National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services Branch

Direction des acquisitions et des services bibliographiques

395 Wellington Street
Ottawa, Ontario
K1A 0N4

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

University of Alberta

A Theory of Grammatical Induction in the Connectionist Paradigm

by

Stefan Charles Kremer

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Department of Computing Science

Edmonton, Alberta

Spring 1996

National Library of Canada

Acquisitions and Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada

Direction des acquisitions et des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-612-10604-7

Canada

to Janis

Abstract

This dissertation shows that the tractability and efficiency of training particular connectionist networks to implement certain classes of grammars can be formally determined by applying principles and ideas that have been explored in the symbolic grammatical induction paradigm. Furthermore, this formal analysis also allows networks to be tailored to efficiently solve specific grammatical induction problems. Had the formal work that is reported in this dissertation been done earlier, it is possible that connectionist researchers would have been able to take a formal, rather than empirical, approach to understanding the computational power of their nets for grammatical induction. As well, our formal approach could have been applied to understand and develop techniques to functionally increase the power of connectionist grammar induction systems. Instead, these techniques are currently being discovered empirically. This dissertation, by considering classical work done over the past three decades, gives a formal grounding to these empirically discovered methods. In doing so, it also suggests a rationale for making the design decisions which define every connectionist grammar induction system. This allows new networks to be better suited to the problems to which they will be applied. Finally, the dissertation provides insights into applying other refinement techniques that connectionist researchers have yet to consider.

Acknowledgement

I would like to thank my supervisors, Dr. Renée E. Elio and Dr. Michael R.W. Dawson, for their suggestions and encouragement throughout the development of this thesis.

Thanks are also due to Dr. William W. Armstrong, Dr. C. Lee Giles, Dr. Terrance M. Nearey, and Dr. Peter G. Van Beek whose diverse comments led to many improvements in this work.

Finally, I would like to thank my family: my parents, whose love and guidance started me in the right direction; and my wife, Janis, whose patience, support and love gave me the strength to keep going.

Table of Contents

Chapter I: Introduction 1

1.1 THE THESIS OF THIS DISSERTATION 1

1.2 OUTLINE 2

1.3 IMPLICATIONS 4

Chapter II: Connectionist Networks for Grammar Induction 6

2.1 INTRODUCTION 6

2.2 TERMINOLOGY AND MATHEMATICAL PRELIMINARIES 7

2.2.1 Spatio-temporal Connectionist Networks (STCNs) 7

2.2.2 Search States and Machine States 8

2.2.3 A Note About Notation 8

2.2.4 Input Dimensions 10

2.2.5 Spatio-Temporal Connectionist Network Memory 11

2.2.6 Output, Teaching, and Error 12

2.3 FEATURES OF A GOOD TAXONOMY 12

2.4 EXISTING TAXONOMIES 13

2.4.1 Mozer (1993) 13

Memory Content 16

Memory Form 17

Memory Adaptability 18

Evaluation of Mozer's Taxonomy 19

2.4.2 Horne and Giles (1994) 22

Evaluation of Horne and Giles's Taxonomy 24

2.4.3 Tsoi and Back (1994) 24

Evaluation of Tsoi and Back's Taxonomy 26

2.4.4 Observations 27

2.5 A NEW TAXONOMY FOR SPATIO-TEMPORAL CONNECTIONIST NETWORKS 27

2.5.1 Computing the State Vector 32

Window In Time (WIT) Memory 33

Connectionist Infinite Impulse Response Filter (CIIR) Memory 35

Single-Layer First-Order Context Computation (SLFOCC) Memory 37

Single-Layer Second-Order Context Computation (SLSOCC) Memory 38

Connectionist Pushdown Automaton (CPA) Memory with Continuous Stack 41

Connectionist Turing Machine (CTM) Memory 44

Locally Recurrent State and Input (LRSI) Memory 45

2.5.2 Computing the Output Vector 47

Zero Layer (0-Layer) 47

One Layer (1-Layer) 48

Two Layer (2-Layer) 48

2.5.3 Computing the Weight Change 49

Full Gradient Descent (FGD) 51

Teacher Forcing (TF) 52

Truncated Gradient Descent (TGD) 54

Auto-Associative Gradient Descent (AAGD) 54

Stack Learning (SL) 56

Other Work Describing Weight Change Functions 57

2.5.4 Computing the Change in State Vector Size 57

Manual Architecture Changes (MAC) 57
Automatically Incrementing Nodes (AIN) 57
2.6 SPECIFIC DESIGNS 59
2.7 CONCLUSIONS 63

Chapter III: The Problems of Grammatical Induction 65
3.1 INTRODUCTION 65
3.2 GRAMMATICAL INDUCTION AS SEARCH 66
3.3 LANGUAGES, GRAMMARS AND MACHINES 68
3.4 THE CHOMSKY HIERARCHY 71
3.4.1 Regular Grammars, Regular Sets, Finite State Automata 71
3.4.2 Context Free Grammars, Context Free Languages, Pushdown Automata 73
3.4.3 Unrestricted Grammars, Recursively Enumerable Languages, Turing Machines 75
3.4.4 Other Formalisms 77
3.5 SPATIO-TEMPORAL CONNECTIONIST NETWORKS AND THE CHOMSKY HIERARCHY 79
3.5.1 Finite State Automaton Equivalence 79
3.5.2 Unbounded Memory 80
3.5.3 Implementing a Stack in a Connectionist Network 82
3.5.4 Turing Machine Equivalence 83
3.5.5 Hypothesis Spaces in Connectionist vs. Symbolic Systems . . 84
3.5.6 Training Networks and Grammatical Induction 85
3.6 THE GRAMMAR INDUCTION PROBLEM 86
3.6.1 The Importance of Grammatical Induction 86
3.6.2 The Difficulty of Inducing Chomsky Grammars 86

3.6.3 Tractable Learning 88

3.6.4 Reducing and Ordering in Grammatical Induction 89

3.7 CONCLUSIONS 92

Chapter IV: A Priori Knowledge and the Selection of Appropriate State and Output Functions 94

4.1 INTRODUCTION 94

4.2 WINDOW-IN-TIME (WIT) MEMORIES 96

4.2.1 WIT Machines 96

4.2.2 WIT Languages 98

4.3 CONNECTIONIST INFINITE IMPULSE RESPONSE FILTER (CIIR) Memories 101

4.4 SINGLE-LAYER FIRST-ORDER CONTEXT COMPUTATION (SLFOCC) MEMORIES 102

4.4.1 SLFOCC Memories Cannot Compute Arbitrary State Functions 102

4.4.2 SLFOCC Memories Can Represent Arbitrary Finite State Automata (Moore Machines) 104

The Parity Problem Revisited 105

Arbitrary Automata 106

Automaton Complexity 112

4.4.3 Turing Machine Equivalence 114

4.5 SINGLE-LAYER SECOND-ORDER CONTEXT COMPUTATION . 117

4.5.1 Single-Layer Second Order Context Computation Memories Can Encode Any State Transition Function without State Splitting 117

4.5.2 Single-Layer Second-Order Context Computation Memories Cannot Encode Every State Transition Function with Binary Encoding . . . 118
4.5.3 Turing Machine Equivalence . . . 123
4.6 CONNECTIONIST PUSH-DOWN AUTOMATA (CPA) MEMORIES . . . 125
4.6.1 Stack Limitations . . . 125
4.6.2 Controller Limitations . . . 126
4.6.3 Turing Equivalence . . . 129
4.7 CONNECTIONIST TURING MACHINE (CTM) MEMORIES . . . 130
4.8 LOCALLY RECURRENT STATE AND INPUT . . . 131
4.8.1 Constant Input . . . 131
4.8.2 Oscillating Input . . . 136
4.8.3 Arbitrary Cycles . . . 139
4.9 OUTPUT FUNCTIONS . . . 143
4.9.1 0-Layer Output Function . . . 143
4.9.2 1-Layer Output Function . . . 144
4.9.3 2-Layer Output Function . . . 147
4.10 SUMMARY . . . 147

Chapter V: Fixing and Initializing Weights . . . 151
5.1 INTRODUCTION . . . 151
5.2 FIXING AND INITIALIZING CONNECTION WEIGHTS TO RESTRICT AND ORDER HYPOTHESIS SPACES . . . 152
5.3 FIXING WEIGHTS TO LIMIT HYPOTHESIS SPACE . . . 153
5.4 CHOOSING INITIAL WEIGHTS TO ORDER HYPOTHESIS SPACE . . . 155

5.5 CHOOSING INITIAL WEIGHTS TO RESTRICT HYPOTHESIS SPACE 157
5.6 ENCODING A PRIORI KNOWLEDGE (WIRING NETWORKS) . . 160
5.6.1 Single Layer First-Order Context Computation Memories . . 161
5.6.2 Single-Layer Second-Order Context Computation Memories 164
5.6.3 Locally Recurrent State and Input Memories 165
5.6.4 Other Types of A Priori Knowledge 166
5.7 CONCLUSIONS 167
Chapter VI: Using a Posteriori Knowledge to Find Target Grammars 168
6.1 INTRODUCTION 168
6.2 INPUT ORDERING 169
6.2.1 Previous Results 170
6.2.2 Input Ordering in STCN Training 173
Lengthening Input Data 173
Alphabetical Input Data 17
Multi-Phase Uniform Complete Input Data 174
Multi-Phase Non-Uniform Complete Input Data 182
6.2.3 Input Order Sensitivity in STCNs 182
Engineered Sensitivity 183
Natural Sensitivity 183
6.2.4 Conclusions 187
6.3 AUTOMATON INFORMATION 188
6.3.1 Using Automaton Information to Make Learning Tractable . 188
6.3.2 When Strings Become Illegal 191

6.3.3 Push/Pop Information 192
6.3.4 Conclusions 194
6.4 FREQUENCY CODED STRINGS 194
6.4.1 Stochastic Grammars, Languages, and Automata 195
6.4.2 Stochastic Grammar Induction by Non-Stochastic Automata 203
6.4.3 Simplicity vs. Similarity 204
An Example 205
Minimization of Stochastic Automata 209
Applying Minimization to the Example 210
6.4.4 Previous Results Concerning Stochastic Grammar Induction 216
6.4.5 Predictor STCNs are Most Probably Correct Stochastic Grammars 221
6.4.6 A New Stochastic Grammar Induction Algorithm that is not New 226
6.4.7 Conclusions 227
6.5 CONCLUSIONS 228

Chapter VII: Conclusions 229
7.1 A THEORY OF GRAMMATICAL INDUCTION IN THE CONNECTIONIST PARADIGM 229
7.2 RESULTS 230
7.3 CONCLUSIONS 235
7.4 LIMITATIONS AND FUTURE WORK 236

Bibliography 238

List of Tables

Table 2-1: Mozer's (1993) taxonomy. 15

Table 2-2: Forms of short-term memory (adapted from Mozer, 1993). 18

Table 2-3: Horne and Giles' (1994) Taxonomy. 23

Table 2-4: Tsoi and Back's (1994) taxonomy. 26

Table 2-5: The four basic questions which must be answered by any STCN, the corresponding dimensions of the taxonomy, and the mathematical functions describing the dimensions. 30

Table 2-6: A generic algorithm for STCNs. Particular network designs differ only in their instantiation of the four fundamental functions. 32

Table 2-7: Specific STCN Designs. 62

Table 3-1: Grammars, languages and machines of the Chomsky hierarchy. 77

Table 4-1: Transition tables for the cycles depicted in the previous figure. 122

Table 4-2: Number of state units and types of output functions required by STCNs to implement various automata. 150

Table 6-1: Grammar for simple sentences (adapted from Elman, 1991a, Table 1, p.202). 177

Table 6-2: Simple stochastic grammar. 197

Table 6-3: Frequently generated strings and their probabilities. 198

Table 6-4: Transition function for a stochastic finite state machine. 199

Table 6-5: Deterministic representation of stochastic grammar. 207

Table 6-6: Sample strings generated by probabilistic grammar/automaton. 208

List of Figures

Figure 2-1: *TIS* and the *standard* architecture. (a) A *TIS* memory architecture. (b) The *standard* architecture. (Adapted from Mozer, 1993.) 21

Figure 2-2: Short term memory of WIT network. 34

Figure 2-3: CIIR memory. 36

Figure 2-4: SLFOCC memory. 37

Figure 2-5: First order vs. second order connections. (a) Three first order connections from nodes *j*, *k*, *l*, to node *i*. (b) Two second order connections from *j* & *k*, *k* & *l*, to node *i*. (c) Diagram of computation performed by first order connections. (d) Diagram of computation performed by second order connections. 39

Figure 2-6: SLSOCC memory. 41

Figure 2-7: The CPA memory. 44

Figure 2-8: The CTM memory. 45

Figure 2-9: The LRSI memory. 46

Figure 2-10: Output functions of STCNs. (a) Zero Layer. (b) One Layer. (c) Two Layer. 48

Figure 2-11: An encoder network. 55

Figure 2-12: Adding nodes to a LRSI Memory. (a) Before any nodes are added. (b) After one node is added. (c) After two nodes are added. (d) After the addition of a third node. 59

Figure 3-1: A logical mechanism to compute the parity of a string of 0's and 1's. . 71

Figure 3-2: The Chomsky hierarchy and a reduced hypothesis space. 91

Figure 4-1: Possible cycles in a two-dimensional binary state space. 120

Figure 4-2: Intersections between node activation and diagonal functions. (a) For $W[i][i] > 0$, and one point of intersection. (b) For $W[i][i] > 0$, and three points of intersection. (c) For $W[i][i] < 0$. 133

