RECURRENT NEURAL NETWORKS FOR GRAMMATICAL INFERENCE

Thesis by
Zheng Zeng

In Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy

California Institute of Technology
Pasadena, California

1994

(Submitted May 10, 1994)

To my parents

Acknowledgments

This work would not have been possible without the help of many people. First of all, I would like to thank my advisor, Dr. Rodney Goodman, for his continuous support, guidance and encouragement throughout my years at Caltech. I am especially grateful to Dr. Padhraic Smyth for countless stimulating discussions and insightful suggestions on this work and other technical issues, his experience and consciousness as a scientist have truly benefitted me in many ways. Thanks are owed to many faculty members of the institute, especially from the Department of Electrical Engineering, whose teachings and love of science have been a source of inspiration for me.

I would like to thank all my friends and colleagues in the MicroSystems Lab during my four and a half years of study here, for many fun discussions, technical and otherwise, and for a very relaxed and friendly environment. My thanks to everyone for putting up with my time-consuming background processes on the machines, especially towards the end of the thesis writing. I am indebted to Jeff Dickson and Bhusan Gupta for their technical help in preparing the manuscript and for sacrificing their weekends to review the thesis. Special thanks to Chuck Higgins, whose LaTeX macros made the formatting of the thesis so much easier. To our wonderful secretary, Bette Linn, I am grateful for all the help and support she provided throughout the years.

Much love and thanks are given to my husband, Xiaofei Huang, with whom I shared many helpful discussions, and whose love, understanding and support I cannot appreciate more.

Finally, for making it all possible and worthwhile, I would like to thank my parents, to whom this thesis is dedicated to, for their unconditional love and guidance throughout my life. I would not have been able to obtain the opportunity for my study and to continue to finish it here in this wonderful institute and group without their unselfish support and encouragement.

Abstract

In this thesis, various artificial recurrent neural network models are investigated for the problem of deriving grammar rules from a finite set of example "sentences." A general discrete network framework and its corresponding learning algorithm are presented and studied in detail in learning three different types of grammars.

The first type of grammars considered is regular grammars. Experiments with conventional analog recurrent networks in learning regular grammars are presented to demonstrate the unstable behavior of such networks in processing very long strings after training. A new network structure to force recurrent networks to learn stable states by discretizing the internal feedback signals is constructed. For training such discrete networks a "pseudo-gradient" learning rule is applied.

As an extension to the discrete network model, an external discrete stack is added to accommodate the inference of context-free grammars. A composite error function is devised to deal with various situations during learning. The pseudo-gradient method is also extended to train such a network to learn context-free grammars with the added option of operating on the external stack.

Another extension to the discrete network structure is made for the purpose of learning probabilistic finite state grammars. The network consists of a discrete portion which is intended to represent the structure of the grammar, and an analog portion which is intended to represent the transition probabilities. Two criteria for the network to verify the correctness of its solution during training are proposed. Theoretical analysis of the necessary and sufficient conditions for the correct solution is presented.

Experimental results show that the discrete network models have similar capabilities in learning various grammars as their analog counterparts, and have the advantage of being provably stable.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

1.1 What is Grammatical Inference?

1.1.1 Languages and Grammars

In everyday life, we depend largely on various languages to accomplish communication, be it natural languages such as English or Chinese for communication between people, or computer languages such as C or UNIX for communication between people and machines.

For every language, there are usually certain corresponding rules we call grammar that anyone who uses it has to obey, so that communication among different parties can be established on a common ground. The grammar, or the set of rules for the language, can be as large as hundreds or even thousands for the case of natural languages, or only a few dozen for the case of computer languages. The former can be very vague, some of the rules can be broken at times without affecting perfect understanding among the communicating parties, while the latter are generally very strict, any broken rule may lead to misunderstanding, or even total failure of understanding, e.g., failure to compile.

Indeed, one can think of very many examples of languages other than the two types mentioned above, such as sign language for the deaf, "body language" in sociology and psychology, or even mathematical or chemical formulae, and drawings for circuit designs, etc.

In this thesis, however, of particular interest is a subset of the so called formal languages, a set of abstract concepts behind natural languages and grammars, which have provided the theoretical basis for computer languages that today guides the design of compilers[HU79].

In the remaining part of this thesis, the word "string" will be used instead of "sentence" to be consistent with formal language theory conventions.

In formal language theory, a language L is defined to be a set of strings, over an alphabet Σ, i.e., $L \subseteq \Sigma^*$, where Σ^* is the closure of Σ. For a language that has a set of specific grammar rules corresponding to it, a string is defined to be a member of the language *if and only if* it is derivable from that set of rules.

1.1.2 Chomsky Hierarchy for Formal Languages

A grammar is formally defined to be a quadruple $G =< \Omega, \Sigma, S, P >$, where:

- Ω is a (nonempty) set of nonterminals.
- Σ is a (nonempty) set of terminal symbols and $\Omega \cap \Sigma = \emptyset$.
- S is the designated start symbol and $S \in \Omega$.
- P is a set of productions of the form $\alpha \rightarrow \beta$, where $\alpha \in (\Omega \cup \Sigma)^+, \beta \in (\Omega \cup \Sigma)^*$.

Given a grammar G, the language generated by G, denoted by $L(G)$, is given by $L(G) = \{x | x \in \Sigma^* \wedge S \rightarrow^* x\}$. That is, the language $L(G)$ contains and only contains all strings x, such that x consists of symbols from the alphabet Σ, and is derivable from the start symbol S by applying any number of the production rules from P.

We will use the words "language" and "grammar" interchangeably from now on, assuming the understanding that when a grammar is defined, the corresponding language is uniquely defined also, and vice versa.

Chomsky[Cho59] defined a hierarchy of formal languages and grammars, by differentiating different forms of production rules:

Type 3 grammars or regular grammars: the most restricted type of grammars. For a regular or type 3 grammar G, either the production rules in P all have the form $A \rightarrow xB$, for which it is also called the right-linear grammar, or they all have the form $A \rightarrow Bx$, for which it is also called the left-linear grammar, where $A \in \Omega, B \in (\Omega \cup \lambda)$ (λ is the empty string), and $x \in \Sigma^*$.

Type 2 grammars or context-free grammars:
For a context-free or type 2 grammar G, all production rules in P have the form $A \rightarrow \beta$, where $A \in \Omega, \beta \in (\Omega \cup \Sigma)^+$. In addition, the grammar can also allow the production of the empty string.

Type 1 grammars or context-sensitive grammars:
For a context-sensitive or type 1 grammar G, all production rules in P have the form $\alpha \rightarrow \beta$, where $\alpha \in (\Omega \cup \Sigma)^+, \beta \in (\Omega \cup \Sigma)^+$, and $|\alpha| \leq |\beta|$. In addition, the grammar can also allow the production of the empty string.

Type 0 grammars or unrestricted grammars:
For an unrestricted or type 0 grammar G, the production rules in P are not restricted in

any sense.

Chomsky has shown that the class of languages of a certain type fully includes all higher-numbered language types.[HU79, CL89]

In this thesis we will only consider grammars of type 2 or 3, discussed respectively in Chapter 3 and 2, and a special variation of type 3 grammars, probabilistic regular grammars, discussed in Chapter 4.

1.1.3 The Task of Grammatical Inference and Its Uses

The inference of grammars is not only an invention of abstract formal language theory. For a natural language, the basic findings of linguists, mostly well accepted grammar rules, have long been used widely in the teachings of the language to either native young children or foreigners.

In formal languages, various automatic inference algorithms are designed to play the role of the "linguist." Although the goal of inferring the underlying grammar rules are the same for every inference algorithm, the assumed condition under which such inference is carried out varies. Some assume only a given finite set of example strings are known. Some assume a black box that contains a "language recognition device" is available, so that the inference algorithm can actively query such a device on whether or not a particular string belongs to the language being considered. Some others assume that a "language generator" is available, which can randomly generate various strings that belong to the language an infinite number of times. Still others assume a special "teacher" of the language exists, who gives out useful "advice" as the inference is being carried out.

While all of the above conditions can map to comparable situations in the case of natural language learning, in this thesis, the assumed condition is the most stringent one (for most of the cases): only a finite set of example strings is available. The remaining few other cases also assume that a special "teacher" in a restricted sense exists.

Thus, the general problem of grammatical inference considered in the whole thesis can be defined as follows:
Given a set of known example strings, labeled as "grammatical" (or "legal") or ungrammatical (or "illegal") according to the underlying language, derive the underlying grammar rules so that given any new string that is not contained in the known examples, a decision of whether it is grammatical or ungrammatical can be made correctly.

Of course, to accomplish the goal, one has to assume that the given example set contains sufficient information about the language and its grammar being learned. Otherwise it would be an impossible task. In addition, it is reasonable to assume that the alphabet of the language is known, because otherwise, it is trivial to obtain from the given example set.

The importance of the study of formal language and inference is two-fold: in the theoretical sense, computer scientists and mathematicians have been studying the topic to explore the properties of the languages and grammars, which in turn may help in the design of computers and systems[HU79, CL89]; in the practical sense, a good inference algorithm can be applied to the modeling of real world sequence analysis problems. An excellent example is the use of formal language modeling in DNA sequence analysis, which has uncovered very interesting properties and structures of DNA[Sea92, Hea87].

Theoretical aspects of grammatical inference have been studied extensively in the past [Ang72, Ang78, Gol72, Gol78]. A variety of direct search algorithms have been proposed for learning grammars from positive and negative examples (strings) [AS83, Fu82, Mug90, KS88, Tom82]. In this thesis, however, we explore the application of an alternative and very different tool, i.e., recurrent neural networks, to the problem of grammatical inference.

1.1.4 Important Results in Formal Language Inference Theory

The problem of formal language and its inference has been extensively studied and well understood since the 1970's. A whole system of theory has been established[Gol78, HU79, AS83, Mug90].

It is not necessary to go much deeper in this thesis into formal languages and their properties — a task which would take a whole book to accomplish. Instead, only several very important and related results will be presented here.

Of great interest are the following:

1. There is a one-to-one correspondence between regular grammars and finite-state-automata[HU79].

2. There is a one-to-one correspondence between context-free grammars and finite-state pushdown automata[HU79].

3. Any non-deterministic finite-state-automaton can be transformed into an equivalent

deterministic finite state automaton. The class of languages recognized by nondeterministic finite automata is exactly the same class of languages recognized by deterministic finite automata[HU79].

4. The inference of a smallest regular grammar from a finite set of positive examples only is impossible[Gol78].

5. The inference of a smallest regular grammar from a finite set of positive and negative examples is an NP-complete problem [Gol78, Ang78].

Items number 1 and 2 tell us that we can try to find the equivalent automaton in solving an inference problem for type 2 or type 3 grammars, if it turns out to be easier. The definitions and examples of finite state automata and pushdown automata will be discussed in detail in Chapter 2 and 3 respectively. Item number 3 tells us that if we can solve the problem of deriving deterministic finite automata from examples, then we effectively have solved the problem of deriving nondeterministic finite automata from examples, and therefore have solved the inference problem for the whole class of regular grammars also. Item number 4 warns us against attempting to make the inference from positive data only. Item number 5 tells us that we can only hope to find a solution to the inference problem that is exponential in time to the complexity of the language, or alternatively, to find a bigger than minimum solution that has less than exponential time complexity.

1.2 What are Feedforward and Recurrent Neural Networks?

1.2.1 Artificial Neural Networks, Back-Propagation Networks

We now turn from the discussion of formal languages to a seemingly unrelated field: artificial neural networks. Ever since the "re-discovery" of the capabilities of perceptrons in the 80's[Hop82, RMtPRG86], artificial neural networks have become the topics of extensive study for researchers in a wide range of disciplines: computer science, electrical engineering, physics, economics, biology, etc. With them, came various models of artificial neural networks, as well as learning algorithms which may or may not be biologically plausible as was intended in the initial stage. One of the most widely used structures, the "back-propagation" networks[RHW86], is the most relevant to the contents of this thesis, and thus will be the only type of model considered here.

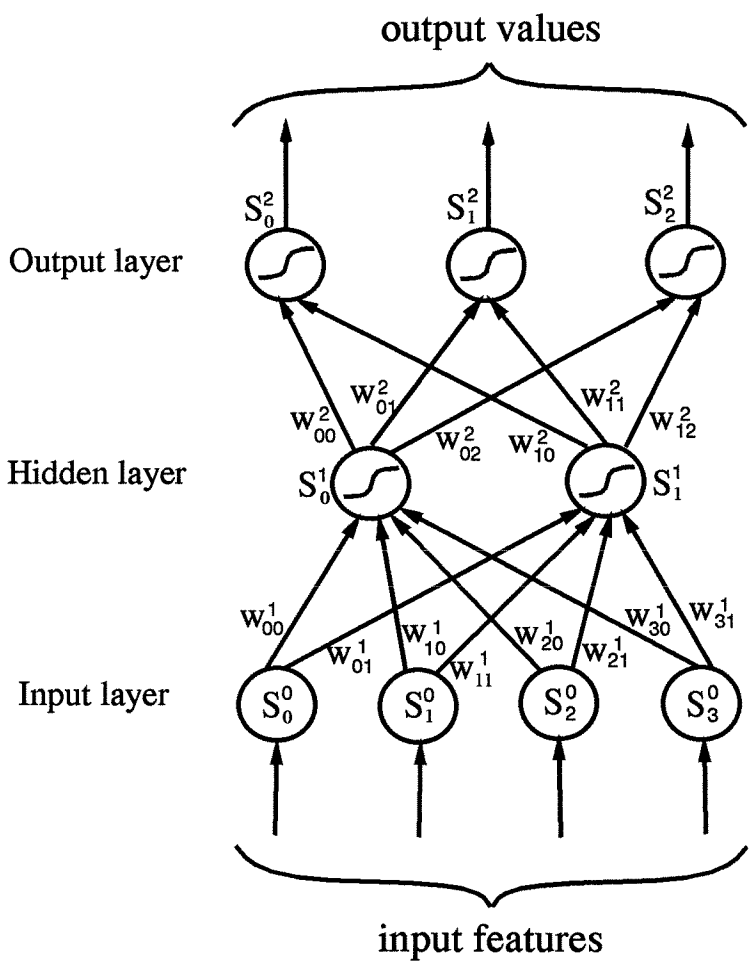


Figure 1.1: A typical back-propagation network

A typical feedforward back-propagation network is shown in Fig. 1.1. The network shown has 3 "layers:" layer 0, the input layer, takes in external input patterns; layer 1, the hidden layer, does computation internally and is not directly connected to the outside world; layer 2, the output layer, produces output signals to the outside. Note there can be more than one hidden layer or sometimes no hidden layer in the structure. Layer l accepts inputs from its previous layer, layer $l-1$, only, and its output is fed into the next layer, layer $l+1$, only. So in the operation of the network, information is processed from the input layer forward, through the network to the output layer.

Define S_i^l to be the activation of unit i in layer l, w_{ij}^l to be the connection weight between unit i of layer $l-1$ to unit j of layer l.

The operational equations of the back-propagation network are:

$$S_i^l = f(\sum_j w_{ij}^l S_j^{l-1}), \quad \forall i, l, \tag{1.1}$$

where f is called the activation function of the network. The most widely used form of f is the sigmoid function:

$$f(x) = \frac{1}{1+e^{-x}}. \tag{1.2}$$

In some cases, different layers can have different activation functions, or even units within the same layer can have different activation functions, depending on the problem at hand.

This structure provides a mapping from the input vector space into the output vector space, and thus is mostly used for function approximation, classification, or pattern recognition tasks. Whatever the problem, the user is asked to compose a "training set," which contains a number of examples of input and desired output pairs. In "training," the network is presented with the "training set." An algorithm is used to train the network to learn from these example pairs and generalize, by adjusting its connection weights between the layers. An error measure is used to evaluate how well the network outputs fit the desired outputs. The hope is that after sufficient training in reducing the network's error on the training set, the connection weights can be fixed, and when a new input pattern (previously unseen by the network) is presented, it can produce an output pattern that is satisfactory, that is, consistent to the general "trends" in the given training set.

The number of nodes in the input and output layers are defined by the problem, while the number of hidden nodes has to be pre-selected before training. The connection weights w_{ij}^n are initially randomized. The most often used error measure is the mean-squared-error(MSE):

$$MSE = \frac{1}{2}(network\ output - desired\ value)^2. \tag{1.3}$$

We will see another popularly used form of error measure in Chapter 4.

The most widely and successfully used training algorithm for this type of network is the "back-propagation" algorithm. In essence, it computes an error between the desired output and the network's true output, then calculates the gradients of the error with respect to each weight, in a manner that the error is propagated back to the input layer, then the whole set of weights is modified according to the gradients in the downhill direction of the error surface in the high-dimensional weight space[RHW86].

1.2.2 Recurrent Networks

The back-propagation network model discussed in the previous subsection is a typical feedforward network model, where the information flow is unidirectional: from input layer through the hidden layer(s) to the output layer. Thus the outputs depend on what are present at the input layer only. There is another type of network model called recurrent networks, where the information flow can be considered bidirectional, or "circular," or recurrent. A typical layered recurrent network is shown in Fig. 1.2.

The differences of this model from the one shown in Fig. 1.1 are the extra time-delayed feedback links. The feedback links are not restricted in the sense that they can be from any layer to any other layer, including itself. Note that in this type of model, any feedback link has a delay element, represented by the digital signal processing symbol Z^{-1}, on it, which means that no instantaneous feedback is allowed. The use of Z^{-1} symbol also implies that we assume a discrete time space. Very often, the feedback links themselves can have connection weights associated to them. In Fig. 1.2, for the sake of clarity, not all possible feedback links are shown, and the labels for connection weights on the feedback links are omitted.

Using similar notations as in the previous subsection, let $S_i^l(t)$ be the activation of unit i in layer l at time step t, w_{ij}^l be the connection weight from current unit $S_j^{l-1}(t)$ to unit

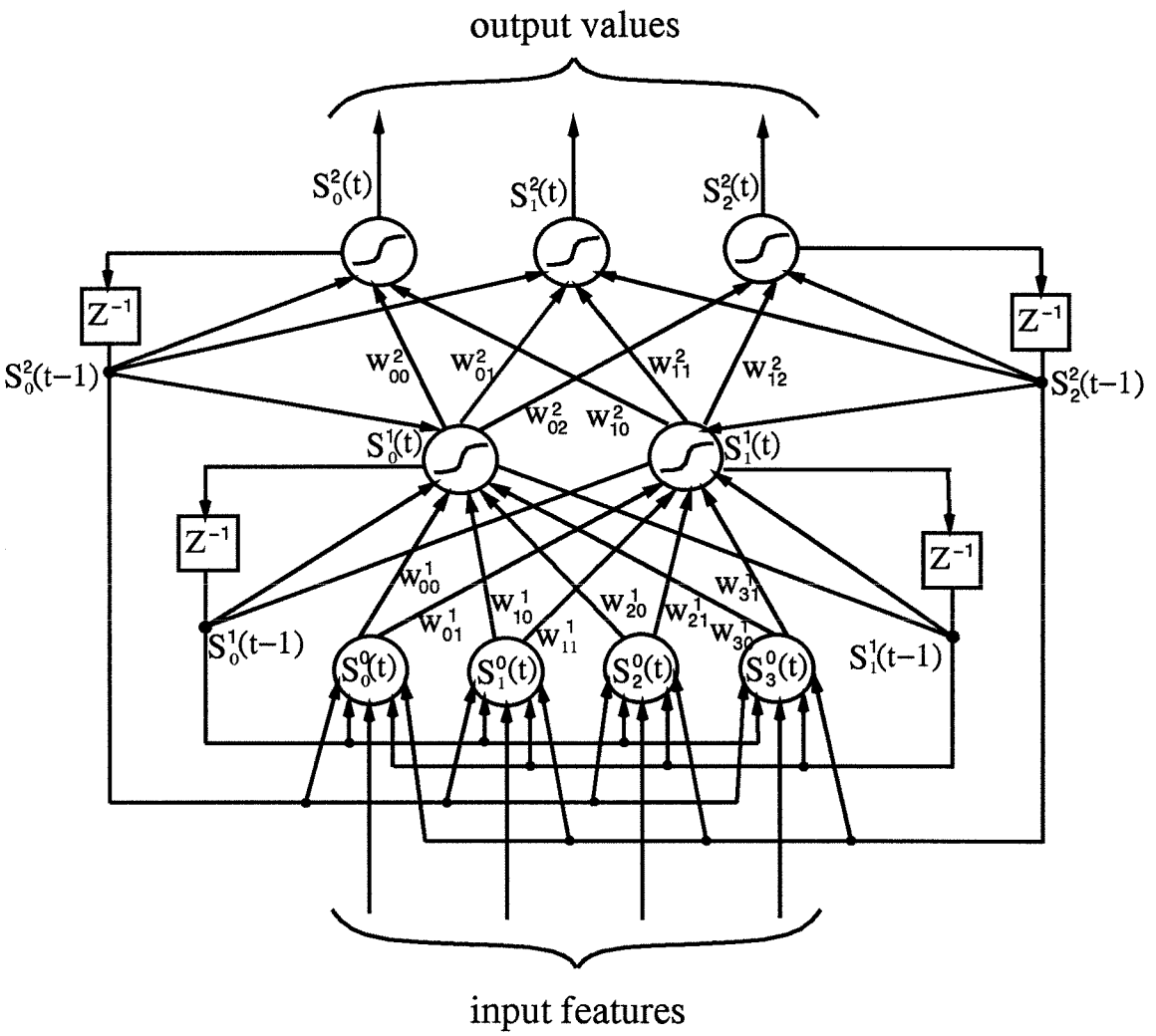


Figure 1.2: A typical layered recurrent network

$S_i^l(t)$, and $\hat{w}_{ij}^{lk}$ be the connection weight from the unit of previous time step $S_j^k(t-1)$ to unit $S_i^l(t)$. The operational equations are then:

$$S_i^l(t) = f(\sum_j w_{ij}^l S_j^{l-1}(t) + \sum_{k,j} \hat{w}_{ij}^{lk} S_j^k(t-1)), \quad \forall i, l, t. \tag{1.4}$$

Note that the above is only one out of very many recurrent structures studied by researchers in the field, other important structures include the Hopfield networks[Hop82], ADALINE networks[Wid62], Kohonen's self-organization networks[Koh88], ART networks [CG87], cascade correlation networks[Fah91], etc. Since the scope of this thesis only covers the structure presented above, henceforth, the phrase "recurrent networks" will only refer to the above structure.

1.2.3 Feedforward vs. Recurrent Networks

An analogy to feedforward networks in signal processing would be FIR filters, where the input signals travels through the filter without any feedback. Similarly, the recurrent network analogy in signal processing would be IIR filters, where feedback links exist.

The advantage of recurrent networks over feedforward networks lies in the power introduced by the feedback links, with which information from time step $t-1$ can still be kept until time step t, or infinitely further into the future. Therefore, recurrent network structures are mostly used for sequential information processing, problems that have time dependencies, or require some sort of "memory." On the other hand, feedforward networks are advantageous in solving straight input/output mapping problems.

As will be seen in the next subsection and the following chapters, recurrent networks are not as well studied and understood as feedforward networks (a fact that was one of the motivations of this thesis), and they are much harder to train.

1.2.4 Overview of Previous Work

With the advent of the concepts of artificial neural networks, much work has been carried out on the study and the establishment of theory of various models, especially on feedforward networks. Here, only the most relevant findings will be presented.

For back-propagation type of networks, the following statements are true:

- For networks with hard-limiting threshold functions as activation functions (i.e., the so-called perceptrons), and no hidden layers, the representation power is very limited:

they cannot properly categorize inputs whose classes are not linearly separable. A good example is their inability to solve the simple XOR problem [MP69].

- For perceptron networks with at least one hidden layer, any Boolean function can be realized, given enough hidden units, but no effective training algorithm has been found[MP69, RMtPRG86].

- For networks with sigmoid activation functions, and at least one hidden layer, any function can be approximated arbitrarily closely given enough number of hidden units, and the back-propagation algorithm can be very effective in training. However, it also can be trapped in local minima at times[RHW86, Lip87].

These results will be useful in Chapter 4, where we discuss the inference of stochastic regular grammars, the learning of both the structure of the language and the probabilities associated with it.

Similar statements of theoretical results cannot be made however, for the case of recurrent networks, where interests are more recent. Much of the study of their basic properties and algorithms is still under way and theoretical analysis is much more difficult. Nevertheless, there have been many investigations on various models, training algorithms, and empirical experiments, as well as analysis on their applications to a wide range of problems: grammatical inference[CSSM89, Elm90, Jor86a, GMC+92a, WK92, ZGS93, ZGS94], time sequence analysis/forecasting[CA91, Sch92], dynamic system modeling/estimation[Lev91, Sør91], state space trajectory learning[Pea89], and temporal association[BC91, HLvH91, KPR91], etc.

As the title of the thesis implies, we will concentrate on recurrent network models and algorithms that address the grammatical inference problem specifically. It is often the case, however, that the model and algorithm being used are applicable to other problems as well. A variety of network architectures[Elm91, FGS92, GSC+90, Pol91, ZGS93], and learning rules[RHW86, WZ89, LC91, SBJ91, dVP91, ZGS93] have been proposed for learning simple grammars. All have shown the capability of recurrent networks to learn different types of simple grammars from examples.

Brief overviews of relevant previous work will be made at the beginning of each of the chapters which follow.

Chapter 2
Inference of Regular Grammars

In this chapter we focus on studying a recurrent network's behavior in learning type 3 or regular grammars, which are the simplest (or most restricted) type of grammar in the Chomsky hierarchy discussed in Chapter 1.

The purpose of the study is to obtain a better understanding of recurrent neural networks, their behavior in learning, and their internal representations, which in turn may give us more insight into their capability for fulfilling other more complicated tasks.

This chapter is organized as follows: Section 2.1 reviews regular grammars and the corresponding finite state automata. Section 2.2 discusses popular first-order recurrent network structures, and their performance in learning regular grammars. Section 2.3 introduces the second-order recurrent network structure. Section 2.4 demonstrates through experiments and analysis the unstable state behavior of analog second-order recurrent networks. Section 2.5 introduces a discrete second-order recurrent network structure, that is capable of forming stable states. Section 2.6 describes the pseudo-gradient learning algorithm for the discrete recurrent network structure. Section 2.7 presents experimental results in learning regular grammars using the discrete network structure and the pseudo-gradient training. Section 2.8 shows the results of empirical investigations on the effectiveness of the pseudo-gradient learning. Section 2.9 gives some general results on the capacity of analog and discrete networks. Section 2.10 summarizes the chapter.

2.1 Regular Grammars and Finite State Machines

As stated in Chapter 1, regular grammars have been shown to have a one-to-one correspondence to finite state machines. Furthermore, they have a one-to-one correspondence to deterministic finite state machines [HU79]. Thus a regular language can be equivalently defined as the language accepted by its corresponding deterministic finite state acceptor: $P =< \Sigma, U, u_0, \delta, F >$, where

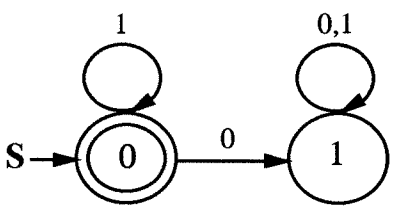
- Σ is the input alphabet.
- U is a finite nonempty set of states.

- u_0 is the start (or initial) state, $u_0 \in U$.
- δ is the state transition function; $\delta : U \times \Sigma \rightarrow U$.
- F is the set of final (or accepting) states, $F \subseteq U$.

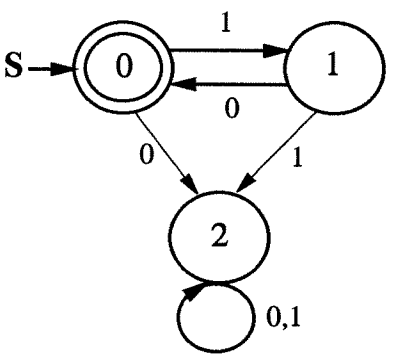
In the experiments described in this chapter we use the following grammars:

- Tomita grammars [Tom82]:
  - #1 — 1*.
  - #2 — (10)*.
  - #3 — any string without an odd number of consecutive 0's *after* an odd number of consecutive 1's.
  - #4 — any string not containing "000" as a substring.
  - #5 — even number of 0's and even number of 1's.
  - #6 — any string such that the difference between the numbers of 1's and 0's is a multiple of 3.
  - #7 — 0*1*0*1*.
- Simple vending machine [CL89]: The machine takes in 3 types of coins: nickel, dime and quarter. Starting from empty, a string of coins is entered into the machine. The machine "accepts," i.e., a candy bar may be selected, only if the total amount of money entered exceeds 30 cents.
- A 10-state machine shown in Fig. 2.1(i), which was used in [GMC+92b].

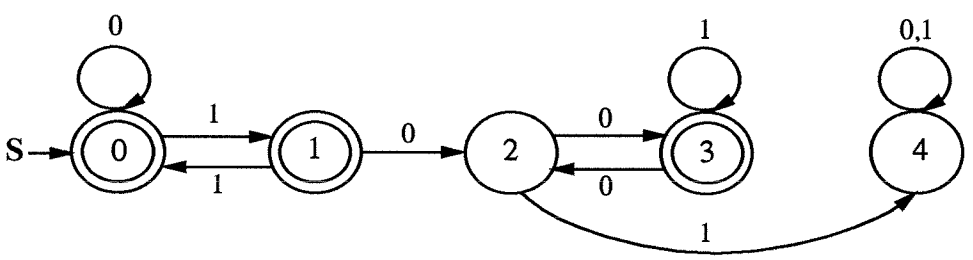
Figures 2.1(a) — (i) show the corresponding equivalent finite state acceptors for the grammars listed above. The start state is indicated by a free arrow with an "S." Double circled states are "accept" states, which means the string processed up to here is a legal one. Single circled states are "reject" states, which means the string processed up to here is an illegal one. The Tomita grammars have been used by researchers in the field as benchmark problems for testing grammatical inference algorithms. The vending machine problem was used as a test for non-binary alphabet grammars (in this case, the alphabet size is 3). The 10-state machine example is the most complex one in the experiments, and was used to test the scaling ability of our inference algorithm.



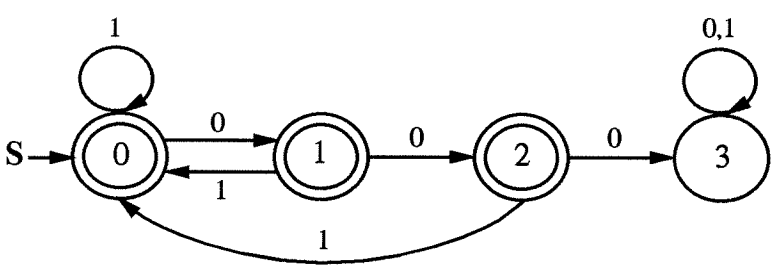
(a) Tomita Grammar #1.



(b) Tomita Grammar #2.

