C H A P T E R 2

PREVIOUS MODELING RESEARCH

This review covers system-identification methods for modeling systems with incomplete state information, including methods for both state estimation and parameter estimation. This chapter is organized into three different sections representing method 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 methods, 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 learning 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 methods according to the state representation employed (see Chapter 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, the delay-coordinate model that employs this representation is defined as follows:

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

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 (from Gill [27]). 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 representation, the system's state is encoded over a fixed number of state variables. In particular, the state-space model that employs this

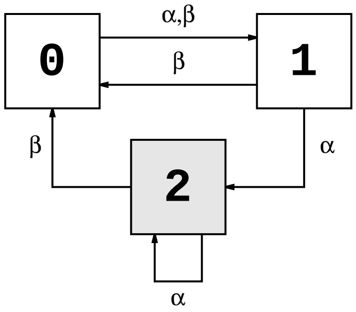

Figure 2.1. An Example of a FSA that Cannot Be Accurately Modeled with a Delay-Coordinate State Representation. This particular FSA has 3 states labeled 0, 1, and 2 and 2 input actions α and β. The shade of the each box codes the single sensation value for that state, where a shaded square codes a sensation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

representation is defined as follows:

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

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

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 methods can be extended to handle the case where the system is stochastic.

The delay-coordinate model 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 function g can be represented as 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 model. Although a state-space model can be transformed into a delay-coordinate model 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 the system's inputs and outputs.

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 employing localist state representations and those emplorying distributed state representations. For the most part, the engineering and connectionist methods utilize distributed state representations, and in automata theory, general systems theory, and computational learning theory the localist state representation predominates.

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

2.1 Methods Based on Linear System Theory

Classical system-identification methods typically assume a certain amount of a priori knowledge about the system being identified. This knowledge is converted into the choice of model inputs, outputs, and constraints on the mathematical form of the model. In order to identify nonlinear systems, traditional methods 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 methods that use the delay-coordinate representation are adaptive filters [103], also known as Deterministic Auto Regressive Moving Average (DARMA) 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:

$$y_j(k) = \sum_{i=0}^{D_u} b_{ji} u_j(k-i), \tag{2.4}$$

where $y_j(k)$ is the j^{th} component of the output of the filter at time k, $u_j(k-i)$ is the j^{th} component of the input to the filter at time $k-i$, each b_{ji} is a parameter, and D_u is one less than the size of the tapped delay line. Widrow and Stearns [103] 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 (independently developed by le Cun [48], Parker [65], Rumelhart, Hinton, and Williams [79], and Werbos [99]) is a generalization of 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:

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

where each 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 delay coordinate state representation. One such model is called the *stochastic linear state-space model.* Consider a system described by:

$$\begin{aligned} \boldsymbol{x}(k) &= \Phi(k-1)\boldsymbol{x}(k-1) + \Lambda(k-1)\boldsymbol{u}(k-1) + \boldsymbol{\nu}(k-1) && (2.6) \\ \boldsymbol{y}(k) &= H(k)\boldsymbol{x}(k) + \boldsymbol{\omega}(k), && (2.7) \end{aligned}$$

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 (Goodwin and Sin [30]).

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 known a priori), and only the current state of the model is unknown. (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 methods which usually involve large matrix inversions.

Kalman filtering can be generalized in a number of ways. The *extended Kalman filter* is a method 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.)

Methods also 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 current value of $\boldsymbol{x}$. One method 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 method for performing both parameter and state estimation involves switching between standard parameter estimation and Kalman filtering. Initially, the Kalman filter produces the estimated state, $\hat{\boldsymbol{x}}_0$, 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 $\hat{\boldsymbol{x}}_1$. 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 at time step $k-1$ (which is the action chosen by the agent at time stek $k-1$), 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 and 2.4 for more details about SLUG, and see Chapter 2.4 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, for each action $\boldsymbol{u}(k-1)$, $\Phi(\boldsymbol{u}(k-1))$ 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 $\Phi_{ij}(\boldsymbol{u}(k-1))$ containing a 1 effectively selects the appropriate state variable j from which to update a given state variable i when the input is $\boldsymbol{u}(k-1)$.

2.2 Connectionist Modeling Methods

In order to mimic a dynamical system, a network must represent the system's state. Two methods 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 differ 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 methods presented are gradient descent algorithms; the weights are adjusted in proportion to estimates of the gradient of the performance criterion with respect to the weights. The algorithms differ in the accuracy of the approximation to this gradient estimates.