Figure 5-1: Initializing weights to limit space. 158
Figure 5-2: How initial weights can reduce the hypothesis space to exclude optimal solutions. 160
Figure 5-3: Automaton devoted to detect the Italian word /Numa/. 164
Figure 6-1: Error spaces for an STCN learning a two-string language. (a) Error space for string 1. (b) Error space for string 2. (c) Combined (total) error space. 185
Figure 6-2: Recursive phrase structure. 193
Figure 6-3: Stochastic finite generator automaton (adapted from Reber, 1967) . . 201
Figure 6-4: Recreated stochastic generator automaton. 214
Figure 6-5: Minimized stochastic generator automaton. 215

List of Abbreviations

WIT	- window in time
CIIR	- connectionist infinite impulse response filter
SLFOCC	- single-layer first order context computation
SLSOCC	- single-layer second order context computation
CPA	- connectionist pushdown automaton
CTM	- connectionist Turing machine
LRSI	- locally recurrent state and input
FGD	- full gradient descent
TF	- teacher forcing
TGD	- truncated gradient descent
AAGD	- auto-associate gradient descent
SL	- stack learning
MAC	- manual architecture changes
AIN	- automatically incrementing nodes

List of Symbols

Symbol	Meaning
α, β, γ	-strings of symbols (terminals or non-terminals)
Γ	-alphabet of stack/tape symbols
δ	-transition function mapping $Q\times(\Sigma\cup\{\epsilon\})\times\Gamma$ to finite subsets of $Q\times\Gamma^*$ or $Q\times\Gamma$ to $Q\times\Gamma\times\{L,R\}$ or $Q\times\Sigma$ to Q
Δ	-output alphabet
Δt	-units of time
ϵ	-null String
ϵ	-push/pop threshold
ϵ	-maximum activation value interpreted as 0
λ	-mapping from Q to Δ
Λ	-net external input to node i
λ_0, λ_1	-net external input to node $i+1$ for even and odd t
$\Xi(j)$	-set of split states corresponding to un-split state j
Σ	-alphabet of symbols used in a grammar / input alphabet
Σ	-alphabet of language
$\sigma(x)$	-sigmoid squashing function $\sigma(x)=1/(1+e^{-x})$
$\vec{\omega}$, $\vec{\mu}$	-kernel parameters
Ω	-lower bound (at least Ω is required)
a, b, c, d, e	-example stack symbols
$\vec{a}(t)$	-action vector
A, B, C	-non-terminal symbols
B	-blank symbol $B\in\Gamma$, $B\notin\Sigma$
b_k	-k^{th} output symbol
c	-large constant
$\vec{c}(t)$	-temporal convolution kernel
$d(\cdot)$ (e.g. $d(\vec{x})$)	-dimensionality of vector
$\vec{E}(t)$	-error vector: $\vec{E}(t)\equiv\vec{y}^*(t)-\vec{y}(t)$

Symbol	Meaning
$f_{\Delta d(\vec{s})}(\ \vec{E}(t)\ )$	-function to compute state vector size change
$f_{\Delta W}(\ \vec{E}(t),\ W,\ \vec{x}(\cdot)\ )$	-function to compute weight change
$f_{\vec{s}}(\ \vec{s}(t-1),\ \vec{x}(t),\ W\ )$	-function to compute state vector
$f_{\vec{y}}(\ \vec{s}(t),\ W\ )$	-function to compute output vector
$\vec{f}(\vec{u})$	-result of applying the function $f(x)$ to each element of $\vec{u}$
F	-set of accepting states ($F \subseteq Q$)
G	-formal grammar
$\vec{i}_0,\ \vec{i}_1$	-vector encoding of symbols "0" and "1"
$i_0, i_1, i_2, \dots, i_{p-1}$	-automaton input symbols
I_t	-input symbol at time t
l	-length of stack
L	-language
$L(G)$	-language described by grammar G
$L(n)$	-number of "really different" automata with n states
m	-size of input alphabet
$M(m)$	-number of "really different" automata which can be implemented by an automaton with m nodes
$M=(Q,\Sigma,\Delta,\delta,\lambda,q_0)$	-Moore machine
$M=(Q,\Sigma,\Gamma,\delta,q_0,Z_0,F)$	-pushdown automaton
$M=(Q,\Sigma,\delta,q_0,F)$	-finite state automaton
$M=(Q,\Sigma,\Gamma,\delta,q_0,B,F)$	-Turing machine
$net_{(q,i)}(t)$	-weighted sum of input activations to state node at time t
o	-size of output alphabet
p	-number of automaton input symbols
P	-production rules
Q	-finite set of states
$q(\mathrm{t})$	-state at time t
q_0	-initial state ($q_0 \in Q$)
$q_0, q_1, q_2, \dots, q_{n-1}$	-automaton states

Symbol	Meaning
$\vec{r}(t)$	-symbol read from external state/tape
$\vec{s}(t)$	-short-term memory vector at time t
$\vec{s}_{10}$, $\vec{s}_{11}$	-state for odd strings ending in "0" and "1" respectively
s	-number of states in finite state stack controller
$\vec{s}_0$, $\vec{s}_1$	-state vectors for even and odd parity
$\vec{s}_{(q,i)}$	-$(q, i)^{th}$ component of state vector
$\vec{s}_i$	-i^{th} component of state vector
$s(t)$	-input symbol at time t
s_1	-input symbol
t	-point in time
T	-terminal symbols in grammar
V	-variable symbols in a formal grammar
w	string of terminal symbols ($w \in T^*$)
$w_{\vec{x}_i(t), \vec{s}_{(q,i)}(t)}$	-weight of connection from input node to state node
w	-possibly empty string of terminals
$w_{1, s_{(q,i)}(t)}$	-bias value of state node
W	-array of connection weights
w_{ijk}	-second order weight connection input node i and context node j to state node k
w_{ijkl}	-third order weight connecting input node i, context node j, and read node k to state node l
W^z	-matrix of weights between state and extraction units
$W[1..i][i]$	-weights of all connections to node i
$W[i][i]$	-recurrent connection to node i
$W[i][j]$	-connection from node i to node j
$W[i][j][k]$	-weight connection input node i and context node j to state node k
$\vec{x}_i(t)$	-i^{th} component of vector at time t
$X(t)$	-input symbol at time t

Symbol	Meaning
x_i	-specific symbols in input alphabet
$\vec{y}(t)$	-output vector at time t
$\vec{y}^*(t)$	-desired response at time t
y_i	-specific symbols in output alphabet
$\vec{z}(t)$	-activations of extraction units
Z_0	-initial stack symbol
$[i..j]$ (e.g. $\vec{x}[i..j]$)	-vector consisting of ith to j^{th} components
$\Rightarrow$	-rewrite operator
$\rightarrow$	-production operator
$\overset{*}{\Rightarrow}$	-reflexive transitive rewrite operator (reflexive transitive closure of $\Rightarrow$)
* (e.g. Σ^*)	-Kleene closure (e.g. of Σ)
$\oplus$	-concatenation of vector components

Chapter 1: Introduction

1.1 THE THESIS OF THIS DISSERTATION

We show that the tractability and efficiency of training particular connectionist networks to implement certain classes of grammars can be formally determined by applying principles and ideas that have been explored in the symbolic grammatical induction paradigm. Furthermore, this formal analysis also allows networks to be tailored to efficiently solve specific grammatical induction problems.

In particular, we examine the problem of grammatical induction as studied by Gold (1967):

Definition: Grammatical induction is the process whereby a learning system attempts to identify a finite representation, called a grammar, for a potentially infinite set of strings

of symbols, called a language, based on a finite set of example strings chosen from the language (and possibly its complement).

We view the process of identifying the grammar for a language as a search through an hypothesis space of candidate grammars. It has long been known that this type of search can be improved by functionally pruning the hypothesis space or ordering the sequence of its exploration. Pruning and ordering techniques used for grammatical induction in symbolic paradigms can be employed in connectionist paradigms as well. In fact, the techniques that are being used by connectionist researchers today to improve the performance of their grammar induction systems can be recognized as techniques that have been formally analysed and explored in the symbolic domain.

1.2 OUTLINE

We establish and explore the implications of this thesis in the following ways: First (in Chapter II), we propose a new taxonomy for characterizing connectionist approaches to solving the grammatical induction problem. Our taxonomy differs from previous ones in that it is based along the four fundamental design decisions which the designer of any grammatical induction system must make. This design-decision approach ensures that the scope of the taxonomy is broad enough to accommodate any current or future connectionist grammatical induction system. We apply our taxonomy to the leading connectionist approaches to grammatical induction and describe their relation to each other. In doing so, we unite networks with common features so that results which are proven for one network can be readily applied to others. We are thus also able to predict the performance of existing and proposed connectionist grammar induction systems without implementing them and generating empirical test results.

Second (in Chapter III), we provide a formal description of the grammatical induction problem. We identify the inherent difficulties involved in identifying grammars based on example strings as described in the theoretical induction work of Gold (1967). By describing the induction problem in the context of a search through an hypothesis space, we define two methods for overcoming the difficulties of the search: (1) pruning the hypothesis space and (2) ordering the exploration of the hypothesis space.

Third (in Chapter IV), we offer formal proofs defining the hypothesis spaces of the leading connectionist networks. Specifically, we identify, for the first time, the types of grammars that can be implemented by window in time memories (Sejnowski and Rosenberg, 1986; Lang, Waibel and Hinton, 1990; Lapedes and Farber, 1987; Waibel, Itanazawa, Hinton Shikano and Lang, 1989). We also prove that single-layer first-order context computation memories (Elman, 1990, 1991a; Pollack 1989, 1990, 1994; Maskara and Noetzel, 1992; Williams and Zipser, 1989) can implement arbitrary finite state automata and that they can do so using at most $n \cdot p$ nodes (where n is the number of states in the automaton, and p is the number of input symbols). We further prove, that single-layer second-order context computation memories (Giles, Chen, Miller, Chen, Sun and Lee, 1991; Giles, Miller, Chen, Chen, Sun and Lee, 1992a; Giles, Miller Chen, Sun, Chen and Lee, 1992b; Giles and Omlin, 1992; Goudreau, Giles, Chakradhar and Chen, 1994; Liu, Sun, Chen, Lee and Giles, 1990; Omlin and Giles, 1994a, in press; Shaw and Mitchell, 1990; Sun, Chen, Giles, Lee and Chen, 1990b; Sun, Chen, Lee and Giles, 1991; Watrous and Kuhn, 1992a, 1992b) are incapable of implementing arbitrary automata using binary state encodings, and that locally recurrent state and input memories (Fahlman, 1991; Giles, Chen, Sun, Chen, Lee and Goudreau, 1995; Kremer, in press, 1995b, 1995c) are incapable of representing certain finite state automata whose state transitions form cycles under cyclical input signals. Together, these results relate classes of networks to the classes of grammars that can be represented. Conversely, the results also identify

which networks are capable and, more importantly, which are incapable of solving specific grammar induction problems. This transforms the task of selecting or designing the ideal network for a given problem from an empirical task to one which can be made on the basis of known theoretical results.

Chapter V then describes how the users of connectionist grammatical induction systems can prune and order hypothesis spaces beyond the restrictions imposed by the choice of architecture. This is done by formalizing the relation between the initial or fixed weights in a network and the hypothesis space of representable grammars. New proofs are presented that show how initial weights can both order and restrict the hypothesis space and how good *a priori* knowledge encoded in initial weights will limit the hypothesis space much more than bad *a priori* knowledge. These results give the users of connectionist grammar induction systems even more control over the exploration of the space of potential grammars.

Fifth (in Chapter VI), the dissertation examines the use of *a posteriori* knowledge, provided during training to guide a grammar induction system's exploration of the hypothesis space. In particular, three different forms of *a posteriori* knowledge are discussed: input ordering, automaton information, and string frequency information. All three of these approaches have been used in symbolic grammar induction systems. We prove that training techniques used in connectionist networks can be recognized as implementations of input ordering, providing automaton information and stochastic grammar induction.

1.3 IMPLICATIONS

Had the formal work that is reported in this dissertation been done earlier, it is possible that connectionist researchers would have been able to take a formal, rather than empirical, approach to understanding the computational power of their nets for

grammatical induction. As well, our formal approach could have been applied to understand techniques available to functionally increase the power of connectionist grammar induction systems which are now being discovered empirically. This dissertation, by considering classical work done over the past three decades, gives a formal grounding to these empirically discovered methods. In doing so, it also suggests a rationale for making the design decisions which define every connectionist grammar induction system. This allows new networks to be better suited to the problems to which they will be applied. Finally, the dissertation provides insights into applying other refinement techniques that connectionist researchers have yet to consider.

Chapter II: Connectionist Networks for Grammar Induction

2.1 INTRODUCTION

This chapter describes a new type of taxonomy for spatio-temporal connectionist networks (STCNs) based around a formal mathematical description of grammatical induction. The new taxonomy is the only one developed around the fundamental design decisions which must be addressed by any grammatical induction system. Because of this, it has superior predictive power when used to compare and analyse how different STCNs might perform on various grammatical induction problems (the most common application of STCNs, and the focus of this thesis). Additionally, the taxonomy is general enough to accommodate all of the leading STCN designs described in the literature. In fact, the exact categorization of each leading design is precisely specified. Furthermore, the fact that the taxonomy is centred around the principles of grammatical induction systems,

rather than specific existing STCN designs, implies that it will easily accommodate future STCN designs as well.

In addition to surveying the field of connectionist grammar induction systems, this chapter provides a simple way of applying the results of the analyses in the chapters which follow to many different STCN designs currently in use as well as future STCN designs. Since the taxonomy can be used to predict STCN performance for various problems, it can serve as a tool for anyone needing to select or design a particular STCN to solve a given problem.

This chapter is organized as follows: First, we define the notation and terminology used. Second, we identify features of good taxonomies. Third, we examine the characteristics and merits of several existing STCN categorization schemes. Fourth, we develop a new taxonomy possessing the ideal features identified earlier. Fifth, we specify how existing STCN designs fit into the new taxonomy. And sixth, we present some conclusions.

