state and action.
6. Go to 1.

and one is that it makes it possible to use a simple learning rule.

To learn the utility function, the agent starts out with a randomly chosen utility function and explores its environment using the loop shown in Table 4. 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. To update the current predicted utility value ($Q(s_t, a_t)$) for a particular state (s_t) and action (a_t), we perform the action in that state and obtain a new prediction $\hat{Q}(s_t, a_t)$ using the following formula:

$$\hat{Q}(s_t, a_t) \; = \; reward_t \; + \; \lambda \; max\{Q(s_{t+1}, a) \mid a \in Actions\} \tag{10}$$

We then take this estimate and update $Q(s_t, a_t)$ with the following rule:

$$\Delta \; Q(s_t, a_t) \; = \; \mu \; [\hat{Q}(s_t, a_t) \; - \; Q(s_t, a_t)] \tag{11}$$

where μ is the learning rate parameter for the system. Watkins (1989) has shown that such an approach will, in the limit, find an optimal policy function.

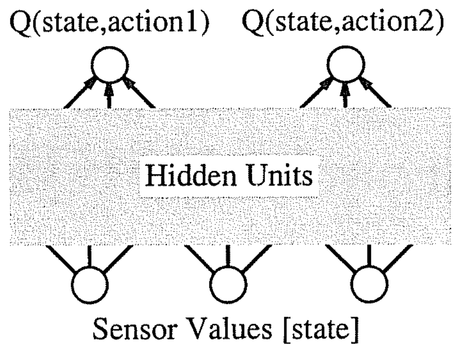


Figure 12: A Q-function can be represented with a neural network. For the neural network, the input is the description of a state and the outputs are the predicted utilities for each of the actions for the input state.

In this thesis I will be employing a generalization of Q-learning called *connectionist* Q-learning (i.e., neural-network Q-learning). In connectionist Q-learning (Lin, 1992; Sutton, 1988; Watkins, 1989), the utility function is implemented as a neural network, whose inputs describe the current state and whose outputs are the utility of each action (see Figure 12). I employ *connectionist* Q-learning because representing a complete Q-function table would be infeasible, both in terms of the size of the table and the time it would take to learn the complete table. In connectionist Q-learning the learner creates a function that may generalize across a number of states when predicting the Q value for an action. Thus, a connectionist Q-learner may be able to represent the entire table with a much smaller set of parameters, if many table entries can be represented by a simple network function. While a connectionist Q-function is advantageous both in terms of space and in time to learn a particular Q-function, because of the gradient-based nature of neural networks, a connectionist Q-function is not guaranteed to find an optimal policy function – a limitation which will become important later.

Chapter 3

A First Approach: FSKBANN

This chapter describes my initial approach (called FSKBANN) to refining a procedural domain theory (Maclin & Shavlik, 1991). FSKBANN (for Finite-State KBANN) translates domain theories that include generalized finite-state automata - FSA (Hopcroft & Ullman, 1979), into corresponding neural networks. FSKBANN then refines the resulting networks using backpropagation (Rumelhart et al., 1986) on a set of training examples. To demonstrate the value of FSKBANN I will present experimental results from refining the Chou-Fasman (1978) algorithm, a method for predicting (an aspect of) how globular proteins fold, an important and particularly difficult problem in molecular biology.

3.1 Overview of FSKBANN

A procedural domain theory deals with tasks that are sequential in nature and that may have some significant contextual aspect to solving the problem. In FSKBANN I use *state* in the domain theory to represent the context of the current problem-solving step. For example, if the task is to drive a car to the market, the state might indicate the current speed limit. Any rules the teacher introduces to help solve this task can therefore take into account the state of the partial solution – rules to accelerate the car might consider the current speed relative to the speed limit, for example. The state thus acts like a memory for the learner.

To deal with the sequential nature of the tasks, I process examples sequentially (i.e., activate the network for the first input vector in the sequence, then the second, then the third, etc.), retaining the appropriate state values for the next input vector. Table 5 shows the type of task to which FSKBANN is applicable – domain theories for state-based problem solvers.

To refine this type of domain theory I extend the KBANN algorithm (Towell et al., 1990) to theories that include finite-state automata. The resulting domain theory is a hybrid one – consisting of finite-state automata, as well as rules that can make reference to the states of the FSAs. In order to translate this hybrid domain theory into a corresponding neural

Table 5: Outline of the type of task to which FSKBANN is applicable.

```
Repeat
    Set input  = current externally-provided information
                                    +
                 previous internal information
    Produce, using domain theory,
        output = results specific to this problem-solving step
                                    +
                 current internal information
Until a Termination Criterion is met.
```

network, I add two processes to the KBANN algorithm: (1) a process for translating FSAs into a set of rules representing each state with a pair of values; and (2) a process for mapping the resulting rules, state values and all, into a neural network. Note that a domain theory may contain more than one FSA. When a domain theory has multiple FSAs, FSKBANN maintains the state of each of the FSAs.

3.2 Transforming FSAs into State-Based Rules

FSKBANN starts by transforming each FSA in the domain theory into a set of corresponding rules incorporating state information. FSKBANN represents each of the states named in each FSA by a pair of values: a previous value and a current value. The previous value (s_{i-1}) for a state s represents that state's value (i.e., whether the state s was *true*) in its FSA before processing the current input vector, and the current value (s_i) represents that state's value after processing the input vector. Collectively, I will refer to the set of previous values for all of the states as the previous state vector and the current values for all of the states as the current state vector.

I represent states in this manner because I approach solving the task as a sequence of steps. The previous state vector acts as a memory of which states were true (what the state was in each FSA) before each step, and the current state vector represents the state in each

FSA after a step. In the next section, I will discuss how I represent state values in the network.

Once FSKBANN sets up the previous and current state values for each state, it replaces each transition in each FSA by a rule. Each transition of the form:

$$\begin{tikzcd}\boxed{A} \arrow[r, "k"] & \boxed{B}\end{tikzcd}$$

results in a rule of the form:

$$B_i \leftarrow A_{i-1} \wedge k$$

The "character" for a transition k does not need to be a single "character" or value from the input vector, but can be a complex proposition calculated by the rules supplied with the domain theory. Also, the teacher can use the current and previous values of a state in the regular rules for the domain theory.

As an example, consider the problem of adding together a sequence of bits representing a number (e.g., 0011 + 1001 = 1100), where we are expected to process the two input numbers one bit at a time and then output the appropriate bit (e.g., first we would receive 1 and 1 and output 0, then we would receive 1 and 0 and output 0 because of the carry, etc.). In order to solve this problem we need to keep track of whether or not we are carrying a bit, which we can do with the FSA shown in Figure 13. This FSA and the rules shown in Table 6 constitute a domain theory to solve this problem. FSKBANN would take this domain theory and start by transforming the FSA into the rules in Table 7, which FSKBANN would add to the rules in Table 6.

After the translation process, the domain theory consists of three lists: (1) a list of each state that FSKBANN must represent (e.g., *carry_is_zero* and *carry_is_one* from the FSA in Figure 13); (2) a list of the start state for each FSA in the domain theory (e.g., *carry_is_zero* from FSA); and (3) a list of rules, which may contain references to previous and current state values (e.g., the rules in Table 7 plus the rules in Table 6). FSKBANN compiles the list of

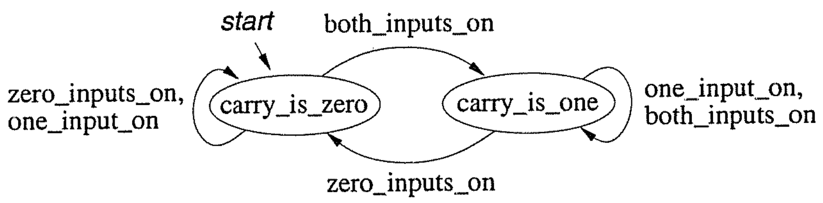


Figure 13: An FSA that could be used to track the carry bit while adding binary numbers.

Table 6: Rules combined with FSA in Figure 13 to form a binary-addition domain theory.

$both\_inputs\_on$	$\leftarrow$	$input1\_on$	$\wedge\ input2\_on$
one_input_on	$\leftarrow$	$input1\_on$	$\wedge\ \neg input2\_on$
one_input_on	$\leftarrow$	$\neg input1\_on$	$\wedge\ input2\_on$
$zero\_inputs\_on$	$\leftarrow$	$\neg input1\_on$	$\wedge\ \neg input2\_on$
$output\_on$	$\leftarrow$	$both\_inputs\_on$	$\wedge\ carry\_is\_one_{i-1}$
$output\_on$	$\leftarrow$	one_input_on	$\wedge\ carry\_is\_zero_{i-1}$
$output\_on$	$\leftarrow$	$zero\_inputs\_on$	$\wedge\ carry\_is\_one_{i-1}$

Table 7: Rules FSKBANN produces from the FSA shown in Figure 13.

$carry\_is\_zero_i$	$\leftarrow$	$carry\_is\_zero_{i-1}$	$\wedge\ zero\_inputs\_on$
$carry\_is\_zero_i$	$\leftarrow$	$carry\_is\_zero_{i-1}$	$\wedge\ one\_input\_on$
$carry\_is\_zero_i$	$\leftarrow$	$carry\_is\_one_{i-1}$	$\wedge\ zero\_inputs\_on$
$carry\_is\_one_i$	$\leftarrow$	$carry\_is\_one_{i-1}$	$\wedge\ one\_input\_on$
$carry\_is\_one_i$	$\leftarrow$	$carry\_is\_one_{i-1}$	$\wedge\ both\_inputs\_on$
$carry\_is\_one_i$	$\leftarrow$	$carry\_is\_zero_{i-1}$	$\wedge\ both\_inputs\_on$

states and the list of start states by examining all of the FSAs in the domain theory. The list of rules contains the rules from the original domain theory as well as all of the rules FSKBANN introduced to represent the transitions in the FSAs. FSKBANN translates these rules into a corresponding network, as described in the next section.

3.3 Mapping State-Based Rules to a Neural Network

The key to mapping rules containing references to states is the type of network into which FSKBANN maps the rules. FSKBANN maps state-based rules into a Simple Recurrent neural Network (SRN) (Elman, 1990; Elman, 1991; Jordan, 1989), in which the *context* units of the network represent the values of the states in the FSA.

FSKBANN's process of mapping rules works similarly to KBANN's. First FSKBANN sets up units for all of the input and output units. FSKBANN diverges from KBANN by next setting up units representing each of the states on the list of states produced by processing the FSAs in the domain theory. FSKBANN represents each state with two units: one for the previous value of the state, and one for the current value of the state. It then connects the current value of the state to the previous value with a time-delayed recurrent link. From that point, FSKBANN works the same as KBANN, treating the previous values of states as input units, and the current values of states as hidden, output, or input units (depending on how they appear in the domain theory). From the rules in Tables 6 and 7, FSKBANN produces the network shown in Figure 14.

FSKBANN uses the list of start states from the FSAs to determine which of the previous state units are active when starting the task. In the example shown in Figure 14, *carry_is_zero* is the start state of the FSA, so FSKBANN would make the unit representing the previous value of *carry_is_zero* highly active and all of the other previous state units from this FSA (i.e., *carry_is_one*) inactive at the start of the binary-addition task. A second FSA would introduce more state units and another initially-active previous state value – the start state for that FSA.

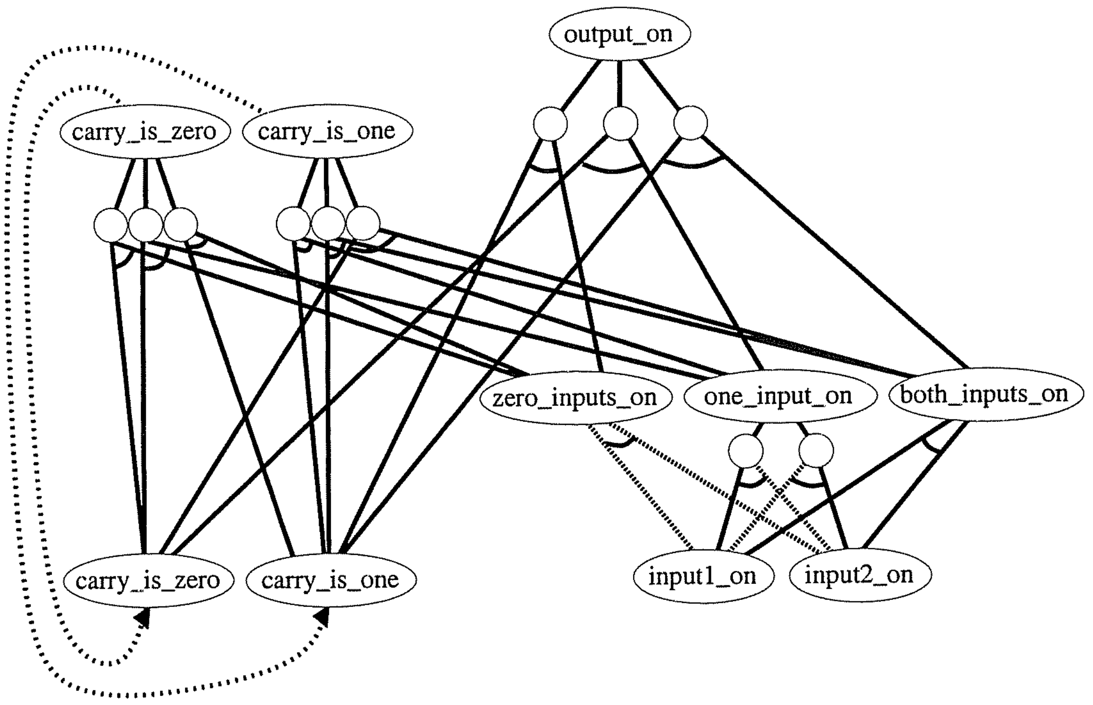


Figure 14: FSKBANN network for rules from Tables 6 and 7. Units constructed as ANDs are shown with arcs, the other units represent ORs. The dashed links with arrow heads are recursive links and the remaining dashed links are negatively-weighted links. The low-weighted links added at the end of the FSKBANN process are not shown.

3.4 Experiments

I chose to experiment by refining the Chou-Fasman (1978) algorithm in order to evaluate the usefulness of FSKBANN. My experiments demonstrate that FSKBANN is indeed able to refine this algorithm, producing a small, but statistically significant, gain in accuracy over the unrefined algorithm and over standard neural-network approaches.

In this section I will start by giving a description of the secondary-structure problem. Then I will present the Chou-Fasman algorithm and describe how I represent this algorithm as a procedural domain theory. Following that I will present results and an in-depth empirical analysis of the strengths and weaknesses of the FSKBANN refined algorithm, the unrefined algorithm, and standard neural-network approaches.

3.4.1 Protein Secondary-Structure Prediction

The Chou-Fasman algorithm attempts to solve the protein secondary-structure prediction task, a sub-task of the protein-folding task. I chose the Chou-Fasman algorithm as a testbed because it is one of the best-known and widely-used algorithms in this field. I also chose the secondary-structure task because a number of machine learning techniques are currently being applied to this task, including neural networks (Holley & Karplus, 1989; Qian & Sejnowski, 1988), inductive logic programming (Muggleton & Feng, 1990), case-based reasoning (Cost & Salzberg, 1993), and multistrategy learning (Zhang et al., 1992). Thus this task provides a good baseline for the effectiveness of my approach.

The protein-folding task is an open problem that is becoming increasingly critical as the Human Genome Project (Watson, 1990) proceeds. Proteins are long strings of amino acids, containing several hundred elements on average. There are 20 naturally occurring amino acids, denoted by different capital letters. The string of amino acids making up a given protein constitutes the *primary* structure of the protein. Once a protein forms, it folds into a three-dimensional shape which is known as the protein's *tertiary* structure. Tertiary structure is important because the form of the protein strongly influences its function.

At present, determining the tertiary structure of a protein in the laboratory is costly and time consuming. An alternative solution is to predict the *secondary* structure of a protein as an approximation. The secondary structure of a protein is a description of the local structure surrounding each amino acid. One prevalent system of determining secondary structure divides a protein into three different types of structures: (1) α-helix regions, (2) β-sheet regions, and (3) random coils (all other regions). Figure 15 shows the tertiary structure of a

Ribbon drawing here (not available in electronic version)

Figure 15: *Ribbon* drawing of the three-dimensional structure of a protein (Reprinted with permission from Richardson & Richardson 1989). The areas resembling springs are α-helix structures, the flat arrows represent β-sheets, and the remaining regions are random coils.

protein and how the shape is divided into regions of secondary structure. For our purposes, the secondary structure of a protein is simply a sequence corresponding to the primary sequence. Table 8 shows a sample mapping between a protein's primary and secondary structures, with amino acids that are part of coil structures shown as underscores (_) since coil is a default class.

Table 9 contains predictive accuracies of some standard algorithms from the biological literature for solving the secondary-structure task. In the data sets used to test the algorithms, 54-55% of the amino acids in the proteins are part of coil structures, so 54% accuracy can be achieved trivially by always predicting coil. It is important to note that many biological

Table 8: Primary and secondary structures of a fragment of a sample protein.

Primary (20 possible amino acids)	P	S	V	F	L	F	P	P	K	P
Secondary (three possible local structures)	_	_	β	β	β	β	_	_	_	α

Table 9: Accuracies of various (non-learning) prediction algorithms.

Method	Accuracy	Comments
Chou and Fasman (1978)	58%	data from Qian and Sejnowski (1988)
Lim (1974)	50%	from Nishikawa (1983)
Garnier and Robson (1989)	58%	data from Qian and Sejnowski (1988)

researchers believe that algorithms that take into account only local information can achieve only limited accuracy (Wilson et al., 1985), generally believed to be at most 80-90%.

I should note that the results in Table 9 are derived from my work on re-implementing these algorithms, as well as work by Nishikawa (1983). The results reported in these papers are significantly higher (up to 70% accuracy is reported), but cannot be replicated since they involved significant human decision-making. Nishikawa (1983) reimplemented these methods as algorithms and tested them on unknown proteins, producing results comparable to those I report in Table 9.

Another approach to the secondary-structure task is to use a learning method such as neural networks (Holley & Karplus, 1989; Qian & Sejnowski, 1988). The neural networks in these efforts have as input a *window* of amino acids consisting of the central amino acid whose secondary structure is being predicted, plus some number of the amino acids before and after it in the sequence (similar to NETTALK networks, Sejnowski & Rosenberg, 1987). The output of the network is the secondary structure for the central amino acid. Figure 16 shows the general structure of this type of network; Table 10 presents results from these studies. Note that the Qian and Sejnowski results reported in this table are somewhat different from those I will report, even though I test on the same dataset. This discrepancy occurs because Qian and Sejnowski report the *best* results they achieve during training – they

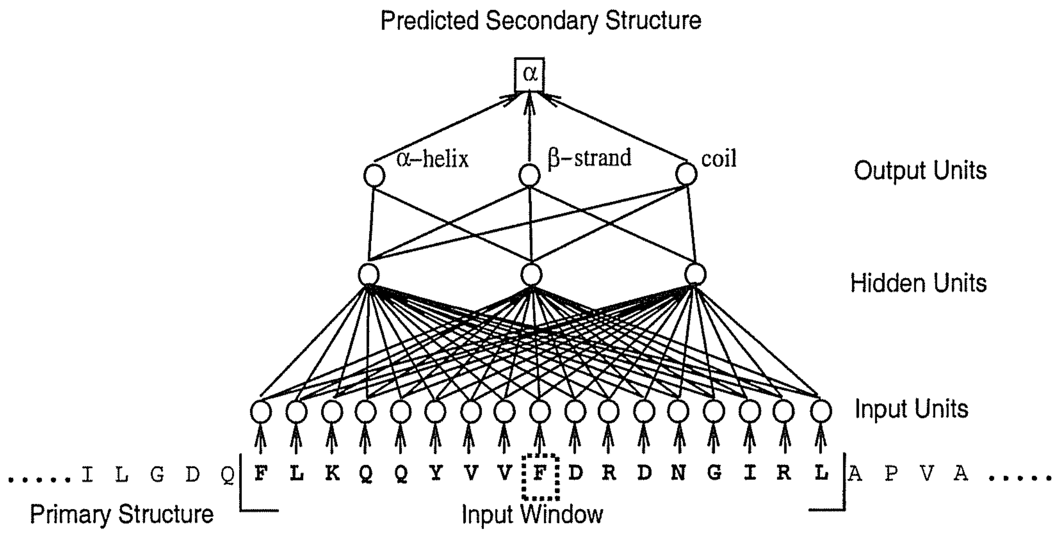

Predicted Secondary Structure
α
α-helix
β-strand
coil
Output Units
Hidden Units
Input Units
.....I L G D Q F L K Q Q Y V V F D R D N G I R L A P V A .....
Primary Structure
Input Window

Figure 16: Neural-network architecture used by Qian and Sejnowski (1988). Each amino acid is represented by 21 units, one unit for each amino acid and one unit to denote "off the end." Thus, if the window on the sequence includes 13 amino acids, there are 273 input units.

Table 10: Neural-network results for the secondary-structure prediction task.

Method	Accuracy	Number of Hidden Units	Window Size
Holley and Karplus (1989)	63.2%	2	17
Qian and Sejnowski (1988)	62.7%	40	13

Table 11: Selected machine learning results for the secondary-structure prediction task. Due to differing data sets and experimental methodologies, these numbers can only be roughly compared.

Method	Accuracy
Qian and Sejnowski (1988)	62.7%
Cost and Salzberg (1993)	65.1%
Zhang et al. (1992)	66.2%
Leng et al. (1993)	69.3%

periodically stop and test the network against the test proteins, retaining the highest results. I use a methodology in which the testset results are determined only after the network used in testing has been determined. Thus, my approach has no access to the test proteins, and my results for the method Qian and Sejnowski report will be somewhat lower than theirs.

Other machine learning results include several algorithms that employ (at least in part) case-based reasoning, including the PEBLs algorithm (Cost & Salzberg, 1993; Salzberg & Cost, 1992), Zhang et al.'s (1992) hybrid approach, and Leng et al.'s (1993) approach. Table 11 show that these approaches achieve significant improvement over approaches using only neural networks. Finally, the work of Rost and Sander (1993), which significantly reformulates the input representation of the task, produces 70.1% accuracy.

3.4.2 The Chou-Fasman Algorithm

The Chou-Fasman approach (1978) is to find amino acids that are likely to be part of α-helix and β-sheet regions, and then to extend these predictions to neighboring amino acids. Figure 17 provides a schematic overview of their algorithm.

The first step of the Chou-Fasman algorithm is to find *nucleation sites.* Nucleation sites are amino acids that are likely to be part of α-helix or β-sheet structures, based on their

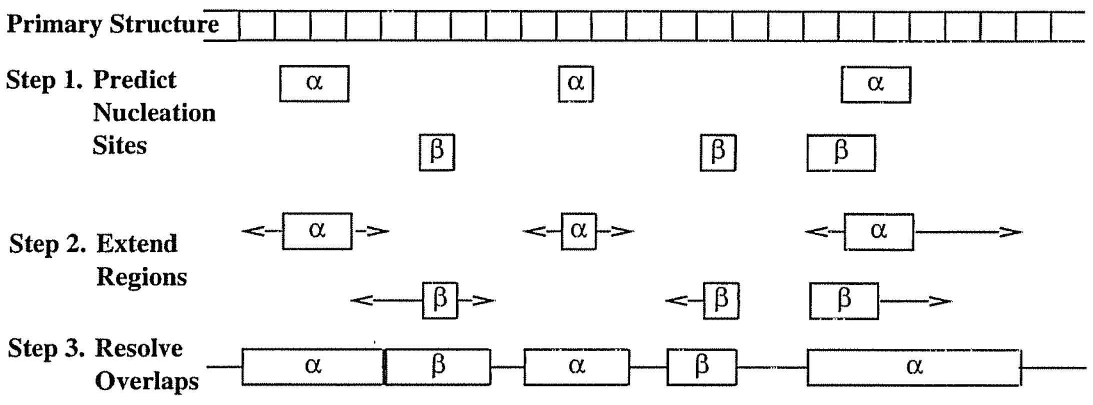


Figure 17: Steps of the Chou-Fasman (1978) algorithm.

neighbors and according to the conformation probabilities and rules reported by Chou and Fasman. Coil structures are predicted as a default; when an amino acid is not part of a helix or sheet structure, it is part of a coil structure.

To recognize nucleation sites, Chou and Fasman assign two *conformation* values to each of the 20 amino acids. The conformation values represent how likely an amino acid is to be part of either a helix or sheet structure, with higher values indicating greater likelihood. They also group the amino acids into classes of similar conformation value. The classes for helix are *formers, high-indifferent, indifferent,* and *breakers*; those for sheet are *formers, indifferent,* and *breakers* (see Table 12). An amino acid is an α-helix nucleation site if it is part of a sequence of six consecutive amino acids, where there are a total of four helix-formers (two high-indifferents count as one former) and fewer than two breakers. Similarly, an amino acid is a β-sheet nucleation site if it is in a sequence of five amino acids with at least three sheet-formers and fewer than two breakers.

The Chou-Fasman algorithm extends a helix or sheet region indefinitely until it encounters a breaker. An α-helix break region occurs when an helix-breaker amino acid is

Table 12: Assignment of the amino acids to α-helix and β-sheet former and breaker classes from Chou and Fasman (1978).

Class	α-helix	β-sheet
Former	E, A, L, H, M, Q, W, V, F	M, V, I, C, Y, F, Q, L, T, W
High-Indifferent	K, I	
Indifferent	D, T, S, R, C	A, R, G, D
Breaker	N, Y, P, G	K, S, H, N, P, E

immediately followed by either another helix-breaker or a helix-indifferent amino acid. A helix is also broken when encountering the amino acid Proline (P). The process of extending β-sheet structures works similarly.

To resolve overlaps, Chou and Fasman suggest that the conformation values of regions

Table 13: Former and breaker values for the amino acids.

helix_former(E) = 1.37	helix_former(A) = 1.29	helix_former(L) = 1.20
helix_former(H) = 1.11	helix_former(M) = 1.07	helix_former(Q) = 1.04
helix_former(W) = 1.02	helix_former(V) = 1.02	helix_former(F) = 1.00
helix_former(K) = 0.54	helix_former(I) = 0.50	
helix_former(*others*) = 0.00		
helix_breaker(N) = 1.00	helix_breaker(Y) = 1.20	helix_breaker(P) = 1.24
helix_breaker(G) = 1.38		
helix_breaker(*others*) = 0.00		
sheet_former(M) = 1.40	sheet_former(V) = 1.39	sheet_former(I) = 1.34
sheet_former(C) = 1.09	sheet_former(Y) = 1.08	sheet_former(F) = 1.07
sheet_former(Q) = 1.03	sheet_former(L) = 1.02	sheet_former(T) = 1.01
sheet_former(W) = 1.00		
sheet_former(*others*) = 0.00		
sheet_breaker(K) = 1.00	sheet_breaker(S) = 1.03	sheet_breaker(H) = 1.04
sheet_breaker(N) = 1.14	sheet_breaker(P) = 1.19	sheet_breaker(E) = 2.00
sheet_breaker(*others*) = 0.00		

I produced these values using the tables reported by Chou and Fasman (1978, p. 51). I normalized the values for formers by dividing the conformation value of the given former by the conformation value of the weakest former. So, for example, the helix former value of Alanine (A) is 1.29, since the helix conformation value of Alanine is 1.45 and the conformation value of the weakest helix former Phenylalanine (F) is 1.12. Breaker values work similarly except that the value used to calculate the breaker value is the multiplicative inverse of the conformation value.

I did not directly use the values of Chou and Fasman for two reasons. One, I wanted smaller values, to decrease the number of times three very strong helix-formers would add up to more than 4 (and similarly for sheets). Two, breaker conformation values tend to be numbers between 0 and 1 with the stronger breakers being close to 0. I wanted the breaker value to be larger the stronger the breaker, so I used the inverse of the breaker's conformation value (restricting the result to not exceed 2).

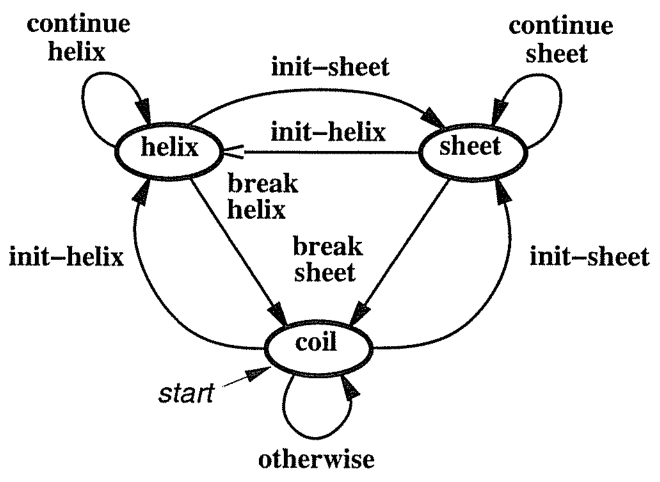


Figure 18: The finite-state automaton interpretation of the Chou-Fasman algorithm.

be compared. To do so, they assign formers and breaker values weighted by likelihood. In Table 13 I transform Chou and Fasman's table to one that I can use to both recognize nucleation sites and compare regions. The values in Table 13 are set so that if, for example, the sum of helix-former values across a sequence of six amino acids is four, then this means that there are four helix formers across this sequence (with high-indifferents counting as one-half). The resulting values can also be used to compare the strength of different combinations of formers. This is done in FSKBANN's networks by weighting the links from various amino acids by the values in Table 13. For example, a combination of four Alanines (`A`'s) will produce a higher activation of the `init_helix` unit than a combination of four Phenylalanines (`F`'s). I will present more details on how this is done in the next section.

3.4.3 The Chou-Fasman Algorithm as a Finite-State Automaton

The Chou-Fasman algorithm cannot be represented using propositional rules, since the prediction for an amino acid depends on the predictions for its neighbors (because nucleation sites are extended). However, one can represent the algorithm as an FSA (see Figure 18). The start state of the FSA is *coil.* To make predictions for a protein, the FSA scans the protein[1], with the input at each step including the amino acid being classified plus its neighbors. The Chou-Fasman domain theory bases each prediction on the current window of

[1]I actually scan each protein twice: from left-to-right and from right-to-left. I then sum the results so as to simulate extending nucleation sites in both directions. This is done so that I can extend nucleation sites in both directions.

Table 14: Rules provided with the Chou-Fasman FSA (see Figure 18) to form the Chou-Fasman domain theory. $x@N$ is true if x is the amino acid N positions from the one whose secondary structure the algorithm is predicting.

Rules for recognizing nucleation sites.

$$\begin{array}{rcl}
\mathit{init\_helix} & \leftarrow & \left(\sum_{pos=0}^{5} \mathit{helix\_former}(\mathit{amino\_acid}@pos)\right) > 4 \\
 & & \wedge \left(\sum_{pos=0}^{5} \mathit{helix\_breaker}(\mathit{amino\_acid}@pos)\right) < 2 \\
\mathit{init\_sheet} & \leftarrow & \left(\sum_{pos=0}^{4} \mathit{sheet\_former}(\mathit{amino\_acid}@pos)\right) > 3 \\
 & & \wedge \left(\sum_{pos=0}^{4} \mathit{sheet\_breaker}(\mathit{amino\_acid}@pos)\right) < 2
\end{array}$$

Rules for pairs of amino acids that terminate helix structures.

$$\begin{array}{rcl}
\mathit{helix\_break}@0 & \leftarrow & N@0 \vee Y@0 \vee P@0 \vee G@0 \\
\mathit{helix\_break}@1 & \leftarrow & N@1 \vee Y@1 \vee P@1 \vee G@1 \\
\mathit{helix\_indiff}@1 & \leftarrow & K@1 \vee I@1 \vee D@1 \vee T@1 \vee \\
 & & S@1 \vee R@1 \vee C@1 \\
\mathit{break\_helix} & \leftarrow & \mathit{helix\_break}@0 \wedge \mathit{helix\_break}@1 \\
\mathit{break\_helix} & \leftarrow & \mathit{helix\_break}@0 \wedge \mathit{helix\_indiff}@1
\end{array}$$

Rules for pairs of amino acids that terminate sheet structures.

$$\begin{array}{rcl}
\mathit{sheet\_break}@0 & \leftarrow & K@0 \vee S@0 \vee H@0 \vee N@0 \vee P@0 \vee E@0 \\
\mathit{sheet\_break}@1 & \leftarrow & K@1 \vee S@1 \vee H@1 \vee N@1 \vee P@1 \vee E@1 \\
\mathit{sheet\_indiff}@1 & \leftarrow & A@1 \vee R@1 \vee G@1 \vee D@1 \\
\mathit{break\_sheet} & \leftarrow & \mathit{sheet\_break}@0 \wedge \mathit{sheet\_break}@1 \\
\mathit{break\_sheet} & \leftarrow & \mathit{sheet\_break}@0 \wedge \mathit{sheet\_indiff}@1
\end{array}$$

Rules for continuing structures.

$$\begin{array}{rcl}
\mathit{cont\_helix} & \leftarrow & \neg P@0 \wedge \neg \mathit{break\_helix} \\
\mathit{cont\_sheet} & \leftarrow & \neg P@0 \wedge \neg E@0 \wedge \neg \mathit{break\_sheet}
\end{array}$$

amino acids plus the last prediction it made, maintained as a state.