2.2.1 Delay-Coordinate Models

McClelland and Elman presented 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, 80], in which the activations of the units are updated by both a bottom-up and top-down process. The extended architecture includes a tapped delay line for storing all the speech data from the beginning of an utterance. In their system the delay line must be 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 method, 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 [87] 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 [96] introduced 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 between events, and the features it learns are invariant under time translations. The architecture is a tapped delay line design where each unit receives a tapped delay line of past values of the outputs of units feeding into it. They improve on the linear adaptive filter method described by Equation 2.4 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" method 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 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 be *directly* applied only to nonrecurrent networks. It requires an "unfolding" procedure to handle recurrent networks (see Rumelhart, Hinton, and Williams [79] for a full account). Unfolding a recurrent network converts it into a feed-forward network by making copies of each unit for each discrete time step. Each recurrent link connecting unit α to unit β becomes a feed-forward connection terminated on a copy of unit β at the next time step. Figure 2.2 shows an example of

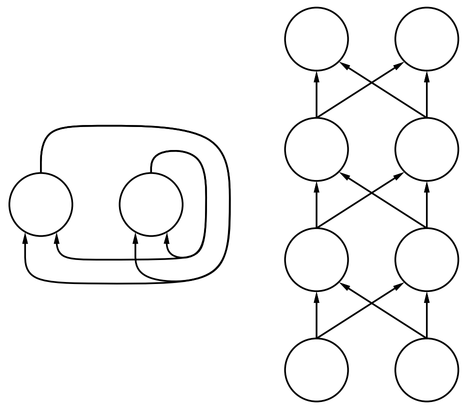

Figure 2.2. An Example of the Unfolding Procedure which Turns Recurrent Networks into Feed-Forward Networks.

this procedure applied to a fully recurrent network consisting of two units. The resulting unfolded network is now a feed-forward network to which 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 2.4 for more details).

Back-propagation provides 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 provides the exact gradient. Unfortunately, because the network can be unfolded only a finite number of times, this method applies 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 in which 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 [98] used back-propagation in time in an architectures that learns 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 of the representations formed by the network. Notice that this architecture cannot represent arbitrary FSA because of the simplicity of the feedback connections.

Pearlmutter [66] presented 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 algorithm differs from back-propagation in time in that his networks are defined in terms of differential equations.

Bachrach [3, 4] presented methods for training simple reverberatory circuits to act as memory devices. Specifically, Bachrach examined 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. This algorithm operates in real-time, computes the exact gradient without unfolding the network, and allows for arbitrarily old inputs to influence weight changes. It uses fixed feedback weights and a symmetric logistic output function (ranging from -1 to $+1$), which overcomes some of the limitations of single unit feedback systems discussed by Watrous and Shastri [98]. The effective search effort in weight space is reduced because each unit starts out with two well-behaved symmetric attractors. Without more complicated machinery, sticky-bits are limited in their representational power. In particular, the combination of sticky-bits and a feed-forward network cannot represent arbitrary FSA's; more feedback connections are necessary.

Mozer [55] derived a similar real-time back-propagation rule for self-recurrent units. He used a variation of the sticky-bit learning algorithm applied to units with tapped delay lines and outputs that were exponentially decaying traces of the usual logistic outputs. He derived real-time gradient algorithm for this output function, and used this algorithm to learn "wicklefeatures" similar to the handcrafted input features of the past-tense model of Rumelhart and McClelland [81]. The main drawback with this method is that Mozer's units necessarily have limited memory because eventually their outputs decay to zero. In contrast, the sticky-bit can latch onto an event indefinitely.

Williams and Zipser [104], Bachrach [3], Robinson and Fallside [76], and others independently developed a real-time learning algorithm for recurrent networks that computes the exact gradient without unfolding. This algorithm applies to continuously-running arbitrarily-connected networks. 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 [26] presented a higher-order recurrent network for grammatical inference. The network is a higher-order design but differs from

SLUG in a number of respects. Instead of having the input select a linear function, the input selects a particular semi-linear function. 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, and $W_{a(k)}$ is the weight matrix associated with the action chosen at time k. The output of their network is interpreted as the acceptance or rejection of the input string and not as a prediction of the sensations at the next time step. Their learning algorithm resets the state of the network before each string is presented and updates the weights only after each string is presented when the teacher declares whether or not the string is in the language. Finally, state estimation is not performed by their algorithm.

Jordan [34] investigated 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.

Jordan considers problems requiring learning desired sequences of outputs. The outputs of context units form exponentially decaying traces of the outputs of the

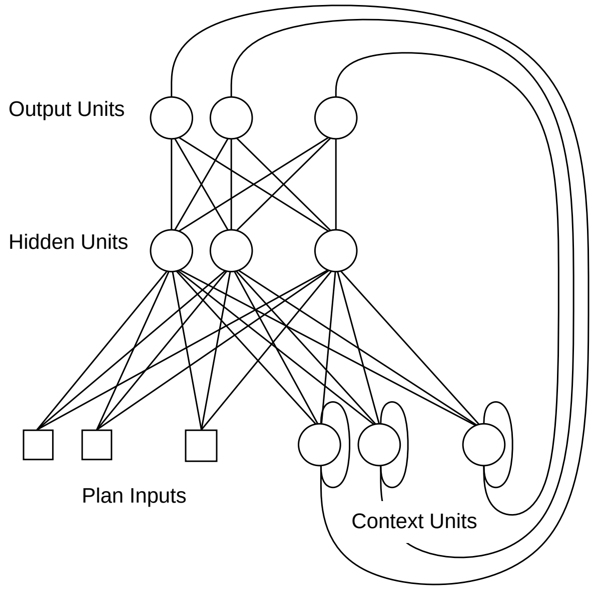

Figure 2.3. The Jordan Feedback Architecture.