2.2 TERMINOLOGY AND MATHEMATICAL PRELIMINARIES

2.2.1 Spatio-temporal Connectionist Networks (STCNs)

A spatio-temporal connectionist network can be defined as a parallel distributed information processing structure that is capable of dealing with input data that are presented across time as well as space. One might argue that this definition is too informal and describes too diverse a set of computational systems in order for any useful theorems about STCNs to be proven. This is a valid observation. However, assigning a more precise mathematical definition to STCNs would contradict the common use of the term to describe a variety of very diverse architectures. Instead, in this chapter, we provide precise mathematical descriptions of the most popular classes of STCNs and their constituent components, as opposed to one definition for all STCNs. This will allow us

to develop formal theorems about STCNs with specific properties, without unnaturally restricting the definition of STCN in general. The following formalisms serve as a basis for the subsequent description of specific STCN components.

2.2.2 Search States and Machine States

In this dissertation, we view the process of grammatical induction as a search for a target grammar from a set of possible or candidate grammars. This type of search is often referred to as a state-space search and each point in the space to be searched is referred to as a state. However, in the context of formal grammars and automata, the word "state" is also used to describe the internal state of a grammar or machine. This creates an ambiguity when the word "state" is used in the context of grammatical induction, where a state-space search is conducted for a target state, which represents a formal grammar or machine, whose internal state is used to determine the grammaticality of sample sentences. As is convention in the grammatical induction literature, we resolve this ambiguity by using the word "state" exclusively to refer to the internal state of a formal computing machine or grammar while describing candidate grammars as "hypotheses" and the space of candidate grammars as the "hypothesis space".

2.2.3 A Note About Notation

We begin by introducing the mathematical formalisms used to describe points along our dimensions. It is convention in algebra to interpret a vector $\vec{x} = (x_1, x_2, x_3, \dots)$, as being equivalent to the column matrix:

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \end{pmatrix}$$

This allows us to rewrite the previous equation for the vector $\vec{y}$ without the transpositions as simply:

$$\vec{y} = W \times \vec{x}$$

Where $\times$ denotes the matrix product. Hence, each element, y_i, of $\vec{y}$ is computed:

$$y_j = \sum_i W_{ji} \cdot x_i$$

In addition to frequent matrix-vector multiplications, we will frequently use vectors whose elements consist of the elements of a series of smaller vectors. In order to simplify the description of these vectors, we will use $\oplus$ to represent to represent a function mapping two vectors, e.g. $\vec{b}$ and $\vec{c}$ in vectors spaces $\boldsymbol{B}$ and $\boldsymbol{C}$, to a new vector $\vec{a}$ in the vector space defined by the Cartesian product of spaces, $\boldsymbol{B}$ and $\boldsymbol{C}$, of the original vectors. The operation is defined such that the representation of vector $\vec{a}$ is formed by concatenating the representation of vector $\vec{b}$ with the representation of vector $\vec{c}$. Specifically, if $\vec{a} = \vec{b} \oplus \vec{c}$ then:

$$a_i = \begin{cases} b_i & \text{if } i \leq d(\vec{b}) \\ c_{i-d(\vec{b})} & \text{if } i > d(\vec{b}) \end{cases}$$

and

$$d(\vec{a}) = d(\vec{b}) + d(\vec{c})$$

where $d(\vec{a})$ represents the dimensionality of $\vec{a}$.

We will further require an inverse of this operator, capable of extracting a smaller vector from a larger vector formed by $\oplus$. To do this, we define the use of square parentheses in the expression $\vec{d} = \vec{e}[i..j]$ such that vector d is equal to a vector whose components are equal to components i through j of vector $\vec{e}$. More precisely we require that the kth component of vector $\vec{d}$ is equal to the $(i+k\text{-}1)^{th}$ component of $\vec{e}$:

$$\vec{d}_k = \vec{e}_{i+k}.$$

and that the dimensionality of vector d is equal to the number of terms in the inclusive sequence from i to j:

$$d(\vec{d}) = j - i + 1 .$$

It is now possible to invert the $\oplus$ operator:

$$\vec{a} = \vec{b} \oplus \vec{c}$$

such that

$$\vec{b} = \vec{a}[1..\|\vec{a}\| - \|\vec{c}\|],$$

and

$$\vec{c} = \vec{a}[\|\vec{b}\| + 1 .. \|\vec{a}\|].$$

2.2.4 Input Dimensions

Spatio-temporal connectionist networks (STCNs) are connectionist systems that are capable of dealing with input and output patterns that vary across time as well as space. For the purposes of simulation on digital computers, it is useful to discretize the temporal dimension and consider a system in which time proceeds by intervals of Δt. We shall use the symbol t to represent a particular point in time, where $t \in \{0, \Delta t, 2\Delta t, 3\Delta t, \ldots\}$. In this formulation Δt can be considered to be the unit of measure for the quantity t, and thus it is simplest to omit the units and express t simply as a member of the set of whole numbers.

The dimension of time must be different from the spatial dimensions in conventional connectionist networks. Components of an input pattern distributed across space (i.e. across various input nodes) can all be accessed at the same time. However, only the current component of patterns distributed across time is accessible at any given instant. We assume that the observable world for a STCN at the instant t is described by an input vector, $\vec{x}(t)$. This vector is supplied to the STCN at time t by setting the

activation values of the input units of the STCN to the components of the vector. Thus, the input vector can be considered a stimulus.

It is important to note the difference between information encoded across the input vector $\vec{x}(t)$ and information encoded in *different* input vectors $\vec{x}(t_1)$, $\vec{x}(t_2)$, ..., presented to the network at different times. Values of the former can be accessed in parallel while the latter must be accessed sequentially (in a forward direction). This difference is similar to that between the random access memory of a computer disk system, and the sequential access memory of a tape device. Both devices can be considered to encode different pieces of information at different locations in space. However, the restricted access available to the sequential access device has resulted in great differences in the uses of tape memory as compared to disk memory. The disparity between the two input dimensions in the connectionist system is even greater since the temporal tape can never be rewound and reread (it is never possible to access input vectors in a backward direction).

2.2.5 Spatio-Temporal Connectionist Network Memory

Conventional connectionist models are equipped with memory in the form of connection weights, denoted by the matrix W. This adjustable parameter is typically updated after each training step and constitutes a memory of all previous training. In the sense that this memory extends back to the first training step, we shall refer to the memory in conventional connectionist models as long-term memory. Once a STCN has been successfully trained, this long-term memory remains fixed during the operation of the network.

The distinguishing characteristic of STCNs is that they also include a form of short-term memory. It is this memory which allows these networks to deal with input and output patterns that vary across time as well as space and thus defines them as STCNs. Conventional connectionist networks compute the activation values of all nodes at time t

based only on the input at time t. By contrast, in STCNs the activations of some nodes at time t are computed based on activations at time t-1. These activations serve as a short-term memory. We use the "state vector", $\vec{s}(t)$, to represent the activations at time t-1 of those nodes that are used to compute the activations of other nodes at time t. Unlike the long-term memory which remains static once training is completed, the short-term memory is continually recomputed with each new input vector (both during training and during operation). This is due to the fact that long-term memory is stored in connection weights (which are updated only during training) while short-term memory is represented by node activations (which are computed with each time-step even after training).

2.2.6 Output, Teaching, and Error

Finally, it is necessary to specify a representation for the response of the network to its stimuli. Like the traditional models, STCNs encode their response in the activations of a special set of units called "output units". Thus, we represent the output of a STCN by a vector, $\vec{y}(t)$. Most connectionist architectures learn by computing the difference between their response and a teacher-supplied desired (or ideal) response and adjusting their long-term memory accordingly. We denote the desired response by another vector, $\vec{y}^*(t)$. The difference between the desired output vector and the actual output vector is the error vector $\vec{E}(t) = \vec{y}^*(t) - \vec{y}(t)$, and the total network error, ε, is defined as one half of the square of the magnitude of this vector:

$$\varepsilon = \sum_{t \in T} \left\{ \frac{1}{2} \|\vec{E}(t)\|^2 \right\}.$$

2.3 FEATURES OF A GOOD TAXONOMY

Before presenting our taxonomy of STCNs for grammar induction, we briefly discuss general properties of good taxonomies. Later we evaluate our new taxonomy as

well as Mozer's (1993), Tsoi and Back's (1994), and Horne and Giles' (1994) according to these criteria. The first property which an effective taxonomy must possess is *descriptive adequacy*. This means that it must exhaustively classify all objects in its domain. A second important property is *simplicity*. It should be a relatively simple task to classify and name any of the objects in the domain. The third, and most important, quality of a good taxonomy is *predictive power*. That is, objects in the taxonomy with similar classifications should posses similar properties, while objects in the taxonomy with differing classifications should have differing properties.

A good taxonomy balances simplicity with predictive power while maintaining descriptive adequacy. In other words, it must have a level of discrimination that is not so low that it groups dissimilar objects together, nor so high that the number of groups required to encompass all objects becomes unwieldy. This can best be accomplished if the taxonomy describes features along multiple orthogonal dimensions rather than along one single dimension. If multiple dimensions are available, then different yet similar objects can be classified as having the same feature along one or more dimensions, while having different features along other dimensions. By contrast, in a single-dimension taxonomy, the two object must be classified as either "the same", or "different".

2.4 EXISTING TAXONOMIES

2.4.1 Mozer (1993)

Mozer (1993) has developed one taxonomy for STCNs which is illustrated in Table 2-1. It is based on the assumption that every STCN consists of two mechanisms: a short-term memory and a predictor. The short-term memory computes the state of the STCN, while the second mechanism, uses the state to compute output. Mozer characterizes the operation of the short-term memory of STCNs along the dimensions of content and form (which define how the short-term memory is computed), and adaptability (which defines

how changes in the computation of the short term memory are made during the induction process). Mozer notes that "the predictor component [of an STCN] will always be a feedforward component of the net" (Mozer, 1993, p. 244). Table 2-1 illustrates Mozer's taxonomy and the three dimensions (indicated in bold) used to describe STCNs. We now discuss each dimension in turn.

Short-Term Memory		
	Computation of Short-Term Memory	
		Content:
		Input (I)
		Transformed Input (TI)
		Transformed Input and State (TIS)
		Output (O)
		Transformed Output (TO)
		Transformed Output and State (TOS)
		Form:
		Delay Line
		Exponential Trace
		Gamma
	Changes in Computation of Short-Term Memory	
		Adaptability:
		Static
		Adaptive
Predictor		
		Feedforward Component of Net

Table 2-1: Mozer's (1993) taxonomy.

Memory Content

Mozer breaks the computation of short-term memory, $\vec{s}(t)$, into two steps: a content computation and a form computation. Both steps are essentially filtering transformations of the available data into a simpler forms. The first step's transformation calculates a new vector, $\vec{x}'(t)$, whose component values are computed based on information distributed in space across the input vector, $\vec{x}(t)$. In this sense, the transformation constitutes a spatial filter. Mozer terms the result of this transformation, $\vec{x}'(t)$, the memory *content*, and identifies six possibilities for the transformation: input, transformed input, transformed input and state, output, transformed output, transformed output and state (indicated in italics in Table 2-1).

The simplest possible spatial transformation is an identity transformation whereby $\vec{x}'(t) = \vec{x}(t)$; Mozer calls this an *input* or *I* memory. A more sophisticated filter involves computing a new vector whose components are the results of applying a squashing function, $\sigma(x)$ to weighted sums of the input vector components. If we now let $\vec{f}(\ \vec{u}\ )$ represent the vector whose elements are the result of applying the function $f(x)$ to each element of $\vec{u}$, then the advanced filter computes:

$$\vec{x}'(t) = \vec{f}(\ W \times \vec{x}(t)\ ).$$

Typical instantiations of $f(x)$ are the sigmoid function, $\sigma(x) = \frac{1}{1+e^{-x}}$, and the hyperbolic tangent, $\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$. Mozer calls this a *transformed input* or *TI* memory. A third filter is based on a transformation of not only the input vector, but also the state vector. Using vector concatenation, it is possible to define a transformation:

$$\vec{x}'(t) = \vec{f}(\ W \times (\vec{x}(t) \oplus \vec{s}(t))\ ).$$

In this situation, the components of vector $\vec{x}'(t)$ represent squashed weighted sums of the components of both the input, $\vec{x}(t)$, and state, $\vec{s}(t)$, vectors. Mozer calls this a *transformed input and state* or *TIS* memory.

The fourth, fifth and sixth filters in Mozer's classification scheme are obtained by replacing the input vector, $\vec{x}(t)$, in all of the equations above by the previous output vector, $\vec{y}(t-1)$. This is typically done for autopredictive tasks where input and previous output are identical. Thus, the memory content for the *I* memory described above is replaced by $\vec{x}\,'(t) = \vec{y}(t)$, and a new memory which Mozer calls *output* or *O* is formed. By performing similar replacements, Mozer also defines the *transformed output*, *TO*, and *transformed output and state*, *TOS*, memories.

