Connectionist Modeling and Control of Finite State Environments

Jonathan Richard Bachrach

COINS Technical Report 92-6

January 1992

CONNECTIONIST MODELING AND CONTROL OF FINITE STATE ENVIRONMENTS

A Dissertation Presented

by

JONATHAN RICHARD BACHRACH

Submitted to the Graduate School of the

University of Massachusetts in partial fulfillment

of the requirements for the degree of

DOCTOR OF PHILOSOPHY

February 1992

Department of Computer and Information Sciences

CONNECTIONIST MODELING AND CONTROL OF FINITE STATE ENVIRONMENTS

A Dissertation Presented

by

JONATHAN RICHARD BACHRACH

Approved as to style and content by:

____

Andrew G. Barto, Chair

____

Michael C. Mozer, Member

____

Roderic A. Grupen, Member

____

Christopher V. Hollot, Member

____

W. Richards Adrion, Department Chair
Computer and Information Sciences

To such
supercalifragilisticexpeeallidoescious
parents,
Lou Baby and Santa Fe,
who made my life possible and
filled my life with love.
I love you.

Acknowledgements

I am indebted to Andy Barto for being such a superior advisor. Above all, Andy taught me how to write. He patiently read countless drafts and gave bunches of helpful comments.

I would like to thank the entire UMass Adaptive NetWorks group: Chuck Anderson, Steve Bradtke, Judy Franklin, Robbie Jacobs, Mike Jordan, Stephen Judd, Brian Pinette, Satinder, Vijay, and Richard Yee. They have helped me greatly. I would especially like to thank Robbie Jacobs and Brian Pinette. They both have been good friends and good critics. Brian has also helped me refine my ideas on robot navigation.

Dave Rumelhart has been very influencial and generous to me. I spent a year in his lab as a visiting graduate student. Most of the navigation ideas were developed with him while visiting his lab. I am constantly awed by his genious.

While in Dave's lab I benefited from my interactions with the other members of the PDP group: Bo Curry, Richard Durbin, Richard Golden, Chris Kortge, Ben Martin, Dan Rosen, Charlie Rosenberg, and Peter Todd. During that time I had the pleasure of working very closely with Richard Durbin. He was extremely helpful and I benefited enormously from my interactions with him. Richard is one smart man.

Mike Mozer and I have collaborated on ideas for several years now. He has been very instrumental in my thinking about the ideas presented in this dissertation. Furthermore, he has directly contributed some of these ideas, especially those presented in Chapter 3. Mike Mozer is one impressive thinking fellow.

Earlier in my graduate career I spent two summers working at GTE labs. There I had the honor of working with Rich Sutton. Above all, Rich taught me how to be a good scientist.

I had the pleasure of working with Mike Jordan while he was a post doc at UMass. His thoughts on motor control have been enormously influencial on me. Mike Jordan is one amazing renaissance fellow.

Finally, this research was supported by funding provided to Andrew G. Barto by the Air Force Office of Scientific Research, Bolling AFB, under Grant AFOSR-89-0526, and by the National Science Foundation under Grant ECS-8912623.

ABSTRACT

CONNECTIONIST MODELING AND CONTROL OF FINITE STATE ENVIRONMENTS

FEBRUARY 1992

JONATHAN RICHARD BACHRACH,

B.S., UNIVERSITY OF CALIFORNIA, SAN DIEGO

M.S., UNIVERSITY OF MASSACHUSETTS

PH.D., UNIVERSITY OF MASSACHUSETTS

Directed by: Professor Andrew G. Barto

A robot wanders around an unfamiliar environment, performing actions and observing their perceptual consequences. The robot's task is to construct a model of its environment that will allow it to predict the outcome of its actions and to determine what action sequences take it to particular goal states. In any reasonably complex situation, a robot that aims to manipulate its environment toward some desired end requires an internal representation of the environment because the robot can directly perceive only a small fraction of the global environmental state at any time; some portion of the rest must be stored internally if the robot is to act effectively. Rivest and Schapire [72, 74, 87] have studied this problem and have designed a symbolic algorithm to strategically explore and infer the structure of finite-state environments. At the heart of this algorithm is a clever representation of the environment called an *update graph*. This dissertation presents a connectionist implementation of the

update graph using a highly specialized network architecture and a technique for using the connectionist update graph to guide the robot from an arbitrary starting state to a goal state. This technique requires a *critic* that associates the update graph's current state with the expected time to reach the goal state. At each time step, the robot selects the action that minimizes the output of the critic. The basic control acquisition technique is demonstrated on several environments, and it is generalized to handle a navigation task involving a more realistic environment characterized by a high-dimensional continuous state-space with real-valued actions and sensations in which a simulated cylindrical robot with a sensor belt operates in a planar environment. The task is short-range homing in the presence of obstacles. Unlike many approaches to robot navigation, our approach assumes no prior map of the environment. Instead, the robot has to use its limited sensory information to construct a model of its environment. A connectionist architecture is presented for building such a model. It incorporates a large amount of a priori knowledge in the form of hard-wired networks, architectural constraints, and initial weights. This navigation example demonstrates the use of a large modular architecture on a difficult task.

Table of Contents

Page

Acknowledgements . . . iv

Abstract . . . vi

List Of Tables . . . x

List Of Figures . . . xi

Chapter

1. INTRODUCTION . . . 1
   - 1.1 Partial State Information . . . 2
   - 1.2 Modeling . . . 3
     - 1.2.1 State Estimation and Real-World Issues . . . 10
     - 1.2.2 Connectionist Networks . . . 11
     - 1.2.3 Our Modeling Approach . . . 12
   - 1.3 Control Acquisition . . . 15
     - 1.3.1 Partial State Information . . . 16
   - 1.4 Navigation . . . 17
   - 1.5 Organization of the Dissertation . . . 19

2. PREVIOUS MODELING RESEARCH . . . 20
   - 2.1 Techniques Based on Linear System Theory . . . 23
   - 2.2 Connectionist Modeling Methods . . . 27
     - 2.2.1 Delay-Coordinate Models . . . 27
     - 2.2.2 State-Space Models . . . 28
     - 2.2.3 Relation of SLUG to Previous Connectionist Architectures . . . 34
   - 2.3 Finite State Automaton Identification . . . 35
     - 2.3.1 Automata Theory . . . 35
     - 2.3.2 General Systems Theory . . . 36
     - 2.3.3 Computational Learning Theory . . . 38
     - 2.3.4 Relation of SLUG to Finite State Automaton Identification . . . 40
   - 2.4 Conclusion . . . 41

3. MODELING FINITE STATE AUTOMATA 42
3.1 Environments 42
3.2 The Update Graph 46
3.2.1 The Rivest and Schapire Learning Algorithm 49
3.3 Connectionist Approach to Modeling Environments 50
3.4 Training SLUG 51
3.5 Results 58
3.6 Comparing SLUG and Other Connectionist Approaches 63
3.7 Limitations of SLUG 70
4. LEARNING TO CONTROL A FINITE STATE AUTOMATON ... 72
4.1 Training 73
4.2 Simulation Results 75
4.3 Relation to Other Work 80
4.4 Discussion of Other Possible Approaches 86
5. NAVIGATION 88
5.1 Artificial Potential Functions 91
5.2 The Navigation Architecture 93
5.3 Training 101
5.4 Perceptual Servoing 107
5.5 Results 108
5.6 Discussion 110
5.7 Future Work 113
5.8 Related Research 119
6. CONCLUSIONS 122
APPENDIX. THE UPDATE GRAPH 126
A.1 The Diversity Representation 126
A.2 The Update Graph 129
A.3 The Rivest-Schapire Learning Algorithm 132
REFERENCES 137

LIST OF TABLES

Table Page

1.1 A Two-Dimensional Categorization of the Various State Representations for Finite State Environments. 5

3.1 Number of Steps Required to Learn Update Graph. 67

3.2 Number of Steps Required to Learn Update Graph as the Number of Units in SLUG Is Varied. 70

4.1 The Control Acquisition Architecture was Applied to these Environments with their Respective Goal States. 75

4.2 An Example State/Action Sequence for the Car Radio Environment. 77

4.3 Performance of the Control Acquisition Architecture. 79

5.1 The Gradient Descent Procedure for Choosing Actions 106

5.2 The Gradient Descent Procedure for Perceptual Servoing. 107

A.1 Diversity-Based Inference Procedure. 133

A.2 Probabilistic Procedure for Determining $t \equiv t'$ Given Tolerable Probability of Error ϵ. 135

List of Figures

Figure Page

1.1 An Example of the Need for Internal State in the $n \times n$ Checkerboard Environment. 3

1.2 The 3-Bit Shift Register Environment. 6

1.3 The State Transition Graph Machine for the 3-Bit Shift Register Environment. 7

1.4 The SLUG Architecture. 13

1.5 Connectionist Control Acquisition Architecture. 16

2.1 An Example of a FSA that Cannot Be Accurately Modeled with a Delay-Coordinate State Representation. 22

2.2 An Example of the Unfolding Procedure which Turns Recurrent Networks into Feed-Forward Networks. 30

2.3 The Jordan Feedback Architecture. 33

2.4 The Elman Feedback Architecture. 34

3.1 The Little Prince Environment. 43

3.2 The Car Radio Environment. 44

3.3 An Example of the n-Bit Register Environment. 44

3.4 An Example of the $n \times n$ Grid Environment. 45

3.5 An Example of the $n \times n$ Checkerboard Environment. 46

3.6 The Mechanics of the Update Graph. 47

3.7 The Update Graph for the 3-Bit Shift Register. 48

3.8 Slug Depicted as a Separate Weight Matrix for Each Action. 52

3.9 SLUG Is Dynamically Rewired Contingent on the Current Action. . . 53

3.10 SLUG's Weights at Three Stages of Training for the 3-Bit Shift Register Environment. 59

3.11 The Underlying z_{aij} Parameters. 60

3.12 The Pattern of Activity in SLUG at Six Consecutive Time Steps. . . 60

3.13 Weights Learned by SLUG with Six Units and Unconstrained Weights for the 3-Bit Shift Register Environment. 63

3.14 Weights Learned by SLUG with Three Units and Unconstrained Weights for the 3-Bit Shift Register Environment. 65

4.1 Connectionist Control Acquisition Architecture. 73

4.2 Summary of the Control Law and Evaluation Function for the 3-Bit Shift Register Environment. 76

4.3 Two Example Trajectories for the 8×8 Checkerboard Environment. . 77

4.4 The Output of the Adaptive Critic at Each of the Squares in the 8×8 Checkerboard Environment. 78

4.5 Learning Curves for the 4-Bit Shift Register Environment. 81

4.6 Control Architectures. 83

5.1 The Simulated Robot. 88

5.2 The Navigation Simulator. 90

5.3 The Artificial Potential Function Method. 92

5.4 The Navigation Architecture. 94

5.5 The Sensor Value Smoothing Process. 96

5.6 The Operation Of The Range-Flow Model. 97

5.7 The Range-Flow Model Network. 99

5.8 Gaussian Interpolation. 100

5.9 A Critic Network. 101

5.10 The Clustering Phase. 103

5.11 The Response of Single Gaussian Hidden Units Across the Robot's Environment. 103

5.12 The Total Response of All the Gaussian Hidden Units Across the Robot's Environment. 104

5.13 Two Examples of Perceptual Servoing in the Door Environment. . . . 108

5.14 The Obstacle Environment. 109

5.15 Contour Plots of the Outputs of the Homing Critic and the Combined Homing and Obstacle-Avoidance Critic for the Obstacle Environment. 109

5.16 Contour Plots of the Outputs of the Homing Critic and the Combined Homing and Obstacle-Avoidance Critic for the Door Environment. . . 110

5.17 Trajectories Formed on the Obstacle Environment Using Only the Homing Critic. 111

5.18 Trajectories Formed on the Door Environment Using Only the Homing Critic. 112

5.19 Trajectories Formed on the Obstacle Environment Using the Homing and Obstacle-Avoidance Critic. 113

5.20 Trajectories Formed on the Door Environment Using the Homing and Obstacle-Avoidance Critic. 114

5.21 An Autoencoder Network for Learning a Two-Dimensional Code for Views. 117

5.22 The Result of Training an Autoencoder on a Simple Environment Without Obstacles. 118

5.23 The Influence of Destination on Obstacle-Avoidance. 118

A.1 Some Test Equivalence Classes for the 3-Bit Shift Register Environment. 127

A.2 The Diversity-Based State Variables for the 3-Bit Shift Register Environment. 129

A.3 The Values for the Diversity-Based State Variables for the Given State of the 3-Bit Shift Register. 129

A.4 Some Relationships Between Canonical Tests for the 3-Bit Shift Register Environment. 130

A.5 The Update Graph for the 3-Bit Shift Register. 130

A.6 A Series of Steps in the Diversity-Based Inference Process of the Update Graph for the 3-Bit Shift Register Environment. 134

Chapter 1

INTRODUCTION

A robot is placed in an unfamiliar environment. It explores the environment by performing actions and observing their perceptual consequences. The robot's task is to construct an internal model of the environment that will allow it to predict the outcome of its actions and to determine what sequences of actions it should take to reach particular goal states. This scenario is extremely general, applying not only to physical environments, but also to abstract and artificial environments such as electronic devices (e.g., a VCR, phone answering machine, car radio), computer programs (e.g., a text editor), and classical AI problem-solving domains (e.g., the blocks world). Any agent—human or computer—that aims to flexibly manipulate its environment toward some desired end requires an internal representation of its environment because, in any reasonably-complex situation, the agent can directly perceive only a small fraction of the global environmental state at any time; some portion of the rest must be stored internally if the agent is to act effectively.

The goal of this thesis is to develop connectionist learning algorithms that can efficiently construct models of, and controllers for, unfamiliar environments. In each environment, the robot has a set of *actions* it can execute to move from one environmental state to another. At each environmental state, a set of *sensations* can be detected by the robot. The sensation values comprise the robot's *view*.

A *model* is used to predict the environment's behavior, and a *controller* is used to determine sequences of actions that efficiently lead to desirable environmental states.

The modeling task involves constructing a model that mimics the behavior of the environment in the sense that, given an identical sequence of actions, the model will yield predictions of the sensations that match those produced by the environment. Although a model cannot be constructed without carefully exploring the environment, the emphasis of this thesis is not on the exploration problem, and the most primitive exploration strategies are assumed. The control task involves producing a minimal length sequence of actions to reach a particular goal state.

1.1 Partial State Information

We assume that an environment is a discrete-time dynamical system with finite state set, and we assume the robot has only partial knowledge of the state set, the state transition function, and the output function. Furthermore, we assume the robot has only limited knowledge of the current environmental state at any time step. One central focus of this thesis is on environments with incompletely observed states. Any particular view can be generated from more than one environmental state; that is, the mapping from environmental states to views is many-to-one. Whitehead and Ballard [104] call this problem *perceptual aliasing* and call a view *perceptually ambiguous* if it can represent multiple environmental states.

Model building and control are particularly difficult when the robot does not have access to the complete state of the environment. In many cases, the robot's current view will not suffice for the particular task because the model and controller might have to output different values in different environmental states having the same or similar views. As an example, consider the problem of learning to predict the view after taking a particular action in the 4×4 checkerboard environment shown in Figure 1.1. Half the squares in the environment yield the same sensations and are thus perceptually ambiguous (assuming the robot sees only one square at a time). For example, the two squares labeled A and B in Figure 1.1 yield exactly the same view, and without some mechanism to disambiguate these squares, it is impossible

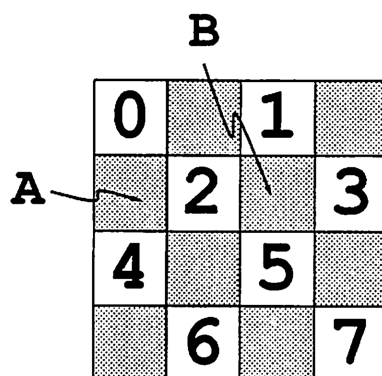

Figure 1.1. An Example of the Need for Internal State in the $n \times n$ Checkerboard Environment. The checkerboard environment consists of an $n \times n$ grid of squares. The robot occupies a particular square and can sense the landmark in that square, if any. There are a number of unique landmarks distributed in a checkerboard pattern across the grid. These are depicted as the numbers in the squares. Half of the squares have no landmark and are indistinguishable from each other. At each time step the robot can move either up, down, left, or right one square. Movement off one edge of the grid wraps around to the other side.

to learn a model of this environment. Without such a mechanism, a model must predict—incorrectly—that the next view after moving right starting in the squares labeled A and B will be the same; whereas in fact, the actual next view would be landmark 2 and landmark 3 respectively. In order to solve this problem, the learning system must represent the state of the environment in some fashion.

1.2 Modeling

A learning system can represent the environment's state by constructing a model of the environment. The learning system is trained to produce a model with *internal states* that correspond to environmental states and that serve to distinguish the perceptually ambiguous environmental states. A *state representation* captures the salient aspects of the unbounded history of the environment's behavior necessary to predict the environment's future behavior. We present two canonical representations that correspond to the extremes of the representational spectrum. These are

the *delay-coordinate* representation and the *state-space* representation. (These two techniques are discussed in greater detail in Chapter 2.) Each succinctly encodes an environment's past behavior.

In the delay-coordinate representation, the environment's state is encoded as a fixed number of past actions and views. Engineering techniques that use this representation include adaptive filters [105] and DARMA (Deterministic Auto Regressive Moving Average) models [30]. Note that some FSA's cannot be accurately modeled using a delay-coordinate state representation. (See Chapter 2 for an example of such an FSA.)

Because of this limitation, we restrict our attention to the state-space representation, where the state of the environment is represented in the model as the value of a vector of state variables. In this case, the state representation corresponds to the particular encoding of the environmental state by the state variables. The many state-space approaches for building models of finite state environments can be compared and contrasted along two main dimensions that refer to the state representation. The first dimension pertains to the range of values that can be taken by each state variable. Two representations are distinguished: the *discrete* representation in which the range of each state variable is a finite set, and the *continuous* representation in which the state variales are real-valued. For the purposes of this discussion and without loss of generality, the state variables in the discrete representation are Boolean variables. The second dimension concerns how states are assigned to patterns of values of the state variables. Again two representations are distinguished: the *localist* representation, which assigns each environmental state to an individual state variable, and the *distributed* representation, which encodes each environmental state as a *pattern* of values across multiple state variables. Table 1.1 summarizes the four possible combinations of range and encoding distinctions we have made. In the following text we are concerned with three of these, which

Table 1.1. A Two-Dimensional Categorization of the Various State Representations for Finite State Environments. The first dimension refers to the range of each state variable, and the second dimension refers to the encoding of state across state variables. Machines that update the values on these state variables are listed in three of the entries.

	Localist	Distributed
Discrete	State Transition Graph Machine	Update Graph
Continuous		State-Space Model SLUG

we call the discrete-localist approach, the discrete-distributed approach, and the continuous-distributed approach.

It is equally important to discuss the mechanism by which the values of state variables are calculated. For the purposes of this dissertation, we consider a restricted class of machines that update the values of the state variables. (For now we do not consider the important problem of how the robot's sensations can be predicted based on the internal state of the machine.) The state variables' values are updated using a network of primitive machines, where each machine updates one of the state variables. Each primitive machine computes the value of its state variable as a function of the robot's chosen action and the values of state variables associated with other machines in the network to which it is connected.

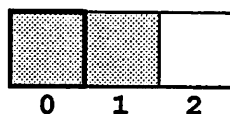

Figure 1.2. The 3-Bit Shift Register Environment. White squares indicate that the bit is on, and shaded squares indicate that the bit is off. The leftmost square is enclosed in a bold frame to symbolize that it is a sensation.

We describe an example machine for each main representational approach shown in Table 1.1 We call these machines the discrete-localist machine, the discrete-distributed machine, and the continuous-distributed machine. Note that although continuous-distributed machines are not FSA's, they can represent the behavior of FSA's. In order to facilitate comparisons, we define the size of a machine to be the number of primitive machines of which it is composed plus the number of connections between the primitive machines. We illustrate these various machines and their associated representations by considering the 3-bit shift register environment shown in Figure 1.2. In this environment, the robot senses the value on the leftmost bit of an n-bit shift register, can rotate the register left or right, and can flip the leftmost bit.

In a discrete-localist machine for the 3-bit shift register, each state of the register is represented by a state variable. The value of the state variable corresponding to state q of the register is one if the register's state is q and is zero otherwise. Thus, only one state variable has value one at any time step. The values of state variables in this discrete-localist representation can be updated using a *state transition graph machine*, where each node in the graph is a primitive machine with one state variable. Labeled links connecting each node of the graph to other nodes correspond to the actions that the robot can execute to move between environmental states.

Associated with each node is a set of discrete-valued sensations that can be detected by the robot when the environment is in the corresponding environmental state. Figure 1.3 shows the state transition graph machine for the 3-bit shift register

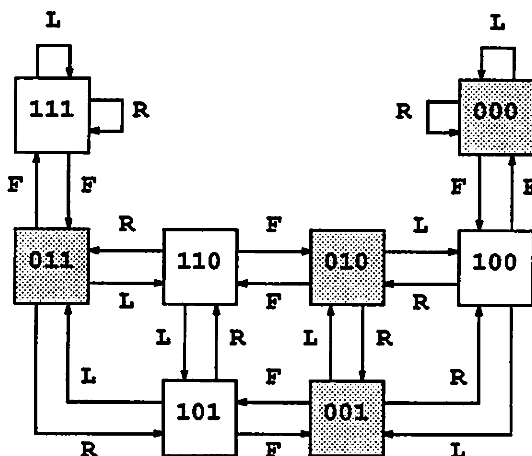

Figure 1.3. The State Transition Graph Machine for the 3-Bit Shift Register Environment. Each box represents a state with the shade of the box coding the single sensation value for that state. A shaded square indicates that the leftmost bit is off, a light square that the leftmost bit is on. The transitions are depicted as arrows labeled with the actions, where `L` is shift left. `R` is shift right, and `F` is flip the leftmost bit.

environment. Each node is identified by the corresponding environmental state, labelled by the settings of bits in the shift register. This environment has $2^3 = 8$ states, one for each of the possible settings of the bits in the register. Here, there is only one sensation, the value of the leftmost bit, which is depicted in Figure 1.3 by the shade of the node. Each link between nodes is labeled with one of the three actions: flip (`F`), shift left (`L`), or shift right (`R`).

The state transition graph machine is one way to represent the behavior of the environment. If the state transition graph machine is known, one can predict the sensory consequences of any sequence of actions. Further, the state transition graph machine can be used to determine a sequence of actions needed to obtain a certain goal state. For example, if the robot wishes to turn on all the bits in the register, it should follow a link or sequence of links in the state transition graph machine that lead to the node in which all the bits are on.

Although one might try developing an algorithm to learn a state transition graph machine, Schapire (in ref. [87]) presents several arguments against it. [1] Most importantly, the state transition graph machine does not take advantage of all the regularities present in the environment. For example, in the n-bit shift register, the action F has the same behavior independently of the value of the unobservable bits (i.e., the bits other than the leftmost bit), yet in the state transition graph machine of Figure 1.3, knowledge about F must be separately encoded for each bit and in the context of the particular values of the other bits. Thus, the simple semantics of an action like F must be encoded repeatedly for each of the 2^n distinct states. The size of a state transition graph machine is insensitive to this kind of environmental regularity, and consequently, the running time of any learning algorithm that produces a localist machine is equally insensitive.

As an alternative, one could construct a *distributed* machine that utilizes fewer primitive machines than environmental states, where the representation of each environmental state is distributed across the state variables of these primitive machines. The hope is that the distributed machine would be smaller than the size of the state transition graph machine, that is, there would be fewer state variables and fewer connections needed. In the shift register environment, the bits of the shift register form a discrete-distributed state representation. A distributed machine using this state representation would have n state variables instead of the 2^n needed for the state transition graph machine. (Distributed machines are discussed in more detail below, in Chapter 3, and in the Appendix.)

One common approach (cf. Kohavi [44]) is to first construct the state transition graph machine, and using this graph, to then create a more compact distributed machine. Although the end product of this process is a smaller machine, the problem

[1] Schapire actually presents arguments against developing an algorithm to learn a state transition graph, but the same arguments apply to algorithms for learning state transition graph machines.

is that the learning time would still be a function of the size of the state transition graph machine instead of the size of the more compact distributed machine.[2]

A better approach—the approach taken in this thesis—is to learn a distributed machine directly. Rivest and Schapire [72, 74, 87] developed a discrete-distributed machine, called an *update graph*, that is a network of primitive FSA's. The update graph maintains a particular discrete-distributed state representation, called the *diversity representation*. (The details of the diversity representation and the update graph are presented in Chapter 3 and in the Appendix.) Rivest and Schapire developed a learning algorithm that explores the environment and incrementally constructs an update graph.

The third approach to constructing an environment model is the continuous-distributed approach. This approach is typified by traditional engineering methods, where the problem of constructing a model is called *system identification*. We consider one particular continuous-distributed machine from the engineering literature called a *state-space model*. Other engineering approaches are discussed in Chapter 2. In contrast to the primitive FSA machines used in the discrete approach, the primitive machines used in the traditional state-space model calculate the values of their state variables as real-valued linear functions of the selected action and the values of the state-variables of the machines to which they are connected. These functions are constructed using a set of real-valued parameters, which we called *weights*. The number and interconnectivity of these primitive machines is fixed a priori. Constructing a state-space model amounts to first choosing the number and interconnectivity of the primitive machines and then choosing a set of weights that gives the best prediction performance. This last step is called *parameter estimation* in the engineering literature. It differs significantly from Rivest and Schapire's symbolic learning algorithm

[2]The usual approach actually produces the state transition graph as the intermediate result, but the complexity arguments still apply.

which constructs a graph of primitive machines by incrementally adding new machines and their connections.

Our connectionist method, called *SLUG*, performs Subsymbolic Learning of Update Graphs. It is related to both the Rivest and Schapire approach and the traditional engineering approaches. We show that our connectionist method offers complementary strengths and new insights into the problem of building a model of a finite state environment. SLUG is a continuous-distributed method utilizing a connectionist network that shares many of the qualities of the engineering approach: it uses real-valued state variables, a fixed network topology, parameterized primitive machines, and parameter estimation to select weight values. In contrast to the traditional engineering approach, SLUG uses primitive machines, called *units*, that calculate the values on their state variables using a nonlinear function. We discuss the general connectionist approach in more detail in the Section 1.2.2, and then discuss SLUG in Section 1.2.3.

1.2.1 State Estimation and Real-World Issues

Before going into more detail, we first address a problem related to the construction of an environment model: the problem of setting a model's state to most closely correspond to the environmental current state. This problem is called *state estimation* in the engineering literature, and it is necessary because only part of the environmental state or a function of the environmental state is directly measurable and/or the available state information is noisy. The engineering approach views this as an optimization problem. Rivest and Schapire propose a different approach for solving this problem in the discrete case, and we discuss their technique in the Appendix.

It is important to point out that parameter and state estimation are particularly difficult for autonomous robots acting in the real world (cf. Kaelbling [38]), where the robot is continually receiving inputs and executing actions in its environment.

The robot must be able to keep up with the important events in the environment and thus cannot spend an unbounded amount of time in computing each action. Therefore, any algorithm must operate within a fixed time frame. In addition, an autonomous robot generally neither has access to a *reset* command, which places the environment in a known state, nor an *undo* command, which restores the state of the environment to the state existing prior to the execution of the last action. Thus, a state estimation algorithm must perform while the robot is continually interacting with its environment.

1.2.2 Connectionist Networks

The proposed connectionist approach differs from the traditional engineering approach by using a novel class of parameterized nonlinear network models. Connectionist networks are composed of simple differentiable nonlinear processing elements called *units*. The units are connected to each other and pass activity through weighted connections. Each unit computes its activity as a nonlinear function of a set of parameters, called *weights*, and its inputs: its output is transmitted to other units to which it is connected. The units, connections, and weights constitute a connectionist *network*.

Gaussian and logistic units are two popular unit types. A Gaussian unit computes its output as follows:

$$y = e^{\frac{-\sum_j (w_j - x_j)^2}{\sigma^2}}, \tag{1.1}$$

where y is the output of the unit, w_j is the jth weight of the unit, x_j is the jth input, and σ is the variance parameter. The logistic unit computes its output as follows:

$$y = \frac{1}{1 + e^{-\sum_j x_j w_j}}. \tag{1.2}$$

The activities of the *input units* serve as the inputs to the network, while the activities of the *output units* provide the output of the network. Units that are neither input nor output units are called *hidden units*. *Feedforward* networks are networks without

cycles in their connection graphs. *Recurrent* networks have cycles, that is, they contain feedback connections in addition to feedforward connections.

Connectionist networks have a number of attractive properties (as pointed out in ref. [6]) that make them appropriate for learning models of, and controllers for, nonlinear systems. They can be implemented in fast parallel hardware, gradient calculations useful for parameter estimation can be computed very efficiently with the back-propagation algorithm (developed by le Cun [48], Parker [65], Rumelhart, Hinton, and Williams [83], and Werbos [101]), and learning rules for connectionist networks are predominantly on-line algorithms. All these properties make networks attractive for real-time processing. Given enough units, networks can represent any function to any desired degree of accuracy (Hornik, Stinchcombe, and White [32]). Because of the statistical nature of connectionist learning rules, connectionist networks are able to learn in spite of noise in the examples, and they can also learn to approximate probabilistic functions in a principled fashion [31]. Furthermore, connectionist networks provide a variety of methods for encoding prior knowledge in the form of particular architectures, weight constraints, and initial weights. Finally, connectionist networks provide the flexibility to enhance the representation of the inputs without modifying the basic learning procedure. For example, the basic back-propagation learning algorithm applies to a large class of networks with various network architectures and unit types.

1.2.3 Our Modeling Approach

Our basic learning architecture is a recurrent network as shown in Figure 1.4. The network is composed of a fixed number of units, called *state units*, that serve as the state variables of the model. Some of these units are called *observable state units* and serve to predict the sensations. The remaining state units are called *hidden state units* and serve to encode state information useful for predicting the sensations and disambiguating perceptually ambiguous environmental states. The state units

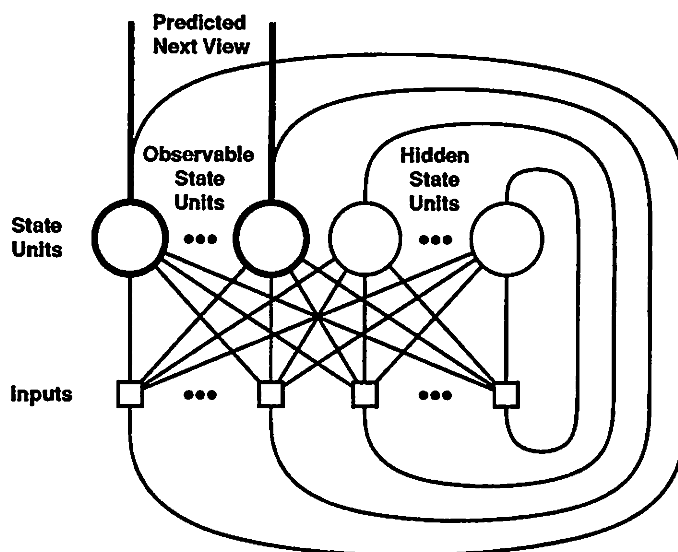

Figure 1.4. The SLUG Architecture. In this architecture there is one set of weights for each action. The weights are determined by the current action.

are fully connected to each other yielding a fully-connected recurrent network. Each unit has a separate set of weights for each of the robot's available actions, and the weights at any moment are determined by the chosen action. The outputs of the network are computed using a weighted sum involving the current weights (i.e., the weights for the chosen action) and the previous output of the state units. Although nonlinearities are introduced by the operation of choosing a set of weights based on the chosen action, the behavior of the network at any one time step is linear.

The technique called *back-propagation in time* [83] is used to modify the weights of the network to improve prediction performance. It is an algorithm for calculating the gradient of an error measure with respect to the weights of recurrent networks. In this case, the error measure is the squared error between the predicted and the actual sensation values.

In order to produce better state estimates, the actual instead of the predicted values of the sensations are used for computing the model's next state, where the sensations are assumed to be noise-free. In keeping with Williams and Zipser [106],

this technique is called *teacher forcing*. The combination of both back-propagation in time and teacher forcing provides a means for performing parameter and state estimation simultaneously.

A unique feature of this type of network architecture is that a special case of it, called SLUG, can learn an update graph realization of an FSA. An update graph has a natural and direct connectionist implementation. It has a number of beneficial properties that differentiate it from typical connectionist representations of FSA. After training, because the activities and weights are discrete, SLUG can perfectly predict the sensations infinitely far into the future. Additionally, the state units are readily interpretable within the update graph formalism. Many formal and empirical results exist (cf. Rivest and Schapire [73, 87]) that lend intuition and rigor.

Others have taken similar approaches to the modeling problem. Our approach is related to more traditional engineering approaches such as state-space and DARMA models. These relations are elaborated in Chapter 2. The idea of back-propagation in time has been used to model nonlinear dynamical systems [79]. Recently, a real-time variant of this procedure was developed and demonstrated by Bachrach [3], Mozer [55], Robinson and Fallside [76], and Williams and Zipser [106]. Rivest and Schapire [73, 87] explore a learning algorithm for inferring update graphs using a discrete-distributed representation (as opposed to the continuous-distributed representation used in this thesis). Giles et al. [26] discuss a bilinear recurrent network for learning regular languages. Whitehead and Ballard [104] discuss an alternative approach for dealing with the problem of perceptually ambiguous environmental states. In their formulation, they separate the perceptual state from the environmental state. For instance, the environmental state might be the position of the robot in its environment, and the perceptual state might be the orientation of the robot's sensors. An unrealistic requirement on their approach—one that does not hold in the environments that we consider—is that each environmental state must have at least one perceptual state that yields an unambiguous view, that is, a view unique to that environmental

state. Unlike their approach, our technique does not require this assumption and will work in environments where their technique fails. Chapter 3 expands on the relationship between the proposed approach and similar approaches taken by others. Mozer and Bachrach have reported on some of the work presented in this dissertation [57, 58, 60, 59, 5].