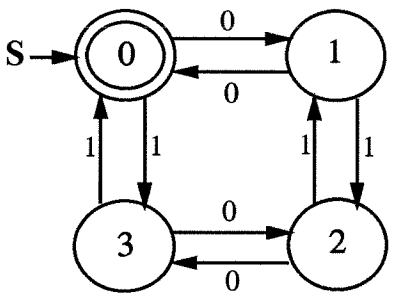


(c) Tomita Grammar #3.

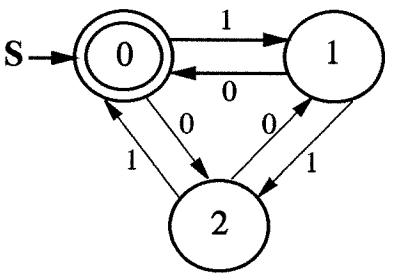


(d) Tomita Grammar #4.

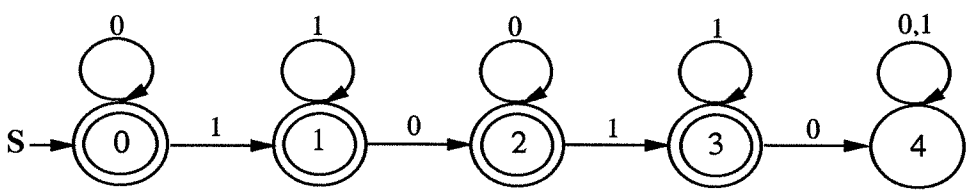
Figure 2.1: Finite state machine representations of regular grammars.



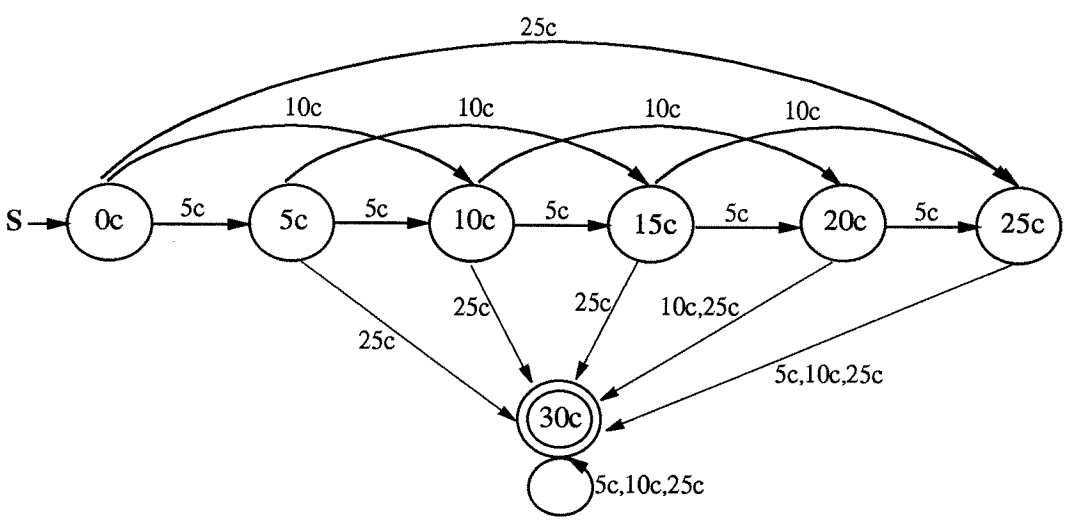
(e) Tomita Grammar #5.



(f) Tomita Grammar #6.

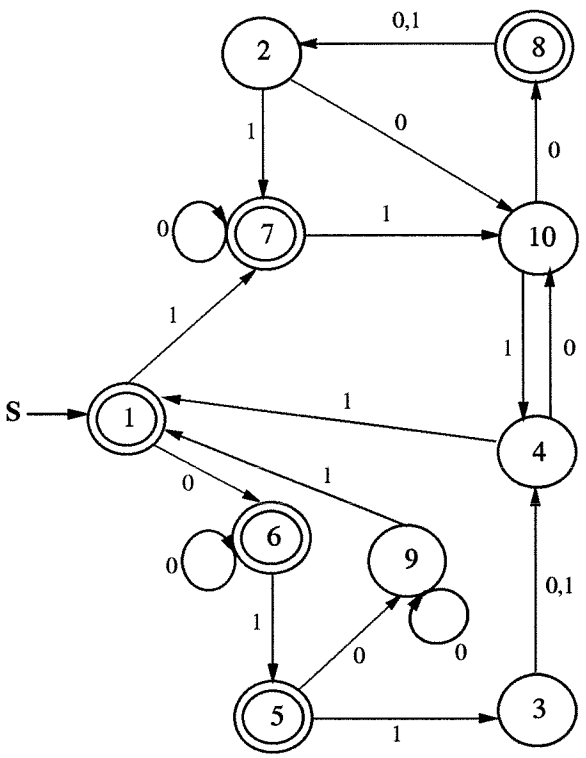


(g) Tomita Grammar #7.



(h) Simple vending machine model.

Figure 2.1: Continued.



(i) A 10-state machine model.

Figure 2.1: Continued.

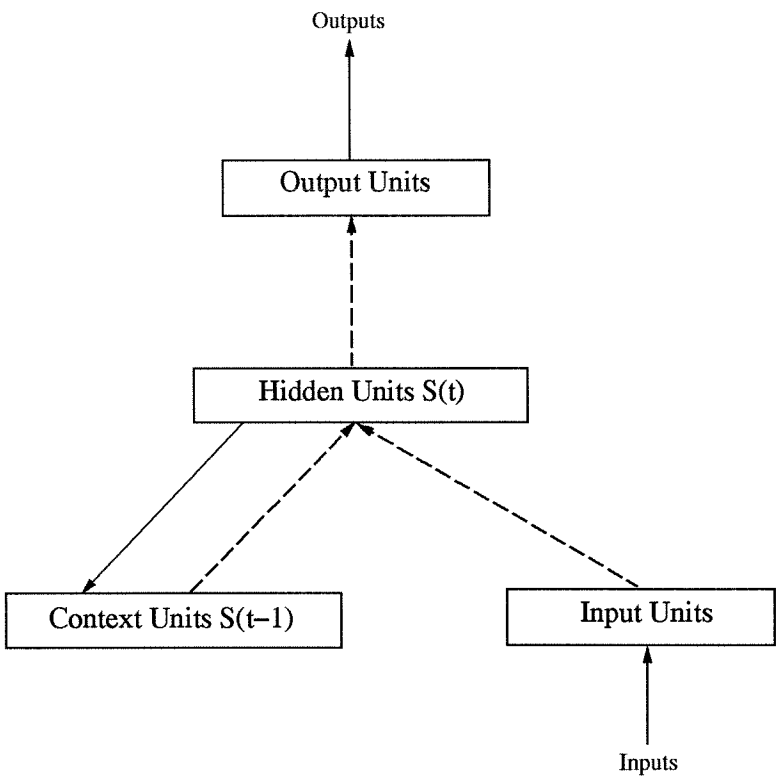


Figure 2.2: The Elman "simple recurrent network" structure.

2.2 First-Order Recurrent Networks

Jordan proposed a recurrent network architecture which uses the basic framework of Fig. 1.2, but restricts the connections by only allowing feedback from the output layer to the input layer [Jor86a, Jor86b]. The structure showed promising results in learning to associate a static pattern with a serially ordered output pattern. Based on that, Elman proposed the well-known "simple recurrent network" structure in [Elm90, Elm91], shown in Fig. 2.2. In the figure, dotted lines represent a set of fully connected trainable weights between the two layers connected. There are the same number of units in the "context" layer as in the "hidden" layer. The feedback connections represented by a solid line are one-for-one. If we use $S(t)$ to denote the hidden units at time t, then the context units are the hidden units delayed by one time step, indicated in the figure by $S(t-1)$. The input units are an unary coded representation of the alphabet of the language, so are the output units. During training, each example string is fed to the network through the input units one symbol at a time. At each time step, the output units are trained to produce

the correct prediction on the symbol that follows based on the $S(t)$ values. After each time step, the $S(t)$ units are copied back to the $S(t-1)$ units to preserve the "state" information. The Elman structure has been shown to successfully learn a simple regular grammar, the Reber grammar, details of which will be discussed in Chapter 4.

Both the Jordan and Elman structures are variations (or restricted forms) of the basic structure of Fig. 1.2. Note that all delay elements on the feedback links are omitted for the sake of brevity. Henceforth, any feedback link shown in the figures implies that there is a one time step delay associated with it. We will call this type of structure first-order recurrent networks, referring to the fact that the activation of any unit is a first-order (or linear) combination of activations of other units (which may or may not include itself), shaped by the activation function.

Experiments with the Elman structure in learning the Tomita grammars have shown great difficulties on the network's part[ZGS93]. For reasons that will be discussed in the next section, we turn to another type of structure, the second-order recurrent networks for our detailed study. It turned out that the average convergence time of the Elman network is several orders of magnitude longer than that of second-order recurrent networks.

2.3 Second-Order Recurrent Networks and Equivalent Representations

Giles et al. have proposed a "second-order" recurrent network structure to learn regular languages [GSC+90, GMC+92a]. A typical network is shown in Fig. 2.3, for the case of binary alphabet grammars. Henceforth, all references to second-order recurrent networks imply the network structure described in [GSC+90] and [GMC+92a].

Different from the structure shown in Fig. 1.2 is the "product layer," (the layer of units with "X"'s in the figure) where each node in the layer corresponds to the product of an input unit and a "state" unit. A product, of course, is a second-order term.

In operation, a string of symbols is fed through the input line to the network one time step at a time. At each time step, either of the input nodes "0" or "1" is on, the product layer and the hidden layer activation values (S^t's) are calculated. At the end of a time step, the S^t values are fed back to the S^{t-1} units to be used for the next time step.

Since the input symbols are unary-coded in the input layer, and only one input unit is on at any given time step, the second-order network can be represented as two separate

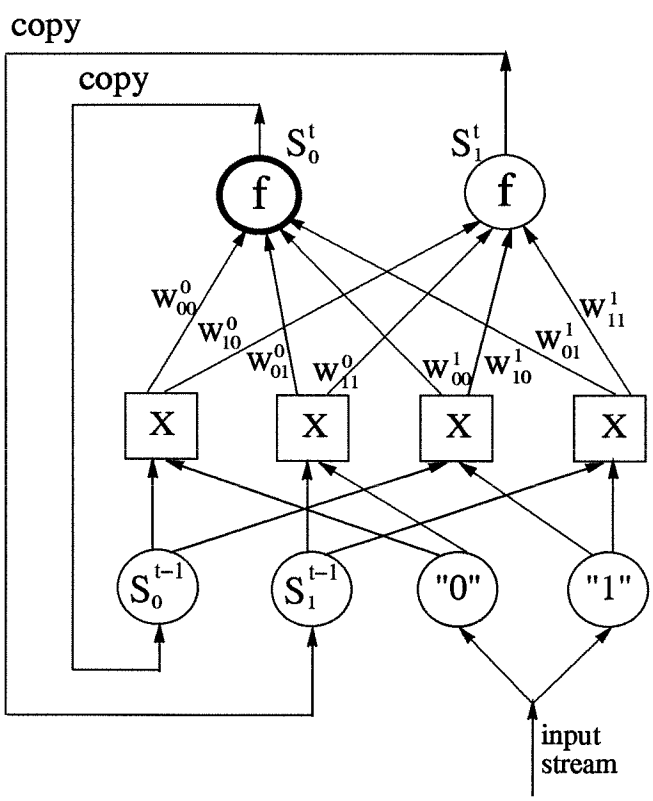


Figure 2.3: A second-order recurrent network structure.

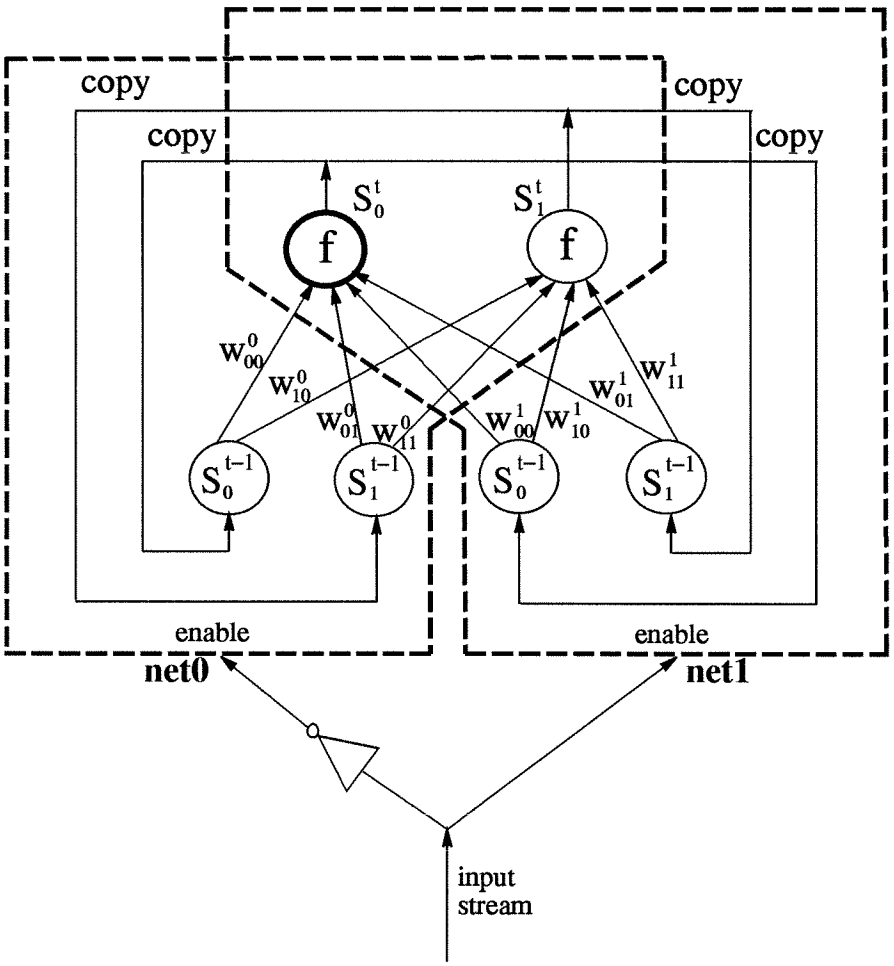


Figure 2.4: Equivalent first-order structure of second-order network

first-order networks controlled by a gating switch (Figure 2.4) as follows: the network consists of two first-order networks with shared hidden units. The common hidden unit values are copied back to both **net0** and **net1** after each time step, and the input stream acts like a switching control to enable or disable one of the two nets. For example, when the current input is 0, **net0** is enabled while **net1** is disabled. The hidden unit values are then decided by the hidden unit values from the previous time step weighted by the weights in **net0**.

The hidden unit activation function is the standard sigmoid function, $f(x) = \frac{1}{1+e^{-x}}$. Note that this representation of a second-order network, as two networks with a gating function, provides insight into the nature of second-order nets, i.e., clearly they have

greater representational power than a single simple recurrent network, given the same number of hidden units. In the structural sense, it is also closer to the representation of a finite state machine: different sets of connection weights are used for different input symbols, just as a finite state machine has different sets of transition rules for different input symbols.

This structure was used in our initial experiments.

2.4 The "Unstable State" Behavior of a Learned Second-Order Analog Network

2.4.1 Training Process

Since there is effectively only one layer in the network, we use the superscript to represent time instead of layer index as in Chapter 1. So S_i^t denotes the activation value of hidden unit number i at time step t. For weights, the superscript is used to represent the index of the subnetwork it is in. So w_{ij}^n is the weight from unit S_j^{t-1} to unit S_i^t in $\mathbf{net}n$. $n \in \{0, 1\}$ in the case of binary inputs.

Hidden node S_0^t is chosen to be a special indicator node, whose desired activation is close to 1 at the end of a legal string, and close to 0 otherwise. At time $t = 0$, initialize S_0^0 to be 1 and all other S_i^0's to be 0, i.e., assume that the null string is a legal string. The network weights are initialized randomly with a uniform distribution between -1 and 1.

A training set consists of randomly chosen variable length strings with length uniformly distributed between 1 and L_{max}, where L_{max} is the maximum training string length. Each string is marked as "legal" or "illegal" according to the underlying grammar. The learning procedure is a gradient descent method in weight space (similar to that proposed by Williams and Zipser [WZ89]) to minimize the error at the indicator node at the end of each training string [GMC+92a].

In a manner different from that described in [GMC+92a], we present the whole training set (which consists of 100 to 300 strings with L_{max} in the range of 10 to 20), all at once to the network for learning, instead of presenting a portion of it in the beginning and gradually augmenting it as training proceeds. Also, we did not add any *end* symbol to the alphabet as in [GMC+92a]. Details of the training process is described in Appendix A. We found that the network can successfully learn the machines (2–10 states) we tested on, with a small number of hidden units (4–5) and less than 500 epochs, agreeing with

the results described in [GMC+92a].

2.4.2 Analysis of Network Behavior During Training

To examine how the network forms its internal representation of states, we recorded the hidden unit activations at every time step of every training string in different training epochs. As a typical example, shown in Figure 2.5(a)-(e) are the $S_0 - S_3$ activation-space records of the learning process of a 4-hidden-unit network. The underlying grammar was Tomita #4, and the training set consisted of 100 random strings with $L_{max} = 15$. Note that here the dimension S_0 is chosen because of it being the important "indicator node," and S_3 is chosen arbitrarily. The observations that follow can be made from any of the 2-D plots from any run in learning any of the grammars in the experiments. Each point corresponds to the activation pattern of a certain time step in a certain string. Each plot contains the activation points of *all* time steps for *all* training strings in a certain training epoch as described in the caption. The following behavior can be observed:

1. As learning takes place, the activation points seem to be pulled in several different directions, and distinct clusters gradually appear (Figure 2.5(a)-(e)).

2. After learning is complete, i.e., when the error on each of the training strings is below a certain tolerance level, the activation points form distinct clusters, which consist of segments of curves (Figure 2.5(e)).

3. Note in particular that there exists a clear gap between the clusters in the S_0 (indicator) dimension, which means that the network is making unambiguous decisions for all the training strings and each of their prefix strings (Figure 2.5(e)).

4. When given a string, the activation point of the network jumps from cluster to cluster as input bits are read in one by one. Hence, the behavior of the network looks just like a state machine's behavior.

2.4.3 Analysis of Network's Internal Representation

It is clear that the network attempts to form clusters in activation space as its own representation of states and is successful in doing so. Motivated by these observations, we applied the k-means clustering algorithm[McQ67] to the activation record in activation

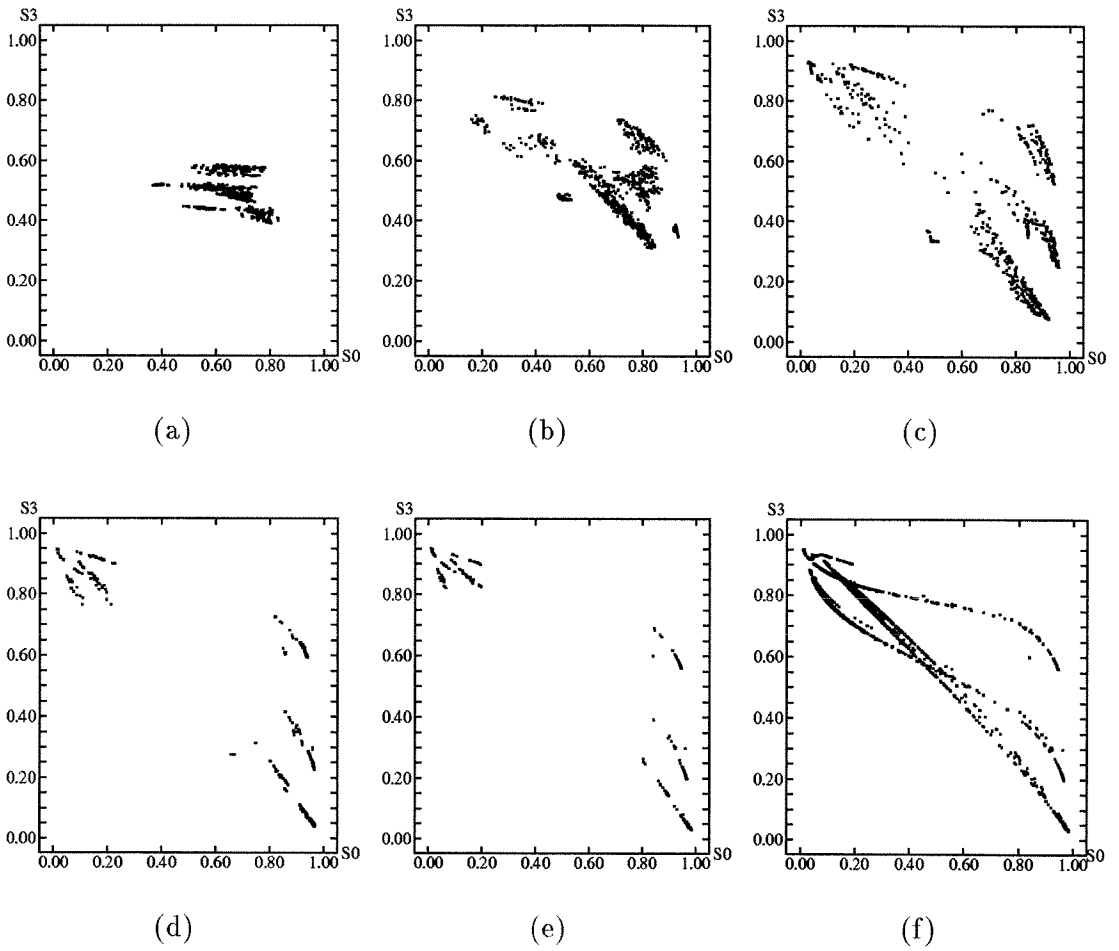


Figure 2.5: Hidden unit activation plot $S_0 - S_3$ in learning Tomita grammar #4. (S_0 is the x axis.) (a)-(e) are plots of all activations on the training data set. (a)During 1st epoch of training. (b)During 16th epoch of training. (c)During 21st epoch of training. (d)During 31st epoch of training. (e)After 52 epochs, training succeeds, weights are fixed. (f)After training, when tested on a set of maximum length 50.

space of the trained network to extract the states (instead of simply dividing up the space evenly as in [GMC+92a]). In choosing the parameter k, we found that if k was chosen too small, the extracted machine sometimes could not classify all the training strings correctly, while a large k always guaranteed perfect performance on training data. Hence, k was chosen to be a large number, for example, 20.

The initial seeds were chosen randomly. We then defined each cluster found by the k-means algorithm to be a "state" of the network and used the center of each cluster as a representative of the state. The transition rules for the resulted state machine are calculated by setting the S_i^{t-1} nodes equal to a cluster center, then applying an input bit (0 or 1 in the binary alphabet case), and calculating the value of the S_i^t nodes. The transition from the current state given the input bit is then to the state that has a center closest in Euclidean distance to the obtained S_i^t values. In all our experiments, the resulted machines were several states larger than the correct underlying minimal machines.

Moore's state machine reduction algorithm was then applied to the originally extracted machine to get an equivalent minimal machine which accepts the same language but with the fewest possible number of states. Similar to the results in [GMC+92a], we were able to extract machines that are equivalent to the minimal machines corresponding to the underlying grammars from which the data was generated.

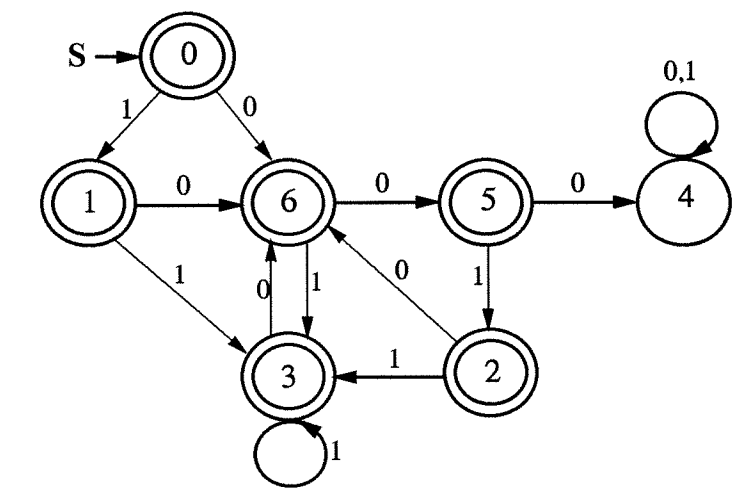
As an example, Fig. 2.6(a) shows the extracted state machine from the same trained network as used in the previous subsection, and (b) shows the equivalent minimal state machine, which is the same as in Fig. 2.1(d).

2.4.4 Analysis of Network Behavior During Testing

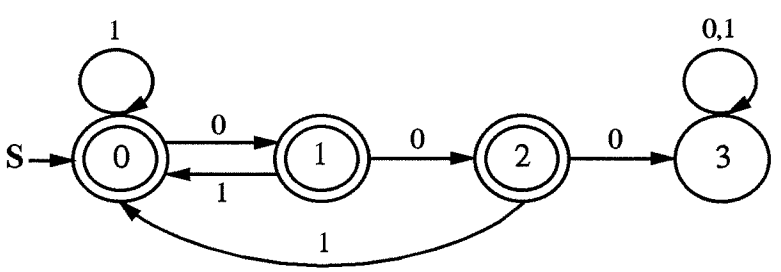
These trained networks perform well in classifying unseen short strings (not much longer than L_{max}). However, as longer and longer strings are presented to the network, the percentage of strings correctly classified drops substantially. Shown in Figure 2.5(f) is the recorded activation points for S_0-S_3 of the same trained network from Figure 2.5(e) when long strings are presented. The original net was trained on 100 strings with $L_{max} = 15$, whereas the maximum length of the test strings in Figure 2.5(e) was 50. Activation points at all time steps for all test strings are shown.

Several observations can be made from Figure 2.5(e):

1. The well-separated clusters formed during training begin to merge together for longer



(a)



(b)

Figure 2.6: (a) Extracted state machine by clustering. (b) Equivalent minimal machine.

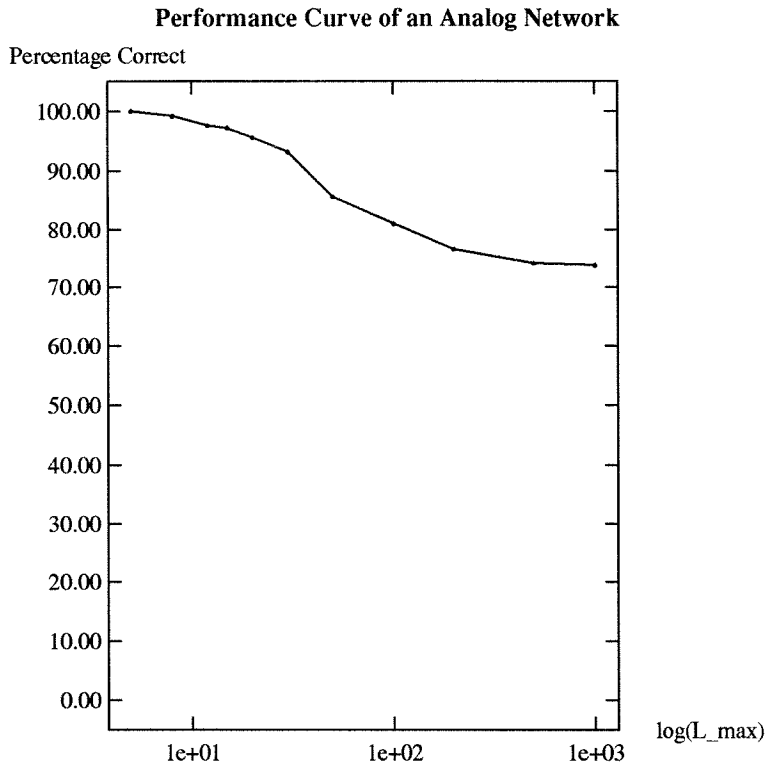


Figure 2.7: The performance curve of a learned second-order network on Tomita Grammar #4.

and longer strings and eventually become indistinguishable. These points in the center of Figure 2.5(e) correspond to activations at time steps longer than $L_{max} = 15$.

2. The gap in the S_0 dimension disappears, which means that the network could not make hard decisions on long strings.

3. The activation points of a string stay in the original clusters for short strings and start to diverge from them when strings become longer and longer. The diverging trajectories of the points form curves with sigmoidal shape.

Fig. 2.7 shows the performance of the same trained network from Figure 2.5(e) tested on randomly generated string sets with different maximum lengths. The x-axis is the maximum string length of the test set in log scale, the y-axis is the percentage of correct classifications by the trained network. Each point on the curve corresponds to a performance on a test set consisting of 500 strings. As can be seen, as the test strings become

longer, the network's performance drops significantly. Note that since the test sets are randomly generated, with uniform string length distribution from 1 to L_{max}, each set contains a certain number of "not too long" strings compared to the training set, for which the network has a better chance in producing the right answers. So had we tested it on pure "long" string sets, the performance would have been even worse.

Similar behavior was observed for 14 out of 15 of the networks successfully trained on different machines, excluding the vending machine model. (Similar behavior in other types of recurrent networks has been found in different contexts [SSCM91, Pol91].) Some of the networks started to misclassify as early as when the input strings were only 30% longer than L_{max}. Each of these 14 trained networks made classification errors on randomly generated test sets with maximum string length no longer than $5L_{max}$. The remaining one network was able to maintain a stable representation of states for very long strings (up to length 1000). Note that the vending machine was excluded because it's a trivial case for long strings, i.e., all the long strings are legal strings so there's no need to distinguish between them. This is not the case for the other machines.

2.5 A Network That Can Form Stable States

From the above experiments it is clear that even though the network is successful in forming clusters as its state representation during training, it often has difficulty in creating *stable* clusters, i.e., forming clusters such that the activation points for long strings converge to certain centers of each cluster, instead of diverging as observed in our experiments. The problem can be considered as inherent to the structure of the network where it uses analog values to represent states, while the states in the underlying state machine are actually discrete. One intuitive suggestion to fix the problem is to replace the analog sigmoid activation function in the hidden units with a threshold function:

$$D(x) = \begin{cases} 1.0 & \text{if } x \geq 0.5 \\ 0.0 & \text{if } x < 0.5. \end{cases} \tag{2.1}$$

In this manner, once the network is trained, its representation of states (i.e., activation pattern of hidden units) will be stable and the activation points won't diverge from these state representations once they are formed. However, there is no known method to train such a network, since one cannot take the gradient of such activation functions.

An alternative approach would be to train the original second-order network as described earlier, but to add the discretization function $D(x)$ on the copy back links during testing. The problem with this method is that one does not know *a priori* where the formed clusters from training will be. Hence, one does not have good discretization values to threshold the analog values in order for the discretized activations to be reset to a cluster center. Experimental results have confirmed this prediction. For example, after adding the discretization, the modified network cannot even correctly classify the training set which it has successfully learned in training. As in the previous example, after training and without the discretization, the network's classification rate on the training set was 100%, while with the discretization added, the rate became 85%. For test sets of longer strings, the rates with discretization were even worse.