Memory Form

The second filtering step which is applied in the computation of short-term memory calculates the short term memory vector, $\vec{s}(t)$, whose value is computed based on information distributed across time in a series of content vectors, $\{\vec{x}\,'(t),\ \vec{x}\,'(t+1),\ \vec{x}\,'(t+2),\ \ldots\}$. In this sense, the second transformation constitutes a temporal filter. According to Mozer, this defines the "form" of memory. The short-term memory vector, $\vec{s}(t)$, is computed based on the temporal convolution of the memory content, $\vec{x}\,'(t)$, with a kernel function, $\vec{c}(t)$:

$$\vec{s}_i(t) = \sum_{\tau=1}^{t} \vec{c}_i(t-\tau) \cdot \vec{x}_i'(\tau).$$

A variety of different possible kernels are discussed by Mozer. These are summarized in Table 2-2. The value of a given component, *i*, of the state vector at time *t*, $\vec{s}_i(t)$, is equal to a weighted sum of previous values of the corresponding component of the memory content vector, $\vec{x}_i'(\tau)$, over time, τ. The kernel function defines the weight, or importance, given to the content vector activation for each point in time. For example, a delay line memory specifies that only one of the previous values of $\vec{x}\,'(\tau)$, namely $\vec{x}\,'(t-\omega_i)$, is used for the state variable $\vec{s}_i(t)$. That is:

$$\vec{s}_i(t) = \vec{x}_i'(t-\omega_i).$$

memory form	*kernel function*
delay line	$c_i(t) = \begin{cases} 1 & \text{if } t = \omega_i \\ 0 & \text{otherwise} \end{cases}$
exponential trace	$c_i(t) = (1 - \mu_i)\mu_i^t$
gamma	$c_i(t) = \begin{cases} \binom{t}{\omega_i}(1 - \mu_i)^{\omega_i + 1}\mu_i^{t - \omega_i} & \text{if } t \geq \omega_i \\ 0 & \text{if } t < \omega_i \end{cases}$

Table 2-2: Forms of short-term memory (adapted from Mozer, 1993).

By contrast, in an exponential trace memory a large weight is given to the most recent memory content component, and exponentially decaying weights are given to earlier values. Other types of kernels operate in a similar fashion. For a more detailed discussion of various kernel functions, the reader is referred to Mozer (1993).

Memory Adaptability

The third dimension used by Mozer to classify connectionist models is memory adaptability. This term refers to the parameters (ω_i, μ_i, W) in the content and form equations. Mozer uses the term *static* memory to indicate that these parameters are constant during the training process. This means that memory state is a predetermined function of the input sequence and cannot be trained to accommodate the peculiarities of

a particular problem. The author uses the term *adaptive* memory to indicate that the connectionist network can adjust memory parameters (typically by means of a gradient descent algorithm) during the training phase. The memory parameters in both static and adaptive memories always remain constant during the performance phase. Mozer also notes that "the Elman SRN algorithm (Elman, 1990) lies somewhere between an adaptive and static memory because of the training procedure--which amounts to back propagation one step in time--is not as powerful as full blown back propagation through time" (Mozer, 1993, p. 253).

Evaluation of Mozer's Taxonomy

Although Mozer's taxonomy is designed as a general system for STCNs, there are two major limitations with using Mozer's framework. The first concerns sacrificing predictive power for descriptive adequacy and simplicity. As noted above, Mozer's taxonomy applies only to the short-term memory component of the network. He makes no attempt to describe the predictor component beyond stating that it is a network. This means that networks with different predictor components (e.g. multi-layer vs. single-layer vs. no-layer, or first-order connections vs. second-order connections) are grouped in the same category. Since it is well known that the number of layers and order of connections greatly affects what a connectionist network can compute (see Lippmann, 1987), predictive power is limited by not categorizing STCNs based on their predictor components.

The same argument applies to the short-term memory component. *TIS* memories encompass a very broad range of connectionist networks (with varying layer numbers and connectivity schemes), including Fahlman's recurrent cascade correlation networks and Giles' second order networks, two approaches which are vastly different in both approach and (as we shall see in Chapter IV) computational power.

The final problem with the predictive power of Mozer's framework has to do with the kernel representation of memory form. If memory form is calculated as:

$$\vec{s}(t) = \sum_{\tau=1}^{t} \vec{c}(\tau) \cdot \vec{x}'(\tau)$$

then this, in general, requires that the values of $\vec{x}'(t)$ be available for all t. But, the values of the $\vec{x}'(t)$ depend directly on the input vectors $\vec{x}(t)$. This in effect changes the temporal dimension to a spatial dimension of infinite size which does not reflect the true nature of the encoding. The solution to this is, of course, to compute the same memory vector defined by the convolution above, incrementally based on the previous memory vector (which may require more memory for the state vector). While Mozer discusses this type of state computation, his framework does not use it. This implies that it is difficult to identify the time and memory complexity of the iterative computation which an implemented network will actually use since it is "disguised" by the convolution operator.

The second major problem with Mozer's taxonomy is that it sacrifices too much descriptive adequacy for simplicity and predictive power. Mozer himself identifies one interesting type of STCN—which he calls the *standard* architecture (e.g. Elman, 1990; Mozer, 1989)—that does not fit within his classification scheme. Although the *standard* architecture is very similar to *TIS*, there are differences between the two approaches. These are illustrated in Figure 2-1. Note that the *standard* architecture is like *TIS*, except that the first two hidden layers have been collapsed into a single layer. Nor does the *standard* architecture fit anywhere else in the different possibilities for "memory contents" discussed by Mozer.

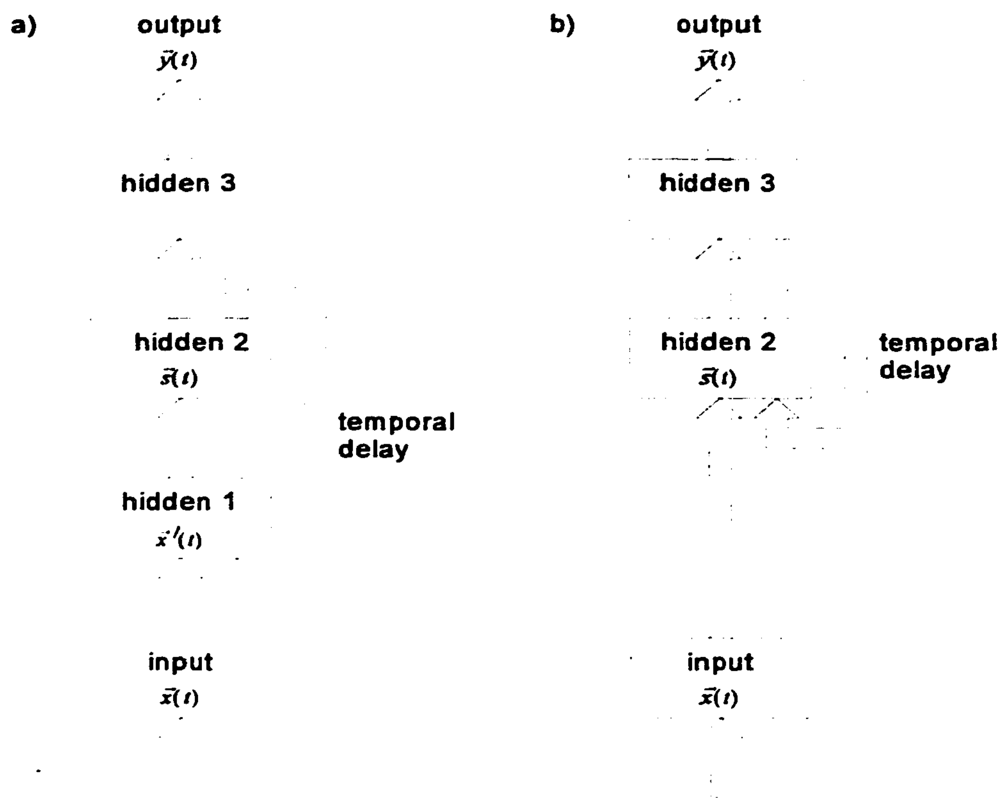

Figure 2-1: *TIS* and the *standard* architecture. (a) A *TIS* memory architecture. (b) The *standard* architecture. (Adapted from Mozer, 1993.)

Another example of sacrificing descriptive adequacy concerns the networks of Narendra and Parthasarathy (1990). These networks compute their outputs based on a history of both input and output patterns. In this sense, these networks could be considered as having *input and output* or *IO* memories. Yet, Mozer's taxonomy does not include this type of memory content.

Additionally, Mozer's scheme does not adequately describe different systems of memory adaptability. Elman's SRN algorithm, for example is described as lying somewhere between *static* and *adaptive*. This seems strange, since Elman's system clearly satisfies the criterion for *adaptiveness* ("the neural net can adjust memory parameters" (Mozer, 1993, p.252)). Yet Elman's algorithm is clearly different (and perhaps less

effective at adapting - see Servan-Schreiber, Cleeremans and McClelland, 1988) than the conventional recurrent adaptation algorithms, because it uses a truncated version of gradient descent. Fahlman has developed yet another adaption algorithm which does not fit into Mozer's taxonomy. Fahlman's networks adapt not only parameters like weights, but also the number of nodes in the network. This is an important distinction, since it implies that Fahlman's networks are capable of increasing their own memory capacity which conventional STCNs are not. Clearly, this type of adaption will have a profound impact on how these networks perform on grammar induction problems.

2.4.2 Horne and Giles (1994)

Horne and Giles (1994) have developed a different taxonomy for STCNs which is illustrated in Table 2-3. Their approach focuses on partitioning the space of existing STCN architectures. The first partition separates those networks whose state representations are encoded in input and output units only from those networks whose state representations are encoded in hidden units. Horne and Giles refer to the former class as networks with "Observable States" and the latter as networks with "Hidden Dynamics". Networks with "Observable States" include Narendra and Parthasarathy's (1990) networks, Lang, Waibel, and Hinton's (1990) Time Delay Neural Networks (TDNN), and deVries and Principe's (1992) Gamma Networks. The class of networks with hidden dynamics is further partitioned into single-layer, multi-layer, and local feedback networks.

Observable States:	
	Narendra and Parthasarathy (1990)
	TDNN (Lang, Waibel and Hinton, 1990)
	Gamma Network (deVries and Principe, 1992)
Hidden Dynamics	
	Single-Layer:
	First Order
	High Order (Giles et al., 1992a)
	Bilinear
	Quadratic (Watrous and Kuhn, 1992b)
	Multi-Layer:
	Robinson and Fallside (1988)
	Simple Recurrent Networks (Elman, 1990 & 1991a)
	Local Feedback:
	Back and Tsoi (1991)
	Frasconi, Gori and Soda (1992)
	Poddar and Unnikrishnan (1990)

Table 2-3: Horne and Giles' (1994) Taxonomy.

Evaluation of Horne and Giles's Taxonomy

It should be noted that Horne and Giles' taxonomy is one-dimensional in the sense that all categories are mutually exclusive. This can be a disadvantage in the sense that it is difficult to propose new architectures by identifying points in the space described by the taxonomy. In a multidimensional taxonomy, new architectures can be defined by identifying points along each of the dimensions that have not been explored in one common STCN. When new types of networks are developed and added to a one-dimensional taxonomy, they cannot be described in terms of combinations of existing categories, but rather must be "lumped into" one existing category or be given an entirely new categorization. This implies that Horne and Giles' taxonomy will tend to suffer from either a lack of predictive power (in the former case), or a lack of simplicity (in the latter case). Horne and Giles' taxonomy also lacks descriptive adequacy since it does not describe some important classes of networks including Recurrent Cascade Correlation (Fahlman, 1991), Recursive Auto-associative Memory (Pollack, 1989, 1990), Auto-associative Recurrent Network (Maskara and Noetzel, 1992), Connectionist Pushdown Automaton (Giles et al., 1990), Connectionist Turing Machine (Williams and Zipser, 1989), and Second Order Constructive Learning (Giles et. al., 1995).

2.4.3 Tsoi and Back (1994)

Tsoi and Back (1994) have developed a taxonomy designed specifically for locally recurrent, globally feedforward networks. These are networks in which all connections are feedforward with the exception of one temporal self-looping connection for each node. Tsoi and Back base their taxonomy on the type delay incorporated in the recurrent connections used, and which value is delayed. Tsoi and Back refer to these two considerations as: "Synapse Type" and "Feedback Location" respectively (whereas Mozer (1993) calls them, "kernel function" and "memory content"). For Synapse Type, they

consider two types of kernel functions: a delay line with $\omega_i = 1$, which they call a "Simple synapse", and a complex linear transfer function with several adaptable poles and zeros, which they call a "Dynamic synapse". The "Feedback location" is subdivided along three dimensions depending on which combination of the nodes' previous activation values, the nodes' previous net inputs, or the previous value are transmitted by the synapse. Thus, Tsoi and Back's taxonomy is composed of four dimensions: Synapse type (which can assume a value of "Simple" or "Dynamic"), Synapse feedback (which can be "Yes" or "No"), Activation feedback (which can be "Yes" or "No"), and Output feedback (which can be "Yes" or "No"). These dimensions are illustrated in Figure 2-4.

Synapse Type	
	Simple
	Dynamic
Feedback Location	
	Synapse *Yes* *No*
	Activation *Yes* *No*
	Output *Yes* *No*

Table 2-4: Tsoi and Back's (1994) taxonomy.

Evaluation of Tsoi and Back's Taxonomy

Tsoi and Back's (1994) taxonomy lacks the descriptive adequacy to classify all connectionist approaches to grammatical induction since it is only capable of dealing with locally recurrent, globally feedforward networks, a small proportion of STCN grammar induction systems. Furthermore, its simplicity is questionable since the authors use four

dimensions to classify a total of four architectures. For these reasons, it too is unsuitable for our needs.

2.4.4 Observations

When Mozer and Horne and Giles first developed their classification schemes, they provided simple, descriptively adequate and predictively powerful frameworks for describing some of the leading STCN approaches of the time. Tsoi and Back also developed a simple taxonomy addressing a popular subset of STCNs. The recent flurry of activity in the field of STCN research, however, has contributed to the increasing obsolescence of these ways of classifying networks. The problems that these three taxonomies exhibit in the face of a rapidly expanding field, can all be traced to a common cause: all three taxonomies are attempts to classify existing STCN designs. Since it is impossible to predict future STCN developments, new designs are difficult to fit into the categories defined earlier. One solution to this problem is simply to omit the new STCN designs from the taxonomy, however, this results in a lack of descriptive adequacy. An alternative solution is to place new designs into the closest possible categories (even if they do not fit well). This situation tends to degrade the descriptive adequacy of the taxonomy since it will be more difficult to make predictions based on tenuous categorizations. The final option is to add a new category for the new network design, but this tends to decrease simplicity due to a proliferation of categories.

