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

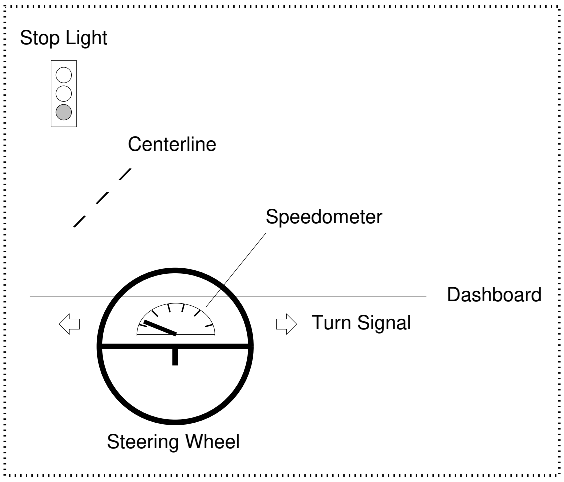


Figure 1: Sample scene showing information that might be available while driving.

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

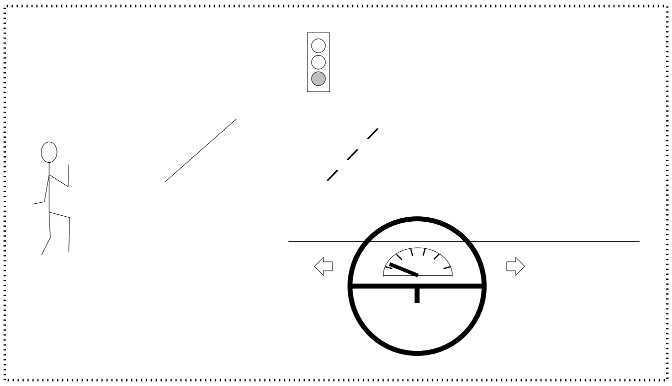


Figure 2: Solving a task that requires a sequence of actions can be difficult if unexpected things can happen – like a pedestrian suddenly appearing.

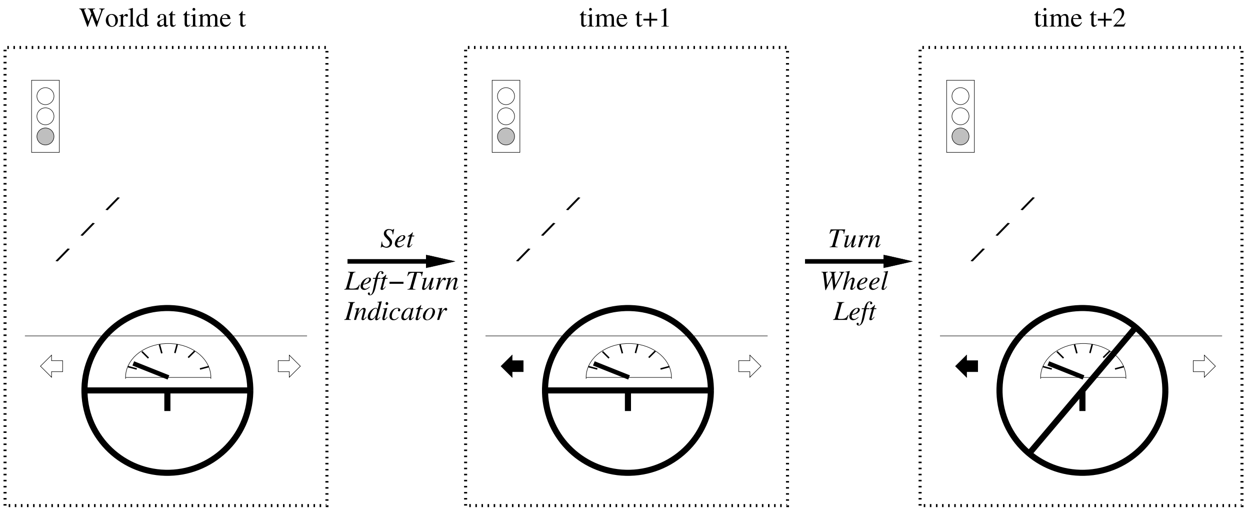


Figure 3: One solution to the problem of representing a series of actions is to represent a task as a series of sub-tasks, with one step per sub-task.