Note that another alternative approach is to use a conventional analog network in training and to then apply various clustering techniques in the hidden unit activation space (after learning) to enforce stability[GMC+92a]. While this is a valid approach, here we are more interested in constructing a network that stabilizes itself (or equivalently, automatically performs the clustering) *during* the learning process. (Das et al.[DM94] have recently proposed a structure where adaptive clustering is performed during learning.)

We propose that the discretization be included in *both* training and testing in the following manner: Figure 2.8 shows the structure of the network with discretization added.

From the formulae below, one can clearly see that in operational mode, i.e., when *testing*, the network is equivalent to a network with discretization only:

$$h_i^t = f(\sum_j w_{ij}^{x^t} S_j^{t-1}), \qquad \forall\ i,t, \tag{2.2}$$

$$S_i^t = D(h_i^t), \qquad where\ \ D(x) = \begin{cases} 1-\varepsilon & \text{if } x \geq 0.5 \\ \varepsilon & \text{if } x < 0.5, \end{cases} \tag{2.3}$$

$$\begin{aligned} \Rightarrow S_i^t &= D(\ f(\sum_j w_{ij}^{x^t} S_j^{t-1})\ ) \\ &\equiv D_0(\sum_j w_{ij}^{x^t} S_j^{t-1}), \qquad where\ \ D_0(x) = \begin{cases} 1-\varepsilon & \text{if } x \geq 0.0 \\ \varepsilon & \text{if } x < 0.0. \end{cases} \end{aligned} \tag{2.4}$$

(Here ε is some constant, x^t is the input bit at time step t. We use h_i^t to denote the analog value of hidden unit i at time step t, and S_i^t the discretized value of hidden unit i

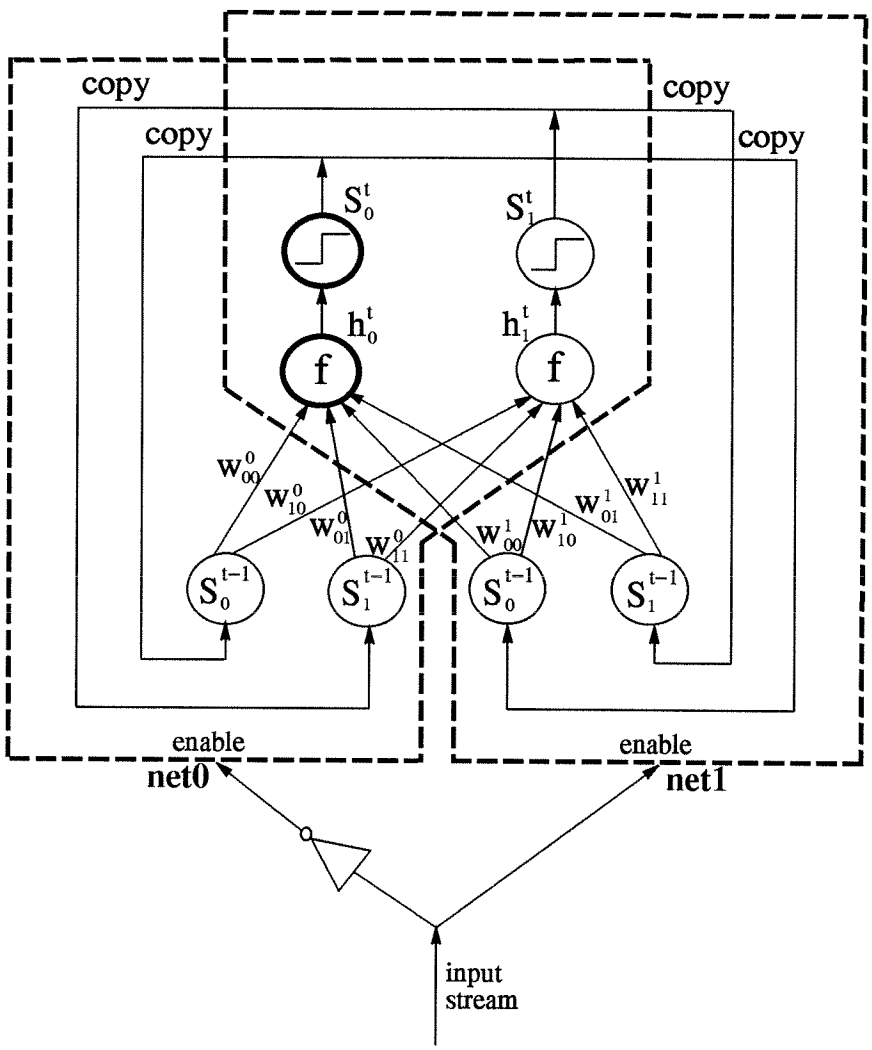


Figure 2.8: A combined network with discretizations

at time step t.)

Hence, the sigmoid nodes can be eliminated in testing to simplify computation.

During training, however, the gradient of the soft sigmoid function is made use of in a pseudo-gradient method for updating the weights. The next section explains the method in more detail.

It is found that the value of the constant ε can be set to any value between the range of 0 and 0.25 without much difference in the network's performance. A larger value of ε results in too small a difference between the two threshold values, which makes it difficult for the network to learn. We use the value 0.2 throughout the experiments in this thesis.

By adding these discretizations into the network, one might argue that the capacity of the net is greatly reduced, since each node can now take on only 2 distinct values, as opposed to infinitely many values (at least in theory) in the case of the undiscretized networks. However, in the case of learning discrete state machines, the argument depends on the definition of the capacity of the analog network. In our experiments, 14 out of 15 of the learned networks have unstable behavior for nontrivial long strings, so one can say that the capabilities of such networks to distinguish different states may start high, but deteriorate over time, and would eventually become zero. Section 2.9 discusses the capacity of discrete and recurrent networks with stable representations.

2.6 The Pseudo-Gradient Learning Method

During training, at the end of each string: $x^0, x^1, ..., x^L$, the mean squared error is calculated as follows (note that L is the string length, h_0^L is the analog indicator value at the end of the string):

$$E = \frac{1}{2}(h_0^L - T)^2, \tag{2.5}$$

where

$$T = target = \begin{cases} 1 & \text{if "legal"} \\ 0 & \text{if "illegal."} \end{cases} \tag{2.6}$$

Update w_{ij}^n, the weight from node j to node i in **net**n, at the end of each string presentation:

$$w_{ij}^n = w_{ij}^n - \alpha \widetilde{\frac{\partial E}{\partial w_{ij}^n}}, \qquad \forall\ n, i, j, \tag{2.7}$$

$$\widetilde{\frac{\partial E}{\partial w_{ij}^n}} = (h_0^L - T)\frac{\widetilde{\partial h_0}^L}{\partial w_{ij}^n}, \qquad \forall\, n,i,j, \tag{2.8}$$

where α is the learning rate, and $\frac{\widetilde{\partial}}{\partial w_{ij}^n}$ is what we call the "pseudo-gradient" with respect to w_{ij}^n. α is chosen to be 0.5 in our experiments. Too large an α may create oscillations in training, and thus lead to non-convergence, while a smaller α makes the learning and convergence speed slower.

To get the pseudo-gradient $\frac{\widetilde{\partial h_0}^L}{\partial w_{ij}^n}$, pseudo-gradients $\frac{\widetilde{\partial h_k}^t}{\partial w_{ij}^n}$ for all t,k need to be calculated forward in time at each time step:

$$\frac{\widetilde{\partial h_k}^t}{\partial w_{ij}^n} = f' \cdot \Big(\sum_l w_{kl}^{x^t}\frac{\widetilde{\partial h_l}^{t-1}}{\partial w_{ij}^n} + \delta_{ki}\delta_{nx^t}S_j^{t-1}\Big), \quad \forall\, i,j,n,k,t. \tag{2.9}$$

(Initially, set: $\frac{\widetilde{\partial h_k}^0}{\partial w_{ij}^n} = 0, \qquad \forall\, i,j,n,k.$)

As can be seen clearly, in carrying out the chain rule for the gradient we replace the real gradient $\frac{\partial S_l^{t-1}}{\partial w_{ij}^n}$, which is zero almost everywhere, by the pseudo-gradient $\frac{\widetilde{\partial h_l}^{t-1}}{\partial w_{ij}^n}$. A heuristic justification of the use of the pseudo-gradient is as follows: suppose we are standing on one side of the hard threshold function $S(x)$, at point $x_0 > 0$, and we wish to go downhill. The real gradient of $S(x)$ would not give us any information, since it is zero at x_0. If instead we look at the gradient of the function $f(x)$, which is positive at x_0 and increases as $x_0 \to 0$, it tells us that the downhill direction is to decrease x_0, which is also the case in $S(x)$. In addition, the magnitude of the gradient tells us how close we are to a step down in $S(x)$. Therefore, we can use that gradient as a heuristic hint as to which direction (and how close) a step down would be. This heuristic hint is what we used as the pseudo-gradient in our gradient update calculation.

Note that the training process described here is very similar to the training described in Appendix A for analog networks. Besides the difference in taking the gradients, the stopping criteria for the two types of networks are different. Unlike the analog network training, the discrete network training does not require a preset error threshold as the stopping criterion. Learning is stopped only when the discrete indicator unit S_0 makes *no error* on all training strings.

2.7 Experimental Results

grammar	*training set*		*# of hidden units*	*mean # of epochs*	*σ of epochs*	*mean # of total characters*
	# of strings	L_{max}				
Tomita #1	50	5	4	36.4	33.4	5205
Tomita #2	100	8	4	38.8	27.3	18120
Tomita #3	150	12	4	82.8	43.2	77040
Tomita #4	100	8	4	76.6	27.0	31712
Tomita #5	100	8	4	64.4	20.7	26662
Tomita #6	100	8	4	20.8	8.3	8611
Tomita #7	100	10	4	138.5	31.1	70774
Vending machine	365	6	5	231.8	22.4	383165
10-state machine	317	12	8	5798	—	14315262

Table 2.1: Experimental results from training the discrete recurrent network on regular grammars. L_{max} is the maximum length of training strings. The numbers for epochs and total characters processed during learning are the average numbers over 5 runs with different random weight initializations, except for the 10-state machine, for which only one run was obtained. σ is the standard deviation of the epochs over the 5 runs. All runs, except one in learning Tomita #3 and one in learning Tomita #7 which failed to converge, have perfect generalization performance, i.e., 100% correct on strings of any length.

Table 2.1 shows the experimental results obtained by training the discrete recurrent network by the pseudo-gradient learning method on various grammars. An epoch is one presentation of the whole training set to the network. The total number of characters processed is the cumulative count of all characters in all strings presented to the network in all training epochs. The results in Table 2.1 demonstrate that the discrete recurrent network model can be successfully trained to recognize simple grammars. Furthermore, because of its discrete nature, the network is inherently stable for strings of arbitrary length. The scaling of the network training time with respect to the complexity of the language being learned agrees with the theoretical results described in Section 1.1.

Shown in Figure 2.9(a),(b),(c) are the $h_0 - h_3$ activation-space records of the learning process of a discretized network (h values are the undiscretized values from the sigmoids). The underlying grammar is again the Tomita Grammar #4. The parameters of the network and the training set are the same as in the previous case. Again, any of the other 2-D plots from any run in learning any of the grammar in the experiments could have been used here.

Figure 2.9(c) is the final result after learning, where the weights are fixed. Notice that

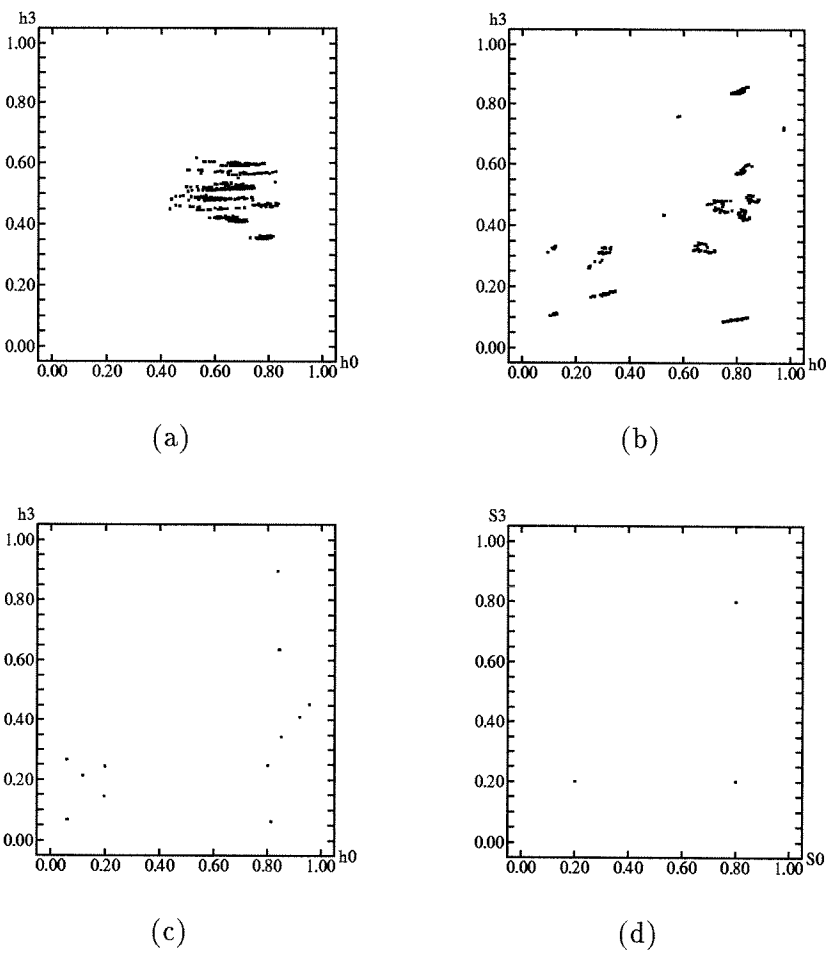
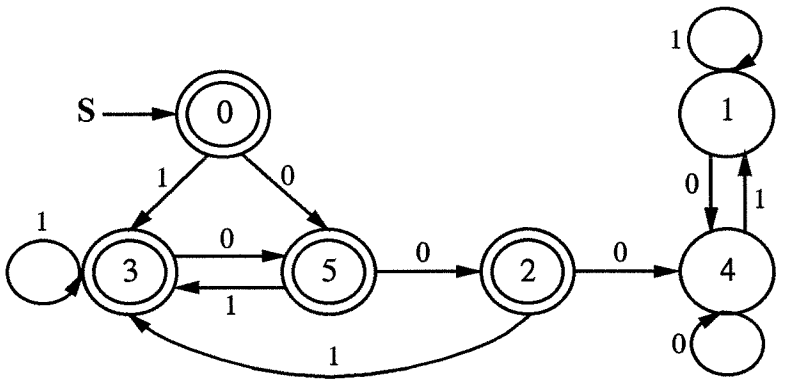
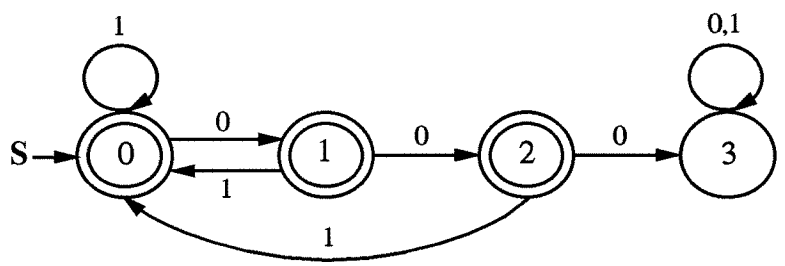


Figure 2.9: Discretized network learning Tomita grammar #4. (a)$h_0 - h_3$ during 1st epoch of training. (b)$h_0 - h_3$ during 15th epoch of training. (c)$h_0 - h_3$ after 27 epochs when training succeeds, weights are fixed. (d)$S_0 - S_3$, the discretized copy of $h_0 - h_3$ in (c).



(a)



(b)

Figure 2.10: Extracted state machine from the discretized network after learning Tomita grammar #4. (double circle means "accept" state, single circle means "reject" state.) (a)6-state machine extracted directly from the discrete activation space. (b)Equivalent minimal machine of (a).

there are only a finite number of points in the final plot in the analog activation h-space due to the discretization. Figure 2.9(d) shows the discretized value plot in $S_0 - S_3$, where only 3 points can be seen. Each point in the discretized activation S-space is automatically *defined* as a distinct state, no point is shared by any of the states. The transition rules are calculated as before, and an internal state machine in the network is thus constructed. In this manner, the network performs self-clustering. For this example, 6 points are found in S-space, so a 6-state-machine is constructed as shown in Figure 2.10(a). Not surprisingly this machine reduces by Moore's algorithm to a minimum machine with 4 states which is exactly the Tomita Grammar #4 (Figure 2.10(b)).

As an example of the ability of the network to learn grammars of medium complexity,

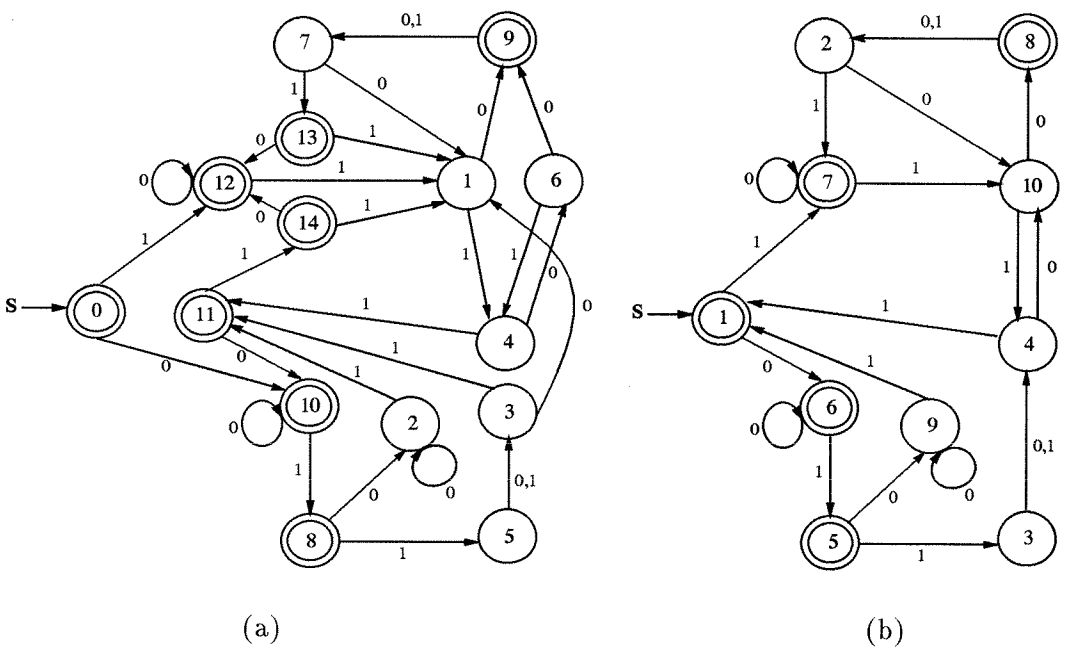


Figure 2.11: Extracted state machine from the discretized network after learning the 10-state machine: (a) 15-state machine extracted directly from the discrete activation space, (b) equivalent minimal 10-state machine of (a). Note that the state structure in (a) and (b) are quite similar, for example, states 1 and 6 in (a) are equivalent to 10 in (b), and states 12, 13, and 14 in (a) play a similar role to state 7 in (b).

Figure 2.11(a) shows the effective automaton learned by the network trained on strings from the 10-state machine. Application of Moore's algorithm to this 15-state network automaton results in a reduction to the correct 10-state machine shown in Figure 2.11(b).

Similar results were observed for all the other grammars in the experiments.

There are several advantages in introducing discretization into the network:

1. Once the network has successfully learned the state machine from the training set, it's internal states are stable. The network will always classify input strings correctly, independent of the length of these strings.

2. No clustering is needed to extract out the state machine, since instead of using vague clusters as its states, the network has formed distinct, isolated points as states. Each point in activation space is a *distinct* state. The network behaves *exactly* like a state machine.

3. Experimental results show that the size of the state machines extracted out in this approach, which need not be decided manually (no need to choose k for k-means) as in the previous undiscretized case, are of a much smaller size than found previously by the clustering method.

It should be noted that convergence has a different meaning in the case of training discrete networks as opposed to the case of training analog networks. In the analog networks' case, learning is considered to have converged when the error for each sample is below a certain *error tolerance level.* In the case of discrete networks, however, learning is only stopped and considered to have converged when *zero error* is obtained on all samples in the training set. In the experiments reported in this chapter the analog tolerance level was set to 0.2. The discretized networks took on average 30% longer to train in terms of learning epochs compared to the analog networks for this specific error tolerance level.

2.8 Empirical Investigation of the Pseudo-Gradient Learning

Theoretical analyses of learning in recurrent networks can be quite non-trivial. In particular, analytical investigation of our proposed pseudo-gradient method for recurrent

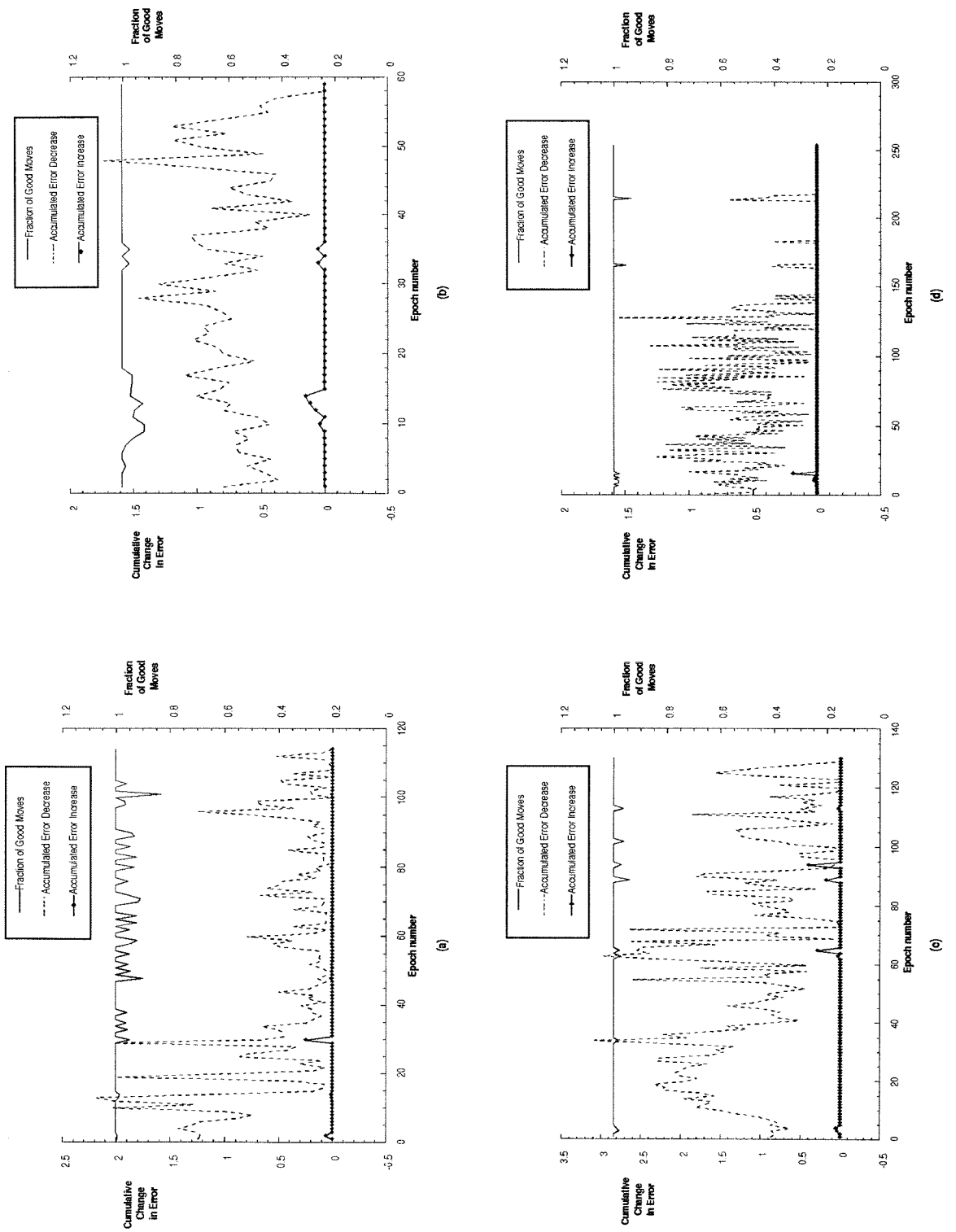


Figure 2.12: Statistical record of the pseudo-gradient learning of regular grammars. (a)Tomita #3. (b)Tomita #5. (c)Tomita #7. (d)The vending machine.

networks, appears intractable. Hence, we are limited to empirical evidence to support our claim that the method indeed appears to work well on non-trivial problems.

Figures 2.12(a) – (d) show the typical learning processes of 4 of the grammars described in Section 2.1 (plots of other grammars and of networks with different initial conditions have similar features and are not shown.) During each epoch of learning, the training strings are presented to the network one by one. In processing each string, pseudo-gradients are calculated and all weights are updated accordingly if the network makes an erroneous decision on that string. After each such weight update, we test the network with the new set of weights on the same string and thus a new error is calculated. If the new error is smaller than the old one, we can then conclude that the pseudo-gradient has successfully decreased the error on this specific string as it was intended to, otherwise, we count it as a failure, or "bad move." Thus, the total fraction of "successful moves" (out of the total number of weight updates) induced by the pseudo-gradient algorithm can be calculated for each epoch. In each of the plots, the solid curve corresponds to this fraction of "good moves" as a function of the epoch number.

To evaluate the severity of the effect of all the "bad moves," we also record the magnitude of each error increase or decrease on a string after each weight update, and sum the error increases and decreases for each epoch. The lower dotted curve in each of the plots corresponds to the summation of all the error increases (or the cumulative effect of all "bad moves") as a function of epoch number, while the remaining oscillating curve is the summation of all error decreases ("good moves") per epoch. The training set and the number of hidden units used for each grammar are the same as in Table 1.

It is clearly evident that the pseudo-gradient algorithm induces successful moves over 80% of the time. In addition, when bad moves occur, their cumulative effect per epoch is always smaller (and often much smaller) than the cumulative effect of the successful moves, except during one epoch while learning the Tomita #7 grammar. Hence, the empirical evidence clearly indicates that the overwhelming tendency of the pseudo-gradient algorithm is to reduce the error on a per-string and per-epoch basis.

Considering the bad moves individually, the magnitude of an error increase by a single bad move is on average much larger than an error decrease caused by a single successful move — however it is the *cumulative* effect that accounts for the convergence of the learning. Also note that the bad moves do not necessarily occur more frequently as the

grammars become more complicated.

In conclusion, our empirical investigations have shown that although following the pseudo-gradient descent direction does not guarantee error reduction, it is certainly an effective way to conduct the training of discrete recurrent networks.

2.9 On the Capacity of Recurrent Networks for Finite State Machine Representation

In this chapter, we have explored the ability of recurrent networks in learning to simulate finite state machines. It is then of great interest to look at the problem from a different perspective: what is the capacity of such networks in representing finite state machines? In other words, if we know the finite state machine before hand, and are asked to *construct* a network to represent this machine specifically, how large does the network need to be?

2.9.1 Discrete Networks

For the case of discrete networks, where threshold units are used, Alon et al. give us the following results for first-order networks in [ADO91]:

Let m the number of states in a finite state machine, let $K_1(m)$ be the number of threshold units needed for a first-order network to represent the m-state machine with binary input alphabet, then:

- If there are no restrictions on how complicated individual threshold units can be, then

$$O\left((mlog(m))^{\frac{1}{3}}\right) \leq K_1(m) \leq O\left(m^{\frac{3}{4}}\right). \tag{2.10}$$

- If the weight alphabet of the network is limited, or if any threshold unit is only allowed a limited number of fan out connections, then the lower bound changes to

$$K_1(m) \geq O\left((mlog(m))^{\frac{1}{2}}\right). \tag{2.11}$$

- If any threshold unit is only allowed a limited number of fan in connections, or if the weight and threshold alphabets and the fan out connections are all limited, then the lower bound changes to

$$K_1(m) \geq O(m). \tag{2.12}$$

- Every m-state machine can be built with a discrete network with $2m+1$ units, which has a fixed weight alphabet $\{-1, 0, 1\}$, a fixed threshold alphabet $\{1, 2\}$, and fan out connections of at most 3.

These results are derived for first-order networks, with direct connections from "current state" units to "next state" units. The arguments are based on a simple counting method with the number of changeable weights as the main factor of concern.

The results can be easily generalized to second-order networks as follows: A second-order network can be represented equivalently as two first-order networks with shared hidden units (see Section 2.3). Or one can view it as a first-order network with twice as many units, some of whose units and weights are tied together. Thus a second-order network is at most as powerful as a first-order network with twice as many units. Let $K_2(m)$ be the number of units needed for a second-order network to represent an m-state machine. From the above argument, we have: $K_2(m) \leq K_1(m/2)$. That is, the above lower bounds derived for $K_1(m)$ also hold for $K_2(m)$ in orders of magnitude. The same is true for machines with non-binary inputs.

2.9.2 Analog Networks

Unlike a threshold unit in a discrete network, a unit in an analog network can have infinitely many possible activation values. One may conclude that if there is a stable state representation in the analog network for a state machine, it should require less units than its discrete counterpart.

Contrary to the above speculation, the following results show that if the network is to have stable state representations, and if it uses "cluster centers" for each state (i.e., single, isolated points for state representation), then it is no more capable (in orders of magnitude) than a discrete network to represent finite state machines.

Theorem 2.1 *If an analog recurrent network uses single points in activation space to accurately represent states of a finite m-state machine, then*

$$K(m) \geq O(m) \tag{2.13}$$

for both first-order and second-order networks.

Proof:

For a finite m-state machine with b-ary alphabet, each state has exactly b transitions going from itself and leading to some other states (possibly itself). Thus there are in total bm different transitions among the m-states.

If we assume that the network uses a single point in activation space to represent a state in the finite state machine, then for a K unit network, each transition corresponds to a set of K equations involving an activation point with K coordinates being mapped to another activation point with K coordinates. Thus there are in total bmK independent such equations.

The "unknowns" in these equations are the network weights, and the coordinates of activation points for all states. Considering the thresholds in each unit as being an adjustable "weight" also, the number of weights are $K(K+1)$ for the case of first-order networks, and $bK(K+1)$ for the case of second-order networks. The number of "unknown" coordinates are mK. So the total number of "unknowns" in these sets of equations are $K(K+1)+mK$ and $bK(K+1)+mK$ for first-order networks and second-order networks, respectively.

The existence of a solution for the bmK equations requires:

$K(K+1)+mK \geq bmK$ for first-order networks, and

$bK(K+1)+mk \geq bmK$ for second-order networks.