1.3 Control Acquisition

Having discussed methods for learning a model of the environment, we turn to a control acquisition architecture that can use this model to determine a minimal length sequence of actions to follow to reach a particular goal state. We call the particular control task addressed in this thesis the *homing task* (which is an example of what is called an *optimal regulation task* in the control literature). It involves producing actions that cause the environment state to change from an arbitrary starting state to a given goal state in minimum time. A technique is described for training a controller to perform the homing task. This training process is referred to as *control acquisition.*

The robot acquires the desired homing behavior using a learning paradigm called *reinforcement learning.* It is useful to distinguish this paradigm from another learning paradigm called *supervised learning.* In supervised learning, the robot is given desired actions, whereas in reinforcement learning, the robot is given only reinforcement signals, which do not directly specify the desired actions. In the latter case, the robot must estimate the desired action by comparing given reinforcement signals for various actions taken over time.

As an additional difficulty, the robot must solve a sequential decision task whose objective is to maximize a performance measure that evaluates its behavior over an extended period of time. Although the performance measure evaluates the efficiency of a homing trajectory, the robot is given only a signal that indicates when it is located at the goal. From this limited reinforcement signal, the robot must determine a minimal length sequence of actions necessary for homing. This poses a difficult

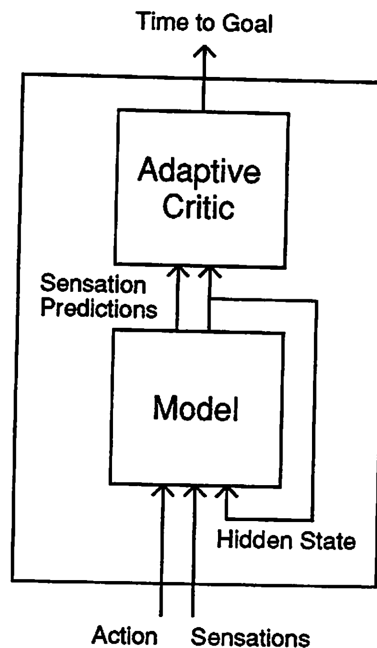

Figure 1.5. Connectionist Control Acquisition Architecture.

credit assignment problem because the consequences of the actions reveal themselves over time and interact with the consequences of other actions.

1.3.1 Partial State Information

In order to make the control acquisition task more realistic, we consider the impact of partial state information on control acquisition. As an example of the difficulty of partial state information, consider the problem of learning to go to the home square, labeled 2, in the 4×4 checkerboard environment shown in Figure 1.1. As before, the two squares A and B on opposite sides of square 2 yield exactly the same view. Without some mechanism to disambiguate these squares it is impossible to learn an optimal control rule for this environment.

To handle the problem of partial state information we propose a control acquisition architecture that uses the model described in Section 1.2.3 to determine a minimal sequence of actions required to reach a goal state. The architecture consists of a *model* and an *adaptive critic* as shown in Figure 1.5. The adaptive critic associates

the model's current state with the expected time to reach the goal state. At each time step, the action is chosen which minimizes the output of the adaptive critic. The adaptive critic is trained using a *temporal difference (TD)* method to make successive predictions of time to reach the goal state decrease by one and to make the prediction at the goal state zero.

The proposed control acquisition technique was motivated by the work of other researchers. Barto, Sutton, and Anderson [8] and Sutton [94, 95] developed the temporal difference method which is related to a technique invented by Samuel [85]. Sutton, Barto, and Anderson [8] and Barto [6] demonstrate a direct reinforcement learning technique involving the temporal difference method for training a controller. Watkins [99], Sutton [94], Barto [6], and Kaelbling [38] discuss reinforcement learning issues and methods. Whitehead and Ballard [104] discuss the partial state information problem in control acquisition and propose a mechanism to handle it. Werbos [102] developed a control network, called the *3-net architecture*, involving an adaptive critic, model, and controller which is most similar to the proposed control network. Jordan and Jacobs [36] and Werbos [102] discuss a related control network, called a *2-net architecture*, involving an adaptive critic and a controller. These and other related approaches are elaborated in Chapter 4.

1.4 Navigation

In chapter5, we describe simulations of a cylindrical robot with a sonar belt in a planar environment. The navigation task is short-range homing in the presence of obstacles. A control acquisition architecture is used for navigation which incorporates a large amount of a priori knowledge in the form of hard-wired networks, architectural constraints, and initial weights. This architecture, with critic and model, is basically the same as shown in Figure 1.5. This example demonstrates the use of a large modular architecture on a difficult task.

We study this task in order to consider the complexities of more realistic environments characterized by high-dimensional continuous state-spaces and real-valued actions and sensations. This portion of this dissertation does not address the problem of partial state information: we assume that all environmental states are perceptually unambiguous. We make this simplification in order to bring to the foreground the issues of control acquisition in high-dimensional continuous state-spaces. Of course, we would eventually like to merge the techniques developed in this portion of this dissertation with those developed to handle partial state information.

The navigation task requires the solution of a number of problems. The sensory information forms a very high-dimensional continuous space, and successful homing generally requires a nonlinear mapping from this space to the space of real-valued actions. Furthermore, training networks with logistic hidden units is not easy in this problem. Instead we use networks with Gaussian hidden units, but because the state space is high-dimensional and continuous it is impractical to just uniformly distribute the receptive fields of these Gaussian units throughout the state space. Instead we use Gaussian units whose initial weights (and thus receptive fields) are determined using expectation maximization. This is a soft form of competitive learning (cf. Nowlan [63] and Durbin [19]) that, in our case, creates spatially-tuned units.

Unlike many approaches to robot navigation, our approach assumes no prior map of the environment. The robot has to use its limited sensory information to construct a model of its environment. Given this sensory information, models of the robot's environment are much more difficult to learn than models for the FSA environments discussed in Section 1.1. For this reason we use a hard-wired model whose performance is good in a wide range of environments. Here the philosophy is to learn only things that are difficult to hard-wire.[3] Finally, it is difficult to reach home using random exploration, thereby making simple trial-and-error learning intractable.

[3]Note that, in the future, when we consider even more realistic environments involving perceptual ambiguity and/or realistic sensors, it will be necessary to construct models.

The solution to this problem involves building a nominal initial controller that chooses straight-line trajectories home.

Our approach is related to other navigation research as follows. The basic approach is strongly related to our control acquisition approach and related research (see Section 1.3). Our approach is also related to a more traditional technique for navigation that uses potential functions (see Latombe [47] for an overview). Briefly, our approach differs from the usual potential function technique in two fundamental ways. First, our technique involves learning and can adapt the robot's behavior to produce efficient trajectories. Second, our technique does not require explicit object models; it constructs representations directly from the sensory information. Our approach is similar to the subsumption architecture proposed by Brooks [13]. Our approach and that of Brooks both involve the orchestration of multiple behaviors through the use of multiple experts, but in contrast to the subsumption architecture, ours involves learning and continuous mappings constructed with connectionist networks. Recent work by Mahadevan and Connell [51] and Maes and Brooks [50] propose preliminary learning techniques for the subsumption architecture. See Chapter 5 for an expanded discussion of related research.

1.5 Organization of the Dissertation

This dissertation elaborates methods and presents empirical results concerning modeling and control. Chapter 2 describes related research by others addressing the issues of modeling FSA environments with partial state information. Chapter 3 presents our approach to modeling FSA environments with partial state information. Chapter 4 presents a method of control acquisition for these environments. Chapter 5 describes an approach to learning to navigate in a simulated environment. Chapter 6 summarizes the research presented in this dissertation, briefly discussing the contributions, problems, and directions for future research. The Appendix discusses the update graph in more detail.

CHAPTER 2
PREVIOUS MODELING RESEARCH

This review covers system-identification methods for modeling systems with incomplete state information, including techniques for both state estimation and parameter estimation. This chapter is organized into three different sections representing approaches from Linear Systems Theory, Connectionism, and FSA Identification, where FSA Identification includes research from automata theory, general systems theory, and computational learning theory. Before we discuss these three different approaches, we first introduce some preliminary concepts.

For the purposes of this review it is important to relate the problem of learning a model to the problem of training a machine to recognize a particular language. Many researchers study this latter problem, and it turns out that there exists an isomorphism between these two problems in the single sensation, discrete-time, finite-state case. Let actions be letters in the alphabet, action sequences be strings, and sensations be the output of the recognizer. Let the output of the recognizer be called the *accept signal.* The model can be designed to answer the question of whether a given string is in the language. In order to answer this question, the model is first set to an initial state, and a string is presented to the model as a particular action sequence. The accept signal is true if the input string is in the language and false otherwise. For every language there exists a model that accepts only strings in that language, or equivalently, outputs a true accept signal when a string is in the language. A particularly relevant class of languages, called *regular languages*, is the class of languages that can be recognized by FSA [44].

It is useful to categorize the modeling techniques according to the state representation employed (see Chapter 1 for more details). In keeping with the engineering terminology, the environment will be called the *system*. When a function from current sensations and current action to next sensations cannot accurately model a system, then it is necessary to represent state information of the system. A state representation captures the salient aspects of the unbounded history of the system's behavior necessary to predict the system's future behavior. Two canonical representations are presented that correspond to the extremes of the representational spectrum. These are the delay-coordinate representation and the state-space representation. Each succinctly represents a system's past behavior.

In the delay-coordinate representation, the system's state is encoded as a fixed number of past inputs and outputs of the system. In particular,

$$\hat{\boldsymbol{y}}(k) = h(\boldsymbol{u}(k), \boldsymbol{u}(k-1), \ldots, \boldsymbol{u}(k-D_u), \boldsymbol{y}(k-1), \boldsymbol{y}(k-2), \ldots, \boldsymbol{y}(k-D_y)). \quad (2.1)$$

where $\boldsymbol{u}(i)$ are the actions chosen at time i for $i = k, k-1, \ldots, k-D_u$, h is the output function. $\hat{\boldsymbol{y}}(k)$ is the prediction of the model at time k, $\boldsymbol{y}(i)$ is the output of the system at time i for $i = k-1, k-2, \ldots, k-D_y$, $D_u > 0$, and $D_y > 0$.

There are FSA's that cannot be accurately modeled using a delay-coordinate state representation. Figure 2.1 shows an example of such a FSA. This FSA cannot be accurately modeled using a delay-coordinate state representation because knowledge that the last D_u actions were β and the last D_y sensations were 0 is not helpful in predicting the response to an applied α regardless of the values of D_u and D_y. The state into which the machine passes after the application of the arbitrarily long sequence of $\alpha\beta\beta\beta\ldots\beta$ is uniquely determined by the response to the first symbol included in this sequence.

In the state-space model, the system's state is encoded over a fixed number of state variables. In particular,

$$\boldsymbol{x}(k) = g(\boldsymbol{x}(k-1), \boldsymbol{u}(k-1)) \quad (2.2)$$

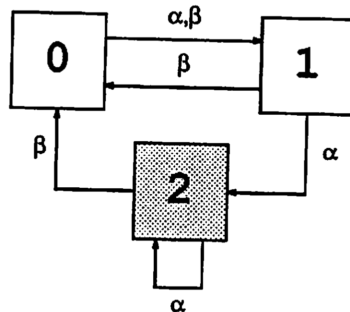

Figure 2.1. An Example of a FSA that Cannot Be Accurately Modeled with a Delay-Coordinate State Representation. This particular FSA has 3 states labeled 0, 1, and 2 and 2 input actions α and β. The shade of the each box codes the single sensation value for that state, where a shaded square codes a sensation value of 1, and a light square codes a sensation value of 0.

$$\hat{\boldsymbol{y}}(k) = f(\boldsymbol{x}(k)), \tag{2.3}$$

where $\boldsymbol{x}(k)$ is a vector of the values of the state variables at time k, and g is the next state function. Both the delay-coordinate and state-space model techniques can be extended to handle the case where the system is stochastic.

The delay-coordinate representation can always be transformed into a state-space model by constructing the state representation as a vector of delayed values of system inputs and outputs. The g function can be represented as a linear operator using a matrix which shifts the state variables in the same fashion as the tapped delay line. Each state variable receives its next value from one other variable or from the system input or output.

In contrast, a state-space model cannot always be transformed into a delay-coordinate representation. Although a state-state model can be transformed into a delay-coordinate representation when f and g are linear, when f and g are nonlinear, then this is not always possible. Intuitively this is true because the state-space model can contain hidden state variables that keep their initial values indefinitely, whereas a delay-coordinate model has no such mechanism (see Gill [27] for more details). The

"state variables" in the delay-coordinate model are past values of either the system's input or output.

The state-space and delay-coordinate representations occur in many different disciplines. They will be used to describe various techniques in the following sections. Finally, the SLUG architecture will be described within these frameworks.

In the state-space model there is a range of state representations, that is, encodings of state in terms of values of state variables. The two extremes are the localist state representation and distributed state representations. In the research to be reviewed there is a strong division between researchers that favor the localist state representation and those that employ the distributed state representation. For the most part, the engineering and connectionist approaches utilize distributed state representations, and in automata theory, general systems theory, and computational learning theory the localist state representation predominates. This results from the fact that the FSA identification and computational learning theory approaches typically infer the underlying FSA utilizing one state variable (or datum) per state, while the engineering and connectionist approaches usually just restrict representations to a finite number of state variables.

2.1 Techniques Based on Linear System Theory

Most of the classical system-identification techniques deal exclusively with linear dynamical systems, but for completeness, the basic ideas are reviewed. These approaches typically assume a certain amount of a priori knowledge about the system to be identified. This knowledge is converted into the choice of model inputs and outputs and constraints on the mathematical form of the model. In order to identify nonlinear systems, traditional approaches typically involve a model that is a linear function of handcrafted feature vectors that code nonlinear functions of the sensations. This permits the use of parameter and state estimation schemes for models that are

linear in the parameters, which are generally well-behaved with many mathematical results providing guidelines for their use.

Examples of linear techniques that use the delay-coordinate representation are adaptive filters [105] and DARMA (Deterministic Auto Regressive Moving Average) models [30]. This formalism can be generalized to nonlinear models, which will be discussed in Section 2.2. Adaptive filters have tunable parameters, and the state of the system is represented in delay coordinates as a buffer of past inputs and outputs, called a *tapped delay line*. The simplest form of a filter, called a Finite Impulse Response (FIR) filter, is defined as:

$$\hat{y}_j(k) = \sum_{i=0}^{D_u} b_{ji} u_j(k-i). \tag{2.4}$$

where $\hat{y}_j(k)$ is the jth output of the filter at time k, $u_j(k-i)$ is the jth input to the filter at time $k-i$, b_{ji} is a parameter, and D_u is one less than the size of the tapped delay line. Widrow and Stearns [105] describe an on-line learning rule, called LMS (least mean squared), that finds weights that minimize the expected mean squared error between the output of the filter and its desired output. Back-propagation generalizes the LMS algorithm.

A more complicated filter is an adaptive recursive filter, also known as an Infinite Impulse Response (IIR) filter, which can be described as follows:

$$\hat{y}_j(k) = \sum_{i=0}^{D_u} b_{ji} u_j(k-i) + \sum_{i=1}^{D_y} a_{ji} \hat{y}_j(k-i), \tag{2.5}$$

where a_{ji} is a parameter. These filters are more powerful than FIR filters because they also include the filter's past outputs in their tapped delay lines. Again, Widrow and Stearns describe a learning algorithm that minimizes expected mean squared error, but in this case they show that the algorithm does not necessarily find the globally optimal solution.

The DARMA model is the same as an IIR adaptive filter and the same learning algorithms apply. DARMA models with enough tapped delay lines can represent

any linear dynamical system to given accuracy [30]. Finally, DARMA models can be extended to the stochastic case and learning algorithms can be derived [30].

State-space models employ a state vector x instead of the fixed delay coordinate state representation. One such model is called the *stochastic linear state-space model.* Consider a system described by:

$$\boldsymbol{x}(k) = \Phi(k-1)\boldsymbol{x}(k-1) + \Lambda(k-1)\boldsymbol{u}(k-1) + \boldsymbol{\nu}(k-1) \tag{2.6}$$

$$\boldsymbol{y}(k) = H(k)\boldsymbol{x}(k) + \boldsymbol{\omega}(k), \tag{2.7}$$

where $\boldsymbol{x}(k)$ is the vector representing the state of the system at time k, $\boldsymbol{u}(k)$ is the vector input to the system at time k, $\boldsymbol{y}(k)$ is the vector output by the system at time k, matrices $\Phi(k)$, $\Lambda(k)$, and $H(k)$ are matrices whose entries are parameters at time k, and $\{\boldsymbol{\nu}(k)\}$ and $\{\boldsymbol{\omega}(k)\}$ are zero mean stationary white noise processes with covariance of a certain form (see Goodwin and Sin [30] for more details).

State estimation is necessary with state-space models because the state cannot always be read directly from the environment. *Kalman filtering* is one procedure for estimating the state of a stochastic linear state-space model. In Kalman filtering, the parameters of the state-space model are assumed to be known (i.e., Φ, Λ, and H are fixed and known a priori), and only the state of the model is unknown. (For more details consult Goodwin and Sin [30] and Elbert [20].) A Kalman Filter provides a way of updating the state estimates after the occurrence of the observables $\boldsymbol{y}(k)$ so as to minimize the squared error between the actual state and the estimated state of the system. The Kalman filter is computationally tractable because the internal variables can be computed recursively and some of them can be precomputed when appropriate. The algorithm is much less computationally intensive than other approaches which usually involve large matrix inversions.

Kalman filtering is a powerful technique and can be generalized in a number of ways. The *extended Kalman filter* is a technique for estimating the state of a nonlinear stochastic dynamical system. The idea is to employ normal Kalman filtering on the

system linearized around the current state estimate. (See Elbert [20], Ljung and Soderstrom [49], or Goodwin and Sin [30] for the details.)

Techniques exist for handling both parameter estimation and state estimation, that is, for estimating the values of the Φ, Λ, and H matrices as well as the value of $\boldsymbol{x}$. One technique involves creating a more complicated state-space model where the state and parameters of the original model are now the state of the new model. The addition of the parameters makes the new state-space model nonlinear and thus requires the extended Kalman filter. Another technique for performing both parameter and state estimation involves switching between standard parameter estimation and Kalman filtering. Initially, the Kalman filter produces the estimated state, $\boldsymbol{x}_0$, for the initial parameters, $\boldsymbol{\theta}_0$. This state is then input to the parameter estimator which produces new parameters, $\boldsymbol{\theta}_1$. These are then input to the Kalman filter resulting in $\boldsymbol{x}_1$. This process is repeated until some criterion is met.

SLUG can be related to the linear state-space and delay-coordinate representations presented in this section. SLUG can be cast into the state-space framework using a time-varying state-transition matrix as follows:

$$\boldsymbol{x}(k) = \Phi(\boldsymbol{u}(k-1))\boldsymbol{x}(k-1) \tag{2.8}$$

$$\hat{\boldsymbol{y}}(k) = H\boldsymbol{x}(k), \tag{2.9}$$

where $\Phi(\boldsymbol{u}(k-1))$ is the $n \times n$ transition matrix corresponding to the input to the system, or equivalently, to the action chosen at time step $k-1$, and H is an $m \times n$ projection matrix: $H = [I|0]$, where I is the $m \times m$ identity matrix.

An interesting special case of SLUG, called the update graph, can be viewed as a generalized tapped delay line (see Chapters 1 and 3 for more details about SLUG, and see Chapter 3 and the Appendix for more details about the update graph formalism). Like the tapped delay line, at each time step, each of the update graph's state variables receives its value from only one other state variable. This state update behavior is determined by the action taken at that time step and can be described by the state

transition matrix Φ. In the case of the update graph, Φ has a special form where in each row there is exactly one element containing a 1 and there are zeroes in the remaining elements. The element Φ_{ij} containing a 1 effectively selects the appropriate state variable j from which to update a given state variable i.

2.2 Connectionist Modeling Methods

In order to mimic a dynamical system, a network must represent the state of the system. Two approaches to this problem are presented: networks that have a delay-coordinate state representation and networks that have a state-space representation. Two different connectionist architectures with delay-coordinate state representations are presented. These architectures are different from the traditional engineering designs because the output of the model is a nonlinear function of this state representation. Then a series of architectures and learning algorithms for learning state-space models are presented. All of the learning techniques presented are gradient descent algorithms; the weights are adjusted in proportion to the gradient of the performance criterion with respect to the weights. The algorithms differ in the accuracy of the approximation to this gradient.

2.2.1 Delay-Coordinate Models

McClelland and Elman present a connectionist architecture for speech recognition called the Trace Model [53]. Their network architecture is based on the interactive processing idea championed in the earlier reading model [54, 81], where the activations of the units are updated by both a bottom-up and top-down process. This architecture is extended to include a tapped delay line filled with all the speech data from the beginning of an utterance. In their system there must be a delay line large enough to accommodate the longest utterance. Also note that this is not a learning system. In order to overcome the deficiencies of the basic delay-coordinate approach, feature detectors are duplicated at each temporal position to get translation invariance,

and feature detectors are "fuzzed out" to achieve scale invariance. Sejnowski and Rosenberg [89] present a similar system, called NetTalk, that learns to translate text to phonemes. Their architecture involves a feed-forward network with a large tapped delay line for the text and uses back-propagation to adjust the weights.

Waibel et. al [98] introduce an architecture, called a Time-Delay Neural Network (TDNN), that is used to solve a speech recognition learning task. It is designed to cope with temporal relationships of events, and the features it learns are invariant under time translations. The architecture is a tapped delay line design where each unit receives a moving window of past values of the outputs of units feeding into it. They improve on the adaptive filter method by allowing feed-forward networks of these tapped delay line units (e.g., the output units receive a moving window of past values of the inputs to the network as well as the outputs of the hidden units). Translation invariance is achieved by constraining the weights to be the same at each of the delays. Unfortunately, the "Time Delay" approach can only represent that an event occurred a fixed number of steps ago within its temporal window. Because this representation is fixed, it fails when the time scale of the speech changes because the spacing between speech events differs.

2.2.2 State-Space Models

Rumelhart, Hinton, and Williams [79] present an extension to back-propagation that allows the training of arbitrary recurrent networks. In turn, these recurrent networks can be used to model nonlinear dynamical systems. The back-propagation algorithm can only be *directly* applied to nonrecurrent networks. It requires an unfolding procedure to handle recurrent networks (see Rumelhart, Hinton, and Williams [79] for a full account). Unfolding a network turns any recurrent network into a feed-forward network by making copies of each unit for each discrete time step. The recurrent links connecting unit α to unit β are then converted to feed-forward connections by terminating the link on a copy of unit β at the next time step. Figure 2.2

shows an example of this procedure applied to a fully recurrent network consisting of two units. The resulting unfolded network is now a feed-forward network where the back-propagation procedure can be applied. This procedure is called *back-propagation in time* and is the learning algorithm used in this thesis (see Chapter 3 for more details).

Back-propagation computes the gradient of the performance criterion with respect to the weights; back-propagation in time permits the choice of the desired accuracy of the approximation to this gradient. If the network is unfolded once for every time step in the simulation, then back-propagation in time will compute the exact gradient. Unfortunately, because the network can only be unfolded a finite number of times, this technique will only apply to a limited class of learning tasks: tasks in which the sequence of training examples can be divided into finite sequences between which the network is reset to a certain initial state. Grammatical inference problems involving strings of bounded length are examples of problems where this training regimen is feasible. For problems where the network is continuously running and there is no logical boundary between sequences of training examples, the particular number of unfoldings must be chosen: the larger the number of unfoldings, the better the approximation to the true gradient.

Watrous and Shastri [100] use a special architecture that uses back-propagation in time to learn to discriminate between two words based solely on time-varying spectral data. They use a network architecture, called "temporal flow", with an input, hidden, and output layer. The input layer is fully connected to the hidden layer, which is in turn fully connected to the output layer. Both the hidden and output layers have self-recurrent connections on each of their units (i.e., a feedback connection from a unit to itself). The self-recurrent connections perform temporal integration of the speech and the representations built by the network. Notice that this architecture cannot represent every possible FSA because of the simplicity of the feedback connections.

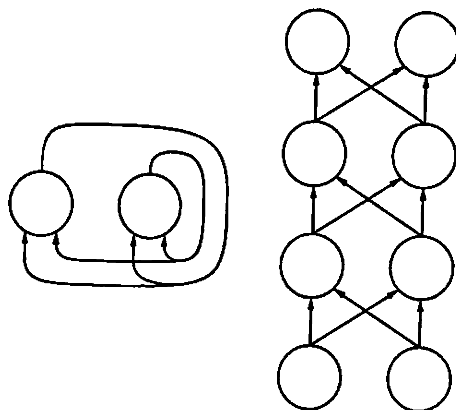

Figure 2.2. An Example of the Unfolding Procedure which Turns Recurrent Networks into Feed-Forward Networks.

Pearlmutter [66] presents a continuous time generalization of back-propagation in time. Like back-propagation in time. his algorithm can learn trajectories through state space and requires that the network be unfolded in time during training. His work differs from back-propagation in time in that his networks are defined in terms of differential equations.

Bachrach [3, 4] report on methods for training simple reverberatory circuits to act as memory devices. Specifically, Bachrach examines a connectionist unit, called a *sticky-bit*, that excites itself through a recurrent connection and can be "set" or "reset" like a SR flipflop. This kind of memory is different from the kind of long-term memory that is stored in connection weights. The problem is to learn *when* to set or reset these bits in a variety of paradigms. This knowledge would be stored in connection weights. My algorithm is real-time, computes the exact gradient without unfolding the network, and allows for arbitrarily old inputs to influence weight changes. Bachrach uses fixed feedback weights and a symmetric logistic output function (ranging from -1 to $+1$). This overcomes some of the limitations of single unit feedback systems discussed in Watrous and Shastri. The effective search effort in weight space is reduced because each unit starts out with

two well-behaved symmetric attractors. Without more complicated machinery the sticky-bits are limited in their representational power. In particular, the combination of sticky-bits and a feed-forward network cannot represent arbitrary FSA; more feedback connections are necessary.

Mozer [55] derived a similar real-time back-propagation rule for self-recurrent units. He uses a variation of the sticky-bit learning algorithm using units with tapped delay lines and outputs that are exponentially decaying traces of the usual logistic outputs. A real-time gradient algorithm is derived for this output function. This algorithm is used to learn "wicklefeatures" similar to the handcrafted input features given to the verb past tense model of Rumelhart and McClelland [82]. The main drawback with this approach is that Mozer's units necessarily have limited memory because eventually their outputs decay away. In contrast the sticky-bit can latch onto an event indefinitely.

Williams and Zipser [106], Bachrach [3], Robinson and Fallside [76], and others developed a real-time learning algorithm for recurrent networks that computes the exact gradient without unfolding. Their algorithm applies to a continuously running arbitrarily connected network. The network is continuously running since each unit samples its inputs on every weight update cycle and a unit can have a teaching signal at any cycle. Each unit can be connected to any other unit in the network and any unit can receive input from the environment. The algorithm is nonlocal because it keeps a derivative of every weight with respect to the output of every unit. On the positive side, the storage requirements are determined solely by the size and connectivity of the network and are independent of time. Unlike back-propagation in time, the distance in time to back-propagate error does not have to be specified. Unfortunately, the storage size is on the order of the third power of the number of units and the computation time is on the order of the fourth power of the number of units. Thus, the exact gradient is computed at a great computational cost.

Giles, Sun, Chen, Lee, and Chen present a higher-order recurrent network for grammatical inference [26]. The network is a higher-order design but differs from SLUG in a number of respects. Instead of having the input selects a linear function, the input selects a particular semi-linear functions. In particular, their network computes its output as follows:

$$\boldsymbol{x}(k) = \boldsymbol{g}(W_{a(k)}\boldsymbol{x}(k-1)), \tag{2.10}$$

where g is the logistic function. The output of their network is interpreted as the acceptance or rejection of the input string instead of the prediction of the sensations at the next time step. Their learning algorithm resets the state of the network before each string is presented and only updates the weights after each string is presented when the teacher declares whether the string is in the language. Finally, state estimation is not performed by their algorithm.

Jordan [34] investigates coarticulation and sequence learning in fixed feedback back-propagation networks. Jordan's network architecture is shown in Figure 2.3. The network has one output layer and one hidden layer with feed-forward connections from the plan units and context units to the hidden units, and from the hidden units to the output units. There are feedback connections from the output units to the context units, where each output unit is connected to one context unit and there are the same number of context units as output units. There are also full feedback connections from the context units to themselves. (Notice that the diagram only shows the self-recurrent connections.) The weights on the feedback connections from output units to context units and from context units to context units are fixed. Jordan's learning algorithm approximates the gradient with standard back-propagation through the feed-forward portion of the network. This approximation ignores the contribution of the recurrent connections. This amounts to performing back-propagation in time with no unfoldings.

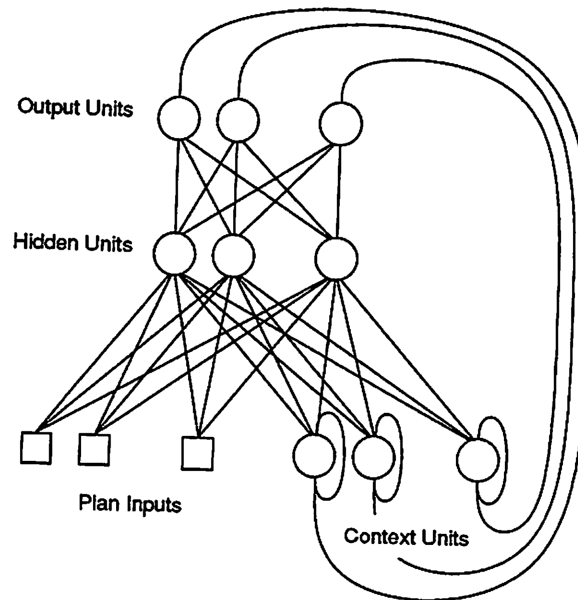

Figure 2.3. The Jordan Feedback Architecture.

Jordan considers problems requiring the learning of sequences. The outputs of context units form exponentially decaying traces of the outputs of the network; the network learns sequences specified by the plan units by learning to produce different outputs for different contexts. Under this formalism Jordan's system can solve learning tasks such as learning to model up/down counters and more impressive tasks involving coarticulation. Jordan's class of tasks is slightly different than tasks in this thesis. Jordan focuses on problems in which the inputs are fixed while the network generates a sequence, whereas the focus of this thesis is on problems where the inputs change over time. Unlike SLUG there are no hidden state variables in Jordan's architecture. This severely limits the representational power of this architecture. In particular, this architecture cannot implement all FSA.

Elman's [21] recurrent network, shown in Figure 2.4, is based on the Jordan's sequential network. It differs in that the only feedback connections are from the hidden units to the context units; there are no feedback connections from the context units to themselves. By using the nonlinearity in the hidden units, this architecture can

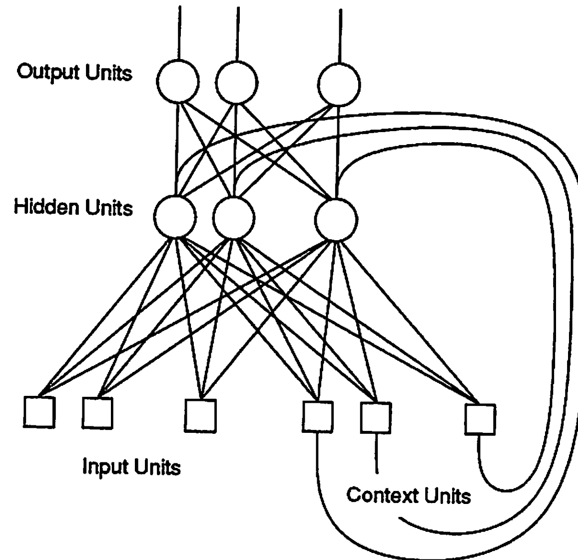

Figure 2.4. The Elman Feedback Architecture.

represent all FSA. Like Jordan's learning algorithm, Elman's learning algorithm also approximates the gradient by back-propagating only through the feed-forward section of the network. This makes learning in this architecture very slow and unreliable for difficult tasks. Servan-Schreibner, Cleereman, and McClelland [90] apply the Elman network to the problem of learning a regular language. Unfortunately, their technique takes on the order of 200,000 time steps to learn a simple language.

2.2.3 Relation of SLUG to Previous Connectionist Architectures

SLUG differs from the other architectures in three main respects. First, SLUG is a time-varying linear system, and important techniques from linear systems theory apply. Second, a special case of the network architecture encourages the discovery of a certain symbolic structure called an update graph. Within the update graph formalism the state units are readily interpretable, and many empirical results exist that lend intuition and rigor. Third, a fair amount of a priori knowledge about the relationship between actions can be encoded as weight constraints. The explicit

representation of the effects of actions on the internal state as separate weight matrices greatly facilitates this process.

SLUG most resembles the higher-order architecture proposed by Giles et. al [26]. It differs from their work in that: (1) the emphasis is on modeling and control of dynamical systems instead of grammatical inference, and (2) the training procedure and architecture differ.

2.3 Finite State Automaton Identification

This section describes a variety of techniques for learning FSA models of environments. These approaches are meant as a small sampling of the large number of techniques from automata theory, general systems theory, and computational learning theory. A common property of the approaches from automata theory and general systems theory is poor computational complexity in both space and time. The first three approaches involve exhaustive search through the space of FSA's. The majority of the techniques presented in this section learn state transition graphs. The last two approaches from general systems theory are more computationally practical and involve delay-coordinate representations.

In this section a few generalizations of the FSA model are presented. When the state transition mapping is one-to-many then the FSA is called a *non-deterministic FSA* or *NFSA*. When the state transition mapping is stochastic then the FSA is called a *probabilistic FSA*. Some of the techniques to be presented produce models from these classes.