As I noted before, the transitions in our FSA need not be input values, and indeed in this FSA they are not – the transitions are marked with propositions that I define to represent the notions of the Chou-Fasman algorithm (i.e. nucleation sites, amino acids that break sequences, etc.). Table 14 shows the rules used to define the propositions in the Chou-Fasman FSA. FSKBANN augments these rules with rules that represent the states and transitions from the FSA in Figure 18; Table 15 shows these extra rules.

One further extension I made to translate this domain theory was to introduce a method to deal with the rules indicating the strength of the initial nucleation sites – the rules for

Table 15: Rules derived from the Chou-Fasman FSA (see Figure 18).

$$\begin{array}{rcll}
helix_i & \leftarrow & sheet_{i-1} & \wedge\ init\_helix \\
helix_i & \leftarrow & coil_{i-1} & \wedge\ init\_helix \\
helix_i & \leftarrow & helix_{i-1} & \wedge\ cont\_helix \\
\\
sheet_i & \leftarrow & helix_{i-1} & \wedge\ init\_sheet \\
sheet_i & \leftarrow & coil_{i-1} & \wedge\ init\_sheet \\
sheet_i & \leftarrow & sheet_{i-1} & \wedge\ cont\_sheet \\
\\
coil_i & \leftarrow & helix_{i-1} & \wedge\ break\_helix \\
coil_i & \leftarrow & sheet_{i-1} & \wedge\ break\_sheet \\
coil_i & \leftarrow & coil_{i-1} & \wedge\ any
\end{array}$$

init_helix and *init_sheet* in Table 14. To translate these rules, FSKBANN creates a unit to represent the summation that has a link to each amino acid that may contribute to the sum. Each of these links is given a weight equal to the standard weight for a positive link

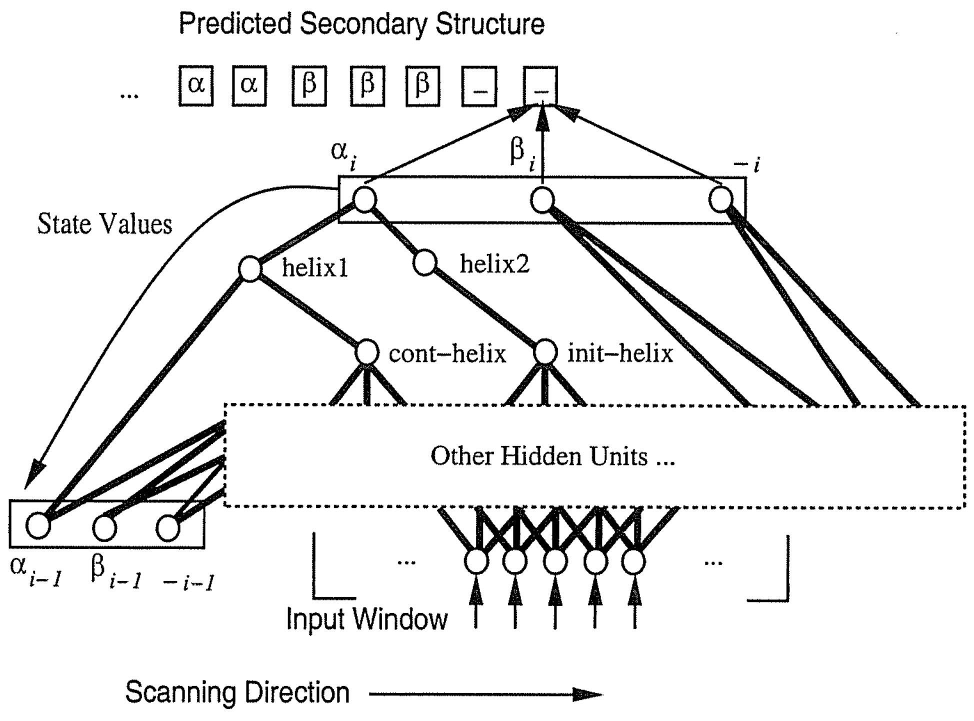


Figure 19: General neural-network architecture used to represent the Chou-Fasman algorithm. Note that the low-weighted links added at the end of the translation process are not shown.

multiplied by the "former" value from Table 13 for that amino acid. So, for example, in summing the helix former values, the link to Alanine at the center of the window (A@0), would be set to 5.16 – the standard positive link weight, 4, times 1.29, the "former" value for Alanine. I then set the bias of the resulting unit to: -4 ($threshold - 0.5$), where threshold is the total the summation must exceed, and 4 is the standard positive link weight used in KBANN and FSKBANN to represent important dependencies.

Figure 19 shows an outline of the network that FSKBANN produces. The FSKBANN network is similar to the standard network for this task shown in Figure 16, but with two major differences. One, the input to the network includes the state values – the predictions made by the network in the previous step. Two, the topology of the hidden units is determined by the rules implementing the Chou-Fasman algorithm.

3.4.4 Methodology

The experiments I performed to evaluate FSKBANN use the data set from Qian and Sejnowski (1988). Their data set consists of 128 segments from 106 proteins with a total of 21,623 amino acids, for an average length of 169 amino acids per segment. Of these amino acids, 54.5% are part of coil structures, 25.2% part of α-helix structures, and 20.3% part of β-sheet structures. I randomly divided the proteins ten times into disjoint training and test sets, which contained two-thirds (85 proteins) and one-third (43 proteins) of the original proteins, respectively.

I used backpropagation (Rumelhart et al., 1986) to train neural networks for *two* learning approaches, our FSKBANN approach and a standard neural-network approach similar to Qian and Sejnowski's (which I will refer to as *standard ANNs*). I terminated training using *patience* as a stopping criterion (Fahlman & Lebiere, 1990). The patience criterion states that training should continue until the error rate on the training set has not decreased for some number of training cycles. For this study I set the number of epochs to be four, a value I determined by empirical testing.

In order to avoid overfitting in this task, I employ the technique of maintaining a validation set. I use the patience criterion discussed above to decide how many training cycles (epochs) to perform, since it can take a significant amount of training before perfect performance on the training set is achieved. I use the validation set to select the "best" network (according to the validation set) from the networks produced at the end of each training cycle. As part of selecting a validation set, I added the criterion that the validation set should be a "representative" validation set; I accepted a validation set as representative if the percentages of each type of structure (α, β, and coil) in the validation set roughly approximate the percentages of all the training proteins. Note that I did not consider the *testing* set when computing the percentages. Through empirical testing, I found that a validation set size of five proteins achieves the best results for both FSKBANN and ANNs.

FSKBANN creates a network with 28 hidden units to represent the Chou-Fasman domain theory. Qian and Sejnowski report that their networks generalized best when they had 40 hidden units. Using the methodology outlined above, I compared standard ANNs containing 28 and 40 hidden units. I found that networks with 28 hidden units generalized slightly better; hence, for experiments I use 28 hidden units in my standard ANNs. This has the added advantage that the FSKBANN and standard networks contain the same number of hidden units.

3.4.5 Results

Table 16 contains results averaged over the 10 test sets. The statistics reported are the percent accuracy overall, the percent accuracy by secondary structure, and the correlation coefficients for each structure[2]. The correlation coefficients are good for evaluating the effectiveness of the prediction for each of the three classes separately. The resulting gain in overall accuracy for FSKBANN over both standard ANNs and the non-learning Chou-Fasman method is statistically significant at the 0.5% level (i.e. with 99.5% confidence) using a t-test. This demonstrates that we can use FSKBANN to effectively refine a procedural domain theory.

The gain in accuracy for FSKBANN over the Chou-Fasman algorithm is fairly large and exhibits a corresponding gain in all three correlation coefficients. It is interesting to note that the FSKBANN and Chou-Fasman solutions produce approximately the same accuracy for β-sheets, but the correlation coefficients demonstrate that the Chou-Fasman algorithm achieves this accuracy by predicting a much larger number of β-sheets.

The apparent gain in accuracy for FSKBANN over ANN networks appears fairly small

[2]The following formula defines the correlation coefficient for the secondary-structure task (Mathews, 1975):

$$C = \frac{PN - FM}{\sqrt{(P + F)(P + M)(N + F)(N + M)}} \tag{12}$$

where C is calculated for each structure separately, and P, N, F, and M are the number of true positives, true negatives, false positives, and misses for each structure, respectively.

Table 16: Results from different prediction methods.

	Testset Accuracy				Correlation Coefficients		
Method	Total	Helix	Sheet	Coil	Helix	Sheet	Coil
Chou-Fasman	57.3%	31.7%	36.9%	76.1%	0.24	0.23	0.26
Standard ANN	61.8	43.6	18.6	86.3	0.35	0.25	0.31
FSkBANN	63.4	45.9	35.1	81.9	0.37	0.33	0.35
ANN (w/state)	61.7	39.2	24.2	86.0	0.32	0.28	0.31

(only 1.6 percentage points), but this number is somewhat misleading. The correlation coefficients give a more accurate picture. They show that the FSkBANN does better on both α-helix and coil prediction, and much better on β-sheet prediction. The reason that the ANN solution still does fairly well in overall accuracy is that it predicts a large number of coil structures, the largest class, and does very well on these predictions.

Also shown in Table 16 are results for ANNs that included state information – networks similar to Qian and Sejnowski's but in which the previous output values are included as state information for the next step – so that they are SRNs. These results show that state information alone is not enough to increase the accuracy of the network prediction.

To evaluate the usefulness of the domain theory as a function of the number of training examples and to allow me to estimate the value of collecting more proteins, I performed a second series of tests. I divided each of the training sets into four subsets: the first contained the first 10 of the 85 proteins, the second contained the first 25, the third contained the first 50, and the fourth had all 85 training proteins. This process produced 40 training sets. I then used each of these training sets to train both the FSkBANN and ANN networks. Figure 20 contains the results of these tests. For comparison purposes I also show the generalization accuracy of the non-learning Chou-Fasman method. FSkBANN shows a gain in accuracy for each training set size (statistically significant at the 5% level, i.e., with 95% confidence).

The results in Figure 20 demonstrate two interesting trends. One, the FSkBANN networks do better no matter how large the training set, and two, the shape of the curve indicates that accuracy might continue to increase if more proteins were used for training. The one anomaly for this curve is that the gain in accuracy of FSkBANN over standard ANNs with 10 training proteins is not very large. One would expect that when the number of training instances is very small, the domain knowledge would be a big advantage. The problem here is that for a small training set it is possible to obtain random sets of proteins that are not very indicative of the overall population. Individual proteins generally do not reflect the overall distribution of secondary structures for the whole population; many proteins have

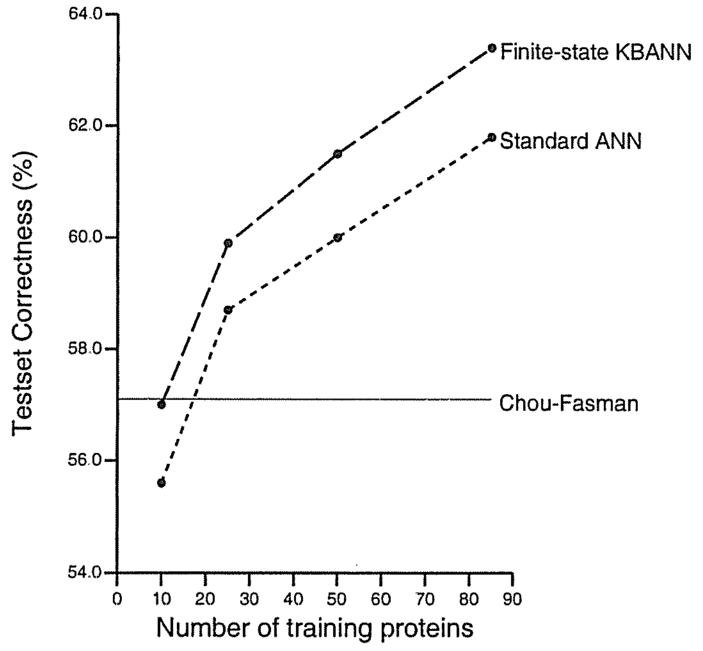


Figure 20: Percent correctness on test proteins as a function of training-set size.

large numbers of α-helix regions and almost no β-sheets, while others have large numbers of β-sheet regions and almost no α-helices. Thus in trying to learn to predict a very skewed population, the network can produce a poor solution. This is mitigated as more proteins are introduced, causing the training population to more closely match the overall population.

Finally, to analyze the detailed performance of the various approaches, I gathered a number of additional statistics concerning the FSKBANN, ANN, and Chou-Fasman solutions. These statistics analyze the results in terms of *regions*. A *region* is a consecutive sequence of amino acids with the same secondary structure. I consider regions because the measure of accuracy obtained by comparing the prediction for each amino acid does not adequately capture the notion of secondary structure as biologists view it (Cohen et al., 1991). For biologists, knowing the number of regions and the approximate order of the regions is nearly as important as knowing exactly the structure within which each amino acid lies. Consider the two predictions in Figure 21 (adapted from Cohen et al., 1991). The first prediction completely misses the third α-helix region, so it has four errors. The second prediction is slightly skewed for each α-helix region and ends up having six errors, even though it appears to be a better answer. The statistics I have gathered try to assess how well each solution does predicting α-helix regions (Table 17) and β-sheet regions (Table 18).

Table 17 and Table 18 give a picture of the strengths and weakness of each approach.

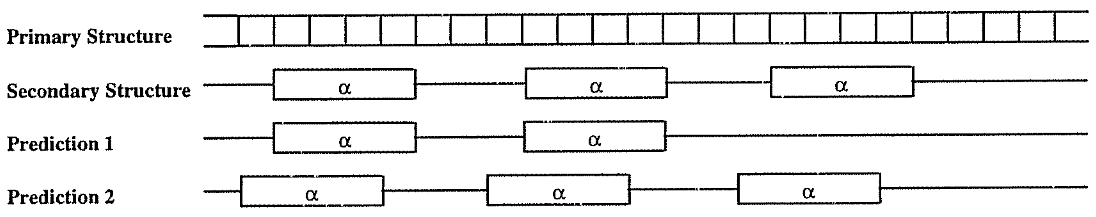


Figure 21: Two possible predictions for secondary structure.

Table 17: Region-oriented statistics for α-helix prediction.

Occurrence		Description	FS KBANN	ANN	Chou–Fasman
Actual	a -helix	Average length of an actual helix region (number of regions).	10.17 (1825)	10.17 (1825)	10.17 (1825)
Predicted	a -helix	Average length of a predicted helix region (number of regions).	8.52 (1774)	7.79 (2067)	8.00 (1491)
Actual Predicted	a -helix a -helix	Percentage of time an actual helix region is overlapped by a predicted helix region (length of overlap).	67% (6.99)	70% (6.34)	56% (5.76)
Actual Predicted	other a -helix	Percentage of time a predicted helix region does not overlap an actual helix region.	34%	39%	36%

Table 17 shows that the FSKBANN solution overlaps slightly fewer actual α-helix regions than the ANNs, but that these overlaps tend to be somewhat longer. On the other hand, the FSKBANN networks *overpredict* fewer regions than ANNs (i.e., predict fewer α-helix regions that do not intersect actual α-helix regions). Table 17 also indicates that FSKBANN and ANNs more accurately predict the occurrence of regions than Chou-Fasman does.

Table 18 demonstrates that FSKBANN's predictions overlap a much higher percentage of actual β-sheet regions than either the Chou-Fasman algorithm or ANNs alone. The overall accuracy for β-sheet predictions is approximately the same for FSKBANN and the Chou-Fasman method because the length of overlap for the Chou-Fasman method is much longer than for FSKBANN (at the cost of predicting much longer regions). The ANN networks do extremely poorly at overlapping actual β-sheet regions. The FSKBANN networks do as well as the ANNs at not overpredicting β-sheets, and both do better than the Chou-Fasman

Table 18: Region-oriented statistics for β-sheet prediction.

Occurrence		Description	FS KBANN	ANN	Chou–Fasman
Actual	b–strand	Average length of an actual strand region (number of regions).	5.00 (3015)	5.00 (3015)	5.00 (3015)
Predicted	b–strand	Average length of a predicted strand region (number of regions).	3.80 (2545)	2.83 (1673)	6.02 (2339)
Actual Predicted	b–strand b–strand	Percentage of time an actual strand region is overlapped by a predicted strand region (length of overlap).	54% (3.23)	35% (2.65)	46% (4.01)
Actual Predicted	other b–strand	Percentage of time a predicted strand region does not overlap an actual strand region.	37%	37%	44%

method. Taken together, these results indicate that the FSKBANN solution does significantly better than the ANN solution predicting β-sheet regions without having to sacrifice much accuracy in predicting α-helix regions.

Overall, the results in Tables 17 and 18 suggest that more work needs to be done developing methods of evaluating solution quality. A simple position-by-position count of correct predictions does not capture adequately the desired behavior. Solutions that find approximate locations of α-helix and β-sheet regions and those that accurately predict all three classes should be favored over solutions that only do well at predicting the largest class.

3.4.6 Discussion

The main conclusion I draw from my experiments is that they support my general thesis – that a method for refining procedural domain theories can produce improvements similar to those that have been observed for non-procedural domain theories. In particular, my experiments demonstrate that after refining the Chou-Fasman algorithm, the resulting refined algorithm is more accurate than both the original algorithm and a standard neural-network approach. Thus my experiments show that the hybrid approach of learning from both theory and data is more powerful than either approach separately.

3.5 Limitations and Future Directions

While FSKBANN does produce gains in performance for the secondary-structure task, these gains have since been eclipsed by other techniques. Two conclusions stand out from more recent research on the secondary-structure task. One is that case-based reasoning methods (Cost & Salzberg, 1993; Leng et al., 1993; Zhang et al., 1992), which look for similar sequences that are already characterized in the training data, are very effective. This suggests that the best approach to this problem might be to build up a large set of known structures and simply compare the new primary structure to old ones to find a secondary-structure mapping. This is borne out by current biological approaches, where researchers look for matches to existing structures in determining new structures.

A second point to note is that significant gains have been achieved by reformulating the input information (as in Rost & Sander, 1993). This suggest that the input information we are currently using may be inadequate to produce a good solution. Considering both of these points, in order to achieve better results for this particular task, I would want to change the input description, using one more like Rost and Sander's, and I would want to incorporate some of the information that case-based reasoning systems use.

With regard to FSKBANN, I would conclude that my experiments do validate my general thesis that it is possible to refine procedural domain theories. This then leaves the question of how applicable FSKBANN is to other real-world tasks. In Table 5, I defined the type of task to which FSKBANN is applicable, and I indeed conclude that FSKBANN could, in theory, work for any problem of this type. The main difficulty with using FSKBANN is its requirement that the user develop a finite-state automaton.

The ability to refine a domain theory containing a finite-state automaton is FSKBANN's main strength, but this also makes it difficult to use, since it may be hard for a relatively novice user to construct a finite-state automaton. Thus it may be advantageous to examine a communication method that allows the user to interact with the learner in a more natural manner, an area that I explore in the following chapters.

Other papers covering the FSKBANN system include Maclin and Shavlik (1991, 1992, 1993, 1994b).

Chapter 4

Advising a Reinforcement Learning Agent – The RATLE System

In this chapter, I present an overview of my system that allows a teacher to provide advice to a reinforcement learning (RL) agent. I call the system RATLE, for **R**einforcement and **A**dvice-**T**aking **L**earning **E**nvironment. The teacher in RATLE observes the behavior of the RL agent, and when she wishes, provides advice using the RATLE advice language. The teacher uses the advice language to give the RL agent recommendations about actions the agent might take under certain circumstances. Once the teacher provides the advice, RATLE translates the advice into a form that the RL agent understands, and then RATLE incorporates it into the RL agent. The agent then returns to reinforcement learning until the teacher provides more advice. Below I discuss my motivation for selecting reinforcement learning as an area to explore, and then describe the important features of RATLE.

I chose to develop a method for instructing an RL agent for several reasons. One reason was my experience with the FSKBANN system. Recall that one limitation of FSKBANN is that it requires the teacher to explicitly define state information in the form of FSAs. In RATLE I developed a language that allows the teacher to reference state information in a natural manner. Reinforcement learning is also interesting because the tasks in RL are inherently sequential, and therefore match my thesis focus. Finally, I selected RL because it is a successful and increasingly popular method for creating intelligent agents (Barto et al., 1990; Barto et al., 1995; Lin, 1992; Mahadevan & Connell, 1992; Tesauro, 1992; Watkins, 1989).

The major drawback of RL is its need for large numbers of training episodes. My work addresses this drawback – instructions from a teacher can (when the instructions are useful) reduce the number of training episodes that the agent needs to learn its policy function. In Figure 22, I show the general structure of a reinforcement learner (from Figure 9), augmented (in bold) with my process that allows the teacher to instruct an RL agent.

In the remainder of this chapter, I present RATLE, my system for implementing the process

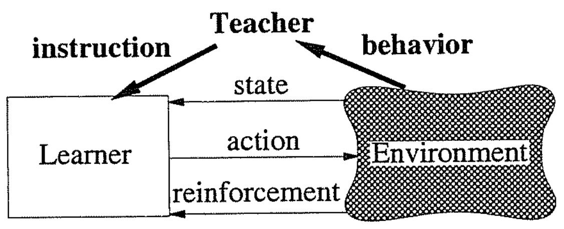


Figure 22: Reinforcement learning with a teacher.

shown in bold in Figure 22. I first discuss the salient aspects of the language that RATLE uses to represent instructions. Then I outline a framework for using instruction from Hayes-Roth, Klahr, and Mostow (1981), and discuss how RATLE fits into this framework. In Chapter 5 I completely define the RATLE language that the teacher uses to articulate instructions. Chapter 6 details exactly how the constructs of the RATLE language are translated into knowledge that the agent can use.

4.1 Overview of RATLE

A common method by which a reinforcement learner improves its performance is to first make a prediction about the utility of an action (how much reward it receives by taking that action), and then obtain an estimate of the actual utility of the action by executing that action and predicting the utility for the resulting state. The learner uses the predicted and estimated utility values to update its prediction function. The learner is thus continuously executing an exploration cycle involving predictions and actions. To this cycle, RATLE introduces a separate process involving a teacher, as shown in Figure 23.

The RL agent in Figure 23 performs standard connectionist Q-learning when it is not receiving instruction. The human teacher in Figure 23 observes the agent and forms her instructions using her knowledge about the task. The teacher's advice-generation process is outside the scope of this thesis. The critical contribution of RATLE is the flow of knowledge from the teacher to the agent. As I discuss below, there are two main issues regarding this flow of instruction: (1) what language the teacher uses to present instructions; (2) how those instructions translate into knowledge that the agent assimilates.

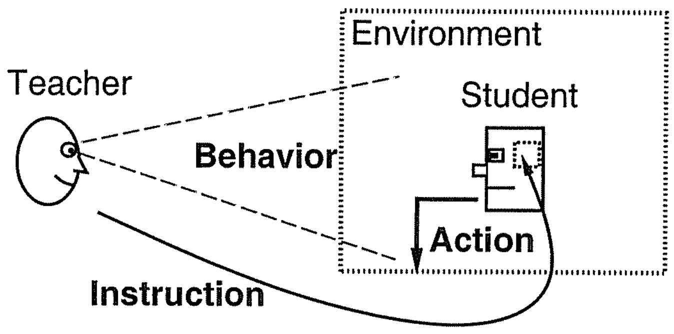


Figure 23: The interaction of the teacher and agent in RATLE. The process is a cycle: the observer watches the agent's behavior to determine what advice to give, and the advice-taking system processes the advice and inserts it into the agent's "brain", which changes the agent's behavior. The agent operates as a normal Q-learning agent when not being instructed by its teacher. (This figure is a duplicate of Figure 4.)

4.1.1 Features of RATLE's Instruction Language

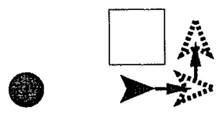
As I argued in Chapter 1, a key aspect of the interaction between a teacher and a student is the language the teacher uses to provide instructions. In RATLE, the teacher is advising an RL agent; therefore the language naturally has constructs that relate to the type of decisions an RL agent makes – which action to select given the current environment. Instructions in RATLE take the form of simple programming language constructs.

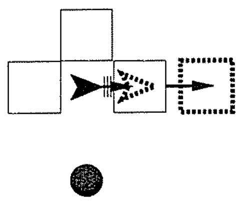
The RATLE language resembles a programming language by design. I use a programming language because it allows me to make use of standard parsing techniques. Thus, I can focus on the process of transferring knowledge to an RL agent, rather than trying to address the natural language problem. I also make use of fuzzy-knowledge ideas (Berenji & Khedkar, 1992; Zadeh, 1965) to permit terms such as "Near" and "East" in the instructions in Table 19. These terms make the resulting language easier to use for the human teacher. RATLE's advice language is not as powerful as English, but it does allow many different types of complex instructions to be articulated quite naturally.

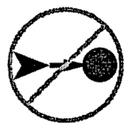
RATLE allows a user to give simple instructions in the form of IF and REPEAT statements specifying conditions of the world and actions to be taken under those conditions. These types of instructions closely correspond to the knowledge that an RL agent is trying to acquire, but in a manner that is "natural" to the teacher. An important feature of the language is that the teacher does not have to understand the reinforcement learning process the agent is executing. The teacher only needs to know how to phrase her instructions in the language I provide – the RATLE system is responsible for translating these instructions

Table 19: Samples of advice in RATLE's instruction language.

Instruction	English Version	Pictorial Version

IF An Enemy IS (Near AND West) AND
  An Obstacle IS (Near AND 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 AND
    OKtoPushEast AND
    An Enemy IS Near
REPEAT
    MULTIACTION
      PushEast
      MoveEast
    END
UNTIL NOT OKtoPushEast OR
    NOT 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.	

IF An Enemy IS (Near AND East)
THEN
  DO_NOT MoveEast
END;
	Do not move toward a nearby enemy.	

into a form that the agent can use.

Table 19 shows some sample instructions a teacher might provide to an agent learning to play a video game. The left column contains the instructions expressed in RATLE's programming language, the center column describes them in English, and the right column illustrates the instructions pictorially (see Chapter 7 for more details on this environment).

One key feature of RATLE's language is that it allows the teacher to specify multi-step *plans* in her instructions; both the first and second instruction in Table 19 are instances of multi-step instructions. As I discussed in Chapters 1 and 2, it is natural for a teacher to specify a *sequence* of steps as a solution to complex planning tasks, such as those addressed in RL.

Implementing a multi-step plan for a learner that thinks in terms of the next step means that the learner needs a mechanism to remember the current step. RATLE automatically constructs state units to implement these multi-step plans. The teacher does not need to understand the mechanism that RATLE is using to implement her instructions.

A related feature of RATLE's language is that it allows the user to specify instructions that contain loops. This feature lets the teacher specify instructions for repeated actions that are very natural in sequential situations (e.g., continuing to drive forward until the next red light is encountered). The second instruction in Table 19 has a looping statement. As with multi-step plans, RATLE uses state units to represent loops.

A final important aspect of RATLE that makes it different from my previous approach (FSKBANN) and other approaches (Lin, 1992; Omlin & Giles, 1992; Towell et al., 1990) is that the teacher can continue to watch the performance of the RL agent and then provide *more* instructions to the agent. RATLE translates instructions into *additions* to the RL agent's current knowledge base. Thus, the teacher can provide instructions multiple times, since each time the instructions simply augment the agent's current knowledge.

An advantage of the continuous nature of the teaching process in RATLE is that the teacher can present instructions that address only one aspect of a task at a time, rather than trying to present a domain theory containing all of the knowledge the student needs. The teacher can observe the behavior of the agent before providing instructions; thus the teacher need only address those aspects of the task in which the agent is deficient. The teacher can also present instructions that address shortcomings of her previous instructions.

Given the more limited nature of instruction in RATLE, I will not refer to a set of instructions in RATLE as a domain theory, but as a piece of *advice*. Once the teacher develops a piece of advice, it is RATLE's job to translate that statement into a form that the agent is able to use. To outline this translation process I will next present a framework for advice-taking developed by Hayes-Roth, Klahr, and Mostow (1981).

4.1.2 A General Framework for Advice-Taking

Recognition of the value of advice-taking has a long history in AI. The general idea of an agent accepting instruction 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 instructions 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, 1994a; Noelle & Cottrell, 1994).

Hayes-Roth, Klahr, and Mostow (1981)[1] developed the following framework for advice-taking:

Step 1. Request/receive the advice.
Step 2. Convert the advice to an internal representation.
Step 3. Convert the advice into a usable form.
Step 4. Integrate the reformulated advice into the agent's knowledge base.
Step 5. Judge the value of the advice.

In order to better explain RATLE's process of advice translation I will outline how RATLE fits into this framework.

Step 1. Request/receive the advice. To begin the process of advice-taking, someone must decide that advice is needed. Often, approaches to advice-taking focus on having the learner ask for instruction when help is needed (Clouse & Utgoff, 1992; Whitehead, 1991). In RATLE I take the opposite tack – the teacher presents advice when she feels it is appropriate. There are two reasons for this: (1) it places less of a burden on the teacher since she need only provide advice when she chooses; (2) how to create the best mechanism for having an agent recognize (and express) its need for advice is an open question. The teacher in RATLE observes the behavior of the agent and then formulates statements in RATLE's advice language to address the limitations she has observed in the agent. An advantage of this approach is that she will hopefully have insights about problems that the agent does not perceive, and therefore the advice will be especially useful.

Step 2. Convert the advice to an internal representation. Once the teacher has created a piece of advice, RATLE parses it using the tools *lex* and *yacc* (Levine et al., 1992).

Step 3. Convert the advice into a usable form. Other techniques, such as *knowledge compilation* (Dietterich, 1991), convert ("operationalize") high-level instructions into a (usually larger) collection of directly interpretable statements (Gordon & Subramanian, 1994; Kaelbling & Rosenschein, 1990; Nilsson, 1994). For example, in chess an agent that is receiving help about what to do when it is in "check" could convert that statement into knowledge about what to do when the king is in check from a queen, from a rook, etc.

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

After parsing advice, RATLE transforms the general advice into terms that it can directly understand. In RATLE I address only a limited form of operationalization, namely the concretization of ill-defined terms such as "near" and "many." Terms such as these allow the teacher to provide natural, yet partially vague, instructions and eliminate the need for her to fully understand the learner's input representation.

In Chapter 5, I present the RATLE language, and also I describe my language for defining the fuzzy terms.

Step 4. Integrate the reformulated advice into the agent's knowledge base. In RATLE I employ a connectionist approach to RL (Anderson, 1987; Barto et al., 1983; Lin, 1992). Hence, to incorporate the teacher's advice, the agent's neural network must be updated. As in FSKBANN, I expand on the basic KBANN algorithm (Towell et al., 1990) to install the advice into the agent.

Figure 24 illustrates my basic approach for adding advice into the reinforcement learner's action-choosing network. This network computes a function that maps sensor readings to the utility of actions. Incorporating instructions involves adding new hidden units that represent the instructions to the existing neural network. Since each piece of advice involves adding units to the existing network, this process can be repeated any number of times; thus the agent can continuously incorporate new instructions.