So we have:

$K(m) \geq (b-1)m-1$ for first-order networks, and

$K(m) \geq \frac{(b-1)m}{b}-1$ for second-order networks.

If the network uses more than 1 point to represent some of the states, let the total number of activation points to represent all m-states be m', then the network has effectively a representation of a m'-state machine ($m' > m$), which minimizes to a m-state machine. Using similar arguments from above, we have:

$K(m) \geq (b-1)m'-1 > (b-1)m-1$ for first-order networks, and

$K(m) \geq \frac{(b-1)m'}{b}-1 > \frac{(b-1)m}{b}-1$ for second-order networks.

Thus $K(m) \geq O(m)$.

Q.E.D.

It remains an open question as to whether the network can use an *infinite* number of points in a cluster to represent a state, and what the capacity is for such cases.

Note the above is only a lower bound on the capacity of a network to represent a state machine, *assuming* that a *stable* representation exists. It does not provide us with any information on whether there indeed exist such solutions in the set of equations described above for all cases, nor does it tell us how easily such solutions can be found by gradient descent. The analysis of the network dynamics is a very difficult task, and appears to be intractable. We have seen from empirical experiments of ours and others' in training analog networks to learn regular grammars that such perfectly stable solutions are extremely difficult to find by gradient descent learning.

2.9.3 More Powerful Networks?

The results from the above two subsections show that under certain conditions, the number of units needed to build a m-state machine from both discrete and analog networks can be linear in m.

These results are based on network structures which map "current state" units directly into "next state" units. Thus the mapping is inherently limited from the well known result for feedforward networks described in Chapter 1. To achieve an arbitrary mapping, an intermediate layer of units is necessary. On the other hand, adding an intermediate layer of units itself results in a larger sized network. Whether the capacity of the network can be increased significantly by adding just a few intermediate units is still an unanswered question, i.e., we do not know if by adding intermediate units, the total number of units needed for a network to represent a m-state machine can be smaller than linear in m.

2.10 Summary

In this chapter we explored the formation of clusters in hidden unit activation space as an internal state representation for analog second-order recurrent networks which learn regular grammars.

These states formed by such a network during learning are not a stable representation, i.e., when long strings are seen by the network the states merge into each other and eventually become indistinguishable.

We suggested introducing hard-limiting threshold discretization into the network and

presented a pseudo-gradient learning method to train such a network. The method is heuristically plausible. The available empirical evidence indicates that the pseudo-gradient learning algorithm is effective in training such a network. Experimental results show that the network has similar capabilities in learning finite state machines as the original second-order network, but is stable regardless of string length since the internal representation of states in this network consists of isolated points in activation space.

The proposed pseudo-gradient learning method suggests a general approach for training networks with threshold activation functions.

We will see in the following chapters two extensions to the discrete recurrent network structure, as well as extensions to the pseudo-gradient learning for higher level language inference tasks.

Chapter 3
Inference of Context-Free Grammars

In this chapter, we extend our discrete recurrent network structure discussed in Chapter 2 to include an external stack for the task of learning context-free or type 2 grammars.

We have observed from the previous chapter the unstable state behavior of an analog second-order recurrent network in learning regular grammars. Similar behavior can be observed from analog networks learning context-free grammars. The new discrete recurrent network structure overcomes this difficulty by using isolated, discrete points in hidden unit activation space to achieve stability for infinitely long strings. We will see that the extended discrete structure for context-free language learning has similar advantages over its analog counterpart.

The chapter is organized as follows: Section 3.1 reviews context-free grammars and the corresponding pushdown automata. Section 3.2 discusses the analog recurrent network structure with "analog" external stacks, and the stability problem associated with it. Section 3.3 introduces discrete recurrent networks which use discrete external stacks. Section 3.4 describes a composite error function to deal with various situations in training networks with stacks. Section 3.5 extends the pseudo-gradient training algorithm necessary for such models. Section 3.6 presents experimental results in learning deterministic context-free grammars using the discrete stack model. Section 3.7 shows the results of empirical investigations on the effectiveness of the extended pseudo-gradient learning. Section 3.8 discusses effective ways to speed up training. Section 3.9 concludes the chapter.

3.1 Context-Free Grammars and Pushdown Automata

Context-free grammars represent a much wider class of languages than do regular grammars — finite state machines are not sufficient enough to represent all such grammars. The theory of finite automata and formal languages states that there exists a one-to-one correspondence between context-free languages and pushdown automata[HU79]. Thus a context-free language can be equivalently defined as the language accepted by its corresponding pushdown automaton: $P =< \Sigma, \Gamma, U, u_0, \delta, B, F >$, where

- Σ is the input alphabet.

- Γ is the stack alphabet.
- U is a finite nonempty set of states.
- u_0 is the start (or initial) state, $u_0 \in U$.
- δ is the state transition function; $\delta : U \times (\Sigma \cup \lambda) \times \Gamma \rightarrow$ the set of finite subsets of $U \times \Gamma^*$.
- B is the bottom of the stack symbol ($B \in \Gamma$).
- F is the set of final (or accepting) states ($F \subseteq U$).

The current configuration of pushdown automaton P as defined above is described by a triple $< u, x, \alpha >$, where

- u is the current state.
- x is the unconsumed portion of the input string.
- α is the current stack contents (with the topmost symbol written as the leftmost).

A context-free grammar L can be defined by either a pushdown automaton P that accepts via final state:

$L = L(P) = \{x \in \Sigma^* \mid \exists r \in F,\ \exists \alpha \in \Gamma^* \ni < u_0, x, B > \vdash^* < r, \lambda, \alpha >\}$,

or a pushdown automaton P' that accepts via empty stack:

$L = \Lambda(P') = \{x \in \Sigma^* \mid \exists r \in U \ni < u_0, x, B > \vdash^* < r, \lambda, \lambda >\}$,

where $\vdash^*$ means the configuration of the left-hand side can reach the configuration of the right-hand side by a sequence of successive moves specified by the state transition function.

A subclass of context-free languages, deterministic context-free languages are defined to be the languages recognized by deterministic pushdown automata. A deterministic pushdown automata is a pushdown automaton $P =< \Sigma, \Gamma, U, u_0, \delta, B, F >$, with the following restrictions on the state transition function δ:

- $(\forall a \in \Sigma)(\forall A \in \Gamma)(\forall u \in U)(\delta(u, a, A)$ is empty or contains just one element).
- $(\forall A \in \Gamma)(\forall u \in U)(\delta(u, \lambda, A)$ is empty or contains just one element).
- $(\forall A \in \Gamma)(\forall u \in U)(\delta(u, \lambda, A) \neq \emptyset \Rightarrow (\forall a \in \Sigma)(\delta(u, a, A) = \emptyset))$.

The above means that given a current automaton state, there cannot be more than one choice of next state, so for any string, there is never any more than one path through the machine. As in [DGS93], we further restrict the scope of deterministic context-free grammars with the following restrictions: the alphabet of the stack symbol is the same as the input alphabet, i.e., $\Sigma = \Gamma$; the language is accepted by a pushdown automaton P via *both* final state and empty stack, i.e., $L = \{x \in \Sigma^* \mid \exists r \in F \ni < u_0, x, B > \vdash^* < r, \lambda, \lambda >\}$; only the current input symbol can be pushed onto the stack; λ transitions (which can make state transitions or stack actions without reading in a new input symbol) are not allowed.

In short, we consider a subset of deterministic pushdown automata, or deterministic context-free grammars. The following simple lemma gives us a better idea of the scope of languages we consider (by using both the stack and final state) in terms of languages accepted via empty stack only.

Lemma 3.1 *The set of languages accepted by deterministic pushdown automata via both empty stack and final state is the same set of languages accepted by deterministic pushdown automata via empty stack only.*

Proof:

Let L_{SF} be the set of languages accepted by deterministic pushdown automata via both empty stack and final state, and L_S and L_F be the set of languages accepted by deterministic pushdown automata via empty stack and final state, respectively.

From definition,

$$L_{SF} = L_S \cap L_F \quad \Rightarrow \quad L_{SF} \subseteq L_S. \tag{3.1}$$

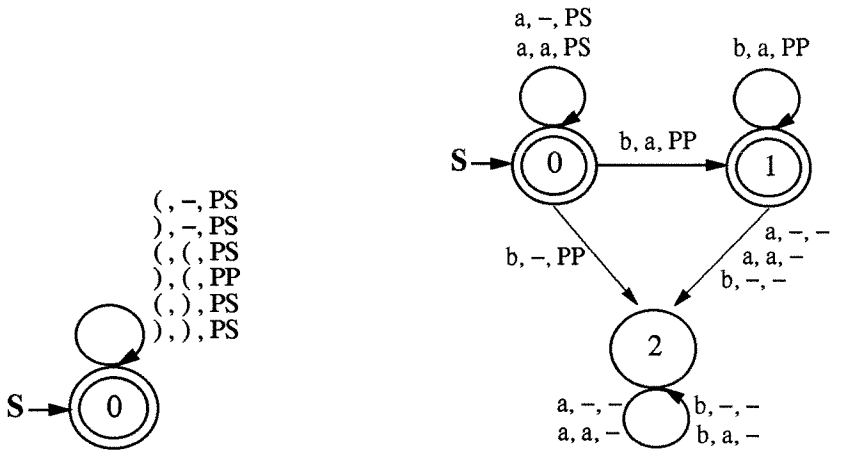
On the other hand, any language accepted by a pushdown automaton $P =< \Sigma, \Gamma, U, u_0, \delta, B, F >$ via empty stack is accepted by a pushdown automaton P' via *both* empty stack and final state, where P' is exactly the same as P except that $F = U$ (all states are final states). So we have:

$$L_S \subseteq L_{SF}. \tag{3.2}$$

From (3.1) and (3.2), we have : $L_S = L_{SF} = L_S \cap L_F$.

Q.E.D.

Although the above restrictions reduce the language class considered, nevertheless this class retains the essential properties of context-free languages and is therefore more complex than any regular language.



(a) The parenthesis matching grammar. (b) The $a^n b^n$ grammar.

Figure 3.1: Pushdown automata representations of context-free grammars.

We experimented with the same grammars as in [DGS93], i.e.,

- The parenthesis matching grammar, where the number of right parenthesis should match the number of left parenthesis at the end of a string, and the former should be smaller or equal to the latter at any point in a string.
- The postfix grammar, where an acceptable string of operators and operands has the reverse Polish form.
- $a^n b^n$.
- $a^{m+n} b^m c^n$.
- $a^n b^n c b^m a^m$.

Figures 3.1(a) and (b) show the corresponding pushdown automata for the parenthesis matching grammar and the $a^n b^n$ grammar, respectively.

As in finite state machine representations, the start state is indicated by a free arrow with an "S," single circled states are "reject" states. Double circled states are final states, but are only *possible* "accept" states, since the criterion for a legal string has two conditions: that the automaton ends up in a final state, *and* that the stack is empty, at the end of processing the string. A transition rule is labeled by "x,y,z," where x stands

for the current input symbol, y stands for the top-of-stack symbol ("-" means an empty stack), and z stands for the operation taken on the stack: "PS" means push, "PP" means pop, and "-" means no action.

3.2 Analog Second-Order Recurrent Networks With External Stacks

From the previous section we know that one needs to have an external stack to operate on besides the finite state machine in order to represent context-free grammars. By training a network to behave like a pushdown automaton we equivalently obtain a finite-state machine with an external stack that accepts the corresponding context-free grammar.

Based on the original analog second-order structure (Figure 2.3), Das et al. have proposed a second-order recurrent network structure which utilizes an external "continuous stack" [DGS93] to learn deterministic context-free grammars.

Similar to the original analog second-order networks, this type of structure has a stability problem in learning context-free grammars. As can be seen from the results shown in [DGS93], the trained networks do not always show 100% correct classifications on long test strings. In addition, the operations on the "continuous stack" are not directly interpretable: in the defined continuous stack, each symbol in the stack was pushed in with a certain "length" in the range of 0 and 1 associated with it. A pop action takes out from the stack a set of symbols with total length 1. In the paper, it is claimed that for most cases, at the end of training, the symbols that are pushed to the stack all have lengths close to 1, so the stack operations simulates closely a conventional discrete stack's operations. However, one can easily imagine what happens when a very "deep" stack is generated: since any symbol in the stack has a length smaller than 1 (no matter how close to 1 it can be), each time a pop action is taken, a total length of exactly 1 worth of symbols are taken out, which introduces a reduction in the length of the next symbol on stack. This "error" accumulates, and eventually, the "pop" action will not take out as "pure" a symbol as before, thus the stack operation becomes un-interpretable in the conventional sense.

3.3 Discrete Recurrent Networks with External Stacks

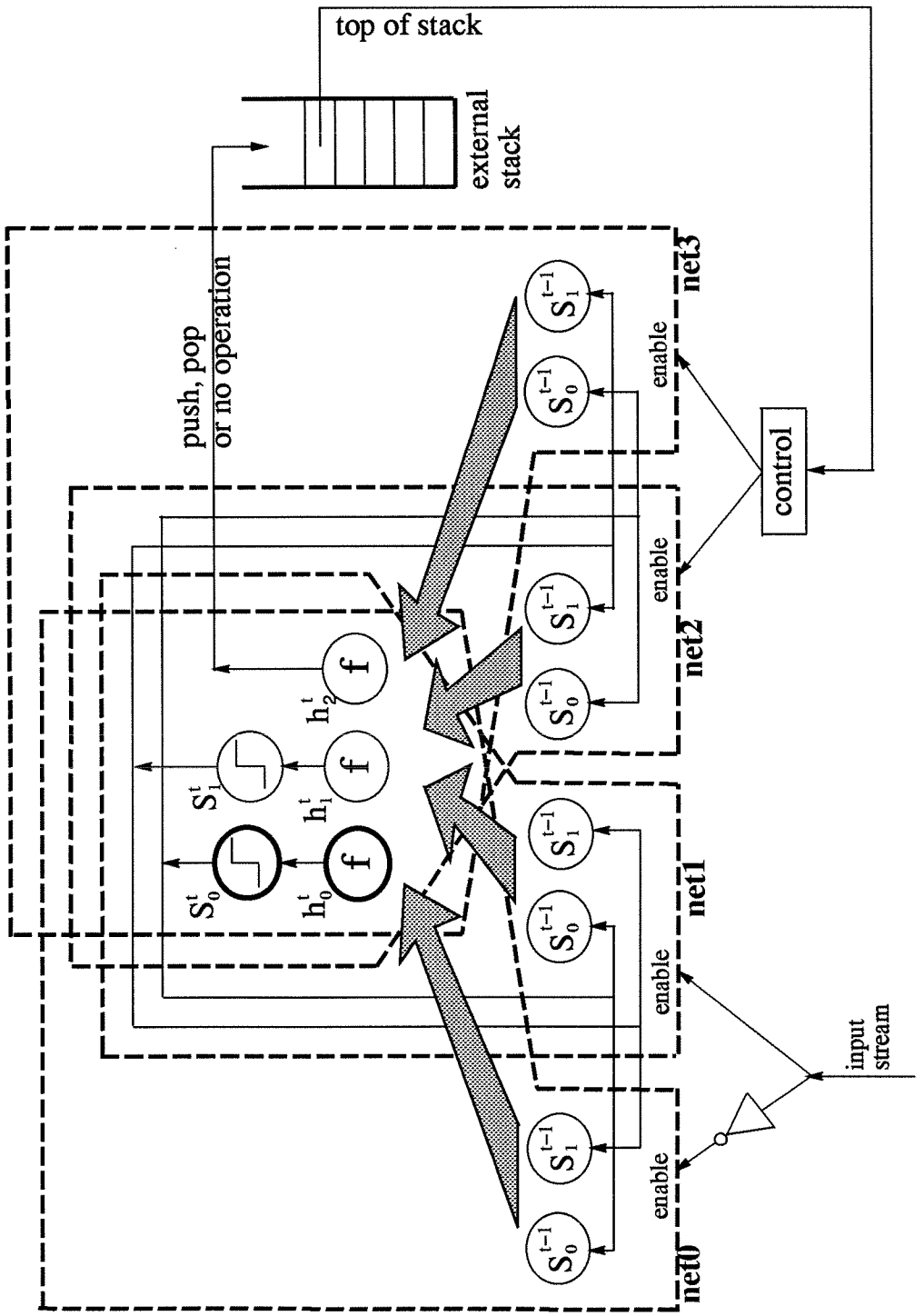


Figure 3.2: A discretized second-order network with an external stack. The thick circled unit h_0^t is the indicator unit: $h_0^t > 0.5$ for legal strings and $h_0^t < 0.5$ for illegal strings.

To overcome the problems associated with the analog recurrent network, we extend our discrete recurrent structure to include an external *discrete* stack. Shown in Fig. 3.2 is the structure of a discrete recurrent network with an external stack for the case of binary input and stack alphabets. The primary differences between this structure and the one proposed in [DGS93] are that we have a discrete stack as well as discretized units.

In Fig. 3.2 we have in effect four first-order networks with shared hidden units. In addition to the input symbol which acts as control to enable or disable **net0** or **net1**, the current top-of-stack symbol also acts as a second gating control which enables or disables **net2** or **net3**. Note that if the stack is empty, then both **net2** and **net3** are disabled, a situation that does not happen to the **net0**–**net1** pair.

As before, the unit h_0 is defined to be the "indicator" unit, whose desired activation is greater than 0.5 at the end of a legal string and smaller than 0.5 otherwise. The last unit, in this case h_2, is singled out to be the "action" unit, whose activation decides what stack operation to take. However, the value of this activation does not get copied back to the next time step. If h_2 is greater than a certain value (for the experiments reported here it is set to 0.6) then the current input symbol is pushed to the stack. If it is smaller than a certain value (0.4 in our case), then a symbol is popped out of the stack. Otherwise no action is taken.

The activation functions of the h units and the discretization function of the S units are the same as defined in the previous chapter.

3.4 A Composite Error Function for Context-Free Grammar Learning

The error functions for training networks with stacks to learn context-free grammars are more complicated than for the simple grammars discussed in the previous chapter. Several situations can be encountered during learning, each requiring the use of a different error function. We start by basing our error functions on those proposed in [DGS93], but there are some significant differences.

Let $h_0, h_1, ..., h_N$ be the hidden units of the network, where h_0 is the "indicator" unit and h_N is the "action" unit. Assume the current string being processed is $x^0, x^1, ..., x^L$, where L is the string length. Let d^t denote the depth of the stack at time step t, and a^t denote the top of stack symbol at time step t. The different error functions are as follows:

- For the case of a legal string:
  - If the end of the legal string is reached (without any attempt to pop an empty stack),
    $$E = \frac{1}{2}((1 - h_0^L)^2 + (d^L)^2). \tag{3.3}$$
    This means that for legal strings we want both the indicator unit to be on and the stack to be empty.
  - If the network attempts to pop an empty stack at time step $t \leq L$ when processing the legal string,
    $$E = \frac{1}{2}(1 - h_N^t)^2 - d^t. \tag{3.4}$$
    This means that for legal strings we want to correct the mistake of attempting to pop an empty stack by forcing the action unit value away from 0, i.e., avoid the "pop stack" action, and at the same time, encourage the stack to become nonempty.
- For the case of an illegal string:
  - If the end of the illegal string is reached (without any attempt to pop an empty stack),
    $$E = \begin{cases} h_0^L - d^L & \text{if the stack is empty} \\ 0 & \text{otherwise.} \end{cases} \tag{3.5}$$
    This means that for illegal strings we want either the stack to be nonempty, or the indicator unit to be off.
  - If the network attempts to pop an empty stack at time step $t \leq L$ when processing the illegal string,
    $$E = 0. \tag{3.6}$$
    We do nothing in this case because the attempt to pop an empty stack is itself considered an indication that the string is illegal.

Das et al. have suggested in [DGS93] that by providing the network with a "teacher" or an "oracle" to give hints, the learning can be sped up significantly. The teacher or oracle works as follows: there are certain illegal strings which are not prefixes to any legal strings, i.e., any symbols that follow such strings do not provide any further information.

For example, strings that have a prefix "(()))" for the parenthesis matching grammar are illegal no matter what comes after that prefix. Henceforth, we will call these strings "dead strings." The teacher is assumed to have the ability to identify such strings. Whenever a point is reached in the input string such that no further processing of the remaining string is necessary, the teacher produces a signal and the learning is halted. The network is then trained to have another special hidden unit, designated as the "dead unit," turn on. After the network has been trained in this way, a string is considered to be classified as illegal whenever the dead unit is turned on during testing. So the criterion for a legal string in testing now has three conditions: the network is in a final state (indicator node is on); the stack is empty; and the network's dead unit is off. The error functions have to be modified accordingly.

We found that it is not sufficient to add an error function only for the dead strings and to keep the other error functions (3.3)-(3.6) the same. For strings other than the dead strings, the network needs to be trained to have the dead unit turned off to avoid confusion. More specifically, letting h_1^t be the dead unit, we have:

- For the case of a legal string:
  - If the end of the legal string is reached (without any attempt to pop an empty stack, and the dead unit has not turned on before the end is reached),

$$E = \frac{1}{2}((1 - h_0^L)^2 + (d^L)^2 + (h_1^L)^2), \tag{3.7}$$

    i.e., we want the indicator unit to be on, the stack to be empty *and the dead unit to be off.*
  - If the network attempts to pop an empty stack at time step $t \leq L$, and the dead unit has not been turned on until now,

$$E = \frac{1}{2}(1 - h_N^t)^2 - d^t. \tag{3.8}$$

    Here we do not try to force the dead unit to turn on or off because it has been behaving as desired so far.
  - If the dead unit turns on at time step $t \leq L$, and there has not been any attempt to pop an empty stack so far,

$$E = \frac{1}{2}(h_1^t)^2, \tag{3.9}$$

i.e., we try to force the dead unit to turn off.

- For the case of an illegal string, but not a dead string:
  - If the end of the illegal string is reached (without any attempt to pop an empty stack, and the dead unit has not turned on before the end is reached),

$$E = \begin{cases} h_0^L - d^L + \frac{1}{2}(h_1^L)^2 & \text{the stack is empty} \\ \frac{1}{2}(h_1^L)^2 & \text{otherwise,} \end{cases} \tag{3.10}$$

    i.e., we want either the stack to be nonempty, or the indicator unit to be off, *and for both cases, the dead unit to be off.* The dead unit should not be on for such strings because they could be prefixes to certain legal strings.
  - If the network attempts to pop an empty stack at time step $t \leq L$, and the dead unit has not been turned on until now,

$$E = 0. \tag{3.11}$$

    We do nothing in this case for the same reason as the corresponding case without the teacher, and because the dead unit has been behaving as desired so far.
  - If the dead unit turns on at time step $t \leq L$, and there has not been any attempt to pop an empty stack so far,

$$E = \frac{1}{2}(h_1^t)^2. \tag{3.12}$$

    We do not want the dead unit to turn on since the string up to this point could still be a prefix to certain legal strings.

- If the string up to time step $t \leq L$ is a dead string,

$$E = \begin{cases} \frac{1}{2}((1 - h_1^t)^2 + (h_0^t)^2) & \text{if the stack is empty} \\ \frac{1}{2}(1 - h_1^t)^2 & \text{otherwise.} \end{cases} \tag{3.13}$$

  This means we want the dead unit to turn on *and either the indicator unit to turn off or the stack to be nonempty.*

3.5 The Extended Pseudo-Gradient Training

First of all, we need to define the operational equations for the network. Let a^t be the top of stack symbol at time t. At each time step t, the network calculates

$$h_i^t = f(\sum_j w_{ij}^{x^t} S_j^{t-1} + \sum_j w_{ij}^{a^t} S_j^{t-1}), \qquad \forall\, i, \tag{3.14}$$

$$where \quad f(x) = \frac{1}{1+e^{-x}}.$$

$$S_i^t = D(h_i^t), \tag{3.15}$$

$$where \quad D(x) = \begin{cases} 1-\varepsilon & \text{if } x > 0.5 \\ \varepsilon & \text{if } x < 0.5 \end{cases}$$

$$\Rightarrow S_i^t = D(\ f(\sum_j w_{ij}^{x^t} S_j^{t-1} + \sum_j w_{ij}^{a^t} S_j^{t-1})\ )$$

$$\equiv D_0(\sum_j w_{ij}^{x^t} S_j^{t-1} + \sum_j w_{ij}^{a^t} S_j^{t-1}), \tag{3.16}$$

$$where \quad D_0(x) = \begin{cases} 1-\varepsilon & \text{if } x > 0.0 \\ \varepsilon & \text{if } x < 0.0 \end{cases}$$

$$d^t = d^{t-1} + D_1(h_N^t), \tag{3.17}$$

$$where \quad D_1(x) = \begin{cases} 1 & \text{if } x > 0.6 \\ -1 & \text{if } x < 0.4 \\ 0 & \text{otherwise.} \end{cases} \tag{3.18}$$

Again, as described in Chapter 2, the value of the constant ε is chosen to be 0.2 in our experiments. Any value between 0 and 0.25 would result in similar performance.

The pseudo-gradients of the composite error function in weight space concern both $\frac{\widetilde{\partial h_k}^t}{\partial w_{ij}^n}$ for all t, k, n, i, j, and $\frac{\widetilde{\partial d}^t}{\partial w_{ij}^n}$ for all t, n, i, j. The former is calculated the same way as before. To calculate the latter, i.e., the pseudo-gradient of the depth of the stack, we use the iterative operational equation (3.17).

Initially, set $\frac{\widetilde{\partial d}^0}{\partial w_{ij}^n} = 0$ for all n, i, j. After each time step, update:

$$\frac{\widetilde{\partial d}^t}{\partial w_{ij}^n} = \frac{\widetilde{\partial d}^{t-1}}{\partial w_{ij}^n} + \frac{\widetilde{\partial h_N}^t}{\partial w_{ij}^n}, \qquad \forall\, n, i, j. \tag{3.19}$$

Here, in place of the gradient of the piece-wise step function D_1, we still use the pseudo-gradient of the action unit h_N. Although the value of the action unit does not get discretized and copied back after each time step, its pseudo-gradient can still be calculated by utilizing the pseudo-gradients of other hidden units:

For weights in the input-controlled subnetworks,

$$\frac{\widetilde{\partial h}_N^t}{\partial w_{ij}^n} = f' \cdot \left(\sum_{l=0}^{N-1} w_{Nl}^{x^t} \frac{\widetilde{\partial h}_l^{t-1}}{\partial w_{ij}^n} + \sum_l w_{Nl}^{a^t} \frac{\widetilde{\partial h}_l^{t-1}}{\partial w_{ij}^n} + \delta_{Ni}\delta_{nx^t} S_j^{t-1}\right), \quad \forall\, t, w_{ij}^n. \tag{3.20}$$

(Similar equations can be derived for weights in the top-of-stack-controlled subnetworks.)

In the above equation, we have left out a term concerning the top-of-stack symbol's dependency on the weights. Since a simple recurrent form of this term is analytically impossible to derive, an approximation was used in [SCG+90]. In our formula, the pseudo-gradient is itself an approximation and so further fine tuning by this term may not be necessary. Empirical results in the Section 3.7 will demonstrate that the networks can indeed perform successful learning without this term in the formula. Thus, the coupling between the stack and the network during learning is reflected only in the previous formula for the gradient of the stack depth.

3.6 Experimental Results

As in [DGS93], a training set consists of all strings up to a certain length, with repeated legal strings so that there are about half as many legal strings as illegal ones.

Table 3.1(a) and (b) show the detailed results for experiments with and without hints, respectively. The numbers in each row are averages over the successful runs (out of 10 possible successful runs) with different initial conditions — a successful run is taken to mean that the network generalizes perfectly for all string lengths. The number of overfitting runs indicates the number of times in the 10 runs that the network overfits the data by using too many internal states and did not generalize. The number of non-convergent runs is the number of times in the 10 runs that the training had not converged after 1000 epochs and was halted. Note that the number of unsuccessful runs are significantly fewer for the case with hints than without hints — hence, hints generally improve the reliability of the learning procedure. It is still an open question as to how to avoid overfitting in general by controlling the size of the derived automaton during learning. It should be noted however that overfitting did not occur for 4 out of the 5 grammars in the experiments when hints were provided.

The hidden unit sizes, L_{max}'s and training set sizes shown in Table 3.1(a) and (b) are the minimum sizes for which generalization could be obtained for each problem —

grammar	*training set*		*# of hidden units*	N_n	N_o	N_s	*mean # of epochs*	σ of *epochs*	*mean # of total characters*
	# of *strings*	L_{max}							
Parenthesis	46	6	3	0	0	10	28.8	16.3	5205
Postfix	63	7	4	1	0	9	62.3	17.1	21131
$a^n b^n$	32	6	4	2	0	8	127.3	4.9	16797
$a^{m+n} b^m c^n$	120	8	5	2	0	8	63	36.0	7560
$a^n b^n c b^m a^m$	150	7	7	3	3	4	328.8	249.1	243275

(a)

grammar	*training set*		*# of hidden units*	N_n	N_o	N_s	*mean # of epochs*	σ of *epochs*	*mean # of total characters*
	# of *strings*	L_{max}							
Parenthesis	180	6	3	0	0	10	12.0	10.5	11208
Postfix	371	7	4	4	2	4	185.8	149.0	408464
$a^n b^n$	760	8	5	4	4	2	150.5	87.5	793436

(b)

Table 3.1: Experimental results from training the discrete recurrent network on context-free grammars (a) with hints; (b) without hints. The training set and hidden unit columns indicate the fixed learning parameters for each grammar. 10 runs with different random initial weights were carried out for each grammar. N_s, the number of successful runs is the number of runs (of the 10 possible) for which the trained network generalized perfectly for strings of any length. The means for the epochs and total characters processed (and the standard deviation for the epochs) were estimated only from the successful runs. N_o, the number of overfitting runs is the number where the network overfitted the data and did not generalize perfectly. N_n, the number of non-convergent runs is the number of runs where the network did not converge on the training data after 1000 epochs.