2.3.1 Automata Theory

Kohavi [44] describes a technique for identifying FSA. Various restrictions are placed on the machine to be identified as follows: the set of possible inputs must be known; the upper bound, n, on the number of states of the machine must be known; the machine is assumed to be reduced, that is, it has the minimal number

of states; the machine is assumed to be strongly connected, that is, every state can be reached from every other state. The technique involves constructing the state and output tables for all machines having n or fewer states. The tables are concatenated together in a particular way to form a new table called the *direct-sum table*. A homing sequence is constructed for the direct-sum table, which identifies the final state of the machine and, in turn, the machine itself (see Kohavi [44] for details). A homing sequence is a sequence of input symbols which uniquely identifies the state reached by the automaton at the end of the homing sequence. This technique is obviously impractical because the number of machines is staggering even for small n.

Booth [11] also describes an algorithm for inferring minimal FSA. The same restrictions are placed on the machine to be identified as Kohavi. The basic algorithm can be summarized as follows: observe the response of the machine to a particular input sequence; form all the partial state diagrams with at most n states that could reproduce the exhibited behavior; apply another input sequence and use the behavior of the machine under investigation to extend or eliminate the proposed partial state diagrams; continue this process until obtaining a complete state diagram that describes the machine under investigation. It is important to make a number of observations about the algorithm. Because of the finite state property the algorithm always terminates. At first the input sequences are chosen at random and then later input sequences are chosen so as to eliminate one or more of the partial state diagrams and to determine the unknown transitions associated with one or more of the states. Finally, the algorithm is computationally inefficient because at any point in the modeling process there could be an exponential number of machines with at most n states that are consistent with the input sequences.

2.3.2 General Systems Theory

Gaines [24] describes a technique, called *ATOM*, for inferring probabilistic FSA. The technique involves an exhaustive search of NFSA starting with 1-state machines.

The output of the procedure is the list of admissible 1-state automata, the list of admissible 2-state automata, etc. where an admissible automata is one that accepts the strings in the language. The search ends when no more admissible models are found or more often after a predetermined amount of time. A non-deterministic model is converted to a probabilistic model by filling in the transition probabilities from the relative frequencies of the transitions from the training information. The algorithm is impractical because the search time grows exponentially with the number of states.

Witten [109] presents a technique for learning NFSA. The technique employs the delay-coordinate representation of state. In Witten's formulation there is one observed symbol per time step. The algorithm constructs length-k models by constructing states corresponding to the overlapping $k-1$ tuples occurring in the training sequence. A length-k model will not generate strings of length less than k that have not occurred in the training sequence. The parameter k corresponds to the length D of the delay-coordinate representation and provides a nice tradeoff between model simplicity defined in terms of state count and the goodness of fit. The first model is formed by creating all overlapping $k-1$ tuple states and then marking transitions between states which occur consecutively. The model can then be reduced by coalescing states which have the same output and the same successor state. An algorithm is presented for incrementally updating the model when a counter-example is discovered. The resulting model will not necessarily be the smallest NFSA, but no minimization procedure exists for non-deterministic models.

Klir [41, 42] describes a technique for learning a probabilistic FSA. His technique employs the delay-coordinate representation of state. A matrix, called the *activity matrix*, is constructed with the values of all observable variables over a segment of D past time steps. The states are defined to be possible values of the activity matrix. A procedure is described for estimating the state transition probabilities. An algorithm is proposed for choosing the best size for the activity matrix based on the reduction of uncertainty the delayed variables provide. Finally techniques are

suggested for reducing the complexity of the resulting model through the removal of states and transitions. The algorithm involves estimating a large number of transition probabilities and computational complexity grows exponentially with the size of the sampling mask.

2.3.3 Computational Learning Theory

Computational learning theorists are primarily interested in the algorithmic properties of learning tasks, and a goal of these researchers is to find computationally efficient algorithms. They have studied the modeling problem considered in this thesis under the guise of inferring a regular language or, equivalently, inferring the structure of an FSA. They have shown that this problem proves to be both interesting and difficult.

Consider the problem of inferring an FSA consistent with a finite set of input/output pairs. This learning paradigm is called the *passive learning protocol.* Angluin [1] and Gold [29] have shown that the problem of inferring a minimal FSA using the passive learning protocol is NP-complete. More recently, other researchers have shown that with the passive learning protocol even finding an approximate solution is intractable [67], where an approximate solution is a solution that has a particular probability of being a correct one. Finally, it has been shown that these results are independent of the representation used for the FSA [39].

These results have led researchers to search beyond passive learning for learning protocols involving *active experimentation.* A number of researchers [28, 2, 87] have developed polynomial-time learning algorithms involving active experimentation. Angluin [2] characterizes the experimentation as queries. Two types of queries are proposed: *membership* and *equivalence* queries. The membership query asks an oracle whether a given string is accepted by the FSA. The equivalence query asks an oracle whether the inferred FSA is equivalent to the correct FSA and if not, the oracle responds with a counter-example string. Angluin's inference procedure runs in

time polynomial in the automaton's size and the length of the longest minimal length counterexample.

Rivest and Schapire [73, 87] also obtain a polynomial-time algorithm through the technique of active experimentation. In contrast to Angluin, Gold, and others, they are interested not in the task of inferring regular languages, but in the modeling problem stated earlier. Unlike the grammatical inference task, they assume no ability to reset the FSA to a known state, nor the ability to undo actions. Their inference procedure plans experiments that test the equivalence of state variables, where the values of all the state variables encode the state. New state variables are created when a proposed state variable is found to be inequivalent to all other state variables. The number of these state variables is called the *diversity* of the environment, and their state representation is called a *diversity representation*. The basic result is that the algorithm is polynomial in the diversity of the environment instead of the number of states of the environment. They claim that for many "natural" environments the diversity is much smaller than the number of states. Their polynomial-time result applies only to a restricted class of FSA's with the permutation property: FSA's in which every action sequence, a, has a fixed inverse, a^{-1}, that, when executed, returns the FSA to the state it was in before executing the original action sequence. They also present a heuristic algorithm that applies to all environments. Their work is presented in detail in the Appendix.

Rivest and Schapire [75] extend Angluin's algorithm to the case where the algorithm has no means of resetting the machine to the start state. They describe an algorithm that with a probability $1-\delta$ learns an arbitrary FSA in time polynomial in the automaton's size, the length of the longest counterexample, and $\log(1/\delta)$. They also present a similar algorithm that learns a diversity representation of the FSA. Rivest and Schapire use new techniques based on *homing sequences* to overcome the absence of the ability to reset the automaton. The state is identified by the sequence of FSA outputs produced during the execution of the homing sequence. They improve

the bounds of their earlier work for learning diversity representations for permutation environments by a factor of $D^3/\log D$, where D is the diversity.

Porat and Feldman [70] present a provably correct algorithm for inferring a FSA from lexicographically ordered examples. The basic algorithm constructs a FSA by iteratively examining the lexicographically ordered positive and negative examples. At each step of the way, the algorithm ensures the consistency of the most recent example with respect to the latest FSA. If the new example is inconsistent, then either state transitions are deleted or a new state is added. After a new state is added all the previous examples must be checked for consistency with the new FSA. They show that their learning algorithm runs in time polynomial in the number of states. Their algorithm has modest space requirements because of the ordering of the examples; the algorithm can remember previous examples just by keeping track of the current example. Porat and Feldman argue that their algorithm is amenable to connectionist networks because of its polynomial space requirements. Unfortunately, unlike typical connectionist algorithms, the resulting algorithm exhibits only a marginal amount of parallelism, cannot handle any noise, and requires a great deal of global synchronization and delegation.

2.3.4 Relation of SLUG to Finite State Automaton Identification

A number of themes distinguish the techniques from automata theory, general system's theory, and computational learning theory from SLUG. First, except for the work of Rivest and Schapire, the models produced by these techniques employ a discrete-localist state representation. In comparison SLUG utilizes a continuous-distributed state representation. Second, most of the models produced by these techniques are not functional until after learning is complete. In contrast SLUG can be used for prediction throughout training. Third, with the exception of the work of Rivest and Schapire, these models do not incorporate a mechanism for state

estimation. Instead they rely on the availability of a reset mechanism which places the system into a specified initial state. Finally, the learning techniques reviewed are primarily serial algorithms, that is they consider and update only one hypothesis at each time step. In contrast, SLUG permits a large degree of parallelism because it considers and updates many hypotheses in parallel at each time step.

2.4 Conclusion

Relevant modeling research by others on similar problems was presented in this chapter. A variety of research was covered including work from engineering, connectionism, automata theory, general systems theory, and computational learning theory. SLUG has been inspired by these various approaches: SLUG is a hybrid approach, borrowing techniques from both engineering and FSA identification.

Chapter 3

MODELING FINITE STATE AUTOMATA

The architecture of our connectionist network model is based on a representation of finite-state environments developed by Rivest and Schapire [72, 74, 87]. Rivest and Schapire have also developed a symbolic algorithm to infer this representation by exploring the environment. We have taken a connectionist approach to the same problem, and we show that the connectionist approach offers complementary strengths and new insights into the problem.[1] We begin by describing several environments of the sort we wish to model.

3.1 Environments

In each environment, the robot has a set of discrete *actions* it can execute to move from one environmental state to another. At each environmental state, a set of discrete-valued *sensations* can be detected by the robot. Descriptions of five sample environments follow, the first four of which come from Rivest and Schapire [87]. Note that in these environments, environmental states are not distinguished by unique sensory configurations; hence, the environmental state cannot be determined from the sensations alone.

[1]This research was conducted in collaboration with M. C. Mozer and is reported in refs. [59, 60, 58, 5].

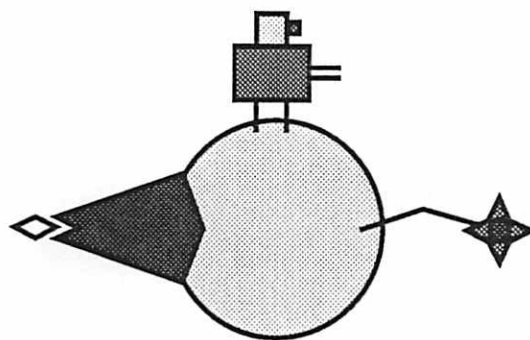

Figure 3.1. The Little Prince Environment.

The Little Prince Environment

The robot resides on the surface of a 2D planet (Figure 3.1). There are four distinct locations on the planet: north, south, east, and west. To the east, there is a rose; to the west, a volcano. The robot has two sensations, one indicating the presence of a rose at the current location, the other a volcano. The robot has available three actions: move to the next location in the direction it is currently facing, move to the next location away from the direction it is facing, and turn its head around to face in the opposite direction.

The Car Radio Environment

The robot can fiddle with knobs on a radio that receives only three stations. It can recognize the type of music that the radio is playing as either classical, rock, or news (Figure 3.2). It can tune in a station with left-scan and right-scan buttons, which scan from one station to the next with wraparound. The robot can also manipulate a set of two presets, X and Y. It can save the current station in either preset, and it can recall a setting in either preset.

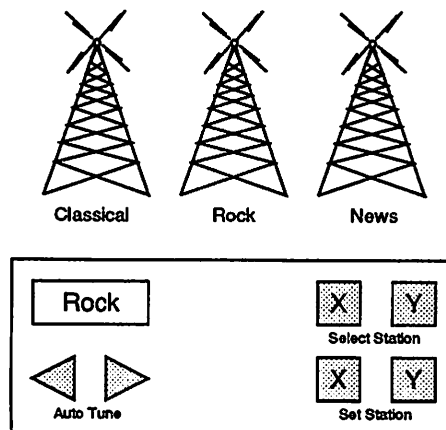

Figure 3.2. The Car Radio Environment.

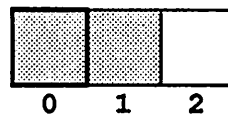

Figure 3.3. An Example of the n-Bit Register Environment. White squares indicate a bit is on. and shaded squares indicate a bit is off. The bold square on the left is the only bit that the robot can sense.

The n-Bit Shift Register Environment

The robot senses the value on the leftmost bit of an n-bit shift register as shown in Figure 3.3. It can rotate the register left or right. The robot can also flip the leftmost bit.

The $n \times n$ Grid Environment

A robot is placed in an $n \times n$ grid with wraparound as shown in Figure 3.4. The robot occupies a square and faces in one of four geographic directions: north, south, east, or west. It can move forward one square or rotate 90° left or right with the left-turn or right-turn actions. Each square is colored either red, green, or blue and the robot has sensors that detect the color of the square it is facing. Moving forward

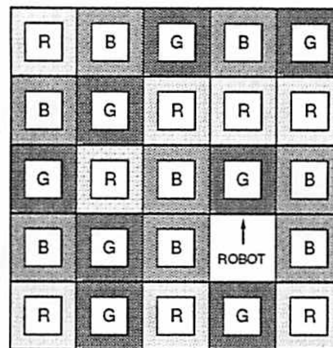

Figure 3.4. An Example of the $n \times n$ Grid Environment.

not only changes the robot's position, but also paints the previous square the color of the square the robot now occupies.

The $n \times n$ Checkerboard Environment

The checkerboard environment consists of an $n \times n$ grid of squares as shown in Figure 3.5. The robot occupies a particular square and can sense the landmark in that square, if any. There are a number of unique landmarks distributed in a checkerboard pattern across the grid. These are depicted as the numbers in the squares. Half of the squares have no landmark and are indistinguishable from each other. At each time step the robot can move either up, down, left, or right one square. Movement off one edge of the grid wraps around to the other side.

3.2 The Update Graph

Rather than trying to learn a state transition graph machine, Rivest and Schapire suggest learning a discrete-distributed machine, called an *update graph*. The advantage of the update graph is that in environments with many regularities, the size of the update graph can be much smaller than the size of the state transition graph machine,

0		1	
	2		3
4		5	
	6		7

Figure 3.5. An Example of the $n \times n$ Checkerboard Environment.

for example, $2n$ versus 2^n state variables in the n-bit shift register environment.[2] The Appendix presents Rivest and Schapire's formal definition of the update graph. Rivest and Schapire's definition is based on the notion of *tests* that can be performed on the environment, and the equivalence of different tests. In this section, we present an alternative, more intuitive view of the update graph that facilitates a connectionist interpretation.

Consider again the 3-bit shift register environment. To model this environment, the essential knowledge required is the values of the bits. Assume the update graph has a node for each of these environment variables, and each node has an associated value indicating whether the bit is on or off.

If we know the values of the variables in the current environmental state, what will their new values be after taking some action, say L? The new value of the leftmost bit, bit 0, becomes the previous value of bit 1; the new value of bit 1 becomes the previous value of bit 2; and the new value of bit 2 becomes the previous value of bit 0. As depicted in Figure 3.6L, this action thus results in shifting values among the three nodes, mimicking the intuitive behavior of a shift register. Figure 3.6R shows the analogous flow of information for the action R. Finally, the action F should cause

[2]The potential disadvantage is that in degenerate, completely unstructured environments, the size of the update graph can be exponentially larger than the size of the state transition graph machine.

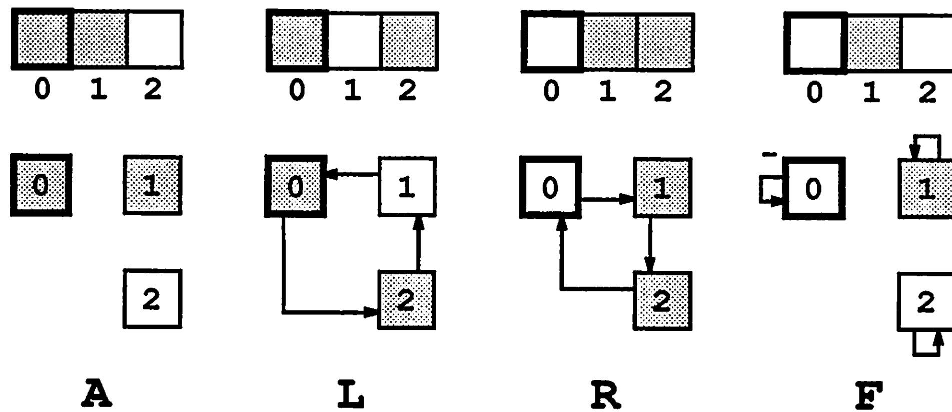

Figure 3.6. The Mechanics of the Update Graph. Panel A shows the 3-bit shift register and the update graph without links. White squares indicate a bit is on, and shaded squares indicate a bit is off. The observable leftmost bit is emphasized with a thicker box. Panels L, R, and F depict the result of performing the designated action in the shift register environment and the corresponding computation of the update graph.

the value of leftmost bit to be complemented while the values of the other bits remain unaffected (Figure 3.6F). In Figure 3.7I, the three sets of links from Panels L, R, and F of Figure 3.6 have been superimposed and have been labeled with their associated actions.

One final detail: the Rivest and Schapire update graph formalism does not make use of the "complementation" link. To avoid it, one may split each node into two values, one representing the value of a bit and the other its complement (Figure 3.7C). Flipping thus involves exchanging the values of bit 0 and bit $\overline{0}$. Just as the values of bit 0, bit 1, and bit 2 must be shifted for the actions `L` and `R`, so must their complements.

Given the update graph in Figure 3.7C and the value of each node for the current environmental state, the result of any sequence of actions can be predicted simply by shifting values around in the graph. Thus, as far as predicting the input/output behavior of the environment is concerned, the update graph serves the same purpose as the state transition graph.

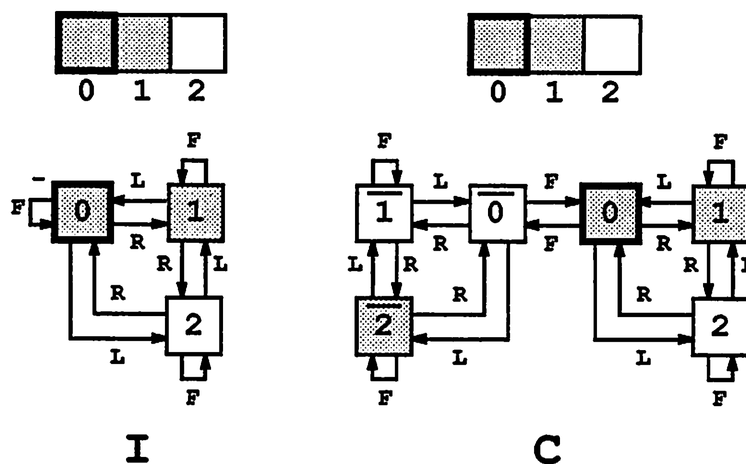

Figure 3.7. The Update Graph for the 3-Bit Shift Register. Panel I shows the complete "intuitive" update graph for the 3-bit shift register environment. Panel C shows the complete update graph (without complementation links) for the 3-bit shift register environment.

For every finite state environment. there exists a corresponding update graph. In fact, the update graph in Figure 3.7C might even be viewed as a distributed representation of the state transition graph in Figure 1.3. In the state transition graph, each environmental state is represented by *one* "active" node. In the update graph, each environmental state represented by a pattern of activity across the nodes.

A defining property of the update graph is that each node has exactly one incoming link for each action. This property, which we call the *one-input-per-action property*, has no intuitive justification based on the description of the update graph we have presented, but see Mozer and Bachrach [60] or Schapire [87] for an explanation. The one-input-per-action property clearly holds in Figure 3.7C; for example, note that bit 0 gets input from bit $\overline{0}$ for the action F, from bit 1 for L. and from bit 2 for R.

In the n-bit shift register environment, it happens that each node has exactly one outgoing link for each action (the *one-output-per-action property*). The one-input-per-action and one-output-per-action properties jointly hold for a class of environments

Rivest and Schapire call *permutation environments*, in which no information is ever lost (i.e., each action can be undone by some fixed sequence of actions). Update graphs of non-permutation environments, such as the car radio environment, do not possess the one-output-per-action property. Thus, one must not consider the one-output-per-action property to be a defining characteristic of the update graph; to do so would restrict the set of environments one could model.

3.2.1 The Rivest and Schapire Learning Algorithm

Rivest and Schapire have developed a symbolic algorithm (hereafter, *the RS algorithm*) to strategically explore an environment and learn its update graph representation. They break the learning problem into two steps: (a) inferring the structure of the update graph, and (b) inferring the values of each node in the update graph. Step (b) is relatively straightforward. Step (a) involves formulating explicit hypotheses about regularities in the environment. These hypotheses are concerned with whether two action sequences are equivalent in terms of their sensory outcome. For example, two equivalent action sequences in the 3-Bit Register environment are LL and R because shifting left twice yields the same value on the leftmost bit as shifting right once, no matter how the bits are initially set. By conducting experiments in the environment, the RS algorithm tests each hypothesis and uses the outcome to construct the update graph.

For permutation environments (described in the previous section), a special version of the RS algorithm is guaranteed to infer the environmental structure—within an acceptable margin of error—in a number of moves polynomial in the number of update graph nodes and the number of alternative actions. The general case of the RS algorithm makes fewer assumptions about the nature of the environment, but has no proof of probable correctness.

If only one hypothesis is tested at a time, the RS algorithm is "single minded" and does not make full use of the environmental feedback obtained. Consequently,

Rivest and Schapire have developed heuristics to test multiple hypotheses at once. These heuristics can improve the efficiency of the basic algorithm by several orders of magnitude [74]. Because reasonable performance is achieved only by incorporating special—and potentially problematic—heuristics to make better use of environmental feedback, it seems worthwhile to explore alternative methods whose natural properties allow them to evaluate multiple hypotheses at once. We have pursued a connectionist approach, which has shown promising results in preliminary experiments as well as suggesting a different conceptualization of the update graph representation.

3.3 Connectionist Approach to Modeling Environments

SLUG is a connectionist network that performs Subsymbolic Learning of Update Graphs. Before the learning process itself can be described, however, we must first consider the desired outcome of learning. That is, what should SLUG look like following training if it is to behave as an update graph? Start by assuming one unit in SLUG for each node in the update graph. The activity level of the unit represents the value associated with the update graph node. Some of these units serve as "outputs" of SLUG. For example, in the 3-bit shift register environment, the output of SLUG is the unit that represents the value of the leftmost bit. In other environments (e.g., the little prince environment), there may be several sensations in which case several output units are required.

What is the analog of the labeled links in the update graph? The labels indicate that values are to be sent down a link when a particular action occurs. In connectionist terms, the links should be *gated* by the action. To elaborate, we might include a set of units that represent the possible actions; these units act to multiplicatively gate the flow of activity between units in the update graph. Thus, when a particular action is to be performed, the corresponding action unit is turned on, and the connections that are gated by this action are enabled.

If the action units form a local representation, i.e., only one is turned on at a time, exactly one set of connections is enabled at a time. Consequently, the gated connections can be replaced by a set of weight matrices, one per action (as shown in Figure 3.8). To predict the consequences of a particular action, say F, the weight matrix for F is simply plugged into the network and activity is allowed to propagate through the connections. In this manner, SLUG is dynamically rewired contingent on the current action. Figure 3.9 shows SLUG executing the action sequence LFLR.

The effect of activity propagation should be that the new activity of a unit is the previous activity of some other unit. A linear activation function is sufficient to achieve this:

$$\boldsymbol{x}(t) = W_{a(t)}\boldsymbol{x}(t-1), \tag{3.1}$$

where $a(t)$ is the action selected at time t, $W_{a(t)}$ is the weight matrix associated with this action, and $\boldsymbol{x}(t)$ is the activity vector that results from taking action $a(t)$. Assuming weight matrices which have zeroes in each row except for one connection of strength 1, the activation rule will cause activity values to be copied around the network.

Although nonlinearities are introduced by the operation of rewiring SLUG based on the current action, the behavior of SLUG *in a single time step* is indeed linear. This is handy because it allows us to use tools of linear algebra to better understand the network's behavior. We elaborate on this point in Mozer and Bachrach [60].

3.4 Training SLUG

We have described how SLUG could be hand-wired to behave as an update graph, and now we turn to the procedure used to *learn* the appropriate connection strengths. For expository purposes, assume that the number of units in the update graph is known in advance. (This is not necessary, as we show below.) SLUG starts off with this many units, s of which are set aside to represent the sensations. These s units are the output units of the network; the remainder are hidden units. A set

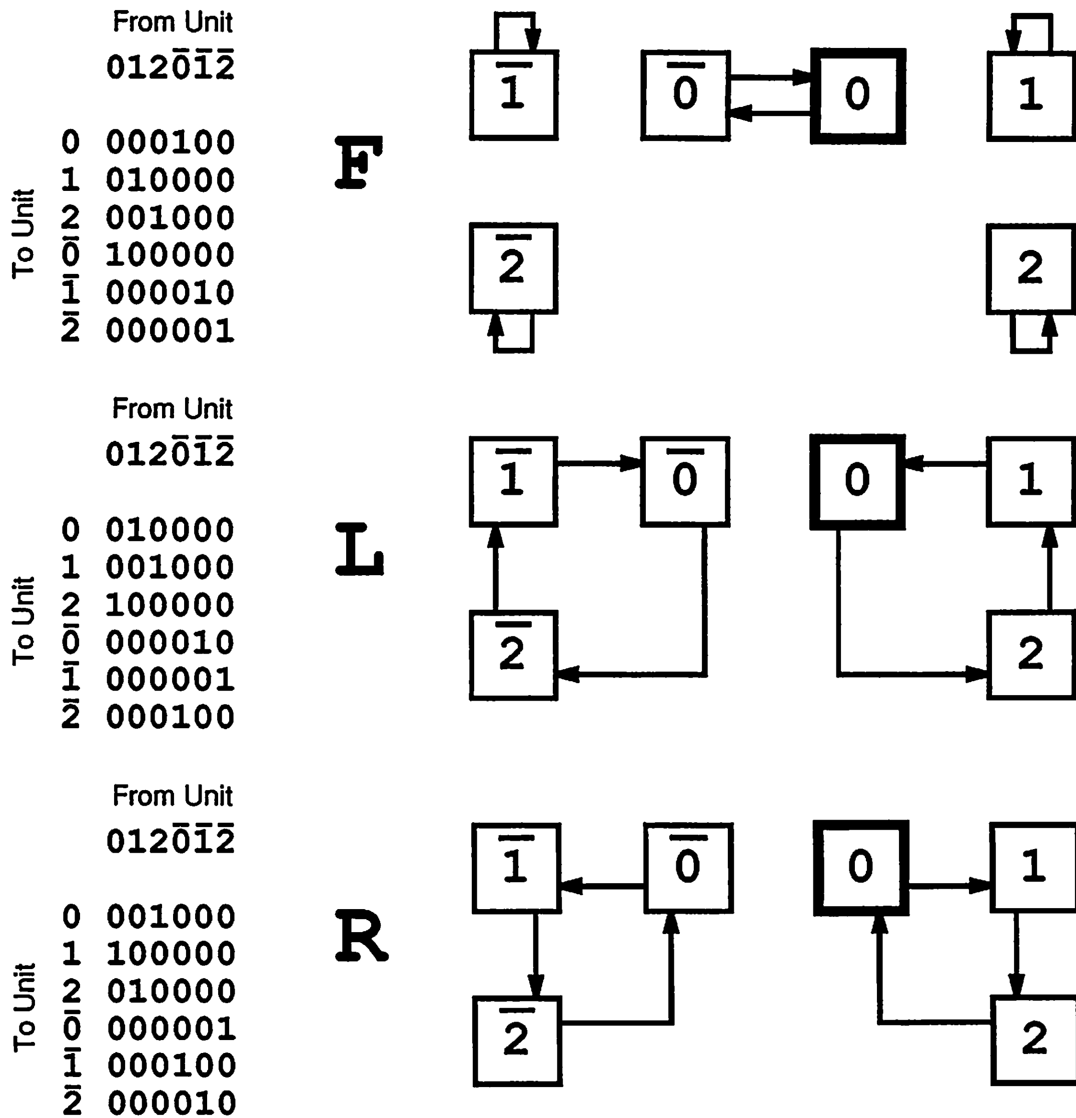

Figure 3.8. SLUG Depicted as a Separate Weight Matrix for Each Action. The left side of the figure shows the separate weight matrices, and the right side shows the connections having nonzero weights.

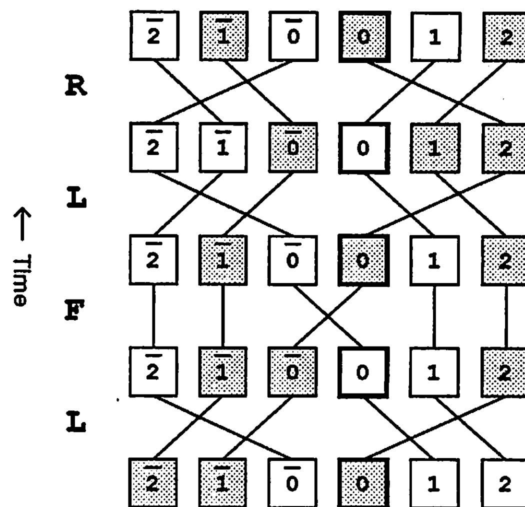

Figure 3.9. SLUG Is Dynamically Rewired Contingent on the Current Action. SLUG is shown executing the action sequence LFLR.

of weight matrices, $\{W_a\}$, is constructed—one per action—and initialized to random values. To train SLUG, random actions are performed on the environment and the resulting sensations are compared with those predicted by SLUG. The mismatch between observed and predicted sensations provides an error measure, and the weight matrices can be adjusted to minimize this error.

This procedure in no way guarantees that the resulting weights will possess the one-input-per-action property, which is required of the update graph connectivity matrices. We can try to achieve this property by performing gradient descent not in the $\{W_a\}$ directly, but in an underlying parameter space, $\{Z_a\}$, from which the weights are derived using a normalized exponential transform:

$$w_{aij} = \frac{e^{z_{aij}/T}}{\sum_k e^{z_{aik}/T}}, \tag{3.2}$$

where w_{aij} is the strength of connection to unit i from unit j for action a, z_{aij} is the corresponding underlying parameter, and T is a constant. This approach [3] permits

[3]This approach was suggested by the recent work of Bridle [12], Durbin [17], and Rumelhart [78], who have applied the normalized exponential transform to activity vectors; in contrast, we have applied it to weight vectors.

unconstrained gradient descent in $\{Z_a\}$ while constraining the w_{aij} to nonnegative values and

$$\sum_i w_{aij} = 1. \tag{3.3}$$

By gradually lowering T over time, the solution can be further constrained so that all but one incoming weight to a unit approaches zero. In practice, we have found that lowering T is unnecessary because solutions discovered with a fixed T essentially achieve the one-input-per-action property.

Below, we report on simulations of SLUG using this approach, which we will refer to as the version of the algorithm with *constrained weights*, and also simulations run with unconstrained weights. In the latter case,

$$w_{aij} = z_{aij}. \tag{3.4}$$

Details of the Training Procedure

Before the start of training, initial values of the z_{aij} are randomly selected from a uniform distribution in the range $[-1, 1]$, and the w_{aij} are derived therefrom. The activity vector $\boldsymbol{x}(0)$ is reset to $\mathbf{0}$. At each time t, the following sequence of events transpires:

1. An action, $a(t)$, is selected at random.
2. The weight matrix for that action, $W_{a(t)}$, is used to compute the activities at t, $\boldsymbol{x}(t)$, from the previous activities $\boldsymbol{x}(t-1)$.
3. The selected action is performed on the environment, and the resulting sensations are observed.
4. The observed sensations are compared with the sensations predicted by SLUG (i.e., the activities of the units chosen to represent the sensations) to compute a measure of error.

5. The back-propagation algorithm (Rumelhart et al. [80]) is used to compute the derivative of the error with respect to each weight, $\partial E/\partial w_{aij}$. Because SLUG contains recurrent connections and back-propagation applies only to feedforward nets, the "unfolding-in-time" procedure of Rumelhart et al. is used. This procedure is based on the observation that any recurrent network can be transformed into a feedforward network with identical behavior, over a finite period of time. The feedforward net will have a layer of units corresponding to the units of SLUG at each time step: the top layer of the feedforward net represents $\boldsymbol{x}(t)$, the layer below represents $\boldsymbol{x}(t-1)$, below that $\boldsymbol{x}(t-2)$, and so on back to a layer that represents $\boldsymbol{x}(t-\tau)$. (We discuss the choice of τ below.) The weights feeding into the top layer are $W_{a(t)}$, the weights feeding into the layer below that are $W_{a(t-1)}$, and so forth. Back-propagation can be applied in a direct manner to this feedforward network. The error gradient for some action i, $\partial E/\partial W_i$, is computed by summing the back-propagated error derivatives over all layers l of the feedforward net in which the action corresponding to that layer, $a(l)$, is equal to i.

6. The error gradient in terms of the $\{W_a\}$ is transformed into a gradient in terms of the $\{Z_a\}$. With constrained weights (Equation 3.2), the relation is

$$\frac{\partial E}{\partial z_{aij}} = \frac{w_{aij}}{T}\left[\frac{\partial E}{\partial w_{aij}} - \sum_k \frac{\partial E}{\partial w_{aik}} w_{aik}\right] \tag{3.5}$$

With unconstrained weights (Equation 3.4), the mapping is simply the identity.

7. The $\{Z_a\}$ are updated by taking a step of size η down the error gradient:

$$\Delta z_{aij} = -\eta \frac{\partial E}{\partial z_{aij}}. \tag{3.6}$$

The $\{W_a\}$ are then recomputed from the new $\{Z_a\}$.

8. The temporal record of unit activities, $\boldsymbol{x}(t-i)$ for $i = 0 \ldots \tau$, which is maintained to permit back-propagation in time, is updated to reflect the new weights. This

involves recomputing the forward flow of activity from time $t - \tau$ to t for the hidden units. (The output units are unaffected because their values are forced, as described in the next step.)

9. The activities of the output units at time t, which represent the predicted sensations, are replaced by the observed sensations. This implements a form of *teacher forcing* (Williams and Zipser [108]). A consequence of teacher forcing is that the error derivative for weights feeding into the output units at times $t-1$, $t-2$, etc. should be set to zero in Step 5; that is, error should not be back-propagated from time t to output units at earlier times. It is not sensible to adjust the response properties of output units at some earlier time i to achieve the correct response at time t because the appropriate activation levels of these units have already been established by the sensations at time i.