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

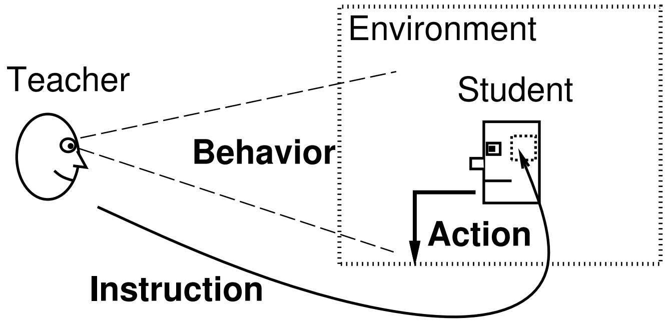


Figure 4: Interaction of the teacher and the student in my system for instructing a reinforcement learner. The process is a continuous loop – the student will continue to explore and learn from experience in its environment until it receives instructions from the teacher; it then stops, incorporates 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 $weight_{j \rightarrow i}$ 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:

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

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:

$$activation_i \;=\; \frac{1}{1 \;+\; e^{-NetInput_i}} \tag{3}$$

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

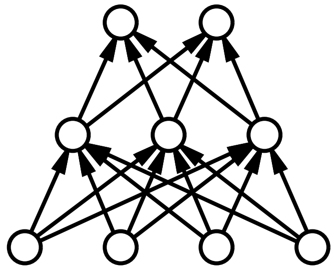


Figure 5: A standard feedforward neural network with one layer of input units, one layer of hidden units, and one layer of output units. Note that input units have links only out of them and output units have links only into them. In future neural-network diagrams, I will leave out the arrow heads on 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

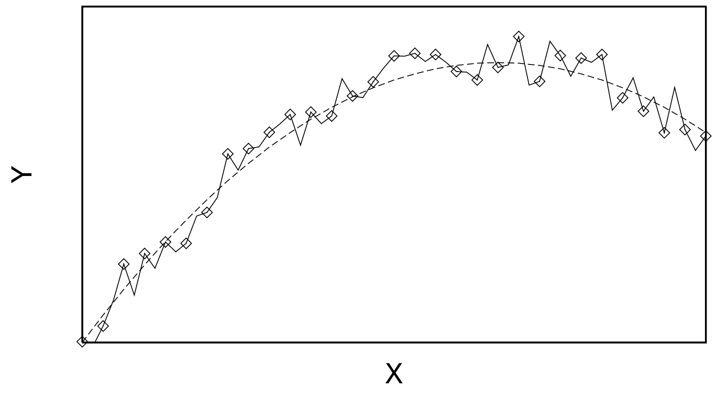


Figure 6: Two possible functions we could fit to a set of points (shown as diamonds). If we assume a certain amount of error in the observed y values associated with the x values, the solid curve is probably a more desirable solution; the dashed line would be a curve that "overfits" the data.

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

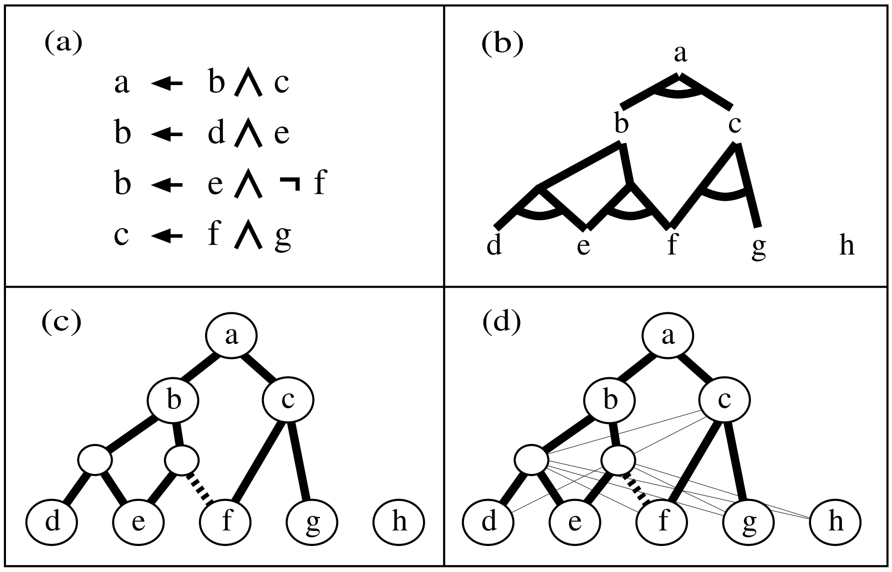


Figure 7: Sample of the KBANN algorithm: (a) a propositional rule set; (b) the rules viewed as an AND-OR dependency graph; (c) each proposition is represented as a unit (extra units are also added to represent disjunctive definitions, e.g., b), and their weights and biases are set so that they implement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

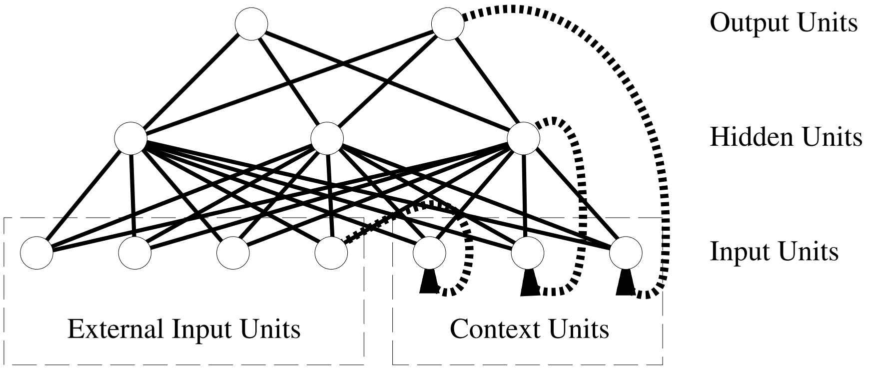


Figure 8: A sample simple recurrent neural network (SRN). The recurrent links are shown as dashed lines. These links are used to create context units, whose activation values are copies of the activation values of the units to which they are connected. The activation value of a context unit is the activation 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