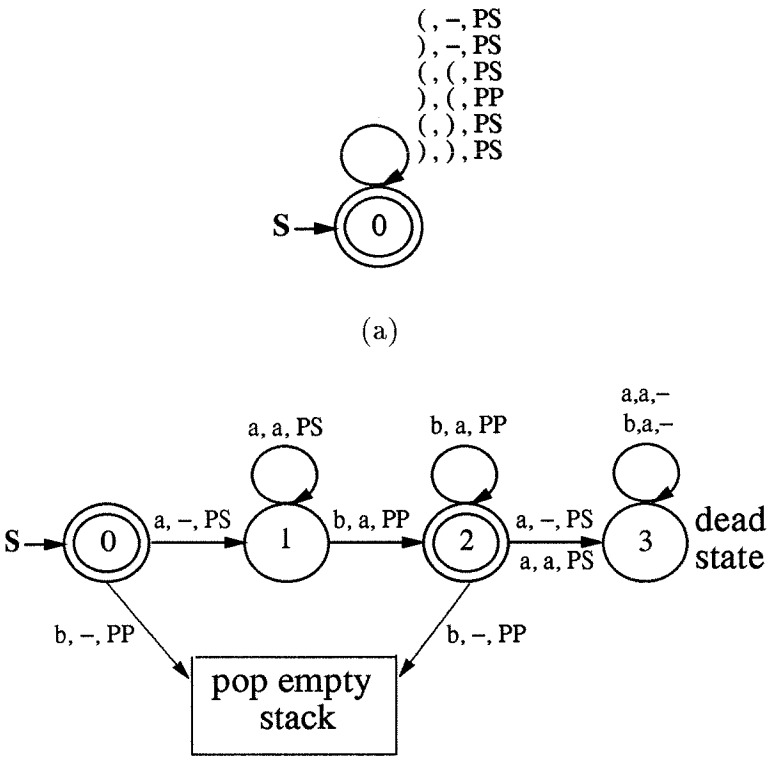


Figure 3.3: Extracted pushdown automata from the discretized network with an external stack after learning (a) the parenthesis grammar without hints; (b) the grammar $a^n b^n$ with hints. Double circled means the state has an indicator unit on, $S_0 = 0.8$: thus a processed string is legal if the automaton arrives at such a state *and* if the stack is empty. A dead state means the state has its dead unit on, $S_1 = 0.8$: a processed string is illegal as soon as the automaton arrives at such a state.

experiments using either less training data or fewer hidden units invariably resulted in less than perfect generalization. Larger data sets or larger networks than what is shown in the table can produce similar performance.

As an example, Fig. 3.3(a) and (b) show the derived pushdown automata from the networks after being trained on the parenthesis matching grammar and the $a^n b^n$ grammar respectively. As before, each state corresponds to one single point in the network's hidden unit activation space and the transition rules are derived similarly: set the S_i^{t-1} units to each of the points(states) in the activation space, give the network different combinations of input and top-of-stack controls, and thus calculate the next state given such input and

stack conditions.

Note that for the parenthesis matching grammar, the network finds the same pushdown automaton as shown in Figure 3.1(a), which has one single state. Starting from an empty stack, when the input is a "(," it pushes this input onto the stack. When the input is a ")," it either pops a "(" from the stack if the top-of-stack is a "(," or pushes the ")" onto the stack otherwise. Thus, whenever there are more ")"'s than "("'s, the machine executes a "push stack" operation no matter what the input symbol is, making the stack nonempty (indicating an illegal string) from this point on.

For the $a^n b^n$ grammar, the network finds a four-state pushdown automaton. Upon close observation, the equivalence between Figure 3.3(b) and Figure 3.1(b) is easily confirmed.

3.7 Empirical Investigation of the Extended Pseudo-Gradient Learning

In a similar manner to the previous chapter, we investigated how well the extended pseudo-gradient learning performed in learning pushdown automata. Plots of the fraction of successful moves by the pseudo-gradient algorithm, and the accumulated error increases and decreases as a function of epoch number are shown in Fig.3.4.

Similar to Figure 2.12, in each plot, the solid curve corresponds to the fraction of successful moves for each learning epoch. The bottom dotted curve corresponds to the summation of error reduction on a string by all successful moves in each epoch, and the remaining curve is the summation of error increases on strings by all bad moves. The training set and the number of hidden units used for each grammar are the same as in Table 3.1.

It can be observed from the plots that the pseudo-gradient algorithm makes bad moves in learning context-free grammars more often than it did in learning regular grammars. However, the percentage of successful moves are still mostly over 80%, and the accumulated error increases (due to bad moves) for any epoch are much smaller than the accumulated error decreases, except during one epoch in learning $a^n b^n$ without hints. Thus, as we found with the regular grammars, the empirical evidence suggests that the pseudo-gradient algorithm is quite effective in training discrete recurrent networks with external stacks.

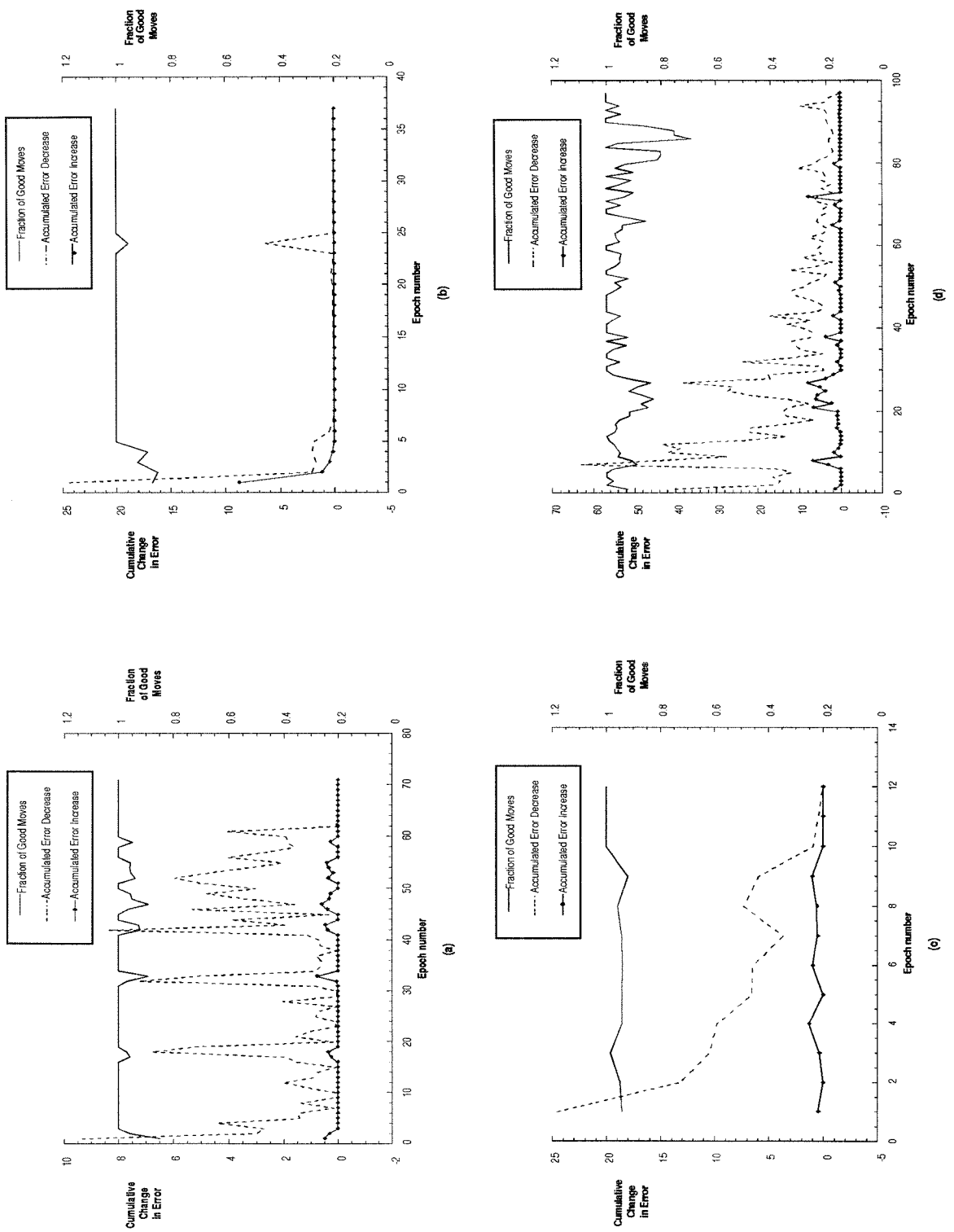


Figure 3.4: Statistical record of the pseudo-gradient learning of pushdown automata. (a)postfix, with hint. (b)$a^{m+n}b^mc^n$, with hint. (c)parenthesis matching, without hint. (d)a^nb^n, without hint.

3.8 Discussion

Using a discrete network as well as a discrete stack results in the advantages of a stable network, and a clear understanding of the operation of the stack. In [DGS93], where a continuous stack was used, the results show that the trained networks do not always generalize perfectly.

From the results in Table 3.1, it can be seen that providing the network with hints can indeed speed up learning, or even enable the learning of the grammars in cases where the grammar could not be learned without hints . However, unlike [DGS93], we did not find incremental presentation of the training data helped in improving the learning. Incremental presentation means that the network is initially given a small data set consisting of only short strings. After it has learned the current data set, more strings longer in length are added to the training set until all training strings are learned. We found in our experiments that once the network finds a configuration to fit the small data set with short strings, it is sometimes very hard to drag it away from that configuration to a desired configuration that will fit the later (longer) strings as well. The training times with and without incremental presentation of strings are comparable in our experiments. The numbers listed in Table 3.1(a) and (b) are of runs with the training data set presented to the network all at once.

We postulate that the reason why incremental learning worked for analog networks but not for discrete networks is due to the nature of analog and discrete networks. The analog network always finds a "soft" solution to a data set, which only has clear decisions for short strings, but is vague on long strings. Thus it is easy for it to "harden" such a solution when more restrictions about longer strings are enforced. The result is a solution whose "hardness" or decisiveness depends on the maximum length of the training strings. On the other hand, the discrete network always finds a "hard" solution to a data set which has clear decisions for strings of any length. Once it settles in such a solution it is hard to enforce restrictions about longer strings which contradict the current solution. So one may as well provide all the restrictions to the network at once. As long as there exists sufficient information in the data set, the resulting solution does not depend on the maximum length of training strings.

3.9 Summary

The primary advantages of introducing discretization into both recurrent networks and the external stack are as follows:

1. The network uses distinct, isolated points in hidden unit activation space as states, which guarantees stability for infinitely long strings once the network has successfully learned the grammar from the training set.

2. The network's operation on the discrete external stack is directly interpretable, no error is introduced for each operation.

3. The discretized network is easier to implement in hardware particularly when an external stack is involved.

In conclusion, we have presented in this chapter the basic ideas and algorithms for implementing stable discrete recurrent networks for learning deterministic context-free grammars. Specifically, we extended our previous discrete network models to include an external discrete stack with discrete symbols, defined an appropriate composite error function for learning, and derived the extended pseudo-gradient learning rule for this error function. The available empirical evidence indicates that the extended pseudo-gradient learning algorithm is effective in training such a network. The overall experimental results show that the proposed network has similar capabilities for learning context-free grammars as the analog second-order networks, while avoiding any problems with instability on long strings.

Chapter 4
Inference of Probabilistic Grammars

In this chapter, we make another extension to our discrete recurrent network structure for the purpose of learning probabilistic regular grammars[Zen94].

Section 4.1 gives a formal description of the type of grammars we will consider. Section 4.2 defines the problem of the inference of probabilistic grammars. Section 4.3 describes the grammars we studied. Section 4.4 reviews published results of learning regular grammars using the Elman network structure discussed in Chapter 1 and the fully recurrent network structure. A comparison of our goal and method with a newly proposed algorithm for deriving hidden Markov models is also made. Section 4.5 presents our initial attempts in experimenting with a simple network structure for learning probabilistic regular grammars, and analysis of results. Section 4.6 describes our final network structure and the corresponding pseudo-gradient learning. Section 4.7 discusses a two-stage verification of the correctness of the network's derived grammar structure during learning and provides theoretical analysis of these verification steps. Section 4.8 describes experimental results of using the proposed network structure and learning algorithm (along with verifications) in learning grammars without identical sub-parts. Section 4.9 describes an adaptive network augmentation process in an attempt to learn grammars with identical sub-parts. Finally, Section 4.10 summarizes the chapter.

4.1 Probabilistic Regular Grammars

A probabilistic grammar is a grammar with probabilities associated with its production rules. In the case of regular grammars, a probabilistic regular grammar can be represented by a finite state automaton with probabilistic transition rules. We consider the class of probabilistic regular grammars defined as: $M =< \Sigma, U, u_0, \delta, P, F >$, where

- Σ is the input alphabet.
- U is a finite nonempty set of states.
- u_0 is the start (or initial) state, $u_0 \in U$.
- δ is the state transition function; $\delta : U \times \Sigma \rightarrow U$.

- P is the state transition probability, a $|U|$ by $|\Sigma|$ matrix. $P_{ij} = P(x_j|u_i)$ is the probability of state $u_i \in U$ emitting symbol $x_j \in \Sigma$ and going to state $\delta(u_i, x_j)$. $\sum_j P_{ij} = 1$, $\forall i$.

- F is the set of final (or accepting) states, $F \subseteq U$.

A more general definition of probabilistic grammars allows a initial probabilistic *distribution* on U, and probabilistic state transitions associated with all symbols emitted from a state [Arb69]. We have restricted the grammar by setting the initial distribution to be with probability 1 on u_0, and 0 on all other states, and having only deterministic transitions for each symbol emitted from a state. However, the symbols emitted from a state are still probabilistic, thus state transitions, as determined by symbols emitted, are also probabilistic.

By definition, a general probabilistic grammar is equivalent to the well known first-order HMM (Hidden Markov Model)[Rab89]. Thus the restricted class of grammars defined above can be considered as a special case of HMMs, where the emission probabilities of a state are tied to transition probabilities: given that a certain symbol is emitted from a state, the next state is then deterministic instead of probabilistic. The Baum-Welch algorithm for deriving HMM models[Rab89] can be modified to derive this class of probabilistic grammars. However, the Baum-Welch algorithm requires knowledge of the number of states, which is not a piece of obvious information available to the learning algorithm. Stolcke and Omohundro recently proposed a model merging algorithm to learn both the number of states and the topology of HMM from examples[SO93]. We will make a brief comparison of our method and theirs in Subsection 2.4.2.

4.2 Problem Description

In a manner different from the inference problems in the previous two chapters, we consider another aspect of grammatical inference for the case of probabilistic grammars: learning to make predictions by "positive" example strings only.

We define this problem as follows: given a finite set of "positive" example strings generated probabilistically according to a probabilistic grammar, derive the underlying grammar rules and probabilities associated with them, so that when given any prefix string, a probabilistic prediction can be made on the following symbol(s), or successor(s).

We put quotation marks around the word "positive," since for probabilistic grammars, any string is associated with a probability, which can be considered as a measure of "positiveness." It is then not a good practice to put binary labels of "positive" or "negative" on probabilistic strings, as we have done for non-probabilistic grammars. Depending on this positiveness measure, strings can have different chances of being present in the example set. The only strings that are absolutely not present in the set are those with zero probabilities. Those can certainly be considered as truly "negative" examples.

At this point, one may recall a well known result we stated in Chapter 1: it is impossible to derive grammar rules for regular grammars from positive examples only. So the next question is, are we dealing with an impossible problem here? The answer is no, and the reason still concerns with probabilities. In our definition of the problem, we have assumed that the example set is generated *probabilistically* according to the probabilistic grammar. Thus the example set should truly reflect not only the grammar rules, but also the probabilities associated with them. Thus the frequency of a string's or a certain prefix's appearance contains crucial information about the grammar. The negative examples are in a way represented among those *not present* in the set. The shorter the string that is absent, the more probable that it is a zero probability string. Of course, we need to assume that the example set is large enough, so that the distribution of strings and prefixes provides sufficient information to all possible transition rules of the grammar. (We will discuss in great detail what is considered sufficient information in Subsection 4.7.2.) Given that, the result on non-learnability of grammars from positive only strings does not apply here.

One should note however, different as they are, the problems considered here and the previous chapters have one eventual goal in common: derive the underlying grammar rules. Once the rules are correctly inferred, any task concerning the language, be it classification or prediction, can be accomplished without further effort. Thus all problems considered above belong to the grammatical inference category.

4.3 Grammars Studied

A specific grammar of concern in this chapter is the Reber grammar, shown in Figure 4.1. This was originally studied as an example of nondeterministic regular grammars with characteristics that have been the subject of investigations into human language learning

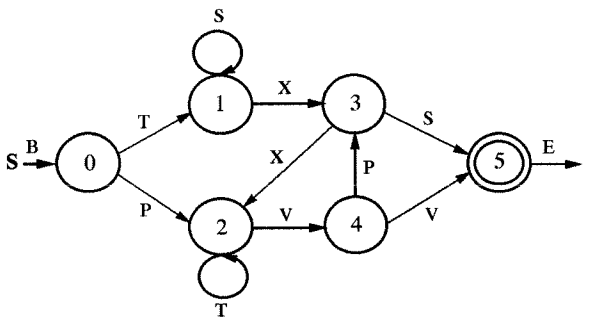


Figure 4.1: The Reber grammar

capabilities. The difficulty of this grammar lies in the fact that different instances of the same symbol lead to different states and hence to different following symbols. The problem can only be solved by maintaining information about characters seen previously[Reb69]. We specifically assigned probabilities to its transitions to make it a probabilistic grammar. In the figure, transitions with probability 0 are not shown. The probability values of the non-zero transitions will be specified when we present experiments on the grammar with different probability assignments later on. The grammar has an alphabet of size 5: T, S, X, P, V. In our experiments, we augment the alphabet to include a start symbol, B, and an end symbol, E to indicate string boundaries: the network is trained to predict the end of a string along with all other symbols.

A much more complicated grammar incorporating the Reber grammar is shown in Figure 4.2[CSSM89]. The numbers in brackets indicate the transition probabilities. This grammar is called the symmetric grammar since two identical copies of the Reber grammar are embedded in its upper and lower arms. We will call it the symmetric embedded Reber grammar here.

We devised another symmetric grammar with a simpler structure, shown in Figure 4.3. It will be called the simple symmetric grammar here.

It has been found empirically that some of the symmetric grammars are especially hard for neural networks to learn, where they must correlate the very last symbol with the very first one in order to discover long distance contingencies[CSSM89, SZ89]. These symmetric grammars all contain identical substructures and the transition probabilities in the substructures are the same. We call these grammars, "grammars with identical sub-parts."

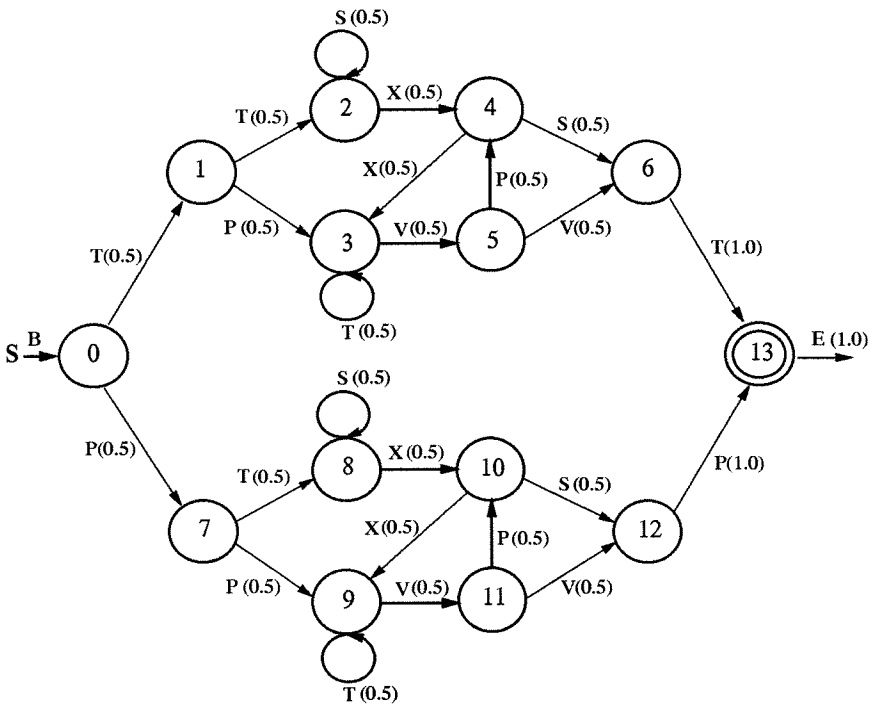


Figure 4.2: The symmetric embedded Reber grammar

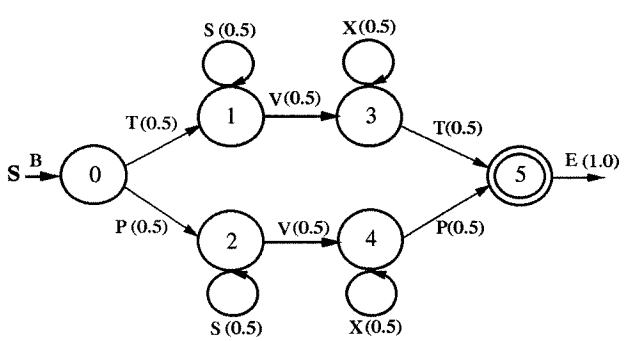


Figure 4.3: The simple symmetric grammar

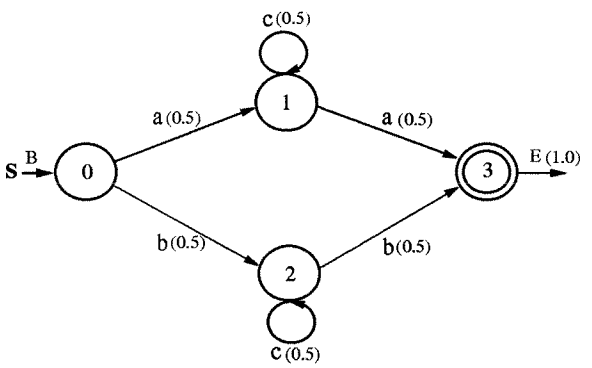


Figure 4.4: The $ac^*a \cup bc^*b$ grammar

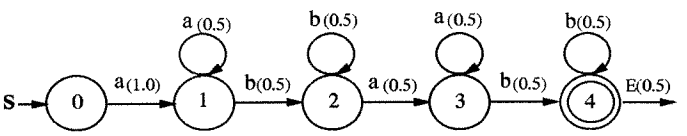


Figure 4.5: The $a^+b^+a^+b^+$ grammar

Stolcke and Omohundro studied two grammars modeled by HMM's[SO93], one of which is $ac^*a \cup bc^*b$. The equivalent probabilistic finite automaton with equiprobable transition splits is shown in Figure 4.4. This is also a symmetric grammar. However, we will see in Section 4.9 that this is not a particularly hard problem for the network, since the dependency between the first and the last letters are not always "long distance," in other words, two shortest example strings aa and bb directly reflect such dependencies. This makes the task of inference much easier for the network than the above two other symmetric grammars.

Another grammar studied in [SO93] is $a^+b^+a + b^+$, for which the probabilistic finite automaton is shown in Figure 4.5. Different from the symmetric grammar structure, this grammar is another example of grammars with identical sub-parts: states 1 and 3 are identical in their emission probabilities. We will see that it is the general class of grammars with identical sub-parts that the network has difficulty in learning.

4.4 Previous Work

4.4.1 Work on Recurrent Networks

Cleeremans et al.[CSSM89] have carried out experiments and analysis of learning the Reber grammar and the embedded Reber grammar by using the simple recurrent network or the

Elman network structure[Elm90]. The network is shown to be able to learn to predict the correct possible successors for the Reber language. But for the symmetric embedded Reber grammar, the network failed to learn even after 1.2 million string presentations. Smith et al.[SZ89] investigated the ability of the fully recurrent network structure in learning the Reber grammar and the symmetric embedded Reber grammar. A fully recurrent network is a network without layer divisions: every unit has connections with every other unit and itself, all units receive the input signals, and only output units receive the teacher signal. Their results show that the fully recurrent network structure is able to learn to make correct predictions for both the Reber grammar and the symmetric embedded Reber grammar.

In the above work, the transition probabilities of the grammars are not of concern during learning. Even though the training sequences are generated according to a defined transition probability matrix, the learning process does not take advantage of such probability information contained in the training data. The goal is to learn the underlying regular grammar, instead of the *probabilistic* regular grammar. The criterion for correct prediction is whether the output unit corresponding to the true successor has an activation value over a certain threshold, which happens to be 0.3 in both of the above cases. Learning is stopped when correct predictions are made consecutively for a certain number of randomly generated strings. A language generator is assumed to exist in both the above works to provide both training and testing sequences on demand.

Here we look at the problem from a different perspective: we will concern ourselves with not only the structure of the grammar, but also the production/transition probabilities. As part of the criterion for successful learning, the probability information contained in the training data is specifically made use of. So the learning takes advantage of not only the information of who is(are) the possible "successor(s)," but also the information of what is the probability of a certain symbol being the "successor." Instead of a language generator, we assume that only a finite set of example strings is available for training.

4.4.2 Work on HMMs

As mentioned in Section 4.1, the grammars we consider form a special class of HMM. Stolcke and Omohundro developed a model merging algorithm to learn both the number of states and the structure of an HMM from examples. The algorithm starts with the most

specific model which accepts the example strings only and no other strings: each string corresponds to a single path in the model with no loops. An exhaustive search is done recursively to merge states according to a probabilistic penalty measure. The algorithm demonstrates good performance on the case studies conducted in [SO93] where small data sets are used. However the exhaustive search can become time consuming when dealing with large data sets. Our theoretical analysis of data sufficiency in Section 4.7 shows that to obtain reliable estimation of transition probabilities, one needs a large enough data set to provide sufficient information. It is unreasonable to assume that the unknown model can provide us with a "minimal" data set that reflects accurately the structure and the probabilistic features of itself. Our approach differs from theirs in the following:

- Our approach requires a large data set, and specifically utilizes the probabilistic distribution information from the data set to evaluate the correctness of the structure of the model derived, and to make transition probability estimation. See Section 4.7 for detail.

- As can be observed from experiments, our network model starts with very few states and tries to grow more states to fit the data as learning proceeds. The model merging algorithm on the other hand, starts with many states and tries to merge more states to generalize as learning proceeds.

Section 4.8 gives experimental results of our algorithm learning one of the examples studied in [SO93], and makes a brief comparison. It would be interesting to conduct more detailed comparisons between the two approaches.

4.5 Initial Attempts

A very simple network structure was used in our initial experiments, shown in Figure 4.6. The structure is similar to a fully recurrent structure for prediction tasks when the inputs are in fact outputs from the previous time step. Since the underlying structure of the grammars we consider are still discrete, we again add discretization before any feedback to enforce stable state representations. For the sake of simplicity, the network is drawn with the discretized state units only, analog state units are eliminated, as in the testing mode. Each connection between two layers represents a set of adjustable weights, except

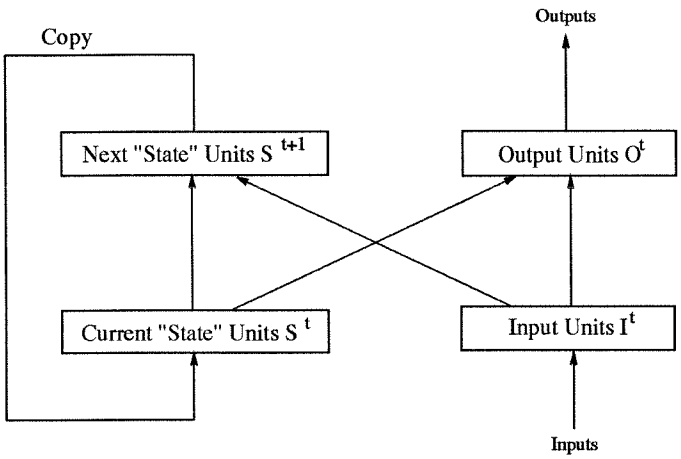


Figure 4.6: A simple discrete network structure.

the copy back link, which consists of one-to-one connections only, with non-adjustable weights fixed at 1. Analog outputs are used for the purpose of producing probabilities. Normalization to enforce the outputs to sum to 1 is only done after training, but not during training. All non-discretized units have sigmoidal activation functions.

We chose this first-order structure instead of a second-order one for the following reasons: the grammars of concern (Figures 4.1 to 4.3) have a relatively large sized alphabet compared to a binary alphabet, and a second-order structure would create a network of 7 subnetworks. (See Section 2.3 for detail.) With 3 being the minimum number of hidden units to represent the 6 states of the Reber grammar, a 7-fold network with 3 hidden units for each subnetwork is obviously somewhat redundant for such a simple grammar. Moreover, our main concern here is how a recurrent network performs in dealing with a different aspect of grammatical inference, as long as the network is large enough. Since the focus is not on the differences between first-order and second-order networks, and the first-order network is shown to be sufficient in learning the structure of the Reber and the symmetric embedded Reber grammar, we opted for the simpler first-order structure.

However, one should note that the methods described in this chapter can be applied to a second-order network structure or any other higher order structures.

The training set contains variable length strings generated probabilistically according to the grammar. The network is trained on every symbol of the training strings to predict the symbol that follows. Experiments on learning the Reber grammar with equiprobable transition splits showed that the network can indeed learn quickly to predict the correct

possible next symbols by producing high values on the units representing those symbols, and low values on all other units. Refer to Appendix B for details.

However, two problems exist. One is that the network has difficulty in producing probability estimates, no matter how long it is trained. For example, for a training set of 200 strings, the network succeeds in producing high outputs for the correct symbols and low outputs for all others within 200 learning epochs. On the other hand, for most of the predictions, the desired outputs have values close to 0.5 on two of the units, and close to 0.0 on all others, but the network's outputs stubbornly stay at values close to 0.3 and 0.7 on the two high outputs respectively, even after going through the training set 1000 times, which is far from satisfactory. The reason for this is explained in accordance with the second problem described below in Appendix B. The conclusion from the analysis in Appendix B is that the network is not powerful enough (not enough layers or hidden units) to produce exact probabilities.

The other problem is that the outputs depend not only on the current state, but also on the current input symbol, whereas the true probabilistic finite state grammar rules do not have either dependency, instead, the prediction on the next symbol should depend on the *next state* only. Due to the configuration of the network, it automatically tries to tie the input to the output closely. The result is its having drastically different outputs for the same "current state" layer values, depending on how it has reached that "state," i.e., the input. So a true "network state" is decided by *both* the "current state" layer and the input. The conclusion is that this network model tries to learn a different type of grammar (where dependency exists for state transition and previous path) to approximate our probabilistic finite state grammar, thus the architecture is ill-suited to the problem.

To overcome these two difficulties, an improved structure is presented in the next section.

4.6 Discrete Network Structure and Pseudo-Gradient Learning

The network structure we used in our main experiments on probabilistic grammars is shown in Figure 4.7. The network consists of two parts — the recurrent part which is discrete and is intended for the representation of the grammar states, and the feedforward part which is analog and is intended for the representation of the transition probabilities.

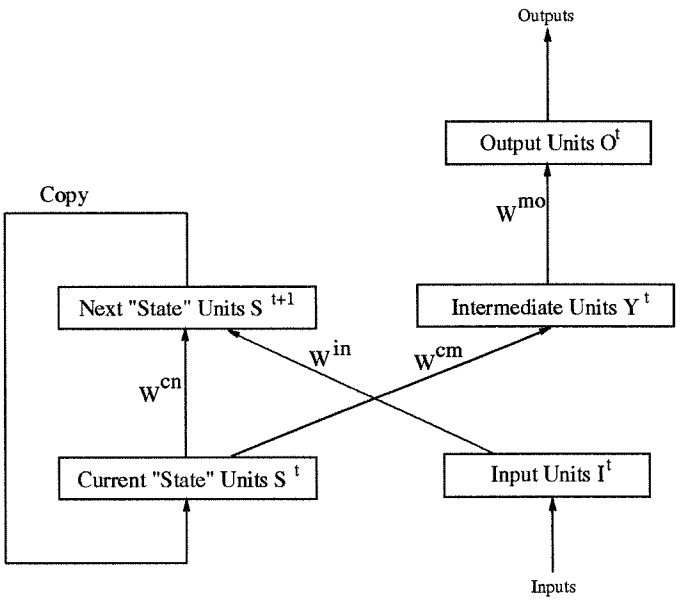


Figure 4.7: The final network structure used for probabilistic grammar learning.