Steps 5 and 8 require further elaboration. One parameter of training is the amount of temporal history, τ, to consider. We have found that, for a particular problem, error propagation beyond a certain critical number of time steps does not help SLUG to discover a solution more quickly, although any fewer does indeed hamper learning. In the results described below, we arbitrarily set τ for a particular problem to one less than the number of nodes in the update graph solution of the problem. Informal experiments manipulating τ revealed that this was a fairly conservative choice. To avoid the issue of selecting a value for τ, one could instead use the on-line recurrent network training algorithm of Williams and Zipser [107].

To back-propagate error in time, a temporal record of unit activities is maintained. However, a problem arises with these activities following a weight update: the activities are no longer consistent with the weights—i.e., Equation 3.1 is violated. Because the error derivatives computed by back-propagation are exact only when Equation 3.1 is satisfied, future weight updates based on the inconsistent activities

are not assured of being correct. Empirically, we have found the algorithm extremely unstable if we do not address this problem.

In most situations where back-propagation is applied to temporally-extended sequences, the sequences are of finite length. Consequently, it is possible to wait until the end of the sequence to update the weights, at which point consistency between activities and weights no longer matters because the system starts afresh at the beginning of the next sequence. In the present situation, however, the sequence of actions does not terminate. We were thus forced to consider alternative means of ensuring consistency. One approach we tried involved updating the weights only after every, say, 25 time steps. Immediately following the update, the weights and activities are inconsistent, but after τ time steps (when the inconsistent activities drop off the activity history record), consistency is once again achieved. A more successful approach involved updating the activities after each weight change to force consistency (Step 8 on the list above). To do this, we propagated the earliest activities in the temporal record, $\boldsymbol{x}(t-\tau)$, forward again to time t, using the updated weight matrices.[4]

The issue of consistency arises because at no point in time is SLUG instructed as to the state of the environment. That is, instead of being given an activity vector as input, part of SLUG's learning task is to discover the appropriate activity vector. This might suggest a strategy of explicitly learning the activity vector, that is, performing gradient descent in both the weight space and activity space. However, our experiments indicate that this strategy does not improve SLUG's rate of learning. One plausible explanation is the following. If we perform gradient descent in just

[4]Keeping the original value of $\boldsymbol{x}(t-\tau)$ is a somewhat arbitrary choice. Consistency can be achieved by propagating *any* value of $\boldsymbol{x}(t-\tau)$ forward in time, and there is no strong reason for believing $\boldsymbol{x}(t-\tau)$ is the appropriate value. We thus suggest two alternative schemes, but have not yet tested them. First, we might select $\boldsymbol{x}(t-\tau)$ such that the new $\boldsymbol{x}(t-i)$, $i=0\ldots\tau-1$, are as close as possible to the old values. Second, we might select $\boldsymbol{x}(t-\tau)$ such that the output units produce as close to the correct values as possible. Both these schemes require the computation-intensive operation of finding a least squares solution to a set of linear equations.

weight space based on the error from a single trial, and then force activity-weight consistency, the updated output unit activities are guaranteed to be closer to the target values, assuming a sufficiently small learning rate. Thus, the effect of this procedure is to reduce the error in the observable components of the activity vector, which is similar to performing gradient descent in activity space directly.

A final comment regarding the training procedure: in our simulations, learning performance was better with target activity levels of -1 and $+1$ (indicating that the leftmost bit of the shift register is *off* or *on*, respectively) rather than 0 and 1. One explanation for this is that random activations and random (nonnegative) connection strengths tend to cancel out in the $-1/+1$ case, but not in the $0/1$ case. For other arguments concerning the advantages of symmetric activity levels see Stornetta and Huberman [93].

3.5 Results

Figure 3.10 shows the weights in SLUG for the 3-bit shift register environment at three stages of training. The "step" refers to how many moves SLUG has taken, or equivalently, how many times the weights have been updated. The bottom diagram in the figure corresponds to the update graph of Figure 3.7C. To explain the correspondence, think of the diagram as being in the shape of a person who has a head, left and right arms, left and right legs, and a heart. For the action `L`, the head—the output unit—receives input from the left leg, the left leg from the heart, and the heart from the head, thereby forming a three-unit loop. The other three units—the left arm, right leg, and right arm—form a similar loop. For the action `R`, the same two loops are present but in the reverse direction. These two loops also appear in Figure 3.7C. For the action `F`, the left and right arms, heart, and left leg each keep their current value, while the head and the right leg exchange values. This corresponds to the exchange of values between the bit 0 and bit $\overline{0}$ nodes of the Figure 3.7C.

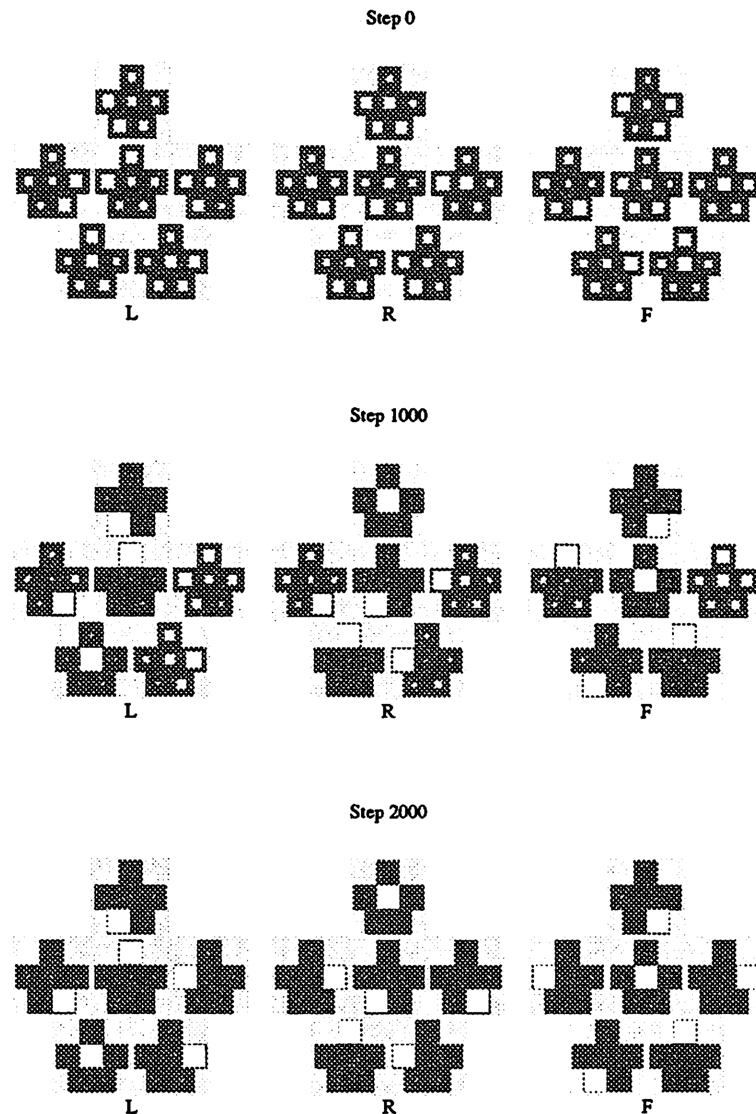

Figure 3.10. SLUG's Weights at Three Stages of Training for the 3-Bit Shift Register Environment. Step 0 reflects the initial random weights, Step 1000 reflects the weights midway through learning, and Step 2000 reflects the weights upon completion of learning. Each large diagram (with a light gray background) represents the weights corresponding to one of the three actions. Each small diagram contained within a large diagram (with a dark gray background) represents the connection strengths feeding into a particular unit for a particular action. There are six units, hence six small diagrams within each large diagram. The output unit, which indicates the state of the light in the current room, is the protruding "head" of the large diagram. A white square in a particular position of a small diagram represents the strength of connection from the unit in the homologous position in the large diagram to the unit represented by the small diagram. The area of the square is proportional to the connection strength, with the largest square having the value 1.

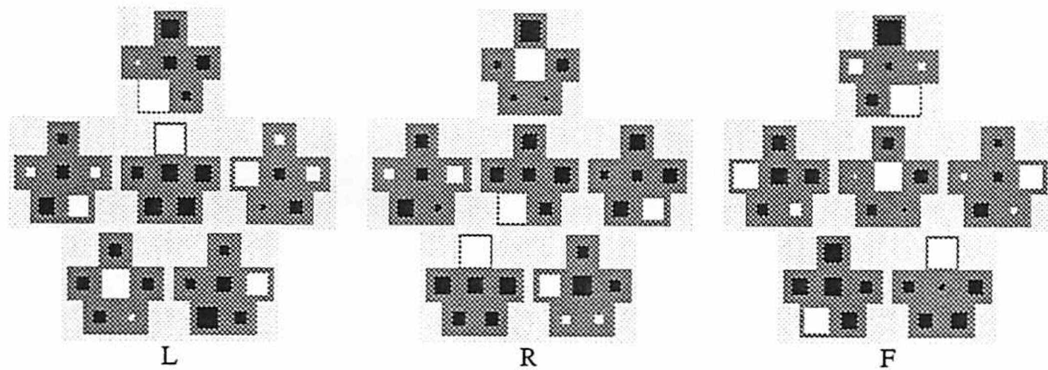

Figure 3.11. The Underlying z_{aij} Parameters. These parameters correspond to the weights in the bottom diagram of Figure 3.10. White squares indicate positive values, black squares negative values. The largest square represents a value of 4.5.

The weights depicted in Figure 3.10 are the w_{aij}'s. These weights are derived from the underlying parameters z_{aij}. The values of z_{aij} corresponding to the weights at Step 2000 are shown in Figure 3.11. The normalized exponential transform maps z_{aij} values in the range $-\infty \rightarrow +\infty$ to values in the range $0 \rightarrow 1$.

In addition to learning the update graph connectivity, SLUG has simultaneously learned the correct activity values associated with each node for the current state of

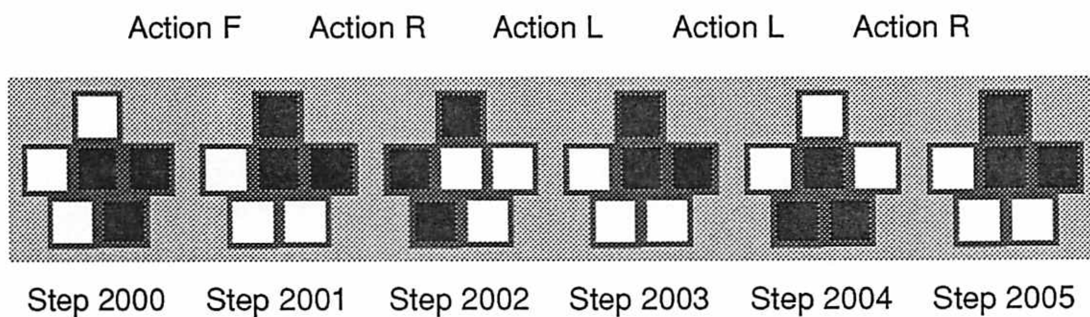

Figure 3.12. The Pattern of Activity in SLUG at Six Consecutive Time Steps. White squares indicate a unit with activity level of +1, black squares a unit with activity level −1. The arrangement of units in this figure matches the arrangement of weights in the previous two figures. Thus, the output unit is the protruding "head" of each diagram. The transition between two consecutive states arises from performing the action printed between the two states.

the environment. Armed with this information, SLUG can predict the outcome of any sequence of actions. Indeed, the prediction error drops to zero, causing learning to cease and SLUG to become completely stable. Figure 3.12 shows the evolution of activity in SLUG at six consecutive time steps. At Step 2000, the value of the leftmost bit is 1, as indicated by the white square in the diagram's head. When the action F is performed, the head and right leg exchange values while all other units maintain their current value. Between Steps 2001 and 2005, the action sequence RLLR is executed. Because shifting left and right are complementary actions, SLUG's state at Step 2005 is the same as at Step 2001.

SLUG shows two advantages over other connectionist approaches to learning finite-state automata (e.g., Elman [22]; Pollack [68]; Servan-Schreiber, Cleeremans, and McClelland [91]). First, because the ultimate weights and activities are discrete, SLUG can operate indefinitely with no degradation in its ability to predict the future. Other connectionist networks have the drawback that states are not represented discretely. Consequently, minor numerical imprecisions can accumulate, causing the networks to wander from one state to another. For example, Pollack [68] trained a network to accept strings of the regular language 1* (i.e., any number of 1's). However, the network would accept strings containing a 0 if followed by a long enough string of 1's. The network would eventually "forget" that it was in the reject state. SLUG with constrained weights is not susceptible to this problem, as illustrated by the fact that the state at Step 2001 in Figure 3.12 is *exactly* the same as the state at Step 2005. The second advantage of SLUG over other connectionist approaches is that the network weights and activities can readily be interpreted—as an update graph. Consequently, the correctness of SLUG's solution can be assessed analytically.

To evaluate SLUG, we examined its performance as measured by the number of actions that must be executed before the outcomes of subsequent actions can be predicted, that is, before a perfect model of the environment has been constructed. One means of determining when SLUG has reached this criterion is to verify that

the weights and activities indeed correspond to the correct update graph. Such a determination is not easy to automate: we instead opted for a somewhat simpler criterion: SLUG is considered to have learned a task by a given time step if the correct predictions are made for at least the next 2500 steps. In all simulation results we report in this paper, performance is measured by this criterion—the number of time steps before SLUG can correctly predict the sensations at the 2500 steps that follow. Because results vary from one run to another due to the random initial weights, we report the median performance over 25 replications of each simulation. On any replication, if SLUG was unable to converge on a solution within 100,000 steps, we stopped the simulation and counted it as a failure.

Now for the bad news: SLUG succeeded in discovering the correct update graph for the 3-bit shift register environment on only 15 of 25 runs. The median number of steps taken by SLUG before training terminated—either by reaching the performance criterion on successful runs or by reaching the maximum number of steps on failed runs—was 6,428. Even on failed runs, however, SLUG does learn most of the update graph, as evidenced by the fact that after 10,000 steps SLUG correctly predicts sensations with an average accuracy of 93%.

With unconstrained weights, SLUG's performance dramatically improves: SLUG reaches the performance criterion in a median of 298 steps. From our experiments, it appears that constraints on the weights seldom help SLUG discover a solution. In the remainder of this article, we therefore describe the performance of the version of SLUG with unconstrained weights. In Mozer and Bachrach [60], we consider why constraining the weights is harmful and possible remedies.

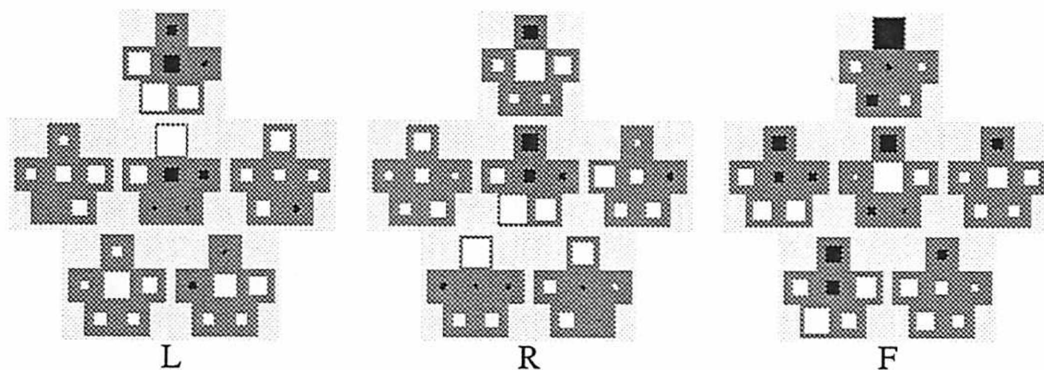

Figure 3.13. Weights Learned by SLUG with Six Units and Unconstrained Weights for the 3-Bit Shift Register Environment. White squares indicate positive weights, black negative.

3.6 Comparing SLUG and Other Connectionist Approaches

With unconstrained weights, SLUG suffers from the two drawbacks common to other connectionist approaches that we described earlier. First, slight inaccuracies in the continuous-valued weights and activities can cause degeneration of SLUG's predictive abilities in the distant future. Consequently, it is essential that SLUG converge on an exact solution. To assist towards this end, we scaled down the learning rate, η, as SLUG approached a solution. (That is, η was set in proportion to the mean squared prediction error.) The second advantage lost when weights are unconstrained is that, although SLUG learns effectively, the resulting weight matrices, which contains a collection of positive and negative weights of varying magnitudes, is not readily interpretable (see Figure 3.13).

The loss of these two benefits might seem to indicate that SLUG is no better than other connectionist approaches to the problem of inducing finite-state environments, but comparisons of SLUG and a conventional connectionist recurrent architecture indicate that SLUG achieves far superior performance. The "conventional" architecture we tested was a three layer network consisting of *input, hidden, and output* units, with feedforward connections from input to hidden and hidden to output layers

as well as recurrent connections within the hidden layer (Bachrach [3]; Elman [22]; Mozer [56]; Servan-Schreiber et al. [90]). Input units represent the current sensations and an action, output units represent the sensations predicted following the action. Our experiments using the conventional architecture were spectacularly unsuccessful. We were unable to get the conventional architecture to learn the 3-bit shift register environment on even one run, despite our best efforts at varying the number of hidden units, the number of steps τ through which error was back-propagated, and learning rates in every conceivable combination. We lowered our sights somewhat and attempted to train the conventional architecture on a simpler environment, the little prince environment. The conventional architecture was able to learn this environment on 24 of 25 replications, with a median of 27,867 steps until the prediction task was mastered. In contrast, SLUG learned perfectly on every replication and in a median of 91 steps.

Besides testing the conventional architecture on the environment inference problem, we also tested SLUG on problems studied using the conventional architecture.[5] Servan-Schreiber et al. [90] have worked on the problem of inferring regular languages from examples, which is formally identical to the environment inference problem except that an exploration strategy is unnecessary because sample strings are given to the system. They trained the conventional architecture on strings of a language. After 200,000 sample strings, the network was able to reject invalid strings. In contrast, SLUG required a mere 150 training examples on average before it mastered the acceptability judgement task.[6]

The fact that SLUG achieves considerably better performance than conventional connectionist approaches justifies its nonintuitive network architecture, dynamics,

[5]This comparison was performed in collaboration with Paul Smolensky.

[6]The training procedure was slightly different for the two architectures. Servan-Schreiber et al.'s network was shown positive examples only, and was trained to predict the next element in a sequence. SLUG was shown both positive and negative examples, and was trained to accept or reject an entire string. We see no principled reason why one task would be more difficult than the other.

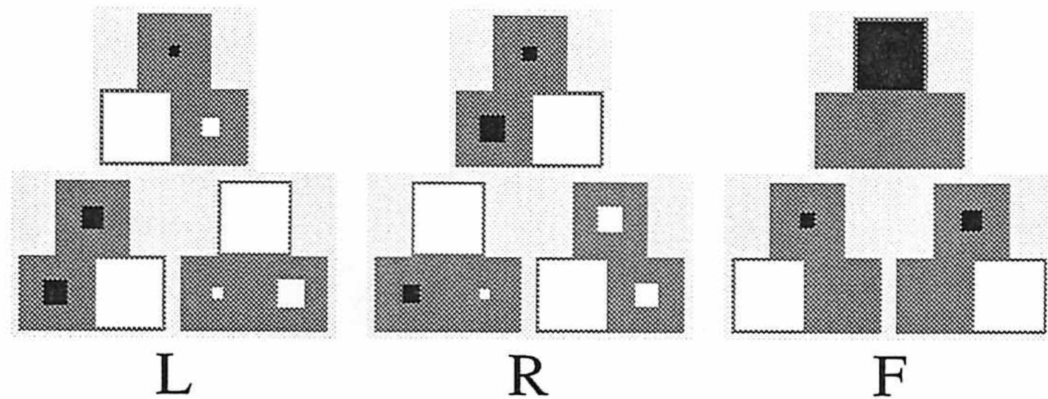

Figure 3.14. Weights Learned by SLUG with Three Units and Unconstrained Weights for the 3-Bit Shift Register Environment.

and training procedure. [7] Thus, Rivest and Schapire's update graph representation, which motivated the architecture of SLUG, has proven beneficial, even if what SLUG (with unconstrained weights) learns does not correspond exactly to an update graph.

Interpreting unconstrained solutions discovered by SLUG

One reason why the final weights in SLUG are difficult to interpret in the case of the 3-bit shift register environment is that SLUG has discovered a solution that does not satisfy the update graph formalism; it has discovered the notion of complementation links of the sort shown in Figure 3.7I. With the use of complementation links, only three units are required, not six. Consequently, the three unnecessary units are either cut out of the solution or encode information redundantly. SLUG's solutions are much easier to understand when the network consists of only three units. Figure 3.14 depicts one such solution, which approximates the graph in Figure 3.7I. Ignoring the connections of small magnitude, the three units are connected in a clockwise loop for action `L`, a counterclockwise loop for action `R`, and the output unit toggles its value while the two internal units maintain their values for action `F`. Although a

[7] It remains to be seen which properties of SLUG that differ from the conventional connectionist approach are crucial for SLUG's improved performance. One key property is undoubtably the gating connections between actions and state units. See Giles et al. [26] for a related architecture that also shares this property.

comparison of Figures 3.14 and 3.7I might seem to indicate that the connections of small magnitude in Figure 3.14 introduce noise, this is in fact not the case. The solution discovered by SLUG is exact—it predicts sensations accurately because the effects of these connections exactly cancel out. SLUG also discovers other solutions in which two of the three connections in the three-unit loop are negative, one negation canceling out the effect of the other. Allowing complementation links can halve the number of update graph nodes required for many environments. This is one fairly direct extension of Rivest and Schapire's update graph formalism that SLUG suggests.

Treating the update graph as matrices of connection strengths has suggested another generalization of the update graph formalism. Because SLUG is a linear system, any rank-preserving linear transform of the weight matrices will produce an equivalent system, but one that does not have the local connectivity of the update graph. Thus, one can view the Rivest and Schapire update graph formalism as one example of a much larger class of equivalent solutions that can be embodied in a connectionist network. While many of these solutions do not obey constraints imposed by a symbolic description (e.g., all-or-none links between nodes), they do yield equivalent behavior. By relaxing the symbolic constraints, the connectionist representation allows far greater flexibility in expressing potential solutions. See Mozer and Bachrach [60] for a further elaboration of this point.

Comparing SLUG and the RS Algorithm

Table 3.1 compares the performance of the RS algorithm and SLUG for a sampling of environments. [8] In these simulations, we ran the version of SLUG with unconstrained weights. All runs of SLUG eventually converged on an adequate set of weights. The learning rates used in our simulations were adjusted dynamically every 100 steps by averaging the current learning rate with a value proportional to

[8]We thank Rob Schapire for providing us with the latest results from his work.

Table 3.1. Number of Steps Required to Learn Update Graph.

Environment	Size of Update Graph	Median Number of Actions	
		The RS Algorithm	SLUG
Little Prince	4	200	91
Car Radio	6	27,695	8.167
3-Bit Shift Register	8	408	298
4-Bit Shift Register	9	1,388	1.509
5×5 Grid	27	583,195	fails
32-Bit Shift Register	32	52,436	fails
6×6 Checkerboard	36	96.041	8.142

the mean squared error obtained on the last 100 steps. Several runs were made to determine what initial learning rate and constant of proportionality yielded the best performance. It turned out that performance was relatively invariant under a wide range of these parameters. Including momentum in the back-propagation algorithm did not appear to help significantly.[9]

In simple environments, SLUG can outperform the RS algorithm. These results are quite surprising when considering that the action sequence used to train SLUG is generated at random, in contrast to the RS algorithm. which involves a strategy for exploring the environment. We conjecture that SLUG does as well as it does because it considers and updates many hypotheses in parallel at each time step. That is, after the outcome of a single action is observed, nearly all weights in SLUG are adjusted simultaneously. In contrast, the RS algorithm requires special-purpose heuristics that are not necessarily robust in order to test multiple hypotheses at once.

A further example of SLUG's parallelism is that it learns the update graph structure at the same time as the appropriate unit activations, whereas the RS algorithm approaches the two tasks sequentially. During learning, SLUG continually

[9]Just as connectionist simulations require a bit of voodoo in setting learning rates, the RS algorithm has its own set of adjustable parameters that influence performance. We experimented with the RS algorithm, and without expertise in parameter tweaking, were unable to obtain performance in the same range as the measures reported by Rivest and Schapire.

makes predictions about what sensations will result from a particular action. These predictions gradually improve with experience, and even before learning is complete, the predictions can be substantially correct. The RS algorithm cannot make predictions based on its partially constructed update graph. Although the algorithm could perhaps be modified to do so, there would be an associated cost.

In complex environments—ones in which the number of nodes in the update graph is quite large and the number of distinguishing environmental sensations is relatively small—SLUG does poorly. Two examples of such, the 32-bit shift register environment and the 5×5 grid environment, cannot be learned by SLUG whereas the RS algorithm succeeds. An intelligent exploration strategy seems necessary in complex environments: with a random exploration strategy, the time required to move from one state to a distant state becomes so great that links between the states cannot be established.

The 32-bit shift register environment and the 5×5 grid environment are extreme: all locations are identical and the available sensory information is meager. Such environments are quite unlike natural environments, which provide a relative abundance of sensory information to distinguish among environmental states. SLUG performs much better when more information about the environment can be sensed directly. For example, learning the 32-bit shift register environment is trivial if SLUG is able to sense the values of all 32 bits at once (the median number of steps to learn is only 1209). The checkerboard environment is another environment as large as the 32-bit shift register environment in terms of the number of nodes SLUG requires, but it is much easier to learn because of the rich sensory information.

Noisy environments. The RS algorithm originally could not handle environments with unreliable sensations, although a variant of the algorithm has recently been designed to overcome this limitation (Schapire [86]). In contrast, SLUG, like most connectionist systems, deals naturally with noise in that SLUG's ability to predict degrades gracefully in the presence of noise. To illustrate this point, we trained

SLUG on a version of the little prince environment in which sensations are registered incorrectly 10% of the time. SLUG was still able to learn the update graph. However, to train SLUG properly in noisy environments, we needed to alter the training procedure slightly, in particular, the step in which the observed sensations replace SLUG's predicted sensations. The problem is that if the observed sensations are incorrectly registered, the values of nodes in the network will be disrupted, and SLUG will require a series of noise-free steps to recover. Thus, we used a procedure in which the predicted sensations were not completely replaced by the observed sensations, but rather some average of the two was computed, according to the formula

$$(1-\omega)p_i + \omega o_i, \tag{3.7}$$

where p_i and o_i are the predicted and observed values of sensation i, and ω is a weighting factor. [10] The value of ω is a function of SLUG's performance level, as measured by the mean proportion of prediction errors. If SLUG is making relatively few errors, it has learned the correct environment model and ω should be set to 0—the observed value should be ignored because it is occasionally incorrect. However, if SLUG is performing poorly, the prediction p_i should not be relied upon, and ω should have a value closer to 1. Rather than computing ω as a function of SLUG's performance, another possibility is to adjust ω via gradient descent in the overall error measure.

Prior specification of update graph size. The RS algorithm requires an upper bound on the number of nodes in the update graph. The results presented in Table 3.1 are obtained when the RS algorithm knows in advance exactly how many nodes are required. The algorithm fails if it is given an upper bound less than the required number of nodes, and performance—measured as the number of steps required to discover the solution—degrades as the upper bound increases above the

[10] A more principled but computationally more expensive technique for updating the predicted sensations can be derived using Kalman filtering theory (Gelb [25]).

Table 3.2. Number of Steps Required to Learn Update Graph as the Number of Units in SLUG Is Varied.

Units in SLUG	Median Number of Steps to Learn Update Graph
4	2028
6	1380
8	1509
10	1496
12	1484
14	1630
16	1522
18	1515
20	1565

required number. SLUG will also fail to learn perfectly if it is given fewer units than are necessary for the task. However, performance does not appear to degrade as the number of units increases beyond the minimal number. Table 3.2 presents the median number of steps required to learn the 4-bit shift register environment as the number of units in SLUG (with unconstrained weights) is varied. Although performance is independent of the number of units here, extraneous units greatly *improve* performance when the weights are constrained: only 4 of 25 replications of the 4-bit shift register environment simulation with 8 units and constrained weights successfully learned the update graph, whereas 19 of 25 replications succeeded when 16 units were used.

3.7 Limitations of SLUG

The connectionist approach to the problem of inferring the structure of a finite-state environment has two fundamental problems that must be overcome if it is to be considered seriously as an alternative to the symbolic approach. First, using a random exploration strategy, SLUG has no hope of scaling to complex environments. An intelligent strategy could potentially be incorporated to encourage SLUG to explore

unfamiliar regions of the environment. One approach we are currently investigating, based on the work of Cohn et al. [14], is to have SLUG select actions that result in maximal uncertainty in its predictions, where uncertainty is defined as how far a predicted sensation is from one of the discrete sensation values. Second, our greatest successes have occurred when we allowed SLUG to discover solutions that are not necessarily isomorphic to an update graph. One virtue of the update graph formalism is that it is relatively easy to interpret; the same cannot generally be said of the continuous-valued weight matrices discovered by SLUG. However, as we discuss in Mozer and Bachrach [60], there is promise of developing methods for transforming the large class of formally equivalent solutions available to SLUG into the more localist update graph formalism to facilitate interpretation.

CHAPTER 4

LEARNING TO CONTROL A FINITE STATE AUTOMATON

In this chapter we discuss a control acquisition architecture that uses a model to determine a minimal length sequence of actions to reach a particular goal state. The architecture consists of a *model* and an *adaptive critic*. The basic architecture is shown in Figure 4.1 and will be called the *control network*. The model maps its current state and the current action to the predicted next state, and the adaptive critic maps the model's next state to the expected time to reach the goal state. The model's state consists of both predicted sensations and hidden state information, corresponding to the output and hidden units of SLUG.

At each time step the control action is chosen which minimizes the output of the adaptive critic. This is done by using the control network to evaluate, at each time step, each possible control action. The control network computes a prediction of the time to reach the goal state if that particular action were taken in the current state. The control network performs this computation in two parts: first, the model computes the state q predicted to result from taking that particular action; second, the adaptive critic computes the prediction of the time to reach the goal state from state q. The action which results in the minimum prediction of time to reach the goal state is then performed.

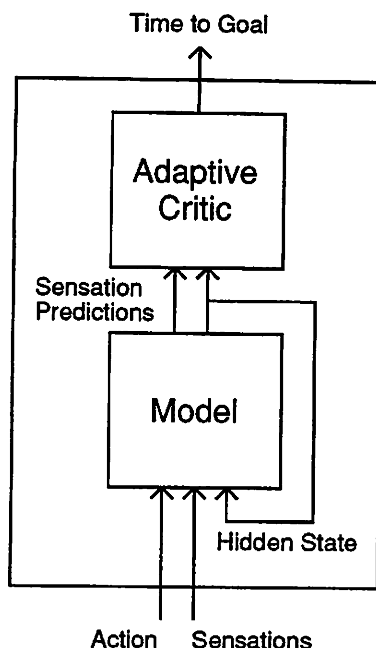

Figure 4.1. Connectionist Control Acquisition Architecture.

4.1 Training

In our experiments, our training procedure is divided into two phases. First, the model is trained during an exploration phase using the unconstrained version of SLUG. The weights of SLUG are then fixed. In the second phase, the control network is trained to produce optimal trajectories from arbitrary initial states to a particular goal state. This phase involves a series of trials which begin and end in the goal state. At the beginning of each trial, the robot explores for a fixed number of steps by taking actions selected randomly from a uniform distribution. The number of such steps is chosen to be the number of states in the environment. This number was chosen so that there is a nonzero probability of the exploration phase ending in any particular environmental state. This places the robot in a random state, which we regard as the initial state for the remainder of the trial. [1] During the rest of the trial, the

[1]There are two motivations for this wandering procedure. First, in a learning situation with minimal supervision, the robot needs to get into the initial state on its own; it cannot depend on

robot performs the actions that produce minimal predictions of the time to reach the goal state according to its current model and adaptive critic. In order to encourage exploration during control trials, the robot sometimes (with probability 0.25 at each time step in all the simulations [2]) selects alternative actions instead of selecting the action that is best according to the current predictions. This exploration strategy is not optimal, but provides the necessary exposure to the state space. Sutton [96] and Watkins [99] present more sophisticated techniques for exploration. Throughout the entire control acquisition phase—during wandering and during the control trials—the model constantly updates itself to reflect the current state of the environment.

The adaptive critic is trained using a *temporal difference (TD)* method developed by Sutton [94, 95]. A system using this method learns to predict by maintaining consistency across successive predictions and by forcing the prediction to agree with any available training signals. In the case of predicting the time to reach the goal state, the adaptive critic is trained to make successive predictions decrease by one and to make the prediction at the goal state zero. The robot receives a reinforcement signal that indicates when it is located at the goal; the reinforcement signal is 1 when the robot is located at the goal and 0 otherwise. Finally, the adaptive critic is trained only when the robot actually selects the action that currently looks best and not when the robot takes exploratory steps (see Watkins [99] for justification).

The adaptive critic is implemented as either a one-layer or two-layer network with a single linear output unit. The hidden layer is unnecessary for the little prince and $n \times n$ checkerboard environments. The adaptive critic's optional hidden layer consists of logistic units with outputs ranging from -1 to $+1$.

---

a supervisor to place it in a random initial state. Second, the model needs to reflect the current environmental state. If the robot is placed in a random state, substantial effort would be required to identify the environmental state. However, by starting at the goal state and wandering from there, the robot can maintain continual knowledge of the environmental state.

[2] After trying a few different probabilities, 0.25 was found to produce the shortest learning times.

Table 4.1. The Control Acquisition Architecture was Applied to these Environments with their Respective Goal States.