Step 5. Judge the value of the advice. The final step of Hayes-Roth et al.'s advice-taking process is to evaluate the advice. There are two perspectives to this process: (1) that

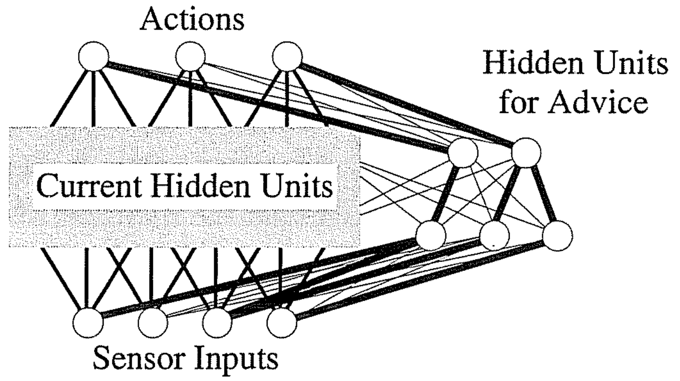


Figure 24: Adding advice to the RL agent's neural network by creating new hidden units that represent the advice. The thick links on the right capture the semantics of the advice. The added thin links initially have near-zero weight; during subsequent backpropagation training the magnitude of their weights can change, thereby refining the original advice. Details and examples appear in Chapter 6.

of the learner, who must decide if an instruction is useful; (2) that of the teacher, who must decide whether an instruction had the desired effect on the behavior of the learner. The learner evaluates advice by operating in its environment and changing its policy function, including the advice, with Q-learning. The feedback provided by the environment offers a crude measure of the quality of the instruction. (One can also envision that in some circumstances – such as a game-learner that can play against itself (Tesauro, 1992) or an agent that builds an internal world model (Sutton, 1991) – it would be straightforward to empirically evaluate the new advice.) The teacher judges the value of her statements similarly (i.e., by watching the learner's post-advice behavior). This may lead to the teacher giving further instructions, thereby restarting the cycle.

4.2 Summary

The RATLE system allows a teacher to instruct a reinforcement-learning agent. The language the teacher uses is a simple programming language that also makes use of fuzzy terms. The programming-language constructs allow the teacher to express knowledge about conditions of the world, along with plans that the agent should follow given those conditions. Thus, the teacher is able to express advice that is closely related to the knowledge the RL agent is trying to acquire. Importantly, the teacher does not have to understand the internal mechanisms of the RL agent or RATLE in order to provide useful instruction. The RATLE language allows the teacher to give instruction in the form of plans involving sequences of steps as well as loops; RATLE makes use of a memory mechanism to implement these plans in the RL agent. The resulting language, while not as powerful as English, is still powerful enough to express a wide range of advice.

In order to incorporate the advice provided by the teacher, RATLE performs a sequence of steps to transform each statement into a form that the RL agent can use. Figure 25 shows the cyclic interaction of the teacher and agent, with the steps RATLE uses to transform instructions shown at the bottom. This translation process follows Hayes-Roth, Klahr and Mostow's (1981) framework for advice taking. First, a decision must be made that advice is needed. In RATLE, the *teacher* decides when to give advice, which means that the teacher only need intervene when she feels it is appropriate. Then RATLE parses the instructions, which is straightforward since the RATLE language was designed to be easy to parse using existing software tools. In the third step, RATLE operationalizes any fuzzy terms the teacher uses in her instructions into terms the RL agent can understand. Next, RATLE translates each

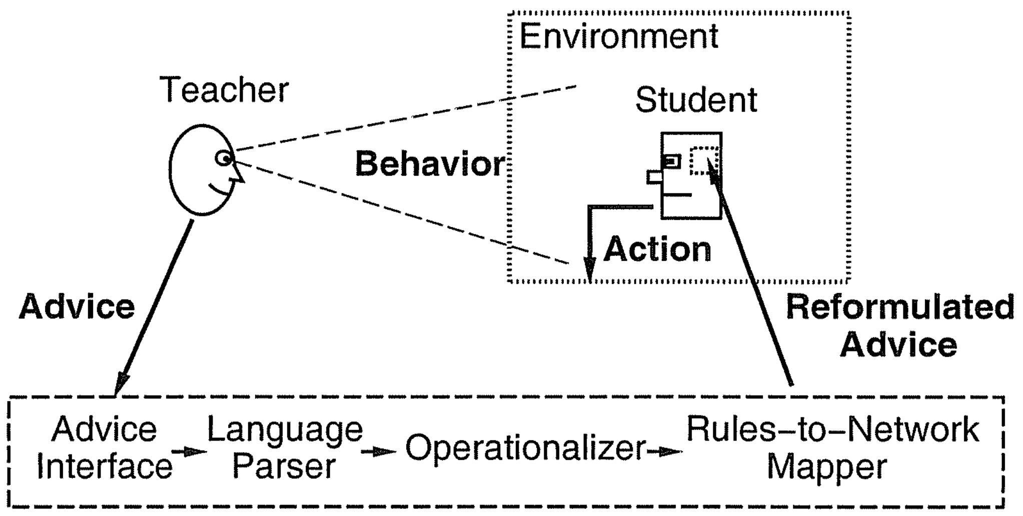


Figure 25: The interaction of the teacher, agent, and advice-taking components of RATLE. Advice developed by the teacher is transformed using a process based on Hayes-Roth, Klahr, and Mostow's (1981) advice-taking formalism.

statement into additions to the agent's neural network that capture the knowledge expressed by the statement, then inserts these additions into the agent's current network. Finally, both the agent and teacher evaluate the advice – the agent by exploring its environment and seeing how well the advice works in practice, and the teacher by watching the agent's resulting behavior to see if any further instruction is warranted. When the teacher is not providing instruction, the RATLE agent performs standard connectionist Q-learning, until the advice-taking process resumes.

Other papers covering the RATLE system include Maclin and Shavlik (1994a, 1996) and Shavlik and Maclin (1995).

Chapter 5

The RATLE Advice Language

The RATLE advice language allows a teacher to communicate her instructions to a reinforcement-learning agent. Using the RATLE language, the teacher suggests an appropriate action (or series of actions) that the agent should take under certain conditions. While the RATLE language is not a panacea for instruction, it does permit the teacher to express a wide range of possible instructions. This chapter presents the complete description of RATLE's advice language.

Advice in the RATLE language consists of a set of simple statements, similar to Pascal (Jensen & Wirth, 1975) programming statements. Recall that I use a programming language rather than natural language because this greatly simplifies the task of parsing the language and lets me focus on the process of transferring advice to the agent.

Statements in the RATLE advice language specify conditions that must be met in order for certain actions to be taken. Conditions are logical combinations of input and intermediate terms, as well as fuzzy conditions (Zadeh, 1965). The teacher can use these conditions to define particular states of the world to signal actions the agent should take. Actions include simple actions, action prohibitions, and multi-step plans; they can be performed separately or in loops. Statements combine conditions and actions into simple IF-THEN clauses and into more complex looping constructs.

Before the teacher can provide instructions to the RL agent performing a task in a given environment, a number of aspects of the task and environment must be defined by a person whom I will refer to as the *initializer*. The initializer need not be the teacher, though he or she could be. The initializer is responsible for defining the set of inputs that the agent sees (i.e., the agent's sensors) and the set of actions the agent may perform. Each input feature is labelled by the initializer using the input-definition language defined in Sections 5.6. Input features are given names and are sometimes given properties by the initializer. Each action is also given a unique name by the initializer. The initializer also defines a set of fuzzy terms that the teacher may use in her instructions. These fuzzy terms are generally specific to the task the agent is addressing, and make instruction-giving easier for the teacher, since she can use imprecise words such as "big" and "near." These fuzzy terms are created using the fuzzy-term language shown in Section 5.7. After creating the input, action, and fuzzy terms, the initializer gives the available task-specific vocabulary to the teacher.

Once the initializer completes his or her work, the teacher begins to observe the RL agent as it explores the task. Whenever the teacher chooses, she may provide instructions to the agent using statements in the advice language defined below.

In the following sections I present the set of allowable statements in the RATLE advice language, then the conditions and actions the teacher may use in a statement. I also discuss some limitations and future directions for the advice language. I conclude by showing the input, action, and fuzzy-term languages the initializer uses to configure RATLE for the teacher.

5.1 The Basic Pieces: Statements

The teacher uses a statement in the RATLE advice language to indicate some condition of the world the agent might see, plus some action or set of actions the agent should take given that condition. Each lesson provided by the teacher has one or more such statements. The RATLE language includes three types of statements: the IF statement, the REPEAT statement, and the WHILE statement. These statements make use of conditions and actions defined in Sections 5.2 and 5.3.

5.1.1 The IF Statement

```
IF Condition THEN
    [ INFER | REMEMBER ] IfConclusion
[ ELSE
    [ INFER | REMEMBER ] ElseConclusion ]
END
```

The IF statement[1] lets the teacher provide straightforward advice: under a certain condition

[1]In defining the constructs of the RATLE language, I make use of certain standard conventions. Square brackets ([and]) represent optional parts of constructs, while curly brackets ({ and }) are used to group parts of the grammar. A vertical line (|) represents alternation, and the plus (+) and star (*) characters indicate that a grammar part can be repeated one or more times and zero or more times, respectively. All punctuation characters of the RATLE language are surrounded by quotes (" and ").

the agent should reach some conclusion. The RATLE language allows a teacher to specify three different types of conclusions: (1) the name of an action for the agent to take; (2) an intermediate term (using the keyword INFER); or (3) the name of a condition to remember (indicated by the keyword REMEMBER). The first type of conclusion in an IF statement is very close to the type of knowledge the RL agent is trying to learn – a function that predicts the utility of actions so that it will know what action to take in each given state. That is, a teacher's recommendation of an action under some condition suggests that the action has high utility under that condition. An INFER conclusion allows the teacher to define intermediate terms that represent logical combinations of the input features and other intermediate terms. Allowing the teacher to create intermediate terms makes her job simpler, since she can build terms that can be used multiple times in other pieces of advice. The teacher uses the REMEMBER keyword to suggest a term that the agent should remember for use at the next step (using a state unit as in Chapter 3's FSKBANN).

An IF statement works in the obvious way – when *Condition* is true, the advice suggests that *IfConclusion* be taken, otherwise the advice suggests the (optional) *ElseConclusion.* We could imagine a sample piece of advice for the children's game *Red Light – Green Light*, which would look something like this:

```
IF LightIsRed THEN
   StayStill
ELSE
   MoveForward
END
```

LightIsRed describes a condition of the world and StayStill and MoveForward are actions the agent might take given the current state of the light. In this IF statement, RATLE assumes that StayStill and MoveForward are the names of actions because the keywords INFER and REMEMBER were left out.

The teacher uses the keyword INFER to indicate that a conclusion of an IF STATEMENT is an intermediate term. For example, imagine that the input vector includes Boolean features that indicate whether the object in view is Small, Medium or Large; the teacher can use an IF statement and the keyword INFER to create a new term, NotLarge, as follows:

```
IF Small OR Medium THEN
   INFER NotLarge
END
```

The teacher can then use the new term NotLarge in future statements.

The teacher can also use the keyword INFER to add a new definition of an existing intermediate term. For example, should there be a fourth input feature, Tiny, the teacher could include the advice:

```
IF Tiny THEN
   INFER NotLarge
END
```

Since the term NotLarge is an existing intermediate term, this has the effect of adding a new definition of the term NotLarge. NotLarge would then be defined as the disjunction of the old and new definitions.

The REMEMBER keyword in a conclusion of an IF statement indicates that the agent should retain the value of the condition at the next time step. For example, the teacher could give the advice:

```
IF SpeedLimitSignSays55 THEN
   REMEMBER SpeedLimitIs55
END
```

This advice tells the agent to remember the previous value of the condition SpeedLimitSignSays55 (using a state unit as in FSKBANN) and to call this "memory" SpeedLimitIs55. The teacher could then give advice that checks the condition SpeedLimitIs55.

Note that state values in RATLE are implemented as they are in FSKBANN – the activation of the state unit disappears after one time step. When the teacher wants to indicate that the agent should remember a piece of advice indefinitely, she would have to add a rule that captures this notion, such as:

```
IF SpeedLimitIs55 THEN
   REMEMBER SpeedLimitIs55
END
```

Otherwise, the activation value of SpeedLimitIs55 would disappear after one step. In Section 6.4, I will discuss alternate methods for maintaining the activation of state units.

5.1.2 The REPEAT Statement

```
[ WHEN WhenCondition ]
REPEAT
   RepeatAction
UNTIL UntilCondition
[ THEN ThenAction ]
END
```

A REPEAT statement allows the teacher to specify an action that should be executed over and over. The statement includes a condition that the agent tests to see if it should stop executing the loop. This statement is useful for planning tasks in which the agent must perform some set of repetitive actions in order to achieve a goal.

The inner part of the REPEAT statement works as in Pascal – the action *RepeatAction* is executed while *UntilCondition* is false. The WHEN and THEN parts of the REPEAT statement are optional, though a REPEAT statement without a WHEN part often does not make sense. The WHEN part of a REPEAT statement sets the initial conditions for starting the REPEAT statement; if no WHEN part is included in the REPEAT statement, the agent assumes that it should *always* be starting the REPEAT loop. The optional *ThenAction* is executed by the agent once the loop terminates.

The REPEAT statement can be used as a memory of an intermittent signal, whereas the IF statement can react only to the current input vector. For example, imagine that our game-playing agent has an input feature that indicates whether "Red Light" or "Green Light" is called out – but that this knowledge disappears at the next time step (e.g., the agent hears nothing). In this case, the agent would not know whether it is safe to move, but a REPEAT loop can retain this knowledge by continuing to suggest an action until some condition occurs:

```
WHEN LightIsGreen
REPEAT
   MoveForward
UNTIL LightIsRed
THEN StopMoving
END
```

This statement says that as soon as "Green Light" is called out, the agent should `MoveForward` and continue doing this until "Red Light" is called out, at which point the agent should StopMoving.

5.1.3 The WHILE Statement

```
WHILE WhileCondition DO
   WhileAction
[ THEN ThenAction ]
END
```

A WHILE statement defines a loop similar to the REPEAT statement. The teacher uses a WHILE statement to indicate some action the agent should take whenever the condition holds. As with the REPEAT statement, the WHILE statement is useful when the agent should perform repetitive actions.

A WHILE indicates that the action *WhileAction* should be repeated continually as long as the condition *WhileCondition* is true. The (optional) action *ThenAction* executes when the condition *WhileCondition* becomes false, but only if the loop was executed at least once. An example of a WHILE statement is:

```
WHILE NOT LightIsRed DO
   MoveForward
THEN StopMoving
END
```

The functionality of the WHILE statement is in many ways subsumed by the IF statement. Since the condition of an IF statement is checked at every step by the agent, an IF statement works essentially as a WHILE statement. The main difference between the IF and the WHILE statements is that the WHILE statement has an optional THEN part. I provide the WHILE statement largely because the teacher may find the WHILE more natural for representing a particular piece of advice.

5.2 Conditions

Each of the basic statements makes use of conditions to describe states of the world; these conditions are the triggers for performing actions. Conditions can be made up of three types of grammar pieces: (1) the names associated with input features, intermediate, and remembered terms; (2) logical combinations of sub-conditions; (3) fuzzy terms (i.e., terms that represent either multiple input features or functions based on input features).

5.2.1 Conditions: Terms

A teacher may indicate a condition that is simply the name of a Boolean term. Terms are the names associated with input features or the names of intermediate or remembered terms that the teacher previously defined (such as the predicate `NotLarge` discussed in Section 5.1.1). A term is true (the condition holds) if and only if the condition or input feature associated with that name is true. In Section 5.6, I define legal names for terms.

5.2.2 Conditions: Logical Combinations

"(" *Condition* ")"
NOT *Condition*
Condition1 AND *Condition2*
Condition1 OR *Condition2*

Conditions can be more complex than simple terms. Conditions may be combined by the teacher using the logical operations NOT, AND, and OR. These operations work in the standard way (e.g., NOT *Condition* is true when *Condition* is false, etc.). These operations can also be employed recursively and parentheses can be used to create any logical combination of the current terms and fuzzy conditions.

5.2.3 Conditions: Fuzzy Conditions

Object { IS | ARE } *Descriptor*	[*Type*1]
Quantifier Object { IS | ARE } *Properties*	[*Type*2]

Zadeh defines a *fuzzy set* as "a class of objects with a continuum of grades of membership" (Zadeh, 1965, pp. 338). A *fuzzy membership function* characterizes the fuzzy set by assigning each object a value in the interval [0, 1] that indicates to what extent that object fits into the class (0 being not at all, 1 being a perfect fit). For example, a fuzzy set could be the set of tall trees. A tree five feet in height would probably have a fuzzy membership function value of 0 for this set, while a 60-foot high tree would likely have a membership value of 1. However, a 40-foot high tree might only be considered somewhat tall, so it might have a membership value of 0.7.

The RATLE language has fuzzy conditions, fuzzy terms, and fuzzy functions. A *fuzzy condition* (e.g., `Many Trees ARE Tall`) is a construct that specifies a fuzzy membership function on the input features. *Fuzzy terms* (e.g., `Many` and `Tall`) are the names used in fuzzy conditions that determine which fuzzy membership function to apply to the input features, and a *fuzzy function* (e.g., the initial definition of `Many`) is the fuzzy membership function associated with a fuzzy term.

The basic idea behind the fuzzy conditions of RATLE is to make it easier for the teacher to provide instructions – rather than having to define functions of input features, the teacher can use general terms such as `Big` and `Near`. Fuzzy conditions also serve the purpose of creating Boolean conditions from the non-Boolean input features.

The set of fuzzy terms available to the teacher is defined by the initializer. These terms are generally specific to the task being learned, and I assume that a new set of terms will have to be defined for each environment. However, once defined, the teacher can use these terms over a lengthy teaching period.

RATLE's language contains two types of fuzzy conditions, the first of which refers to input features associated with single objects. For example, a fuzzy condition of the first type might be:

```
Tree1 IS Tall
```

The second type of fuzzy condition is more complex, and refers only to input features associated with objects that have properties (e.g., an input feature that counts trees, but only ones that are 40 to 60 feet in height). An example of the second type of fuzzy condition is:

```
Many Trees ARE Tall
```

Note that IS and ARE are equivalent; the teacher may use the word that makes the most sense.

In the first type of fuzzy condition, *Object* (e.g., Tree1) is a name. The *Descriptor* (e.g., Tall) in the first fuzzy condition is matched to the set of descriptors that have been supplied by the initializer. Additional details on this process are provided below.

In the second type of fuzzy condition, the *Object* (e.g., Trees) must correspond exactly to the name of an input feature or features that have properties. The *Quantifier* (e.g., Many) is a value that indicates some value or range of values that *Object* (e.g., Trees) should have, and *Properties* (e.g., Tall) are used to select the input features to be examined to see if the appropriate quantity exists. *Properties* in the second type of fuzzy condition can be single descriptors, or conjunctions or disjunctions of many descriptors. For example, a sample fuzzy condition of the second type is:

```
Many Trees ARE (Tall AND Branchless)
```

5.3 Actions

A statement generally specifies an action or actions the agent should take given that the conditions of the statement are met. Possibilities for actions include: (1) a single action; (2) a set of possible actions (any of which would be appropriate); (3) a prohibition from taking an action; and (4) a sequence of actions. In this section, I describe the different types of actions the teacher can indicate.

5.3.1 Actions: Single Actions

The simplest action is the name of a single action. Actions are named using the rules for defining input names, discussed in Section 5.6.

5.3.2 Actions: Alternative Actions

"(" *ActionName* { OR *ActionName* }+ ")"

A more complex form of action is an *alternative action*, in which the teacher indicates a set of actions, any of which would be appropriate for the situation. An example of the use of an alternative action is:

```
IF GoalIsNorthAndEast THEN
   ( MoveEast OR MoveNorth )
END
```

No preference is attached to the ordering of the actions in the alternation.

5.3.3 Actions: Action Prohibitions

DO_NOT *ActionName*

The *action prohibition* is not a suggestion that the student take an action in the traditional sense, but rather that under certain conditions a particular action is *not* a good idea. For example, the teacher could indicate:

```
IF LightIsRed THEN
   DO_NOT MoveForward
END
```

This IF statement suggests the agent should not `MoveForward` under the `LightIsRed` condition – advice which would be useful both in a car-driving task and for playing the game *Red Light - Green Light.*

5.3.4 Actions: Multi-Step Plans

MULTIACTION *ActionName+* END

The MULTIACTION is the most complex form of action; it specifies a *plan* – a sequence of actions to perform. The sequence can be of arbitrary length and is made up of an ordered list of action names. A MULTIACTION looks like this:

```
WHEN GoalIsNorthAndEast
REPEAT
   MULTIACTION
      MoveEast
      MoveNorth
   END
UNTIL NOT GoalIsNorthAndEast
END
```

This statement says that when the condition `GoalIsNorthAndEast` is true, the agent should execute the plan of first moving east, then at the next step moving north, as long as the condition `GoalIsNorthAndEast` is true.

Note that in a looping construct such as a REPEAT statement, the condition of the loop is checked only at the *end* of the sequence of actions – the condition is not checked at each step of the sequence (e.g., after executing the action `MoveEast` in the above MULTIACTION, the agent does not check to see whether `GoalIsNorthAndEast` is still true before executing `MoveNorth`).

5.4 The RATLE Preprocessor

In order to make articulating certain sets of related statements easier, I included a preprocessor to the RATLE advice language. A preprocessor statement takes one of the following forms:

```
FOREACH Variable IN "{" Value [ "," Value ]* "}" statement+ ENDFOREACH
FOREACH "(" Variable1 [ "," Variable2 ]* ")" IN
   "{" "(" Value1 [ "," Value2 ]* ")" [ "," "(" Value2 [ "," Value2 ]* ")" ]* "}"
   statement+ ENDFOREACH
```

The teacher uses a preprocessor statement to define a set of statements that are similar. A preprocessor statement indicates the name of one or more variables and the string(s) with which those variables are to be replaced in order to produce the statements.

For example, imagine that the agent is able to move in four directions (`East`, `North`, `West`, and `South`). The teacher may want to give advice in the following form:

```
IF GoalIsNorth THEN
   MoveNorth
END
```

But the teacher may want to indicate that this advice applies to any of the four directions. The teacher can use the preprocessor to specify these four rules at once:

```
FOREACH dir IN { East, North, West, South }
   IF GoalIs$(dir) THEN
      Move$(dir)
   END
ENDFOREACH
```

The form `$(dir)` in the statements within the FOREACH indicates the strings to be replaced with the list of values (e.g., `East, North, West, South`) to form the actual statements. Hence, the above FOREACH would produce four statements.

The teacher uses the second type of preprocessor command to assign a group of values to a set of variables *en masse*. For example, a FOREACH could take the following form:

```
FOREACH (ahead, back) IN
  { (East,West), (West,East), (North,South), (South,North) }
  IF EnemyIs$(ahead) THEN
    Move$(back)
  END
ENDFOREACH
```

This statement defines four rules, where each rule checks whether there is an `Enemy` in one direction, and if so, suggests the agent move in the opposite direction. The teacher may also nest preprocessor statements.

Note that following preprocessing, the resulting rules, which may share similar structures, are decoupled. Thus, each statement produced by the preprocessor is treated as a separate statement. (See Sun (1992) and Shastri (1988) for work on variable binding in neural networks that do not learn.) A method such as soft-weight sharing (Nowlan & Hinton, 1992), would allow the agent to maintain connections between similar pieces of advice, but requiring the advice to do so would detract from the agent's ability to refine each piece of advice individually.

5.5 Limitations of and Extensions to the RATLE Advice Language

My intent in developing RATLE was to provide a language that allows a teacher to express a wide range of advice in a form that could be easily transferred into an RL agent. The statements I implemented were chosen in part because of my experiences acting as the teacher for the testbeds I present in Chapter 7. But I also tried to focus on programming statements that seem to exist in one form or another in most programming languages. In this section I will discuss some features that are missing from the RATLE language and ways in which those features might be included in future versions of the language.

5.5.1 Limitation One: Embedding Statements within Statements

RATLE statements cannot be embedded within other statements. Such a capability would be very useful for implementing multi-step plans. Currently, when the teacher indicates a multi-step plan, she may specify only a condition that holds true at the start of the plan. It would be useful if the teacher could add conditions that should still hold during the execution of the plan. For example, imagine we have a robot agent attempting to pick up blocks in an environment where multiple robots are trying to perform this task. A useful piece of advice might be:

```
IF NextToBlock1 AND OnGroundBlock1 THEN
    MULTIACTION
        ExtendRobotArm
        GraspObject
        RetractRobotArm
    END
END
```

Should another robot pick up the block while the first robot is extending its arm, the first robot would waste time trying to grasp a block that is no longer on the ground.

If RATLE allowed embedded statements, the teacher could address this problem with an IF statement within the MULTIACTION of the outer IF statement:

```
IF NextToBlock1 AND OnGroundBlock1 THEN
    MULTIACTION
        ExtendRobotArm
        IF OnGroundBlock1 THEN
            MULTIACTION
                GraspObject
                RetractRobotArm
            END
        END
    END
END
```

This type of embedded statement would be straightforward to implement, since the condition that occurs before the second step of the inner multi-step plan could be added to the unit that checks if the second step of the plan should be taken (see Section 6.3.4). Allowing loops within loop statements would require more care, since the state units defining the loop have to be connected together appropriately, but would still be straightforward to implement.

A related approach would be to give the teacher a language keyword (e.g., ALWAYS) that she could use to indicate that a condition must hold before each step of a multi-step plan. The teacher then could give the following advice in the above situation:

```
IF ALWAYS (NextToBlock1 AND OnGroundBlock1) THEN
  MULTIACTION
    ExtendRobotArm
    GraspObject
    RetractRobotArm
  END
END
```

The ALWAYS would state that the condition (NextToBlock1 AND OnGroundBlock1) should be checked before the agent attempts each action in the plan.

5.5.2 Limitation Two: Defining Procedures

Simple procedures are another helpful feature that could be added to the advice language. The teacher could give a name to a set of actions that she could then use in multiple other pieces of advice. For example, the teacher could define a procedure PickupObject as follows:

```
PROCEDURE PickupObject
  MULTIACTION
    ExtendRobotArm
    GraspObject
    RetractRobotArm
  END
END
```

Implementing procedures would be fairly straightforward, since RATLE could simply parse the body of the procedure and substitute the body every time the procedure name is used by the teacher.

5.5.3 Limitation Three: Using Complex Functions

Currently, the teacher may specify only logical combinations of Boolean functions in conditions. RATLE relies on the initializer to create fuzzy conditions that turn the non-Boolean input features into Boolean conditions. Relaxing this limitation and letting the teacher construct conditions that include arithmetic functions would be helpful, especially in domains in which many of the features are not Boolean. The main difficulty with allowing other functions is that RATLE currently uses sigmoidal activation functions for all of its neural-network units. In order to implement other types of functions, RATLE would have to allow other types of units (e.g., units that can perform multiplication, division, square roots, etc.), and different neural-network learning rules would be needed for these units.

5.5.4 Limitation Four: Providing Agent Goals

Goals are a useful form of advice a teacher can give to an RL agent (Gordon & Subramanian, 1994; Mataric, 1994); they represent conditions of the environment that the agent should try to achieve. The teacher can use goals to divide a task into sub-tasks that the agent may find easier to learn.

For example, imagine an agent trying to transport a box from a warehouse to a factory. A good sub-goal of this task might be for the agent to be holding a box (e.g., GOAL `HoldingBox`). In RATLE, I could implement goals as teacher-defined reinforcement signals – when the environment matches the condition of the goal the agent would receive a reinforcement (e.g., the agent would get a positive reward for holding a box). This implementation might be problematic, however, since we want the agent to eventually ignore the teacher-defined reinforcements (since they are not "real," in some sense) and concentrate only on the actual reinforcements. One way to solve this problem would be to have the reinforcement decay a bit every time the agent receives the reinforcement, so that over time the teacher-defined reinforcements would disappear.

More complex forms of goals might use fuzzy terms to allow the teacher to indicate the strength of a reinforcement the agent should receive (e.g., the teacher could attach names like `strong`, `medium`, and `weak` to goals). We could also imagine advice about conditions to avoid in the environment, which could be implemented as negative reinforcements.

5.6 The Input Language

Recall that before the teacher can provide advice to the agent, the initializer must attach names to the each of the input features. The teacher uses these names to reference input features that are combined to form conditions. Currently, RATLE understands two types of input features: BOOLEAN and REAL values. The initializer gives a name to each feature of the feature vector using one of the input language structures explained below. The question of who exactly defines the set of features used in the feature vector is left open. The features of the feature vector could be defined by the initializer, but they could also be defined by yet another person who defines the task being addressed. The job of the initializer is to name the input features, no matter who defines them.

Although the specifics of defining inputs and fuzzy terms are somewhat task-dependent, it is important to know how input features are described in order to understand how I implement fuzzy conditions in RATLE. Readers not interested in these details may skip this section and Section 5.7.

5.6.1 Inputs: Name Strings

The teacher uses names to refer to input features, intermediate terms, and actions. RATLE uses *name strings* that are similar in definition to Pascal name strings: a name string begins with a lower or upper case character of the alphabet (i.e., from 'a' to 'z' or 'A' to 'Z'); followed by an arbitrary number of alphabet, numeric (i.e., '0' to '9'), or underscore (_) characters. Exceptions are the keywords of the RATLE language (e.g., WHILE, REPEAT, END, MULTIACTION, etc.) which cannot be used as names by the teacher.

5.6.2 Inputs: BOOLEAN Features

BOOLEAN *Name*

The simplest type of input is a BOOLEAN feature. The name in a BOOLEAN feature is simply a name string. This type of input is assumed to have a value of 0 or 1 (i.e., false or true), and can be used in a condition by simply including the name.

The initializer can use BOOLEAN features to implement Nominal features as well. For example, if the color of a `Block` can be `Red`, `Green`, or `Blue`, the initializer would create three

BOOLEAN input features: `BlockIsRed`, `BlockIsGreen`, and `BlockIsRed`. The initializer could also represent the color of the block using a single REAL feature, but then he or she would have to assign arithmetic values to each of the possible colors, `Red`, `Green`, and `Blue`.

5.6.3 Inputs: REAL Features

REAL *FeatureName Property∗*
  "[" *LowValue* ".." *HighValue* ["," *NormLowValue* ".." *NormHighValue*] "]"

The initializer represents a feature that has an arithmetic value (e.g., the height of `Tree1` in feet) using a REAL construct. The string *FeatureName* in a REAL feature is the name of the feature. The optional *Properties* are characteristics that each of the object(s) being measured share – the definition of properties appears below. The name of a REAL feature that has no properties must have a unique string for *FeatureName* – a string not used for any other input, action, or fuzzy term. The *LowValue* and *HighValue* part of a REAL definition indicates the minimum and maximum value that the REAL will take on, and the *NormLowValue* and *NormHighValue* are included if the input value is normalized in the feature vector. These values must be numbers. For example:

REAL `Tree1Height` [0 .. 20 , 0 .. 1]

is a REAL value indicating the height of *Tree1* is a value between 0 and 20, but that the value is normalized to be between 0 and 1 (e.g., if *Tree1* was height 16, the value of this input would be 0.8).

A REAL input feature that has properties does not need a unique name string; in fact, it is assumed that generally several such variables share name strings. REAL features may have properties to indicate that multiple input features measure the same aspect for different groups of objects. For example, the set of input features could include three "tree-height" features, one that counts the number trees of height 0 to 20, another that counts the number of trees of height 20 to 40, and a third that counts trees of height 40 to 60. In this case, the initializer can give each of the input features the same base name (e.g. `NumberOfTrees` or `Trees`) and then indicate the differences between the three input features using properties. The advantage of this approach is that the teacher can use fuzzy conditions that perform operations that look at all of the inputs (e.g., an operator that counts the number of trees disregarding how tall they are). More details on how such fuzzy terms are defined are given in Section 5.7.

REAL feature properties have the following possible forms:

PropertyName IN "[" *LowPropValue* ".." *HighPropValue* "]"
PropertyName "=" *PropertyValue*

A property name is a name string, and property values are numbers. The first type of property indicates that the object(s) being measured have values of *PropertyName* that are between *LowPropValue* and *HighPropValue*. The second type of property indicates that each object being measured has the value *PropertyValue* for *PropertyName*. An example of a REAL feature with properties is:

```
REAL Trees  Height IN [ 40 .. 60 ]  Branches = 0  [ 0 .. 20 , 0 .. 1 ]
```

This is an input feature that represents how many trees are of Height 40 to 60 with zero Branches. Many input features can have the *FeatureName* `Trees`, each defined by a different set of properties.