To overcome the first difficulty mentioned in the last section, the output layer is connected through an intermediate layer to the "previous state" layer. This intermediate layer is added for the purpose of making the network powerful enough to produce accurate prediction probabilities from the previous state information. Recall from Chapter 1, we mentioned a well known result for feedforward neural networks, which states that at least 3 layers of units are needed for arbitrary function approximations, thus the 3 layers in our structure are required for arbitrary probability estimation.

To overcome the second difficulty, the coupling between the output and the input is removed — the output does not have any direct connections from the input (or current symbol), nor does the intermediate layer. At any time step, the output's only source of information about previous inputs (symbols) is the previous state layer.

During the training phase, we train the output to "predict" what the *current* symbol is based on the *current* state information. Clearly this is equivalent to putting the output layer and the intermediate layer on top of the "next state" layer, and training it to predict what the *next* symbol is based on the *next* state information. (Training for either of the two configurations would yield the same results.) After the network is successfully trained, during the testing phase, prediction of the next symbol can be carried out either by simply copying current state values back to the previous state values one time step ahead of time,

or by moving the intermediate and the output layer on top of the current state layer with all connection weights unchanged. The latter configuration turns out to be similar to an Elman structure, except that we have an extra intermediate layer and discrete units are used for the current and previous state layers.

Both input and output layers are unary coded representations of the alphabet, each containing 7 units for all grammars except the $ac^*a \cup bc^*b$ and the $a^+b^+a^+b^+$ grammar, for which there are 5 and 4 units, respectively.

As shown in Figure 4.7, let S_i^t be the state unit i at time t, I_i^t be the input unit i at time t, Y_i^t be the intermediate unit i at time t, and O_i^t be the output unit i at time t. Let w_{ij}^{cn}, w_{ij}^{in}, w_{ij}^{cm}, w_{ij}^{mo} represent connection weights between current state and next state layers, input and next state layers, current state and the intermediate layers, and the intermediate and output layers, respectively, with i being the unit number in the "to" layer and j being the unit number in the "from" layer. The operational equations of the network are:

$$S_i^{t+1} = D_0(\sum_j w_{ij}^{cn} S_j^t + \sum_j w_{ij}^{in} I_j^t), \qquad \forall\ i,t, \tag{4.1}$$

$$Y_i^t = f(\sum_j w_{ij}^{cm} S_j^t), \qquad \forall\ i,t, \tag{4.2}$$

$$O_i^t = f(\sum_j w_{ij}^{mo} Y_j^t), \qquad \forall\ i,t, \tag{4.3}$$

where f and D_0 are as defined in Chapter 2.

At each time step during the processing of a training string, the error between the output units and the target is calculated. Since our aim is to have the output reproduce/predict the input presented at the input layer, the target should be the same as the input, so the error at time step t is:

$$E^t = \frac{1}{2}\sum_i (O_i^t - I_i^t)^2, \qquad \forall\ t. \tag{4.4}$$

Note, the "true" targets which we aim for are the emission/transition probabilities of a given state, which are unknown. The probabilistic interpretation of the output units suggests the use of cross entropy as an error measure [EJM90]. S. Solla et al. have shown in [SLF88] that cross entropy error functions have steeper error surfaces than MSE

and thus can speed up the gradient descent learning process. This was confirmed in our experiments. The cross entropy error function is defined to be:

$$E^t = \sum_i \left[T_i^t \cdot log\left(\frac{T_i^t}{O_i^t}\right) + (1 - T_i^t) \cdot log\left(\frac{1 - T_i^t}{1 - O_i^t}\right) \right], \tag{4.5}$$

where $T_i^t = I_i^t$ is the target of output unit i at time t.

The experiments described henceforth in this chapter all use cross entropy as error measures.

For the weights w_{ij}^{cm}'s and w_{ij}^{mo}'s, i.e., weights in the analog feedforward part of the network, the error gradients with respect to these weights are calculated the same way as in the standard back-propagation algorithm [RHW86]. For the remaining weights, which are in the discrete recurrent part of the network, pseudo-gradient is used for calculating the error gradients:

$$\frac{\widetilde{\partial E}^t}{\partial w_{ij}^n} = \sum_k \frac{\partial E^t}{\partial O_k^t} \cdot \frac{\widetilde{\partial O}_k^t}{\partial w_{ij}^n}, \qquad n = cn, in, \ \forall\ i, j, t, \tag{4.6}$$

where

$$\frac{\partial E^t}{\partial O_k^t} = -\frac{I_k^t}{O_k^t} + \frac{1 - I_k^t}{1 - O_k^t} = \begin{cases} -\frac{1}{O_k^t} & \text{if } I_k^t = 1 \\ \frac{1}{1 - O_k^t} & \text{if } I_k^t = 0, \end{cases} \qquad \forall\ k, t, \tag{4.7}$$

$$\frac{\widetilde{\partial O}_k^t}{\partial w_{ij}^n} = f' \cdot \left(\sum_l w_{kl}^{mo} \cdot \frac{\widetilde{\partial Y}_l^t}{\partial w_{ij}^n}\right), \qquad n = cn, in, \ \forall\ i, j, t, \tag{4.8}$$

$$\frac{\widetilde{\partial Y}_k^t}{\partial w_{ij}^n} = f' \cdot \left(\sum_l w_{kl}^{cm} \cdot \frac{\widetilde{\partial S}_l^t}{\partial w_{ij}^n}\right), \qquad n = cn, in, \ \forall\ i, j, t, \tag{4.9}$$

$$\frac{\widetilde{\partial S}_k^{t+1}}{\partial w_{ij}^n} = f' \cdot \left(\sum_l w_{kl}^{cn} \cdot \frac{\widetilde{\partial S}_l^t}{\partial w_{ij}^n} + \delta_{ki}\delta_{n,cn} S_j^t + \delta_{ki}\delta_{n,in} I_j^t\right), \qquad n = cn, in, \ \forall\ k, i, j, t. \tag{4.10}$$

As shown by Miller[Mil93] and others[HP90], both the mean squared error measure and the cross entropy error measure minimize to the expected value of the target, provided that the network is sufficiently powerful. Thus in our binary target case, they both minimize to the probability of the target taking on value 1. Therefore we can be assured that if our network with sufficient number of units reaches a global minimum on the error surface, its output units should give the desired approximate probability of the symbol predicted.

4.7 Verification During Training

Since our aim is to have the output produce probabilities, we cannot use a preset threshold as a stopping criterion for learning, as in [CSSM89] and [SZ89]. A different criterion is needed.

In our initial experiments, we let the network run for a fixed number of epochs, and manually extract states from the network to check if it has reached a correct configuration. The disadvantage of doing this is that networks with different initial conditions need different training times to reach a good solution. Different grammars also require different training times. No universal preset training time exists for every situation.

A more adaptive criterion is needed to verify the goodness of the state configuration found by the network and to stop the learning of states in time to prevent overfitting. We employ a two stage verification process which makes use of the distribution information of the training data set in evaluating the goodness of a network's state configuration.

4.7.1 First Stage Verification: a Necessary Condition

The first stage is called a first-order verification. After each learning epoch, a finite state machine is extracted from the discrete recurrent part of the network by running all of the training data through the network once without training and recording the state transition information. The emission probabilities of each state in the extracted machine are set to the frequencies of the symbols being emitted when going through that state.

The verification process involves verifying a necessary condition of the extracted machine based on the assumption that the training data was generated from a finite state probabilistic automaton. We use a property implied by the definition of finite state probabilistic automata which states that the emission probabilities of a state depend on that state itself *only* and nothing else, i.e., it is a first-order Markov process.

To state it mathematically, let $x_0, x_1, \ldots, x_{i-1} \in \Sigma^*$ be any string for which the automaton $M =< \Sigma, U, u_0, \delta, P, F >$ arrives at state $u_i \in U$ after processing that string. By the definition of the automaton, there is a unique sequence of transitions and states associated with the processing of the string, i.e., $\exists u_0, u_1, \ldots, u_{i-1}, u_i$ such that

$$\delta(u_0, x_0) = u_1,\ \delta(u_1, x_1) = u_2,\ \ldots,\ \delta(u_{i-1}, x_{i-1}) = u_i.$$

The property of the automaton M says that given the present, the past and future are

independent. In other words, all relevant information about the past is contained in the present state[Arb69].

Thus a necessary condition for the extracted automaton to be a correct one is: *the emission probabilities of a state depend on the state itself, and are independent of the transitions that lead to it,* i.e.,

Condition 4.1 (First-Order Necessary Condition)

$$P(x|u_{i-1}, u_i) = P(x|u_i) \tag{4.11}$$

holds for $\forall x \in \Sigma$, $\forall x_0, x_1, \ldots, x_{i-1} \in \Sigma^*$, $\forall u_i \in U$, *such that the ending state of* $x_0, x_1, \ldots, x_{i-2}$ *is* u_{i-1}, *and the ending state of* $x_0, x_1, \ldots, x_{i-1}$ *is* u_i, *i.e.,* $\delta(u_{i-1}, x_{i-1}) = u_i$.

This is called first-order condition since we only check one time step back into the past for past-future dependencies. The full dependency check will be discussed in the next subsection.

We use Condition 4.1 in a hypothesis test. The hypothesis states that the extracted machine M' is equivalent to the true underlying machine M that generated the data. If the hypothesis is true, then the data can be equivalently generated by M'. To test the hypothesis, we pretend that the data was generated by M', and look for possible contradictions as follows: during the running of all data through the network when extracting the machine, for each transition rule $\delta(u_{i-1}, x_{i-1}) = u_i$ in M', we record probabilities(frequencies) of symbols *following* that transition, and compare this "emission probability" of a *transition*, denoted as $\hat{p}(x|u_{i-1}, u_i)$, with the emission probability of the *state* u_i that the transition is going to, denoted as $\hat{p}(x|u_i)$. If our hypothesis is true, then Condition 4.1 has to be satisfied, i.e., these two emission probabilities should be equal. Otherwise, the hypothesis is false.

Since we only have limited data at hand, event frequencies have to be used for probabilities. Due to irregularities in the data, two probabilities/frequencies may not be exactly equal even if the extracted machine is the correct one. Hence, some error should be allowed between the two probability estimates. Due to the fact that the size of the data available for probability comparison varies from situation to situation, we should allow larger errors for smaller sized data. Moreover, different values of the probabilities may require different error tolerances as well. Thus an adaptive threshold is needed, so that if two

probability estimates are close enough such that their difference is within the threshold, they are considered statistically equal. First, we need a probabilistic measure to evaluate our confidence in a frequency count from a data set.

Lemma 4.1 *Let p be the probability of event X happening for a probabilistic test T. Let n be the number of tests, and k be the number of times event X happened. Then*

$$E\left(\frac{k}{n}\right) = p, \tag{4.12}$$

$$Var\left(\frac{k}{n}\right) = \frac{p(1-p)}{n}. \tag{4.13}$$

The above lemma tells us that the frequency count is an unbiased estimation of p, and provides us quantitatively the variance of the frequency count in terms of the true probability and the sample size. The proof is given in Appendix C.

Lemma 4.1 suggests an adaptive measure to evaluate the goodness of our estimates. Since the events of emitting any symbols from the alphabet are mutually exclusive, we can treat each symbol separately.

To check Condition 4.1, we need to evaluate the distance between two frequency counts as probability estimates: $\hat{p}(x|u_{i-1}, u_i)$ and $\hat{p}(x|u_i)$. Note that the sample for $\hat{p}(x|u_{i-1}, u_i)$ is a subset of the sample for $\hat{p}(x|u_i)$. The next lemma gives us the variance of the difference between two frequency counts, one of which is computed from a sub-sample of another.

Lemma 4.2 *Let p be the probability of event X happening for a probabilistic test T. Let m be the number of tests, and k_m be the number of times event X happened. Let n be the size of a subset of the above tests ($n \leq m$), and k_n be the number of times event X happened in that subset ($k_n \leq k_m$). Then*

$$Var\left[\left(\frac{k_m}{m} - \frac{k_n}{n}\right)\right] = p(1-p)\left(\frac{1}{n} - \frac{1}{m}\right). \tag{4.14}$$

The proof is also given in Appendix C. This lemma gives us a quantitative estimate of how far apart two frequency counts will be, thus we can use this measure to evaluate the similarity between two probability estimates. However, the true probability information is not available, our own estimation has to be used in the formula in place of the true probability p. Since we are comparing two probability estimates, it is reasonable to use

the one with a larger sample size, i.e., the emission frequency directly from the state, in place of the true probability. Thus our empirical criterion for the first stage verification is:

Criterion 4.1 *Let m be the sample size of all instances of a state u, n be the sample size of all instances of a transition $\delta(u_i, y) = u$ that leads to u ($m \geq n$). Let $\hat{p}_u = \hat{p}(x|u) = \frac{k_m}{m}$, be the estimated probability of emitting a symbol x by frequency count in the m state sample, and $\hat{p}_\delta = \hat{p}(x|u_i, u) = \frac{k_n}{n}$ be the probability estimation of the successor being x by frequency count in the n transition sample. If*

$$(\hat{p}_u - \hat{p}_\delta)^2 \leq \alpha \hat{p}_u(1 - \hat{p}_u)\left(\frac{1}{n} - \frac{1}{m}\right), \tag{4.15}$$

where α is some constant, then $\hat{p}_u$ and $\hat{p}_\delta$ are considered statistically equivalent, denoted as $\hat{p}_u \stackrel{*}{=} \hat{p}_\delta$. Otherwise, they are considered statistically not equivalent, denoted as $\hat{p}_u \stackrel{*}{\neq} \hat{p}_\delta$. If $\hat{p}_u \stackrel{*}{=} \hat{p}_\delta$ for $\forall x$, $\forall u$ in M', and $\forall \delta$ leading to u, then Condition 4.1 is considered statistically true. Otherwise, it is considered statistically false.

The constant α is defined by the user, depending on how confident he/she wants the network's solution to be. Let $\eta = \frac{(\hat{p}_u - \hat{p}_\delta)^2}{\hat{p}_u(1-\hat{p}_u)(\frac{1}{n} - \frac{1}{m})}$, the ratio of the squared difference between the two probability estimates and the variance of the difference. Then in the above inequality, α is the upper bound allowed on all η's derived from the extracted machine.

The best way to choose α from the sampling distribution theory point of view is as follows: Find the distribution of η from the true machine, given the sample sizes n and m. Then for a certain confidence level, e.g., 95%, calculate, according to the distribution of η, the cut point value α, such that η has less than 5% probability of lying outside α. That way, α is chosen adaptively according to the machine considered and the available sample size, and the extracted machine that passes the criterion has an accurate confidence level associated with it. However, there are several difficulties with this method. First of all, the true machine is not known so we do not have the accurate information of states and transition probabilities. Secondly, even if we do know the true machine, or as before, assuming the extracted machine probability can be used in place of the true machine probabilities as approximations, the calculation of the distribution of a single η is a combinatorial problem, and appears to be very complicated. Thirdly, not all η's are

independent: the emission probabilities of a given state should sum up to 1, thus their corresponding η's are dependent with each other. In addition, states that are connected with transitions also have non-independent samples. There is no obvious way of calculating the joint distribution of all η's in a given machine.

In our experiments, we have heuristically chosen a fixed α, which has worked well with all the grammars we considered. By doing so, our heuristic assumption is that there is sufficient data to provide all necessary information, and that there is not too much fluctuation in terms of frequency counts from the data set. We cannot provide an accurate confidence level to associate with our solutions, nor can it be guaranteed that a fixed α would work for all machines and data sets, but from our empirical results of various experiments on different machines, the algorithm with our fixed α appears to be robust. It remains an open question as to whether there is a simple, adaptive way of setting α according to the machine and data size.

In essence, the criterion considers two probability estimates as statistically not equivalent if they are further apart from each other than $\sqrt{\alpha}\sigma$, where σ is the standard deviation of the distance between the two estimates. Thus the larger the value of α, the less reliable the solution will be. We have found that the algorithm is not sensitive to α in the range of 3.5 and 10 for the grammars we considered. A smaller α sometimes results in a correct structure not passing the criterion check due to data irregularity, while a larger α can result in the acceptance of an incorrect machine. In the experiments described later, α is always set to 3.5.

Note that we only need one round of data presentation to carry out both the state machine extraction and the emission symbol count for both transitions and states. The checking of the necessary condition is carried out with one scan of all transitions.

4.7.2 Second Stage Verification: the Sufficient Condition

The satisfaction of the first stage verification does not guarantee that the network has arrived at a correct solution of state configuration, since it is only a necessary condition that has been verified.

A good example of a finite state machine satisfying the necessary condition yet which is not equivalent to the true machine is shown in Figure 4.8 for the simple symmetric grammar. This machine was initially found by a network trained on the simple sym-

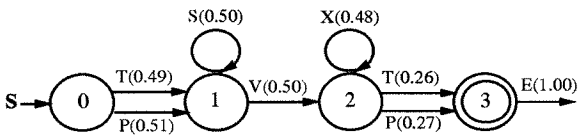


Figure 4.8: Initial results from a network learning the simple symmetric grammar

metric grammar. Similar such false solutions were found initially by all networks trained on grammars with identical sub-parts (except the $ac^*a \cup bc^* b$ grammar) described in Section 4 with a sufficient number of hidden units. It is interesting to see that the network always tries to commit the same states to those that have identical behaviors in the true grammar. For example, in the simple symmetric grammar, states 1 and 2 have the same emission probabilities, and the states that follow them also have the same emission probabilities, respectively. They are merged into one single state 1 in the network's false solution. States 3 and 4 are also merged, since once states 1 and 2 are merged, there is no way to distinguish 3 and 4. On close observation, one can see that these false solutions do indeed satisfy Condition 4.1.

To identify such false solutions, we need a stronger condition for further verification. Recall that the true solution satisfies the condition that the state emission probabilities are independent of *all previous information* except the state itself, where "all previous information" includes all preceding states and/or symbols. During the first stage of verification, we only tested the fact that the state emission probabilities are independent of its immediate previous state/symbol. The most powerful check is clearly on the dependencies of *all* previous states/symbols, i.e., a *complete* check such that the following is satisfied:

Condition 4.2 (Sufficient Condition) *Let D_I be an infinite set of strings generated by a probabilistic finite state automaton. Let R_I be the set of prefix strings contained in D_I. Let $M' =< \Sigma, U', u'_0, \delta', P', F' >$ be the derived machine being checked.*

The condition says that for all $x_0, x_1, \ldots, x_{i-1} \in R_I$, $\exists$ a unique state sequence $u'_1, u'_2, \ldots, u'_{i-1}, u'_i \in M'$, such that

$$\delta'(u'_0, x_0) = u'_1,\ \delta(u'_1, x_1) = u'_2,\ \ldots,\ \delta(u'_{i-1}, x_{i-1}) = u'_i,$$

and

$$p(x|x_0, x_1, \ldots, x_{i-1}) = P'(x|u'_0, u'_1, \ldots, u'_{i-1}, u'_i) = P'(x|u'_i), \tag{4.16}$$

where p stands for the true language probabilities from D_I, and P' stands for the transition probabilities of M'.

This condition checks against all possible language strings, assuming we have infinite data. The following theorem shows that this condition is a sufficient condition on the equivalence between the extracted machine and the true machine. The proof is very straight-forward.

Theorem 4.1 *Given an infinite data set D_I generated by a probabilistic automaton M as defined in Section 4.1, if a machine M' satisfies Condition 4.2 for all strings in D_I, then:*

$$M' \equiv M,$$

where "$\equiv$" means that the left-hand side is equivalent to the right-hand side in the sense that they accept/generate the same language strings with the same probabilities.

Proof:

Let $M =< \Sigma, U, u_0, \delta, P, F >$, and $M' =< \Sigma, U', u'_0, \delta', P', F' >$.

Let R_I be the set of prefixes contained in D_I.

Consider any $r = x_0, x_1, \ldots, x_{i-1} \in R_I$. Since D_I is generated by M, $\exists$ a unique state sequence $u_1, u_2, \ldots, u_{i-1}, u_i \in M$, such that

$$\delta(u_0, x_0) = u_1,\ \delta(u_1, x_1) = u_2,\ \ldots,\ \delta(u_{i-1}, x_{i-1}) = u_i,$$

and

$$p(x|r) = P(x|u_i).$$

Since M' satisfies Condition 4.2, we have: $\exists$ a unique state sequence $u'_1, u'_2, \ldots, u'_{i-1}, u'_i \in M'$, such that

$$\delta'(u'_0, x_0) = u'_1,\ \delta(u'_1, x_1) = u'_2,\ \ldots,\ \delta(u'_{i-1}, x'_{i-1}) = u'_i,$$

and

$$p(x|x_0, x_1, \ldots, x_{i-1}) = P'(x|u'_0, u'_1, \ldots, u'_i) = P'(x|u'_i).$$

Thus,

$$p(x|r) = P'(x|u'_i) = P(x|u_i) \quad \forall r \in R_I,$$

i.e., M and M' accept/generate the same language strings with the same probabilities.

Q.E.D.

To verify the condition for every possible string is impractical. We can only use limited data to support our findings. Our verification proceeds as follows. The training data set is first sorted in alphabetical order. Starting from the shortest possible prefix "B," which is present in all strings, find out what are the probabilities/frequencies of symbols following it, and compare this set of probabilities with the emission probabilities of the state the prefix leads to. Since we do not have perfect data, similar to the error tolerance derived for the first stage verification, our empirical criterion for the second stage verification is:

Criterion 4.2 *Let m be the sample size of all instances of a state u, n be the sample size of all instances of a prefix r that leads to that state, then $m \geq n$. Let $\hat{p}_u = \hat{p}(x|u) = \frac{k_m}{m}$ be the estimated probability of emitting a symbol x by frequency count in the state sample, and $\hat{p}_r = \hat{p}(x|r) = \frac{k_n}{n}$ be the probability estimation of the successor being x by frequency count in the prefix sample. If*

$$(\hat{p}_u - \hat{p}_r)^2 \leq \beta \hat{p}_u(1 - \hat{p}_u)\left(\frac{1}{n} - \frac{1}{m}\right), \tag{4.17}$$

where β is some constant, then $\hat{p}_u$ and $\hat{p}_r$ are considered statistically equivalent, denoted as $\hat{p}_u \stackrel{*}{=} \hat{p}_r$. Otherwise, they are considered statistically not equivalent, denoted as $\hat{p}_u \stackrel{*}{\neq} \hat{p}_r$. If $\hat{p}_u \stackrel{*}{=} \hat{p}_r$ for $\forall x$, $\forall u$ in M', and $\forall r$ leading to u, then Condition 4.2 is considered statistically true. Otherwise, it is considered statistically false.

Again, the constant β here is defined by the user, depending on how confident he/she wants the network's solution to be, and consideration of data irregularity should be taken. Similar to the choice of α for Criterion 4.1, the best way to choose β from the sampling distribution theory point of view is very complicated, thus we choose to use a fixed β instead. The experimental results show that the algorithm with our fixed β value is robust on all grammars considered in this chapter. We have found that the algorithm is not sensitive to β in the range of 3.5 and 10. In the experiments described later, β is always set to 3.5.

This is repeated alphabetically for all prefixes that have more than a certain number of occurrences present in the data set. The remaining prefixes that have relatively small numbers of occurrences are ignored to make the criterion check more statistically reliable.

The reasonable assumption here is that those prefixes that are being checked provide sufficient information about the grammar. In our experiments, the minimum number of occurrences needed for a prefix to be checked is set to 30. As can be seen from the following analysis of sufficient data conditions, the algorithm requires that this number be large in order to be able to distinguish two close together but different probabilities. The larger the number, the more capable the algorithm is in differentiating small differences.

Note that this verification process also requires only one scan of all symbols in the training set, provided that the set is read and sorted inside the memory, where symbols can be randomly accessed. Note also that this second state verification process which is more time-consuming than the first stage, does not get initiated unless the first stage necessary condition is satisfied.

The problem still remains as to whether the satisfaction of Criterion 4.2 guarantees that the extracted machine has an equivalent structure to the true underlying machine, since we have no way to check the full sufficient Condition 4.2. To find out the answer, we need to elaborate more on our initial assumption that the data provides "sufficient information" on the true underlying machine M. Exactly what is the definition of "sufficient information?"

Obviously, the definition has to be related to the true machine M itself: machines with different structure and transition probabilities would require different amounts of information for reliable inference. One intuitive definition is:

To provide sufficient information on M, the data set D has to be large enough so that M is the smallest unique machine to describe D statistically.

This would guarantee that if we find a machine that describes the data statistically, and that no smaller machines can do this, then this machine is the true one. However, the above definition is still too vague: how do we define "smallest and unique?" And how do we define "describe something statistically?" First of all, the following theorem answers the question of what makes M non-redundant and unambiguous on an infinite data set:

Theorem 4.2 *Given an infinite data set D_I generated by a certain finite state probabilistic automaton $M =< \Sigma, U, u_0, \delta, P, F >$, and let R_I be the set of prefix strings contained in D_I, if M is the smallest and unique machine that generates D_I, then it satisfies:*

1. $\forall u_i \in U$, $\exists r_i \in R_I$, *such that* r_i *leads to* u_i.

2. $\forall u_i, u_j \in U$, $u_i \neq u_j$, *the following is true:*

$\forall r_i$ that leads to u_i, and $\forall r_j$ that leads to u_j, $\exists$ suffix $w \in \Sigma^*$, $x \in \Sigma$, such that $r_i w, r_j w \in R_I$, $r_i w$ leads to state u_{i_w}, $r_j w$ leads to state u_{j_w}, $u_{i_w} \neq u_{j_w}$, and $P(x|u_{i_w}) \neq P(x|u_{j_w})$.

The first condition guarantees that no redundant state exists. The second condition ensures that each state is uniquely different from any others. The proof is given in Appendix D. The next theorem answers the question of what makes the data statistically sufficient for M.

Condition 4.3 (Sufficient Data Condition) *Let $M =< \Sigma, U, u_0, \delta, P, F >$ be a probabilistic automaton as defined in Section 4.1, and satisfying the conditions in Theorem 4.2. Let D be a finite set of strings generated statistically by M, and R be the set of prefix strings contained in D that has more than N appearances, where N is the threshold set for the Criterion 4.2 check to happen. Let $r_i^s \in \Sigma^*$ be the shortest (or one of the shortest) prefix(s) that leads to state $u_i \in U$. The conditions on the data are:*

1. *$r_i^s \in R$.*

2. *$\forall u_i, u_j \in U$, $u_i \neq u_j$, i.e., $r_i^s \neq r_j^s$, then*
   for some suffix $w \in \Sigma^*$, $x \in \Sigma$, that satisfies condition 2 in Theorem 4.2 for r_i^s and r_j^s, the following is true for D:
   $r_i^s w, r_j^s w \in R$, and $|\hat{p}_r(x|r_i^s w) - \hat{p}_r(x|r_j^s w)| > \sqrt{\frac{\beta}{N}}$,
   where β is the constant defined in Criterion 4.2.

3. *$\forall u_i \in U, x \in \Sigma$, such that $P(x|u_i) \neq 0$, $r_i^s x \in R$.*

4. *$\forall u_i, u_j \in U$, $x \in \Sigma$, such that $P(x|u_i) \neq 0$, and $j \neq \delta(u_i, x)$, then*
   for some suffix $w \in \Sigma^*$, $y \in \Sigma$ that satisfies condition 2 in Theorem 4.2 for $r_i^s x$ and r_j^s, the following is true for D:
   $r_i^s x w, r_j^s w \in R$, and $|\hat{p}_r(y|r_i^s x w) - \hat{p}_r(y|r_j^s w)| > \sqrt{\frac{\beta}{N}}$.

5. *M passes Criterion 4.2 check on D,*

where $\hat{p}_r(x|r)$ is the probability estimate of emitting x given prefix r by frequency count from D.

The first two conditions ensure that the data contains information that reflects each one of M's unique states. The next two conditions guarantee that the data contains information that reflects each one of M's unique transitions: each transition unambiguously leads to a unique state. The last condition makes sure that the data is statistically faithful to M.

The condition also suggests a way to calculate a lower bound on the size of the data set D in order for it to be sufficient: For each prefix string r among the $r_i^s w$'s and $r_i^s xw$'s as described above for all $u_i \in U$, calculate p_r, the probability of M generating r, by multiplying consecutive transition probabilities of M when processing r. Find r_{min}, the least probable prefix string among all of the above. The lower bound for the size of D is then $N/p_{r_{min}}$. For example, for the Reber grammar with equal probable transition splits on all states, r_{min} of all $r_i^s w$'s and $r_i^s xw$'s happens to be also the longest prefix among them, since all transitions that are neither B nor E have the same probability 0.5. The r_i^s's for states $0, 1, 2, 3, 4, 5$ are: $B, BT, BP, BTX, BPV, BTXS (or BPVV)$, respectively. The two states that are hardest to distinguish are states 1 and 3, both having the same emission probabilities. The shortest suffix w to distinguish them can be either S or X, both of length 1. Thus $r_{min} = r_1^s xw$, where $r_1^s = BT$, $x = S$, $w = S$ or X. Therefore, $p_{r_{min}} = (0.5)^3 = 1/8$, with N being 30 for our experiments, the lower bound on the size of sufficient data for this grammar is then $N/p_{r_{min}} = 240$.

Finally, the next theorem shows that if the above set of conditions are satisfied, then our intuitive definition for sufficient data is guaranteed to be true for the statistical Criterion 4.2. In other words, given a data set D that satisfies Condition 4.3, the smallest machine that satisfies Criterion 4.2 is guaranteed to be statistically equivalent to the correct smallest true machine.

Theorem 4.3 *Given data D that satisfies Condition 4.3 for a minimum and unique machine M, the smallest extracted machine M' that passes Criterion 4.2 satisfies:*

$$M' \stackrel{*}{=} M,$$

where "$\stackrel{*}{=}$" stands for "statistically equivalent" as defined in Criterion 4.2.

The proof is provided in Appendix E. With this theorem, we can be assured that with a large enough data set reflecting all aspects of M, Criterion 4.2 is sufficient for the

testing of our extracted machine M' within a statistical limit, provided that M' cannot be minimized.