Environment	Goal State
Little Prince	Robot on the north location facing east
Car Radio	Playing Classical, Rock in preset X, and News in preset Y
3-Bit Shift Register	Bit pattern `101`
4-Bit Shift Register	Bit pattern `1010`
4×4 Checkerboard	Square `(1,1)`
6×6 Checkerboard	Square `(2,2)`
8×8 Checkerboard	Square `(3,3)`

We use the 3-bit shift register environment to illustrate the control acquisition architecture. Initially, an unconstrained SLUG model of this environment was constructed with the techniques described in Chapter 3. Next, using the units of SLUG as inputs to an adaptive critic network, the critic was trained to indicate the minimal number of time steps to achieve the bit pattern `101` from any possible initial bit pattern. Once trained, the control network implicitly defines a *control law* that specifies the shortest (or equivalently minimal time) action sequence required to achieve the goal state.

4.2 Simulation Results

The control acquisition architecture was applied to the environments listed in Table 4.1. In all the environments, goal states were chosen to be indistinguishable from other states based only on the sensations. For example, to be considered in the goal state in the car radio environment, the robot must have the rock station stored in preset X, the news station stored in preset Y, and must be listening to the classical station. In the $n \times n$ checkerboard environment, the goal location is a particular square that does not contain a landmark and that is located in the middle of the grid. The coordinates given in Table 4.1 are relative to the bottom left square `(0,0)`.

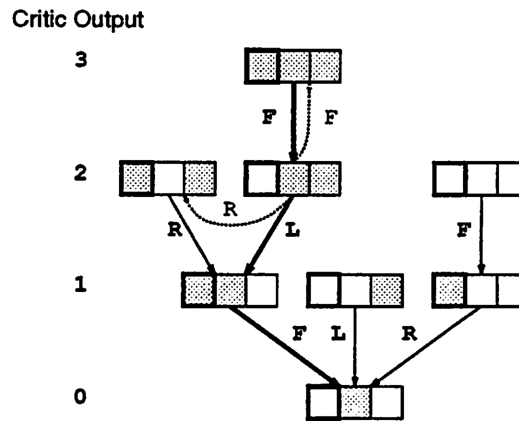

Figure 4.2. Summary of the Control Law and Evaluation Function for the 3-Bit Shift Register Environment. Each state is depicted as a 3-bit shift register where grey denotes a value of 0 in a particular cell, white a value of 1. Arrows indicate some of the possible transitions from one state to another. Solid arrows correspond to the optimal actions; dashed arrows correspond to suboptimal actions.

Figure 4.2 summarizes the resulting control law and evaluation function learned for the 3-bit shift register environment. All states of the shift register are shown, including the goal state, 101, which is at the bottom of the figure. Each state is depicted as a shift register with the corresponding bit settings, where grey denotes a value of zero, and white denotes a value of one. The states are depicted in decreasing order from top to bottom in terms of the output of the adaptive critic for that state (i.e., expected minimum number of time steps needed to reach the goal state). This figure summarizes the implicit control law by showing the optimal transitions from state to state. For example, suppose the robot is in state 100 and is going to choose the best next action. The robot could choose action L resulting in state 001 with value 1, or action R resulting in state 010 with value 2, or action F resulting in state 000 with value 3. The best action available to the robot, according to the model and adaptive critic, is L which is shown with the solid line; the suboptimal actions (F and R) are shown by dotted lines in Figure 4.2.

Table 4.2. An Example State/Action Sequence for the Car Radio Environment.

State			action
current	preset X	preset Y	
classical	classical	classical	scan left
news	classical	classical	store in preset Y
news	classical	news	scan left
rock	classical	news	store in preset X
rock	rock	news	scan left
classical	rock	news	—

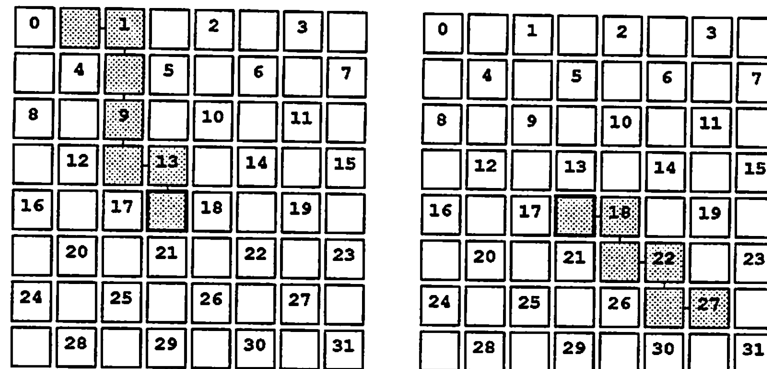

Figure 4.3. Two Example Trajectories for the 8×8 Checkerboard Environment. Each trajectory is shown as a sequence of shaded squares connected by line segments and ending in the goal square.

During non-exploratory steps, the robot chooses the estimated optimal action given the current state. For example, if the shift register starts in state `000`, the optimal action sequence is `FLF`. This trajectory is depicted in Figure 4.2 as a sequence of bold arrows connecting state `000` to state `101` via states `100` and `001`.

As a further demonstration of the control network, we describe the behavior of the network on the car radio and 8×8 checkerboard environments after training. In the car radio environment, the robot learns to store the proper stations in the presets and then to play classical music. One example sequence of state/action pairs is shown in Table 4.2.

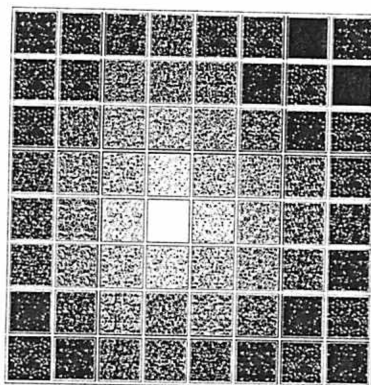

Figure 4.4. The Output of the Adaptive Critic at Each of the Squares in the 8×8 Checkerboard Environment. The evaluations are portrayed with white being the smallest time to goal and black being the largest time to goal.

In the 8×8 checkerboard environment, the goal state was chosen to be square `(3,3)`. The resulting controller is able to choose optimal actions from each square in the grid, i.e., actions that move it one step closer to the goal position. Figure 4.3 shows two trajectories for this environment, and Figure 4.4 shows the relative output of the adaptive critic in all the squares. The evaluations decrease steadily to zero at the goal square `(3,3)`.

To evaluate the control acquisition architecture, we examined its performance as measured by the total number of actions that must be executed before the system produces optimal goal-seeking behavior. The total number of actions include wandering actions, exploration actions, as well as actual control actions during which learning occurs. The system was considered to have learned a task by a given time step if the optimal control actions are taken for at least the next 2500 actions. In order to judge whether a control action was optimal, a table of the minimum number of time steps needed to reach home for each state was constructed prior to training.[3]

[3]One might wonder why we need an adaptive critic if we can construct a table. First, an adaptive critic permits the learning of minimal length sequences of actions in nonstationary environments. Secondly, an adaptive critic permits the learning of minimal length sequences of actions for larger environments for which constructing a table is impractical.

Table 4.3. Performance of the Control Acquisition Architecture.

Environment	# of Env. States	# of Hidden Units	Learning Rate	Median Number Needed to Learn Optimal Behavior		
				Actions		Trials
				All	Control	
Little Prince	4	0	0.100	20	8	2
3-Bit Shift Register	8	8	0.050	4443	778	373
4-Bit Shift Register	16	16	0.020	7662	1937	282
Car Radio	27	16	0.010	12512	2199	355
4×4 Checkerboard	16	0	0.001	6061	1114	262
6×6 Checkerboard	36	0	0.001	57915	5692	1327
8×8 Checkerboard	64	0	0.001	106507	8777	1446

An action that changes the environmental state from x to y was considered optimal if the minimum number of time steps needed to reach home from state y is one less than the minimum number needed from state x. For each environment. several runs were made to determine the learning rate for the adaptive critic that yielded the best performance. Because results varied from one run to another due to the random initial weights, we report the median performance over 25 replications of each simulation. On all replications, the control network was able to converge to an optimal solution.

Table 4.3 reports the performance of the control acquisition architecture for the environments listed in Table 4.1. Table 4.3 lists the number of environmental states, the number of logistic hidden units contained in the adaptive critic network (where 0 in this column means that a hidden layer was unnecessary), and the learning rate used for training the adaptive critic. Table 4.3 also reports the median total number of actions, number of control actions, and number of trials needed to learn to produce optimal behavior. Because the control network was only trained during control actions, the number of control actions equals the number of weight updates. The results of Table 4.3 do not include the time needed to train the model.

Because of the large number of wandering and exploratory actions, the ratio of the number of control actions to the total number of actions is small. More

sophisticated wandering and exploration procedures would substantially increase this ratio. Figure 4.5 shows some learning curves for the 4-bit shift register environment. The curves demonstrate that the system rapidly learns the majority of the control law and then takes significantly longer to learn the residual behavior.

4.3 Relation to Other Work

In this section only a small sample of the connectionist control literature is covered. Consult Barto [6] and Narendra [61] for more comprehensive overviews of this subject. In this section. the focus is on the methods utilized in our control acquisition architecture.

Supervised learning requires an informative teacher that can provide appropriate target values. Unfortunately. in control tasks the environment can rarely give the desired control signals. Jordan and Rumelhart [37] describe a technique that permits supervised learning to be applied in these cases. The technique, called *forward modeling*, involves building a model of the environment. and it allows the controller to be optimized in terms of a performance criterion defined in distal coordinates.

Consider the problem of learning the inverse kinematics of a multi-joint robot arm with excess degrees of freedom (as described in Jordan and Rumelhart [37]). The inverse kinematics is a mapping from a spatial position vector $\boldsymbol{x}$ to the joint angles $\boldsymbol{\theta}$ necessary to move the endpoint of the arm to that position. Learning the inverse kinematics directly is difficult when there are excess degrees of freedom because there are many joint angles that result in the same position. Standard function approximation techniques would tend to learn the average of the joint angles for a given position. Unfortunately, the average might not be a valid solution. As an additional difficulty, the teaching signal provides desired positions and not desired joint angles. Therefore, the learning algorithm must somehow convert position errors into joint angle errors.

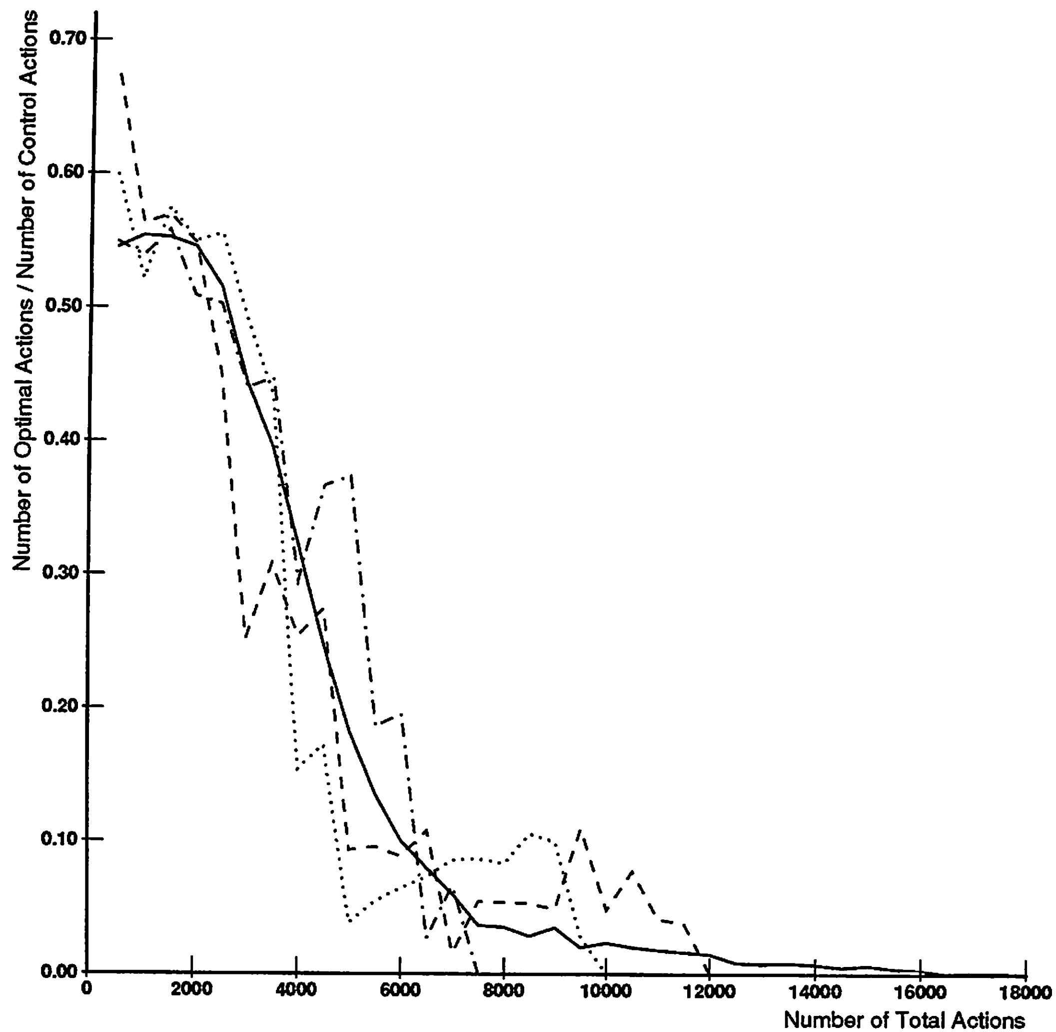

Figure 4.5. Learning Curves for the 4-Bit Shift Register Environment. The y-axis shows the fraction of suboptimal actions chosen during the past 500 time steps. The bold line shows the average performance over time for the control network over 25 replications. The dotted, dashed, and stippled lines show the performance over time for three different individual runs.

Jordan and Rumelhart address both of these issues. Their learning system is composed of a forward kinematics network and an inverse kinematics network. The inputs to the forward kinematics network are the joint angles $\boldsymbol{\theta}$, and the output is the position of the endpoint of the arm $\boldsymbol{x}$. The input to the inverse kinematics network is the position of the desired endpoint of the arm $\boldsymbol{x}$, and the outputs are the joint angles $\boldsymbol{\theta}$. Although the forward modeling technique will work with a variety of learning algorithms, we restrict our attention to the forward modeling method using back-propagation. In this case, the method involves differentiating the forward kinematics network in order to translate errors in cartesian space to errors in joint angles in order to provide training information for the inverse kinematics network. Their approach starts with a modeling phase, where the model is trained with random joint angle inputs and the resulting positions as targets. Once a reasonably accurate forward kinematics model has been learned, the output of the inverse kinematics model is connected to the input of the forward kinematics model, and the inverse kinematics model is trained by back-propagating through the forward kinematics model into the inverse kinematics model.

This technique, called *differentiating a model* by Barto [6], has some advantages over various other control acquisition approaches. Jordan and Rumelhart [35] note that this technique can identify a controller even if the controller is not well-defined (i.e., excess degrees of freedom). This approach has the potential to simultaneously differentiate the model and adjust the parameters of the model and controller. Furthermore, the derivative of the model's output with respect to its input can be efficiently computed by the back-propagation learning algorithm.

Our control acquisition architecture is related to an architecture called the *3-net* architecture that uses the idea of differentiating a model to obtain training information. The 3-net architecture shown in Figure 4.6 was proposed by Werbos [103]. The 3-net architecture consists of a controller, a model, and an adaptive critic. The controller maps states to actions, the model maps state/action pairs to next

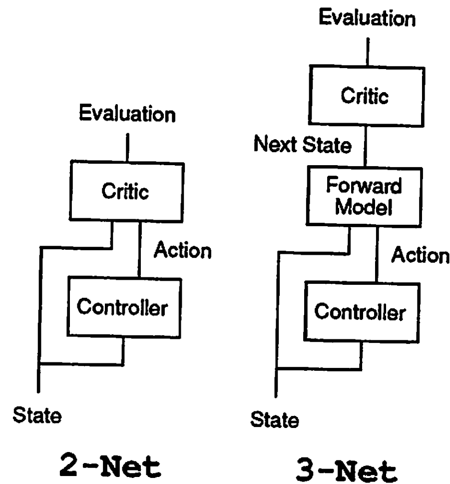

Figure 4.6. Control Architectures.

states, and the adaptive critic maps states to evaluations. The basic idea is to learn an evaluation function and then train the controller by differentiating this function with respect to the controller weights. These derivatives indicate how to change the controller's weights in order to minimize or maximize the evaluation function.

Our control acquisition architecture is a version of a 3-net architecture without an explicit controller. Instead of training a controller, our technique chooses actions based solely on the output of the adaptive critic; our technique does not involve an explicit representation of the control function. In the 3-net architecture the controller and adaptive critic are intimately related: the controller must adapt to changes in the adaptive critic and vice-versa. In contrast, in our architecture this relationship is simplified in that the adaptive critic directly defines the actions. In particular, the actions are chosen according to the current evaluation function—the action with the best evaluation is chosen.

It is useful to compare our architecture to the 2-net architecture shown in Figure 4.6 as proposed by Werbos [103] and Jordan and Jacobs [36]. The 2-net architecture consists of a controller and an action-dependent adaptive critic. The controller maps states to actions, and the action-dependent adaptive critic maps state/action

pairs to evaluations. Jordan and Jacobs [36] show how to train a pole balancer using the method of differentiating the adaptive critic in a 2-net architecture. The adaptive critic learns to predict the reciprocal of time-until-failure given the current state and the controller's action. Given the current state as input, the controller produces a real-valued action that is interpreted as a force on the cart. The adaptive critic should output zero when the controller is performing as desired because in this case the time-until-failure should be infinite. Therefore, the desired output of the adaptive critic is zero, and the difference between the actual adaptive critic output and zero gives a distal error. This error can then be used to train the controller by back-propagating it through the weights of the adaptive critic, through the action output, and into the controller. In this way the controller can be changed so as to maximize the time until failure.

It has been said that it is easier to train a 2-net architecture than a 3-net architecture because there is no model (see Mahadevan and Connell [51]). One problem with building a model is that it is not sensitive to the particular control task. [4] Some features of the environment might be unrelated to the control task, and any technique that employs a model would have to spend a disproportionate amount of time learning about aspects of the environment that are irrelevant to the control task. With a 2-net architecture, an action-dependent adaptive critic is trained based on state/action input pairs, and a model is not required. But what if a model already exists, or if prior knowledge exists from which a model can be constructed off-line? Knowledge of relations between actions and their effects can be hardwired into the weights of a network environment model. In SLUG, the task of coding this knowledge is simplified because of the explicit representation of the effects of actions on a model's state as separate weight matrices. For example, symmetries in actions (such as action inverses) can be coded into a model. In the extreme case, the entire

[4] On the other hand, building an environment model might be advantageous because a model could be used for many different tasks.

model can be hardwired as illustrated in Chapter 5. Furthermore, if this is possible, it can significantly accelerate up the overall learning process because the adaptive critic learns an evaluation function based only on states and not state/action pairs as in the action-dependent adaptive critic.

Barto, Sutton, and Anderson [9] discuss a direct reinforcement method for control acquisition where the teacher provides only a reinforcement signal. The reinforcement signal indicates how well the controller is performing and not how to change the control signal. This is different than supervised learning, where the teacher provides a desired control signal. Barto, Sutton, and Anderson address the problem of learning to balance an inverted pendulum on a moving cart, where the reinforcement signal only indicates failure. This delayed reinforcement problem is particularly difficult because the consequences of the actions reveal themselves over time and interact with the consequences of other actions. This problem is solved by constructing an adaptive critic that learns to predict the discounted sum of all future penalties, where a penalty occurs only upon failure. The discounting gives greater weight to near-term penalties than to temporally distant penalties. The adaptive critic produces a reinforcement signal more informative than the reinforcement signal provided by the teacher. This new reinforcement signal gives the controller immediate feedback about the long-term consequences of its current action. In contrast to the technique of differentiating a model, the controller is trained using a simple reinforcement learning algorithm that uses the adaptive critic's output as the reinforcement signal.

Sutton [95] further develops the theory of learning algorithms appropriate for the training of adaptive critics. This class of algorithms, called temporal difference (henceforth referred to as TD) methods, address the problem of developing predictions in a delayed reinforcement setting. TD algorithms work well in multi-step prediction paradigms, where new information comes in at each step leading up to the outcome.

Sutton and Pinette [97] present a technique for learning a Markov model of stochastic maze environments with a connectionist network. This model is then

used to learn the best path through a maze. The solution to the maze problem is learned much more quickly after initially learning a model of the maze. Their learning algorithm is based on the TD learning algorithm and produces a model that predicts the discounted expectation of the future sensations. Discounting gives greater weight to near-term sensations than to temporally distant sensations. The network is a fully connected recurrent network of linear units. These units encode the state, and unlike in the tasks considered in this thesis, the state is completely observable at each time step. The network is trained to construct a weight matrix that matches the environment's transition probability matrix. This network is used to make predictions by running the network forward until the predictions stabilize.

Barto, Sutton, and Watkins [10] (see also Watkins [99]) discuss the relationship between adaptive critic control techniques and stochastic dynamic programming and parameter estimation. The dynamic programming framework formalizes the adaptive critic technique (using TD methods) and suggests other learning methods.

4.4 Discussion of Other Possible Approaches

Our architecture provides the possibility of learning a model and control law simultaneously (although this technique is currently unimplemented in our system). This would allow the robot to learn in dynamic environments and would permit the use of a model for performing "mental experiments" as demonstrated by Sutton [96]. Sutton shows how a model can dramatically speed up the control acquisition process.

A variation of our control architecture can be used in environments where learning a model would be very time consuming. All components of the architecture remain the same, but the model would no longer be required to produce predictions of the sensations. Instead of predicting the next sensation values, the model would produce state information used by the adaptive critic to form an evaluation function. Consequently, the model would produce only state information necessary for the control task. In this case, the combination of an action-independent adaptive critic

and model form an action-dependent adaptive critic. Judging from the superior performance of unconstrained SLUG over a "conventional" connectionist architecture, we would expect that unconstrained SLUG used in conjunction with an adaptive critic network would outperform one large, undifferentiated connectionist 2-net architecture (such as an Elman network).

In connectionist control acquisition, there are many viable alternative architectures and training procedures. However, at some level, all approaches require some sort of model of the environment in order to predict the consequences of actions. An explicit environmental model is especially important in environments whose state cannot be directly perceived. Thus a flexible, powerful model—such as the model constructed by SLUG—is an important component of a goal-seeking system operating in complex environments.

CHAPTER 5

NAVIGATION

This chapter presents a connectionist control acquisition architecture tailored for simulated short-range homing in the presence of obstacles. The kinematics of a cylindrical robot (Figure 5.1) moving in a planar environment are simulated. The robot has wheels that propel it independently and simultaneously in both the x and y directions with respect to a fixed orientation. It can move up to one radius per discrete time step. Each obstacle in the robot's environment has an associated real-valued grey level between 0 and 1 representing the darkness of the grey: the higher the value, the darker the grey, with 0 corresponding to white and 1 corresponding to black. The robot has a 360 degree sensor belt with 64 distance sensors and 64 grey-scale sensors evenly placed around its perimeter. These 128 values are smoothed in a preprocessing step to produce 32 values that form the robot's *view*.

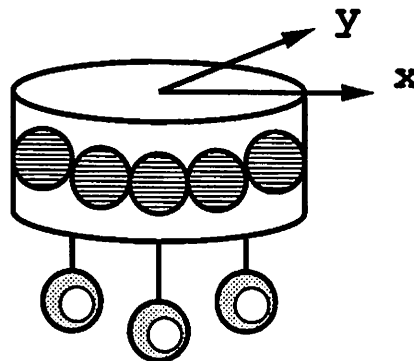

Figure 5.1. The Simulated Robot.

Figure 5.2 is a display created by the navigation simulator. The bottom portion of the figure shows a bird's-eye view of the robot's environment. In this display, the bold circle represents the robot's "home" position, with the radius line indicating the fixed home orientation. The other circle with radius line represents the robot's current position and fixed orientation. The top panel shows the grey-scale view from the home position, and the next panel down shows the grey-scale view from the robot's current position. For the reader's better viewing, the distance and grey-scale sensor values are shown on the same panel, where the height of the profile is the reciprocal of the distance. Thus as the robot gets closer to objects they get taller in this display, and when the robot gets farther away from objects they get shorter.

The robot can neither move nor "see" through obstacles (i.e., obstacles are opaque). The task is for the robot to align itself with the home position from arbitrary starting positions in the environment while not colliding with obstacles. This task is performed using only the sensory information—the robot does not have access to the bird's-eye view.

This is a difficult control task. Although the sensory information provides complete state information, it forms a high-dimensional continuous task, and successful homing generally requires a nonlinear mapping from this space to the space of real-valued actions. Further, it is not easy to train networks on this space. The robot has no comprehensive world model and receives no global information; it receives only sensory information that is inherently limited. Furthermore, because it is difficult to reach home using random exploration due to the large number of environmental states, simple trial-and-error learning is intractable. In order to solve this task an architecture was designed that facilitates the coding of generic domain knowledge in the form of hard-wired networks, architectural constraints, and initial weights.

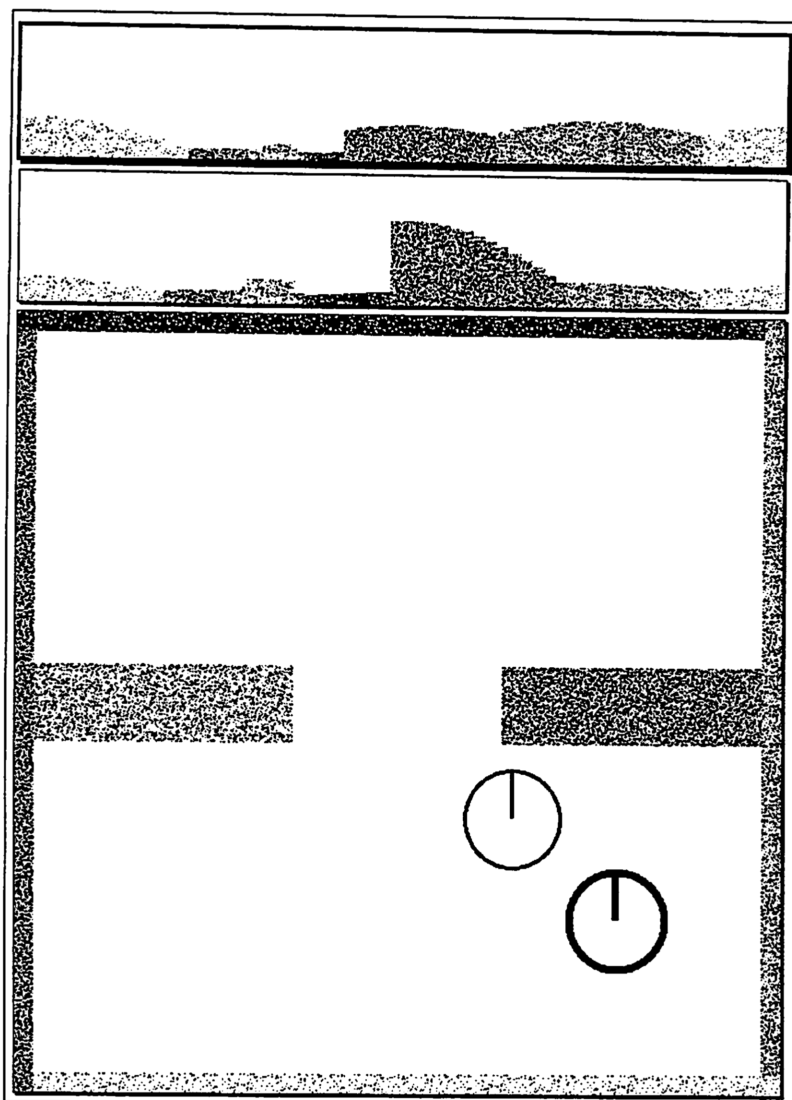

Figure 5.2. The Navigation Simulator.

5.1 Artificial Potential Functions

Before we describe our architecture, we briefly discuss a more traditional technique for navigation that uses artificial potential functions (e.g., Connolly, Burns, and Weiss [15], Khatib [40], Koditschek [43], Krogh [45], LaTombe [47], and Koditschek and Rimon [71]). This technique involves building explicit object models representing the extent and position of objects in the robot's environment. Repelling potential fields are then placed around obstacles using the object models, and the center of an attracting potential field is placed on the goal. This can be visualized as a terrain where the global minimum is located at the goal, and where there are bumps around the obstacles. The height at each point in this terrain is the sum of all the potential functions and will be called the *navigation function* [71]. The robot goes home by descending the terrain.

The contour diagram in Figure 5.3 shows such a terrain. The task is to go from the top room to the home position in the bottom room via the door. Unfortunately, there can be local minima. [1] In this environment there are two prime examples of minima: the right-hand wall between the home location and the upper room, where opposing forces exactly counteract each other to produce a local minimum in the right-hand side of the upper room, and the doorway, where the repelling fields on the door frame create an insurmountable bump in the center of the door as well as a local minimum just above the doorway.

In contrast this traditional technique, our technique involves creating a sensor-based potential function model through learning. Instead of hard-wiring static potential functions from the object models, the proposed architecture learns potential functions, automatically adjusting them to both avoid local minima and to produce

---

[1] Rimon and Koditschek [71] present a technique that produces a navigation function with a single minimum for a restricted class of environments. Connolly, Burns, and Weiss [15] present a technique for computing a navigation function with a single local minimum. This navigation function can be used for planning smooth robot paths and their technique works for all environments. Like the other artificial potential function techniques, both of these techniques assume explicit object models.

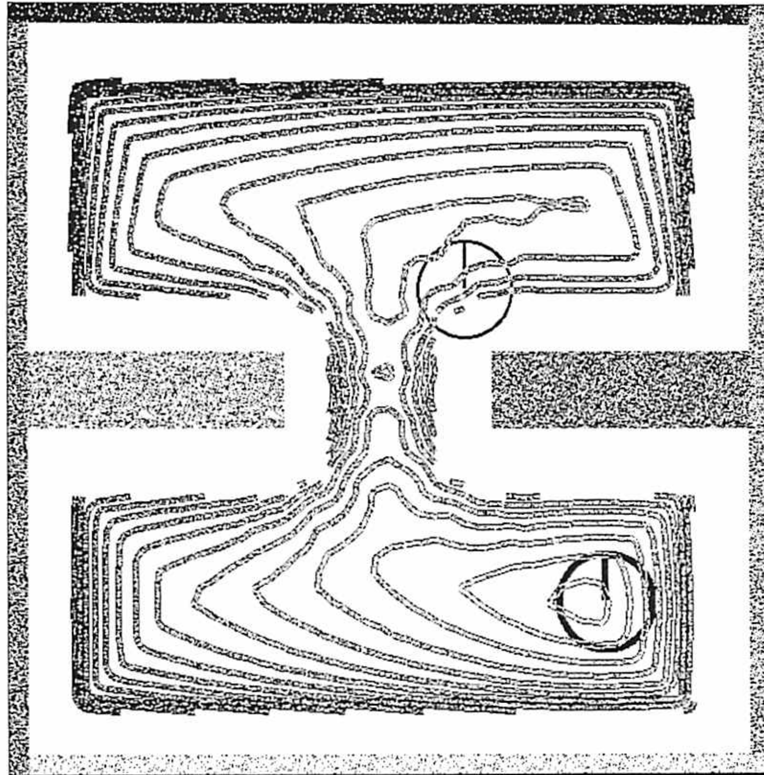

Figure 5.3. The Artificial Potential Function Method. This figure shows a contour plot of a terrain created using artificial potential functions generated from object models. The contour diagram shows level curves where the grey level of the line depicts the height of the line: the maximum height is depicted in black, and the minimum height is depicted in white.

efficient trajectories. Furthermore, it does this without object models, using only sensory information.

5.2 The Navigation Architecture

The navigation architecture is a version of our control acquisition architecture (described in Chapter 4) tailored for navigation, where the state is the robot's view and the evaluation is an estimate of the length of the shortest path from the robot's current location to home. The basic architecture is shown in Figure 5.4 and will be called the *navigation network*. It consists of a *range-flow model* and two adaptive critics. The range-flow model maps view/action pairs to next views; the *homing critic* maps views to path length home using a straight line trajectory; the *obstacle-avoidance critic* maps views to estimated minimum additional path length needed to avoid obstacles. The sum of the outputs of the homing critic and the obstacle-avoidance critic equals the total estimated shortest path length home. The range-flow model is a hard-wired differentiable network incorporating geometric knowledge about the sensors and space. Each critic is a radial basis network using Gaussian hidden units.

The controller is implicit in this architecture. At each time step the action is chosen that locally minimizes the sum of the outputs of the critics. Because the actions are real-valued, it is intractable to present all possible actions in turn as input to the navigation network. Instead, gradient descent is used to find a feasible action $(\Delta x, \Delta y)$ that minimizes the output of the sum of the current outputs of the critics, where feasible means that the action has length less than the radius of the robot. This action—the one that is the first action in a sequence of actions that yields the minimum path length needed to reach the goal state according to the current critics' estimates—is then executed.