In RATLE, conditions are made up of logical combinations of Boolean arguments. REAL features, since they are not Boolean in nature, can only be referenced using the fuzzy terms described below. These fuzzy terms allow the teacher to create BOOLEAN features based on the values of REAL input features.

5.7 Fuzzy Language Terms

Besides naming the input features and actions, the initializer may also provide a set of fuzzy terms. These terms allow the teacher to make reference to input features using general terms like `Big` and `Near`. These terms also convert the REAL features of the input language into Boolean terms that the teacher may combine into conditions.

Recall that fuzzy conditions come in two forms:

Object { IS | ARE } *Descriptor*	[*Type*1]
Quantifier Object { IS | ARE } *Properties*	[*Type*2]

In the first type of fuzzy condition, the term *Descriptor* refers to a fuzzy term of type DESCRIPTOR. In the second type of fuzzy condition, the terms *Quantifier* and *Properties* refer to fuzzy terms of type QUANTIFIER and PROPERTY. The DESCRIPTOR, QUANTIFIER, and PROPERTY terms are defined by the initializer.

5.7.1 Fuzzy Terms: Descriptors

DESCRIPTOR *PartialName* "::=" *PartialName Operator Value*
DESCRIPTOR *PartialName* "::="
  "(" ["–"] *PartialName* { {"+"|"–"} *PartialName* }* ")" *Operator Value*

The initializer uses a DESCRIPTOR to describe a simple fuzzy function of the input features. A fuzzy condition of the first type contains the name of an object and the name of a function (*Descriptor*) to calculate with respect to that object. RATLE determines the appropriate fuzzy function to calculate by matching the name *Descriptor* from the fuzzy condition to the names of the DESCRIPTORs. These DESCRIPTORs may contain variables (as described below); thus RATLE applies a string-matching process, instantiating the variables in the names to find the appropriate function(s). These variables allow the initializer to define fuzzy terms that may apply to a number of different objects (e.g., one term for "big" that applies to the input features `SizeOfObject1`, `SizeofObject2`, etc.). When more than one descriptor matches, RATLE creates multiple network units and then creates a disjunction of those units. After the variables in the DESCRIPTOR names are instantiated by RATLE, each name in the resulting function has to correspond to an existing input feature of type REAL (that has no properties). RATLE discards any match that results in names that are not defined.

Following the string-matching process, the DESCRIPTOR defines a simple sum of the input features (the portion of a DESCRIPTOR to the right of the "::=" before the *Operator*). The *Operator* in the DESCRIPTOR must be one of the following: ">"; "<"; "<="; ">="; "="; and "! =" (not equals). *Value* is a number that is used in the comparison to the sum of input features. The resulting function is fuzzy because it is implemented using sigmoidal neural-network units. Therefore, these functions cannot have any discontinuities.

The key to a DESCRIPTOR is the *PartialName* construct. A *PartialName* has the form:

{ *NameString* | "?" "(" *NameString* ")" }+

That is, a *PartialName* consists of a series of name strings and variable strings of the form ?(*Name*). Each DESCRIPTOR can have any number of these variable strings. One predefined variable string for each DESCRIPTOR is ?(Object), which is always bound to the value of *Object* from the fuzzy condition. For example, ?(Object) would be bound to Tree1 in this condition:

```
Tree1 IS Tall
```

To create a fuzzy unit for this fuzzy condition, RATLE matches the DESCRIPTOR in the fuzzy condition to all of the DESCRIPTORs defined previously and then instantiates each of the DESCRIPTORs that match. For most fuzzy descriptors, this is simple; for example, the DESCRIPTOR Tall might match only a single DESCRIPTOR:

```
DESCRIPTOR Tall ::= ?(Object)Height > 40
```

But this process can be more complex when a fuzzy term contains variables other than the object variable. For example, a fuzzy condition could be:

```
Tree1 IS CloserThanTree2
```

where CloserThanTree2 could be defined by the following DESCRIPTOR:

```
DESCRIPTOR CloserThan?(Other) ::=
    (DistTo?(Object) - DistTo?(Other)) < 0
```

This DESCRIPTOR also contains the variable ?(Other), which RATLE must instantiate in order to match this DESCRIPTOR.

5.7.2 Fuzzy Terms: Properties

PROPERTY *Name* "::=" *Property*

In the second type of fuzzy condition, the teacher uses a set of *Properties* to select the input features that the fuzzy function will examine. The second type of fuzzy condition can be applied only to input features that have properties.

PROPERTY descriptors are simple names associated with properties; they are defined in the same manner as properties for input features (see Section 5.6.3). An example of a PROPERTY is:

```
PROPERTY Tall ::= Height IN [ 50 .. 70 ]
```

This says that an input feature is considered to match the PROPERTY `Tall` if it represents items of height 50 to 70. PROPERTYs are used to select input features whose properties match the properties indicated in a fuzzy condition (e.g., input features that represent objects that are of height 50 to 70 would match the PROPERTY `Tall`). Properties are not fuzzy in the standard sense, but they can result in partial matches, because a PROPERTY range of values does not have to exactly match a range associated with an input feature. For example, if an input feature counted trees with heights ranging from 40 to 60, RATLE would assume that only half of the trees match the PROPERTY `Tall` defined above. In general, RATLE assumes that the uniform distribution applies to ranges.

5.7.3 Fuzzy Terms: Quantifiers

QUANTIFIER *Name* "::=" [SUM] *Operator Value*
QUANTIFIER *Name* "::=" [SUM] IN "[" *LowValue* ".." *HighValue* "]"

The second type of fuzzy condition *Type*2 specifies a type of object being examined (e.g., `Trees`), a property or properties that those objects must have (e.g., `Tall`), and a QUANTIFIER (e.g., `Many`) which indicates the type of fuzzy function that should be applied to all of the matching objects. For example:

`Many Trees` ARE `Tall`

This condition holds if `Many Trees` satisfy the property `Tall`.

QUANTIFIERs specify an operator and a threshold that the input features must meet. The operators available are the same as for descriptors (e.g., ">"; "<"; "<="; ">="; "="; and "! ="). *Number* in a QUANTIFIER is the number against which the function is measured. The form of a QUANTIFIER using an IN is a shorthand for specifying that the desired value is *between* the two values *LowValue* and *HighValue.*

An example of a QUANTIFIER is:

QUANTIFIER `Many` ::= > 3

Since the keyword SUM is omitted, RATLE finds the input features that match the property `Tall`, and then determines if any of these inputs meets the fuzzy criterion `Many`. For example, if three input features all count the number of `Trees` that are `Tall`, RATLE would create fuzzy functions that determine whether there are `Many Trees` at *each* of these input features individually. Then RATLE creates a disjunction of these fuzzy functions that determines if *any* of the input features meet the condition.

When the QUANTIFIER includes the keyword SUM, the above process is differently. For example, the above QUANTIFIER could be rewritten as:

QUANTIFIER Many ::= SUM > 3

In this case, RATLE determines whether the total across *all* of the inputs matching the property *Tall* meet the test. If three inputs count the number of `Trees` and match the property *Tall*, RATLE creates a fuzzy function that determines if the sum of `Trees` across these three input units meets the test associated with `Many`.

To illustrate the difference between using a QUANTIFIER with the keyword SUM and one without, consider the following example. Assume that the input vector includes three features that count the number of `Tall Trees`: one feature counts `Tall Trees` with zero to four branches, one counts `Tall Trees` with five to eight branches, and one counts `Tall Trees` with more than eight branches. Further assume that in the current input vector, there is one `Tall Tree` with zero branches, two `Tall Trees` with five branches, and one `Tall Tree` with nine branches. Assuming the QUANTIFIER `Many` is defined as above without the keyword SUM, the fuzzy condition `Many Trees` ARE `Tall` would be false, since none of the input features meets the condition `Many` individually (i.e., none of these inputs indicates more than three trees). However, if the keyword SUM is specified by the initializer, the condition would be true, since there are more than three `Trees` that are `Tall` across the set of matching input features.

5.8 Limitations of the Input and Fuzzy Term Languages

There are a number of types of input features and fuzzy terms that I have not implemented. As I noted above, my choices for allowable input features and fuzzy terms were driven largely by the environments I explored in my tests. An array exemplifies an input feature construct that would be useful. This could be helpful when the input vector includes a 2D array of pixel variables from a picture, for example, so that the teacher could make reference to a pixel and its surrounding pixels. For fuzzy terms, a means to attach initial fuzzy values to nominally-valued features would be helpful. For example, the teacher may want to refer to "Dark" and "Light" objects. The initializer could make this possible by assigning initial fuzzy values to different colors. `Black` would closely match the term "Dark" and the term

"Light" not at all, while `Purple` would match "Dark" somewhat less and "Light" somewhat more and `Yellow` would match "Dark" very little and "Light" quite a bit. As with input constructs, implementing new types of fuzzy terms will likely be driven by the needs of a task, though the current constructs should already cover a large percentage of cases.

5.9 Summary

The RATLE advice language allows a teacher to give an RL agent advice about actions to take, given conditions of the world. Table 20 presents a complete grammar for RATLE's advice language. Note that RATLE typechecks the instructions after parsing them, to determine

Table 20: The grammar used for parsing RATLE's advice language; ε is the empty string.

$$
\begin{array}{lll}
\textit{stmts} & \leftarrow & \textit{stmt} \mid \textit{stmts} \text{ ";" } \textit{stmt} \\
\\
\textit{stmt} & \leftarrow & \text{IF } \textit{ante} \text{ THEN } \textit{conc else} \text{ END} \\
 & & \mid \text{WHILE } \textit{ante} \text{ DO } \textit{act postact} \text{ END} \\
 & & \mid \textit{pre} \text{ REPEAT } \textit{act} \text{ UNTIL } \textit{ante postact} \text{ END} \\
\\
\textit{else} & \leftarrow & \varepsilon \mid \text{ELSE } \textit{conc} \\
\textit{postact} & \leftarrow & \varepsilon \mid \text{THEN } \textit{act} \\
\textit{pre} & \leftarrow & \varepsilon \mid \text{WHEN } \textit{ante} \\
\\
\textit{conc} & \leftarrow & \textit{act} \mid \text{INFER } \textit{name} \mid \textit{remember name} \\
\textit{act} & \leftarrow & \textit{cons} \mid \text{MULTIACTION } \textit{clist} \text{ END} \\
\textit{clist} & \leftarrow & \textit{cons} \mid \textit{cons clist} \\
\textit{cons} & \leftarrow & \text{Name} \mid \text{DO\_NOT Name} \mid \text{"("} \textit{ corlst } \text{")"} \\
\textit{corlst} & \leftarrow & \text{Name} \mid \text{Name OR } \textit{corlst} \\
\\
\textit{ante} & \leftarrow & \text{Name} \mid \text{"("} \textit{ ante } \text{")"} \mid \text{NOT } \textit{ante} \\
 & & \mid \textit{ante} \text{ AND } \textit{ante} \mid \textit{ante} \text{ OR } \textit{ante} \\
 & & \mid \textit{quant} \text{ Name } \textit{isare desc} \\
\\
\textit{quant} & \leftarrow & \varepsilon \mid \text{Name} \\
\textit{isare} & \leftarrow & \text{IS} \mid \text{ARE} \\
\\
\textit{desc} & \leftarrow & \text{Name} \mid \text{NOT } \textit{desc} \mid (\ \textit{dexpr}\ ) \\
\textit{dexpr} & \leftarrow & \textit{desc} \mid \textit{dexpr} \text{ AND } \textit{dexpr} \mid \textit{dexpr} \text{ OR } \textit{dexpr}
\end{array}
$$

if the *Name* constructs are appropriate for their location in the grammar. Examples of statements provided to RATLE are shown in Table 19 and in Chapter 7.

The RATLE language is designed around three programming language constructs similar to the Pascal IF, REPEAT, and WHILE statements. An IF statement allows the teacher to specify a condition and a corresponding action, as well as an action to take should the condition be false. The teacher can also use an IF statement to build new intermediate terms based on existing terms. A REPEAT statement specifies a loop that is terminated under a particular condition, and a WHILE statement is a loop with an entry condition. In each loop body, the teacher specifies an action to be taken during the loop.

RATLE provides a complex language for specifying conditions. Conditions can range from single Boolean input features to complex logical combinations of inputs. Conditions may also be fuzzy conditions, which are based on input features that are not Boolean-valued. Fuzzy conditions are based on fuzzy terms defined prior to the teacher's instruction by the initializer; these terms will generally be specific to a particular environment. Grammars for the language used by the initializer to define the input features and the fuzzy terms are shown in Tables 21 and 22, respectively.

Each type of statement in RATLE's language indicates an action or actions that the teacher suggests the agent should perform (possibly in loops) given certain conditions of the world. Actions can be single actions, action alternatives (the teacher may indicate a list of actions that are reasonable), and action prohibitions (suggestions to *not* take an action). Actions can also be plans (sequences of actions). In the following chapter, I provide further details

Table 21: The grammar used for parsing RATLE's input language.

```
inputs  ← input | inputs ";" input

input   ← BOOLEAN Name
          | REAL proplst "[" Number ".." Number normvs "]"

proplst ← prop | proplst prop
prop    ← Name IN "[" Number ".." Number "]"
          | Name "=" Number

normvs ← ε | "," Number .. Number
```

Table 22: The grammar used for parsing RATLE's fuzzy terms.

terms	←	*term* | *terms* ";" *term*
term	←	DESCRIPTOR *partial* "::=" *psum op Number*
		| PROPERTY *Name* "::=" *prop*
		| QUANTIFIER *Name* "::=" *sum qrest*
qrest	←	*op Number*
		| IN "[" *Number* ".." *Number* "]"
partial	←	*namev* | *partial namev*
namev	←	*Name* | "?(" *Name* ")"
psum	←	*partial* | "(" *osign partial plist* ")"
plist	←	ε | *sign partial plist*
osign	←	ε | "−"
sign	←	"+" | "−"
op	←	"<" | ">" | "<=" | ">=" | "=" | "! ="
prop	←	*Name* IN "[" *Number* ".." *Number* "]"
		| *Name* "=" *Number*
sum	←	ε | SUM

about the implementation of the features of the RATLE advice language.

Chapter 6

Transferring Advice into a Connectionist Reinforcement-Learning Agent

Once the teacher specifies advice for the reinforcement-learning agent, RATLE transfers the advice into the agent. To do this, RATLE must translate the teacher's instructions into a form that the agent can use. In standard reinforcement learning (RL), the agent senses the current world state, chooses an action to execute, and occasionally receives rewards and punishments. Based on these reinforcements from the environment, the agent's task is to improve its action-choosing module so that the total amount of reinforcement it receives is increased. Since my RL agent uses connectionist Q-learning (Lin, 1992; Sutton, 1988; Watkins, 1989), the agent's action-choosing module is a neural network. Therefore, to translate the advice of the teacher, RATLE must make changes to the agent's neural network that capture the meaning of the advice. This is done by translating statements in the RATLE language into new network units and links that correspond to the concepts represented by the statements.

Table 23 shows the main loop of an agent employing connectionist Q-learning, augmented (in italics) by my process for incorporating advice. The main difference between my approach and standard connectionist Q-learning is that the agent continually checks if the teacher has any recommendations to offer, and if so, incorporates those recommendations into its utility function. In order to understand how statements from the RATLE language are transformed, this chapter defines the steps of the subroutine *IncorporateAdvice* from Table 23. But first, I will discuss the steps that must be taken before the agent can begin the process shown in Table 23.

Recall that a connectionist Q-learner uses a neural network, where the input to the network is a vector of features describing the current state, and the outputs of the network are the predicted utilities for each of the agent's possible actions. In RATLE, I assume that the initial neural network is set up by a user (who need not be the teacher), referred to as

Table 23: The cycle of a RATLE agent. My additions to the standard connectionist Q-learning loop are Step 6 and the subroutine *IncorporateAdvice* (all shown in italics).

Agent's Main Loop	*IncorporateAdvice*
1. Read sensors. 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. 3. Perform selected action. 4. Measure reinforcement, if any. 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. 6. *Advice pending? If so, call IncorporateAdvice.* 7. Go to 1.	6a. *Parse advice.* 6b. *Operationalize any fuzzy conditions.* 6c. *Translate advice into neural-network components.* 6d. *Insert translated advice directly into RL agent's neural-network based utility function.* 6e. *Return.*

the *initializer*. As part of defining the agent's inputs and actions, the initializer must also provide names for each input feature using the language described in the previous chapter. The initializer also defines the set of fuzzy terms the teacher may employ in giving advice about the task the agent will be addressing. After the initializer finishes, he or she gives the teacher a description of the input features, actions, and fuzzy terms. The agent then starts the process shown in Table 23.

The routine *IncorporateAdvice* from Table 23 follows Hayes-Roth, Klahr, and Mostow's framework for advice-taking outlined in Chapter 4. Once the teacher gives a set of instructions, RATLE parses them using the standard Unix tools *lex* and *yacc* and the grammars shown in Chapter 5. RATLE then operationalizes any fuzzy conditions in the advice. Fuzzy conditions map to new units in the agent's neural network that capture the appropriate fuzzy membership function. After mapping fuzzy conditions, RATLE translates all remaining parts of the teacher's advice into new units and links. The resulting new units and links are added to the agent's neural network by RATLE and the agent then returns to connectionist Q-learning. The process of adding units and links to a neural network is accomplished by simple extensions to a neural-network simulator. Thus, the key parts of *IncorporateAdvice* are: (6b) how fuzzy conditions are mapped to network units, and (6c) how RATLE translates the other constructs of the RATLE language into neural-network components.

In the following sections, I define how each of the constructs in the RATLE language is translated into corresponding neural-network components. My methods are based on the KBANN algorithm (Towell et al., 1990) which is extended in the following ways: (1) advice can contain multi-step plans, (2) it can contain loops, (3) it can suggest actions to avoid, (4) it can contain fuzzy conditions, and (5) it can be given more than once. Details of these extensions are shown as I discuss how RATLE translates each construct.

The process of translating language constructs into neural-network parts can be divided into separate processes for translating the three basic language components of RATLE: statements, conditions, and actions. A condition represents the teacher's description of some state of the world that must be met in order for an action or actions to be taken. To translate a condition, RATLE creates a hidden unit that is highly active when the condition is true and inactive when the condition is false. RATLE may also create new hidden units combining conditions in order to translate certain types of statements. RATLE connects the resulting units to the action or actions specified by the teacher. In the following sections I show how RATLE combines conditions and actions to map statements, how RATLE turns conditions into corresponding units, and how the units from statements are connected to actions.

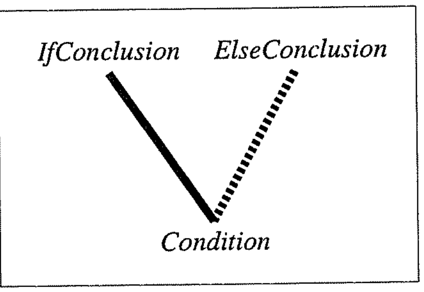
6.1 Translating Statements

RATLE maps statements into neural-network components[1] by connecting the conditions and actions of the statements in appropriate ways. Remember that RATLE maps a condition to a unit that is highly active when the condition is true. To map statements, RATLE adds links from the units representing conditions to the units representing actions as indicated in the statements. As part of representing more complex statements, RATLE may introduce new units that combine conditions.

[1]In demonstrating how RATLE represents the teacher's instructions I often show diagrams containing the construct being translated (on the left) and a corresponding neural-network diagram (on the right). In the neural network part of each of these diagrams, all of the links and units shown are new links and units introduced as part of mapping the construct, unless I specifically state otherwise. Links shown as solid lines represent links with large, positive weights (i.e., as in KBANN's highly weighted links). Straight, dashed lines represent links with large, negative weights, and curved, dashed lines represent recurrent links RATLE uses to connect state units. Arcs connecting links in the diagram indicate units that represent conjuncts, units without arcs show disjuncts. Recall also that the flow of activation in my neural-network diagrams is upwards, from the units at the bottom of the diagram to the units at the top.

6.1.1 Translating the IF Statement

```
IF Condition THEN
  [ INFER | REMEMBER ] IfConclusion
[ ELSE
  [ INFER | REMEMBER ] ElseConclusion ]
END
```



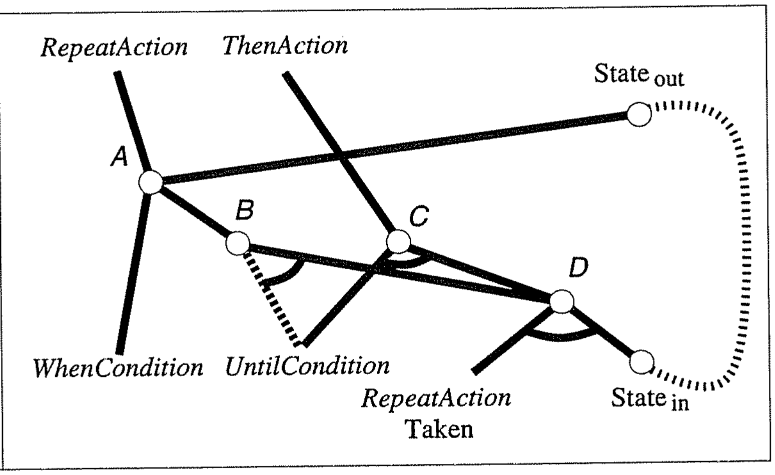
The teacher uses an IF statement to indicate some conclusion that the agent should reach when the unit for *Condition* is true. An ELSE condition can be used by the teacher to recommend a conclusion the agent should reach when *Condition* is false. Recall the three types of conclusions the teacher can suggest: an action the agent should take under that condition, an intermediate term that is true under that condition, and a term to remember. I will discuss the specifics of how these three conclusions are implemented in Section 6.3.1.

To map an IF statement, RATLE first creates a unit representing the condition of the IF (as described in Section 6.2), and then connects this condition to *IfConclusion*. When there is an ELSE part of the statement, RATLE creates a link from the unit representing *Condition* with a negative weight and connects the link to *ElseConclusion*.

Recall that conditions in RATLE can be negated. When this is the case, RATLE negates the weights of the links it constructs (i.e., the link to *IfAction* would have a negative weight and the link to *ElseAction* would have a positive weight). This process is essentially the mapping algorithm of KBANN, which was explicitly designed for IF-THEN rules.

6.1.2 Translating the REPEAT Statement

```
[ WHEN WhenCondition ]
REPEAT
  RepeatAction
UNTIL UntilCondition
[ THEN ThenAction ]
END
```



The teacher uses a REPEAT statement to indicate that the agent should execute *RepeatAction* as long as *UntilCondition* is false. The statement may have an optional *WhenCondition* that must be true for the agent to start the loop (i.e., to start executing *RepeatAction*). The teacher may also include a *ThenAction* to be taken after the agent finishes the loop, should *UntilCondition* be true.

In order to show how to map a REPEAT statement, I will assume that both a WHEN and THEN structure are used in the REPEAT (the diagram shown above is for this case). RATLE starts by constructing a unit (A) that is true if the loop should execute. It connects this unit to *RepeatAction*.

Unit A is a disjunction of the two possible ways the loop may start: (1) if *WhenCondition* is true, or (2) if the agent has just finished executing the loop and *UntilCondition* is false. RATLE determines that condition (1) holds by constructing a unit for *WhenCondition* and then creating a link from this condition to unit A. The second case requires a conjunction (unit B). Unit B is active when the agent has just finished executing the loop (unit D) and the *UntilCondition* is false.

In order to determine that the agent has just finished executing the loop (unit D), RATLE must check: (1) that the REPEAT statement indicated the loop should have started in the last step, and (2) that the agent actually took *RepeatAction* in the last step. RATLE knows the statement indicated the loop should start in the last step if unit A was true in the last step. Therefore, to know that the loop should have started in the last step, RATLE creates a state unit that remembers the value of unit A from the previous step. To know that the agent actually took *RepeatAction* in the last step, RATLE looks at the input features indicating the agent's last action. Unit D is the conjunction of the input feature indicating that action *RepeatAction* was taken in the previous step and the state unit that remembers the previous value of unit A.

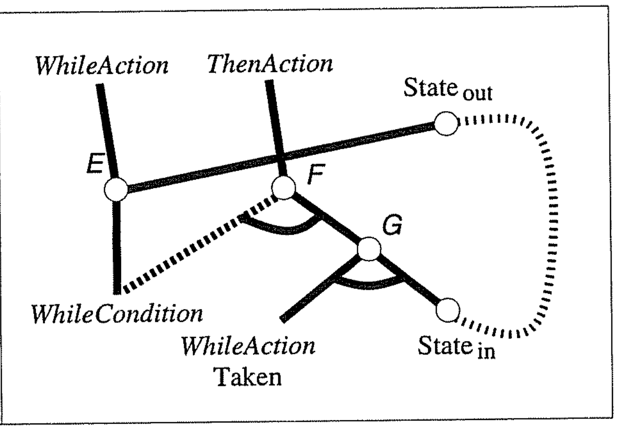
Unit C is constructed for the THEN part of the REPEAT. Unit C is a conjunction that checks that the loop has just finished executing (by making a link to unit D) and that *UntilCondition* is true. RATLE connects unit C to *ThenAction*. This unit is omitted when there is no THEN action.

If there is no WHEN part of the REPEAT statement, RATLE sets up unit A so that it is always true. This is done by setting the bias of the unit to a large positive number. The rest of the REPEAT statement is mapped as described above.

The action *RepeatAction* may be a multi-step plan in a REPEAT statement. If so, RATLE determines that the agent has just finished executing the loop (unit D) by remembering the value of the unit that predicts the *last* step of the multi-step plan, rather than remembering the value of unit A, which would predict the *first* step of the multi-step plan. For more details and an example of how this works, see Section 6.3.4.

6.1.3 Translating the WHILE Statement

```
WHILE WhileCondition DO
  WhileAction
[ THEN ThenAction ]
END
```



The teacher uses a WHILE statement to recommend that the agent take *WhileAction* as long as *WhileCondition* holds. A WHILE statement may have an optional THEN part, which specifies an action to take when the loop has been executing and *WhileCondition* is false.

A WHILE loop starts for two reasons: (1) *WhileCondition* is true, or (2) the agent has just finished executing the action of the loop and *WhileCondition* is still true. This second case is redundant, since the agent must start the loop when *WhileCondition* is true, regardless of whether the agent has just finished executing the loop. Therefore, RATLE does not check the second case. So, RATLE starts by creating a unit for the *WhileCondition.* A link is then made by RATLE to unit E, which is true when the loop should start. RATLE connects unit E to *WhileAction.*

If there is a THEN part of the WHILE loop (the diagram shows this case), RATLE creates unit F, which recommends *ThenAction.* Unit F is a conjunction that checks that *WhileCondition* is false, and the agent just finished executing the loop (unit G). Unit G is similar to unit D for the REPEAT statement. It is a conjunction that determines that the *WhileAction* was the agent's last action and whether the loop should have started at the last step. The agent determines that the loop should have started in the last step by remembering the value of unit E with a state unit.

6.2 Translating Conditions

RATLE represents a condition in a statement with a unit that is true (i.e., highly active) when the condition is true, and false (i.e., inactive) when the condition is false. RATLE's process is the same as KBANN's for mapping conditions, except that RATLE also supports fuzzy conditions. Since fuzzy conditions involve fuzzy membership functions, RATLE creates a unit that maps each fuzzy function – the more active the unit the better the fuzzy membership criterion is met.

6.2.1 Translating Conditions: Term Names

When the condition of a statement in RATLE is the name of a Boolean input unit or an intermediate term, RATLE represents this condition by simply using the unit that corresponds to the term.

6.2.2 Translating Conditions: Logical Combinations

"(" *Condition* ")"
NOT *Condition*
Condition1 AND *Condition2*
Condition1 OR *Condition2*

RATLE uses the standard KBANN method to map conditions that use the logical operators NOT, AND, and OR. Parentheses in the RATLE language are used by the teacher to indicate the precedence of operators. The negation operator NOT is implemented in RATLE by recursively calling the process for mapping conditions on the condition being negated. RATLE then remembers that links from the negated condition should have negative weights. To translate the AND and OR functions, RATLE maps conditions *Condition1* and *Condition2* to units and then creates a new unit that is a conjunction (AND) or a disjunction (OR) of these units. Note that RATLE also does some network compression when mapping logical conditions; if several units are combined with ORs or ANDs, RATLE will construct a single large disjunctive or conjunctive unit, rather than a series of binary units, up to the limits specified by Towell (1991).

6.2.3 Translating Fuzzy Conditions

To map fuzzy conditions, RATLE creates a hidden unit that captures the fuzzy membership function specified. The set of possible fuzzy membership functions available to the teacher is defined by the initializer before the instruction process begins. Recall the two types of fuzzy conditions: the first type makes use of fuzzy terms that I call descriptors, and the second type makes use of fuzzy terms that I call quantifiers and properties.

Translating Fuzzy Conditions: Descriptors

Object { IS | ARE } *SimpleDescriptor*

Recall that to translate a fuzzy condition of this form, RATLE matches the string *SimpleDescriptor* (and *Object* as well) to the set of predefined descriptors. An example of such a fuzzy condition from the previous chapter is:

`Tree1 IS CloserThanTree2`

where `CloserThanTree2` was defined by the descriptor:

```
DESCRIPTOR CloserThan?(Other) ::=
          (DistTo?(Object) - DistTo?(Other)) < 0
```

Instantiating this descriptor (with ?(Object) =`Tree1`) we get:

```
CloserThanTree2 ::= (DistToTree1 - DistToTree2) < 0
```

RATLE translates this term into a unit that calculates the specified sum and uses the threshold (0 in this case) as its bias.

To do this, RATLE uses an approach similar to Berenji and Khedkhar's (1992). RATLE uses the DESCRIPTOR function's possible range of values, the descriptor's operator, and the descriptor's threshold to determine the membership function. For example, assume that `DistToTree1` and `DistToTree2` are input variables that range from 0 to 100. The sum of the calculation above (`DistToTree1 - DistToTree2`) ranges from -100 to 100. Since the threshold is 0, and the operator is $<$, I want a unit that is true when the sum is from -100 up to (but not including) 0. But, since the agent uses sigmoidal activation functions for its units, I cannot get a unit with such a discontinuity in the activation value. Instead RATLE selects two threshold values. One value represents the point at which the function will produce high activation and the other the point at which the function produces low activation.

For the above example, RATLE selects the value -0.5, below which the activation will be high, and the value 0.5, above which the activation will be low. For the region between -0.5 and 0.5 the function is "fuzzy" – values in this range partially meet the fuzzy condition. RATLE achieves this effect by weighting the input values being summed so as to calculate the appropriate function. If we define high activation as 0.9 and low activation as 0.1, RATLE would achieve the desired function by adding a link to `DistToTree1` with a weight of 4.39 and a link to `DistToTree2` with a weight of -4.39. RATLE determines these weights using the threshold values -0.5 and 0.5 and the expected activation values of 0.9 and 0.1 in Equation 3 in Chapter 2. Figure 26 shows a diagram of the resulting hidden unit.

The remaining question for RATLE is how to choose the values around which to base the membership function (the values -0.5 and 0.5 in the above discussion). To do this I again follow Berenji and Khedkhar's (1992) approach. I first calculate the range of possible values. Then I select a value up to which the membership function has low activity and a second value where the membership function starts being highly active. I chose to set these values one unit apart (unless the maximum range of the sum being calculated is less than 10, in which case I use two values that are 10% of the maximum range apart). For the "> *Number*" test, RATLE sets the low value to ($Number - 0.5$) and the high value to ($Number + 0.5$). For the ">= *Number*" test, RATLE sets the low value to ($Number - 1.0$) and the high value to *Number*. The range for >= insures that the value *Number* has a high activation, while for > the value *Number* is only somewhat active. The < and <= operations work similarly (with the obvious swaps of signs and high and low values). RATLE builds the = operation

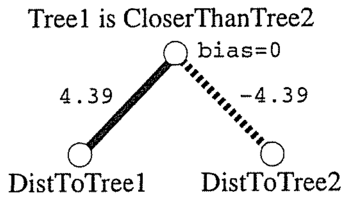


Figure 26: Hidden unit that represents the fuzzy condition `Tree1 IS CloserThanTree2`. The resulting unit is a sum of the distance to `Tree1` minus the distance to `Tree2`. This unit is neither a conjunction nor a disjunction, since the link weights and the bias are determined by the fuzzy function being mapped. Note that RATLE only creates the unit for the fuzzy condition and the weights leading into that unit; the other units are assumed to be input units.

as a conjunction of two operations: a >= and a <= operation. Similarly, RATLE builds a != as a disjunction of a < and a > operation.