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 the tasks studied in this thesis. Whereas Jordan focuses on problems in which the inputs are fixed while the network generates a sequence, the focus of this thesis is on problems in which 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 arbitrary FSA's.

Elman's [21] recurrent network, shown in Figure 2.4, is based on Jordan's architecture. 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 represent all FSA. Like Jordan's learning algorithm, Elman's learning algorithm approximates the

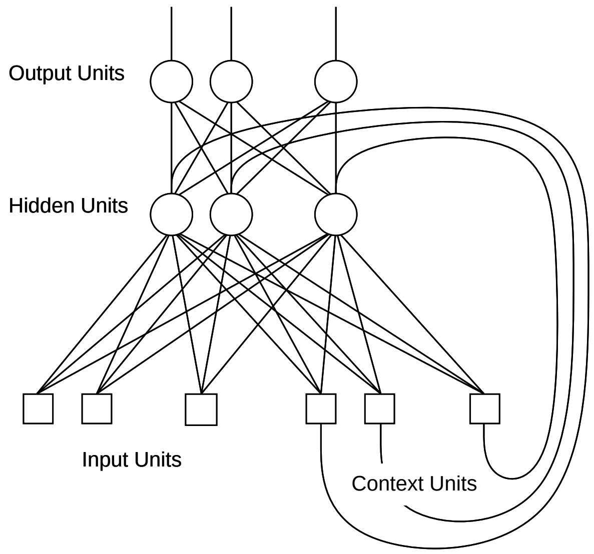

Figure 2.4. The Elman Feedback Architecture.

gradient by back-propagating only through the feed-forward section of the network. This can make learning in this architecture very slow and unreliable for difficult tasks. Servan-Schreibner, Cleereman, and McClelland [88] applied the Elman network to the problem of learning a regular language. Their method 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 (described in the previous section) in three main respects. First, SLUG is a time-varying linear system, and methods 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. The state units are readily interpretable within the update graph formalism, and many theoretical results exist that lend intuition and rigor. Third, 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.

2.3 Finite State Automaton Identification

This section describes a variety of methods for learning FSA models of environments. These methods are meant as a small sampling of the large number of methods from automata theory, general systems theory, and computational learning theory. A common property of the methods from automata theory and general systems theory is poor computational complexity in both space and time. The first three methods involve exhaustive search through the space of FSA's. The majority of the methods presented in this section learn state transition graphs. The last two methods from general systems theory are more computationally practical and involve delay-coordinate representations. Some of the methods to be presented produce models which are generalizations of FSA. When the state transition mapping is one-to-many, an FSA is called a *non-deterministic FSA*, or an *NFSA*. When the state transition mapping is stochastic, the FSA is called a *probabilistic FSA*.

2.3.1 Automata Theory

Kohavi [44] described a method 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 is assumed to have 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 method 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 that uniquely identifies the state reached by the automaton at the end of the homing sequence. This method is obviously impractical because the number of machines is staggering even for small n.

Booth [11] also described 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. 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 can 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] described a method, called *ATOM*, for inferring probabilistic FSA. This method 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 with 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 [107] presented a method for learning NFSA which 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 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 that occur consecutively. The model can then be reduced by coalescing states having 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] described a method for learning a probabilistic FSA. His method also 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. Procedures are described for estimating the state transition probabilities and for choosing the best size for the activity matrix based on the reduction of uncertainty the delayed variables provide. Finally methods 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 its 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, 85] 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, 85] also obtain a polynomial-time algorithm through the method 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 in which the algorithm has no means of resetting the machine to the start state. They describe an algorithm that with 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 methods 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 an FSA from lexicographically ordered examples. The basic algorithm constructs an

FSA by iteratively examining the lexicographically ordered positive and negative examples. At each step, 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 methods 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 methods employ a discrete-localist state representation. In comparison SLUG utilizes a continuous-distributed state representation. Second, most of the models produced by these methods 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 methods reviewed are primarily serial algorithms, because 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 Summary

This chapter presented modeling research relevant to this dissertation and on problems similar to those addressed in this dissertation. A variety of research was covered including research from engineering, connectionism, automata theory, general systems theory, and computational learning theory. SLUG has been inspired by these various methods; SLUG is a hybrid method, borrowing methods from both engineering and FSA identification.