Before describing the range-flow model and the critics, we first describe the robot's view in more detail. The view is constructed using 64 distance sensors

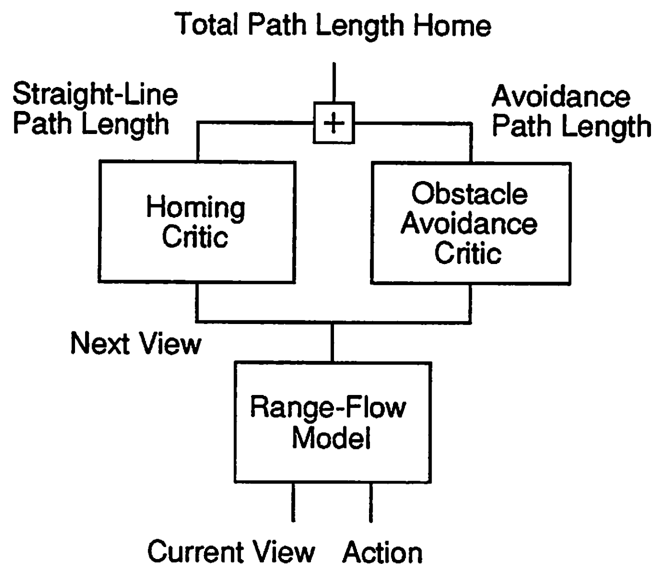

Figure 5.4. The Navigation Architecture.

and 64 grey-scale sensors evenly placed around the robot's perimeter. The values on these 128 sensors are smoothed in a preprocessing step to create 32 values that comprise the robot's view. We denote the 64 distance measurements and the 64 grey-scale measurements by $(\breve{\boldsymbol{d}}, \breve{\boldsymbol{g}}, \breve{\boldsymbol{\theta}})$, where $\breve{\boldsymbol{d}}$ is the vector of distance measurements, $\breve{\boldsymbol{g}}$ is the vector of grey-scale measurements, and $\breve{\boldsymbol{\theta}}$ is the vector of fixed sensor angles, with $\breve{\theta}_k = \frac{2k\pi}{64}$, $k = 0, \ldots, 63$. Each distance measurement $\breve{d}_k$ is the distance to the closest obstacle in the direction of its respective sensor angle $\breve{\theta}_k$, and each grey-scale measurement $\breve{g}_k$ is the grey-value of the closest obstacle in the direction of its respective sensor angle $\breve{\theta}_k$. We denote the 32 smoothed values by $(\boldsymbol{d}, \boldsymbol{g}, \boldsymbol{\theta})$, where $\boldsymbol{d}$ is a vector of smoothed distance values, $\boldsymbol{g}$ is a vector of smoothed distance values, and $\boldsymbol{\theta}$ is the vector of angles corresponding to these smoothed values, where $\theta_k = \frac{2k\pi}{16}$, $k = 0, \ldots, 15$.

The sensor measurements are smoothed with a bank of 16 Gaussian smoothing units for each sensory modality (i.e., distance and grey-scale). The jth Gaussian smoothing unit has a receptive field represented by a Gaussian distribution with

mean θ_j and fixed variance σ. The variance is such that the Gaussian distribution has value .5 at the midpoint between two neighboring angles θ_k and θ_{k+1}. This produces overlapping receptive fields and gives good smoothing performance in practice. The jth Gaussian distance smoothing unit weights each distance measurement $\breve{d}_i$ by the extent to which the corresponding angle $\breve{\theta}_i$ falls in its receptive field, where the extent is calculated with a Gaussian function as shown below. Finally, this Gaussian distance smoothing unit computes its output by summing these weighted measurements and then normalizing by the total amount of weight contributed by these measurements:

$$d_j = \frac{\sum_i \breve{d}_i \epsilon^{-(\theta_j - \breve{\theta}_i)^2/\sigma}}{\sum_i \epsilon^{-(\theta_j - \breve{\theta}_i)^2/\sigma}}. \tag{5.1}$$

Grey-scale smoothing proceeds in exactly the same fashion. Figure 5.5 shows this smoothing process for the case of 16 distance sensors and 4 Gaussian distance smoothing units.

For the purpose of this description, it is easiest to think of these 32 smoothed values as corresponding to 32 virtual sensors that have overlapping Gaussian receptive fields. In order to simplify the description, throughout the rest of the paper we treat these virtual sensors as the robot's real sensors.

The hard-wired range-flow model predicts the next view given an action and the current view. The range-flow model forms its prediction in two steps: change of coordinates followed by interpolation. The distance values of the current view and their respective angles $(\boldsymbol{d}, \boldsymbol{\theta})$ define a set of points specified in polar coordinates on obstacles in the robot's environment relative to the robot's own body-centered coordinate system. This is shown in Figure 5.6. When the robot moves, it can recalculate these points $(\boldsymbol{d}, \boldsymbol{\theta})$ with respect to its new coordinate system giving $(\boldsymbol{d}', \boldsymbol{\theta}')$. This step is simply a change of coordinates. But to predict the sensor values, the robot needs to predict the distances corresponding to the sensors' fixed angles $\boldsymbol{\theta}$ instead of $\boldsymbol{\theta}'$. To do this, it interpolates the $(\boldsymbol{d}', \boldsymbol{\theta}')$ points to form distance predictions aligned with the original angles $\boldsymbol{\theta}$, resulting in $(\hat{\boldsymbol{d}}, \boldsymbol{\theta})$. This process is shown in Figure 5.6. The

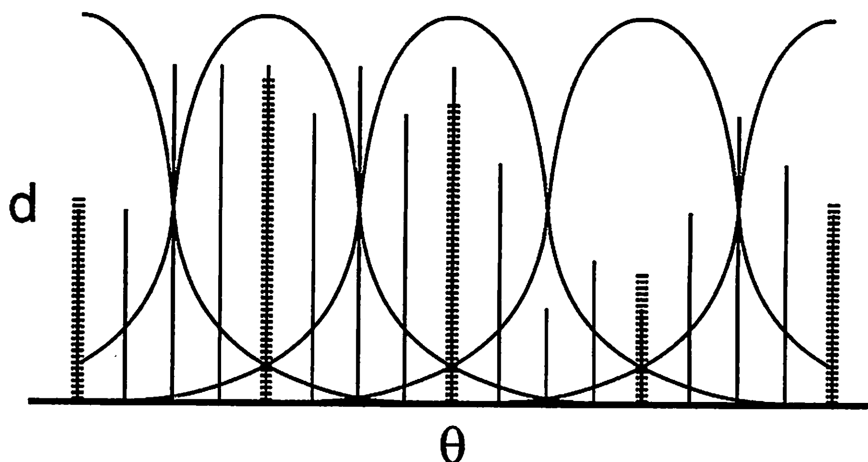

Figure 5.5. The Sensor Value Smoothing Process. It is shown for the case of 16 distance measurements that are used to produce 4 smoothed values. All sensor values are shown in polar coordinates using a bar-chart plotting style, where the top of each line defines its polar coordinates. The sensor measurements are represented as solid lines, and the smoothed values are represented as dashed lines. Because of wrap-around, the leftmost and rightmost sensor values are the same.

θ' sensor angles are also used to interpolate the grey-scale values g to form grey-scale predictions $\hat{g}$.

It is important to distinguish our range-flow model from a model of the robot's environment and kinematics. A range-flow model merely predicts how the observable range data (i.e., points measured on obstacles in the robot's environment) will be transformed upon the execution of a proposed action. In this fashion, it is independent of any particular environment, and therefore does not have to be retrained on new environments. Unfortunately, this generality has its price: the range-flow model is inexact. There are two major sources of errors in a range-flow model: approximation error due to interpolation, and prediction error due to environmental influences. In contrast to an environment-specific model, the range-flow model cannot handle certain environmental influences. In particular, the range-flow model cannot predict the effects of infeasible actions or the emergence into view of hidden objects; in general, it cannot cope with discontinuities in the environment. If we assume that environ-

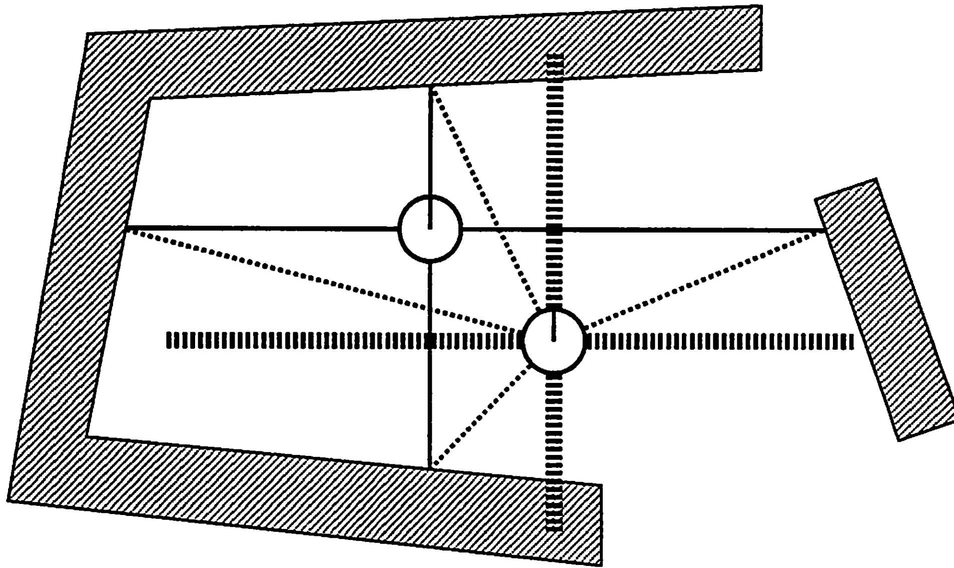

Figure 5.6. The Operation Of The Range-Flow Model. The distance values of the robot's view are shown as a set of vectors to points on objects in the robot's environment. The left robot represents the robot before a translation, and the right robot represents the robot after a translation. The robot has four evenly spaced distance sensors. (Note that the actual robot has more sensors.) The solid rays emanating from the left robot's center represent vectors to the closest points on obstacles along the sensors' angles. The narrow dashed rays emanating from the right robot's center represent vectors to those same points but with respect to the robot's new body-centered coordinate system. The thick dashed rays emanating from the right robot's center show the interpolated distance predictions along the sensors' angles. Although there will always be prediction errors, the ones depicted in this example are much larger than would be expected in practice, where there would be many more sensors, and the actions would be much smaller.

ments are continuous almost everywhere, then we can bound the approximation and prediction errors. In particular, the range-flow model's accuracy increases with an increased number of sensors, and its accuracy decreases with an increase in the size of the robot's movement.

The connectionist range-flow model consists of a set of hard-wired units that perform the translation and interpolation as shown in Figure 5.7. [2] The translation units simultaneously transform the input view from polar to rectangular coordinates and calculate the translation. In particular, the x translation units compute their outputs as follows:

$$x'_j = d_j \cos\theta_j - \Delta x, \tag{5.2}$$

where d_j is the jth distance measurement, θ_j is the sensor angle corresponding to d_j, and Δx is the x translation. The y translation units compute their outputs as follows:

$$y'_j = d_j \sin\theta_j - \Delta y, \tag{5.3}$$

where Δy is the y translation. The rectangular-to-polar units compute their outputs as follows:

$$d'_j = \sqrt{x'^2_j + y'^2_j}, \tag{5.4}$$

and

$$\theta'_j = \tan^{-1}(x'_j, y'_j). \tag{5.5}$$

Interpolation calculations are performed by a series of Gaussian interpolation units, one per sensor. Gaussian interpolation was chosen because it is simple and differentiable. The jth Gaussian interpolation unit has a receptive field, by a Gaussian distribution with mean θ_i and fixed variance σ. The variance is set such that the Gaussian distribution has height .5 at the midpoint between two neighboring

[2] Although the transformation could be calculated without the use of a network, in using a network, the model can be easily integrated into the entire navigation network. Furthermore, back-propagation can be employed in order to efficiently compute derivatives.

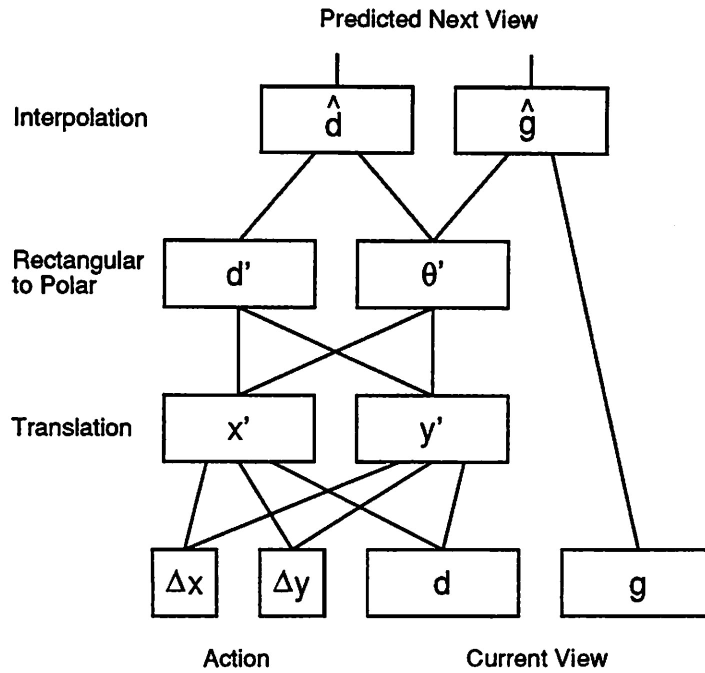

Figure 5.7. The Range-Flow Model Network. In this figure, $\boldsymbol{d}$ is a vector of distance measurements, $\boldsymbol{g}$ is a vector of grey-scale measurements, $(\boldsymbol{x}', \boldsymbol{y}')$ are the translated points in cartesian coordinates, $(\boldsymbol{d}', \boldsymbol{\theta}')$ are the translated points in polar coordinates, and $\hat{\boldsymbol{d}}$ and $\hat{\boldsymbol{g}}$ are respectively the interpolated distance and grey-scale values aligned with the sensor angles $\boldsymbol{\theta}$.

sensor angles θ_k and θ_{k+1}. This produces overlapping receptive fields and gives good interpolation performance in practice.

The range-flow model calculates the points $(\boldsymbol{d}', \boldsymbol{\theta}')$ on objects in the robot's environment with respect to the robot's body-centered coordinate system after a proposed translation. The jth Gaussian interpolation unit takes each of these points (d'_i, θ'_i) and weights each distance value d'_i by the extent to which the angle θ'_i falls in its receptive field, where the extent is calculated with a Gaussian function as shown below. Finally, a Gaussian interpolation unit computes its output by summing these weighted distances and then normalizing by the total amount of weight contributed by these points:

$$\hat{d}_j = \frac{\sum_i d'_i e^{-(\theta_j - \theta'_i)^2/\sigma}}{\sum_i e^{-(\theta_j - \theta'_i)^2/\sigma}}. \tag{5.6}$$

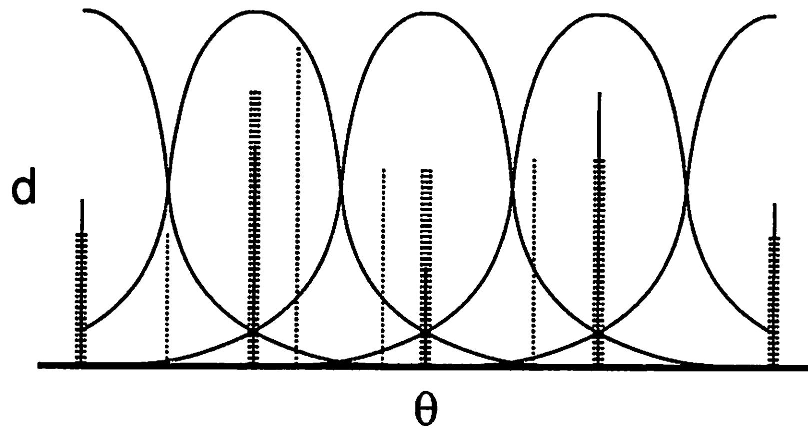

Figure 5.8. Gaussian Interpolation. The rays from Figure 5.6 are shown in polar coordinates, with angle on the abscissa and distance on the ordinate. The tops of the lines represents the end points of the vectors. The robot has four evenly spaced distance sensors. The solid lines represent the closest points on obstacles along the sensors' angles. The narrow lines represent those same points after a change of coordinates. The thick dashed lines represent the interpolated distance predictions along the sensors' angles. In this case, four Gaussians interpolation units perform the interpolation step. These units are represented by four Gaussian distributions evenly spaced across the abscissa. Because of wrap-around, the leftmost prediction is the same as the rightmost. The leftmost Gaussian distribution shows the case of a single unaligned point (where the other points' contribution is negligible). In this case, the interpolation scheme aligns the point's angle with the sensor's angle while maintaining the original distance. The second Gaussian distribution from the left shows the case of multiple unaligned points. In this case, Gaussian interpolation produces a distance prediction that is a weighted average of all the points.

This calculation handles both the case of unaligned angles and the case of multiple θ_i''s mapping to a particular θ_j.

The basic operation of these units is shown in Figure 5.8. The Gaussian interpolation process is very similar to the Gaussian smoothing process. The difference between the two processes is that smoothing reduces the dimensionality of the input and has fixed input angles, whereas the interpolation process maintains the dimensionality of the input and has variable input angles. These similarities and differences can be seen by comparing Figures 5.5 and 5.8.

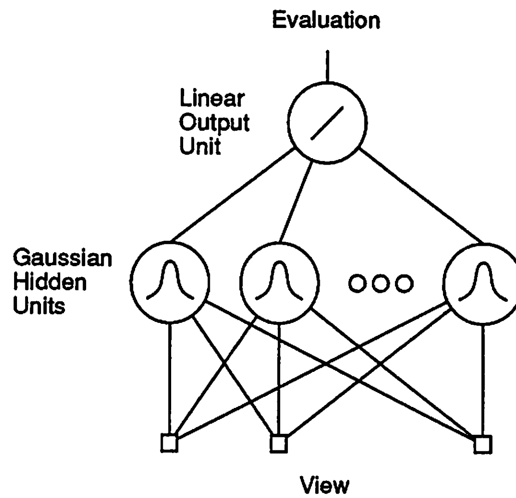

Figure 5.9. A Critic Network.

Each critic network is composed of two layers of units as shown in Figure 5.9. The output layer consists of a single linear unit. The hidden layer consists of a number of Gaussian units which contain vector mean and scalar variance parameters. These units have Gaussian receptive fields and calculate their activities based on the responses of their receptive fields to their inputs:

$$y_j = \epsilon^{\frac{\sum_i (x_{ij} - w_{ij})^2}{\sigma_j}}, \tag{5.7}$$

where y_j is the output of the jth Gaussian unit, x_{ij} is the ith input to the jth Gaussian unit, w_{ij} is the ith weight to the jth Gaussian unit, and σ_j is the variance of the jth Gaussian unit.

5.3 Training

The training process is divided into three major phases. Initially the critics' Gaussian hidden units are trained using an unsupervised learning procedure in order to produce good initial weights that place the Gaussian distributions in regions of view space that are densely populated by views from the particular environment. Next, the homing critic is trained, and finally, while the weights of the homing critic

are fixed, the avoidance critic is trained. Throughout the entire training process the range-flow model remains fixed.

The critic networks' Gaussian hidden units are trained initially using a maximum likelihood technique called *expectation maximization*, which is a soft form of competitive learning (see Duda and Hart [16] and Nowlan [63, 62] for details). The units are trained on a number of views generated by the robot from a uniform distribution of positions in the environment. Expectation maximization assumes a parameterized class of models, in this case an additive mixture of multidimensional spherical Gaussian distributions, and it selects the model from that class that is most likely to have produced the data, in this case the views. In short, this process produces a set of Gaussian units that best cover the view data. Figure 5.10 shows the means and variances of the Gaussian distributions after training.

The Gaussian units turn out to be spatially-tuned, that is, they are most responsive to a spatially local region of the robot's workspace, despite the fact that they are trained only on view data. Figure 5.11 shows the response of a few Gaussian units to views generated from a grid of evenly spaced positions in the door environment shown in Figure 5.2. Each Gaussian unit covers a spatially local region of the robot's workspace. Some reasons this occurs are 1) the view space is smooth, 2) by inspection, there appears to be a one-to-one mapping between views and positions in the robot environments that we consider, and 3) the Gaussian units have local receptive fields. Figure 5.12 shows the sum of the responses of all the Gaussian units to this grid of views and demonstrates that the Gaussian distributions provide good coverage of the robot's workspace. In summary, each position in the robot's workspace activates at least one, and no more than a few, Gaussian units. The set of weights produced by this procedure provides a good starting point for training the critics.

Initially, the navigation network is trained to produce straight-line trajectories to home. The homing critic is trained using *dead-reckoning*. Dead-reckoning is a technique for keeping track of the distance to home by accumulating the incremental

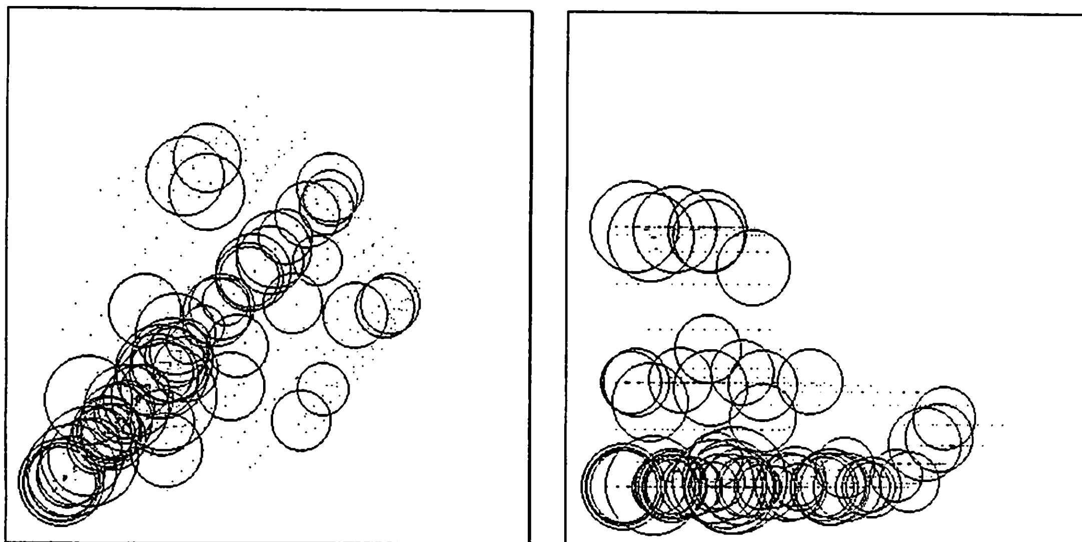

Figure 5.10. The Clustering Phase. The two panels show the Gaussian distributions' coverage of view space using 64 Gaussian distributions plotted on two different two dimensional slices of view space. The view data are depicted as points that were generated at each of the feasible positions of a uniform 50 × 50 grid that was superimposed on the robot's environment, where a feasible position is a location where the robot does not intersect an obstacle. Each Gaussian distribution is indicated by a circle whose center position is the mean and whose radius is the variance. The left panel plots the views and Gaussian distributions using the values of the first two distance sensors (i.e., θ_0 and θ_1) as the abscissa and ordinate respectively. The right panel plots the views and Gaussian distributions using the values on the first two grey-scale sensors as the abscissa and ordinate respectively.

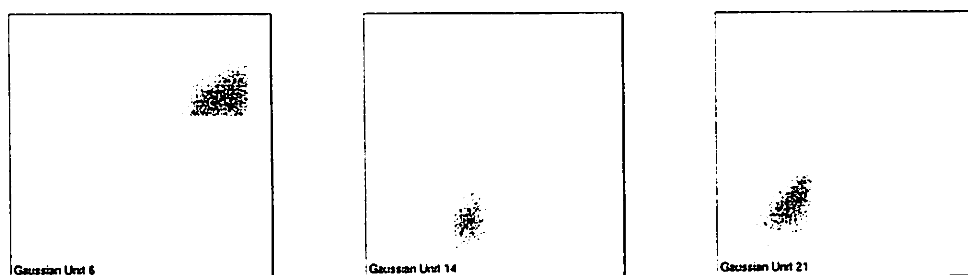

Figure 5.11. The Response of Single Gaussian Hidden Units Across the Robot's Environment. The three panels show the response of single Gaussian units to views generated at each of the feasible positions of a uniform 50 × 50 grid that was superimposed on the robot's environment (shown in Figure 5.3), where a feasible position is a location where the robot does not intersect an obstacle. Furthermore, the responses of all 64 Gaussian units at each spatial position sum to one. The real-valued responses are plotted as grey-values, where the value one is depicted as black, and the value zero is depicted as white.

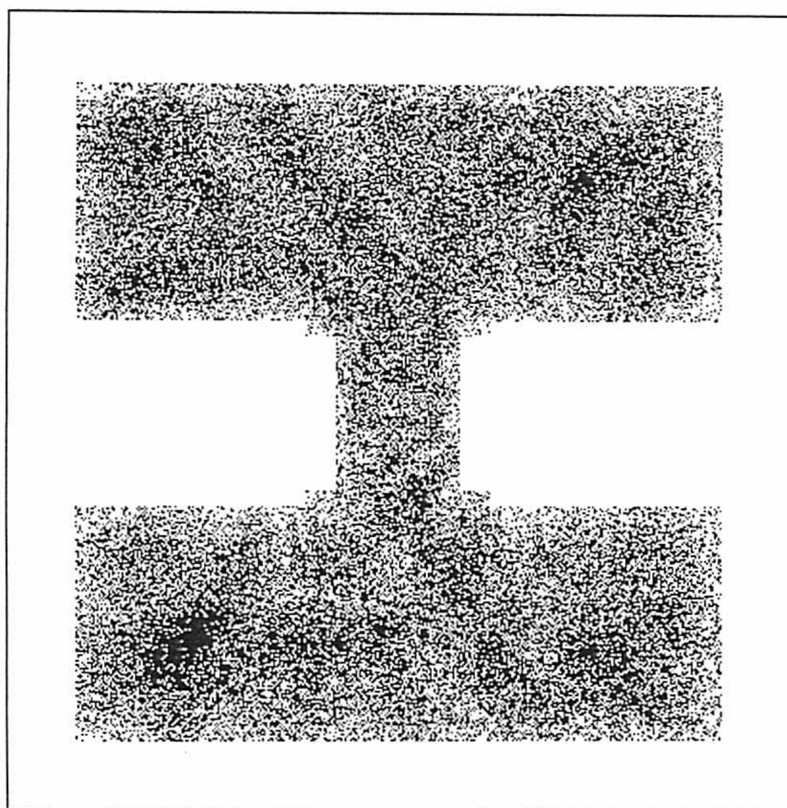

Figure 5.12. The Total Response of All the Gaussian Hidden Units Across the Robot's Environment. Shown are the sums of the responses of all 64 Gaussian units to views generated at each of the positions of a uniform 50×50 grid that was superimposed on the robot's environment (shown in Figure 5.3). To show the actual coverage across the robot's workspace, the responses of all the Gaussian units at each spatial position are not constrained to sum to one. The real-valued responses are plotted as grey-values, where the highest response is black and the lowest response is white.

displacements during exploration from home. This distance provides the target output for training the homing critic via supervised learning. While the robot randomly explores the environment, the weights of the homing critic—the weights of both the output unit and the hidden layer—are adjusted using back-propagation. After training, the output of the critic on a particular view will be the straight-line distance to the home position from the position in the robot's environment that generated that view, and a 3-d surface plot of the output of the homing critic versus position is a cone having a minimum at the home position and increasing with constant slope with increasing distance from the home position.

Next, the navigation network is further trained to avoid obstacles. In this phase, the obstacle-avoidance critic is trained while the weights of the homing critic are frozen. Using a temporal difference (TD) method (Sutton [94, 95] and Barto, Sutton, and Anderson [8]), the obstacle-avoidance critic is adjusted so that the expected path length decreases by one radius per-time-step. After training, the robot takes successive one-radius steps toward its home location.

The training during this phase is divided into a series of trials, each of which begins in the home position and ends when the robot either is inside one radius of the home position or collides with an obstacle. At the beginning of each trial, the robot wanders for a fixed number of steps by taking actions selected randomly from a uniform distribution. During this random walk the robot avoids collisions with obstacles by only executing feasible actions. If an action results in a collision then the action's length is clipped so that the robot stops just short of the obstacle. This random walk places the robot in a random initial state. During the rest of the trial, the robot uses both critics to choose actions resulting in minimal predictions of the path length to reach the home position. Gradient descent is used to choose the best action. The details of the action selection procedure are summarized in Table 5.1.

After each move the obstacle-avoidance critic is trained using a TD method. A system using this method learns to predict by maintaining consistency across

Table 5.1. The Gradient Descent Procedure for Choosing Actions. r is the radius of the robot, η is a learning rate constant, and ϵ is a small constant.

1. Initialize $(\Delta x, \Delta y) = (0, 0)$.
2. Loop
   (a) Feed $(\Delta x, \Delta y)$ and the current view into the network and calculate the predicted total path length to the home position $p(t)$.
   (b) Compute $(-\frac{\partial p(t)}{\partial \Delta x}, -\frac{\partial p(t)}{\partial \Delta y})$ using back-propagation. (This can be computed by back-propagating a -1 into the navigation network.)
   (c) $(\Delta x, \Delta y) \leftarrow (\Delta x - \eta\frac{\partial p(t)}{\partial \Delta x}, \Delta y - \eta\frac{\partial p(t)}{\partial \Delta y})$.
3. Until $|(-\frac{\partial p(t)}{\partial \Delta x}, -\frac{\partial p(t)}{\partial \Delta y})| < \epsilon$ or $|(\Delta x, \Delta y)| > r$.
4. If $|(\Delta x, \Delta y)| > r$ then $(\Delta x, \Delta y) \leftarrow (\frac{r\Delta x}{|(\Delta x, \Delta y)|}, \frac{r\Delta y}{|(\Delta x, \Delta y)|})$. This step ensures the feasibility of the movement.

successive predictions and by forcing the prediction to agree with any available training signals. In the case of predicting the path length to the home position, the critic is trained to make successive predictions decrease by one radius and to make the prediction at the home position zero. Furthermore, the critic is penalized when the robot hits an obstacle. The exact error signal is:

$$\epsilon(t) = \begin{cases} h, & \text{if the robot just hit an obstacle;} \\ d(t) - p(t), & \text{if the robot is within one radius of the home position;} \\ p(t+1) - p(t) - r, & \text{otherwise,} \end{cases} \tag{5.8}$$

where h is a fixed penalty factor greater than zero for hitting an obstacle, $d(t)$ is the distance from the center of the robot to the center of the goal position at time t, $p(t)$ is the sum of the outputs of the homing and obstacle-avoidance critics at time t, and r is the radius of the robot. This error signal is back-propagated to adjust the obstacle-avoidance critic's weights.

Table 5.2. The Gradient Descent Procedure for Perceptual Servoing. $(\Delta x, \Delta y)$ is an action, $d(t)$ is the distance between the position of the center of the robot and the home position. η is a learning rate, and δ and ϵ are small constants.

1. Initialize $(\Delta x, \Delta y) = (0, 0)$.
2. While $d(t) > \delta$ Do Begin
   (a) Loop
      i. Feed $(\Delta x, \Delta y)$ and current view into the range-flow model and calculate predicted view.
      ii. Back-propagate the difference between the prediction and the home view giving $(-\frac{\partial E}{\partial \Delta x}, -\frac{\partial E}{\partial \Delta y})$.
      iii. $(\Delta x, \Delta y) \leftarrow (\Delta x - \eta\frac{\partial E}{\partial \Delta x}, \Delta y - \eta\frac{\partial E}{\partial \Delta y})$.
   (b) Until $|(\frac{\partial E}{\partial \Delta x}, \frac{\partial E}{\partial \Delta y})| < \epsilon$
   (c) Execute $(\Delta x, \Delta y)$.
3. End

5.4 Perceptual Servoing

Our differentiable range-flow model can be used to perform *perceptual servoing*, a technique for determining an action given only perceptual information (see Fennema, Hanson, and Riseman [23]). It calculates a single action for getting home given only the current view and the home view. This is useful for final homing when the robot is one action away from the home location. Our technique uses gradient descent to choose an action such that the predicted next view is the view generated from the home position. The procedure is summarized in Table 5.2. Figure 5.13 shows two example trajectories created using perceptual servoing in the door environment.

Perceptual servoing can be used in conjunction with the navigation technique described earlier to form complete trajectories. Initially actions are chosen that minimize the predicted path length to the home position. Then perceptual servoing is employed when the robot is within one radius of the home position.

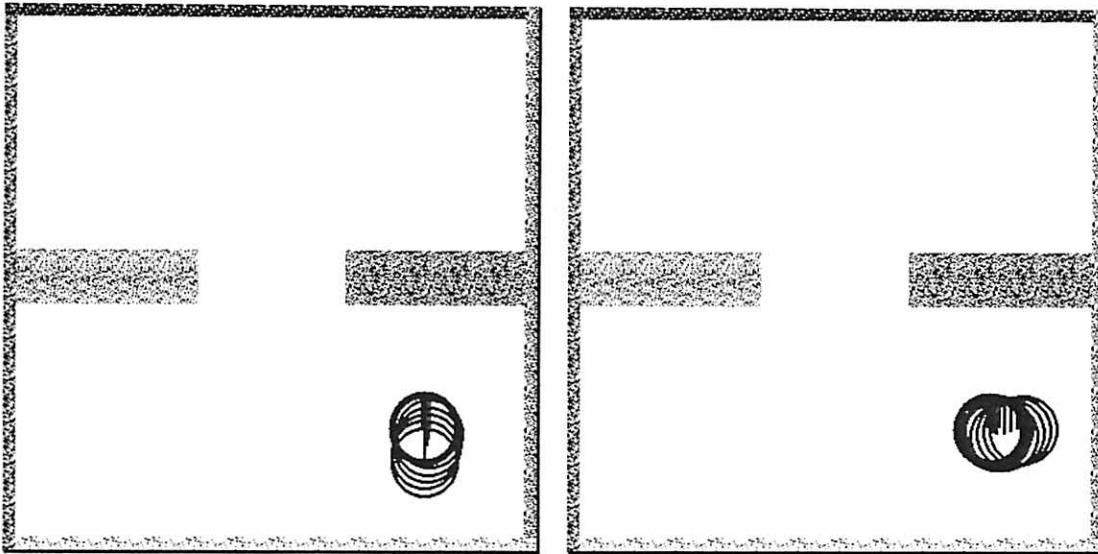

Figure 5.13. Two Examples of Perceptual Servoing in the Door Environment.

5.5 Results

The architecture was applied to the obstacle environment shown in Figure 5.14 and the door environment shown in Figure 5.2. The homing and obstacle-avoidance critic networks were composed of 200 Gaussian units, whose weights were trained initially using expectation maximization on views generated from the feasible positions of an evenly spaced grid of 64×64 positions superimposed on the robot's environment. Expectation maximization converged in approximately 200 iterations. These weights were then used as the initial weights for the hidden layer of both critics. The weights of each critic's output unit were initialized to zero. A learning rate of 0.025 was used for training the homing critic and 0.0005 for training the obstacle-avoidance critic. The penalty factor h was chosen to be 12 in all the experiments.