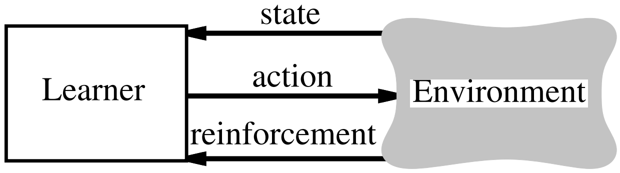


Figure 9: A reinforcement learning agent interacts with the environment in three ways: it receives a description of the state from the environment, selects an action that changes the environment, and then receives a reinforcement signal based on the action it chose.

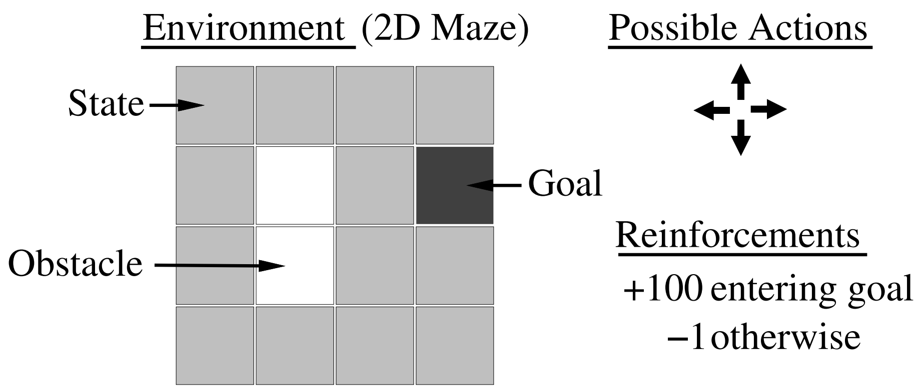


Figure 10: A sample reinforcement learning problem, with the set of possible states, the actions that the agent may take to change (move around in) the environment and the reinforcement signals.

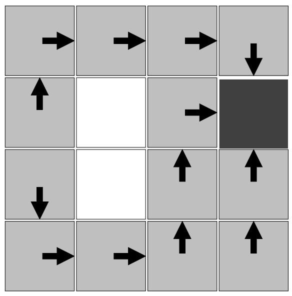


Figure 11: An optimal policy function for the problem shown in Figure 10. An optimal policy function should achieve the maximum future discounted reward.

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.

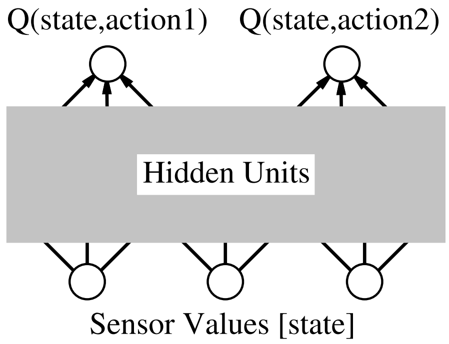


Figure 12: A Q-function can be represented with a neural network. For the neural network, the input is the description of a state and the outputs are the predicted utilities for each of the actions for the input state.

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.

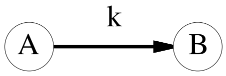
3.2 Transforming FSAs into State-Based Rules

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:



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

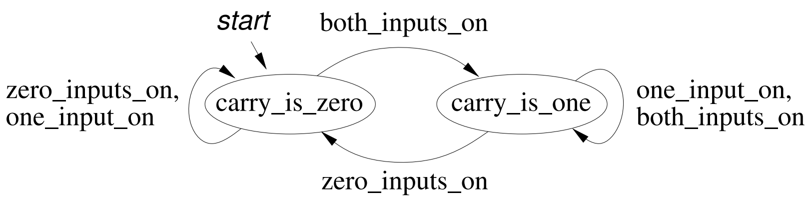


Figure 13: An FSA that could be used to track the carry bit while adding binary numbers.