2.5 A NEW TAXONOMY FOR SPATIO-TEMPORAL CONNECTIONIST NETWORKS

An alternative to attempting to classify existing STCNs based on their respective properties, is to examine what all STCNs have in common. For the purposes of this thesis, all STCNs are assumed to be applied to grammatical induction. Thus, we develop

our taxonomy around the computational theory of grammatical induction (examining the goals of computation), rather than around the representations and algorithms of STCNs. This approach is related to Marr's (1982, Chapter 1) tri-level hypothesis which describes information processing systems at three levels: (1) the computational theory level, which describes the goal of the computation, (2) the representation and algorithm level, which describes how the computation can be implemented, and (3) the hardware implementation level, which describes the physical realization of the computation. While other taxonomies describe STCNs at the representation and algorithm, level focussing on how computations are implemented, we base our taxonomy around the computational theory level. By focusing on the goals of grammatical induction, we can guarantee that future STCN designs will be accommodated in our taxonomy as long as they are applied to grammatical induction problems. At first, focussing "only" on grammatical induction may seem like a serious limitation of the proposed taxonomy, however, as we shall see in the following chapter, a wide variety of problems in artificial intelligence can be cast into a grammatical induction framework.

Our taxonomy is based on the perspective of anyone wishing to develop a grammatical induction system. Researchers in this position are forced to answer two basic questions. The first is: how does the researcher select, limit and represent the set of grammars which can be induced? This set of grammars is referred to as the hypothesis space. The second question is: what algorithm is used to identify, from the hypothesis space, the particular grammar which best suits the training data? This is the search algorithm. By basing our taxonomy on these two fundamental questions which need to be answered for any hypothesis space search, we can group STCNs into categories that are predictive of performance and applicability to particular classes of induction problems.

We now break down the first question—how does one represent the hypothesis space—into two components. For grammatical induction, a hypothesis represents a

particular network with a specific set of weights or, equivalently, the formal computing machine or grammar defined by the weights. Every formal computing machine can be defined by two components. One component which computes the new internal state of the machine based on the input and previous state, and one component which computes the output of the machine based on the new state. Thus, it is natural to break the issue of representing the hypothesis space along these lines, and to identify two new questions which must be answered. The first is: how does one represent the possible computations which could be performed in order to compute the system's state? The second is: how does one represent the possible computations which could be performed in order to compute the system's output? These two questions, together, are equivalent to the original question (what is the hypothesis space?). This is because the representation of any grammar mechanism can be broken into state and output components, and because these two components completely define the grammar mechanism.

The second question—what is the search algorithm—can also be broken down into two components. The search algorithm effectively defines operators for moving the induction system from one point in the hypothesis space to another. Since every hypothesis represents a specific connectionist network with (1) a specific set of connection weights and (2) a specific number of hidden units, it is natural to break the question along these lines, and identify two new questions. The first is: how does one search for a set of connection weights. The second is: how does one search for an appropriate state vector size. These two sub-questions are also equivalent to the original query (what is the search algorithm?) since the performance of any connectionist network is defined by its architecture and its weights.

Question	*Dimension*	*Function*
How does one represent the possible computations which could be performed in order to compute the system's state?	how the state vector is computed	$f_s(\ \vec{s}(t-1),\ \vec{x}(t),\ W\ )$
How does one represent the possible computations which could be performed in order to compute the system's output?	how the output vector is computed	$f_y(\ \vec{s}(t),\ W\ )$
How does one search for a set of connection weights?	how the change in weights is computed	$f_{\Delta W}(\ \vec{E}(t),\ W,\ \vec{x}(\cdot)\ )$
How does one search for an appropriate state vector size?	how the change in state vector size is determined	$f_{\Delta d(\vec{s})}(\ \vec{E}(t)\ )$

Table 2-5: The four basic questions which must be answered by any STCN, the corresponding dimensions of the taxonomy, and the mathematical functions describing the dimensions.

This leaves us with four fundamental questions that must be answered by any STCN. These questions, which are illustrated in Table 2-6, form the basis-dimensions of our taxonomy. Any grammatical induction system can be defined by its position in this four dimensional space. Specific coordinates along each dimension represent specific answers to the four questions. A set of four such coordinates completely defines an induction system. The four dimensions are: (1) how the state vector is computed, (2) how the output vector is computed, (3) how the change in weights is computed, and (4) how the change in state vector size is determined. These dimensions are illustrated with their corresponding questions in Table 2-5. Together, the dimensions account for the entire operation of the STCN.

Each dimension of the taxonomy can be described as a generic function and the specific instantiation of each function will determine the behaviour of the STCN. The first function, defines how the state vector, $\vec{s}(t)$, is computed, based on: the previous state vector, $\vec{s}(t-1)$, the current input vector, $\vec{x}(t)$, and the weights in the STCN, W. This function will be denoted $f_s(\vec{s}(t-1), \vec{x}(t), W)$. The second function computes the output vector, $\vec{y}(t)$, based on: the current state vector, $\vec{s}(t)$, and the weights in the STCN, W. It will be denoted $f_y(\vec{s}(t), W)$. The third function computes the change in the weights of the STCN, ΔW, in response to the error, $\vec{E}(t)$, the weights, W, and the input history, $\vec{x}(\cdot)$, of the STCN. It is denoted $f_{\Delta W}(\vec{E}(t), W, \vec{x}(\cdot))$. The fourth and final function updates the size of the state vector. This function is denoted $f_{\Delta d(\vec{s})}(\vec{E}(t))$. These functions are illustrated opposite their corresponding dimensions in Table 2-5. Note that we have, as yet, made absolutely no assumptions about the computations performed each of the four functions. In the following sections we will provide example instantiations of the functions that correspond to the processing performed in a variety of popular STCNs.

```
algorithm STCN
  do                                  {try different architectures}
    initialize s(0), t←0, W
    do                                {try different weights}
      increment t
      determine input vector
      s(t) ← f_s( s(t-1), x(t), W )   {compute state}
      y(t) ← f_y( s(t), W )           {compute output}
      E ← y*(t)-y(t)                  {compute error vector}
      ε = Σ_t { 1/2 ||E^2|| }         {compute error scalar}
      W ← W + f_ΔW(E,W,x(·))          {update weights}
    while t<t_max and ε>threshold
    Δd(s) ← Δd(s) + f_Δd(s)(E)        {update architecture}
  while E>threshold
```

Table 2-6: A generic algorithm for STCNs. Particular network designs differ only in their instantiation of the four fundamental functions.

Using these four functions, it is now possible to describe the operation of any STCN by a generic algorithm. This algorithm is described in Table 2-6. In a sense, the algorithm can be thought of as a mathematical definition of STCN parallelling the informal definition provided in Section 2.2.1. Note that, at this point, we have developed a general framework that is applicable to any STCN. By specifying the instantiations of the four functions described above, it is possible to precisely define points along the four dimensions of the taxonomy and hence specific STCN induction systems.

2.5.1 Computing the State Vector

We begin by examining instantiations of the state vector function, $f_s(\vec{s}(t-1), \vec{x}(t), W)$ used in existing STCNs. In particular, we discuss seven popular alternatives: Window In Time memories, Connectionist Infinite Impulse Response Filter

memories, Single-Layer First-Order Context Computation memories, Single-Layer Second-Order Context Computation memories, Connectionist Pushdown Automata memories, Connectionist Turing Machine memories, and Locally Recurrent State and Input memories.

Window In Time (WIT) Memory

The Window In Time (WIT) memory approach to computing the state vector is one of the first types of short-term memory in spatio-temporal networks. Its best known implementation is NETtalk (Sejnowski and Rosenberg, 1986), but it has also been used by a variety of other authors including Lang et. al. (1990), Lapedes and Farber (1987), and Waibel et. al. (1989). The short-term memory vector, $\vec{s}(t)$, is always computed in the same manner throughout the training process and represents a finite, so-called "window in time" on the input symbols. Thus, if the input symbol at time t is $X(t)$, then a window of size 7 time-steps at time t would consist of seven sequential symbols, e.g.: ($X(t)$, $X(t-1)$, $X(t-2)$, $X(t-3)$, $X(t-4)$, $X(t-5)$, $X(t-6)$).

Of course, in order to present these successive input symbols as inputs to the network, it is necessary to represent each symbol as an activation vector across the input units in the network. Thus, the series of input symbols becomes a series of input vectors: ($\vec{x}(t)$, $\vec{x}(t-1)$, $\vec{x}(t-2)$, $\vec{x}(t-3)$, $\vec{x}(t-4)$, $\vec{x}(t-5)$, $\vec{x}(t-6)$). The concatenation of these vectors forms the short-term memory:

$$\vec{s}(t) = \vec{x}(t) \oplus \vec{x}(t-1) \oplus \vec{x}(t-2) \oplus \vec{x}(t-3) \oplus \vec{x}(t-4) \oplus \vec{x}(t-5) \oplus \vec{x}(t-6)$$

However, since only one input vector is available to the network at any time, an STCN using a WIT memory must compute each state based on the previous state and the current input vector. Using the inverse concatenation operator (described in Section 2.2.1), we can now truncate the right-most input vector, $\vec{x}(t-6)$, from the previous state vector and

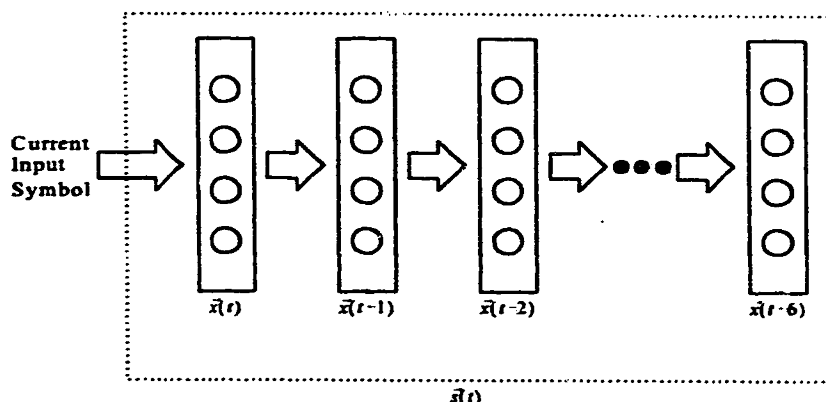

Figure 2-2: Short term memory of WIT network.

prefix the current input vector to define a recurrent version of the state computation function:

$$f_s(\ \vec{s}(t-1),\ \vec{x}(t),\ W\ ) \equiv \vec{x}(t) \oplus \vec{s}(t-1)[1..6\cdot\|\vec{x}(t)\|]$$

Figure 2-2 illustrates a WIT memory. Each input symbol is assumed to be encoded as a four-dimensional vector represented by a rectangular box; the components of the vector are illustrated as circles. At time t, the input vector, $\vec{x}(t)$, which corresponds to the current input symbol, is added to the left side of the state vector, $\vec{s}(t)$. All other vectors are shifted right. It is critical to note that the equation to compute the next state of the STCN does not make use of either of the trainable parameters, W and $\|\vec{s}\|$. This means that the function remains completely static—the state vector is computed the same way every time.

The equations above considered only a window of width seven. Certainly other window widths could be used. If the width of the window is n, then the state of the STCN is given by:

$$\vec{s}(t) = \vec{x}(t) \oplus \vec{x}(t-1) \oplus \cdots \oplus \vec{x}(t-n+1)$$

and the function to recursively compute state is defined as:

$$f_s(\ \vec{s}(t-1),\ \vec{x}(t),\ W\ ) \equiv \vec{x}(t) \oplus \vec{s}(t-1)[1..(n-1)\cdot\|\vec{x}(t)\|]$$

For the case, $n=7$, these new equations reduce to the previous ones.

Connectionist Infinite Impulse Response Filter (CIIR) Memory

A variation on the Window in Time memory is to use two temporal windows. One window on the input symbols (as in WIT memories) and a second on the output symbols produced by the network. Because of its similarity to infinite impulse response filters (IIRs), this type of memory has been called neural network IIR, but we shall prefer the term connectionist IIR or CIIR, for short. Narendra and Parthasarathy (1990) have used this type of short-term memory. The state in this type of network is equivalent to:

$$\begin{aligned}\vec{s}(t) = &\ \{\ \vec{x}(t) \oplus \vec{x}(t-1) \oplus \cdots \oplus \vec{x}(t-n+1)\ \} \\ &\oplus \{\ \vec{y}(t-1) \oplus \vec{y}(t-2) \oplus \cdots \oplus \vec{y}(t-m)\ \}\end{aligned}$$

Clearly, the WIT memory described above is a special case of the CIIR memory where $m=0$ (i.e., the size of the output window is zero). CIIR memory and WIT memories are classified separately in this taxonomy because the more restrictive WIT memory has important representational limitations which are not shared by CIIR memory (these limitations will be discussed in the next chapter). The state function for the CIIR memory can be defined as the recurrent function:

$$\begin{aligned}f_s(\ \vec{s}(t-1),\ \vec{x}(t),\ W\ ) \equiv &\ \vec{x}(t) \oplus \vec{s}(t-1)[1..(n-1)\cdot\|\vec{x}(t)\|] \\ &\oplus \vec{y}(t-1) \oplus \vec{s}(t-1)[(n+1)\cdot\|\vec{x}(t)\|..(n+m-1)\cdot\|\vec{x}(t)\|]\end{aligned}$$

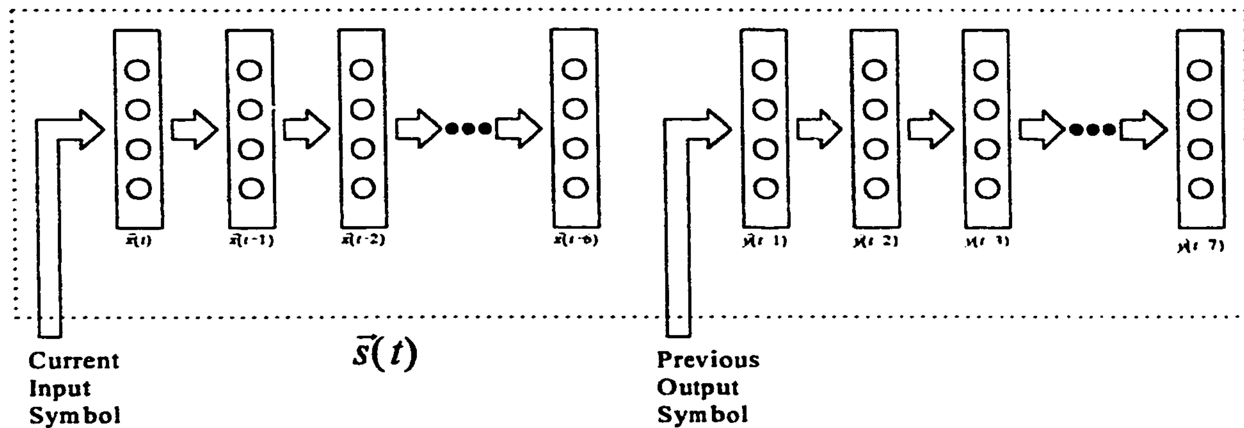

Figure 2-3: CIIR memory.