If more than one fuzzy term matches the fuzzy condition, RATLE builds a unit for each term that matches and constructs a disjunction of the resulting units.

Translating Fuzzy Conditions: Quantifiers

Quantifier Object { IS | ARE } *PropertyDescriptors*

RATLE translates the other type of fuzzy condition similarly. There are two main differences with this type of fuzzy condition: properties are used to select inputs that match those properties; and the resulting function may be applied to each input that matches separately or may be summed across all the inputs that match. When the teacher specifies a fuzzy condition of this form, RATLE starts by looking for inputs of the appropriate type, and then determines which inputs match the specified properties.

For example, when the fuzzy condition is:

`Many Trees ARE (Tall AND Leafy)`

RATLE would look for input features of type `Trees` and then determine how well each of these inputs match the properties `Tall` and `Leafy`. For instance, if an input represents `Trees` of heights 40 to 60 and `Tall` is defined by the initializer as being of height 50 to 80, this input unit matches with a value 0.5 (since half of the range 40 to 60 is considered `Tall`). When there is more than one property, RATLE multiplies the match values (e.g., in the above example, if the input described above matched the property `Leafy` with a value 0.6, the total match value of that unit is $0.3 = 0.5 * 0.6$). Once RATLE determines how well each input matches the properties in the teacher's fuzzy condition, it calculates the appropriate weight as described in the previous section, but it then multiplies the weight for each link by that input's match value.

The other difference for this type of fuzzy condition is that two different types of functions can be calculated. In one function (when SUM is specified in the quantifier) RATLE calculates the appropriate function by determining all of the input units that match and then calculating the total input across all of the matching inputs. If SUM is not specified in the quantifier, RATLE calculates the appropriate function with respect to each of the matching inputs, and then creates a disjunction of each of these units. Figure 27 shows a sample interpretation of

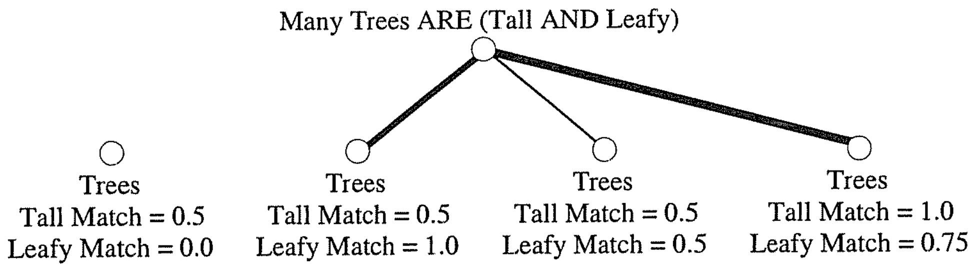


Figure 27: Unit that represents the fuzzy condition `Many Trees` ARE (`Tall` AND `Leafy`), assuming that `Many` is defined as a SUM quantifier. Note that the weights on the links from the inputs are affected by how the the input matches the properties `Tall` and `Leafy`. This unit totals the number of trees across the matching input units and then calculates the fuzzy operator associated with `Many`.

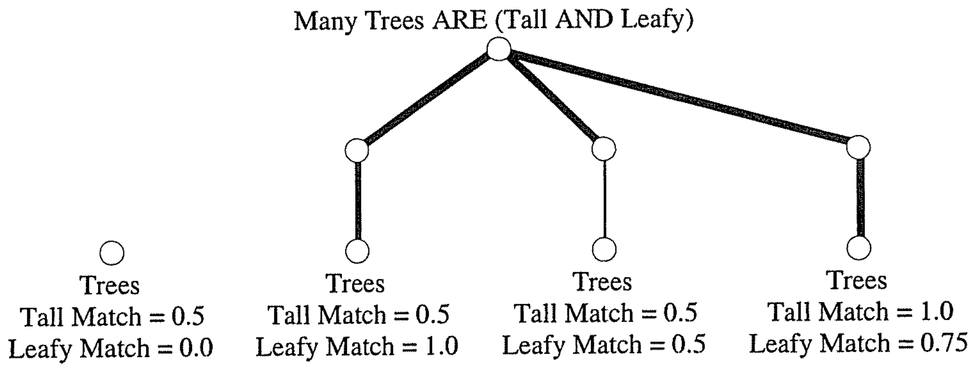


Figure 28: Unit that represents the fuzzy condition `Many Trees` ARE (`Tall` AND `Leafy`), assuming that `Many` is not defined as a SUM quantifier. RATLE creates a unit for each matching input unit to determine when individual input unit meets the condition, and then creates a disjunction to determine if there at least one matching input unit.

the condition above assuming that `Many` is defined as a SUM across all the matching input units. In this case, the resulting hidden unit totals how many `Tall` and `Leafy` `Trees` there are across all the input features to determine whether there are `Many Trees`. Figure 28 shows the same fuzzy condition should SUM not be specified in defining `Many`. Here, RATLE constructs hidden units that determine whether any of the input features that match the properties `Tall` and `Leafy` meet the criterion *Many*. The unit representing the fuzzy condition is the disjunction of all these tests.

Quantifiers may also use the operation IN. RATLE calculates an IN function as a conjunction of a >= and a <= operation.

6.3 Translating Actions

The teacher uses statements to indicate actions that the agent should take under certain conditions. As I have shown above, RATLE builds units to represent conditions that indicate when the agent should take an action. In this section, I give details on how RATLE connects conditions to the different types of actions: single actions, alternative actions, action prohibitions, and multi-step actions.

6.3.1 Translating Actions: Single Actions

Single actions occur in statements where the teacher gives a single name as an action in that statement. Recall that the teacher can also use an IF statement in two other ways, to create or add to the definitions of intermediate terms (using the INFER keyword), and to create and add to memory terms (using the REMEMBER keyword). So I will break down the set of single actions to five cases: (1) creating a new intermediate term, (2) creating a new memory term, (3) adding to the existing definition of an action, (4) adding to the definition of an intermediate term, and (5) adding to the definition of a memory term. There is no case for creating a new action since the set of possible actions is determined by the initializer before the agent starts learning.

The case where the single action is the name of a new intermediate term is straightforward. RATLE first creates a unit corresponding to the condition leading to the action. RATLE then attaches the name of the new intermediate term to this new unit. Figure 29 depicts a sample of this case. Note that should a condition correspond to a unit that already has a name (e.g., a condition that is just the name of an input), RATLE creates a new unit with a link to the condition. RATLE assigns the name of the intermediate term to this new unit.

```
IF (Small OR Medium) THEN
  INFER NotLarge
END
```

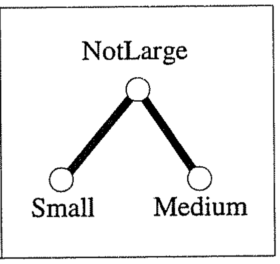


Figure 29: Translating a new intermediate term creates a new hidden unit that RATLE labels with the new intermediate term name (in this case, NotLarge).

When creating a new memory term, RATLE again constructs a unit representing the condition, but RATLE then adds a state unit with a recurrent link from the unit representing the condition. RATLE then gives the memory term name to this state unit. The unit labelled `NotLarge` in the advice in Figure 30 will be active when the condition `Small` OR `Medium` was active at the last step.

In the three remaining cases (adding a definition for an existing intermediate term, a memory term, or an action), RATLE has to add to the definition of a unit that already exists. For the first two cases (adding to an intermediate term or memory term) RATLE uses a similar method, while it uses a much different approach for adding to the definition of an action (see Figure 31).

For additional definitions of existing intermediate terms and memory terms, RATLE creates a new hidden unit. This new hidden unit is a disjunction of the old definition of the intermediate or memory term and the new definition. For actions, RATLE uses a different approach: it connects the condition associated with an action directly to the unit representing the action. I use this approach because the output units in a RL agent are not like units in a traditional KBANN network – an output unit represents the utility that the agent expects to receive by following the action associated with the output unit. Thus, creating a disjunction of the old action unit with the new condition is not appropriate, since the old action unit does not have a Boolean value.

For the units representing actions, RATLE must try to achieve the goal of the teacher. Unfortunately, there are many possible interpretations for a teacher's instruction. For example, when a teacher suggests an action, she could be indicating that the action will have a high utility. On the other hand, the teacher could be saying that the action is the best thing to do under those conditions, but that the action is merely the best of a set of bad choices – it may have a low expected utility even though it is the "best" action. A third

```
IF (Small OR Medium) THEN
  REMEMBER NotLarge
END
```

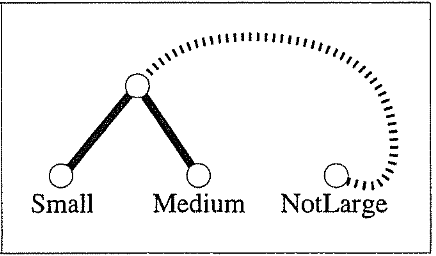


Figure 30: Translating a memory term creates a new state unit that RATLE labels with the new memory term name (e.g., `NotLarge`).

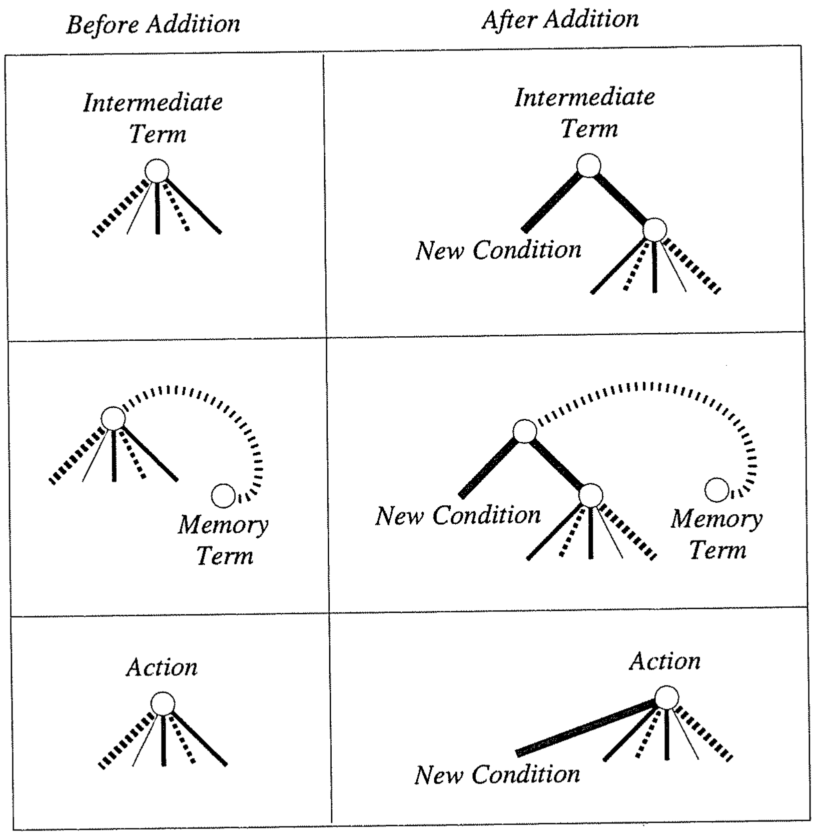


Figure 31: Adding a new definition of an existing intermediate term, memory term, or action changes the definition of an existing hidden unit in the agent's network. RATLE adds a definition to an existing intermediate term or memory unit by creating a unit that is a disjunction of the old term and the condition specifying the new definition. For an action, only a highly-weighted link from the condition to the action is added by RATLE.

interpretation is that RATLE should examine the network and reset the weights so that the suggested action is always the best choice. This approach can be faulty if the action is already considered the best of several bad choices. In this case, RATLE may conclude that no further work needs to be done since the action is already preferred. But the teacher's advice may be indicating 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 determining that the action is the "best" choice may not always capture the teacher's intentions. This approach also requires that RATLE find an appropriate set of weights to ensure that the suggested action be selected first. Determining these weights appears to be equivalent to solving a non-linear programming problem.

I chose the simplest interpretation for the teacher's recommendation of an action – RATLE

```
IF NOT AtGoal THEN
  MoveForward
END
```

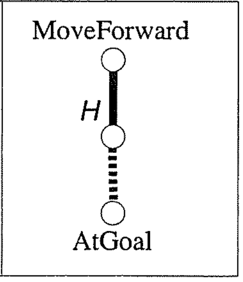


Figure 32: A negated condition leading to an action causes RATLE to introduce a new unit (H) that is true when the condition is false. RATLE then adds a positively weighted link from unit H to the recommended action.

simply connects the new condition to the action with a highly weighted link. This has the effect of increasing the predicted utility of the action under the conditions suggested by the teacher (but does not guarantee that the resulting utility will be the highest). Through empirical testing I found that weights of magnitude 2.0 are effective for these links.

Because of this approach for adding links to action units, RATLE may also have to introduce an extra hidden unit when the link being connected to the action is negated. Consider the IF statement shown in Figure 32. RATLE first maps the condition of the IF statement, representing `AtGoal` with a unit. It also remembers that the condition is negated, and therefore the weight from `AtGoal` should be negative. If RATLE then simply connected this negative weight to `MoveForward` this would cause the advice to have the effect "when `AtGoal` is true, do not move forward," which obviously does not match the instruction's intent. To avoid this problem, RATLE introduces a new hidden unit (H) that checks the condition (NOT `AtGoal`) – this unit is true when `AtGoal` is false. RATLE then creates a link from unit H to the unit for `MoveForward` with a positive weight. This link captures the idea that when unit H is true (which means that `AtGoal` is false) the utility of `MoveForward` should be increased.

6.3.2 Translating Actions: Alternative Actions

The teacher uses an alternative-action structure to indicate that more than one action could be taken under a particular condition. In the IF statement in Figure 33, the teacher indicates that both `MoveNorth` and `MoveEast` are appropriate when the condition `GoalIsNorthAndEast` is true. RATLE creates a link from the unit representing `GoalIsNorthAndEast` to the unit representing the action `MoveNorth` and another link from `GoalIsNorthAndEast` to `MoveEast`. One link is created by RATLE for each alternative action.

```
IF GoalIsNorthAndEast THEN
  (MoveNorth OR MoveEast)
END
```

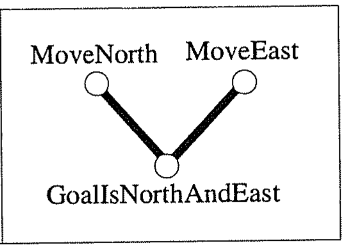


Figure 33: When the teacher indicates more than one action is appropriate given a condition, RATLE creates a link to each of the appropriate actions.

6.3.3 Translating Actions: Action Prohibitions

The teacher can indicate that the agent should not take an action. RATLE implements an action prohibition by connecting a negatively-weighted link to the action being prohibited. Note that, as with the case of single actions described above, RATLE may need to introduce a new unit if the link it is trying to attach is already negative. Figure 34 shows an example of such a piece of advice. The link from the condition `AtGoal` must be negated since the condition is (NOT `AtGoal`). Should RATLE negate the weight of this link again to represent the idea that the action is prohibited, connecting this link to `ApplyBrake` produces the effect that `AtGoal` implies `ApplyBrake`, which may be true, but does not capture the advice. RATLE solves this problem by creating a unit (J) to which it connects the negated link from `AtGoal` – unit J is true in Figure 34 if `AtGoal` is false. RATLE then adds a negative link from the unit J to `ApplyBrake` (this link reduces the utility of `ApplyBrake` when `AtGoal` is false).

```
IF NOT AtGoal THEN
  DO_NOT ApplyBrake
END
```

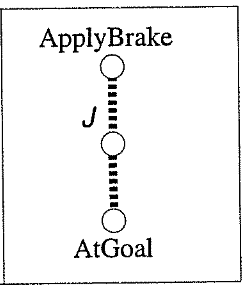
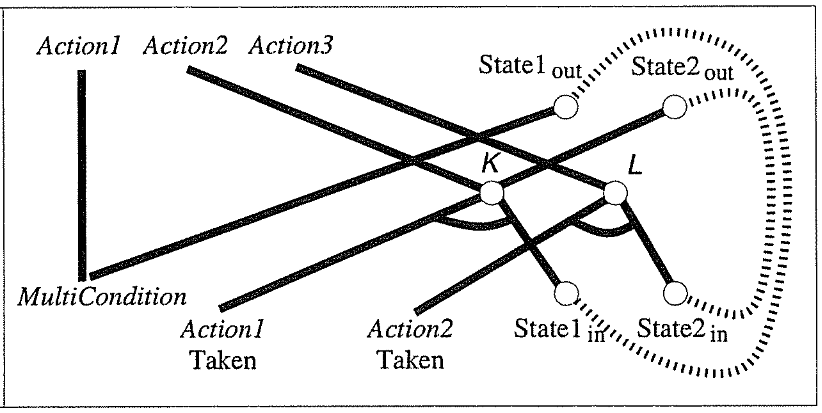


Figure 34: When the condition of an action prohibition is negated, RATLE creates a unit (J) that is true when the condition is false, and then connects unit J to the action with a negatively weighted link.

6.3.4 Translating Actions: Multi-Step Plans

```
MULTIACTION
  Action1
  Action2
  Action3
END
```



A MULTIACTION is used by the teacher to indicate a sequence of steps that the agent should take. The teacher may use a MULTIACTION in any statement where an action is allowed. In the diagram above, I show just the MULTIACTION part of the network (the condition *MultiCondition* would be determined as part of mapping the rest of the statement).

RATLE translates a MULTIACTION using a set of state units to track the step of the plan. It connects the *MultiCondition* that indicates the start of the plan to the first step of the plan (i.e., *Action1*), and creates a copy of this link that it connects to a new state unit ($State1_{out}$). This state unit remembers that the condition *MultiCondition* was true at the previous time step. RATLE then creates a unit (K) that is the conjunction of the input value of *State1* plus a link to the input unit that indicates whether *Action1* was taken in the previous state. A link is created by RATLE from unit K to *Action2*, since K indicates that the second step of the plan should be taken. A copy of this link is connected to *State2* – state unit *State2* remembers the value of unit K from the previous step. RATLE then repeats the process, creating unit L which predicts *Action3*. Note that RATLE does not have to create a unit that remembers whether *Action3* was indicated, since for the MULTIACTION shown there is no fourth action.

MULTIACTIONs require RATLE to generalize the translation of REPEAT and WHILE statements presented earlier. This is needed because a loop does not end until the last action of a MULTIACTION has been taken. When a REPEAT loop predicts a single action, RATLE construct a state unit to remember the condition predicting that the first (and only) action of the loop should be taken. But when a loop contains a MULTIACTION, RATLE uses a state unit to remember the condition that predicts the *last* action of the multi-step plan. RATLE would use this state unit to determine if the loop has just finished executing.

As an example, consider the second piece of advice from Table 19. Figure 35 shows

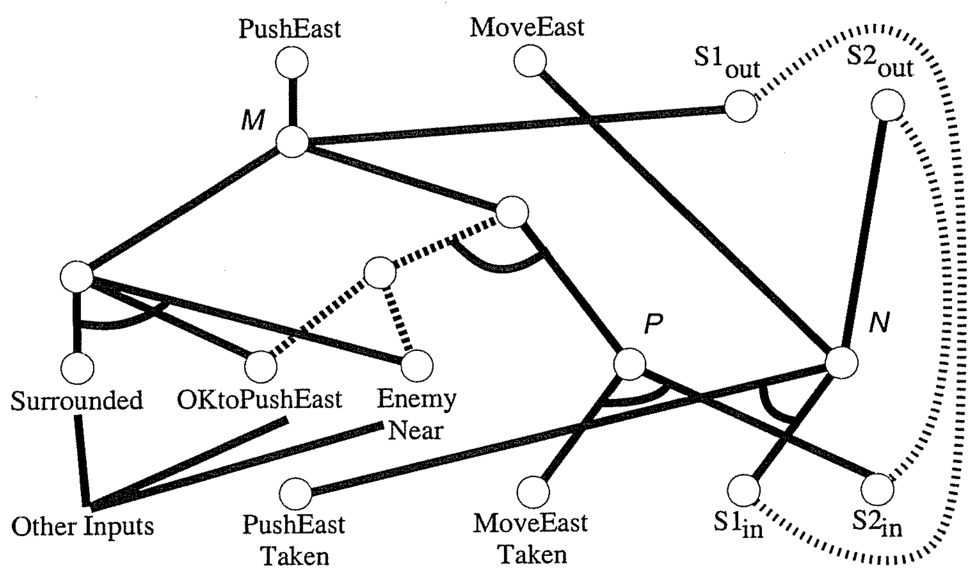


Figure 35: RATLE's translation of Table 19's second piece of advice.

RATLE's mapping of this piece of advice. This REPEAT statement contains a MULTIACTION within the loop. If the REPEAT statement had specified a single action, RATLE would have constructed a state unit to remember the value of unit M, the unit indicating that the first (and only) action of the loop should be taken. But this loop contains a MULTIACTION, so RATLE instead remembers the value of unit N with a state unit S2. Unit N recommends the last action of the MULTIACTION. RATLE creates a unit P to determine if the last action of the plan was recommended at the last step and that the agent actually took that action. Unit P is a conjunction of the input value of S2 and the unit that indicates whether the action `MoveEast` (the last action of the MULTIACTION) was taken in the last step. RATLE uses unit P to indicate that the loop has just ended.

6.4 Limitations of the RATLE Implementation

In implementing the RATLE system, I made a number of design decisions that greatly influence the performance of the system. In this section I outline the reasons for some of these decisions and discuss possible alternatives.

6.4.1 Mapping Action Recommendations

One basic issue, which I have already discussed in detail (see Section 6.3.1), was my decision to translate advice suggesting an action by simply increasing the utility of that action. There are many possible interpretations for a piece of advice, and therefore there are many ways to translate a piece of advice. My choice of increasing the utility by a fixed amount has the advantage of being simple (as opposed to some of the more complex interpretations I discussed previously). Besides using more complex mapping methods, I could also augment the RATLE language to require the teacher to indicate exactly what is meant by specifying an action (e.g., provide a keyword "BEST_ACTION" to indicate that the teacher thinks a particular action is the best). The problem with this approach is that it makes the process of building advice more difficult for the teacher.

6.4.2 Making Network Additions to Represent Advice

One possible problem with the current RATLE implementation is that each time the teacher provides advice, RATLE adds new hidden units and links to the agent's neural network. As the agent's neural network grows, the learning process becomes slower, since a larger network requires more computation. Also, an overly large neural network can often produce overfitting (Holder, 1991). One solution to this problem (see Chapter 2) introduces a penalty term to prevent overfitting, such as weight decay (Hinton, 1986), which can cause the learner to gradually remove unused links. A related approach periodically prunes the network (Le Cun et al., 1990) to find and remove hidden units that are no longer being used. Utilizing these approaches is a subject of future work. Other approaches to advice-taking that do not suffer from this problem will be discussed in Chapter 8.

Another method to deal with the problem of an ever-growing network would be to introduce a mechanism for the teacher to remove advice that is no longer in use. The teacher could give "FORGET" commands that would indicate antecedents or intermediate terms that RATLE should remove from the agent's neural network. One reason this approach is unsatisfactory is that it expects the teacher to better understand the internal workings of the agent so that the teacher can monitor when the agent's neural network has become unwieldy. I would prefer to reserve this method as a supplementary approach, allowing the teacher to suggest terms to remove, but not requiring that the teacher provide this type of advice.

6.4.3 Decaying State-Unit Activations

The activation of the state units in RATLE disappears after one time step. For example, when an agent is executing a multi-step plan, the agent must execute each step of the plan consecutively or the plan will no longer be active. This approach limits the ability of the agent to pause while executing a multi-step plan to perform opportunistic actions. In order to produce such a pause, the agent would have to use its Q-learning and experiences to learn that it is advantageous to pause, and refine the advice to reflect this understanding.

An alternative method would be to have the activation value of state units decay over a number of steps, so that the agent could resume a plan even after several intervening actions. While this approach is appealing for certain situations, in other situations the teacher may be indicating that the agent does indeed need to perform all of the steps of the plan in order, in which case having state units that decay would be undesirable. Another implementation problem of this approach is that we really want only the state unit corresponding to the current action in a multi-step plan to be active. If state units for previous actions are still partially active, the agent might take the action again, which would not capture the sequential aspect of the plan.

6.4.4 Adding "Extra" Hidden Units when Representing States

The reader may have noted that when I create state units I always add a hidden unit that records the value of the unit to be remembered, and then connect that hidden unit to an input with a recurrent link. I could simply connect the unit whose value is being remembered directly to the new input unit with a recurrent link. For example, in the diagram for translating a MULTIACTION in Section 6.3.4, I could leave out the unit $State1_{out}$ and simply connect *MultiCondition* to $State1_{in}$ directly. I chose not to implement states this way because in certain cases a state input could not be changed. If *MultiCondition* in the diagram in Section 6.3.4 was the name of input unit, and I made a direct connection from this input unit to $State1_{in}$, the activation value of this state would always be the same as the previous input unit value, since no learning occurs along the recurrent links. By introducing $State1_{out}$, I allow the agent to alter this state value, since the weight on the link between the unit representing *MultiCondition* and $State1_{out}$ can change.

6.4.5 Mapping Fuzzy Membership Functions with Sigmoidal Activation Functions

Another choice I made was to use sigmoidal activation functions to implement RATLE's fuzzy membership functions. I chose sigmoidal activation functions mostly for simplicity, since I was already using sigmoidal units for mapping other constructs. Other activation functions might make more sense for capturing fuzzy functions, especially radial basis functions (Moody & Darken, 1988, 1989). A unit using a radial basis function is defined by a prototype point in input space. The unit is more active the closer the current input vector is to the prototype (a second parameter determines how dispersed the activation function is). For a fuzzy function this would make sense, since the fuzzy membership function could be captured by selecting a point that is the "center" of the membership function and the dispersion parameter would depend on how much of the range of possible input values are to be considered part of the membership function. Choosing this type of activation function might simplify representing other types of fuzzy functions.

6.5 Summary

RATLE translates the teacher's statements in the RATLE language into additions to the agent's current neural network. It is important to remember that the resulting parts of the neural network do not have to be correct – the agent can adjust the advice, including fuzzy conditions, given its subsequent experiences.

To translate advice, RATLE works like the KBANN algorithm. RATLE converts the condition of a statement to a corresponding unit that is true when the condition is true (i.e., highly active) and false (i.e., inactive) otherwise. The resulting unit, which may be combined with other units as part of mapping a complex statement, leads to an increase in the the expected utility of the action suggested by the teacher.

RATLE extends the KBANN algorithm in a number of ways. One, advice in the RATLE language can contain multi-step plans that not only indicate the current action to take, but also subsequent actions to take. Two, RATLE uses state units to implement these plans in a way that is transparent to the teacher. Three, loops in the form of REPEAT and WHILE statements may also appear in the teacher's instructions – RATLE maps these statements using state units that remember the state of the loop from the previous step.

A teacher may also give advice about actions to avoid. RATLE uses weighted links to lower the utility of the prohibited action under the appropriate circumstances. The teacher can also make use of fuzzy terms in her statements. These fuzzy terms (which are generally problem-specific) are defined prior to the start of the instruction process. RATLE uses these fuzzy terms to build fuzzy membership functions that can then be adjusted during subsequent learning.

Once RATLE completes the process of translating the teacher's instructions, it performs KBANN's final process: RATLE adds connections between units at adjacent levels of the network that are not already connected (see Section 2.2 for more details). These connections allow RATLE to refine the teacher's recommendations by adding new antecedents to the conditions the teacher has specified.

Since advice is represented with differentiable sigmoidal units, the agent can refine the connections mapping the advice, including the connections defining fuzzy terms, using back-propagation learning. Thus the agent is able to evaluate and alter the teacher's advice through its experiences performing the task.

Finally, RATLE implements the advice translation process as a series of *additions* to the agent's network. This means that the teacher may instruct the agent any number of times, with each new set of advice introducing new units and links to the agent's network.

Chapter 7

Experimental Study of RATLE

In this chapter, I present experiments to evaluate the RATLE system described in Chapters 4, 5, and 6. The primary goal of these experiments is to explore the basic question of this thesis: does a technique for refining *procedural* domain theories show the same benefits (reviewed below) that techniques for refining non-procedural domain theories show? A secondary focus of my experiments is to exercise the features of RATLE, both to demonstrate their use, and to evaluate their effectiveness. I performed my experiments on two simulated environments: (1) an environment similar to the Pengo video game, and (2) a Soccer environment. Below I outline my hypotheses for the experiments and then discuss the important characteristics of these testbeds.

We generally expect to see two benefits from a theory-refinement technique: (1) instruction should provide a "head-start" for the inductive learner, and (2) the refined domain theory should be more accurate than the original theory. By "head-start" I mean that a student employing advice should see a gain in performance (i.e., cumulative reward) that a student without instruction would only achieve after seeing more task examples (e.g., a student who has driving explained to him should learn faster than a student who has to learn to drive without any instruction). Note that we cannot expect that the student with instruction will forever outperform the student without instruction. Since both students are able to learn, and there is generally a limit to how well one can perform a task, it is possible that both students will eventually achieve the same performance.

I use my testbeds to demonstrate that RATLE produces the two effects discussed above: (1) the RATLE agent shows a gain in performance after receiving advice that a standard RL agent would only achieve through lengthy exploration, and (2) the RATLE agent outperforms an agent that is unable to refine the advice (i.e. an agent that assumes that the advice is always correct in any situation where it applies). I will also analyze the results to show that RATLE agents produce effects that are consistent with the goals of the advice. Other experiments demonstrate that the effects of advice are independent of when the teacher presents the advice, that the agent can overcome the effects of "bad" advice, and that the agent is able to make use of a subsequent piece of advice (i.e., a piece of advice presented after the agent has received an initial piece of advice and returned to reinforcement learning).

In the next section of this chapter I present the extensive experiments I performed on the Pengo testbed. I will outline the important aspects of the task environment, the methodology I use in my experiments, and the results, then present some discussion of those results. Following that, I present supporting results on the Soccer testbed.

7.1 Experiments on the Pengo Testbed

I used the Pengo testbed to evaluate the hypotheses I present above. This testbed is similar to those explored by Agre and Chapman (1987) and Lin (1992). Using a similar testbed allows me to build on this previous work. In particular, I can make use of parameter settings empirically determined by Lin.

Like Agre and Chapman's work, my Pengo testbed is based on the Pengo video game. One major difference between my task and theirs is that my agent has only a limited sensor array – the agent is unable to see through objects in its maze-like world, it only receives agent-centered information about objects in direct line-of-sight. In Agre and Chapman's task the agent has an "aerial view" of the entire environment (i.e., the view of a player of this video game). I chose to provide limited sensor information to my agent so that the results would be applicable to real-world robot agents that have similarly limited sensors.

My Pengo testbed is also similar to Lin's in that he also uses a limited set of sensors, though his sensors are not constrained by line-of-sight limitations. My task differs from Lin's in that I have a more complex set of reinforcement signals. In Lin's task the agent focuses only on evading capture by enemies. In my task the agent receives reinforcements in three situations: when it is captured by an enemy, when it picks up food, and when it eliminates an enemy. This more complex set of reinforcement signals makes the Pengo task more difficult to learn, since the agent must learn to balance the different reinforcement signals.