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

$$\begin{array}{rcll}
both_inputs_on & \leftarrow & input1_on & \wedge\ input2_on \\
one_input_on & \leftarrow & input1_on & \wedge\ \neg input2_on \\
one_input_on & \leftarrow & \neg input1_on & \wedge\ input2_on \\
zero_inputs_on & \leftarrow & \neg input1_on & \wedge\ \neg input2_on \\
output_on & \leftarrow & both_inputs_on & \wedge\ carry_is_one_{i-1} \\
output_on & \leftarrow & one_input_on & \wedge\ carry_is_zero_{i-1} \\
output_on & \leftarrow & zero_inputs_on & \wedge\ carry_is_one_{i-1}
\end{array}$$

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

$$\begin{array}{rcll}
carry_is_zero_i & \leftarrow & carry_is_zero_{i-1} & \wedge\ zero_inputs_on \\
carry_is_zero_i & \leftarrow & carry_is_zero_{i-1} & \wedge\ one_input_on \\
carry_is_zero_i & \leftarrow & carry_is_one_{i-1} & \wedge\ zero_inputs_on \\
carry_is_one_i & \leftarrow & carry_is_one_{i-1} & \wedge\ one_input_on \\
carry_is_one_i & \leftarrow & carry_is_one_{i-1} & \wedge\ both_inputs_on \\
carry_is_one_i & \leftarrow & carry_is_zero_{i-1} & \wedge\ both_inputs_on
\end{array}$$

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.

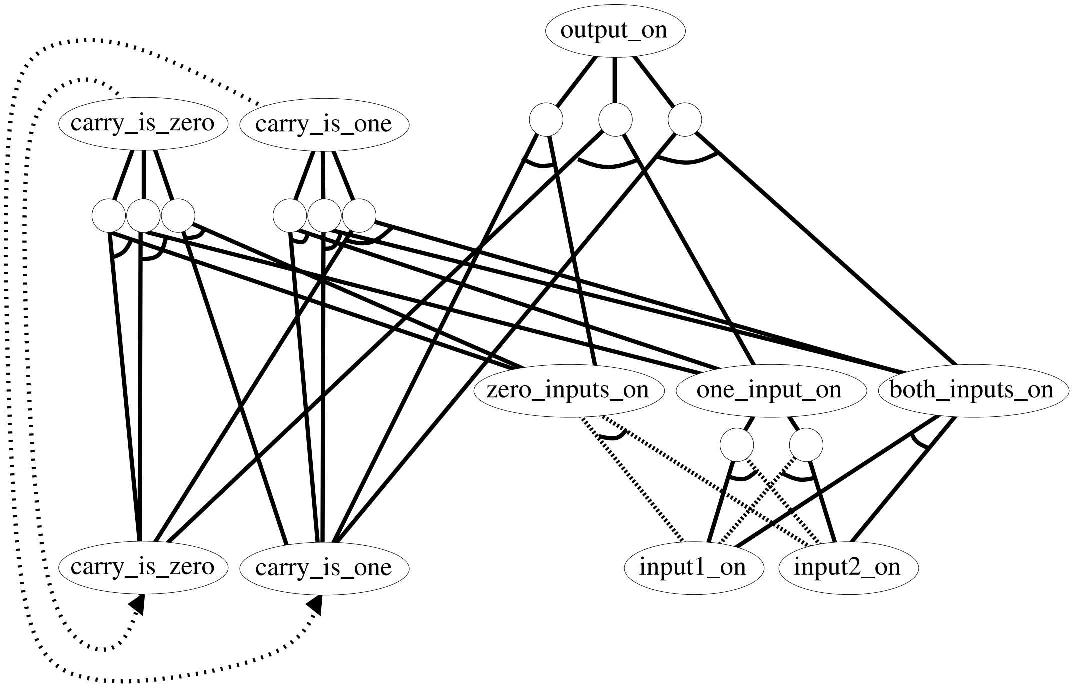


Figure 14: FSKBANN network for rules from Tables 6 and 7. Units constructed as ANDs are shown with arcs, the other units represent ORs. The dashed links with arrow heads are recursive links and the remaining dashed links are negatively-weighted links. The low-weighted links added at the end of the FSKBANN 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