A CIIR memory with input window of length, $n=7$, and output window of length, $m=7$, is illustrated in Figure 2-3. This figure assumes that input and output symbol are encoded as four dimensional vectors, represented by rectangles containing four circles. The current input symbol is encoded in $\vec{x}(t)$, while the previous output symbol is encoded in $\vec{y}(t-1)$. At each time step the vectors are propagated to the right through the short-term memory. The concatenation of all seven old input and seven output vectors forms the short-term memory, $\vec{s}(t)$.

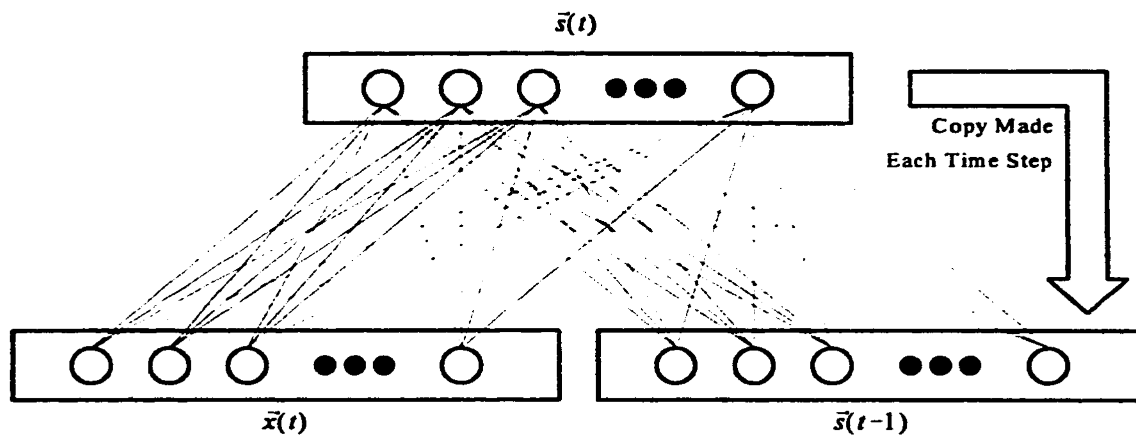

Figure 2-4: SLFOCC memory.

Single-Layer First-Order Context Computation (SLFOCC) Memory

Another approach to short-term memory, which has been widely used, is based on computing the STCN's state using a single-layer first-order feedforward network (Rumelhart, Hinton and Williams, 1986) which uses the previous state (also called context) and the current input symbol as input. This approach, abbreviated SLFOCC memory is used in Elman's (1990, 1991a) Simple Recurrent Networks (SRN), Pollack's (1989, 1990, 1994) Recurrent Auto-Associative Memory (RAAM), Maskara and Noetzel's (1992) Auto-Associative Recurrent Network, and Williams and Zipser's (1989) Real Time Recurrent Learning (RTRL) networks.

The SLFOCC short-term memory is stored in a set of hidden units whose activations are computed based on the activations in a layer of input units and a layer of context units, and on the weights from these two layers to the state units. By convention (see Elman, 1990), the activations of the context units are initially set to 0.5, and subsequently set to the activation values of the hidden units at the previous time step (the number of context and hidden units must be equal). Specifically, the hidden unit activation values are copied to the context units at each time step. Thus, at any given time, the

hidden unit activations represent the current state, while the context unit activations represent the previous state. Thus, the function used to compute the state vector is:

$$f_s(\ \vec{s}(t-1),\ \vec{x}(t),\ W\ ) \equiv \vec{\sigma}(\ W^1 \times \{1 \oplus \vec{x}(t) \oplus \vec{s}(t-1)\}\ )$$

where W^1 is a two-dimensional sub-matrix of W representing the weights of the connections in the first layer. As in Section 2.4.1, $\vec{\sigma}(\vec{x})$ represents the result of applying the function $\sigma(x) = \frac{1}{1 - e^{-x}}$ to each component of vector $\vec{x}$. The SLFOCC short-term memory is depicted in Figure 2-4.

Unlike WIT and CIIR memories, whose state vector is equivalent of a finite number of previous input and output vectors, SLFOCC memories use states based on the previous state vector and input vector. This implies that the state vector of this type of memory can contain information not found in recent input and output vectors.

Single-Layer Second-Order Context Computation (SLSOCC) Memory

A variation on SLFOCC memory involves using a single-layer **second-order** network to compute state based upon previous state and current input. Second-order connections are connections which connect three nodes (rather than just two), and are a restricted type of the Sigma-Pi units studied by Rumelhart, Hinton, Williams (1986). In these connections, the activation of one of the nodes is modulated by that of another, and transmitted as a signal to the third which computes a weighted sum of all of the products and passes the value through a squashing function. The processing performed by first-order and second-order connections is illustrated in Figure 2-5.

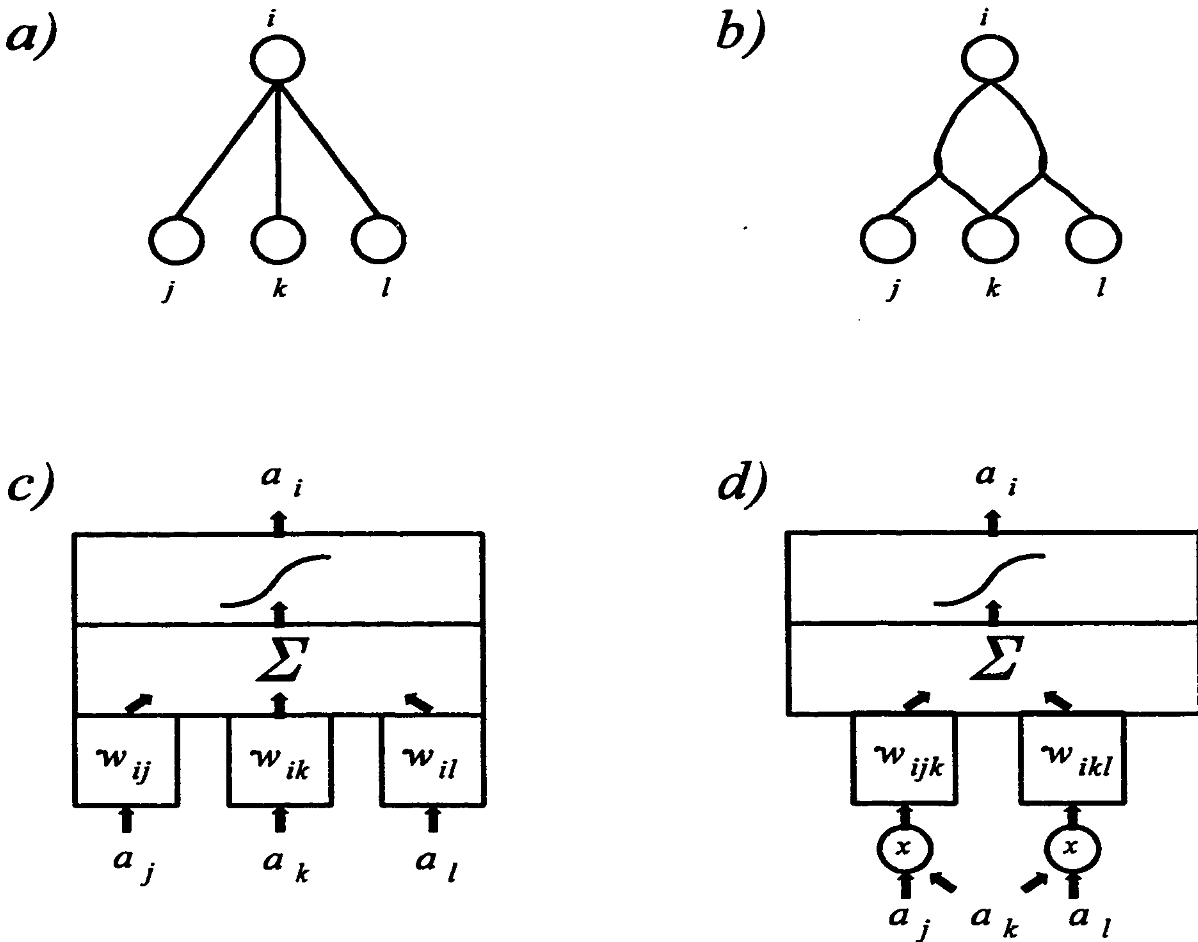

Figure 2-5: First order vs. second order connections. (a) Three first order connections from nodes *j, k, l,* to node *i.* (b) Two second order connections from *j & k, k & l,* to node *i.* (c) Diagram of computation performed by first order connections. (d) Diagram of computation performed by second order connections.

In the SLSOCC memory, every input unit is connected with every state unit to every other state unit via a second order connection with a temporal delay of one time step. In place of this complex recurrent network, it is simpler to consider an equivalent feed-forward network similar in style to the SLFOCC memory memory, described earlier. This simplified network consists of three layers: input units, state units, and context units. The context units in this network operate in a manner analogous to the SLFOCC

memory—that is, their activation values are equal to the activation values of the corresponding state units at the previous time step. In this network, second order connections then connect every possible pair of one input and one context unit to every state unit. Thus,

$$f_s(\ \vec{s}(t-1),\ \vec{x}(t),\ W\ ) \equiv \vec{\sigma}(\ (W^1 \times \vec{x}(t)) \times \vec{s}(t-1)\ )$$

where W^1 is a three-dimensional weight array component of W, and $W^1 \times \vec{x}(t)$ is a two dimensional matrix, M, whose components M_{ij} are computed:

$$M_{ij} = \sum_k W^I_{ijk} \cdot \vec{x}_k(t) .$$

Figure 2-6 depicts the SLSOCC short-term memory. Note that only the connections leading to one of the state nodes are depicted here. The remaining 48 connections leading into the other 3 state nodes in the diagram are not illustrated in the interest of clarity. SLSOCC memories have been used extensively by Giles et. al. (1991, 1992a, 1992b, 1992c), Goudreau et. al. (1994), Liu et. al. (1990), Miller and Giles (1993), Omlin et. al. (1994a, in press), Shaw (1990), Sun et. al. (1990, 1991), and Watrous and Kuhn (1992a, 1992b).

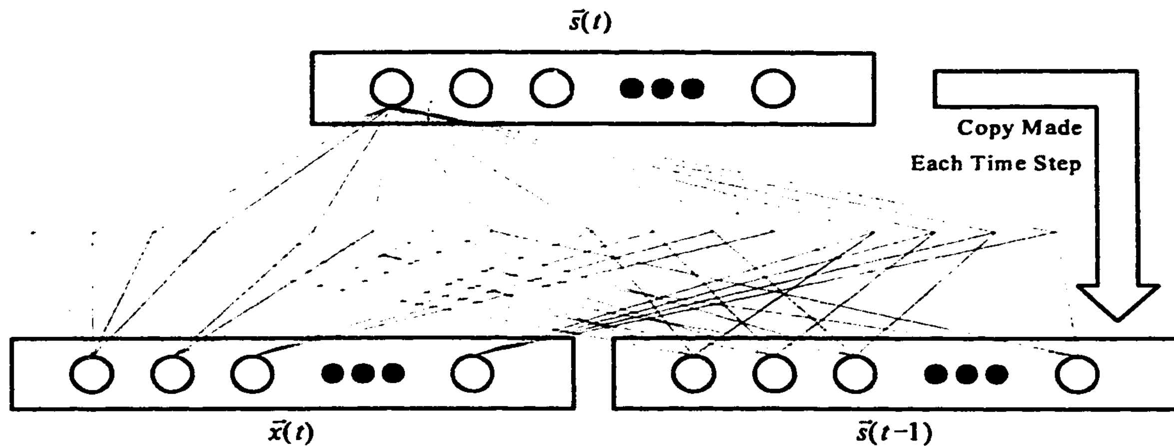

Figure 2-6: SLSOCC memory.