Note that the second and the fourth constraints described in Condition 4.3 put a limit on the true machine M that can be derived. The successor probabilities of two prefixes (which lead to different states) may have to differ at least by the amount $\sqrt{\frac{\beta}{N}}$. Thus if the true machine M has two states such that *none* of their emission probabilities of *any* symbol differs by more than $\sqrt{\frac{\beta}{N}}$, *and* that they both go to the same next states with the emission of the same symbols, the data would never satisfy Condition 4.3, and our algorithm would never be able to distinguish the two states. On the other hand, these can be considered rare cases with large N. And even if there exist two such states as above, they can be merged into one single state without an error more than $\sqrt{\frac{\beta}{N}}$ in probability estimation. In order to be able to distinguish such states, we can always increase the number N, and require a larger data size. It is then a trade-off between the cost and the accuracy of the inference. In our experiments, $\sqrt{\frac{\beta}{N}} = 0.34$, which is relatively large, but is still easily satisfied by all the machines we studied: any two states in the machines that transition to the same next states have a difference of at least 0.5 in their emission probabilities.

As we will see in the next section, the network typically starts with very few states, and tries to "grow" more and more states as learning proceeds, which is similar to a "greedy" search. After passing both stages of the verification process, it usually arrives at a solution equal to or not much larger than the minimal one. To make sure that we have a minimum machine, we can try to minimize it by a process similar to what is described in the proof of Theorem 4.2 in Appendix D. Due to frequency fluctuations of the data however, we sometimes arrive at a machine that is larger than the minimum one and cannot be reduced further based on the information provided in the data, yet they are still relatively close to what is intended, as will be seen in the next section.

The verification process can be easily applied to discrete networks, where a stable finite automaton can be extracted at any point during learning. For any given connection weight setting, a discrete network behaves exactly like *some* automaton, with stable state representations. For analog networks, however, this is not the case, and so the verification can not be carried out due to unstable internal states.

4.7.3 What Comes After the Verification

Once the state configuration structure of the network is verified, our inference task can be said to be accomplished, since the verification process provides us a side benefit of the state transition probability estimates, which together with the extracted machine structure make a complete description of the language being learned.

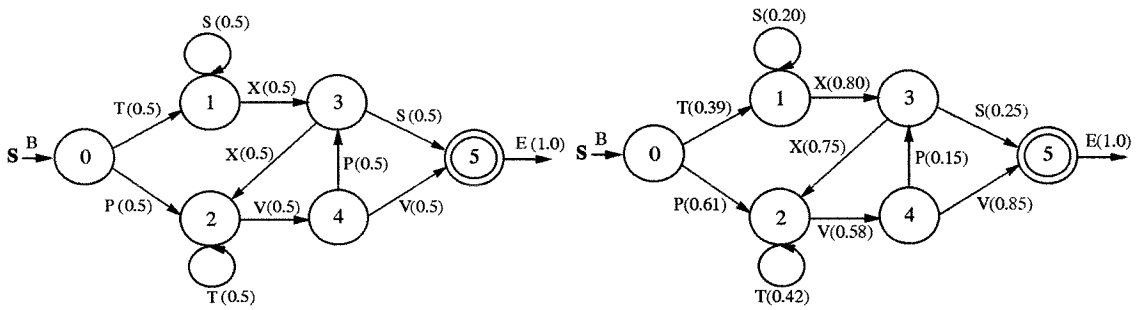
If we insist on having a network version of the language generator/predictor/classifier, i.e., if we were to use the stand-alone network for future language related tasks without referring to the extracted automaton, then further training of the feedforward part of the network is needed.

Note that when the verification process declares that the network has found a good state configuration, it only means that the discrete recurrent part of the network has reached a desired status. It does not give us direct information on the performance of the analog feedforward part of the network. On the other hand, the process of the recurrent part reaching a good solution is only with the help of information input through the feedforward part, where prediction error is back-propagated to the "previous state layer." Our observation is that by the time the recurrent part of the network has reached a good solution, the feedforward part is still in its initial learning stage, i.e., the outputs do not match the true probabilities very well. It still needs further training, preferably with the recurrent part of the network weights frozen, if the network were to be used as the finite state automaton.

With the help of the emission probability estimates obtained during the verification process, the training of the feedforward part of the network is very straight-forward: a second-stage training set that consists of all state-emission probability pairs can be generated, and the feedforward part is trained with simple back-propagation. This avoids any possible "over-emphasizing" of states that have large probabilities to be visited when learning the probabilities directly from the initial data.

4.8 Experimental Results on Grammars Without Identical Sub-Parts

Incorporating the two adaptive criteria for verification during learning indeed helped the learning process to identify a good solution, and stop the training as soon as it is



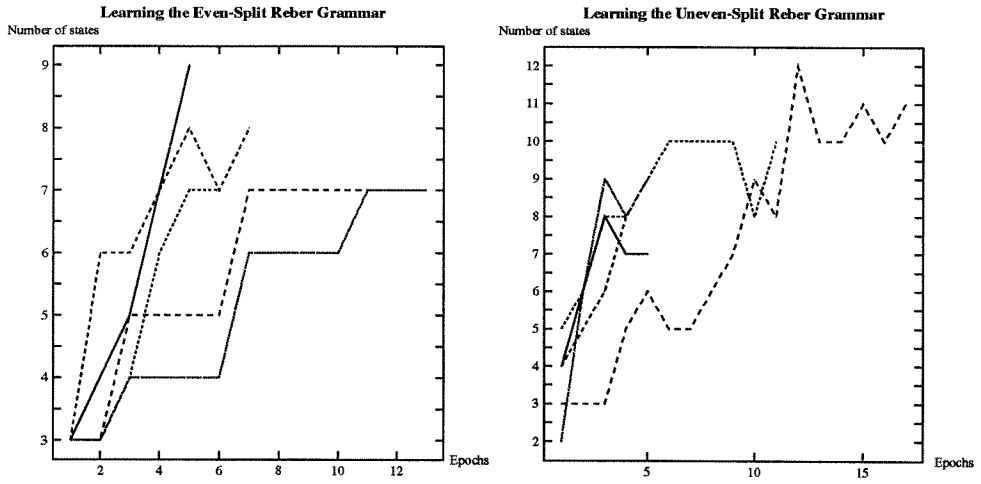
(a) (b)

Figure 4.9: Probabilistic Reber grammars

found.

For the case of the Reber grammar, it turns out that the network converges very quickly, and learning is stopped mostly within 20 epochs through a data set of at most 1000 strings, a fact that was not known to us when the verification process was not implemented. Two cases of the Reber grammar with different transition probabilities are studied. They are shown in Figures 4.9(a) and (b). Figure 4.9(a) is the case where even split probabilities are assigned to all state transitions. Figure 4.9(b) has uneven split probabilities.

As mentioned in Subsection 4.7.2, a stable finite state automaton can be extracted from a discrete network at any time during training by freezing all weights and running the training data through the network. We can thus record the number of states in the automaton the network has found after every learning epoch, and observe how the number of states evolve with training. Note again that the advantage of the discrete network is that one can easily identify states during the learning process, i.e., they are explicit. It is interesting to observe that the network starts with an internal representation consisting of a very small number of states, and tries to "grow" the states during learning to fit the data. In figures 4.10(a) and (b), the processes of "state growing" are shown for two sets of networks learning the grammars in Figures 4.9(a) and (b), respectively. Both figures contain plots for the same five networks with different weight initializations. All networks have 4 units in the recurrent "state" layer, and 5 units in the feedforward intermediate layer. The number of states in a network's internal representation is plotted against the learning epochs. As can be seen that the network adds more and more states almost



(a) (b)

Figure 4.10: The process of growing states during network learning.

monotonically as training proceeds.

Figures 4.11(a) and (b) show the machines derived a network trained on the even split and uneven split Reber grammar, respectively, each after learning was stopped when the verification process returned true. Probabilities are assigned directly from the results obtained by the verification process. As can be seen, they are equivalent to the true machines in Figure 4.9 within a very small error tolerance. The small errors in the transition probabilities derived are strictly due to frequency fluctuations in the data.

For each Reber grammar, 10 runs are made using different network weight initializations. All networks have the same number of units as described above. The training set of the even split Reber grammar contains 600 strings. For the uneven split Reber grammar, more strings are needed to make the data statistically reliable for the derivation of those transitions that have relatively small probabilities associated with them. The training set of the uneven split Reber grammar contains 1000 strings. Table 4.1 gives detailed results on these runs. Each row corresponds to one of the network trained on both grammars. The number of epochs needed to find a correct solution, and the number of states that solution contains are shown for each grammar. All networks converged to equivalent structures to the Reber grammar. The last row shows the average numbers over all runs. Note that the minimum machine contains 6 states. On average, the networks were able

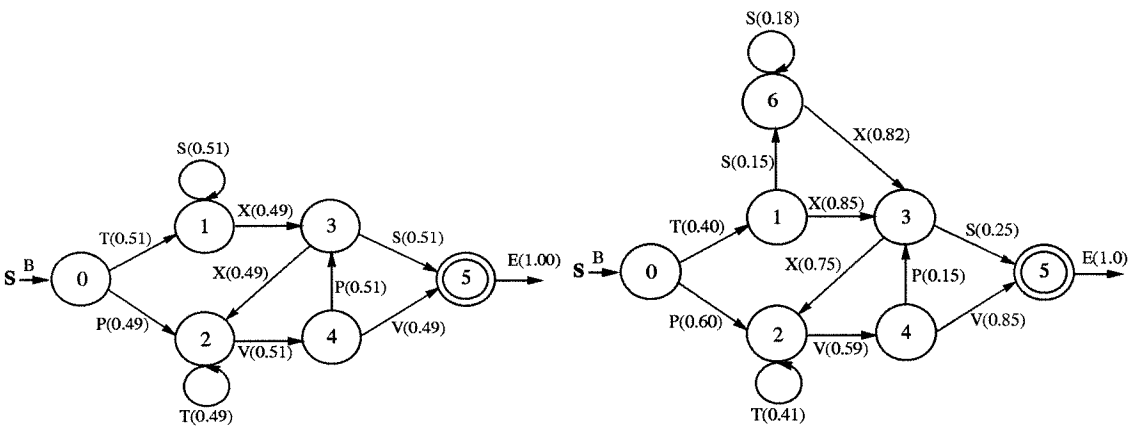


Figure 4.11: Network-derived probabilistic Reber grammar

network	*Even-split Reber grammar*		*Uneven-split Reber grammar*	
number	*epochs*	*states*	*epochs*	*states*
1	7	7	5	6
2	13	6	17	10
3	6	6	11	9
4	5	8	4	7
5	13	6	5	8
6	11	6	6	8
7	3	6	7	6
8	8	7	3	7
9	11	6	6	6
10	4	6	6	7
average	8.1	6.4	7.0	7.4

Table 4.1: Experimental results on learning probabilistic Reber grammars

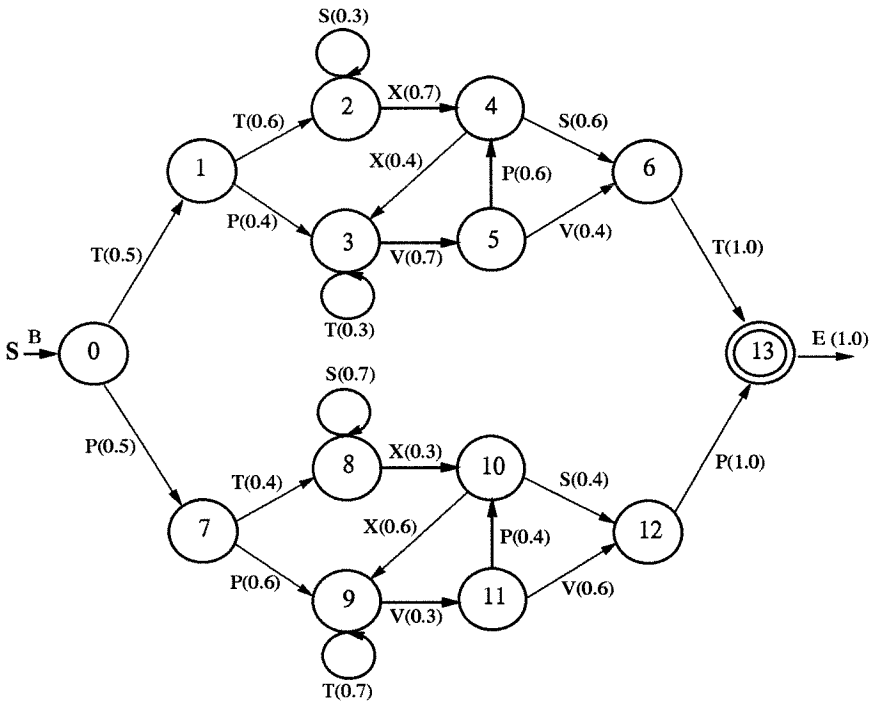


Figure 4.12: The non-symmetric embedded Reber grammar.

structures to the Reber grammar. The last row shows the average numbers over all runs. Note that the minimum machine contains 6 states. On average, the networks were able to find machines very close to minimum size to describe the data for the even split Reber grammar. Note that even though the average number of epochs needed for the uneven split grammar is smaller than that for the even split grammar, it in fact takes longer to train since the training set for the former has 1000 strings, as compared to only 600 for the latter. It is reasonable to see longer training times needed and bigger machines derived by the networks to describe the uneven split Reber grammar due to increased fluctuations in the data set with some probabilities much smaller than others.

To compare the network's behavior in learning grammars with and without identical sub-parts, we experimented with the embedded Reber grammar structure with non-identical transition probabilities assigned to the two Reber grammar sub-parts. It is shown in Figure 4.12. Note this grammar has the same structure as the symmetric embedded Reber grammar, but the small differences in probability assignments make it non-symmetric, thus it is a grammar without identical sub-parts. The grammar is a much more complicated one than the Reber grammar, and thus requires a much larger network to learn.

network number	*training epochs*	*number of states*
1	41	26
2	130	34
3	>200	-
4	66	38
5	48	25
average	71.25	30.75

Table 4.2: Experimental results on learning the non-symmetric embedded Reber grammar

(See the discussion on network capacity in Chapter 2.) Five networks with 11 state units and 9 intermediate units and different initial weights are trained on a data set of 2400. The results are shown in Table 4.2. As will be seen in the next section, a small change in transition probabilities that makes the grammar symmetric leads to great difficulty in learning for the network. Similar observation is made in [CSSM89].

4.9 The Difficulty in Learning Grammars With Identical Sub-Parts

It is observed in our experiments that in learning grammars with identical sub-parts (except the $ac^*a \cup bc^*b$ grammar), the network always quickly finds a false solution that satisfies Criterion 4.1 first. It then becomes extremely difficult for it to split states that should be different but are merged into one in the false solutions. Prolonged training does not seem to help. Similar observations can be found in [CSSM89, SZ89].

As discussed in Section 4.3, the symmetric $ac^*a \cup bc^*b$ grammar turns out not to be a hard problem for the network, which discovers the first and last letter dependency from the two shortest strings aa and bb before any possible false solution is to be formed. Table 4.3 shows the results of five networks trained on the grammar, each with 3 hidden units and 4 intermediate units. The training set contains 1000 strings. Since our algorithm requires a large data set, and the experiments done in [SO93] uses only very small data sets (up to 20 sample strings), we cannot conduct a direct comparison. (Due to the small sample size, the probability estimates resulting from the model merging algorithm in [SO93] are far off from the correct ones.) It would be interesting to apply the model merging algorithm to the large sample set we used for our network training and compare the results from the

network number	*training epochs*	*number of states*
1	8	4
2	79	5
3	19	4
4	20	4
5	182	4
average	61.6	4.2

Table 4.3: Experimental results on learning the $ac^*a \cup bc^*b$ grammar

two algorithms.

For the hard to learn grammars with identical sub-parts, we employed a new training algorithm which dynamically augments the network when such false solutions are found, and forces the network to learn the long distance dependencies while retaining what it has already learned. A false solution is defined to be one that satisfies Criterion 4.1 but fails Criterion 4.2.

The training starts with the original pseudo-gradient learning process as described in Section 4.6, for a fixed number of state and intermediate units as before, until Criterion 4.1 is found satisfied. If Criterion 4.2 is satisfied as well, as were the cases during the learning of grammars without identical sub-parts and the $ac^*a \cup bc^*b$ grammar, then the learning can be stopped with a verified good solution. Otherwise a false solution has been found and the training is switched to the augmentation mode: The network is augmented by one new state unit, denoted by S_E. Figure 4.13 shows the augmented network structure.

The new unit S_E is allowed connections to itself, and to the intermediate layer, but not to any of the other state units. The recurrent part of the network weights from the original network, i.e., w_{ij}^{cn}'s and w_{ij}^{in}'s, are frozen, indicated in the figure by solid lines. The feedforward part of the network weights, along with the newly added connection weights are to be adjusted in the learning that follows. These "free" weights are indicated by dashed lines. The reason for this arrangement is that the passing of Criterion 4.1 indicates that the network has learned part of the structure of the language and that part of the structure is worthwhile to keep. By freezing the original recurrent weights, we effectively keep the state transition relations already found so far.

The aim for adding the new unit is to have this unit trained specifically to distinguish

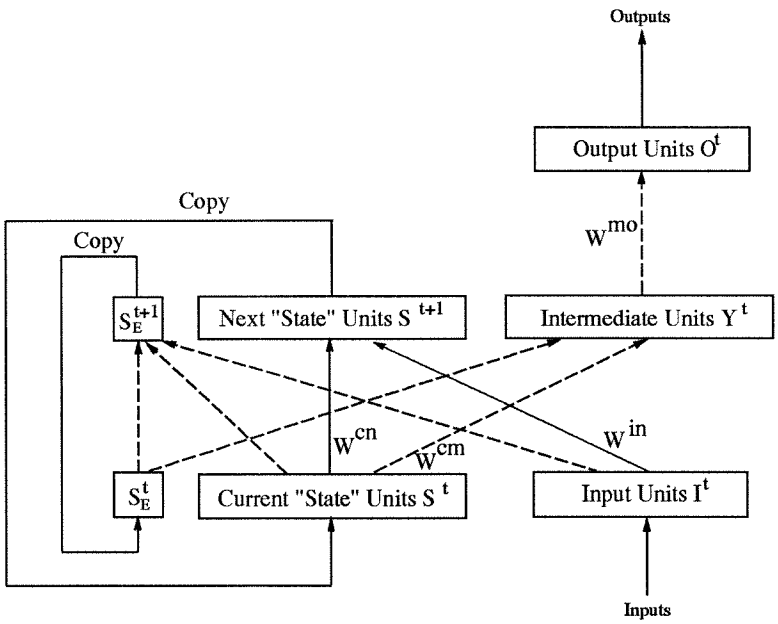


Figure 4.13: The augmented network

the long distance relations that have not been learned by the network. To make sure that the information already correctly learned by the network does not interfere with the augmented network training, we label in each string the prefixes for which the network fails the Criterion 4.2 check. The augmented network is then trained on these labeled prefixes only. Some strings may not contain any such prefixes. Thus training the augmented network differs from the training of the original network in the following:

- Instead of updating the weights after *every symbol* presentation, we wait till the *end of a labeled prefix* is reached, and update the weights specifically to reduce the error on those prefixes.

- Instead of calculating the pseudo-gradient on *all* weights, we only do the calculation on the *free* weights, i.e., those weights that are not frozen.

Note both of the above make the learning epoch significantly shorter than that of the original training. To deal with cases when adding one unit is still not sufficient for the network to learn the correct grammar, we repeat the augmentation process by adding more and more units until the network reaches a satisfactory solution, or the maximum number of epochs allowed has been reached. The criterion for adding another unit is that the augmented network has not been able to find a good solution after a certain maximum

number of epochs. This number is set to be 100 in our experiments. It remains an open question as to whether there can be an adaptive criterion for adding more units.

network	*False solution*		*Correct solution*		
number	*epochs*	*states*	*epochs*	*new units*	*states*
1	22	4	>350	>4	-
2	17	4	97	1	7
3	25	6	223	2	12
4	4	4	142	2	9
5	17	5	305	3	7
average	15.75	4.75	191.75	2.0	8.75

Table 4.4: Experimental results on learning the simple symmetric grammar

Table 4.4 shows the experimental results on five networks with different initial weights learning the simple symmetric grammar. All networks initially contain 4 units in the recurrent "state" layer, and 5 units in the feedforward intermediate layer. The training set contains 1000 probabilistically generated strings. Each row corresponds to one of the network trained on the grammar. The number of epochs needed for and the number of states contained in a solution are shown for both the false solution initially found, and the correct solution finally arrived at. For the correct solution, the number of new units added to the network during the augmented learning is also shown. All networks except one converged to equivalent structures to the simple symmetric grammar within 350 epochs. The last row shows the average numbers over the successful runs. Note that the network always finds a false solution very quickly, and even with augmented learning, it takes much longer for the newly added units to discover long distance dependencies.

For the symmetric embedded Reber grammar and the $a^+b^+a^+b^+$ grammar however, the network fails to find a correct solution even with augmentation for most of the runs. For the symmetric embedded Reber grammar, five networks each having 8 state units and 8 intermediate units with different initializations are trained on a data set of 2000. For the $a^+b^+a^+b^+$ grammar, five networks each having 5 state units and 6 intermediate units with different initializations are trained on a data set of 1000. For all networks, false solutions are found within 10 training epochs. Augmentation is employed for up to 350 epochs, with one new unit added every 100 epochs. For each of the grammars, 4 out of the 5 networks fail to find a correct solution within 350 epochs, with up to 4 new units

added. Table 4.5 and 4.6 show the results of the five runs for the symmetric embedded Reber grammar and the $a^+b^+a^+b^+$ grammar, respectively.

network	*False solution*		*Correct solution*		
number	*epochs*	*states*	*epochs*	*new units*	*states*
1	3	20	>350	>4	-
2	4	11	>350	>4	-
3	2	13	>350	>4	-
4	2	14	140	2	26
5	9	13	>350	>4	-

Table 4.5: Experimental results on learning the symmetric embedded Reber grammar

network	*False solution*		*Correct solution*		
number	*epochs*	*states*	*epochs*	*new units*	*states*
1	2	3	>350	>4	-
2	3	3	6	1	5
3	3	3	>350	>4	-
4	2	3	>350	>4	-
5	2	5	>350	>4	-

Table 4.6: Experimental results on learning the $a^+b^+a^+b^+$ grammar

It appears that the network model has an extremely strong tendency to commit the same internal states to those true machine states that have the same *immediate* behaviors: same emission probabilities. Note that states 1 and 2 in the $ac^*a \cup bc^*b$ grammar (Figure 4.4) do not have the same emission probabilities even though they are symmetric parts in the grammar, thus the network does not have trouble distinguishing the two. As discussed in Section 4.7, to differentiate between such "similarly behaved" states, a suffix is needed so that by processing the suffix from each of the two "similarly behaved" states the automaton is led to two "differently behaved" states: states that have different emission probabilities. The longer such suffix that is minimally necessary, the further back the the network is required to remember, and the harder it is for it to learn. In addition, the more such "similarly behaved" states there are in a grammar, the harder the problem.

In the simple symmetric grammar (Figure 4.3), the only pair of "similarly behaved" states are states 1 and 2. To distinguish them, the suffix V is needed, which is only of

length 1. Suffix V leads states 1 and 2 to states 3 and 4, respectively, which have very different emission probabilities. With the help of the augmented learning, the network is able to differentiate the two states most of the time.

The symmetric embedded Reber grammar on the other hand, has 5 pairs of "similarly behaved" states: one less than the size of the Reber grammar. The pair that requires the longest suffix to be differentiated is states 1 and 7, for which the suffix is of length 3: either TXS or PVV. This makes it a very difficult problem for the network, and even the augmented learning fails most of the time to force it to learn.

It is puzzling to see the difficulty the network has in learning the simple $a^+b^+a^+b^+$ grammar, where only one pair of "similarly behaved" states exists: states 1 and 3. The shortest suffix that is needed to distinguish the two states is b, which is only of length 1. In a sense, this grammar is very similar to the simple symmetric grammar, except that the two "similarly behaved" states are cascaded, instead of in parallel with each other. It takes at least two more symbols to reach state 3 from the beginning than to reach state 1 (Figure 4.5). The results show that the network has more difficulty in distinguishing "similarly behaved" states in cascade than in parallel.

It is clear from these experiments that the gradient-descent algorithm is not a powerful one in training the network to learn grammars with identical sub-parts. The high frequency and speed with which the network converges to false solutions via gradient descent and the difficulty for the algorithm to pull it away from such false solutions demonstrate that there are many local minimum points in the search space for such grammars, and very few global ones. To overcome this difficulty, a better training algorithm that avoids local minima, or a different network structure that does not create as many false solution local minima points in the solution space is worth investigating.

4.10 Summary and Future Work

We have presented in this chapter a recurrent network structure for the inference of probabilistic regular grammars. The network consists of a recurrent part which is aimed to represent the structure of the grammar, and a feedforward part which is aimed to represent the probabilities of the grammar. The pseudo-gradient learning is extended to train the recurrent part of such a network by using error information provided by the feedforward part. Theoretical analysis of necessary and sufficient conditions for the network's derived

structure to be correct are given. A two-stage verification process with adaptive error thresholds is derived for the case of limited data, and incorporated during training. In addition, an adaptive network augmentation process is used for the learning of grammars with identical sub-parts, which are known to be hard problems for recurrent networks. The experimental results demonstrate the effectiveness of the verification criteria and the learning algorithm in learning grammars without identical sub-parts. The augmented learning made a small improvement in the learning of grammars with identical sub-parts. The two criteria described in this chapter do not restrict their use in recurrent network learning: they can be applied to any inference algorithm for this class of probabilistic grammars, provided that the algorithm greedily grows states during inference, and derives the transition probabilities directly by frequency count from the given data.

The advantage of using discrete units to represent the structure of the grammar is again demonstrated by the facts that the verification process can be applied *during* learning, and that the learning of the structure of the grammar and of the probabilities can be easily separated.

It has been found that for grammars with identical sub-parts, there are many false solutions existing as local minima in the network's solution space and they are much easier to reach than the few correct solutions which are global minima, even with our augmentation process. Future research lies in investigation of other algorithms that do not rely on gradient descent, and other network structures that inherently prevent false solutions from becoming local minima.

To increase the network's representation power, one suggestion is to add an intermediate layer between the current state and the next state layers, so that limitations on the mapping of current and next state configurations can be eliminated. In addition, our algorithm can also be applied to higher ordered networks.

Chapter 5
Conclusion

5.1 Summary of Results

We have presented in this thesis a general scheme for training artificial recurrent neural networks to solve grammatical inference problems. The scheme consists of a discrete recurrent network structure to overcome the stability problem associated with conventional analog network structures, and the pseudo-gradient learning method to effectively train discrete networks. The discrete network and pseudo-gradient learning are applied to the inference of three types of grammars.

For regular grammars, a second-order discrete network structure along with the idea of pseudo-gradient training is introduced. Experimental results on training such networks to learn grammars with various levels of difficulty demonstrate the capability of the new network to be similar to its analog counterpart, while sustaining stable representations for arbitrary long strings after learning.

For a class of deterministic context-free grammars, the second-order discrete network structure is extended to include a discrete external stack. Pseudo-gradient learning is also extended. A composite error function is constructed to deal with various situations during learning. Empirical evidence supports our claim that the learning scheme is effective. Unlike its analog counterpart, the discrete network's operation on the external stack is directly interpretable.

For a class of probabilistic grammars, a combined discrete/analog structure is suggested from analysis of the inference problem. By using discrete units to represent the structure of the grammar, the learning of the structure can be separated from the learning of the probabilities. Pseudo-gradient learning is again extended for the combined network. Theoretical aspects of necessary and sufficient conditions on the correctness of a derived grammar are analyzed in detail. Empirical criteria for limited data are constructed and incorporated during learning. Results on learning several probabilistic grammars without identical sub-parts indicate the effectiveness of the verification criteria and the learning scheme. For grammars with identical sub-parts, where long distance dependencies exist, and which are particularly hard to learn, an adaptive network augmentation scheme is

used which results in some improvement of learning. The gradient descent mechanism is found not powerful enough for this particular type of grammars.

The general scheme of discrete network and pseudo-gradient learning can be extended and applied to any problems with discrete hidden structures, as well as to feedforward networks [GZ94]. Any successfully trained discrete recurrent network has the ability to process arbitrarily long strings without error.

The general criteria derived in Chapter 4 can also be applied to any inference algorithms for the same class of probabilistic grammars.

5.2 Future Research Directions

We have studied various aspects of using recurrent networks to derive grammars rules from examples: learning behavior, internal representation, capacity of networks and difficulties in learning certain grammars.

Yet several questions still remain unanswered and are worth investigating for future research.

1. Theoretical analysis of learning for analog networks: Why do analog networks tend to form clusters in activation space in learning? Can it be proven that they form unstable clusters most of the time?

2. Are there stable solutions for analog networks which have infinitely many points in a state representation? If so, what is the network capacity for such representations? And is there an effective training algorithm for finding such solutions?

3. Theoretical analysis of learning for discrete networks: What is the convergence property of discrete networks in learning an arbitrary grammar?

4. How can we design new learning algorithms and network structures for probabilistic grammars that can overcome the difficulty associated with the scheme presented in Chapter 4?

5. Extensions on the probabilistic network training: Can a network learn a general HMM? In addition, can we extend the structure even further to outperform HMM, in cases where the underlying relationships are not first-order Markov?

6. Can our model be applied to very large-scale problems, such as DNA sequence modeling and analysis? And can it be applied in problems other than grammatical inference, such as time sequence prediction and modeling?

The answers to the above questions will be useful for a even better understanding of recurrent networks in general, and for their applications in solving real-world problems.

Appendix A
Detailed Training Process for Regular Grammars

The operational equations of the network shown in Figure 2.4 are:

$$S_i^t = f(\sum_j w_{ij}^{x^t} S_j^{t-1}), \qquad \forall i, t,$$

where

$$f(x) = \frac{1}{1+e^{-x}}.$$

Error is calculated at the end of each string $x^0, x^1, ..., x^L$:

$$E = \frac{1}{2}(S_0^L - T)^2,$$

where

$$T = target = \begin{cases} 1 & \text{if "legal"} \\ 0 & \text{if "illegal"}, \end{cases}$$

and S_0^L is the "indicator" unit's activation at time step L.

Update w_{ij}^n, the weight from node j to node i in **net**n, at the end of each string presentation according to the error gradient:

$$w_{ij}^n = w_{ij}^n - \alpha \frac{\partial E}{\partial w_{ij}^n}, \qquad \forall\ n, i, j,$$

$$\frac{\partial E}{\partial w_{ij}^n} = (S_0^L - T)\frac{\partial S_0^L}{\partial w_{ij}^n}, \qquad \forall\ n, i, j,$$

where α is the learning rate, which is chosen to be 0.5 in all of our experiments.

To get the gradient value $\frac{\partial S_0^L}{\partial w_{ij}^n}$, the gradient values $\frac{\partial S_k^t}{\partial w_{ij}^n}$ for all t, k need to be calculated forward in time at each time step:

$$\frac{\partial S_k^t}{\partial w_{ij}^n} = f' \cdot (\sum_l w_{kl}^{x^t} \frac{\partial S_l^{t-1}}{\partial w_{ij}^n} + \delta_{ki}\delta_{nx^t} S_j^{t-1}), \quad \forall\ i, j, n, k, t.$$

(Initially, set: $\frac{\partial S_k^0}{\partial w_{ij}^n} = 0, \qquad \forall\ i, j, n, k.$)

The stopping criterion for training is that the network error for every training string is below a certain threshold. In all of our experiments, this threshold is set to 0.02.