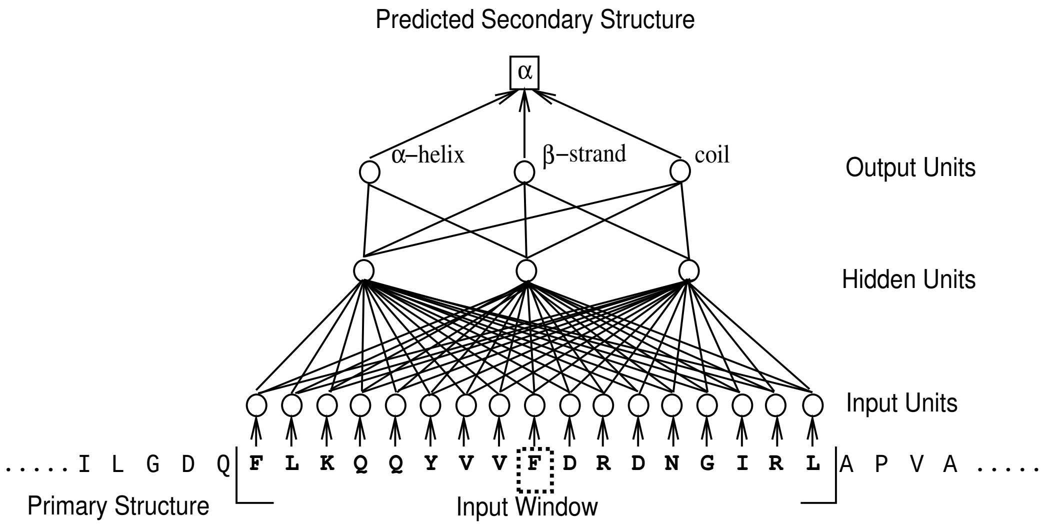


Figure 16: Neural-network architecture used by Qian and Sejnowski (1988). Each amino acid is represented by 21 units, one unit for each amino acid and one unit to denote "off the end." Thus, if the window on the sequence includes 13 amino acids, there are 273 input units.

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

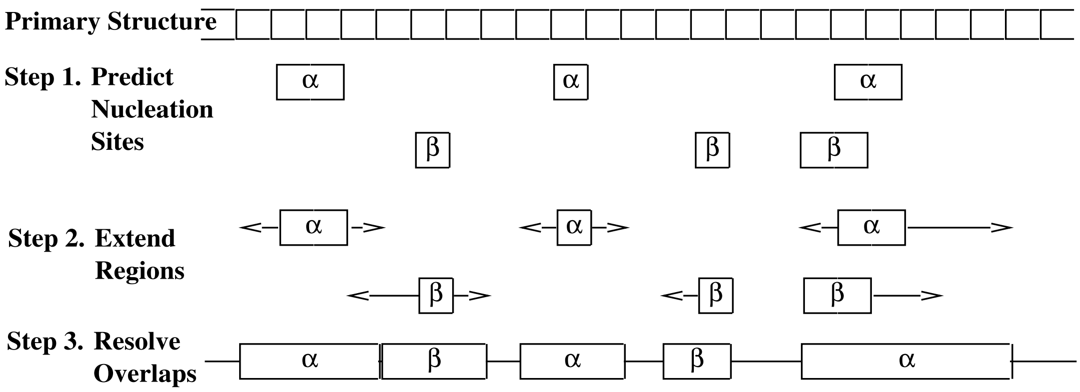


Figure 17: Steps of the Chou-Fasman (1978) algorithm.

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).

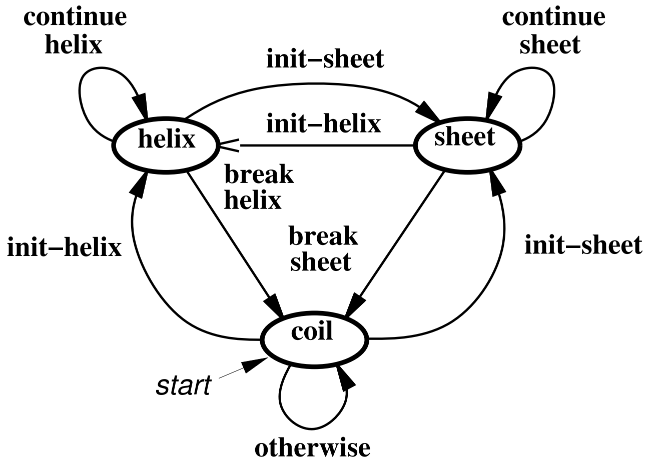


Figure 18: The finite-state automaton interpretation of the Chou-Fasman algorithm.

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.

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

Rules for pairs of amino acids that terminate helix structures.

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

Rules for pairs of amino acids that terminate sheet structures.

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

Rules for continuing structures.

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

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).

$$\begin{array}{rcl}
helix_i & \leftarrow & sheet_{i-1} \wedge init_helix \\
helix_i & \leftarrow & coil_{i-1} \quad \wedge init_helix \\
helix_i & \leftarrow & helix_{i-1} \wedge cont_helix \\[4pt]
sheet_i & \leftarrow & helix_{i-1} \wedge init_sheet \\
sheet_i & \leftarrow & coil_{i-1} \quad \wedge init_sheet \\
sheet_i & \leftarrow & sheet_{i-1} \wedge cont_sheet \\[4pt]
coil_i & \leftarrow & helix_{i-1} \wedge break_helix \\
coil_i & \leftarrow & sheet_{i-1} \wedge break_sheet \\
coil_i & \leftarrow & coil_{i-1} \quad \wedge any
\end{array}$$