Figures 5.15 and 5.16 show the results of training. The left panel of each figure is a contour plot of the output of the homing critic and reflects only the straight-line distances to the home location from each respective starting position. The right panel of each figure is a contour plot of the combined outputs of the homing critic and the obstacle-avoidance critic. This plot now reflects the actual path length home.

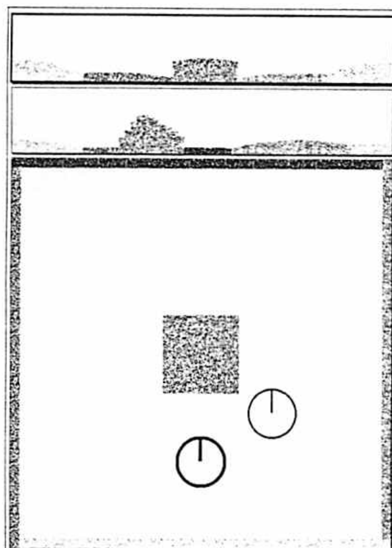

Figure 5.14. The Obstacle Environment.

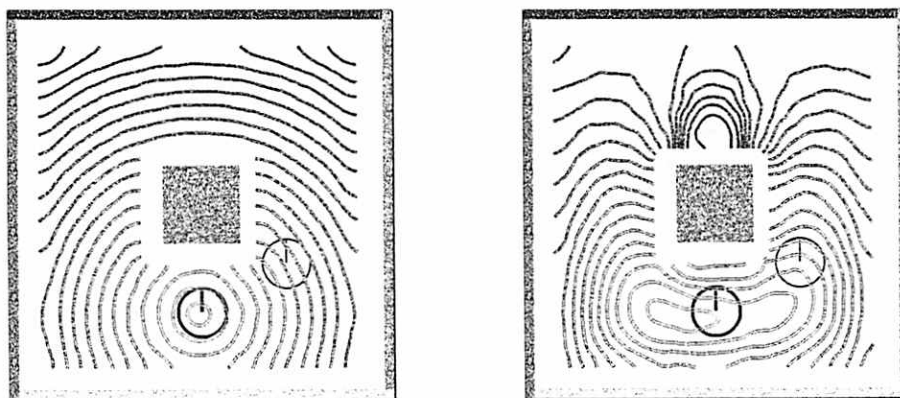

Figure 5.15. Contour Plots of the Output of the Critics After Training on the Obstacle Environment. The left panel shows a contour plot of the output of the homing critic. The right panel shows a contour plot of the sum the outputs of the homing and obstacle-avoidance critics.

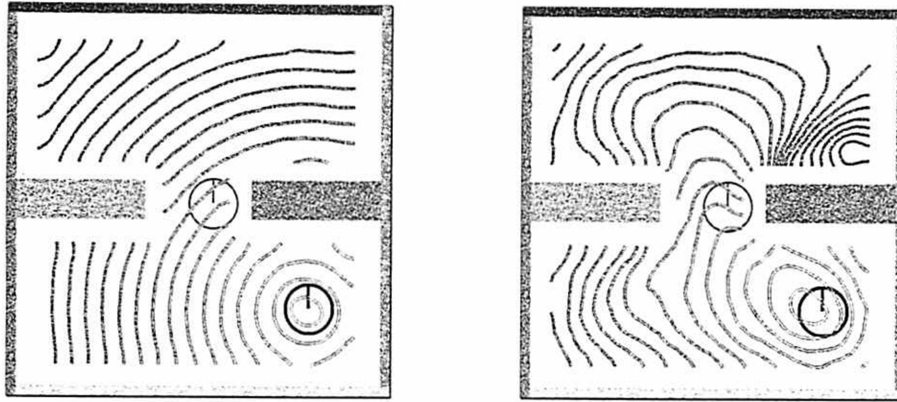

Figure 5.16. Contour Plots of the Outputs of the Homing Critic and the Combined Homing and Obstacle-Avoidance Critic for the Door Environment. The left panel shows a contour plot of the output of the homing critic. The right panel shows a contour plot of the sum of the outputs of the homing and obstacle-avoidance critic.

After training, the robot is able to form efficient homing trajectories starting from anywhere in the environment. Figures 5.17 and 5.18 show trajectories formed using only the homing critic. These demonstrate that the homing critic alone cannot adequately navigate home in the presence of obstacles. Figures 5.19 and 5.20 show trajectories formed using both the homing and obstacle-avoidance critics. The combination of the homing and obstacle-avoidance critics permits the robot to find a path home that is both efficient and avoids obstacles.

5.6 Discussion

The homing task represents a difficult control task requiring the solution of a number of problems. The first problem is that there is a small chance of getting home using random exploration. Unlike in the FSA environments described in Chapter 1, trial-and-error learning is intractable. The solution to this problem involves building a nominal homing critic that chooses straight-line trajectories home.

Next, because the state space is high-dimensional and continuous, it is impractical to uniformly place Gaussian units throughout the state space, and the training of networks with logistic hidden units is not easily achieved on this space. Instead we use

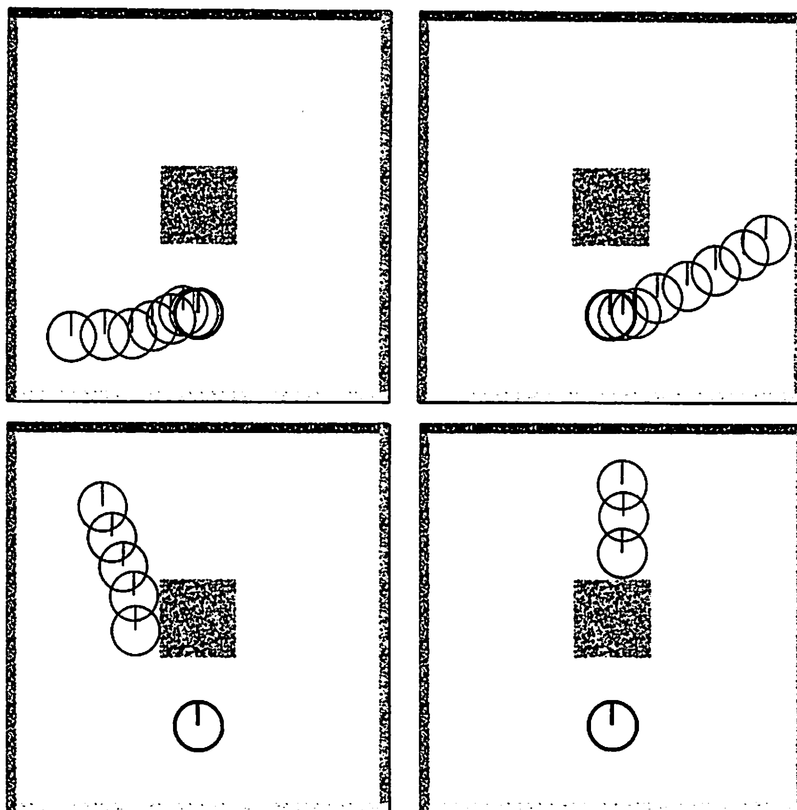

Figure 5.17. Trajectories Formed on the Obstacle Environment Using Only the Homing Critic.

Gaussian units whose initial weights are determined using expectation maximization, that in our case, creates spatially-tuned units.

Next, a model for the robot's environments is very difficult to learn. For this reason we used a hard-wired range-flow model whose performance is good in a wide range of environments. Here the philosophy is to learn only things that are difficult to hard-wire.

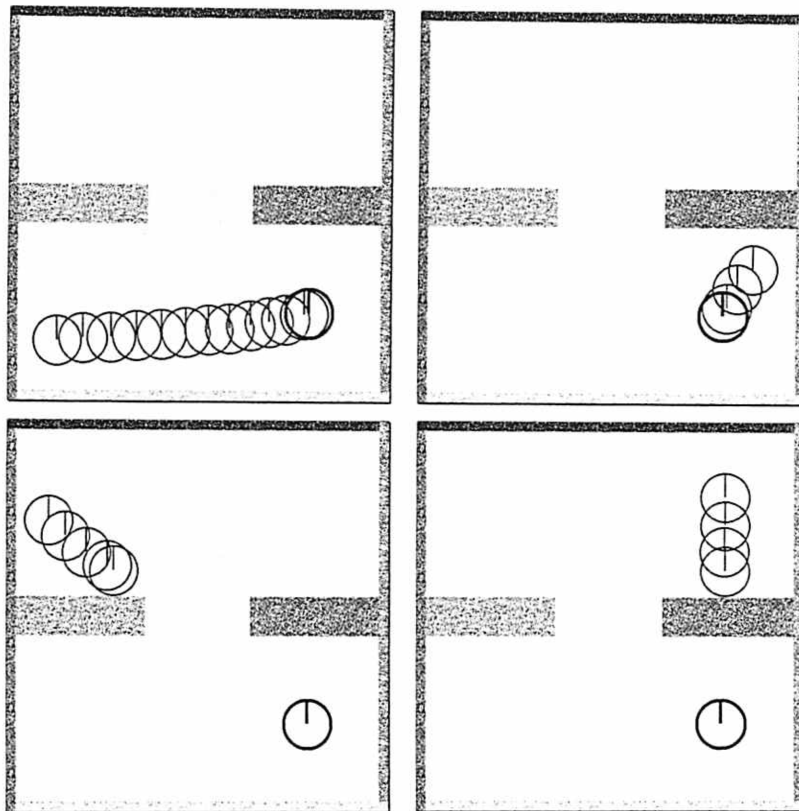

Figure 5.18. Trajectories Formed on the Door Environment Using Only the Homing Critic.

Despite our successful results, the current approach has many limitations. The training procedure is unrealistic for environments that are nonstationary because there is no mechanism for incrementally adjusting to a changing environment. A more practical training regimen would involve invoking the separate learning phases simultaneously. In particular, the Gaussian hidden units would be trained with an unsupervised procedure at the same time the homing and obstacle-avoidance critic were being trained.

Another limitation is that the obstacle-avoidance behavior does not generalize well to new environments. One way to fix this problem is to incorporate a hard-wired obstacle avoider into the navigation architecture. The hard-wired obstacle avoider would insure that the robot never bumped into obstacles and would then give the robot time to learn efficient goal-seeking avoidance trajectories (see Brooks [13]).

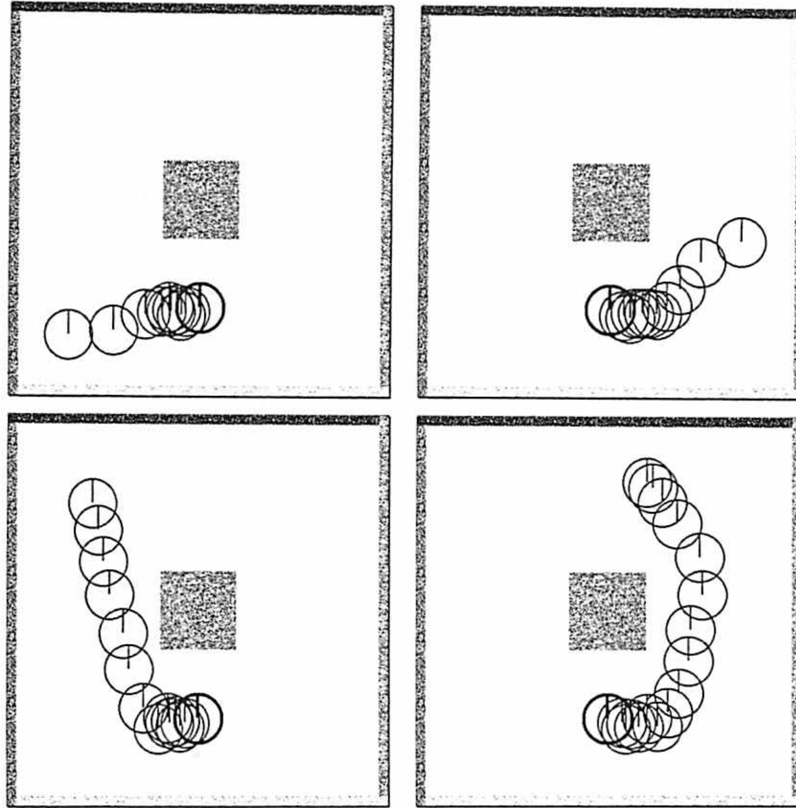

Figure 5.19. Trajectories Formed on the Obstacle Environment Using the Homing and Obstacle-Avoidance Critic.

The current training procedure will not always produce efficient trajectories. One reason is because it lacks a sophisticated exploration strategy. Without an exploration strategy, the robot will randomly choose a particular direction which might result in an inefficient trajectory. It would be easy to incorporate a simple exploration strategy like the one described in Chapter 4.

5.7 Future Work

There are many directions in which this work might be extended. First, we would like to apply this architecture to real robots using realistic sensors and dynamics. Real sensors are noisy and have a host of idiosyncrasies. An architecture must be robust in the face of sensor noise and must be able to accommodate the differences between individual sensors and the peculiarities of particular types of sensors, such as sonar. Connectionist networks are well suited for these problems for the following reasons.

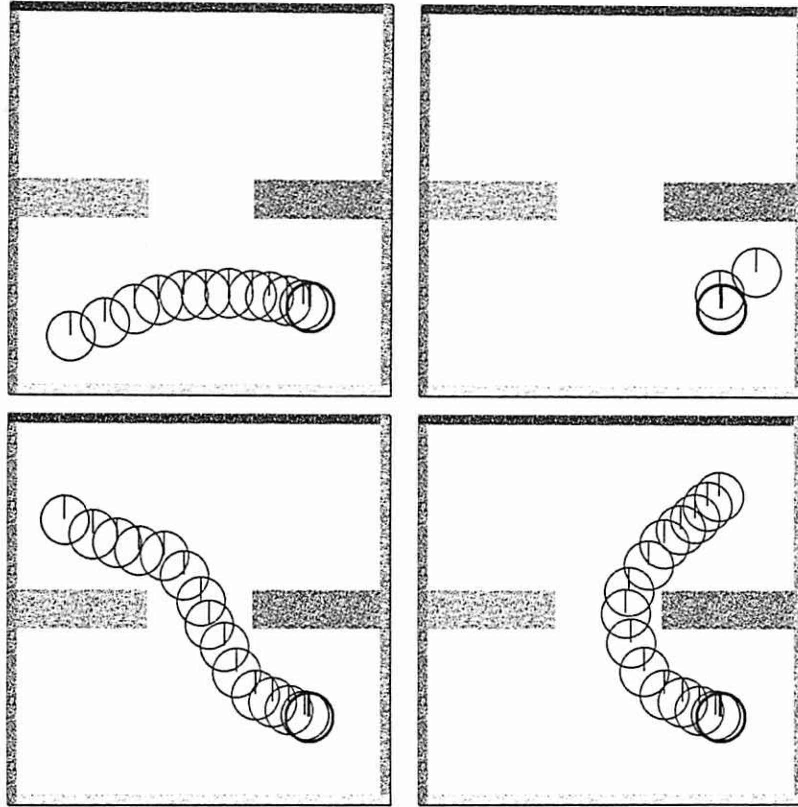

Figure 5.20. Trajectories Formed on the Door Environment Using the Homing and Obstacle-Avoidance Critic.

First, because of the statistical nature of connectionist learning rules, connectionist networks are able to learn in spite of noise in the examples, and they can also learn to approximate probabilistic functions in a principled fashion [31]. Secondly, the flexibility of connectionist networks as function approximators makes them suitable for modeling sensor idiosyncrasies.

We now present two alternatives for sensor modeling. The conservative approach is to add a few tunable parameters to our hard-wired range-flow model. These parameters could be adjusted during a calibration phase. The more radical approach is to replace the hard-wired model with a trainable network with a large number of hidden units. The range-flow model network would then learn the sensor idiosyncrasies.

In this chapter, we have not addressed the problem of dynamics. The basic training procedure is general enough to handle dynamics, but the range-flow model

would have to be modified in order to account for the effects of dynamics. Barto, Bradtke, and Singh [7] have demonstrated a related approach to a robot control task involving dynamics.

To permit the robot to keep up with the real-time demands of real robots, it is important to improve the execution speed of the training and performance of the networks. One major improvement could come from caching the control law in a controller network instead of relying solely on the expensive gradient descent action selection search. The controller network could learn the action chosen by the gradient descent action selection search. and then the controller could be used without the search, once it reliably predicts the best action.

We also want to study long-range homing. Our current technique does not scale well with longer ranges for the following reasons. First, dead-reckoning is infeasible in larger environments, and thus a scheme must be developed for training the homing critic. We outline one such scheme in the next paragraph. Secondly, the TD method used to train the obstacle-avoidance critic does not scale efficiently. In order to effectively operate in larger environments, a robot must decompose its trajectories into subgoals. Recently, Jacobs [33] reported on an architecture for task decomposition, and Singh [92] has generalized this architecture for the learning of sequences of trajectories using TD methods. See also Schmidhuber [88] for related ideas.

While it is clear that a hierarchical mechanism is needed, it is not clear what form it should take. In navigation, there is a significant amount of a priori knowledge, such as knowledge of spatial relationships, that can be used to devise such a mechanism. One idea is to use the notion of hubs. In order to navigate from one place to another, a robot would navigate to the closest hub on the way, navigate to the closest hub near the destination, and then navigate to the destination. Another possibility is the

idea[3] of using the metaphor of major highways and local streets. This scheme would involve navigating to a major highway using local streets, travelling on this major highway, and then getting off this major highway and navigating to the destination using local streets.

We would also like to investigate navigation tasks involving multiple destinations, as opposed to a single fixed destination. For example, the robot might be required to efficiently navigate to a variety of destinations that are either internally or externally specified. The obvious but inelegant solution is to use a separate critic for each destination. A better idea (due to Rumelhart [77]) is to have the homing critic learn to predict the x-y position of each view with respect to a particular fixed coordinate axis, instead of merely predicting the distance to the destination. In order to navigate to a particular destination, the straight-line potential function could be generated by the distance between the destination and the current position. The homing critic calculates these positions using their respective views. In short-range homing, dead-reckoning can be used to train an x-y homing critic. Instead of merely keeping track of the distance to the home position, dead-reckoning can keep track of separate x and y displacements from a given starting position.

In large-scale environments where dead-reckoning is not practical, another more sophisticated technique (due to Rumelhart [77]) can be employed. This technique utilizes an autoencoder as shown in Figure 5.21. The autoencoder learns to compute the identity mapping from views to views, with the constraint that the network's middle layer contains only two units. The learning procedure must construct an approximation to a two-dimensional invertible code for each view. The idea is to reconstruct the two degrees of freedom used to generate the views. The bottom half of the autoencoder could then be used as the homing critic. Figure 5.22 shows the results of training the autoencoder on a very simple environment without obstacles. It

[3]This is a classic idea from hierarchical planning (e.g., Sacerdoti [84] and Laird, Newell, and Rosenbloom [46]). Rumelhart [77] suggested this idea for connectionist navigation.

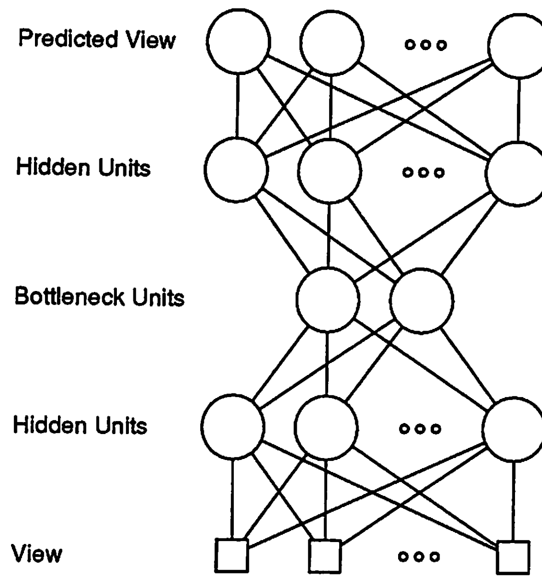

Figure 5.21. An Autoencoder Network for Learning a Two-Dimensional Code for Views. The network's task is to take a particular view as input and produce the same view on its output units using only two bottleneck units. One particular representation of views on the bottleneck units would be the x-y position where the view was generated. The autoencoder would then perform the transformation from view to x-y and then from x-y back to view. The bottom half of the encoder could be used to calculate the x-y position of a given view and could be used as a homing critic for multiple destination tasks as discussed in the text.

remains to be seen whether this technique will scale to more complicated environments with many obstacles.[4]

Addressing the problem of multiple destinations also introduces the problem of needing different obstacle-avoidance strategies for different destinations. Figure 5.23 shows an environment (called the culdesac environment) where there are two distinct destinations. Obviously, the obstacle-avoidance critic must have access to the particular destination in calculating its estimate.

[4]Rumelhart has suggested a couple techniques for finding better position codes on the bottleneck units. One technique is to add additional weight constraints and another is to start with more bottleneck units and to then gradually remove them.

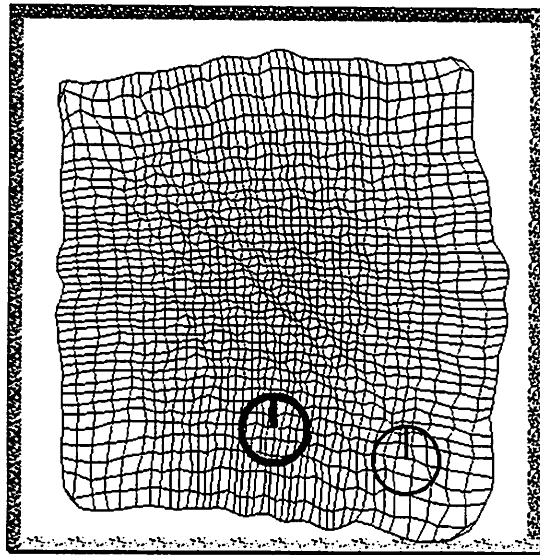

Figure 5.22. The Result of Training an Autoencoder on a Simple Environment Without Obstacles. This figure shows the responses of the two bottleneck units from Figure 5.21 to views generated at each of the feasible positions of a uniform 50×50 grid superimposed on the robot's environment. The responses of the bottleneck units to a particular view are plotted as an x-y point. Points generated by the bottleneck units for neighboring positions in the grid of views are connected by a line segment.

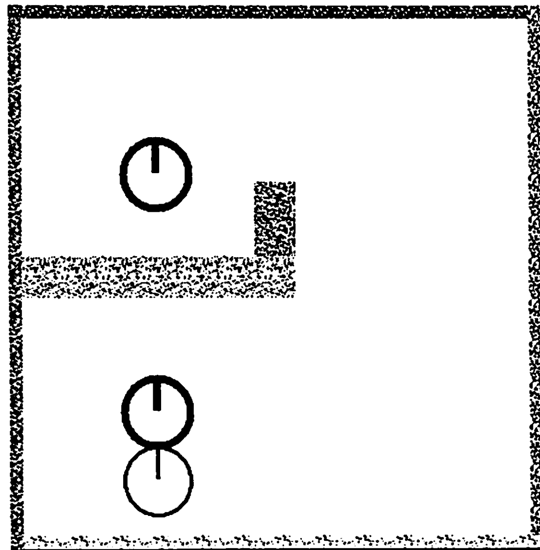

Figure 5.23. The Influence of Destination on Obstacle-Avoidance. Two different destinations are shown: one outside the culdesac and one inside the culdesac. In the first case, the robot should avoid the culdesac entirely, and in the second case, the robot should enter the culdesac.

5.8 Related Research

Our technique is most strongly related to navigation techniques that use potential functions. Our approach differs from these techniques in a number of ways. First, our approach involves learning. Our network is able to learn potential functions, automatically adjusting them to avoid both local minima and to produce efficient trajectories. In contrast, traditional potential function techniques suffer from the existence of minima and do not produce efficient trajectories. Secondly, our approach does not require object models because it learns potential functions using only sensory information. In contrast, the traditional technique involves building explicit object models representing the extent and position of objects in the robot's environment.

Our work is related to the subsumption architecture proposed by Brooks [13]. Both approaches involve the orchestration of multiple behaviors through the use of multiple experts. In our case, we examine obstacle-avoidance and homing behaviors. Also, neither approach utilizes explicit world models, but rather use only sensory information in choosing actions. Furthermore, neither technique utilizes classical planning, but instead use reactive planning, where actions are chosen based solely on the current view.

Our approach differs from the subsumption architecture in the following ways. The subsumption architecture involves an arbitration scheme which decides which behavior to invoke, while our technique has no such explicit mechanism. The various behaviors are invoked based on their contribution to the combined potential field. One could imagine more sophisticated techniques for combining behaviors within the potential function framework, such as the gating network proposed by Jacobs [33]. The subsumption architecture is based on finite state automata where the computations are performed primarily on binary signals, whereas our architecture involves continuous mappings constructed with connectionist networks. Finally, the subsumption architecture is handcrafted and involves no learning.

Recent work by Mahadevan and Connell [51] and Maes and Brooks [50] propose preliminary learning techniques for the subsumption architecture. Maes and Brooks present a reinforcement learning technique for learning an arbitration strategy that determines when to activate a hard-wired behavior. Mahadevan and Connell examine the learning of the behaviors given a fixed arbitration strategy. Singh [92] presents a technique that combines these two complementary approaches within the connectionist framework.

Mataric [52] reports on a subsumption architecture tailored for navigation. The method utilizes a set of simple. incrementally-designed behaviors that receive sonar and compass data. These behaviors cause the robot to wander around tracing the boundaries of the environment while detecting landmarks and creating a map of the environment. The map is a graph in which the vertices represent landmarks, and the edges represent adjacency relations between landmarks. Spreading activation is used to generate paths to the goal. The paths created using this technique are not necessarily efficient. They are composed of a series of boundary tracing trajectories that connect a series of landmarks leading to the goal position. In contrast. our technique utilizes a learning algorithm that produces time efficient trajectories. In practice. the trajectories formed by our navigation network are efficient but not optimal. On the other hand. our algorithm scales poorly. whereas Mataric's algorithm can be applied to large-scale environments.

We discuss two relevant and noteworthy connectionist navigation techniques. Pomerleau [69] describes an approach for training a 3-layer back-propagation network for road following. His system takes video images and range information as input and produces a steering command for the vehicle to follow the road. In contrast to our approach, Pomerleau's system is trained using a human teacher that provides steering commands as targets for the network. In our approach, we assume no such training information. The training regimen is completely unsupervised.

Park and Lee [64] present a technique for collision-free path planning using a connectionist network. They investigate the case of a polyhedral vehicle travelling through a set of polyhedral obstacles. For each obstacle, a penalty function is defined using a three-layer network. The total energy of a path is computed based on the collision penalty and the length of the path. Gradient descent is used to move the path such that the energy is minimized. In contrast to our sensory-based approach, their approach requires explicit object models. Furthermore, our approach is reactive, that is, it calculates its trajectory on the fly, whereas their technique requires a planning step at the outset.

CHAPTER 6
CONCLUSIONS

In this dissertation we addressed an extremely general robot scenario. A robot is placed in an unfamiliar environment. It explores the environment by performing actions and observing their perceptual consequences. The robot's task is to construct an internal model of the environment that will allow it to predict the outcomes of its actions and to determine what sequences of actions it should take to reach particular goal states.

We presented a technique, called SLUG, for building internal models of finite state environments. SLUG combines techniques from computational learning theory and connectionism. Several benefits of SLUG have been demonstrated. It improves on the Rivest and Schapire's learning alogrithm (hereafter, *the RS algorithm*) in the following ways:

- Connectionist learning algorithms allow for a great deal of parallelism. The network can update nearly all its weights after each action. In contrast, the RS algorithm gains information about at most a few update graph arcs at a time after each action. Furthermore, SLUG performs parameter estimation and state estimation simultaneously, whereas the RS algorithm performs parameter estimation and state estimation as two separate and distinct tasks.

- In contrast to the RS algorithm, SLUG's performance degrades gracefully as the number of units decreases below the required number of units, and its performance does not appear to degrade, and often improves, as the number of units increases above the required number of units [58].

- SLUG permits the continous availability of predictions, which gradually improve with experience. Furthermore, before learning is complete, the predictions can be substantially correct. In contrast, the RS algorithm can only make predictions after the update graph is complete.

- SLUG can be generalized to accomodate stochastic environments, including Markov environments (see Durbin and Bachrach [18]) and environments in which the sensations are somewhat unreliable. The RS algorithm is designed specifically for deterministic environments.

- As pointed out in Mozer and Bachrach [57], treating the update graph as a connectionist network suggests generalizations of the update graph formalism that don't arise from a more traditional analysis. First, there is the direct extension of allowing complementation links. Second, because SLUG can be considered a time-varying linear system where the weight matrix is chosen according to the current action, any rank-preserving linear transform of the weight matrices will produce an equivalent system, but one that does not have the local connectivity of the update graph. The linearity of the network also permits the use of linear algebra tools for analysis of the resulting connectivity matrices. Finally, the update representation can be even further extended (e.g., by adding hidden units) without changing the basic learning algorithm.

Rivest and Schapire's approach offers a few advantages over SLUG. First of all, their learning algorithm is provably polynomial for a restricted class of environments. There are no analogous results for SLUG. Secondly, their algorithm scales well for complex environments—ones in which the number of nodes in the update graph is quite large and the number of distinguishing environmental sensations is relatively small. One example environment is the n-bit shift register. This is probably the greatest weakness of SLUG. In these environments, an intelligent exploration strategy is crucial.

SLUG has a number of beneficial properties that differentiate it from typical connectionist learning methods:

- After training, because the activities and weights are binary, the update graph can perfectly predict the sensations infinitely far into the future; the state units are readily interpretable within the update graph formalism; many formal and empirical results exist that lend intuition and rigor.

- SLUG outperforms other "conventional" recurrent networks. These recurrent networks—Elman network [22] and Jordan network [34]—are spectacularly unsuccessful at learning to model FSA's (see Section 3.6).

- Unlike other networks, SLUG greatly facilitates the coding of a priori knowledge about the environment. Knowledge of relations between actions and the effect of particular actions can be hardwired into the weights of the network. The explicit representation of the effects of actions on the internal state as separate weight matrices eases this process.

- Unlike other connectionist architectures (including Giles et al's second-order network [26]), SLUG can be considered a time-varying linear system. A large number of well-studied techniques from optimal estimation apply to linear time-varying systems including the Kalman filter [25].

We also presented a control acquisition architecture that can use SLUG to determine a minimal sequence of actions to follow to reach a particular goal state. This architecture borrows heavily from other control acquisition techniques. In particular, our control acquisition architecture can be considered a 3-net architecture without the controller. This simplifies the relationship between the critic and the chosen actions—the critic directly defines the actions. Empirically, this was found to improve the learning performance.

We also considered the complexities of more realistic environments, characterized by high-dimensional continuous state-spaces with real-valued actions and sensations. We presented a connectionist control acquisition architecture tailored for simulated short-range homing in the presence of obstacles. Our architecture overcomes previous limitations of the artificial potential function approach (see also Koditschek [43], LaTombe [47], and Connolly, Burns, and Weiss [15] for techniques that also overcome these limitations). In particular, our approach can avoid local minima and produce efficient trajectories.

In solving this control task, we demonstrated several techniques. First, we demonstrated the use of Gaussian units whose initial weights are determined using a soft form of competitive learning, called expectation maximization, that, in our case, create spatially tuned units. Second, we demonstrated the use of a hard-wired range-flow model whose performance is good in a wide range of environments. Finally, we demonstrated the use of a nominal initial controller.

Much work is needed to scale the navigation architecture to real-world navigation tasks involving real robots and large-scale environments. Lessons learned from the simulated small-scale environments will be very beneficial.

APPENDIX

THE UPDATE GRAPH

In this section. Rivest and Schapire's *diversity representation* [73, 87] is described in detail. It is a particular encoding of an FSA where the state variables form a distributed representation of the FSA's state. The *update graph* is then presented. It is a particular machine that updates the values of the internal state variables. Rivest and Schapire's algorithm for learning update graphs is then described.

A.1 The Diversity Representation

The diversity representation concepts are now presented:

Definition 1 (Action Sequences): *Let $S = A^*$ denote the set of all strings of zero or more actions, where A is the set of actions, and the Kleene star $*$ denotes the closure operation. Each such string of actions is called an action sequence. Let λ be the zero-length action sequence.*

The test state variables can now be defined:

Definition 2 (Test): *A test $t = sp$ is an action sequence followed by a predicate.*

Let $T = SP = \{sp | s \in S, p \in P\}$ denote the set of all tests. The *value* of a test $t = sp$ in a state q is denoted by qt, where $qt = q(sp) = qsp = (qs)p$, qs denotes the state resulting from executing action sequence s in state q, and $(qs)p$ is the value of predicate p in state qs. A test $t = sp$ is said to *succeed at state* q if $qt =$ **true**. Otherwise test t is said to *fail at* q. Executing a test $t = sp$ in state q means executing action sequence s in state q and observing the resulting value of p. The *length* $|t|$ of a

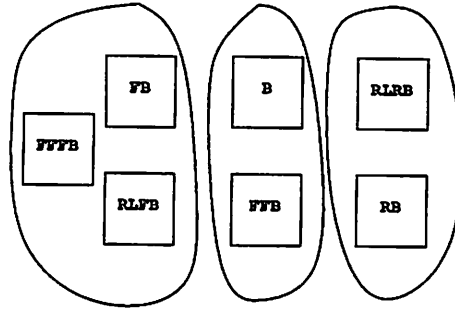

Figure A.1. Some Test Equivalence Classes for the 3-Bit Shift Register Environment.

test t is one plus the number of actions it contains. If the robot knows the values of all the tests in the current state q, then it can predict the sensations resulting from executing any action sequence. In that sense it has a *perfect model* of the environment in state q.