Appendix B
Detailed Results and Analysis of the Initial Network Structure Learning the Reber Grammar

The Reber grammar with equiprobable transition splits is shown in Figure 4.9(a). A data set of 200 strings is generated randomly according to the grammar. Five networks with different random weight initializations are trained on the data set. All networks have 3 state units. Both the number of input and output units are 7, the same as the alphabet size. For each run, learning is stopped after 200 epochs of data presentation. Table B.1 shows the results of the 5 runs. A successful run is defined to be that the network has found a solution after 200 epochs where it produces high outputs (> 0.3) for correct successors and low outputs (< 0.3) for non-successors for all the training data. As can be seen from the table, 4 out the 5 runs are successful runs. The number of states derived from a network for the successful runs are also shown. Since the output prediction can be drastically different for different inputs, with the same "state" unit values, we define a "network state" to be a configuration of *both* the "state" units *and* the input units. Two such configurations are considered to be the same "network state" if they both predict the same successors with high output values and go to the same next "state" units.

network number	*successful run?*	number of *"states"*
1	Yes	7
2	Yes	9
3	No	-
4	Yes	8
5	Yes	8

Table B.1: Initial experimental results on learning the Reber grammar

Although there can be two different configurations of the "state" unit and input values belonging to the same network state and they both give high output values on the same units and low values on the others, the exact values they produce can differ. These are

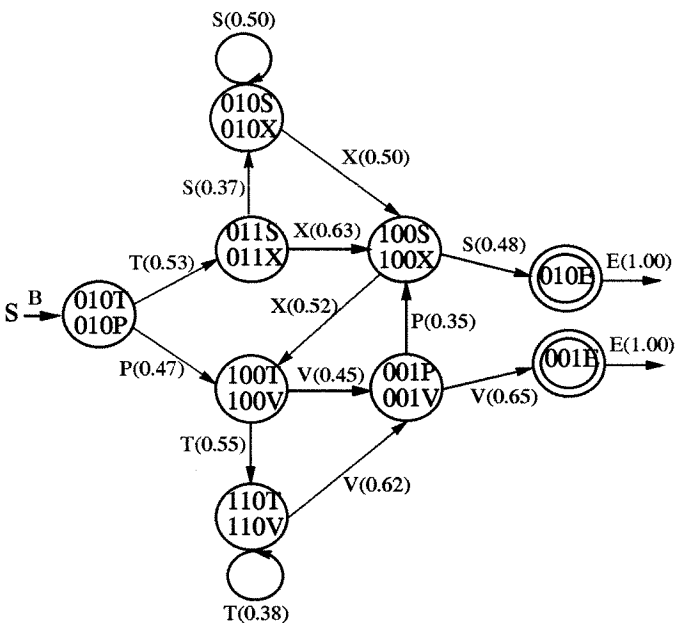


Figure B.1: An extracted state machine from a trained network.

values we would like to interpret as probability estimates, and thus they are desired to be equal for any configurations belonging to the same network state. Our experiments have shown however, that this goal is very difficult to reach. Prolonged training does not make the network outputs for each configuration closer to ideal.

Figure B.1 shows an example of extracted state machine from one of the successfully trained networks. Each state is labeled by a binary number denoting the state unit values and one or more letters denoting the input values: each letter along with the binary number is a configuration of that network state. Depending on how many possible ways there are to reach a network state u_i from some other network state u_j, and how many configurations there are in u_j, a transition from network state u_i can have several different probability estimates produced by output units. For simplicity, only one set of normalized transition probabilities which are closest to the ideal numbers are shown in the figure for each state. It should be noted that most of the other sets of probabilities not shown are far worse than the ones shown. As can be seen from the figure, these numbers are far from the true probabilities shown in Figure 4.9(a).

The reason for this behavior can be explained as follows: For networks with n state units, assume that it has found a minimum configuration of network states to represent the grammar, i.e., 6 network states and 11 transitions. As mentioned above, each transi-

tion can correspond to different paths from different network configurations of the same previous network state. To produce the ideal probabilities, each such path defines a set of equations mapping the configuration of a network state to the output probabilities. There is one equation for each output unit in any of the sets of equations. From our example above, we know that in fact, there are 2 configurations in all of the network states, except the ending states (this is true for all the machines extracted), thus we have in fact at least $11 + 10 = 21$ equations for each output unit to satisfy. From the network architecture, there are only $n + 7 + 1 = n + 8$ weights connecting the state units, input units and a bias unit to every output unit, $n = 3$ in our case. It is then very unlikely that the network can use the limited set of 11 adjustable weights to satisfy all 21 equations. This is true for all of the 7 output units. Also can be seen from Table B.1, none of the networks in fact found the minimal machine — a larger machine results in more transitions, which in turn results in more equations for the outputs to satisfy.

The conclusion is that the network is not powerful enough to fully represent the probabilities. One way to compensate for this is to add more units, but from the above analysis, we need a lot more units, much more than the number of states in the true machine. An alternative way is then to add an intermediate layer between the output and the state units, and to remove the possibility of having more than 1 configuration for a network state. Section 4.6 describes an improved network structure to accomplish the goals.

Appendix C
Proof of Lemma 4.1 and 4.2

Lemma

Let p be the probability of event X happening for a probabilistic test. Let n be the number of tests, and k be the number of times event X happened. Then

$$E\left(\frac{k}{n}\right) = p,$$

$$Var\left(\frac{k}{n}\right) = \frac{p(1-p)}{n}.$$

Proof:

Let i be an index over the tests, $1 \leq i \leq n$.

Let X_i be the event that X happens for test i,

then $P(X_i) = p$ by definition.

Let $I_{X_i} = \begin{cases} 1 & \text{if event } X \text{ happens for test } i \\ 0 & \text{otherwise,} \end{cases}$

then $E[I_{X_i}] = p$.

By definition, $k = \sum_{i=1}^{n} I_{X_i}$, we have:

$$E\left[\frac{k}{n}\right] = \frac{1}{n}E\left[\sum_{i=1}^{n} I_{X_i}\right] = \frac{1}{n}\sum_{i=1}^{n} E[I_{X_i}] = \frac{np}{n} = p.$$

Note that $I_{X_i}^2 = I_{X_i}$, so

$$Var[I_{X_i}] = E[I_{X_i}^2] - (E[I_{X_i}])^2 = E[I_{X_i}] - p^2 = p - p^2 = p(1-p).$$

Since the n events are independent,

$$Var\left[\frac{k}{n}\right] = \frac{1}{n^2}Var[k] = \frac{1}{n^2}Var\left[\sum_{i=1}^{n} I_{X_i}\right] = \frac{1}{n^2}\sum_{i=1}^{n} Var[I_{X_i}] = \frac{p(1-p)n}{n^2} = \frac{p(1-p)}{n}.$$

Q.E.D.

Lemma

Let p be the probability of event X happening for a probabilistic test T. Let m be the number of tests, and k_m be the number of times event X happened. Let n be the size of a subset of the above tests ($n \leq m$), and k_n be the number of times event X happened in that subset ($k_n \leq k_m$). Then

$$Var\left[\left(\frac{k_m}{m} - \frac{k_n}{n}\right)\right] = p(1-p)\left(\frac{1}{n} - \frac{1}{m}\right).$$

Proof:

From the previous Lemma, we have:

$$E(k_n) = np, \quad E(k_m) = mp, \quad Var(k_n) = np(1-p),$$

$$Var\left(\frac{k_m}{m}\right) = \frac{p(1-p)}{m}, \quad Var\left(\frac{k_n}{n}\right) = \frac{p(1-p)}{n},$$

$$E\left[\left(\frac{k_m}{m} - \frac{k_n}{n}\right)\right] = p - p = 0.$$

Let $k_{m-n} = k_m - k_n$, the number of times X happened in the m tests excluding the n sub-samples, we have:

$$\begin{aligned}
Var\left[\left(\frac{k_m}{m} - \frac{k_n}{n}\right)\right] &= E\left[\left(\frac{k_m}{m} - \frac{k_n}{n}\right)^2\right] - \left(E\left[\left(\frac{k_m}{m} - \frac{k_n}{n}\right)\right]\right)^2 \\
&= E\left[\left(\left(\frac{k_m}{m} - p\right) - \left(\frac{k_n}{n} - p\right)\right)^2\right] - 0 \\
&= E\left[\left(\frac{k_m}{m} - p\right)^2\right] + E\left[\left(\frac{k_n}{n} - p\right)^2\right] - 2E\left[\left(\frac{k_m}{m} - p\right) \cdot \left(\frac{k_n}{n} - p\right)\right] \\
&= Var\left(\frac{k_m}{m}\right) + Var\left(\frac{k_n}{n}\right) - \frac{2}{mn} E\left[(k_n - np)(k_m - mp)\right].
\end{aligned}$$

Since k_n and k_{m-n} are independent,

$$\begin{aligned}
E\left[(k_n - np)(k_m - mp)\right] &= E\left[(k_n - np)(k_n - np + k_{m-n} - (m-n)p)\right] \\
&= E\left[(k_n - np)^2\right] + E\left[(k_n - np)(k_{m-n} - (m-n)p)\right] \\
&= Var(k_n) + E\left[(k_n - np)\right] \cdot E\left[(k_{m-n} - (m-n)p)\right] \\
&= Var(k_n) + 0 = np(1-p).
\end{aligned}$$

So

$$\begin{aligned} Var\left[\left(\frac{k_m}{m} - \frac{k_n}{n}\right)\right] &= Var\left(\frac{k_m}{m}\right) + Var\left(\frac{k_n}{n}\right) - \frac{2}{mn} \cdot np(1-p) \\ &= \frac{p(1-p)}{m} + \frac{p(1-p)}{n} - \frac{2p(1-p)}{m} \\ &= p(1-p)\left(\frac{1}{n} - \frac{1}{m}\right). \end{aligned}$$

Q.E.D.

Appendix D
Proof of Theorem 4.2

Theorem *Given an infinite data set D_I generated by a certain finite state probabilistic automaton $M =< \Sigma, U, u_0, \delta, P, F >$, and let R_I be the set of prefix strings contained in D_I. If M is the smallest and unique machine that generates D_I, then it satisfies:*

1. $\forall u_i \in U$, $\exists r_i \in R_I$, *such that* r_i *leads to* u_i.

2. $\forall u_i, u_j \in U$, $u_i \neq u_j$, *the following is true:*
   $\forall r_i$ *that leads to* u_i, *and* $\forall r_j$ *that leads to* u_j, $\exists$ *suffix* $w \in \Sigma^*$, $x \in \Sigma$, *such that* $r_i w, r_j w \in R_I$, $r_i w$ *leads to state* u_{i_w}, $r_j w$ *leads to state* u_{j_w}, $u_{i_w} \neq u_{j_w}$, *and* $P(x|u_{i_w}) \neq P(x|u_{j_w})$.

Proof by contradiction:

Given that M is the smallest and unique machine that generates D_I,

1. If condition 1 is not satisfied, then $\exists u_i \in U$ that no prefix strings in R_I leads into. Then we can construct a new machine by eliminating u_i from M while keeping everything else the same. The new machine is a smaller machine than M, and can generate D_I exactly the same way as M — a contradiction.

2. If condition 2 is not satisfied, then $\exists u_i \in U$, $u_j \in U, u_i \neq u_j$, that satisfies the following:
   For $\forall r_i$ that leads to u_i, $\forall r_j$ that leads to u_j, and $\forall$ suffix $w \in \Sigma^*$ such that $r_i w, r_j w \in R_I$, and $r_i w$ leads to state u_{i_w}, $r_j w$ leads to state u_{j_w}, the equality $P(x|u_{i_w}) = P(x|u_{j_w})$ holds for $\forall x \in \Sigma$. I.e., u_i and u_j behaves exactly the same for all prefix strings leading to them. Thus we can construct a new machine by eliminating state u_j and redirecting all transitions going into u_j to u_i instead. Keeping everything else the same, the new machine is one state smaller than M, and can generate D_I exactly the same way as M — a contradiction.

Q.E.D.

Appendix E
Proof of Theorem 4.3

Theorem

Given data D that satisfies Condition 4.3 for a minimum machine M, the smallest extracted machine M' that passes Criterion 4.2 satisfies:

$$M' \stackrel{*}{=} M,$$

where "$\stackrel{*}{=}$" stands for "statistically equivalent" defined as in Criterion 4.2.

Proof:

Let $M =< \Sigma, U, u_0, \delta, P, F >$, $M' =< \Sigma, U', u'_0, \delta', P', F' >$.

- If $|M'| > |M|$, then from Condition 4.3-5, M also passes Criterion 4.2. This leads to a contradiction to the fact that M' is the smallest of such machines.

- If $|M'| \leq |M|$,

  - Consider $\forall u_i, u_j \in U$, such that $u_i \neq u_j$, i.e., $r_i^s \neq r_j^s$.
    From Condition 4.3-1: $r_i^s, r_j^s \in R$, thus $\exists u'_i, u'_j \in U'$, such that r_i^s leads to u'_i, r_j^s leads to u'_j.
    From Condition 4.3-2 and condition 2 in Theorem 4.2: $\exists$ suffix $w \in \Sigma^*, x \in \Sigma$, such that $r_i^s w, r_j^s w \in R$, thus $\exists u'_{i_w}, u'_{j_w} \in U'$, such that $r_i^s w$ leads to state u'_{i_w}, $r_j^s w$ leads to state u'_{j_w}.
    Also from Condition 4.3-2, $|\hat{p}_r(x|r_i^s w) - \hat{p}_r(x|r_j^s w)| > \sqrt{\frac{\beta}{N}}$,
    By Criterion 4.2: $P'(x|u'_{i_w}) \stackrel{*}{=} \hat{p}_r(x|r_i^s w)$, and $P'(x|u'_{j_w}) \stackrel{*}{=} \hat{p}_r(x|r_j^s w)$,
    thus $|P'(x|u'_{i_w}) - \hat{p}_r(x|r_i^s w)| \leq \sqrt{\beta\left(\frac{1}{n_{iw}} - \frac{1}{m_{iw}}\right) \cdot P'(x|u'_{i_w})(1 - P'(x|u'_{i_w})}$
    $< \sqrt{\beta\left(\frac{1}{n_{iw}}\right) \cdot 0.5(1 - 0.5)} \leq \frac{1}{2}\sqrt{\frac{\beta}{N}}$,
    where m_{iw} is the number of instances of state u'_{i_w} from D, and $n_{iw} \leq N$ is the number of instances of prefix $r_i^s w$ from R. Similarly,
    $|P'(x|u'_{j_w}) - \hat{p}_r(x|r_j^s w)| < \frac{1}{2}\sqrt{\frac{\beta}{N}}$.
    Therefore, $|P'(x|u'_{i_w}) - P'(x|u'_{j_w})|$
    $= |[P'(x|u'_{i_w}) - \hat{p}_r(x|r_i^s w)] - [P'(x|u'_{j_w}) - \hat{p}_r(x|r_j^s w)] + [\hat{p}_r(x|r_i^s w) - \hat{p}_r(x|r_j^s w)]|$
    $\geq |\hat{p}_r(x|r_i^s w) - \hat{p}_r(x|r_j^s w)| - |[P'(x|u'_{i_w}) - \hat{p}_r(x|r_i^s w)] - [P'(x|u'_{j_w}) - \hat{p}_r(x|r_j^s w)]|$

$\geq |\hat{p}_r(x|r_i^s w) - \hat{p}_r(x|r_j^s w)| - [|P'(x|u'_{i_w}) - \hat{p}_r(x|r_i^s w)| + |P'(x|u'_{j_w}) - \hat{p}_r(x|r_j^s w)|]$
$> \sqrt{\frac{\beta}{N}} - \frac{1}{2}\sqrt{\frac{\beta}{N}} - \frac{1}{2}\sqrt{\frac{\beta}{N}}$
$= 0.$

Thus $P'(x|u'_{i_w}) \neq P'(x|u'_{j_w})$, and $u'_{i_w} \neq u'_{j_w}$, so $u'_i \neq u'_j$.

Thus the number of states in U' is at least the same as that of U:

$|M'| \geq |M|$.

Therefore, $|M'| = |M|$, and the states in U has a one to one correspondence to the states in U': $u_i \overset{1-1}{\longleftrightarrow} u'_i$.

- Consider $\forall u_i \in U, x \in \Sigma$, such that $P(x|u_i) \neq 0$. Let $u_k = \delta(u_i, x)$, and take $\forall u_j \neq u_k$. From Condition 4.3-1 and Condition 4.3-3: $r_i^s x, r_j^s \in R$, thus $\exists u'_{i'_x}, u'_j \in U'$, such that $r_i^s x$ leads to $u'_{i'_x}$, i.e., $\delta'(u'_i, x) = u'_{i'_x}$, and r_j^s leads to u'_j. From Condition 4.3-4 and condition 2 in Theorem 4.2: $\exists$ suffix $w \in \Sigma^*$, such that $r_i^s xw, r_j^s w \in R$. Thus $\exists u'_{i'_{xw}}, u'_{j'_w} \in U'$, such that $r_i^s xw$ leads to state $u'_{i'_{xw}}$, $r_j^s w$ leads to state $u'_{j'_w}$.
  Also from Condition 4.3-4, $\exists y \in \Sigma$ such that $|\hat{p}_r(y|r_i^s xw) - \hat{p}_r(y|r_j^s w)| > \sqrt{\frac{\beta}{N}}$.
  By Criterion 4.2: $P'(y|u'_{i'_{xw}}) \overset{*}{=} \hat{p}_r(y|r_i^s xw)$, and $P'(y|u'_{j'_w}) \overset{*}{=} \hat{p}_r(y|r_j^s w)$.
  Similar to the derivation above, we have:
  $P'(y|u'_{i'_{xw}}) \neq P'(y|u'_{j'_w})$, and $u'_{i'_{xw}} \neq u'_{j'_w}$.
  Since $u'_{i'_{xw}}$ and $u'_{j'_w}$ are arrived at by processing the same suffix string w starting from states $u'_{i'_x}$ and u'_j, respectively, we have:
  $\delta'(u'_i, x) = u'_{i'_x} \neq u'_j$ holds for $\forall j$, such that $u_j \neq u_k (= \delta(u_i, x))$. Therefore, $\delta'(u'_i, x) = u'_k$.

In conclusion, we proved that

- $|M'| = |M|$.
- For $\forall u_i \in U$, $\exists u'_i \in U'$ such that r_i^s leads to u_i in M, and u'_i in M'. That is, $u_i \overset{1-1}{\longleftrightarrow} u'_i$.
- For $\forall \delta(u_i, x) = u_k \in U$, M' has $\delta'(u'_i, x) = u'_k \in U'$.

That completes the proof of the fact that the structures of M and M' are the same. For the probability part, the statistical equivalence is automatically satisfied by the verification of Criterion 4.2. *Q.E.D.*

References

[ADO91] N. Alon, A. K. Dewdney, and T. J. Ott, "Efficient simulation of finite automata by neural nets," *Journal of the Association for Computing Machinery*, 38(2):495–514, 1991.

[Ang72] D. Angluin, "Inference of reversible languages," *Journal of the Association for Computing Machinery*, 29(3):741–765, 1972.

[Ang78] D. Angluin, "On the complexity of minimum inference of regular sets," *Information and Control*, 39:337–350, 1978.

[Arb69] M. A. Arbib, *Theories of abstract automata*, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1969.

[AS83] D. Angluin and C. H. Smith, "Inductive inference: theory and methods," *ACM Computing Survey*, 15(3):237, 1983.

[BC91] B. Bartell and G. W. Cottrell, "A model of symbol grounding in a temporal environment," In *Proceedings of International Joint Conference on Neural Networks*, volume I, pages 805–810, Seattle, WA, 1991.

[CA91] J. Connor and L. Atlas, "Recurrent neural networks and time series prediction," In *Proceedings of International Joint Conference on Neural Networks*, volume I, pages 301–306, Seattle, WA, 1991.

[CG87] G. Carpenter and S. Grossberg, "A massively parallel architecture for a self-organizing neural pattern recognition machine," *Computer Vision, Graphics and Image Processing*, 37:54-115, 1987.

[Cho59] N. Chomsky, "On certain formal properties of grammars," *Information and Control*, 1(2):137–167, 1959.

[CL89] J. Carroll and D. Long, *Theory of Finite Automata*, Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[CSSM89] A. Cleeremans, D. Servan-Schreiber, and J. L. McClelland, "Finite state automata and simple recurrent networks," *Neural Computation*, 1:372–381, 1989.

[DGS93] S. Das, C. L. Giles, and G. Z. Sun, "Using prior knowledge in an NNPDA to learn context-free languages," In S. J. Hanson, J. D. Cowan, and C. L. Giles, editors, *Advances in Neural Information Processing Systems 5*, pages 65–72. Morgan Kaufmann, San Mateo, CA, 1993.

[DM94] S. Das and M. C. Mozer, "A unified gradient-descent/clustering architecture for finite state machine induction," In J. D. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems 6*. Morgan Kaufmann, San Mateo, CA, 1994.

[dVP91] B. de Vries and J. C. Principe, "A theory for neural networks with time delays," In R. P. Lippmann, J. E. Moody, and D. S. Touretzky, editors, *Advances in Neural Information Processing Systems 3*, pages 162–168. Morgan Kaufmann, San Mateo, CA, 1991.

[EJM90] A. El-Jaroudi and M. Makhoul, "A new error criterion for posterior probability estimation with neural nets," In *Proceedings of International Joint Conference on Neural Networks*, volume III, pages 185–192, San Diego, CA, 1990.

[Elm90] J. L. Elman, "Finding structure in time," *Cognitive Science*, 14:179–211, 1990.

[Elm91] J. L. Elman, "Distributed representations, simple recurrent networks, and grammatical structure," *Machine Learning*, 7(2/3):195–225, 1991.

[Fah91] S. E. Fahlman, "The recurrent cascade-correlation architecture," In J. E. Moody R. P. Lippmann and D. S. Touretzky, editors, *Advances in Neural Information Processing Systems 3*, pages 190–196. Morgan Kaufmann, San Mateo, CA, 1991.

[FGS92] P. Frasconi, M. Gori, and G. Soda, "Local feedback multilayered networks," *Neural Computation*, 4:120–130, 1992.

[Fu82] K. S. Fu, *Syntactic Pattern Recognition and Applications*, Prentice-Hall, Englewood Cliffs, New Jersey, 1982.

[GMC+92a] C. L. Giles, C. B. Miller, D. Chen, H. H. Chen, G. Z. Sun, and Y. C. Lee, "Learning and extracting finite state automata with second-order recurrent neural networks," *Neural Computation*, 4(3):393–405, 1992.

[GMC+92b] C. L. Giles, C. B. Miller, D. Chen, G. Z. Sun, H. H. Chen, and Y. C. Lee, "Extracting and learning an unknown grammar with recurrent neural networks," In J. E. Moody, S. J. Hanson, and R. P. Lippmann, editors, *Advances in Neural Information Processing Systems 4*, pages 317–324. Morgan Kaufmann, San Mateo, CA, 1992.

[Gol72] E. M. Gold, "System identification via state characterization," *Automatica*, 8:621–636, 1972.

[Gol78] E. M. Gold, "Complexity of automaton identification from given data," *Information and Control*, 37:302–320, 1978.

[GSC+90] C. L. Giles, G. Z. Sun, H. H. Chen, Y. C. Lee, and D. Chen, "Higher order recurrent networks and grammatical inference," In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 380–387. Morgan Kaufmann, San Mateo, CA, 1990.

[GZ94] R. Goodman and Z. Zeng, "A learning algorithm for multi-layer perceptrons with hard-limiting threshold units," In *Proceedings of IEEE International Conference on Neural Networks*, Orlando, FL, 1994, to appear.

[Hea87] T. Head, "Formal language theory and dna: An analysis of the generative capacity of specific recombinant behaviors," *Bulletin of Mathematical Biology*, 49(6):737–759, 1987.

[HLvH91] A. V. M. Herz, Z. Li, and J. Leo van Hemmen, "Statistical mechanics of temporal association in neural networks with delayed interactions," In R. P. Lippmann, J. E. Moody, and D. S. Touretzky, editors, *Advances in Neural Information Processing Systems 3*, pages 176–182. Morgan Kaufmann, San Mateo, CA, 1991.

[Hop82] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proceedings of the National Academy of Sciences*, 79:2554–2558, 1982.

[HP90] J. Hampshire and B. Pearlmutter, "Equivalence proofs for multi-layer perceptron classifiers and the Bayesian discriminant function," In D. Touretzky et al, editor, *Proceedings of the 1990 Connectionist Models Summer School*, pages 159–172, San Mateo, CA, 1990. Morgan Kaufmann.

[HU79] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, Reading MA, 1979.

[Jor86a] M. I. Jordan, "Attractor dynamics and parallelism in a connectionist sequential machine," In *Proceedings of the 1986 Cognitive Science Conference*, pages 531–546, Lawrence Erlbaum, 1986.

[Jor86b] M. I. Jordan, "Serial order: A parallel distributed processing approach," Technical Report 8604, University of California at San Diego, Institute for Cognitive Science, 1986.

[Koh88] T. Kohonen, *Self-Organization and Associative Memory*, Springer-Verlag, Berlin, second edition, 1988.

[KPR91] A. Kuh, T. Petsche, and R. L. Rivest, "Learning time-varying concepts," In R. P. Lippmann, J. E. Moody, and D. S. Touretzky, editors, *Advances in Neural Information Processing Systems 3*, pages 183–189. Morgan Kaufmann, San Mateo, CA, 1991.

[KS88] M. Kudo and M. Shimbo, "Efficient regular grammatical inference techniques by the use of partial similarities and their logical relationships," *Pattern Recognition*, 21(4):401–409, 1988.

[LC91] R. R. Leighton and B. C. Conrath, "The autoregressive backpropagation algorithm," In *Proceedings of International Joint Conference on Neural Networks*, volume II, pages 369–377, Seattle, WA, 1991.

[Lev91] E. Levin, "Modeling time varying systems using hidden control neural architecture," In R. P. Lippmann, J. E. Moody, and D. S. Touretzky, editors, *Advances in Neural Information Processing Systems 3*, pages 147–154. Morgan Kaufmann, San Mateo, CA, 1991.

[Lip87] R. P. Lippmann, "An introduction to computing with neural nets," *IEEE ASSP Magazine*, pages 4–20, April 1987.

[McQ67] J. B. McQueen, "Some methods of classification and analysis of multivariate observations," In *Proceedings of Fifth Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1967.

[Mil93] J. Miller, *Building Probabilistic Models from Databases*, Ph.D. thesis, California Institute of Technology, 1993.

[MP69] M. Minsky and S. Papert, *Perceptrons*, MIT Press, Cambridge, MA, 1969.

[Mug90] S. Muggleton, *Grammatical Induction Theory*, Addison-Wesley, Turing Institute Press, 1990.

[Pea89] B. A. Pearlmutter, "Learning state space trajectories in recurrent neural networks," *Neural Computation*, 1:263–269, 1989.

[Pol91] J. B. Pollack, "The induction of dynamical recognizers," *Machine Learning*, 7(2/3):227–252, 1991.

[Rab89] L. R. Rabiner, "A tutorial on hidden Markov models and selected applications in speech recognition," *Proceedings of the IEEE*, 77(2):257–286, 1989.

[Reb69] A. S. Reber, "Implicit learning of artificial grammars," *Journal of Verbal Learning and Verbal Behavior*, 6:855–863, 1969.

[RHW86] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Learning internal representations by error propagation*, chapter 8, MIT Press, Cambridge, MA, 1986.

[RMtPRG86] D. E. Rumelhart, J. L. McClelland, and the PDP Research Group, *Parallel Distributed Processing*, MIT Press, Cambridge, MA, 1986.

[SBJ91] S. Santini, A. Del Bimbo, and R. Jain, "An algorithm for training neural networks with arbitrary feedback structure," Technical Report DSI 10/91, Dipartimento di Sistemi e Informatica, Università di Firenze, 1991.

[SCG+90] G. Z. Sun, H. H. Chen, C. L. Giles, Y. C. Lee, and D. Chen, "Connectionist pushdown automata that learn context-free grammars," In *Proceedings of the International Joint Conference on Neural Networks*, volume I, page 577, Washington D. C., 1990.

[Sch92] J. Schmidhuber, "Learning complex, extended sequences using the principle of history compression," *Neural Computation*, 4:234–242, 1992.

[Sea92] D. B. Searls, "The linguistics of DNA," *American Scientist*, 80:579–591, 1992.

[SLF88] S. Solla, E. Levin, and M. Fleisher, "Accelerated learning in layered neural networks," *Complex Systems*, 2:625–640, 1988.

[SO93] A. Stolcke and S. Omohundro, "Hidden Markov model induction by Bayesian model merging," In S. J. Hanson, J. D. Cowan, and C. L. Giles, editors, *Advances in Neural Information Processing Systems 5*, pages 11–18. Morgan Kaufmann, San Mateo, CA, 1993.

[Sør91] E. Sørheim, "ART2/BP architecture for adaptive estimation of dynamic processes," In R. P. Lippmann, J. E. Moody, and D. S. Touretzky, editors, *Advances in Neural Information Processing Systems 3*, pages 169–175. Morgan Kaufmann, San Mateo, CA, 1991.

[SSCM91] D. Servan-Schreiber, A. Cleeremans, and J. L. McClelland, "Graded state machines: the representation of temporal contingencies in simple recurrent networks," *Machine Learning*, 7(2/3):161–193, 1991.

[SZ89] A. W. Smith and D. Zipser, "Encoding sequential structure: experience with the real-time recurrent learning algorithm," In *Proceedings of International Joint Conference on Neural Networks*, volume I, pages 645–648, Washington, DC, 1989.

[Tom82] M. Tomita, "Dynamic construction of finite-state automata from examples using hill-climbing," In *Proceedings of the Fourth International Cognitive Science Conference*, pages 105-108, Ann Arbor, Michigan, 1982. Lawrence Erlbaum Assoc.

[Wid62] B. Widrow, "Generalization and information storage in networks of ADALINE neurons," In G. T. Yovitts, editor, *Self-Organizing Systems*. Spartan Books, Washington DC, 1962.

[WK92] R. L. Watrous and G. M. Kuhn, "Induction of finite-state languages using second-order recurrent networks," *Neural Computation*, 4(3):406–414, 1992.

[WZ89] R. J. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," *Neural Computation*, 1(2):270–280, 1989.

[Zen94] Z. Zeng, "Discrete recurrent neural networks for probabilistic grammar learning," to be submitted to the *IEEE Transactions on Neural Networks*, 1994.

[ZGS93] Z. Zeng, R. Goodman, and P. Smyth, "Learning finite state machines with self-clustering recurrent networks," *Neural Computation*, 5(6):976–990, 1993.

[ZGS94] Z. Zeng, R. Goodman, and P. Smyth, "Discrete recurrent neural networks for grammatical inference," *IEEE Transactions on Neural Networks*, 5(2):320–330, March 1994.