7.1.1 The Pengo Environment

Figure 36a illustrates the Pengo environment. 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

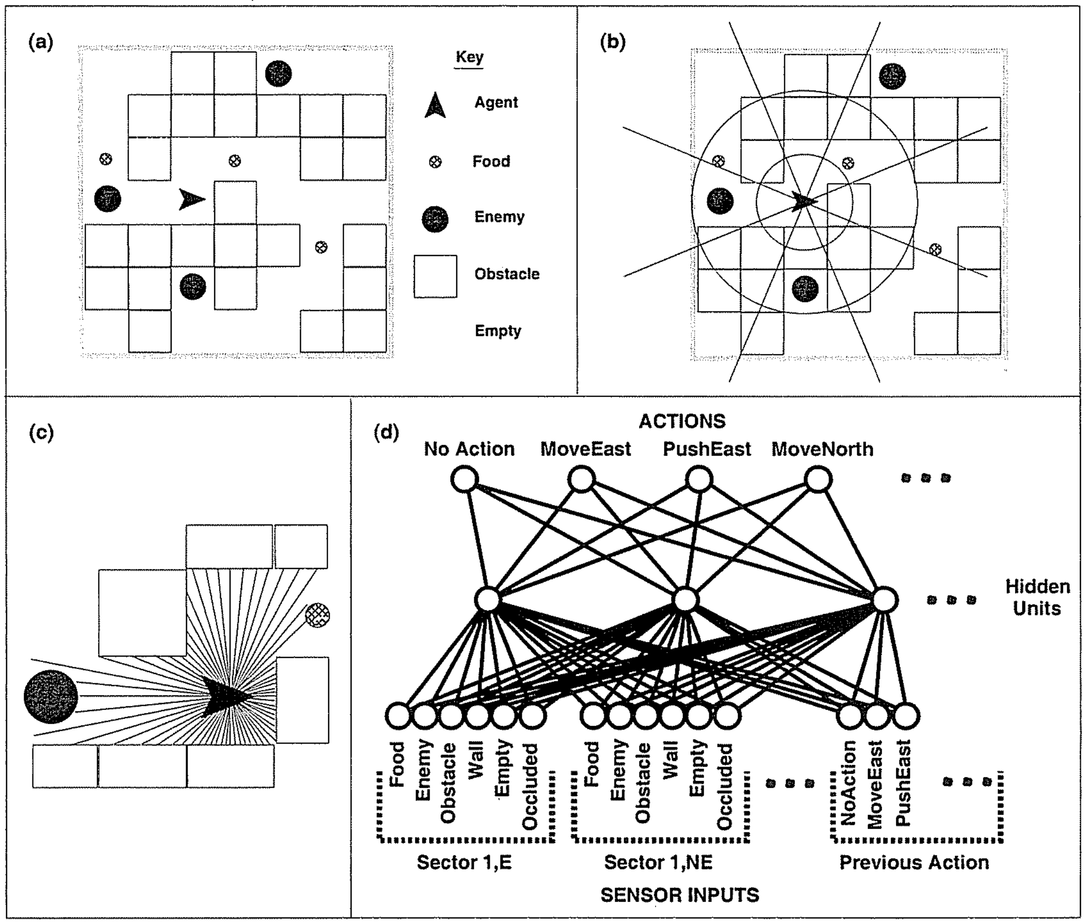


Figure 36: The Pengo environment: (a) sample configuration, (b) sample division of the environment into sectors, (c) distances to the nearest occluding object along a fixed set of arcs (measured from the agent), (d) a neural network that computes the utility of actions. (See text for further explanation.)

edge of the board. If the obstacle is unable to move (there is an obstacle or wall behind it), the obstacle disintegrates when pushed. Food is eaten when the agent or an enemy touches it.

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 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. The agent, enemies, and food are randomly deposited on the initial 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. More details on these processes appear in Appendix A.

The agent receives reinforcement signals when: (1) an enemy eliminates the agent by touching the agent (-1.0), (2) the agent collects one of the food objects (+0.7), and (3) the agent destroys an enemy by pushing an obstacle into it (+0.9). I chose these values empirically during my initial experimentation with the baseline learning agents (i.e., agents without advice).

As mentioned above, I 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 36b). I define each sector 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, I 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 36c). 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 reflecting off an object of the given type, divided by the maximum number of arcs that could end in the sector. So for example, given Figure 36b, the agent's percentage for "obstacle" would be high for the sector to its right. The agent also calculates how much of each sector is empty and how much is occluded. The agent also records as input which action the agent took in the previous state. Appendix A presents the definitions of the input and and fuzzy terms used in these experiments.

The sector percentages and previous action features discussed above constitute the input to the agent's neural network (see Figure 36d). The agent's sensors for these experiments measure objects in 32 sectors (four "rings" divided into eight equal "wedges", where a ring is defined by a minimum and maximum distance and a wedge is defined by a a minimum and maximum angle). In each sector I count the percentage taken up by each of the six objects. There are nine output units for the network representing the nine possible actions.

7.1.2 Methodology

I train the RL agents for a fixed number of *episodes* for each experiment. An episode consists of placing the agent into a randomly generated, initial Pengo environment, and then allowing it to explore until it is captured or a threshold of 500 steps is reached. I report my results by training episodes rather than number of training actions because I believe episodes are a more useful measure of "meaningful" training done – an agent having collected all the food and eliminated all 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 signals. In any case, for all of my results the results appear qualitatively similar when graphed by the number of training actions (i.e., the agents take a similar number of actions per episode during training).

Each Pengo environment contains a 7x7 grid with approximately 15 obstacles, 3 enemy agents, and 10 food items. I use three randomly generated sequences of initial environments as a basis for the training episodes. I train 10 randomly initialized networks on each of the three sequences of environments; hence, I report the averaged results of 30 neural networks. I estimate the average total reinforcement (the average sum of the reinforcements received by the agent)[1] by "freezing"[2] the network and measuring the average reinforcement on a testset of 100 randomly-generated environments. This same set of 100 environments is used for all of the test results reported in the next section.

I chose parameters for the 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, I experimented with various numbers of hidden units, settling on 15 since that number resulted in the best average reinforcement for the baseline system. I also experimented with giving the system recurrent units (recall that RATLE uses recurrent units to represent multi-step plan and loop statements), but these units did not lead to improved performance for the baseline system and, hence, my baseline results are for a system without recurrent links. I further experimented by giving the baseline network extra hidden units after initial training (to simulate the effect of adding units to the network when advice is given), but these extra hidden units did not produce gains in performance, so my

[1]I 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 this chapter.

[2]I freeze a network during testing by turning off learning, which means that the weights and biases in the network remain unchanged.

baseline system does not receive extra hidden units during training.

After choosing an initial network topology, I then spent time acting as a teacher in RATLE, observing the behavior of the agent at various times. Based on these observations, I wrote several collections of advice. For use in my experiments, I chose four sets of advice, two that use multi-step plan and loop plans (referred to as *ElimEnemies* and *Surrounded*), and two that do not (*SimpleMoves* and *NonLocalMoves*). Figures 37, 38, 39 and 40 pictorially depict these four sets of instructions. Appendix A shows the advice as RATLE statements, as well

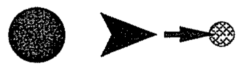


Figure 37: *SimpleMoves* advice for the Pengo agent. This advice suggests the agent move towards any Food near to the agent. The advice also suggests the agent move anyway from any Enemy near to the agent.

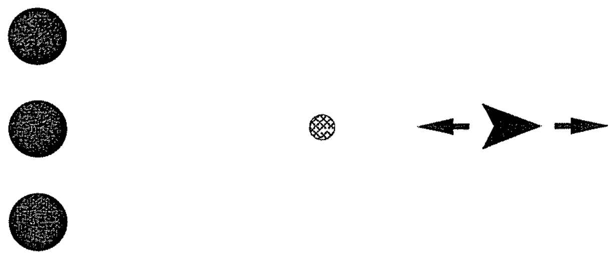


Figure 38: *NonLocalMoves* advice for the Pengo agent. This advice suggests running away if there are many Enemies in one direction, but it also suggests moving towards Food even if there is an Enemy in that direction (as long as the Enemies are distant).

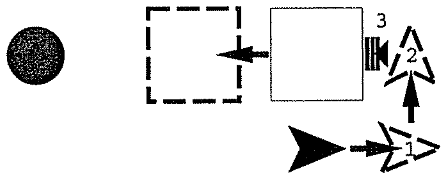


Figure 39: *ElimEnemies* advice for the Pengo agent. This advice gives a three-step plan for eliminating Enemies involving moving behind convenient Obstacles and pushing them towards the chosen Enemy.

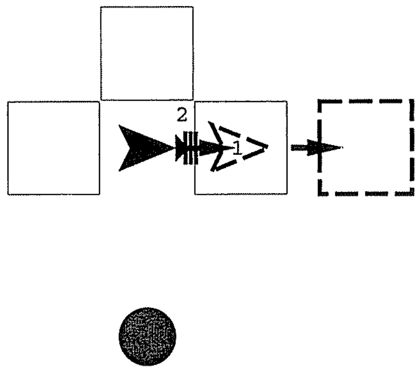


Figure 40: *Surrounded* advice for the Pengo agent. This advice uses a loop and a two-step plan for escaping when the agent is surrounded by `Obstacles` and `Enemies`. The plan involves pushing the nearest `Obstacle` out of the way and moving into the vacated spot until the agent is no longer surrounded.

as the input and fuzzy terms used in these statements.

The advice *SimpleMoves* provides advice about short-term reinforcement situations: when the agent is near to a food item or an enemy. *NonLocalMoves* gives advice about more strategic situations: moving away from groups of enemies and collecting food as long as the food is not too closely guarded. Eliminating enemies is extremely difficult since the agent is unable to see through obstacles to determine when there is an enemy behind the obstacle; the advice *ElimEnemies* provides a multi-step plan to overcome this difficulty. *Surrounded* provides advice in the form of a multi-step plan within a loop that applies to environments where the agent is trapped; the advice indicates how to get out of this trap.

7.1.3 Results

Hypothesis: An agent can make use of advice

In my first experiment, I evaluate the hypothesis that a RATLE agent can in fact make use of advice. After 1000 episodes of initial learning, I judge the value of (independently) providing each of the four sets of advice to my agent via RATLE. I train the system for 2000 more episodes after adding the advice, then measure the average cumulative reinforcement on the testset. (The baseline is also trained for 3000 episodes.) Table 24 reports the averaged testset

Table 24: 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
None (baseline)	1.32
SimpleMoves	1.91
NonLocalMoves	2.01
ElimEnemies	1.87
Surrounded	1.72

Table 25: Mean number of enemies captured, food collected, and number of actions taken for the experiments summarized in Table 24.

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

reinforcement; all gains over the baseline system are statistically significant[3]. Note that the gain is higher for the simpler pieces of advice, *SimpleMoves* and *NonLocalMoves*, which do not incorporate MULTIACTION or loop constructs. This suggests the need for further work on taking complex advice; however the multi-step advice may simply be less useful.

Each of my 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 25 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, the *ElimEnemies* advice leads to a much larger number of enemies eliminated than the baseline or any of the other pieces of advice.

Hypothesis: The effects of advice are independent of when the advice is given

In my second experiment I investigate the hypothesis that the observer can beneficially provide advice at any time during training. To test this, I insert the four sets of advice at

[3]All results reported in this chapter as statistically significant are significant at the $p < 0.05$ level (i.e., with 95% confidence).

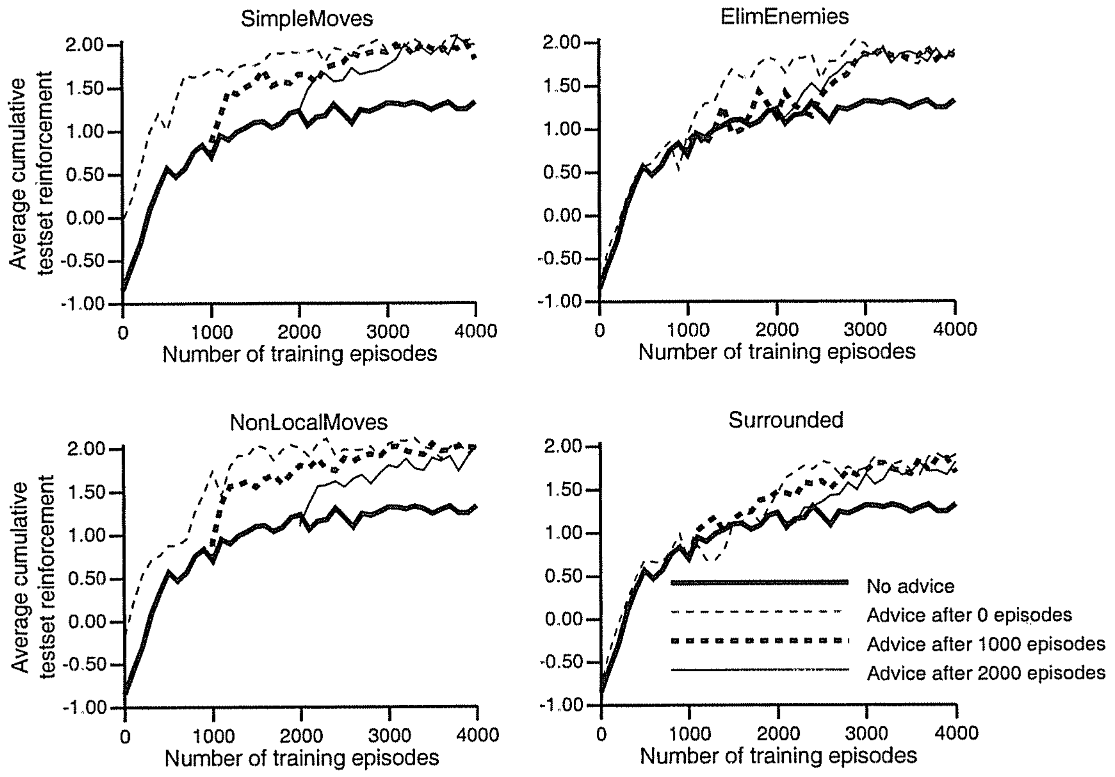


Figure 41: Average total reinforcement for my four sample pieces of advice as a function of amount of training and point of insertion of the advice.

different points in training (after 0, 1000, and 2000 episodes). Figure 41 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.