In the n-bit shift register environment, B, FB, LRB, FFFFFB, and LLLLFB are a few of the infinite number of tests. In this environment let 1 denote **true** and 0 denote **false**. For a 3-bit register in state $q = 011$, the values of qt for these tests are B= 0, FB= 1, LRB= 0, FFFFFB= 1, LLLLFB= 0.

In order to make a finite state representation, a finite number of tests must be chosen to serve as the state variables. This is done by grouping the infinite set of tests into equivalence classes. The equivalence between two tests is defined as follows:

Definition 3 (Equivalence): $t_1 \equiv t_2 \Leftrightarrow (\forall q \in Q)(qt_1 = qt_2)$.

The equivalence class containing test t is denoted by $[t]$. The *canonical* member of a class is the shortest length test in that class and is denoted by $\langle t \rangle$. Figure A.1 shows some of the equivalence classes for the shift register environment. All the tests within a circle are equivalent to each other. Note that there are an infinite number of members within any class; only a few members of each class are shown in the figure.

Rivest and Schapire show that the canonical tests can serve as the internal state variables, such that the values of the canonical tests completely describe the state of

the environment. An environment is assumed to be *reduced*, that is, an environment has the property that for any two states there is a test that will distinguish them:

Definition 4 (Reduced Environment):

$$(\forall q_1 \in Q)(\forall q_2 \in Q)(q_1 \neq q_2 \Rightarrow (\exists t \in T) q_1 t \neq q_2 t). \qquad (A.1)$$

For a reduced environment, two states are equal if and only if the values of all the canonical tests are the same in both states. For each state of an FSA there is a corresponding set of test values over the canonical tests. In this sense, these values form a distributed representation of the state.

Diversity can now be defined:

Definition 5 (Diversity): *The diversity of the environment $\mathcal{E}$, denoted $D(\mathcal{E})$, is the number of equivalence classes of $\mathcal{E}$:* $D(\mathcal{E}) = |\{[t] | t \in T\}|$.

Rivest and Schapire show that $\log_2 |Q| \leq D(\mathcal{E}) \leq 2^{|Q|}$. They claim that for "natural" environments, the diversity can be much smaller than the number of environmental states. For instance, in the grid environment, they show that $D(\mathcal{E}) = 3(n^2 - 1)$, but $|Q| = 3^{n^2-1}$.

Figure A.2 shows the canonical members of the six equivalence classes for the 3-bit shift register environment. In the n-bit shift register environment, only $2n$ state variables are needed to represent the 2^n states. Figure A.3 shows the values of the canonical tests for the given state of the 3-bit register. It turns out that for the shift register environment, there are two state variables for each bit in the shift register, one for the actual value and one for the negated value of that bit. Clearly, this is not the best possible state representation, but it is far superior to explicitly representing all 2^n states.

Now that the state variables are defined, a method is needed to update their values and properly initialize them. First the canonical tests can be related to each other. Figure A.4 shows the canonical tests for the shift register environment and

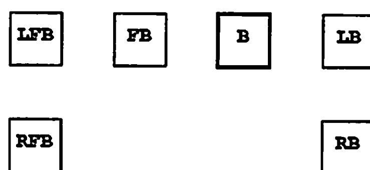

Figure A.2. The Diversity-Based State Variables for the 3-Bit Shift Register Environment.

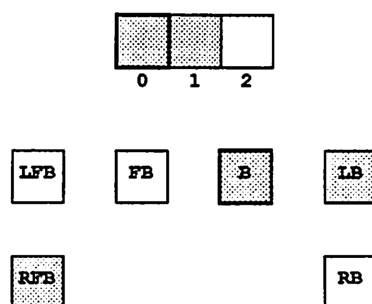

Figure A.3. The Values for the Diversity-Based State Variables for the Given State of the 3-Bit Shift Register. The 3-bit shift register is depicted above. A shaded box means the element contains a one and otherwise contains a zero.

also a few relationships between the tests as depicted by the arcs. These relationships are denoted with an arc $t_1 \xrightarrow{a} t_2$ which means that $t_1 \equiv a t_2$, where t_1 and t_2 are tests and a is an action. The simplest example is the LB $\xrightarrow{\mathtt{L}}$ B arc, which means that the value of LB before executing L is the same as the value of B after executing L, or LB ≡ LB. The arc B $\xrightarrow{\mathtt{R}}$ LB is a more complicated example and means that B ≡ RLB. These relationships provide a technique for updating the values of the canonical tests after executing actions. For example, if the value of test LB is known and the robot executes an R, then the value of B will be known.

A.2 The Update Graph

If an arc $t_1 \xrightarrow{a} t_2$ is known for each canonical test t_1 and each action a, then a machine can be built that maintains the correct values for the canonical tests. Rivest and Schapire call this machine the *update graph*:

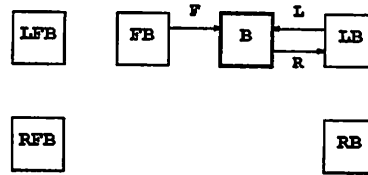

Figure A.4. Some Relationships Between Canonical Tests for the 3-Bit Shift Register Environment.

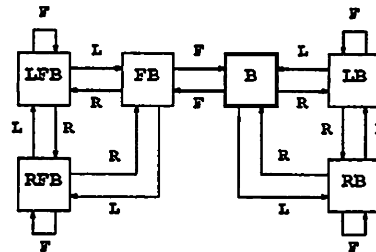

Figure A.5. The Update Graph for the 3-Bit Shift Register.

Definition 6 (Update Graph): *An update graph is a directed graph in which each vertex represents an equivalence class, and an edge labeled $a \in A$ is directed from vertex $[t]$ to $[t']$ iff $t \equiv at'$.*

Figure A.5 shows the update graph for the shift register environment. The update graph has a number of useful properties. Since there is one vertex for each equivalence class, there are $D(\mathcal{E})$ vertices. For each vertex of the update graph, there is exactly one incoming edge for each action (because $(\forall a \in A)t \equiv t' \Rightarrow at \equiv at'$).

If the value of the test in the current state q is assigned to each vertex $[t]$, qt, in a complete update graph, then the value of any test after executing any action can be found. First, it is known that an edge $t_1 \xrightarrow{a} t_2$ exists iff $t_1 \equiv at_2$. With a complete update graph, each vertex has one incoming edge per action. With each execution of an action $a \in A$ the value of each vertex $[t]$ is updated synchronously with the value of the vertex $[at]$ at the tail of $[t]$'s incoming a-edge. This is correct because after executing a, $[t]$ will have the value of $[at]$ before executing a. Rivest and Schapire

formally show that this accurately computes the value of each canonical test and that the class of machines represented by update graphs, namely *simple assignment automata*, are as powerful as FSA's.

In the register environment, shown in Figure A.5, it is easy to see how the update graph is updated. When an R action is executed, the values in the left half of the update graph are rotated counterclockwise, and in the right half clockwise; an L action rotates the values in the opposite direction. An F action causes the FB and B tests to exchange values, whereas all the other tests keep their old values. This updating process maintains the bits of the shift register in the right half of the update graph and the negated bits in the left half.

Assuming that the structure of the update graph is known, Rivest and Schapire provide a state estimation algorithm for inferring the values of the tests. Assign x_i to each canonical test, where x_i represents the value of the i'th canonical test, t_i, at the beginning of this process. In order to solve for the values of the canonical tests it is sufficient to solve for each x_i because all tests get their values only from other canonical tests. After the first action is executed x_i will move from t_i to t_k, and in general, a given t_i will have value x_j. The value of a test t_i with unknown value x_j can be solved for by executing the test t_i. This will serve to predict the value of sensation predicate p associated with test $t_i = sp$ with the unknown value x_j. The true value of x_j can then be found by reading the value of the sensation predicate p from the actual sensations produced by the environment. The procedure for inferring the values of all the tests in an update graph is: while $\exists$ a test t with an unknown value, execute t. The initial values of the tests can be solved for in $O(D(\mathcal{E})^2)$, because there are $D(\mathcal{E})$ unknown variables and each test is no longer than $D(\mathcal{E})$.

The update graph keeps track of the state of the FSA environment through tests. The tests code this information with respect to the actions available to the robot. The number of these canonical tests is the diversity of the environment. It is useful to add to the concepts borrowed from Rivest and Schapire the concept of *depth*:

Definition 7 (Depth): *The depth of an environment is the length of the longest canonical test.* $\mathcal{H} = \max_{t \in T} |\langle t \rangle|$.

A *low depth* environment has the property that the depth is much smaller than the diversity:

Definition 8 (Low Depth): $\mathcal{H} \leq c \log_{|A|} D(\mathcal{E})$, *where c is a fixed constant.*

A.3 The Rivest-Schapire Learning Algorithm

The basic idea of Rivest and Schapire's learning approach is to incrementally build the update graph one edge at a time adding new vertices as needed. The update graph starts out with the predicates as the only vertices and with their incoming edges undefined. Note that since all vertices will have exactly one incoming edge per action, the inference procedure can keep track of which edges are undefined and when the update graph is complete. The main loop of the inference procedure sequentially discovers the vertices at the tails of the undefined incoming edges. It can infer the vertex at the tail by running an equivalence procedure that compares the test defined at the tail with the current set of canonical tests. Let a test with an undefined incoming edge be called t, and let the incoming edge correspond to action a. The vertex at the tail of the incoming edge will be equivalent to at. If at is found to be equivalent to an already existing test t', then $[t'] \xrightarrow{a} [t]$ is added. Otherwise, a new vertex, $[at]$, is added, the edge $[at] \xrightarrow{a} [t]$ is added, and all $[at]$'s incoming edges are added to the undefined edge list. This continues until there are no more undefined incoming edges. If an oracle is available to decide whether two tests are equivalent, then the running time of this algorithm is $O(2|A|D(\mathcal{E})^2)$, where A is the set of actions and $D(\mathcal{E})$ is the diversity. The pseudocode for this inference procedure is given in Table A.1.

Suppose we want to learn a model for the 3-bit shift register. We start with Vertices = {`B(it?)`}, the only sensory predicate. Because there are three actions, `L`,

Table A.1. Diversity-Based Inference Procedure.

Vertices $\leftarrow$ Sensations
while $\exists$ a vertex $[t]$ missing incoming edge a
  if $at \equiv t'$ for some vertex $[t']$
  then add $[t'] \xrightarrow{a} [t]$
  else
    create new vertex $[at]$ and add to Vertices
    add $[at] \xrightarrow{a} [t]$
  end else
end while

R. and F. there are three undefined incoming arcs for the B vertex. By definition, the vertices at the tails of those arcs are LB. RB, and FB shown in Panel A of Figure A.6. where we depict the proposed vertices as shaded and the known vertices as white. The equivalence oracle decides that these proposed tests are inequivalent to the canonical test B as well as to each other. and thus these proposed canonical tests are added to the list of canonical tests. This is shown in Panel B of Figure A.6 along with all newly proposed vertices corresponding to undefined incoming arcs. At this point the set of vertices is {B, LB, RB. FB} and all the incoming arcs on LB, RB, and FB are undefined. Examining the proposed FFB node in Panel B of Figure A.6, we see that it is equivalent to a canonical test already defined, namely B. Therefore this proposed node, FFB, is redundant and when a F action is taken. FB can get its value from B instead (of FFB) as shown in Panel C of Figure A.6. Continuing in this fashion, the rest of the vertices and arcs are inferred and the final update graph is shown in Panel D of Figure A.6.

In order to make the inference procedure polynomial in diversity, one needs a polynomial time equivalence procedure. Unfortunately, in order to know whether two tests are equivalent one must compare their values when executed in all environmental

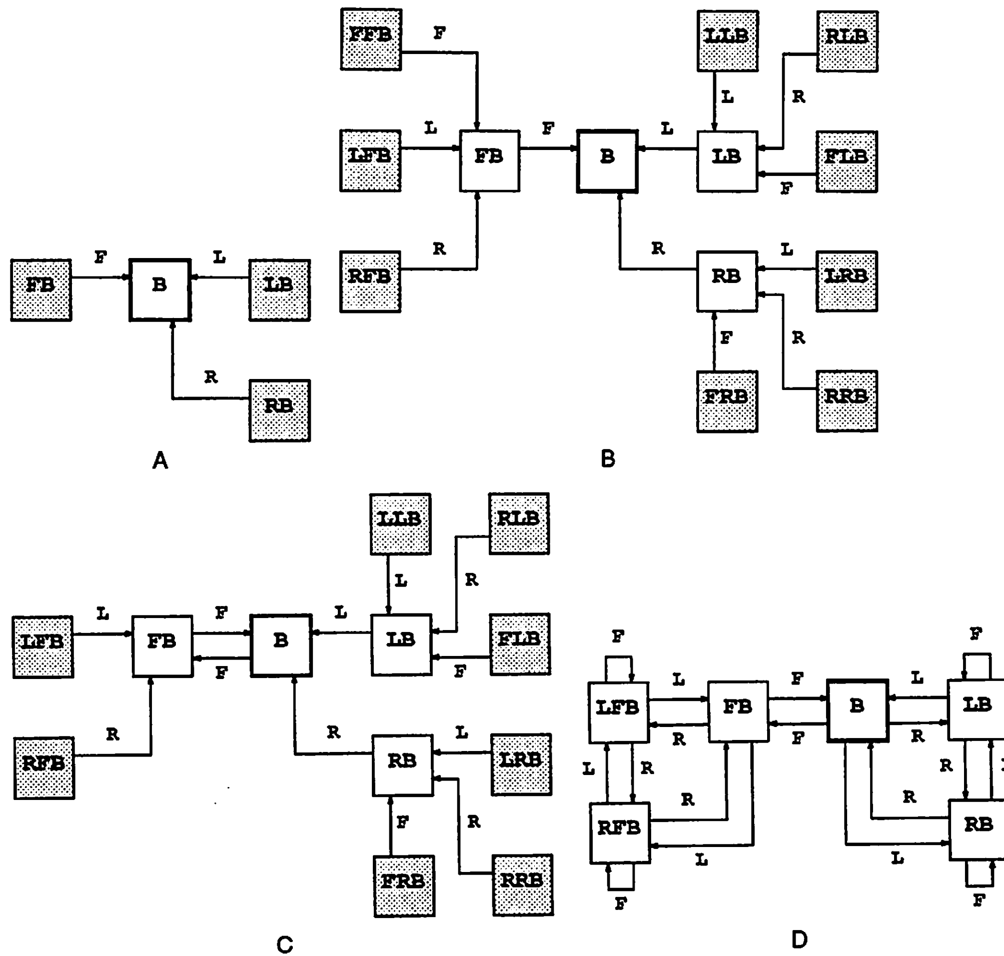

Figure A.6. A Series of Steps in the Diversity-Based Inference Process of the Update Graph for the 3-Bit Shift Register Environment.

Table A.2. Probabilistic Procedure for Determining $t \equiv t'$ Given Tolerable Probability of Error ϵ.

Repeat k times, where k is chosen so as to make
    the probability of an error less than ϵ
  Get into a random environmental state q
  Perform test t
  Get back into same environmental state q
  Perform test t'
  If $qt \neq qt'$ **then return false**
return true

states. This would be acceptable if one were interested in an equivalence procedure that is polynomial in the number of FSA states. The best that can be hoped for is an equivalence procedure that samples the global states and determines either: (1) that two tests are equivalent with a given confidence or (2) are inequivalent if a counterexample is found (since if we can find just one counterexample where they disagree, then the two tests are inequivalent). Rivest and Schapire's probabilistic equivalence procedure for asking $t \equiv t'$ is summarized in Table A.2. This is one of their suggested heuristic methods; they showed that other more complicated techniques are provably polynomial time.

Rivest and Schapire cast the equivalence problem in terms of finding counterexamples and state that there are two problems to face. The first is the problem of *accessibility of counterexamples*: it might be very difficult to get into an environmental state where the two tests disagree. The second is the problem of *irreversibility of actions*: once the first test is executed in state q, it is not always possible to go back to state q (i.e., undo the effects of an action) in order to execute the second test.

It turns out that for permutation environments, it can be proved that these problems can be overcome. In a *permutation environment* there exists a fixed inverse

for any test (see Schapire [87] for more details). In permutation environments a counterexample (if one exists) has a high probability of occurring during a long enough random walk (where the actions are chosen from a uniform distribution), and two tests can always be compared without undoing actions. Rivest and Schapire describe an equivalence procedure that they prove to be polynomial time on permutation environments. The result requires very long random walks that require running time on the order of $|A|D^8 \log(|A|\frac{D^2}{\epsilon})$, where the parameter ϵ is the tolerable probability of error. In practice, however, they have found that much shorter random walks suffice. They also describe a heuristic algorithm that is reasonably effective for nonpermutation environments [73, 87].

References

[1] D. Angluin. On the complexity of minimum inference of regular sets. *Information and Control*, 15(3):237–269, September 1978.

[2] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, November 1987.

[3] J. R. Bachrach. Learning to represent state. Master's thesis, University of Massachusetts, Amherst, COINS Department, Amherst, MA 01003, 1988.

[4] J. R. Bachrach. Learning to represent state. In *ICNN Conference Proceedings*, page 288. International Neural Network Society, Pergamon Press, September 1988.

[5] J. R. Bachrach and M. Mozer. Connectionist modeling and controlling of finite state systems in the absence of complete state information. In Yves Chauvin and David E. Rumelhart, editors, *Back-propagation: Theory, Architectures and Applications*. Erlbaum, Hillsdale, NJ, to appear.

[6] A. G. Barto. Connectionist learning for control: An overview. In T. Miller, R. S. Sutton, and P. J. Werbos, editors, *Neural Networks for Control*. The MIT Press, Cambridge, MA, 1990.

[7] A. G. Barto, S. J. Bradtke, and S. P. Singh. real-time learning and control using asynchronous dynamic programming. Technical Report 91-57, University of Massachusetts at Amherst, 1991.

[8] A. G. Barto, R. S. Sutton, and C. W. Anderson. Neuronlike elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, 13:835–846, 1983. Reprinted in J. A. Anderson and E. Rosenfeld, *Neurocomputing: Foundations of Research*, The MIT Press, Cambridge, MA, 1988.

[9] A. G. Barto, R. S. Sutton, and C. W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(15), September/October 1985.

[10] A. G. Barto, R. S. Sutton, and C. Watkins. Learning and sequential decision making. In M. Gabriel and J. W. Moore, editors, *Learning and Computational Neuroscience*. The MIT Press, Cambridge, MA, 1991.

[11] T. L. Booth. *Sequential Machines and Automata Theory*. John Wiley and Sons, Inc., 1967.

[12] J. Bridle. Training stochastic model recognition algorithms as networks can lead to maximum mutual information estimation of parameters. In David S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, San Mateo, CA, 1990. IEEE Conference on Neural Information Processing Systems—Natural and Synthetic, Morgan Kaufmann Publishers.

[13] R. A. Brooks. A robot that walks: emergent behaviors from a carefully evolved network. *Neural Computation*, 1, 1989.

[14] D. Cohn, L. Atlas, R. Ladner, M. El-sharkawi, R. Marks II, M. Aggoune, and D. Park. Training connectionist networks with queries and selective sampling. In David S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, San Mateo, CA, 1990. Morgan Kaufmann Publishers.

[15] C. I. Connolly, J. B. Burns, and R. Weiss. Path planning using laplace's equation. In *International Conference on Robotics and Automation*, 1990.

[16] R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis*. Wiley, New York, 1973.

[17] R. Durbin. Principled competitive learning in both unsupervised and supervised networks. Poster presented at the conference on Neural Networks for Computing, Snowbird, UT, April 1990.

[18] R. Durbin and J. R. Bachrach. Diversity-based representations of stochastic finite state automata. Unpublished Manuscript, 1990.

[19] R. Durbin and D. E. Rumelhart. Product units: A computationally powerful and biologically plausible extension. *Neural Computation*, ?(?), ? 1989.

[20] T. F. Elbert. *Estimation and Control of Systems*. Van Nostrand Reinhold Company, 1984.

[21] J. L. Elman. Finding structure in time. Technical Report 8801, Center for Research in Language: University of California, San Diego, April 1988.

[22] J. L. Elman. Finding structure in time. *Cognitive Science*, 14:179–212, 1990.

[23] C. Fennema, A. Hanson, and E. Riseman. Towards autonomous mobile robot navigation. Technical Report 89-104, Computer Science Department, University of Massachusetts, 1989.

[24] B. R. Gaines. Behavior/structure transformations under uncertainty. *International Journal of Man-Machine Studies*, 8:337–365, 1976.

[25] A. Gelb. *Optimal Estimation*. MIT Press, 1974.

[26] C. L. Giles, G. Z. Sun, H. H. Chen, Y. C. Lee, and D. Chen. Higher order recurrent networks and grammatical inference. In David S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, San Mateo, CA, 1990. IEEE Conference on Neural Information Processing Systems—Natural and Synthetic, Morgan Kaufmann Publishers.

[27] A. Gill. *Introduction to the theory of finite-state machines*. McGraw-Hill, 1962.

[28] E. M. Gold. System identification via state characterization. *Automatica*, 8:621–636, 1972.

[29] E. M. Gold. Complexity of automaton identification from given data.. *Information and Control*, 37, 1978.

[30] G. C. Goodwin and K. S. Sin. *Adaptive Filtering Prediction and Control*. Prentice Hall, 1984.

[31] G. E. Hinton. Connectionist learning procedures. Technical Report CMU-CS-87-115, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213, 1987.

[32] K. Hornik, M. Stinchcombe, and H. White. Multi-layer feedforward networks are universal approximators. Technical report, Department of Economics, University of California, San Diego, CA, June 1988.

[33] R. Jacobs. *Task decomposition through competition in a modular connectionist architecture*. PhD thesis, COINS Department, University of Massachusetts, Amherst, MA 01003, 1990.

[34] M. I. Jordan. *The learning of representations for sequential Performance*. PhD thesis, University of California, San Diego, 1985.

[35] M. I. Jordan. Indeterminate motor skill learning problems. In M. Jearnnerod, editor, *Attention and Performance*, volume XIII. MIT Press, 1990.

[36] M. I. Jordan and R. Jacobs. Learning to control an unstable system with forward modeling. In David S. Touretzky, editor, *Advances in Neural Information Processing Systems*, P.O. Box 50490, Palo Alto, CA 94303, 1989. Morgan Kaufmann Publishers.

[37] Michael I. Jordan and David E. Rumelhart. Supervised learning with a distal teacher. *Cognitive Science*, 1990. Submitted.

[38] L. Kaelbling. *Learning in Embedded Systems*. PhD thesis, Stanford University, 1990.

[39] M. Kearns and L. G. Valiant. Learning boolean formulae or finite automata is as hard as factoring. Technical Report TR 14-88, Harvard Univsity Aiken Computation Laboratory, 1988.

[40] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. In *International Conference on Robotics and Automation*, pages 500–505. IEEE, March 1985.

[41] G. J. Klir. On the representation of activity arrays. *International Journal of General Systems*, 2:149–168, 1975.

[42] G. J. Klir. Identification of generative structures in empirical data. *International Journal of General Systems*, 3:89–104, 1976.

[43] D. E. Koditschek. Exact robot navigation by means of potential functions: some topological considerations. In *International Conference on Robotics and Automation*, pages 1–6. IEEE, April 1987.

[44] Z. Kohavi. *Switching and finite automata theory*. McGraw-Hill, 1978.

[45] B. H. Krogh. A generalized potential field approach to obstacle avoidance control. In *Robotics Research: The Next Five Years and Beyond*. Society of Manufacturing Engineers, August 1984.

[46] J. E. Laird, A. Newell, and P. S. Rosenbloom. SOAR: An architecture for general intelligence. *Artificial Intelligence*, 33:1–64, 1987.

[47] J. Latombe. *Robot Motion Planning*. Kluwer Academic Press, 1990.

[48] Y. le Cun. Une procedure d'apprentissage pour reseau a sequil assymetrique [A learning procedure for asymmetric threshold network]. *Proceedings of Cognitiva*, 85:599–604, 1985.

[49] L. Ljung and T. Soderstrom. *Theory and Practice of Recursive Identification*. MIT Press, 1982.

[50] P. Maes and R. Brooks. Learning to coordinate behaviours. In *Proceedings of the eighth AAAI*, pages 796–802. Morgan Kaufmann, 1990.

[51] S. Mahadevan and J. Connell. Automatic programming of behavior-based robots using reinforcemnt learning. Technical report, IBM Research Division, T.J. Watson Research Center, Box 704, Yorktown Heights, NY 10598, 1990.

[52] M. J. Mataric. Environment learning using a distributed representation. In *1990 IEEE International Conference on Robotics and Automation*, Los Alamitos, CA, 1990. IEEE Computer Society Press.

[53] J. L. McClelland and J. L. Elman. *Interactive Processes in Speech Perception: The TRACE Model*, pages 58–121. MIT Press, 1986.

[54] J. L. McClelland and D. E. Rumelhart. An interactive activation model of context effects in letter perception: Part 1. an account of basic findings. *Psychological Review*, 88:375–407, 1981.

[55] M. C. Mozer. A focused back-propagation algorithm for temporal pattern recognition. Technical Report ?, Department of Psychology and Computer Science University of Toronto, Toronto, Ontario M5S1A1, CANADA, ? 1988.

[56] M. C. Mozer. A focused back-propagation algorithm for temporal pattern recognition. *complex systems*, 3(4):349–381, 1989.

[57] M. C. Mozer and J. R. Bachrach. Discovering the structure of a reactive environment by exploration. Technical Report CU-CS-451-89, University of Colorado at Boulder Department of Computer Science, November 1989.

[58] M. C. Mozer and J. R. Bachrach. Discovering the structure of a reactive environment by exploration. In David S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, San Mateo, CA, 1990. IEEE Conference on Neural Information Processing Systems—Natural and Synthetic, Morgan Kaufmann Publishers.

[59] M. C. Mozer and J. R. Bachrach. Discovering the structure of a reactive environment by exploration. *Neural Computation*, 2(4), 1990.

[60] M. C. Mozer and J. R. Bachrach. Discovering the structure of a reactive environment by exploration. *Machine Learning Journal*, to appear.

[61] K. S. Narendra and K. Parthasarathy. Identification and control of dynamical systems using neural networks. *IEEE transactions on neural networks*, 1(1):4–27, 1990.

[62] S. J. Nowlan. Competing experts: an experimental investigation of associative mixture models. Technical Report CRG-TR-90-5, Department of Computer Science, University of Toronto, 1990.

[63] S. J. Nowlan. A generative framework for unsupervised learning. In David S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, San Mateo, CA, 1990. IEEE Conference on Neural Information Processing Systems—Natural and Synthetic, Morgan Kaufmann Publishers.

[64] J. Park and S. Lee. Neural computation for collision-free path planning. In *Proceedings of the international joint conference on neural networks*, Hillsdale, NJ, 1990. Lawrence Erlbaum. Volume 2: Applications Track.

[65] D. B. Parker. Learning logic. Technical Report TR-47, Massachusetts Institute of Technology, 1985.

[66] B. A. Pearlmutter. Learning state space trajectories in recurrent neural networks. *Neural Computation*, 1(2):263–269, Summer 1989.

[67] L. Pitt and M. K. Warmuth. The minimum DFA consistency problem cannot be approximated within any polynomial. In *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*, Seattle, Washington, May 1989. ACM.

[68] J. B. Pollack. Language acquisition via strange automata. In *Proceedings of the Twelfth Annual Conference of the Cognitive Science Society*, Hillsdale, NJ, 1990. Cognitive Science Society, Erlbaum.

[69] D. A. Pomerleau. Alvinn: an autonomous land vehicle in a neural network. In David S. Touretzky, editor, *Advances in Neural Information Processing Systems 1*, San Mateo, CA, 1989. IEEE Conference on Neural Information Processing Systems—Natural and Synthetic, Morgan Kaufmann Publishers.

[70] S. Porat and J. A. Feldman. Learning automata from ordered examples. Technical Report TR241, University of Rochester, March 1988.

[71] E. Rimon and D. E. Koditschek. Exact robot navigation using cost functions: the case of distinct spherical boundaries in ϵ^n. In *International Conference on Robotics and Automation*, pages 1791–1796. IEEE, April 1988.

[72] R. L. Rivest and R. E. Schapire. Diversity-based inference of finite automata. In *Proceedings of the Twenty-Eighth Annual Symposium on Foundations of Computer Science*, pages 78–87, 1987.

[73] R. L. Rivest and R. E. Schapire. Diversity-based inference of finite automata. Technical report, Massachusetts Institute of Technology, 1987.

[74] R. L. Rivest and R. E. Schapire. A new approach to unsupervised learning in deterministic environments. In Pat Langley, editor, *Proceedings of the Fourth International Workshop on Machine Learning*, pages 364–375, 1987.

[75] R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. In *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*, May 1989.

[76] A. J. Robinson and F. Fallside. The utility driven dynamic error propagation network. Technical Report CUED/F-INFENG/TR.1, Cambridge University Engineering Department, Cambridge, England, 1987.

[77] D. E. Rumelhart. Personal communication.

[78] D. E. Rumelhart. Specialized architectures for back propagation learning. Paper presented at the conference on Neural Networks for Computing, Snowbird, UT., April 1989.

[79] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. Technical Report 8506, Institute for Cognitive Science, C-015; University of California, San Diego; La Jolla, CA 92093, 1985.

[80] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, vol.1: Foundations*. Bradford Books/MIT Press, Cambridge, MA, 1986.

[81] D. E. Rumelhart and J. L. McClelland. An interactive activation model of context effects in letter perception: Part 2. the contextual enhancement effect and some tests and extensions of the model. *Psychological Review*, 89:60–94, 1982.

[82] D. E. Rumelhart and J. L. McClelland. On learning the past tenses of verbs. In D. E. Rumelhart, J. L. McClelland, and the PDP research group, editors, *Parallel Distributed Processing: Psychological and biological models*, chapter 18, pages 216–272. MIT press, 1986.

[83] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, vol.1: Foundations*. Bradford Books/MIT Press, Cambridge, MA, 1986.

[84] E. D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5:115–135, 1974.

[85] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal on Research and Development*, pages 210–229, 1959. Reprinted in E. A. Feigenbaum and J. Feldman, editors, *Computers and Thought*, McGraw-Hill, New York, 1963.

[86] R. E. Schapire. Personal communication.

[87] R. E. Schapire. Diversity-based inference of finite automata. Master's thesis, MIT, 1988.

[88] J. H. Schmidhuber. Towards compositional learning with dynamic neural networks. Technical Report FKI-129-90, Technische Universitat Munchen, 1990.

[89] T. J. Sejnowski and C. R. Rosenberg. NETtalk: A parallel network that learns to talk. Technical Report EECS-8601, Johns Hopkins University, Department of Electrical and Computer Engineering, Baltimore, MD, 1986.

[90] D. Servan-Schreiber, A. Cleeremans, and J. L. McClelland. Learning sequential structure in simple recurrent networks. In David S. Touretzky, editor, *Advances in Neural Information Processing Systems 1*, San Mateo, CA, 1989. IEEE Conference on Neural Information Processing Systems—Natural and Synthetic, Morgan Kaufmann Publishers.

[91] D. Servan-Schreiber, A. Cleermans, and J. L. McClelland. Encoding sequential structure in simple recurrent networks. Technical Report CMU-CS-88-183, Carnegie-Mellon University, Department of Computer Science, Pittsburgh, PA, 1988.

[92] S. P. Singh. Transfer of learning by composing solutions for elemental sequential tasks. In *Proceedings of the Machine Learning Workshop*, 1991.

[93] W. S. Stornetta and B. A. Huberman. An improved three-layer back propagation algorithm. In *Proceedings of the IEEE First Annual International Conference on Neural Networks*, pages 637–643, 1987.

[94] R. S. Sutton. *Temporal Credit Assignment in Reinforcement Learning*. PhD thesis, Department of Computer and Information Science, University of Massachusetts at Amherst, 1984.

[95] R. S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

[96] R. S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning*, pages 216–224, San Mateo, CA, 1990. Morgan Kaufmann.

[97] R. S. Sutton and B. Pinette. The learning of world models by connectionist networks. In *Proceedings of the Seventh Annual Conference of the Cognitive Science Society*, Irvine, CA, 1985.

[98] A. Waibel, T. Hanazawa, G. Hinton, K. Shikano, and K. Lang. Phoneme recognition using time-delay neural networks. Technical report, ATR Interpreting Telephony Research Laboratories, 1987.

[99] C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, Cambridge University, Cambridge, England, 1989. Pending.

[100] R. L. Watrous and L. Shastri. Learning phonetic features using connectionist networks: An experiment in speech recognition. Technical report, Department of Computer and Information Science; Moore School; University of Pennsylvania, 1986.

[101] P. J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University, 1974.

[102] P. J. Werbos. Building and understanding adaptive systems: A statistical/numerical approach to factory automation and brain research. *IEEE Transactions on Systems, Man, and Cybernetics*, 1987.

[103] P. J. Werbos. Reinforcement learning over time. In T. Miller, R. S. Sutton, and P. J. Werbos, editors, *Neural Networks for Control*. The MIT Press, Cambridge, MA, 1990.

[104] S. D. Whitehead and D. H. Ballard. Active perception and reinforcement learning. In *Proceedings of the seventh international conference on machine learning*, Austin, TX, June 1990.

[105] B. Widrow and S. D. Stearns. *Adaptive Signal Processing*. Prentice Hall, 1985.

[106] R. J. Williams and D. Zipser. A learning algorithm for continually running fully recurrent neural networks. Technical Report 8805, Institute for Cognitive Science, C-015; University of California, San Diego; La Jolla, CA 92093, 1988.

[107] R. J. Williams and D. Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1(2):270–280, 1989.

[108] R. J. Williams and D. Zipser. Gradient-based learning algorithms for recurrent connectionist networks. In Y. Chauvin and D. E. Rumelhart, editors, *Back-propagation: Theory, Architectures and Applications*. Erlbaum, Hillsdale, NJ, to appear.

[109] I. H. Witten. Approximate, non-deterministic modeling of behavior sequences. *International Journal of General Systems*, 5(1):1–12, Jan 1979.