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

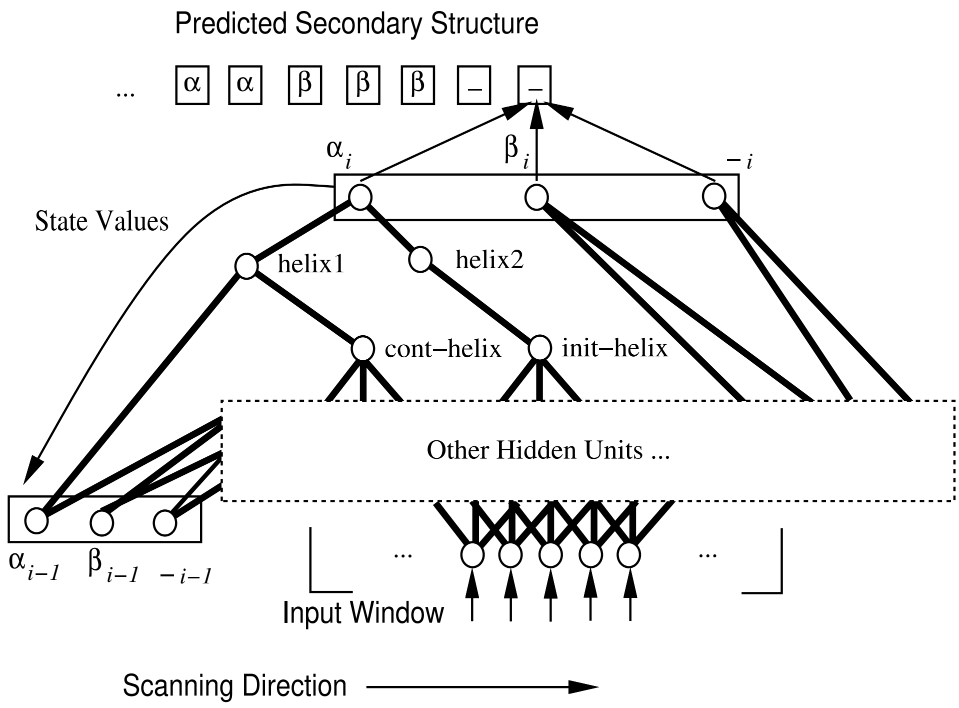


Figure 19: General neural-network architecture used to represent the Chou-Fasman algorithm. Note that the low-weighted links added at the end of the translation process are not shown.

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):

$$C = \frac{P\,N - F\,M}{\sqrt{(P + F)(P + M)(N + F)(N + M)}} \qquad (12)$$

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

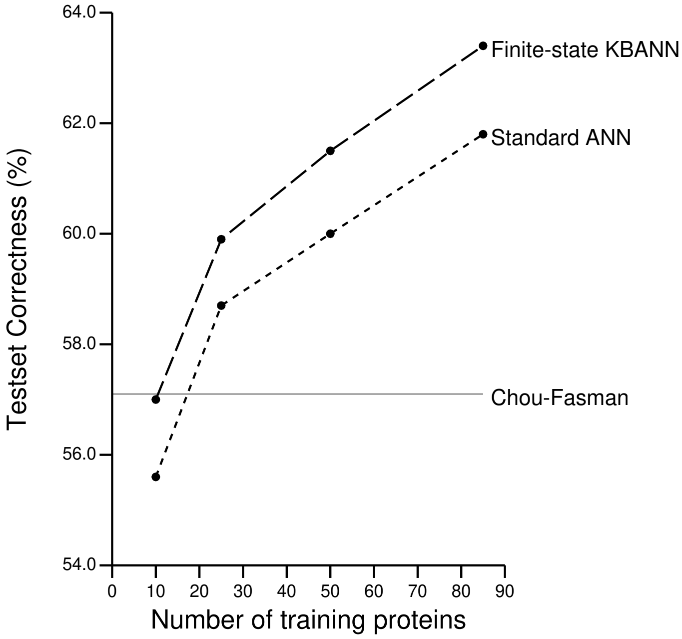


Figure 20: Percent correctness on test proteins as a function of training-set size.

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.

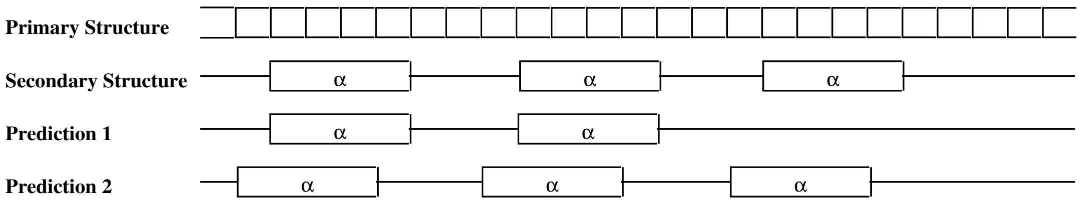


Figure 21: Two possible predictions for secondary structure.

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

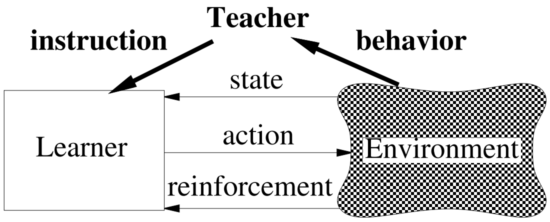


Figure 22: Reinforcement learning with a teacher.