Hypothesis: The ability to refine advice is important

In my third experiment, I investigate an approach for using advice where the advice is not refined. This simple strawman algorithm exactly follows the observer's advice when it applies; otherwise it uses a "traditional" connectionist Q-learner to choose its actions. When evaluated on the test set, the strawman employs a loop similar to that shown in Table 23, with the following differences: Step 6 (incorporating advice) is left out; and 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 of these randomly,
   Else choose the maximum utility action from the connectionist Q-network.
```

Table 26: Average testset reinforcement using the strawman approach of literally following advice compared to the RATLE method of refining advice based on subsequent experience.

Advice	STRAWMAN	RATLE
SimpleMoves	1.63	1.91
NonLocalMoves	1.46	2.01
ElimEnemies	1.28	1.87
Surrounded	1.21	1.72

The performance of this strawman is reported in Table 26. In all cases, RATLE performs statistically significantly better than the strawman on cumulative reinforcement received. 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).

Hypothesis: An agent can profitably make use of subsequent advice

In my fourth experiment, I investigate the hypothesis that subsequent advice (i.e., advice provided after the agent has already incorporated and refined an initial piece of advice) will lead to further gains in performance. To test this hypothesis, I supplied each of my four pieces of advice to an agent after 1000 episodes (as in my 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 get statistically significant results. Table 27 shows the results of this experiment.

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. I suspect this occurs due to a combination of factors: (1) there is an upper limit to how well the agents can do – though it is difficult to estimate this upper bound; (2) the pieces of advice interact – they may suggest different actions in different situations, and in the process of resolving these conflicts, the agent may use one piece of advice less often than it would if there was only one piece of advice; and (3) 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 27: Average testset reinforcement for each of the possible pairs of my 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 (for total of 4000 training episodes). Shown in parentheses next to the first pieces of advice are the performance results from my 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 receives just the first piece of advice.

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

Hypothesis: RATLE agents can overcome the effects of "bad" advice

In my fifth experiment, I investigate the effect that "bad" advice has on a RATLE agent. To form the advice used in these experiments I altered the advice *SimpleMoves.* I produced one set of bad advice, *NotSimpleMoves*, that is similar to *SimpleMoves*, except that whenever *SimpleMoves* suggests an action, *NotSimpleMoves* suggests that the agent should *not* take that action (using the DO_NOT construct). My second set of bad advice, *OppositeSimpleMoves*, is also similar to *SimpleMoves*, except that whenever *SimpleMoves* suggests an action, *OppositeSimpleMoves* suggests that the action in the opposite compass direction (e.g., when *SimpleMoves* suggests the agent should `MoveEast`, *OppositeSimpleMoves* suggest the agent should `MoveWest`). The actual constructs used to define all three pieces of advice appear in Appendix A.

I tested these two pieces of "bad" advice by separately adding each after 1000 training episodes and then training the agent for 3000 more training episodes. Figure 42 presents the results of these experiments. These results demonstrate that while bad advice does in fact cause an initial immediate drop in performance, the agent is quickly able to refine the advice to return to reasonable behavior. It is also interesting to note that, over time, the agent may actually produce gains in performance from the "bad" advice (compared to learning without advice). After 4000 training episodes, the advice *OppositeSimpleMoves* produces a statistically significant gain in performance over the agents with no advice, but produces significantly worse performance than the original *SimpleMoves* advice. The advice *NotSimpleMoves* does not produce a significant gain over not receiving advice after 4000 training

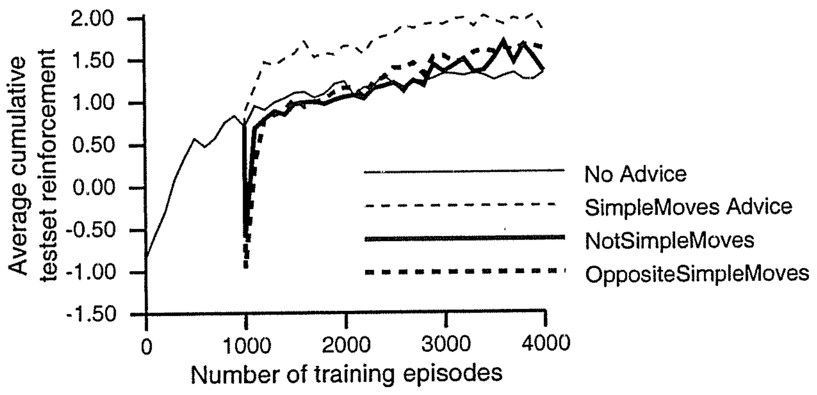


Figure 42: Average total reinforcement for the baseline as well as agents with the advice *SimpleMoves*, *NotSimpleMoves*, and *OppositeSimpleMoves*.

episodes. The improvements over the no advice case are likely due to the fact that, while I changed the predicted actions of *SimpleMoves* to produce the bad advice, the intermediate terms constructed in forming the bad advice (i.e., the hidden units that trigger the actions) are the same in all three sets of advice; the agent can take advantage of these intermediate inferences to learn the correct actions to take.

Hypothesis: Replay will improve the performance of RATLE's agents

In my final experiment, I evaluate the usefulness of combining my advice-giving approach with Lin's "replay" technique (1992). Lin introduced the replay method as a means 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) reports that replay can be used 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, my experiment addressed two related questions: (1) does the advice provide any benefit over simply reusing the agent's experiences multiple times?; and (2) could my approach benefit from replay, for example, by needing fewer training episodes to achieve a given level of performance? My hypothesis was that the answer to both questions is "yes."

To test my hypothesis, I implemented two approaches to replay in RATLE and evaluated them using the *NonLocalMoves* advice. In one approach, which I will call *Action Replay*, I 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 random order. A second approach (similar

Table 28: 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 (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 Reinforcement
Standard RATLE (no replay)	1.74
Action Replay Method	1.48
Sequence Replay Method	1.45

to Lin's), which I will call *Sequence Replay*, is more complicated. Here, I keep the last N *sequences* of actions that ended when the agent received a non-zero reinforcement. Once a sequence completes, I train on all of the saved sequences, again in a random order. To train the network with a sequence, I 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 utility of states nearest reinforcements are best estimated by the network. Results for keeping 250 state-action pairs and 250 sequences[4] appear in Table 28. Due to time constraints, I trained these agents for only 1000 episodes. Note that for replay this represents much more training than the standard system receives, since after each actions or episodes the agent is trained with multiple actions or multiple episodes. For example, with the Action Replay approach, each state-action pair is used in training 250 times, so there is 250 times as much network training for this approach.

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

[4]I also experimented with keeping only 100 pairs or sequences; the results using 250 pairs and sequences were superior.

(i.e., [0.4,**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 I 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 I repeat this training a number of times, the Q values for other actions in a state may become greatly distorted. 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. I plan to further investigate this topic, since replay intuitively seems to be a valuable technique for reducing the amount of experimentation on the external world that an RL agent has to perform.

7.1.4 Discussion of the Pengo Experiments

My experiments demonstrate that: (1) advice can in fact improve the performance of an agent, (2) advice produces approximately the same resulting performance no matter when it is added, (3) it is important for the agent to be able to refine advice based on subsequent experience, (4) a second piece of advice can produce further gains, (5) "bad" advice can produce an initial drop in performance that the agent can overcome with learning, and (6) a straightforward approach to replay does not produce significant benefits. One key question that might arise from the results presented is why the baseline agent does not eventually achieve the same results that the system with advice achieves? To answer this question, I will address a more general question – what effect do I expect advice to have on the agent?

A connectionist Q-learner uses gradient-based learning to select a function that minimizes the error between the predicted and estimated Q values:

$$Error \;=\; \sum_{e \in Examples} \left[\tilde{Q}(s_e, a_e) \;-\; \hat{Q}(s_e, a_e)\right]^2 \tag{13}$$

where $\hat{Q}$ is the estimate obtained by actually taking an action and looking at the predicted utility for all of the actions in the resulting state, and $\tilde{Q}$ is the current Q value predicted by the learner's Q function, and s_e and a_e are the state and action chosen in example e. One problem with this approach is that the learner may select a minimum through gradient-based learning that is "good," but sub-optimal with respect to the optimal Q function (represented by a Q table). For example, assume that for the vast majority of the states and actions that the reward is zero, and that, as a consequence, most of the Q values are zero. In this case it may be advantageous to the network to always predict zero, since the total error for predicting zero all the time would be relatively small. One hopes that the network will be able to find features of those states that have non-zero Q values that distinguish those states, but if these states are very similar to other states with zero Q values this may not occur. Clearly, it is possible for a connectionist Q learner to settle on a Q function that is less than optimal.

When I introduce "good" advice into an agent, I expect it to have one or more of several possible effects. One possible effect is that the advice will change the network's predictions of some of the Q values to ones 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 (or decreasing) certain Q values the advice changes which states are explored by the agent. In this case, good advice will 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. A third possible effect is that the addition of advice alters the weight space of possible solutions that the learner is exploring. This occurs because the new weights and hidden units locally change the space that the learner is in. 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-based 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, I 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 my experiments, I see the effect of speeded convergence in the graphs of Figure 41, where the advice, generally after a small amount of training, leads the agent to achieve a high level of performance quickly. These graphs also demonstrate the effect of convergence to a better solution. Note that these effects are a result of what I would call "good" advice. It is possible that "bad" advice could have equally deleterious effects. So, a related question is how do I 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 43. This example demonstrates

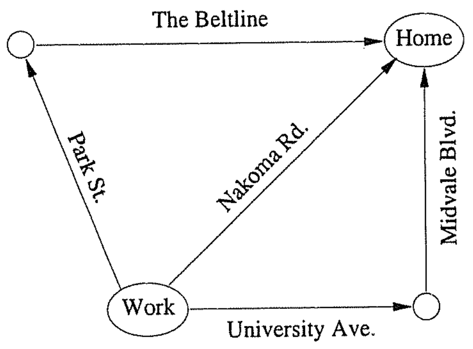


Figure 43: A sample problem where advice can fail to enhance performance. Assume the goal of the agent is to go from Work to Home, and that the agent receives a large reward for taking Nakoma Road, and a medium-sized reward for taking University Avenue followed by Midvale Boulevard. If the agent receives 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 likelihood of achieving reward). Thus it may take a long time for the learner to try Nakoma Road often enough to learn it is a better route, since every time the learner follows the advice it will not learn much. A learner without advice might try both routes equally often and quickly learn the best route.

a case where advice, though providing useful information, could in fact cause the agent to take longer to converge to an optimal policy. This example suggests that I may want to re-think the stochastic mechanism I use to select actions, but other mechanisms seem to suffer from similar problems. In any case, it thus appears that defining the properties of "good" advice is a topic for future work. As a first (and admittedly vague) approximation, I would expect advice to be "good" when it causes one of the effects mentioned above: (1) it reduces the overall error in the agent's predicted Q values, (2) it causes the agent to pick actions that lead to states that are important in finding a solution, or (3) it transforms the network's weight space so that the agent is able to perform gradient-based learning to find a better solution.

7.2 Experiments on the Soccer Testbed

I chose to experiment on a Soccer testbed to confirm the results I obtained in the Pengo testbed. In designing the Soccer testbed, I deliberately set out to build a task that was significantly different from the Pengo environment. Below I discuss the differences between these tasks. Using this testbed, I will show that an agent receiving advice outperforms an agent that does not receive advice. I also discuss why the agent refining the advice outperforms an agent that does not refine the advice.

One major difference between the Pengo and Soccer testbeds is that the agent in the Soccer task receives complete information; no objects are occluded in the Soccer environment. The agent's sensors are also different. The agent in Pengo received information about the occupancy of various sectors around the agent; the agent in Soccer receives a set of measures indicating the distance and direction to all of the other objects in the environment. The tasks also differ in that Soccer has cooperating agents in addition to competing agents. Finally, the reinforcement signals are more sparse in the Soccer environment; the agent receives a positive signal when it scores and a negative signal when the other agent's score. These signals are made more difficult to interpret by the agent because there is more than one agent per team and the positive reinforcement for a goal is received by all of the agents on the scoring team (similarly, the negative reinforcement for allowing a goal is received by all the players on a team).

7.2.1 The Soccer Environment

Figure 44 shows the basic Soccer environment. Each player can perform eleven actions: the agent may *move* in each of the directions Forward, Back, Left, and Right; the agent may

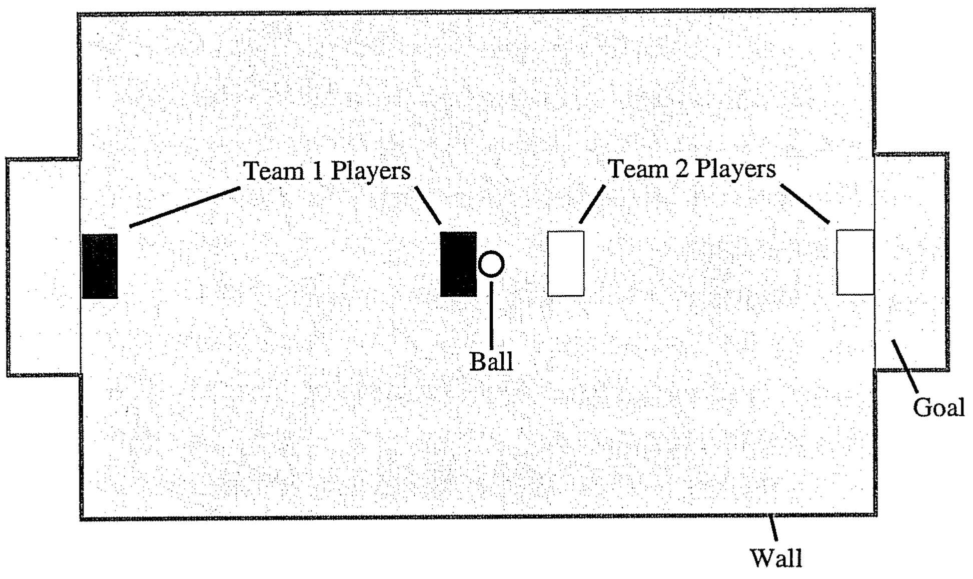


Figure 44: The Soccer test environment. Each player wants to kick the ball into the opponent's goal while preventing the ball from going into their own goal.

kick the ball in each of the directions Forward, Back, Left, Right, ForwardLeft (diagonally) and ForwardRight; and the agent may *do nothing.* Moves cause the agent to change position on the field, while kicking actions cause the ball to leave the agent (who stays still) and roll on the field. All of the moves (both by players and by the ball) take place simultaneously. Moves that would cause two players to collide are resolved randomly (only one player may occupy a location on the field at the time). I assume that the Soccer game is played in an indoor (walled) arena, so that the agent is not able to move off the field of play, and that a ball kicked against the wall ricochets off. A kicking action affects the ball only when the agent actually controls the ball. An agent controls the ball when the agent moves in the path of the moving ball or when the agent moves into the location where the ball currently rests. Should two players collide while one has the ball, the other player gains control of the ball half of the time. A score occurs when the ball rolls into one of the goals.

The players receive a positive reinforcement (1.0) when a goal for their team is scored and a negative reinforcement (-1.0) when a goal is scored against them. I do not differentiate goal-scoring situations; a player receives a positive reinforcement when a goal is scored for its team even if the player had no part in scoring the goal (either because some other player on its team scored, or because some player on the other team mistakenly kicked the ball in their own goal).

In the experiments I performed, there are two agents on each team. In order to reduce the complexity of the task, I focused on learning in situations which I call *breakaways* (see Figure 45). A breakaway occurs when one team has control of the ball and only one of the players on the opposing team is in position to defend their goal. I will refer to the team that initially has control of the ball as the *attacking team* and the other team as the *defending team.* The breakaway is defined to last for 20 actions or until either team scores. I further limited the complexity of the task by only allowing the attacking team to learn. The attacking team uses reinforcement learning, while the defending team follows a hand-crafted policy I developed. In my hand-crafted policy, the player that is initially in position to defend acts as a goalie, keeping itself between the ball and the goal. The other player rushes back until it too can act like a goalie.

Recall that in the Soccer testbed, unlike the Pengo testbed, each agent has complete information about the environment. Each agent receives information describing the distance and direction to the following objects in the environment: the ball; the players on the opposing team; the other player on the same team; and the left post, right post, and center of each goal. Each agent also has input features representing the distance to each wall,

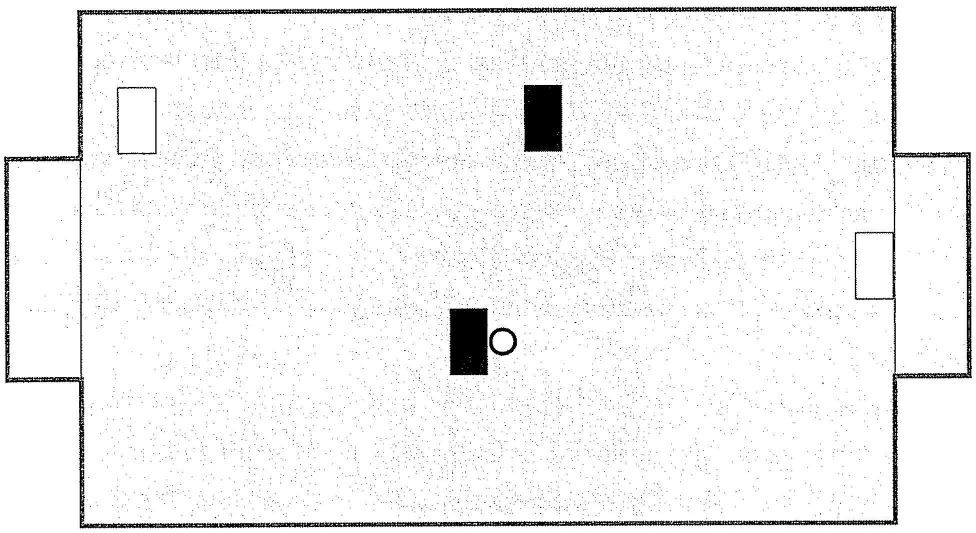


Figure 45: An example of a *breakaway* Soccer environment. The attacking team (black) has control of the ball, and only one of the defenders is between the attackers and the defenders goal.

whether or not that player has the ball and a counter indicating how much time is left for the team to score. I simultaneously represent each distance in two ways: as the x and y differences between the agent's current position and the object observed; and as the distance, plus the *sine* and *cosine* of the angle to the object. I include both representations because different values are useful in different situations, and the agents do no currently have the ability to calculate one representation from the other. Appendix B describes the input and fuzzy terms used in these experiments.

7.2.2 Methodology

I trained the agents for 400,000 games. Each game starts with an initial randomly generated breakaway situation, and continues until either team scores or until 20 actions have been taken. Similar to the Pengo experiments, I report results by the number of breakaway situations rather than the number of training actions taken; the results are qualitatively similar when graphed by number of actions.

Each breakaway environment is an 11x7 grid with two players on each team. Initially, one of the attacking team players has the ball. In a breakaway environment, I distribute the attacking team players randomly in the center five columns of the field, and the defending team players are split, one randomly in the three far-left columns of the field, and one randomly in the three far-right columns.

I generated 40 sequences of random breakaway situations and trained one attacking team for each sequence; my results are averaged over 40 attacking teams. To test the teams, I freeze the attacking team's neural networks and count the number of times the different teams win for a sequence of 100 randomly generated breakaway environments. The identical environments are used for all tests. I report the net number of goals for the attacking team (i.e. the number of games the attacking team scored minus the number of games where the defending team scored).

I use the same parameters for connectionist Q-learning that I used in the Pengo experiments. The learning rate for the network is 0.15, with a discount factor of 0.9. My baseline network (chosen empirically) has five hidden units for each action (for a total of 55 hidden units), and I connect each hidden unit to all of the inputs. I chose to use a policy network that has a separate sub-network for each action because tests with a completely connected

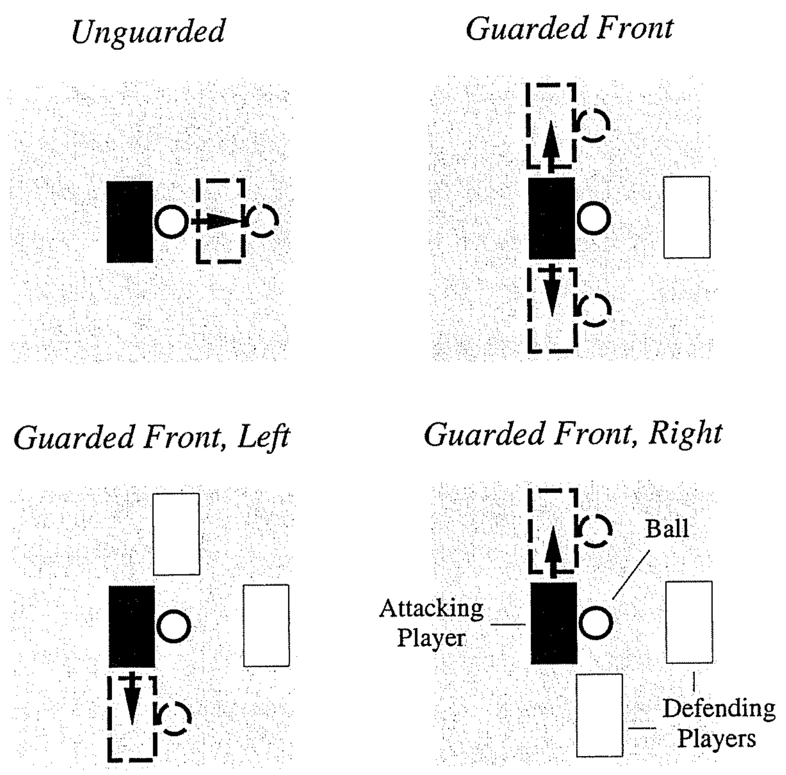


Figure 46: Advice for soccer players. The advice characterizes situations where the player has the ball according to how well guarded the player is; it suggests a move for the various situations.

network produced very poor performance, apparently for the reasons I mention in the discussion in Section 7.1.4 about the value of advice – the reinforcements signals are so sparse in this environment that the network often settles on a solution that simply predicts zero future reward. Separating the actions ameliorates this effect.

The advice I gave to the attacking team is shown pictorially in Figure 46. I give the same advice to both players on the attacking team. The advice suggests moves that a player should make when they have the ball and depending on how many opposing players are guarding the player. Appendix B includes the actual statements used to give the advice.

7.2.3 Results and Discussion

Figure 47 shows the average net testset games won by the team using the advice pictured in Figure 46 and the net test games won by a baseline team (without advice). Note that both teams play against my hand-crafted defending team (described above). The resulting gains in performance for the team using advice over the team without advice are significant for all of the results after 90,000 training games. These results show that agents can profitably

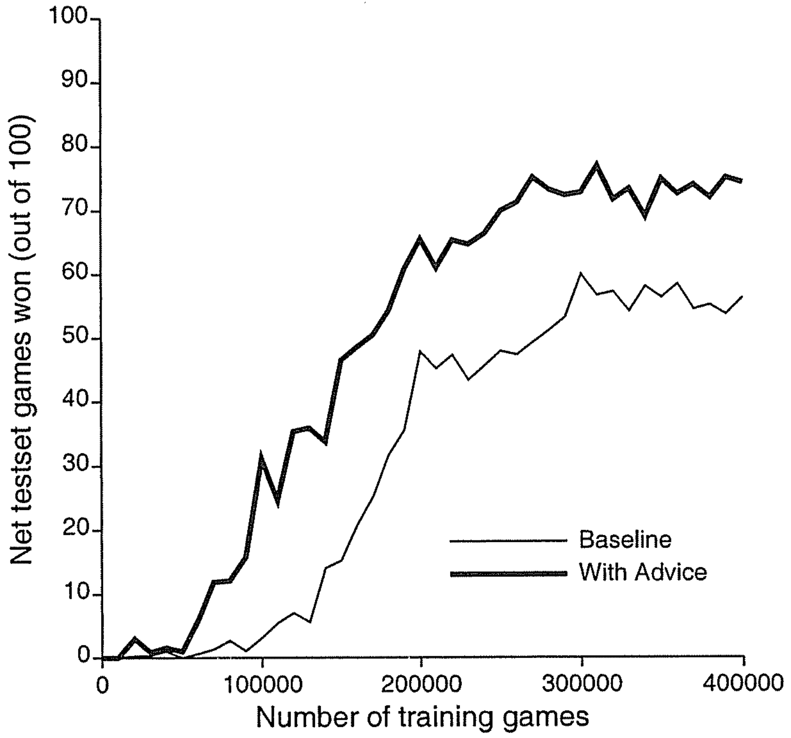


Figure 47: Net testset games won, as a function of training games, by an RL team with advice and a baseline RL team without advice.

make use of advice in an environment where multiple agents learn.

I also performed tests to determine how a team of agents that are not able to refine the advice performs. As with the strawman I constructed in the Pengo task, I constructed agents that follows the advice when any is applicable, and otherwise choose the highest utility action according to the networks trained for the baseline above. In experiments with this strawman, a team of agents following this approach *never* scored. In my advice I suggested that a player move forward whenever it has the ball and it is unguarded. But this prevents the player from shooting when they are near the goal (and the player cannot carry the ball into the goal). Thus, this strawman, when applying the advice, will never shoot in situations where it is appropriate to do so. This result demonstrates the need to refine a teacher's advice when that advice is not precisely correct.

7.3 Summary

My experiments with the Pengo and Soccer testbeds show that my system, RATLE, which allows a reinforcement-learning agent to refine procedural domain theories. An agent in RATLE, when receiving good advice, achieves performance that is superior to the performance that an agent achieves without the advice. My experiments also demonstrate that an agent that is able to refine the advice it receives is able to outperform an agent that receives the same advice but does not refine it. Thus, my experiments support the central hypothesis of this thesis.

I also investigated other aspects of the RATLE system. Analyses of the effects of advice show that advice seems to produce effects that are consistent with the focus of the advice (e.g., advice designed to eliminate enemies produces agents that eliminate more enemies). In the Pengo testbed, I performed experiments that indicate that the gains in performance due to advice occur no matter when the agent receives advice. I also show that an agent is able to profitably make use of a second piece of advice, which demonstrates that the interaction of the teacher and agent in RATLE need not stop after a single piece of advice. I also demonstrated that an agent supplied with "bad" advice may see an initial drop in performance, but that with learning the agent can overcome the effects of "bad" advice. Experiments with the technique *replay*, a method for reusing the experience that the agent obtains during exploration, indicate that a straightforward approach to this technique does not produce the gains in performance reported by others in their testbeds (Lin, 1992; Thrun, 1994).

My tests with the Soccer task support my results from the Pengo task. The Soccer task is interesting because it is different from the Pengo task in a number of ways: the agents in Soccer have complete world information, where the Pengo agent has only limited line-of-sight information; the agents in Soccer work cooperatively against a set of competitive agents, while the Pengo agent works alone against a set of competitive agents; and the set of reinforcements in the Soccer environment (goals scored) are much more sparse than those in the Pengo environment. Despite these differences, the results for the Soccer task are similar to those from the Pengo task; the agents receiving advice in the Soccer task also show gains in performance over agents that do not receive instruction. These results suggest that the RATLE system is applicable to a wide range of reinforcement-learning tasks.

Chapter 8

Additional Related Work

My work relates to research in a number of areas, some of which I presented in Chapter 2. In this chapter, I describe other research that is closely related to my own. As I present other research, I will discuss their differences with respect to RATLE, since it encompasses much of FSKBANN (related work for the protein-folding problem was discussed in Chapter 3).

One related research area involves methods for giving advice to a problem solver – techniques that use knowledge provided by a teacher. I first discuss these systems, and then discuss ones specifically designed to give advice to a reinforcement learner.

Research on refining domain theories relates both to advice-giving research in general and to my work in particular. These techniques take advice in the form of an initial theory about a task and refine it using samples of the task. An especially relevant theory-refinement approach is to create knowledge-based neural networks, since my work augments methods from this field. I first present a number of alternatives for incorporating advice into a neural network and discuss how these approaches relate to mine. I then give an overview of some work using inductive-logic programming (Muggleton, 1992) to refine a domain theory.

Next, I present research on inducing finite-state automata. Early work in this field focused on theoretical limits for learning algorithms and methods using constructs such as oracles (i.e., a device that can always produce an answer to a query if an answer exists). More recently, techniques such as recurrent neural networks and knowledge-based neural networks have been applied to this problem, and I describe the relation of these approaches to mine.

Finally, I discuss some work on developing programming languages for interacting with agents, and discuss how RATLE differs from these efforts.

8.1 Providing Advice to a Problem Solver

As I previously noted, the concept of giving advice has a long history in AI. Many advice-taking systems focus on the problems of understanding teacher instructions, including how to operationalize them into a form that the computer learner can use. Generally these approaches focus on understanding rather than refining the advice provided by the teacher.

One early example of an algorithm that makes use of advice for planning is ABSTRIPS (Sacerdoti, 1974). In ABSTRIPS, a human assigns initial "criticalities" to preconditions to cause the planner to focus on making key planning steps. In both ABSTRIPS and RATLE a human is in some way directing the search; however, while the ABSTRIPS mechanism requires that the teacher have some understanding of the search process, RATLE does not need the teacher to understand how the agent works internally. Also, RATLE agents are able to learn.

In FOO (Mostow, 1982), general advice is *operationalized* by reformulating the advice into search heuristics. FOO applies these search heuristics during problem solving. In FOO, Mostow assumes that the advice is correct, and the learner's task is to convert the general advice into an executable plan based on its knowledge about the domain. RATLE is different from FOO in that RATLE tries to directly incorporate general advice, but does not provide a sophisticated means of operationalizing advice – RATLE only operationalizes fuzzy terms like "big" and "near." Also, RATLE does not assume that the advice is correct; instead, it uses reinforcement learning to refine and evaluate the advice.

More recently, Laird et al. (1990) created an advice-taking technique called ROBO-SOAR that is based on the SOAR architecture (Laird et al., 1987). SOAR uses its knowledge to select operators to achieve goals. When it is unable to select from a set of operators, or no operator is applicable, an *impasse* arises. When this happens, SOAR creates a subgoal and applies itself to that subgoal. In ROBO-SOAR, an observer 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, ROBO-SOAR uses advice to guide the learner's reasoning process, while RATLE directly incorporates the advice into the learner's knowledge and then refines that knowledge with subsequent experience.

Huffman and Laird (1993) developed another SOAR-based method called INSTRUCTO-SOAR 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 task, and uses SOAR's form of explanation-based learning (Mitchell et al., 1986) to generalize the instruction into a rule that it uses in similar situations. RATLE differs from INSTRUCTO-SOAR in that it provides a language for providing general advice rather than attempting to generalize specific advice.

8.2 Providing Advice to a Problem Solver that Uses Reinforcement Learning

Recently, several researchers have defined methods for providing advice to computer learners and, in particular, reinforcement-learning agents. A number of these approaches are closely related to mine.

Lin

Lin (1991, 1992, 1993) uses RL on a simulation task that is similar to my Pengo task. Both of our agents employ connectionist Q-learning and explore a robot world using a set of sensors providing limited information. Lin (1992) designed the technique "replay" that uses advice expressed as sequences of teacher's actions. In his approach, the agent periodically trains on the sequence of teacher-provided actions. Thus, the resulting agent should be biased towards the actions chosen by the teacher. One difference between my approach and Lin's is that RATLE accepts advice in a general form. Also, RATLE directly installs advice into the RL agent; in Lin's system, the agent must spend time training its policy function on the teacher's actions to try to determine the conditions that lead to these actions.

Lin (1993) has also investigated the idea of having a learner use prior state knowledge. He uses an RL agent whose input contains not only the current input description, but also some number of the previous input descriptions. The difference between Lin's approach and mine is that his agents retain entire input descriptions, while in RATLE the teacher gives advice that suggests which information to retain. The advice thus limits the amount of information the agent has to process and greatly focuses the learning process.

Clouse and Utgoff

Utgoff and Clouse (1991) developed a learner that consults a set of teacher actions when the action the learner chose resulted in significant error. Their system has the advantage that the agent 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 she feels she has something to say.

Clouse and Utgoff (1992) created a second technique that takes advice in the form of actions suggested by the teacher. Whenever the teacher feels it is appropriate, she can suggest an action that the agent should take. Their approach is therefore similar to RATLE in that the teacher determines when to provide advice. The main difference is that in Clouse and Utgoff's approach the advice is specific to a single situation (i.e., the action the teacher suggests for a state), whereas the advice in RATLE is broadly applicable. In Clouse and Utgoff's approach, the agent must employ empirical learning to generalize the advice given.

Whitehead

Whitehead (1991) examined an approach similar to both Lin's and Utgoff & Clouse's. In his approach the agent can learn by receiving advice in the form of *critiques*, which are rewards indicating whether the agent's current action is optimal or not. The agent can also learn by observing the actions of the teacher. As with the above approaches, RATLE differs from Whitehead's approach in that RATLE is able to accept general advice about actions to take in multiple states.

Gordon and Subramanian

Gordon and Subramanian (1994) developed a system that is closely related to mine. Their system employs genetic algorithms (Holland, 1975), an alternate approach to 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. Their agent then uses genetic search to explore its environment and alter its initial background knowledge.

My work primarily differs from Gordon and Subramanian's in that RATLE uses connectionist Q-learning instead of genetic algorithms. Also, my advice language focuses on actions to take rather than goals to achieve. Finally, RATLE allows advice to be given at any time during the training process. One drawback, however, is that RATLE does not have the operationalization capability of Gordon and Subramanian's approach.

Thrun and Mitchell

Thrun and Mitchell (1993) investigated a method that allows RL agents to make use of prior knowledge in the form of neural networks. In their approach they assume the existence of neural networks that have been previously trained on similar tasks. They focus on using the knowledge already embedded in these networks to give their learner an initial theory for the new task. This proves to be effective, but requires previously-trained neural networks that are related to the task being addressed. RATLE takes advice in a more general form and does not require previous training.

8.3 Refining Prior Domain Theories

There has been a growing literature on automated theory refinement (Cohen, 1994; Fu, 1989; Ginsberg, 1988; Ourston & Mooney, 1994; Pazzani & Kibler, 1992; Towell & Shavlik, 1994). Theory refinement is closely related to the idea of advice taking, since the domain theory provided to the learner can be thought of as advice to the learner. Theory refinement differs from most advice-taking approaches in that theory-refinement systems generally focus on the process of refining the advice, rather than on understanding advice.

My work differs from standard theory-refinement approaches by its novel emphasis on refinement of procedural domain theories in multi-actor worlds, as opposed to refinement of theories for categorization and diagnosis. Finally, unlike previous approaches, I allow domain theories to be provided at any time during the training process, as the need becomes apparent to the teacher. In complex tasks where learning is slow, it is not desirable to simply restart learning from the beginning whenever one wants to add something to the domain theory.

8.3.1 Incorporating Advice into Neural Networks

One popular method for refining domain theories is to refine them with a neural network. KBANN, the system RATLE and FSKBANN are built on, is an example of a system for refining domain theories with neural networks. I presented details on KBANN in Chapter 2. Below I discuss a number of other techniques for using advice in neural networks and how these techniques relate to my work.

One approach closely related to my own is Siegelman's (1994) technique for converting programs expressed in a general-purpose, high-level language into a type of recurrent neural network. Her system is especially interesting in that it provides a mechanism for performing arithmetic calculations. She also provides mechanisms for looping constructs and other statements that are very similar to the constructs of RATLE. Siegelman's approach differs from mine in that she uses network units with piecewise linear activation functions. Also, her work has not been empirically demonstrated, due to her reliance on these non-standard network units.

Gruau (1994) developed a compiler that translates Pascal programs into neural networks. While his approach has so far only been tested on simple programs, it may prove applicable to the task of programming agents. Gruau's approach includes two methods for refining the networks he produces: (1) a genetic algorithm, and (2) a hill-climber. The main difference between Gruau's technique and RATLE is that the networks RATLE produces can be refined using standard connectionist techniques such as backpropagation. Also, 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.

Noelle and Cottrell (1994) suggest a novel approach for making use of advice in neural networks. In their approach, the connectionist model itself performs the process of incorporating advice, where advice is expressed as inputs to the network, and the "knowledge" of the network is maintained in a set of memory units employing recurrent links. This contrasts with RATLE's approach of directly adding new "knowledge-based" units to the neural network. RATLE leads to faster assimilation of advice, though Noelle and Cottrell's approach is arguably a better psychological model.

Diederich (1989) devised a method that accepts instructions in a symbolic form. He uses instructions to create examples and then incorporates the instructions by training a neural network with these examples. RATLE differs from Diederich's approach in that it directly installs instructions into the neural network.

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 (e.g., 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, my work differs from Abu-Mostafa's in that I directly install 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 to construct a simple function for the 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 my approach incorporates advice immediately.

8.3.2 Refining Domain Theories via Inductive Logic Programming

In inductive-logic programming (ILP) (Muggleton & Feng, 1990; Quinlan, 1990), the learner induces a set of Horn clauses to represent a desired relation. It receives as input a set of facts, in the form of predicates (e.g., `sibling(tom,jane)` and `male(tom)`), defining a set of samples of the relation to be induced. For example, an ILP system could induce a set of clauses to define the relationship "uncle" from background facts extensionally defining useful predicates for this relationship (e.g., parent, sibling, male, etc.). ILP is related to RATLE in that ILP addresses problems that may require procedural solutions, such as ones that require recursive solutions like the "ancestor" relationship. My work mostly differs from ILP systems in that I focus on reinforcement learning rather than learning logical relationships, and in that I use connectionism as my induction method.

Recent work in ILP (Cohen, 1994; Pazzani & Kibler, 1992; Richards & Mooney, 1995) has produced methods that can refine an initial set of clauses with respect to a set of examples defined intensionally by background predicates. These systems introduce operators that can make changes to the current theory. They work by selecting operators that refine the current domain theory in order to better fit the background predicates. They are also able to make use of other advice such as indications of the types of the arguments for a clause. As with other ILP approaches, my work differs in that RATLE applies to RL tasks, rather than Horn-clause learning. Also, RATLE is able to assimilate advice provided at any time during the learning process.

8.4 Inducing Finite-State Information

My work with FSKBANN and RATLE relates to the problem of learning an finite-state automaton (FSA) from examples. Early theoretical work on this problem looked at the problem of learning an FSA from a set of example sequences. Gold (1972) designed an algorithm that makes a hypothesis about a set of states and then suggests a test string to further define the automaton. An oracle identifies this test string as an accepted or rejected string, and Gold's algorithm then adjusts its hypothesis. Gold (1978) later showed that the problem of inferring the *minimum* (in terms of number of states) automaton from a set of data is an *NP*-complete problem. Angluin (1987) designed an alternate learning method involving hypotheses and counterexamples. Her system makes a hypothesis about the correct automaton and then asks an oracle to suggest a counterexample. The system uses the counterexample to fix the original automaton.

Rivest and Schapire (1987) investigated an alternate method. Their technique is based on the idea that there are a limited number of sets of actions resulting in new states, where states are defined as combinations of environmental sensor values. The structure they produce, an *update graph*, specifies the compositionality of series of actions. Rivest and Schapire developed a method for learning an update graph by exploration. An FSA can be exhaustively built from the update graph, starting with the start state and applying each possible operator to determine the set of possible result states.

More recently, researchers have proposed neural-network architectures for incorporating information about state. Jordan (1989) and Elman (1990, 1991) introduced the Simple Recurrent neural Network (SRN) architecture that I presented in Chapter 2. The idea of retaining a state or context across training patterns occurs primarily in work addressing natural language problems (Cleeremans et al., 1989; Elman, 1990). The idea of using the type of network introduced by Jordan to represent a finite-state automaton was first discussed by Cleeremans et al. (1989). They show that this type of network can perfectly learn to recognize a grammar derived from an FSA. The major difference between my research and Cleeremans et al. is that we focus on using an initial domain theory expressed as a finite-state automaton, rather than attempting to learn it solely from training examples.

Omlin and Giles

Giles et al. (1992) demonstrated the idea of learning an FSA with a second-order recurrent neural network (see Figure 48). A second-order recurrent neural network's input vector consists of the current input character to the FSA and the current state (mapped with recurrent links). The next layer of a second-order recurrent neural network is a fixed set of hidden units (the "second-order" units) that calculate the product of each of the input character units with each of the state units (i.e., there are *#inputs x #states* of these units). These units correspond to the set of all possible transitions in an FSA. Finally, there are a set of links from these second-order units to the set of output units representing states. The learning algorithm must learn the links from the second-order units to the states in order to

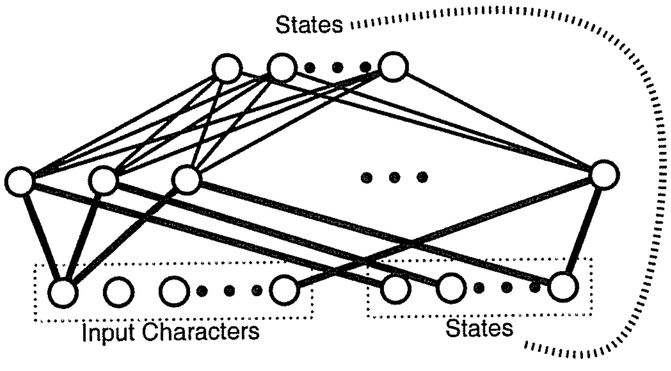


Figure 48: A second-order recurrent neural network. This type of network has a set of hidden units ("second-order" units) that calculate the multiplicative combination of each of the input values with each of the states. These second-order units are fixed; the only links that may be altered in a second-order network are the links from the second-order units to the outputs (states).

produce a network that predicts whether a string is accepted or rejected by the FSA. Giles et al. (1992) demonstrated that such an architecture can be used to both learn an FSA and to extract the learned FSA. As with the Cleeremans et al. work, RATLE differs from the Giles et al. work in that I focus on inserting prior knowledge about the finite-state task.

Omlin and Giles (1992) present an approach that closely relates to FSKBANN. In their work they employ a second-order recurrent neural network to learn FSAs. They also make use of "hints" in the form of instructions that indicate some of the transitions in the FSA. To map these hints, Omlin and Giles simply make the link corresponding to the hint a strong link. For example, a hint stating that there is a transition from state A to B on character x would cause a strong connection from the second-order unit representing conjunction of state A and input x to state B. My work differs from Omlin and Giles' work in several ways. In FSKBANN, I can translate complex transitions that are dependent on combinations of input features, while Omlin and Giles can only map conditions dependent on a single input feature. FSKBANN can map a domain theory containing more than one FSA. Finally, the general instructions of RATLE allow the teacher to provide information about multiple state transitions at once.

Frasconi, Gori, Maggini and Soda

Frasconi et al. (1995) have also examined an an approach for learning a finite-state automaton where some information about the FSA is known. In their approach, they use Local

Feedback Multi-Layered Networks (Frasconi et al., 1992). This type of network has recurrent connections within the hidden layer of the network. They show that this type of network will achieve a stable activation state for each input value in a finite number of steps. To encode the set of transitions in an initial FSA, they define a set of constraints on the weights and biases of the network based on the initial transitions and use linear programming to select a feasible point in weight space that meets those constraints. My work differs from theirs in that I insert the knowledge about the finite-state process into the network directly, without having to use linear programming to select the weights. Also, I allow the teacher to specify general information about multiple transitions.

8.5 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). In general, these languages focus on turning a set of instructions, often in the form of a programming language, into a form that is easy for the agent to execute. My work differs from these approaches in that I focus on how to give instruction to an agent that is able to learn, thus the instructions need not be completely correct.

Kaelbling and Rosenschein (Kaelbling, 1987; Kaelbling & Rosenschein, 1990) built the language REX that produces a reactive plan for a robot agent. The REX language, like RATLE's, is based on a programming language. REX contains constructs to specify the effects of acts that the agent may take and goals the agent may want to achieve. Kaelbling and Rosenschein (1990) built a compiler that could turn a set of constructs in REX into a circuit that could efficiently determine the appropriate next action given the current world state. Thus, REX and RATLE are related in that they both produce reactive agents. The main difference between them is that in RATLE the agent can refine its instructions, while REX must receive correct instructions.

Crangle and Suppes (1994) investigated how a robot can understand a human's instructions that is expressed in ordinary English. Their work focuses on mapping certain types of English statements, such as simple imperative sentences, into corresponding robot commands. However, unlike RATLE they do not address corrections, by the learner, of approximately correct instruction.

8.6 Summary

The work in this thesis relates to research in a number of areas, which I presented in this chapter. In the first section, I described approaches to giving advice to a problem solver. My work primarily differs from these approaches in that they focus on understanding the teacher's instructions, while my techniques try to refine the instructions. Thus my work is able to deal with advice that is not completely correct. However, my work is less able to perform the elaborate operationalization that allows these approaches to understand vague advice.

More recently, a number of researchers have investigated methods for supplying hints to a reinforcement-learning agent, techniques which closely relate to my work. My work differs from most of these methods in that I allow the teacher to provide general advice, which may refer to an action or actions to take in many situations, while the teacher in other approaches provides a recommendation about a single action to take in a state. These techniques require the agent to use empirical learning in order to generalize the teacher's recommendation.

Theory refinement is a second major area of related research. My work differs from typical theory-refinement approaches in that I focus on procedural domain theories. Also, my work addresses reinforcement-learning tasks in multi-agent environments, while most theory-refinement techniques have been designed for classification tasks. A major difference with my work and other theory-refinement approaches is that my work incorporates advice provided at any time by the teacher, rather than only at the beginning of training.

One closely related area of theory refinement is knowledge-based neural networks. My work augments the knowledge-based neural network technique KBANN (Towell et al., 1990), to translate a much broader instruction language. My work differs from some techniques for producing knowledge-based networks in that I produce networks that can be refined with standard learning algorithms, rather than requiring new learning methods. My work differs from other techniques because I directly incorporate advice into the network, rather than using neural-network training to incorporate advice.

Inductive logic programming has been applied to theory refinement. Such approaches differ from mine in that they focus on learning logical relationships, rather than learning utility functions for RL agents.

A third major area of related research are techniques for learning finite-state automata. Several researchers have developed recurrent network mechanisms that can induce a finite-state automaton. My work differs from these mainly in that I am able to provide *general* advice to the network about the finite-state process.

Finally, my work differs from most languages for instructing robots in that I refine the instructions given to the agent (robot), where other techniques assume that the provided instructions are correct.

Chapter 9

Conclusions

In this thesis, I define and evaluate two systems, FSKBANN and RATLE, that expand the language a teacher can use to instruct a computer learner. Both FSKBANN and RATLE are techniques for refining domain theories with training examples. These systems are novel because they accept *procedural* domain theories from their teacher; RATLE also allows the teacher to continuously give advice to the learner. To evaluate FSKBANN and RATLE, I performed several experiments. The central question of these experiments is: does a technique for refining procedural domain theories demonstrate the same type of appealing benefits as do techniques for refining non-procedural domain theories? That is, does the learner with instruction learn faster than a learner without instruction, and is the learner able to produce a refined theory that is more accurate than the original domain theory? My experiments with my two systems indicate that the answer to these questions are yes.

In the next section, I outline the specific contributions made by this thesis. Following that I will discuss some limitations and future directions for my work.

9.1 Contributions of this Thesis

The contributions of this thesis are as follows:

- My FSKBANN system allows a teacher to provide domain theories in the form of propositional rules augmented with finite-state automata (FSAs). The teacher uses an FSA to capture the contextual aspects of a procedural task. For example, the teacher can specify a rule for predicting the current secondary-structure of an amino acid that is based on the prediction the learner made for the previous amino acid. The states of the FSA act as the learner's "memory" for information from previous steps of the task.

  FSKBANN inserts the knowledge from the propositional rules and FSAs into a knowledge-based neural network. My approach is based on the KBANN technique (Towell et al., 1990), which I extend to represent the states of the FSAs. The states of the teacher's

FSAs are represented in FSKBANN using a Simple Recurrent neural Network (Elman, 1990; Jordan, 1989). Each state corresponds to a unit with a recurrent link that remembers the activation of the state from the previous step of the network. FSKBANN translates the transitions from the FSAs to rules that determine whether the units representing states are active or inactive.

- Tests of FSKBANN on the Chou-Fasman (1978) algorithm, a method for predicting the secondary structure of globular proteins, show that FSKBANN is able to successfully refine a procedural domain theory. My tests also show that the refined theory is more accurate than FSKBANN's underlying inductive learning method, neural networks, would be without the domain theory.

  My in-depth analysis of these results indicates that the refined domain theory does a good job of predicting all three secondary structure classes, whereas the neural network approach achieves good performance only by focusing on predicting the largest secondary-structure class (coil, the default class). This is important, since biologists prefer an approach that focuses on predicting the other, more meaningful, classes of secondary structure (α-helices and β-sheets).

- I also define the RATLE advice language, which allows a teacher to communicate instructions to a reinforcement-learning agent. Instructions in this language take the form of simple programming-language constructs. These constructs indicate actions the agent should take under certain conditions in the world. The teacher defines these conditions using logical combinations of input features and teacher-defined intermediate terms. The resulting language provides a simple, yet powerful, mechanism for instruction. Novel features of the RATLE advice language include:

  - The advice language allows the teacher to describe general states of the world, thus the teacher is able to give instruction about the appropriate action to take in multiple situations. This is an improvement, since most other techniques for advising a reinforcement learner only accept recommendations about the action to take in a specific situation, and require the learner to generalize the recommendation.
  - RATLE has statements that the teacher can use to define loops (i.e., REPEAT-UNTIL) the agent should perform. RATLE represents these loops using state units in a way that is completely transparent to the teacher.

  - A teacher can give the agent instructions indicating multiple-step plans the agent should take. These multiple-step plans are composed of sequences of single actions, and, as with loops, the plans are implemented with state units.
  - The teacher can also indicate actions that the agent should NOT take under certain circumstances.
  - RATLE includes conditions that allow the teacher to use fuzzy terms like "small" and "light." The set of fuzzy terms available to the teacher is defined by a separate person that I refer to as the initializer. The initializer uses RATLE's fuzzy-term language to create fuzzy terms for a task. RATLE automatically maps any fuzzy terms the teacher specifies to the definition supplied by the initializer.
  - Advice in RATLE can be supplied continuously. RATLE translates language statements into *additions* to the agent's neural network. RATLE maps as many instructions as the teacher is willing to supply. The teacher can thus select advice that addresses current problems with the agent's behavior.

- I tested RATLE on two testbeds: Pengo and Soccer. In Pengo, an agent explores a maze-line environment, trying to eat food, while avoiding or eliminating enemies. The agents in Soccer are part of a team attempting to score a goal while preventing the other team from scoring. In the Soccer testbed the agent receives complete information about the environment, while in Pengo the agent only receives information about objects in line-of-sight around it. In both testbeds, the agents work against a group of competitive agents, but in the Soccer testbed there are also cooperative agents (i.e., players on the same team).

  Tests on these two testbeds indicate that a RATLE agent receiving advice outperforms an agent that does not. My experiments also show that it is important for RATLE's agent to be able to refine the instructions, since when the agent refines instructions, it outperforms a version that does not refine the teacher's statements. An analysis of the performance of the agents after refining advice shows that the agent produces results that are consistent with intent of the statement (e.g., a statement aimed at helping the agent collect food does indeed lead the agent to collect more food).

  Other experiments I performed indicate that the agent achieves similar results no matter when the teacher provides instruction. I also tested RATLE by providing an agent with a second piece of advice to show that agents are able to profitably make use of subsequent instruction. I further showed that a RATLE agent provided with "bad" advice may experience an initial loss of performance, but that with learning the agent can overcome the effects of the "bad" advice.

9.2 Limitations and Future Directions

Based on my experiences with RATLE and FSKBANN, I have identified a number of limitations and possible future directions for my research, some of which I presented previously in Sections 3.5, 5.5, and 6.4. Below I discuss other possible future research topics.

One important future topic is to evaluate my approach in other domains. In particular, I intend to explore giving advice to agents in situations where there are other competing agents that are able to learn. Preliminary tests I have performed in such situations show that agents receiving advice demonstrate initial gains in performance, but that the opposing agents adjust their performance to counter the teacher's instructions (Shavlik & Maclin, 1995). This suggests that the ability of the teacher to give recommendations continuously is crucial in this type of situation. Such a domain would also be interesting in that the teacher could provide instructions about how learners on the same team can cooperate.

Another domain of interest is software agents (Dent et al., 1992; Maes & Kozierok, 1993; Riecken, 1994). For example, a human could advise a software agent that looks for "interesting" papers on the World-Wide Web.

I also see a number of useful algorithmic extensions to RATLE. These extensions fit into three broad classes:

1. Broadening what I allow the teacher to "say" to the learner.

2. Improving the technical details involving the mapping of advice into a neural network and then refining it with connectionist Q-learning.

3. Designing algorithms that convert refined advice back into human-readable terms, thereby allowing human inspection of the refined advice and also allowing communication of the learned knowledge to other machine learners.

9.2.1 Broadening the Advice Language

My experience with RATLE has led me to consider a number of extensions to the current advice language, many of which I discussed in Section 5.5. Another form of information the teacher could supply are statements that indicate the relative value of two actions in a given situation (e.g., one action is *preferred* under some conditions). The teacher could also provide indications of the utility of actions, perhaps using fuzzy terms such as "good" or "poor." I also plan to explore mechanisms for specifying multi-user plans when I further explore domains with multiple agents.

RATLE could also be extended to work similarly to Sutton's (1991) DYNA system. The agent in RATLE could simultaneously learn both a utility function and a model of the world. The agent would then use the world model to perform simulated actions in order to obtain more training experiences, which can be important in domains where examples are very difficult or costly to obtain. The teacher could give advice about the effects of the agent's actions (i.e., the world model) in addition to advice about actions to take. This information would be used to improve the agent's world model.

Another method the teacher could use to help the learner would be to select configurations of the environment that would be particularly useful for the agent to experience, given the skill level of the learner. For example, a teacher selecting environments for a novice driver might choose to bring the driver to a large, open parking lot so that the novice could learn to handle the car without other considerations (other drivers, traffic signals, etc.). With a more advanced student such as an expert airline pilot, a teacher might select environments that the student might not ordinarily experience (e.g., taking the pilot to a flight simulator to experience emergency situations like the loss of power to an engine). In selecting the environments the agent experiences, the teacher is indirectly selecting features of the environment for the agent to focus on, and counts on the agent's inductive learning mechanism to determine those features. This type of approach would be complementary to my current work, and investigation of the potential synergy is a promising topic for future research.

9.2.2 Improving the Algorithmic Details

There are four issues involving the mapping or refining of advice that I have not previously discussed. Firstly, in my approach to connectionist Q-learning, I use a neural network as the agent's utility function, where the input vector to the network includes the state of the world, as well as one output for each of the agent's possible actions. Another possible approach would be to use a network where the input vector contains features describing a possible action in addition to the description of the current state of the world. The output of the network would then be a single unit indicating the utility of the represented input action. While this approach would make the learning task more difficult in some ways, it might be useful for tasks where actions can be parameterized. For example, a robot agent might be able to rotate itself by differing degrees. This type of action is difficult to represent as a set of discrete actions. With a representation where the action is an input to the network, the angle of rotation could be an input feature to the network. The agent would then select an action by sampling the space of possible rotations to find an appropriate rotation. The agent could also analyze the derivatives of the the weights to the features representing actions with respect to the predicted utility in order to find an appropriate action.

Mapping the RATLE language to this form of network would be straightforward. To translate a recommended action, RATLE would create a unit that is a conjunction of the unit representing the condition leading to the action and the input feature corresponding to the action, similar to Omlin and Giles' (1992) approach. It would then connect this unit to the output unit with a strong link.

Secondly, in another approach to reinforcement learning, the agent predicts the utility of a state rather than the utility of an action in a state (Sutton, 1988); in this approach the learner has a model of how its actions change the world. In this approach the agent 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 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.

Thirdly, further tests with the replay technique (Lin, 1992) are important, since this technique allows the agent to re-use the experiences it accumulates. This ability is crucial for domains where the cost of exploration is prohibitive.

Finally, RATLE currently maintains no statistics that record how often a piece of advice was applicable and how often it was followed. I intend to add such statistics-gatherers and use them to inform the teacher that a given piece of advice was seldom applicable or followed. I 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 the teacher that some instruction has substantially changed (I plan to use the rule-extraction techniques described below when I present the refined statement to the teacher).

9.2.3 Converting Refined Advice into Human Comprehensible Terms

One interesting area of future research is the "extraction" (i.e., conversion to a easily comprehensible form) of learned knowledge from my connectionist utility function. In order to extend previous rule-extraction techniques (Craven & Shavlik, 1994; Towell & Shavlik, 1993) to advice-taking tasks, I plan to examine two tasks: extending Towell and Shavlik's (1993) language for extracted rules, and employing rule extraction to transfer learned knowledge between agents operating in the same or similar domains.

Previous rule-extraction work has been applied to domains with discrete-valued input features and has used a language of propositional inference rules to express extracted concept representations. I plan to extend these techniques so that they can handle real-valued input features, and so that they can express extracted concept representations using the same language that is used to give advice to the learners. The RATLE advice language incorporates aspects, such as multi-step operators and fuzzy conditions, for which current rule-extraction techniques provide no support. I plan to extend these rule-extraction techniques to extract concept descriptions expressed in this richer language.

I also plan to investigate the use of rule-extraction methods for knowledge transfer among machine learners. For example, one learner might ask another what it would do given a particular sensor reading. Transferring acquired knowledge from one learning system to another can speed up learning, which is particularly important in domains where training is slow or where training examples are sparse or expensive. My approach to knowledge transfer would use rule-extraction methods to elicit knowledge from trained networks, and my advice-giving method to insert transferred knowledge into other networks. That is, my advice-taking language would serve as the interlingua for communication between machine learners.

9.3 Final Summary

In this thesis I present the FSKBANN and RATLE systems. These techniques allow a teacher to give advice to a computer learner solving a procedural task. The learner can then refine the teacher's instructions. Experiments presented in Chapters 3 and 7 demonstrate that the refined domain theories produced by FSKBANN and RATLE outperform both the original domain theories and the underlying inductive learning mechanism, neural networks, when that underlying mechanism does not receive the teacher's instructions.

In Chapter 3 I outlined my initial approach, FSKBANN, which allows a teacher to give advice to a computer learner in the form of propositional rules and FSAs. FSKBANN maps the knowledge from the rules and FSAs into a neural network that represents the states from the FSAs with recurrent links. Tests with FSKBANN on the Chou-Fasman (1978) algorithm produce a refined theory that outperforms the non-learning Chou-Fasman algorithm as well as a standard neural-network approach.

I defined RATLE in Chapters 4, 5, and 6. RATLE allows a teacher to communicate advice to a connectionist, reinforcement learning agent in the form of statements in a simple programming language. The advice language allows the teacher to indicate conditions of the environment and actions the agent should take under those conditions. Novel features of the language include the following: (1) a statement may indicate *loops* the agent should execute, (2) it may suggest *multi-step* plans for the agent to follow, (3) it may recommend actions the agent should *not* take, (4) it can include references to *fuzzy* terms such as "near" and "short", and (5) advice may be given *continuously.* RATLE translates the teacher's statements into additions to the RL agents neural-network policy function, using recurrent units to represent loops and multi-step plans.

In Chapter 7, I presented experiments applying RATLE to two simulated testbeds involving multiple agents: Pengo and Soccer. These tests demonstrated that a RATLE agent receiving advice outperforms both an agent that does not receive advice and an agent that receives the instruction but is unable to refine it. Analysis of these results indicate that the effects of the instruction conform to my expectations (e.g., recommendations about avoiding enemies helps the agent survive longer). Other experiments show that the effect of instruction seems to be independent of when the teacher supplies the instruction to the agent, that the agents are able to profitably make use of a subsequent piece of advice, and that the agents can overcome the effects of "bad" advice with learning.

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

Appendix A

Details of the Pengo Testbed

In this appendix I present further details about the Pengo testbed investigated in Chapter 7. I start by describing the processes used to create Pengo environments. I then define input features, fuzzy terms, and advice used in Chapter 7's experiments.

A.1 Creating Initial Pengo Environments

In order to perform experiments in the Pengo environment, I randomly construct a series of initial Pengo environments. To construct an initial environment, I start with an empty 7x7 grid and then to deposit a series of randomly generated "rooms" on the grid. Each "room" is a hollow rectangle whose border is composed of obstacle objects, and where the length and width of the rectangle are random numbers between 1 and 7 (inclusive). I randomly select coordinates for one of the corner points and place the room on the grid (any portion of the room that is off the grid is then discarded). I continue this process until the grid is at least 35% full of obstacles.

I then use a standard algorithm for connected components (Sedgewick, 1983, pp. 384–385) to insure that all of the open spaces in the grid are reachable from any other. To do this, I construct the connected components of the grid, and then remove obstacles in a line between the two closest components until they are connected. I repeat this process until all of the components are connected. The resulting grid generally has approximately 15 obstacles.

After setting up an initial set of obstacles, I choose a location for the agent randomly among the remaining open spots of the grid. I then choose initial locations for the foods from among the remaining open spots on the grid. Finally, I choose initial locations for the enemies from the remaining open spots, with the caveat that these locations must be at least two grid units distant from the agent's initial location.

A.2 Input Features

As I discussed in Chapter 7, the sensors of the agent in the Pengo consist of a series of values indicating how many of each kind of object exist in a set of sectors around the agent. The agent measures six values for each sector: the number of enemies, foods, walls, and obstacles, as well as how much of each sector is empty and how much is occluded. I define each sector by four values – a minimum and maximum angle and a minimum and maximum distance from the agent. There are eight pairs of minimum and maximum angles around the agent, with the 360° range of values evenly divided. There are four sets of minimum and maximum distances: {0,1}, {1,3}, {3,6}, and {6,10}. Thus, there are 192 sensor input features. To this I add nine Boolean input features representing, in an 1-of-N encoding, the previous action taken by the agent. The final total is 201 input features.

Sensor inputs are REALs of the form:

```
REAL Enemy Angle IN [ 0.393 .. 1.178 ] Distance IN [ 3 .. 6 ] [ 0 .. 10.6 , 0 .. 1 ]
```

This example describes an input unit that represents up to 10.6 `Enemies` (parts of an object may be in multiple sectors), where the value is normalized to be between 0 and 1. These `Enemies` are from three to six units distant from the agent, and the angle to these `Enemies` is between 0.393 ($\frac{\pi}{8}$) to 1.178 ($\frac{3\pi}{8}$).

A.3 Fuzzy Terms

The fuzzy terms set up for the Pengo environment are as follows:

```
QUANTIFIER No   ::= SUM < 1
QUANTIFIER A    ::= SUM = 1
QUANTIFIER An   ::= SUM = 1
QUANTIFIER Many ::= SUM > 2

PROPERTY NextTo ::= Distance IN [ 0   .. 1 ]
PROPERTY Near   ::= Distance IN [ 0   .. 2.5 ]
PROPERTY Medium ::= Distance IN [ 2.5 .. 5 ]
PROPERTY Far    ::= Distance IN [ 5   .. 10 ]

PROPERTY East   ::= Angle IN [ -0.785 .. 0.785 ]
PROPERTY West   ::= Angle IN [  0.785 .. 2.356 ]
PROPERTY North  ::= Angle IN [  2.356 .. 3.927 ]
PROPERTY South  ::= Angle IN [  3.927 .. 5.498 ]
```

Note that the thresholds for these functions can be changed by the agent in its network through training.

A.4 Advice Provided

The statements I use to create the six pieces of advice used in the Pengo experiments in Chapter 7 appear below. Figures 37, 38, 39, and 40 pictorially depict the first four pieces of advice, respectively.

Good Advice

SimpleMoves:

```
FOREACH dir IN { East, North, West, South }
  IF An Obstacle IS (NextTo AND $(dir)) THEN
    INFER OkPush$(dir)
  END;
  IF No Obstacle IS (NextTo AND $(dir)) AND
    No Wall IS (NextTo AND $(dir)) THEN
    INFER OkMove$(dir)
  END;
  IF An Enemy IS (Near AND $(dir)) THEN
    DO_NOT Move$(dir)
  END;
  IF OkMove$(dir) AND
    A Food IS (Near AND $(dir)) AND
    No Enemy IS (Near AND $(dir)) THEN
    Move$(dir)
  END;
  IF OkPush$(dir) AND
    An Enemy IS (Near AND $(dir)) THEN
    Push$(dir)
  END
ENDFOREACH
```

NonLocalMoves:

```
FOREACH dir IN { East, North, West, South }
  IF No Obstacle IS (NextTo AND $(dir)) AND
    No Wall IS (NextTo AND $(dir)) THEN
    INFER OkMove$(dir)
    END;
    IF Many Enemy ARE (NOT $(dir)) AND
      No Enemy IS (Near AND $(dir)) AND
      OkMove$(dir) THEN
      Move$(dir)
    END;
    IF OkMove$(dir) AND
      An Enemy IS ($(dir) AND {Medium OR Far}) AND
      No Enemy IS ($(dir) AND Near) AND
      A Food IS ($(dir) AND Near) THEN
      Move$(dir)
    END
  ENDFOREACH
```

ElimEnemies:

```
  FOREACH dir IN { East, North, West, South }
    IF No Obstacle IS (NextTo AND $(dir)) AND
      No Wall IS (NextTo AND $(dir)) THEN
      INFER OkMove$(dir)
    END;
  ENDFOREACH;
  FOREACH ( ahead, back, side1, side2 ) IN
          {( East, West, North, South ), ( East, West, South, North ),
          ( North, South, East, West ), ( North, South, West, East ),
          ( West, East, North, South ), ( West, East, South, North ),
          ( South, North, East, West ), ( South, North, West, East )}
    IF OkMove$(ahead) AND
      An Enemy IS (Near AND $(back)) AND
      An Obstacle IS (NextTo AND $(side1)) THEN
      MULTIACTION
          Move$(ahead)
          Move$(side1)
          Move$(side1)
          Move$(back)
          Push$(side2)
      END
    END
  ENDFOREACH
```

Surrounded:

```
  FOREACH dir IN { East, North, West, South }
```

```
    IF An Obstacle IS (NextTo AND $(dir)) THEN
      INFER OkPush$(dir)
    END;
    IF An Enemy IS (Near AND $(dir)) OR
      A Wall IS (NextTo AND $(dir)) OR
      An Obstacle IS (NextTo AND $(dir)) THEN
      INFER Blocked$(dir)
    END;
    WHEN Surrounded AND OkPush$(dir) AND An Enemy IS Near
      REPEAT
          MULTIACTION
            Push$(dir)
            Move$(dir)
          END
      UNTIL NOT OkPush$(dir)
    END
  ENDFOREACH;
  IF BlockedEast AND BlockedNorth AND
    BlockedSouth AND BlockedWest THEN
    INFER Surrounded
  END
```

Bad Advice

NotSimpleMoves:

```
  FOREACH dir IN { East, North, West, South }
    IF An Obstacle IS (NextTo AND $(dir)) THEN
      INFER OkPush$(dir)
    END;
    IF No Obstacle IS (NextTo AND $(dir)) AND
      No Wall IS (NextTo AND $(dir)) THEN
      INFER OkMove$(dir)
    END;
    IF An Enemy IS (Near AND (NOT $(dir))) THEN
      DO_NOT Move$(dir)
    END;
    IF OkMove$(dir) AND
      A Food IS (Near AND $(dir)) AND
      No Enemy IS (Near AND $(dir)) THEN
      DO_NOT Move$(dir)
    END;
    IF OkPush$(dir) AND
```

```
      An Enemy IS (Near AND $(dir)) THEN
      DO_NOT Push$(dir)
    END
  ENDFOREACH
```

OppositeSimpleMoves:

```
  FOREACH (dir, oppdir) IN { (East, West), (North, South),
                             (West, East), (South, North) }
    IF An Obstacle IS (NextTo AND $(dir)) THEN
      INFER OkPush$(dir)
    END;
    IF No Obstacle IS (NextTo AND $(dir)) AND
      No Wall IS (NextTo AND $(dir)) THEN
      INFER OkMove$(dir)
    END;
    IF An Enemy IS (Near AND $(dir)) THEN
      Move$(dir)
    END;
    IF OkMove$(dir) AND
      A Food IS (Near AND $(dir)) AND
      No Enemy IS (Near AND $(dir)) THEN
      Move$(oppdir)
    END;
    IF OkPush$(dir) AND
      An Enemy IS (Near AND $(dir)) THEN
      Push$(oppdir)
    END
  ENDFOREACH
```

Appendix B

Details of the Soccer Testbed

In this appendix I present the input terms, fuzzy terms, and advice used in the Soccer testbed investigated in Chapter 7.

B.1 Input Features

The input features for each Soccer agent describe the current location for every object on the field. The location and direction of an object is calculated relative to the agent's current position. I describe each object with five values: the *distance* to the object, the x component of the distance to the object, the y component of the distance to the object, the *sine* of the angle to the object, and the *cosine* of the same angle. Some of these features are redundant, but they each provide information that can be useful. The name for an input feature is formed using one of the five prefixes (`DistTo`, `XDistTo`, `YDistTo`, `CosTo`, or `SinTo`), followed by the name of the object. The objects whose distances are measured are: the ball (`Ball`); the opposing players (`Opposition1` and `Opposition2`); the player on the same team (`TeamMate1`); and the left post, right post, and center of each goal (`MyGoalLeftPost`, `OppositionGoalLeftPost`, `MyGoalCenter`, `OppositionGoalCenter`, `MyGoalRightPost`, and `OppositionGoalRightPost`). A sample input feature is `YDistToMyGoalCenter`, which is the distance in the y plane to the player's own goal. The input vector also includes distances to each wall (`LeftEnd`, `RightEnd`, `FarEnd`, and `NearEnd`). All of these features are REAL input features. There is also a BOOLEAN input feature that is true when the player has the ball `HasBall` and a REAL value representing the clock `TimeLeft`. Finally, there are BOOLEAN input features that indicate the previous action taken (in the form *Action*`Taken`). There are 67 total input features.

B.2 Fuzzy Terms

The fuzzy terms for the Soccer environment are:

```
DESCRIPTOR Near  ::= DistTo?(Object) <  2.0
DESCRIPTOR East  ::= CosTo?(Object)  >  0.95
DESCRIPTOR North ::= SinTo?(Object)  >  0.95
DESCRIPTOR West  ::= CosTo?(Object)  < -0.95
DESCRIPTOR South ::= SinTo?(Object)  < -0.95
```

B.3 Advice Provided

The advice used in the Soccer experiments in Chapter 7 appears below. Figure 46 pictorially depicts this advice.

```
FOREACH opp IN { Opposition1, Opposition2 }
  FOREACH dir IN { East, North, West, South }
    IF $(opp) IS Near AND
      $(opp) IS $(dir)) THEN
      INFER Guarded$(dir)
    END
  ENDFOREACH
ENDFOREACH;

IF HaveBall AND NOT GuardedForward THEN
  MoveForward
END;

IF HaveBall AND GuardedForward THEN
  (MoveRight OR MoveRight)
END;

IF HaveBall AND GuardedForward AND GuardedLeft THEN
  MoveRight
END;

IF HaveBall AND GuardedForward AND GuardedRight THEN
  MoveLeft
END;
```

Bibliography

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.

Angluin, D. (1987). Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106.

Atlas, L., Cole, R., Muthusamy, Y., Lippman, A., Connor, J., Park, D., El-Sharkawi, M., & Marks, R. (1990). A performance comparison of trained multilayer perceptrons and trained classification trees. *Proceedings of the IEEE*, 78:1614–1619.

Barto, A., Bradtke, S., & Singh, S. (1995). Learning act using realtime dynamic programming. *Artificial Intelligence*, 72:81–138.

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., 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.

Chou, P. & Fasman, G. (1978). Prediction of the secondary structure of proteins from their amino acid sequence. *Advances in Enzymology*, 47:45–148.

Cleeremans, A., Servan-Schreiber, D., & McClelland, J. (1989). Finite state automata and simple recurrent networks. *Neural Computation*, 1:372–381.

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.

Cohen, B., Presnell, S., Cohen, F., & Langridge, R. (1991). A proposal for feature-based scoring of protein secondary structure predictions. In *Proceedings of the AAAI91 Workshop on AI Approaches to Classification and Pattern Recognition in Molecular Biology*, (pp. 5–20), Anaheim, CA.

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

Cohen, W. (1994). Grammatically biased learning: Learning logic programs using an explicit antecedent description language. *Artificial Intelligence*, 68:303–366.

Cost, S. & Salzberg, S. (1993). A weighted nearest neighbor algorithm for learning with symbolic features. *Machine Learning*, 10:57–78.

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.

Dent, L., Boticario, J., McDermott, J., Mitchell, T., & Zabowski, D. (1992). A personal learning apprentice. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, (pp. 96–103), San Jose, CA.

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.

Elman, J. (1991). Distributed representations, simple recurrent networks, and grammatical structure. *Machine Learning*, 7:195–225.

Fahlman, S. & Lebiere, C. (1990). The cascade-correlation learning architecture. In Touretzky, D., editor, *Advances in Neural Information Processing Systems (volume 2)*. Morgan Kaufmann, Palo Alto, CA.

Fisher, D. & McKusick, K. (1989). An empirical comparison of ID3 and back-propagation. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, (pp. 788–793), Detroit, MI.

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.

Frasconi, P., Gori, M., & Soda, G. (1992). Local feedback multi-layered networks. *Neural Computation*, 4:120–130.

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

Garnier, J. & Robson, B. (1989). The GOR method for predicting secondary structures in proteins. In Fasman, G., editor, *Prediction of Protein Structure and the Principles of Protein Conformation*. Plenum Press, New York.

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).

Giles, C., Miller, C., Chen, D., Chen, H., Sun, G., & Lee, Y. (1992). Learning and extracting finite state automata with second-order recurrent neural networks. *Neural Computation*, 4:393–405.

Ginsberg, A. (1988). *Automatic Refinement of Expert System Knowledge Bases.* Pitman, London.

Gold, E. M. (1972). System identification via state characterization. *Automatica*, 8:621–636.

Gold, E. M. (1978). Complexity of automaton identification from given data. *Information and Control*, 37:302–320.

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.

Hinton, G. E. (1986). Learning distributed representations of concepts. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, (pp. 1–12), Amherst, MA.

Holder, L. B. (1991). *Maintaining the Utility of Learned Knowledge Using Model-Based Control.* PhD thesis, Computer Science Department, University of Illinois at Urbana-Champaign.

Holland, J. (1975). *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, MI.

Holland, J. (1986). Escaping brittleness: The possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In Michalski, R., Carbonell, J., & Mitchell, T., editors, *Machine Learning: An AI Approach (volume 2).* Morgan Kaufmann, San Mateo, CA.

Holley, L. & Karplus, M. (1989). Protein structure prediction with a neural network. *Proceedings of the National Academy of Sciences (USA)*, 86:152–156.

Hopcroft, J. & Ullman, J. (1979). *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, Reading, MA.

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.

Jensen, K. & Wirth, N. (1975). *PASCAL: User Manual and Report.* Springer-Verlag, New York.

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.

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.

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

Lang, K., Waibel, A., & Hinton, G. (1990). A time-delay neural network architecture for isolated word recognition. *Neural Networks*, 3:23–43.

Le Cun, Y., Denker, J., & Solla, S. (1990). Optimal brain damage. In Touretzky, D., editor, *Advances in Neural Information Processing Systems (volume 2).* Morgan Kaufmann, Palo Alto, CA.

Leng, B., Buchanan, B., & Nicholas, H. (1993). Protein secondary structure prediction using two-level case-based reasoning. In *Proceedings of the First International Conference on Intelligent Systems for Molecular Biology*, (pp. 251–259), Washington, DC.

Levine, J., Mason, T., & Brown, D. (1992). *Lex & Yacc.* O'Reilly, Sebastopol, CA.

Lim, V. (1974). Algorithms for prediction of α-helical and β-structural regions in globular proteins. *Journal of Molecular Biology*, 88:873–894.

Lin, L. (1991). Programming robots using reinforcement learning and teaching. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, (pp. 781–786), Anaheim, 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. & Shavlik, J. (1991). Refining domain theories expressed as finite-state automata. In *Proceedings of the Eighth International Machine Learning Workshop*, (pp. 524–528), Evanston, IL.

Maclin, R. & Shavlik, J. (1992). Using knowledge-based neural networks to improve algorithms: Refining the Chou-Fasman algorithm for protein folding. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, (pp. 165–170), San Jose, CA.

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. (1994a). Incorporating advice into agents that learn from reinforcements. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, (pp. 694–699), Seattle, WA.

Maclin, R. & Shavlik, J. (1994b). Refining algorithms with knowledge-based neural networks: Improving the Chou-Fasman algorithm for protein folding. In Hanson, S., Drastal, G., & Rivest, R., editors, *Computational Learning Theory and Natural Learning Systems (volume 1)*. MIT Press, Cambridge, MA.

Maclin, R. & Shavlik, J. (1996). Creating advice-taking reinforcement learners. *Machine Learning.*

Maes, P. & Kozierok, R. (1993). Learning interface agents. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, (pp. 459–465), Washington, DC.

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

Mataric, M. (1994). Reward functions for accelerated learning. In *Proceedings of the Eleventh International Conference on Machine Learning*, (pp. 181–189), New Brunswick, NJ.

Mathews, B. (1975). Comparison of the predicted and observed secondary structure of T4 Phage Lysozyme. *Biochimica et Biophysica Acta*, 405:442–451.

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.).

Minsky, M. & Papert, S. (1969). *Perceptrons.* MIT Press, Cambridge.

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

Moody, J. & Darken, C. (1988). Learning with localized receptive fields. In Hinton, G., Sejnowski, T., & Touretzky, D., editors, *Proceedings of the 1988 Connectionist Models Summer School*, (pp. 133–143), San Mateo, CA. Morgan Kaufmann.

Moody, J. & Darken, C. (1989). Fast learning in networks of locally-tuned processing units. *Neural Computation*, 1:281–294.

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.

Muggleton, S. (1992). *Inductive logic programming.* Academic Press, London.

Muggleton, S. & Feng, C. (1990). Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithmic Theory*, (pp. 1–14), Tokyo, Japan.

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

Nishikawa, K. (1983). Assessment of secondary-structure prediction of proteins: Comparison of computerized Chou-Fasman method with others. *Biochimica et Biophysica Acta*, 748:285–299.

Noelle, D. & Cottrell, G. (1994). Towards instructable connectionist systems. In Sun, R. & Bookman, L., editors, *Computational Architectures Integrating Neural and Symbolic Processes.* Kluwer Academic, Boston.

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 *Proceedings of the Ninth International Conference on Machine Learning*, (pp. 361–366), Aberdeen, Scotland.

Opitz, D. & Shavlik, J. (1993). Heuristically expanding knowledge-based neural networks. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, (pp. 1360–1365), Chambery, France.

Ourston, D. & Mooney, R. (1990). Changing the rules: A comprehensive approach to theory refinement. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, (pp. 815–820), Boston, MA.

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.

Pineda, F. (1987). Generalization of back-propagation to recurrent neural networks. *Physical Review Letters*, 59:2229–2232.

Qian, N. & Sejnowski, T. (1988). Predicting the secondary structure of globular proteins using neural network models. *Journal of Molecular Biology*, 202:865–884.

Quinlan, J. (1990). Learning logical definitions from relations. *Machine Learning*, 5:239–2666.

Richards, B. & Mooney, R. (1995). Automated refinement of first-order Horn-clause domain theories. *Machine Learning*, 19:95–131.

Richardson, J. & Richardson, D. (1989). Principles and patterns of protein conformation. In Fasman, G., editor, *Prediction of Protein Structure and the Principles of Protein Conformation*. Plenum Press, New York.

Riecken, D. (1994). Special issue on intelligent agents. *Communications of the ACM*, 37(7).

Rivest, R. & Schapire, R. (1987). A new approach to unsupervised learning in deterministic environments. In *Proceedings of the Fourth International Workshop on Machine Learning*, (pp. 364–375), Irvine, CA.

Rost, B. & Sander, C. (1993). Prediction of protein secondary structure at better than 70% accuracy. *Journal of Molecular Biology*, 232:584–599.

Rumelhart, D., Hinton, G., & Williams, R. (1986). Learning internal representations by error propagation. In Rumelhart, D. & McClelland, J., editors, *Parallel Distributed Processing: Explorations in the microstructure of cognition. (volume 1).* MIT Press, Cambridge, MA.

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

Salzberg, S. & Cost, S. (1992). Predicting protein secondary structure with a nearest-neighbor algorithm. *Journal of Molecular Biology*, 227:371–374.

Samuel, A. (1959). Some studies in machine learning using the game of checkers. *IBM Journal on Research and Development*, 3:210–229. (Reprinted in E. Feigenbaum and J. Feldman, eds., 1963, *Computers and Thought.* McGraw-Hill, New York).

Schoppers, M. (1994). Estimating reaction plan size. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, (pp. 1238–1244), Seattle, WA.

Sedgewick, R. (1983). *Algorithms.* Addison Wesley, Redding, MA.

Sejnowski, T. & Rosenberg, C. (1987). Parallel networks that learn to pronounce English text. *Complex Systems*, 1:145–168.

Shastri, L. (1988). A connectionist approach to knowledge representation and limited inference. *Cognitive Science*, 12:331–392.

Shavlik, J. & Maclin, R. (1995). Learning from instruction and experience in competitive situations. In *Proceedings of the ML95 Workshop on Agents that Learn from Other Agents*, Tahoe City, CA.

Shavlik, J., Mooney, R., & Towell, G. (1991). Symbolic and neural learning algorithms: An experimental comparison. *Machine Learning*, 6:111–143.

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.

Sun, R. (1992). On variable binding in connectionist networks. *Connection Science*, 4:93–124.

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 International Joint Conference on Artificial Intelligence*, (pp. 930–936), Chambery, France.

Towell, G. (1991). *Symbolic Knowledge and Neural Networks: Insertion, Refinement, and Extraction.* PhD thesis, Computer Sciences Department, University of Wisconsin, Madison, WI.

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.* PhD thesis, King's College, Cambridge.

Watson, J. (1990). The Human Genome Project: Past, present, and future. *Science*, 248:44–48.

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.

Wilson, I., Haft, D., Getzoff, E., Tainer, J., Lerner, R., & Brenner, S. (1985). Identical short peptide sequences in unrelated proteins can have different conformations: A testing ground for theories of immune recognition. *Proceedings of the National Academy of Sciences (USA)*, 82:5255–5259.

Zadeh, L. (1965). Fuzzy sets. *Information and Control*, 8:338–353.

Zhang, X., Mesirov, J., & Waltz, D. (1992). Hybrid system for protein secondary structure prediction. *Journal of Molecular Biology*, 225:1049–1063.