Connectionist Pushdown Automaton (CPA) Memory with Continuous Stack

Connectionist Pushdown Automaton (CPA) memories are radically different from the STCN memories that have previously been discussed. This is because in addition to storing state in the activations of nodes, these memories also store state on an external continuous stack. In this sense, they are like classical Pushdown Automata (Hopcroft and Ullman, 1979) whose state can be considered to be equivalent to the state of the finite controller and the contents of an unbounded stack. Hopcroft and Ullman refer to this type of extended state as an *instantaneous description* (*ID*).

While a conventional finite state controller can send three discrete signals to the stack (pop, push, no-action), the stack of a CPA memory must respond to continuous (numerical) signals received from the nodes in the connectionist controller. Similarly, while a conventional finite state controller retrieves discrete symbols from its stack, the connectionist controller of a CPA memory must be able to accept continuous values from its stack. These are requirements of the gradient descent learning algorithms used in STCNs which require a differentiable error function.

), the stack controller in a CPA memory has two additional node classes and read nodes) which are used to manipulate the external stack. There is ɔn node which is used to send signals (pop a symbol, push a symbol, no stack. The read nodes are used to encode the symbol at the top of the stack. ɜs also use third order connections instead of second order connections.

:ed above, the stack used by the CPA memory must be continuous in order dient descent learning. This is an indirect consequence of the fact that a omaton only accepts input strings if, after presentation of the entire string, mpty. Thus, the error function for a CPA memory must take into account :he stack for legal strings. Since the error function for any gradient descent st be continuous, the stack length of a CPA memory must also be represented ous value. This continuity is achieved by giving each symbol in the stack a ı and defining the stack length to be equal to the sum of the lengths of the ıe stack. Symbols are pushed onto or popped off of the stack based on the lue of the action node. This activation value can range from -1 to +1. ılues greater than a small constant ϵ are interpreted as a signal to push the symbol onto the stack, while activation values less than $-\epsilon$ are interpreted as and intermediate values are interpreted as no change in stack content.

a symbol (encoded as a vector) is pushed onto the stack, its length in the ned to be equal to the activation value of the action node. When a pop urs, the action node's activation value determines the length of stack which ince symbol lengths in the stack can vary from ϵ to 1, and deletions can vary it is possible that the pop operation will only remove part of a symbol from ducing the symbol's length), an entire symbol from the stack, or multiple

symbols from the stack. This seems odd, but is really only a consequence of the mechanism used to permit gradient descent learning. After CPA memories have been trained, push and pop operations typically become discretized.

The second additional node class consists of read nodes whose activation values are equal to the symbol encoded at the top of the stack. If the symbol at the top of the stack has a length of less than one, its activation vector is multiplied by its length and added to subsequent symbol vectors to form the read node activation values until a total length of 1 is read. In the CPA memory, the read nodes are treated as input nodes, while the action node is implemented as a special output node. Third order connections lead from every triple, consisting of an Input, State, and Read node, to every Output unit as well as the Action node.

For our purposes, the state vector $\vec{s}(t)$ will represent only the state of the stack controller and not the stack itself, since it is external to the network. Thus,

$$f_s(\ \vec{s}(t-1),\ \vec{x}(t),\ W\ ) = \vec{\sigma}(\ ((W^1 \times \vec{x}(t)) \times \vec{s}(t-1)) \times \vec{r}(t)\ )$$

where $\vec{r}(t)$ is the vector representing the symbol read from the stack at time t, W^1 is a four-dimensional sub array of W corresponding to the connection weights to the state units. Here, the multiplications are performed such that:

$$((W^1 \times \vec{x}(t)) \times \vec{s}(t-1)) \times \vec{r}(t) = \sum_l \sum_k \sum_j W^1_{ijkl} \cdot x_l(t) \cdot s_k(t-1) \cdot r_j(t)\,.$$

The CPA memory is illustrated in Figure 2-7. Note that only two of the (320) third order connections are shown in order to simplify the illustration. CPA memories have been studied by Das, Giles and Sun (1993), Giles, Sun, Chen, Lee and Chen (1990), Sun et.

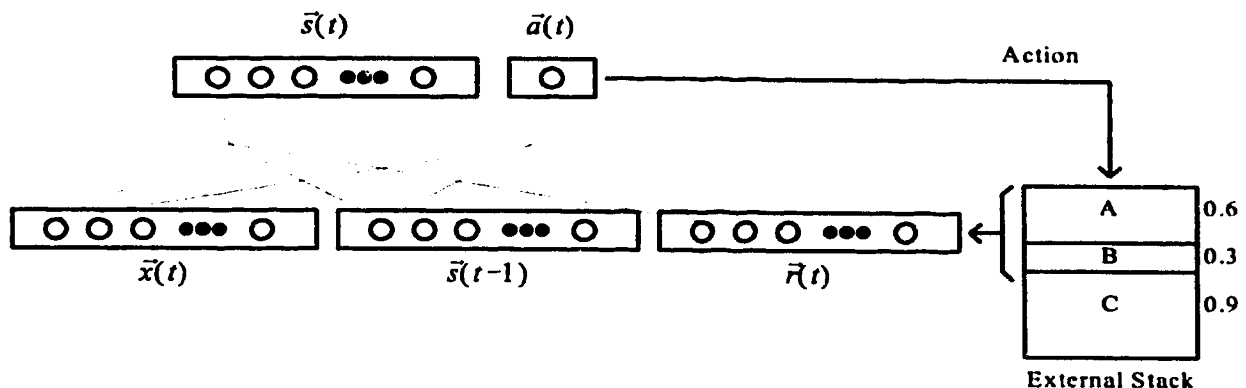

Figure 2-7: The CPA memory.

al. (1990a), Sun, Giles, Chen and Lee (1993), and Zeng, Goodman and Smyth (1994).

Connectionist Turing Machine (CTM) Memory

Williams and Zipser (1989) developed a Turing Machine connectionist memory (CTM) by replacing the finite state controller of a classical Turing Machine by a SLFOCC memory in order to design a STCN capable of universal computation. The connectionist network controls the operations performed on a standard Turing Machine tape (move left, no change, write "1", write "0", move right) based on the activations of five action neurons and reads the current symbol at the tape head into its Read Nodes.

This network was never trained to learn the tape operations. Instead, the tape operations were supplied to the network during training, so only the control mechanism needed to be learned. For this reason a continuous version of a tape was not required. The state vector for a CTM memory (which does not represent the external tape) is defined:

$$f_{\vec{s}}(\ \vec{s}(t-1),\ \vec{x}(t),\ W\ ) \equiv \vec{\sigma}(\ W \times \{\ 1 \oplus \vec{s}(t-1) \oplus \vec{r}(t)\ \}\ )$$

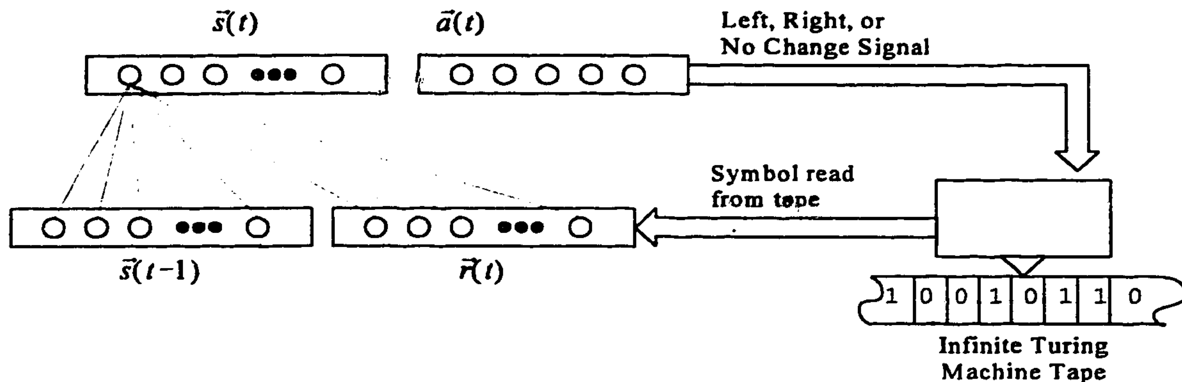

Figure 2-8: The CTM memory.

A CTM memory is illustrated in Figure 2-8. Only the connections to one of the state nodes are shown.

Locally Recurrent State and Input (LRSI) Memory

The Locally Recurrent State and Input (LRSI) memory is computed incrementally. That is, each component of the state vector depends on all lower numbered components for the same time step. The first components of the state vector are equivalent to the input vector. Additionally, each node receives a recurrent connection from itself with time delay of one unit. Since no node receives delayed connections from nodes other than itself, this represents local recurrence. It is the local recurrence which introduces temporal dependency. Note that unlike the SLFOCC and SLSOCC memories, which have delayed connections from other nodes, all temporal connections in an LRSI memory are strictly locally recurrent. This type of memory is used in Fahlman's Recurrent Cascade Correlation (RCC) which has been studied by Fahlman (1991), Giles et al. (1995) and Kremer (1995b; 1995c; in press).

Once again, we shall not show recurrent connections in our diagram, but rather an additional set of Context Units whose activations are equal to the state units at the previous time step. The computation of state vector in an LRSI memory is defined:

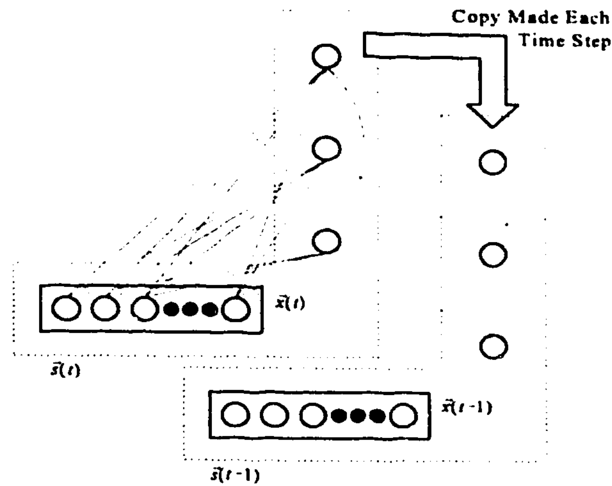

Figure 2-9: The LRSI memory.

$$f_{\vec{s}}(\ \vec{s}(t-1),\ \vec{x}(t),\ W\ )[i] \equiv \begin{cases} \vec{x}_i(t) & \forall i \leq \|\vec{x}\| \\ \vec{\sigma}\left(W^1_{\cdot,i} \times \left\{ 1 \oplus \vec{s}(t)[1..i-1] \oplus \vec{s}(t-1)[i] \right\} \right) & \text{otherwise} \end{cases}$$

where W^1 represents a submatrix of W containing the connections to the state nodes. The architecture which computes this state vector is depicted in Figure 2-9. Each component of the state vector is computed based on the values of all lower numbered components at the same time step and the value of the component itself at the previous time step.

2.5.2 Computing the Output Vector

We now examine the instantiations of the output vector function, $f_y(\vec{s}(t), W)$. In particular, we discuss the three leading alternatives: Zero Layer, One Layer, and Two Layer output functions.

Zero Layer (0-Layer)

The simplest possible manner in which to compute the output of an STCN given the state vector, is simply to use the state vector (or a portion thereof) as the output. For simplicity we assume that the first few components of the state vector represent the output. This simple way of computing STCN output can then be easily represented:

$$f_y(\vec{s}(t), W) \equiv \vec{s}(t)[1..\|\vec{y}\|]$$

This zero layer approach to output computation has been used by Giles et. al. (1991, 1992a, 1992b, 1992c), Goudreau et. al. (1994), Liu et. al. (1990), Miller and Giles (1993a), Omlin and Giles (1994a, in press), Shaw (1990), Sun et. al. (1990b, 1991), Watrous and Kuhn (1992a, 1992b), Das, et al. (1993), Giles et. al. (1990), Sun et. al. (1990a, 1993), Zeng, et al. (1994) and Williams and Zipser (1989). It is illustrated in Figure 2-10(a).

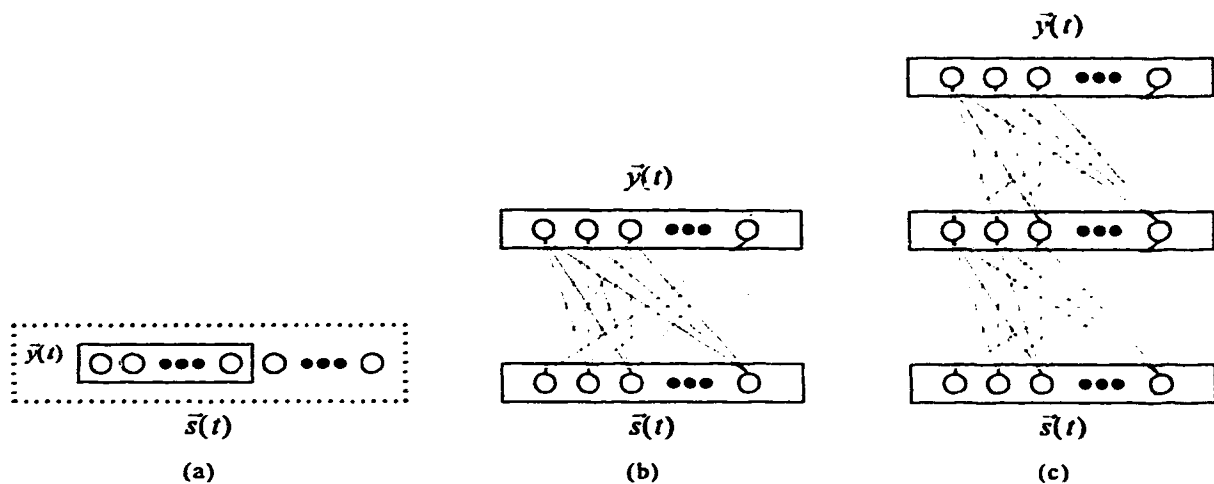

Figure 2-10: Output functions of STCNs. (a) Zero Layer. (b) One Layer. (c) Two Layer.