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.

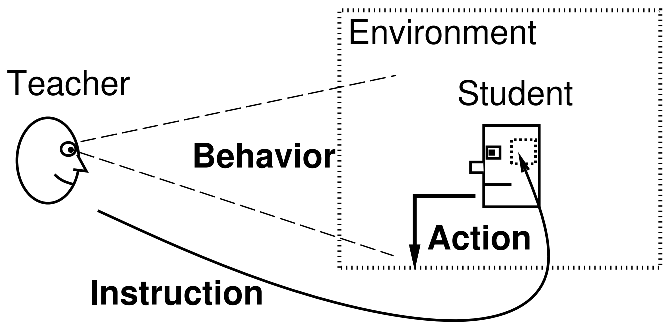


Figure 23: The interaction of the teacher and agent in RATLE. The process is a cycle: the observer watches the agent's behavior to determine what advice to give, and the advice-taking system processes the advice and inserts it into the agent's "brain", which changes the agent's behavior. The agent operates 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

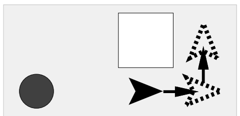
As I argued in Chapter 1, a key aspect of the interaction between a teacher and a student is the language the teacher uses to provide instructions. In RATLE, the teacher is advising an RL agent; therefore the language naturally has constructs that relate to the type of decisions an RL agent makes – which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

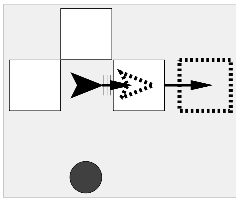
The RATLE language resembles a programming language by design. I use a programming language because it allows me to make use of standard parsing techniques. Thus, I can focus on the process of transferring knowledge to an RL agent, rather than trying to address the natural language problem. I also make 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.

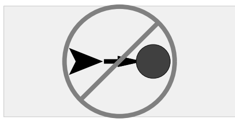
RATLE allows a user to give simple instructions in the form of IF and REPEAT statements specifying conditions of the world and actions to be taken under those conditions. These types of instructions closely correspond to the knowledge that an RL agent is trying to acquire, but in a manner that is "natural" 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

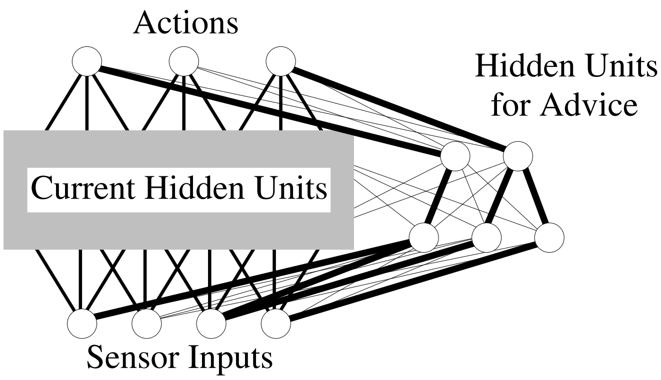


Figure 24: Adding advice to the RL agent's neural network by creating new hidden units that represent the advice. The thick links on the right capture the semantics of the advice. The added thin links initially have near-zero weight; during subsequent backpropagation training the magnitude of their weights 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

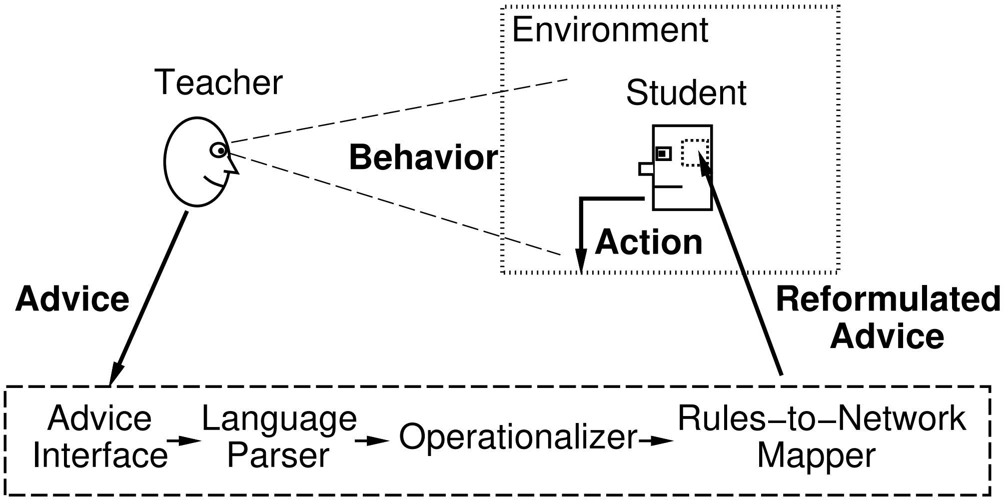


Figure 25: The interaction of the teacher, agent, and advice-taking components of RATLE. Advice developed by the teacher is transformed using a process based on Hayes-Roth, Klahr, and Mostow's (1981) advice-taking formalism.

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*	$[Type1]$
Quantifier Object { IS | ARE } *Properties*	$[Type2]$

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*	[$Type1$]
Quantifier Object { IS | ARE } *Properties*	[$Type2$]

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.

stmts	←	*stmt* | *stmts* ";" *stmt*
stmt	←	IF *ante* THEN *conc else* END
		| WHILE *ante* DO *act postact* END
		| *pre* REPEAT *act* UNTIL *ante postact* END
else	←	ε | ELSE *conc*
postact	←	ε | THEN *act*
pre	←	ε | WHEN *ante*
conc	←	*act* | INFER *name* | *remember name*
act	←	*cons* | MULTIACTION *clist* END
clist	←	*cons* | *cons clist*
cons	←	Name | DO_NOT Name | "(" *corlst* ")"
corlst	←	Name | Name OR *corlst*
ante	←	Name | "(" *ante* ")" | NOT *ante*
		| *ante* AND *ante* | *ante* OR *ante*
		| *quant* Name *isare desc*
quant	←	ε | Name
isare	←	IS | ARE
desc	←	Name | NOT *desc* | (*dexpr*)
dexpr	←	*desc* | *dexpr* AND *dexpr* | *dexpr* OR *dexpr*

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.

$$
\begin{array}{lll}
\textit{inputs} & \leftarrow & \textit{input} \mid \textit{inputs} \text{ ";" } \textit{input} \\[6pt]
\textit{input} & \leftarrow & \text{BOOLEAN } \textit{Name} \\
 & & \mid \text{REAL } \textit{proplst} \text{ "[" } \textit{Number} \text{ ".." } \textit{Number normvs} \text{ "]"} \\[6pt]
\textit{proplst} & \leftarrow & \textit{prop} \mid \textit{proplst prop} \\
\textit{prop} & \leftarrow & \textit{Name} \text{ IN "[" } \textit{Number} \text{ ".." } \textit{Number} \text{ "]"} \\
 & & \mid \textit{Name} \text{ "=" } \textit{Number} \\[6pt]
\textit{normvs} & \leftarrow & \varepsilon \mid \text{ "," } \textit{Number} \text{ .. } \textit{Number}
\end{array}
$$

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

$$
\begin{array}{lll}
\mathit{terms} & \leftarrow & \mathit{term} \mid \mathit{terms} \text{ ``;'' } \mathit{term} \\[6pt]
\mathit{term} & \leftarrow & \textsc{descriptor}\ \mathit{partial} \text{ ``::='' } \mathit{psum}\ \mathit{op}\ \mathit{Number} \\
 & & \mid \textsc{property}\ \mathit{Name} \text{ ``::='' } \mathit{prop} \\
 & & \mid \textsc{quantifier}\ \mathit{Name} \text{ ``::='' } \mathit{sum}\ \mathit{qrest} \\
\mathit{qrest} & \leftarrow & \mathit{op}\ \mathit{Number} \\
 & & \mid \textsc{in} \text{ ``['' } \mathit{Number} \text{ ``..'' } \mathit{Number} \text{ ``]''} \\[6pt]
\mathit{partial} & \leftarrow & \mathit{namev} \mid \mathit{partial}\ \mathit{namev} \\
\mathit{namev} & \leftarrow & \mathit{Name} \mid \text{ ``?('' } \mathit{Name} \text{ ``)''} \\[6pt]
\mathit{psum} & \leftarrow & \mathit{partial} \mid \text{ ``('' } \mathit{osign}\ \mathit{partial}\ \mathit{plist} \text{ ``)''} \\
\mathit{plist} & \leftarrow & \varepsilon \mid \mathit{sign}\ \mathit{partial}\ \mathit{plist} \\
\mathit{osign} & \leftarrow & \varepsilon \mid \text{ ``}-\text{''} \\
\mathit{sign} & \leftarrow & \text{``}+\text{''} \mid \text{ ``}-\text{''} \\[6pt]
\mathit{op} & \leftarrow & \text{``}<\text{''} \mid \text{``}>\text{''} \mid \text{``}<=\text{''} \mid \text{``}>=\text{''} \mid \text{``}=\text{''} \mid \text{``}!=\text{''} \\[6pt]
\mathit{prop} & \leftarrow & \mathit{Name}\ \textsc{in} \text{ ``['' } \mathit{Number} \text{ ``..'' } \mathit{Number} \text{ ``]''} \\
 & & \mid \mathit{Name} \text{ ``='' } \mathit{Number} \\[6pt]
\mathit{sum} & \leftarrow & \varepsilon \mid \textsc{sum}
\end{array}
$$

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