One Layer (1-Layer)

The next output function, increasing in complexity, computes the output vector using an additional layer of nodes. These nodes are massively parallelly connected to all the nodes that form the state vector, giving an output function defined by:

$$f_y(\vec{s}(t), W) \equiv \vec{\sigma}(W^2 \times \{1 \oplus \vec{s}(t)\}).$$

Where W^2 is a matrix of connection weights between the state and output nodes (W^2 is a submatrix of W). This one layer approach to output computation has been used in Elman's SRNs (1988, 1990), Pollack's RAAM (1989, 1990), Maskara and Noetzel's AARNs (1992) and Fahlman's RCC (1991), and is illustrated in Figure 2-10(b).

Two Layer (2-Layer)

Of course, the logical next step is to use a two layer system where:

$$f_y(\ \vec{s}(t),\ W\ ) \equiv \vec{\sigma}(\ W^3 \times \vec{\sigma}(\ W^2 \times \vec{s}(t)\ )\ ),$$

and W^2 and W^3 are submatrices of W representing the weights to the second and third layers of connections respectively. This approach to computing output is employed by NETtalk (Sejnowski and Rosenberg, 1986) and by Narendra and Parthasarathy (1990), and is illustrated in Figure 2-10(c).

Note that we assume here, that "end" symbols used to mark the end of sample sentences during training and operation, are part of the language. If STCNs are used to render grammaticality judgements, and if end symbols are used, these network could be interpreted as having an additional layer of processors in the output layer, since during processing of the end symbol, the last symbol of the string goes through an extra step of processing. For the remainder of this dissertation, we shall assume that end symbols, if supplied, are part of the grammar which is to be induced.

2.5.3 Computing the Weight Change

All of the weight change functions described here and in the STCN literature are based on the gradient descent algorithm: changes are made in proportion to the negative of the error gradient in weight space according to the equation:

$$\Delta W \equiv -c \cdot \nabla \varepsilon$$

where c is a small, real-valued constant, and ε is the total network error defined in Section 2.2.6. Expanding ε gives:

$$\Delta W \equiv -c \cdot \nabla \left\{ \sum_t \left\{ \frac{1}{2} \|\vec{E}(t)\|^2 \right\} \right\}.$$

In conventional connectionist learning tasks, it is computationally convenient to evaluate this function piecewise over time, by making weight changes after presenting each input pattern. For grammatical induction problems, this type of evaluation becomes a necessity since there are a potentially infinite number of sample strings, and hence it would be impossible to wait until all had been presented before adjusting the weights. Thus, weight changes are made according to the formula:

$$\Delta W \equiv \sum_t \Delta W(t),$$

where:

$$\Delta W(t) \equiv -c \cdot \nabla \left\{ \frac{1}{2} \|\vec{E}(t)\|^2 \right\}.$$

Note that, in the limit as c approaches $1/\infty$, the two definitions for weight change are equivalent. However, for larger values of c the value of the gradient will change between time steps and thus result in differences between the two definitions. The implications of these differences will be described later (see Section 6.2.3). We now turn our attention to evaluating the gradient:

$$\nabla\{\frac{1}{2}\|\vec{E}(t)\|^2\} = (\sum_{i,j} I_{i,j} \frac{\partial}{\partial W_{i,j}}) \frac{1}{2}\sum_k \vec{E}_k(t)^2,$$

where $I_{i,j}$ represents a matrix whose components are all 0 except for the $(i,j)^{th}$ component whose value is 1, and where $W_{i,j}$ and $\vec{E}_k$ represent the $(i,j)^{th}$ and k^{th} components of W and $\vec{E}$ respectively. This equation can be simplified to:

$$\nabla\{\frac{1}{2}\|\vec{E}(t)\|^2\} = \sum_{i,j} I_{i,j} \sum_k \frac{\partial \vec{E}_k(t)}{\partial W_{i,j}},$$

and further to:

$$\nabla\{\frac{1}{2}\|\vec{E}(t)\|^2\} = \sum_{ij} I_{ij} \sum_k (\vec{y}_k^{\,*}(t)-\vec{y}_k(t))\frac{\partial \vec{y}_k(t)}{\partial W_{ij}}.$$

The weight change functions discussed below differ in how they compute the final gradient in this equation.

Full Gradient Descent (FGD)

The calculation of the final gradient can be computed by expanding the numerator to its maximum extent (using the function which computes state) and then computing the gradient of the resulting expression:

$$\frac{\partial \vec{y}_k(t)}{\partial W_{ij}} = \frac{\partial}{\partial W_{ij}} f_y(f_s(f_s(f_s(\ldots f_s(\vec{s}(0), \vec{x}(0), W) \ldots , \vec{x}(t-2), W), \vec{x}(t-1), W), \vec{x}(t), W), W)_k .$$

If the state function does not use the value of W (i.e. WIT memory) the computation of this derivative is quite simple since it reduces to the computation of the weight gradient in error space in a non-recurrent network.

If the state function does use the value W, then the computation of this derivative is much more complicated due to its recurrent nature. A number of different algorithms have been suggested to perform the computation in an efficient manner. Typically, the approaches trade space complexity for time complexity. Since all of the different algorithms compute the same result, and since we will not be doing a space or time complexity analysis of the learning algorithms used by STCNs, we will not discuss differences in implementations of full gradient descent and refer the interested reader to Rumelhart, Hinton and Williams (1986) for a description of back-propagation through

time (BPTT), Williams and Zipser (1988, 1989) for a description of real-time recurrent learning (RTRL) and Schmidhuber (1992) for a BPTT/RTRL hybrid approach. Full gradient descent has been used by Sejnowski and Rosenberg (1986), Giles et al. (1991, 1992a, 1992b, 1992c), Goudreau et al. (1994), Liu et al. (1990), Miller and Giles (1993), Omlin et al. (1994a, in press), Shaw (1990), Sun et al. (1990b, 1991), Watrous and Kuhn (1992a, 1992b), Das, et al. (1993), Giles et al. (1990), Sun et al. (1990a, 1993), Zeng et al. (1994) and Williams and Zipser (1989). The equation for weight change for full gradient descent is:

$$f_{\Delta W}(\ \vec{E}(t),\ W,\ \vec{x}(\cdot)\ ) = \sum_{i,j} I_{i,j} \sum_{k} (\vec{y}_k^{\,*}(t) - \vec{y}_k(t)) \frac{\partial}{\partial W_{i,j}} f_y(\ f_s(\ f_s(\ f_s(\ \ldots \\ f_s(\ \vec{s}(0),\ \vec{x}(0),\ W\ )\ \ldots\ , \vec{x}(t-2),\ W\ ),\ \ \vec{x}(t-1),\ W\ ),\ \ \vec{x}(t),\ W\ ),\ \ W\ )\ .$$

Teacher Forcing (TF)

If a STCN's state (or part thereof) is equivalent to the STCN's previous output, then a simplification of full gradient descent, called teacher forcing, can be used. CIIR memories use a state based on output. Also, any STCN which uses a zero-layer output function uses part of the output vector as its state vector. In either of these cases, it is possible to substitute the value of the desired output, $\vec{y}^{\,*}(t)$, for the value of the actual output, $\vec{y}(t)$, whenever the latter occurs in the state vector's computation. This can eliminate a great deal of the recurrence in the equation and thus simplify computation. The equation for teacher forced weight update is:

$$f_{\Delta W}(\ \vec{E}(t),\ W,\ \vec{x}(\cdot)\ ) \equiv \sum_{i,j} I_{i,j} \sum_{k} (\vec{y}_k^{\,*}(t) - \vec{y}_k(t)) \frac{\partial}{\partial W_{i,j}} f_y(\ f_s(\ f_s(\ f_s(\\ \ldots f_s(\ \vec{s}(0),\ \vec{x}(0),\ W\ )\big|^{\vec{y}(0)=\vec{y}^{\,*}(0)}\ \ldots\ , \\ \vec{x}(t-2),\ W\ )\big|^{\vec{y}(t-2)=\vec{y}^{\,*}(t-2)}, \\ \vec{x}(t-1),\ W\ )\big|^{\vec{y}(t-1)=\vec{y}^{\,*}(t-1)}, \\ \vec{x}(t),\ W\ )\big|^{\vec{y}(t)=\vec{y}^{\,*}(t)},\ \ W\ )\ .$$

Intuitively, this is equivalent to assuming that the network has already learned to produce the desired output for all previous time-steps when computing the weight change for the current time step. This approach has been used by Williams and Zipser (1989), and Narendra and Parthasarathy (1990).

Truncated Gradient Descent (TGD)

Another approach to simplifying the gradient computation is to not completely expand the equation to compute output, prior to computing the derivative. This is referred to as truncated gradient descent. In this method the current state is evaluated, but the previous state is not. This results in the following weight update equation:

$$f_{\Delta W}(\ \vec{E}(t),\ W,\ \vec{x}(\cdot)\ ) \equiv \sum_{ij} I_{ij} \sum_{k} (\vec{y}^{*}_{k}(t) - \vec{y}_{k}(t)) \frac{\partial}{\partial W_{ij}} f_{\vec{y}}(\ f_{\vec{s}}(\ \vec{s}(t-1),\ \ \vec{x}(t),\ W\ ),\ W\ ) .$$

Since this equation is only an approximation to the true gradient, an STCN using this technique is not guaranteed to follow the true gradient in its search for a local minimum. Further, a local minimum in the error space defined by this function may not correspond to a local minimum in the original error space. This form of weight adjustment has been used by Elman (1988, 1991a), Cleeremans, Servan-Schreiber and McClelland (1989), Servan-Schreiber, Cleeremans and McClelland (1988, 1989, 1991).

Auto-Associative Gradient Descent (AAGD)

By definition, the memory vector of an STCN must contain some information about previous inputs. This implies that the weights in the network should be adapted to preserve input information in the memory vector. One way of accomplishing this task is to explicitly force the state to encode the input and the previous state by using an encoder network. An encoder network is a non-recurrent network consisting of an input layer, a hidden layer and an output layer. The input and output layers are of equal size while the

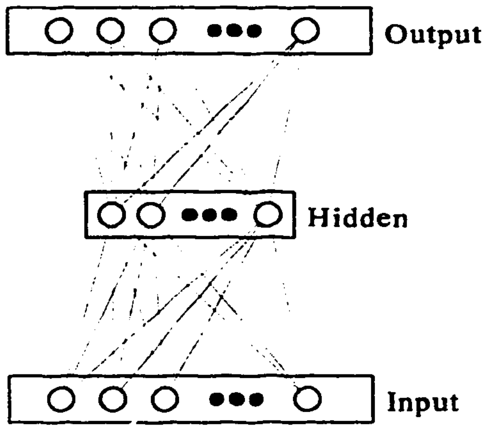

Figure 2-11: An encoder network.

hidden layer is smaller (see Figure 2-11). The target vector for the output units is always equivalent to the current input vector. By training the output layer to reproduce the input, the smaller hidden layer adopts a compressed representation of the input.

Auto-associative gradient descent weight update works by creating an additional set of "extraction" units. These units are connected to the state units and are trained to reproduce the previous state. It is the difference between the activations of the "extraction" units and the activations of the input units which defines the error which is minimized by gradient descent. By training in this fashion, the state computation becomes the first step of an encoder network and is thus trained to encode the previous state as a compressed encoding of input and previous state. Once training is complete, the extraction units can be removed, leaving a network which encodes state. If the vector $\vec{z}(t)$ represents the activations of the "extraction units", then the weight update equation is:

$$f_{\Delta W}(\ \vec{E}(t),\ W,\ \vec{x}(\cdot)\ ) \equiv$$
$$\sum_{i,j} I_{i,j} \sum_k (\vec{x}_k(t) - \vec{z}_k(t)) \frac{\partial}{\partial W_{i,j}} f(\ W^{z \times f_y}(\ f_s(\ \vec{s}(t-1),\ \vec{x}(t),\ W\ ),\ W\ ))'$$

where W^z is an additional matrix of weights which is simultaneously trained according to the equation:

$$\Delta W^z \equiv \sum_{i,j} I_{i,j} \sum_k (\vec{x}_k(t) - \vec{z}_k(t)) \frac{\partial}{\partial W^z_{i,j}} \vec{\sigma}(W^z \times \vec{y}(t)) .$$

AAGD weight update is used by Maskara and Noetzel (1992), and Pollack (1989, 1990).

Stack Learning (SL)

Adjustment of weights in a STCN with a continuous stack (see Section 2.5.1 above) can be accomplished by means of a full gradient descent procedure. In order to incorporate the use of the stack an additional error term is added to the conventional Euclidean error measure. This additional term is equal to the square of the stack length at the end of a string, l, and is only added for legal strings. This implies that after processing any legal string, the stack should be empty. The length of the stack is equal to the sum of the activation values of the action node throughout the string, so it is possible to minimize error, by back-propagating the error from the action node. Then,

$$f_{\Delta W}(\vec{E}(t), W, \vec{x}(\cdot)) = -\frac{1}{2} c \cdot \nabla(\|\vec{E}(t)\|^2 + l^2) .$$

Stack Learning has been studied by Das et al. (1993), Giles et al. (1990), Sun et al. (1990a, 1993), and Zeng et al. (1994).

Other Work Describing Weight Change Functions

For an excellent discussion of weight update functions in the context of non-linear adaptive filtering, the reader is referred to Nerrand, Roussel-Ragot, Personnaz, Dreyfus, and Marcos (1993).

2.5.4 Computing the Change in State Vector Size

Finally, we turn our attention to the process which adjusts the size of the state vector.

Manual Architecture Changes (MAC)

By far the most common approach to adjusting state vector size to a given problem is a manual trial-and-error procedure. That is, some heuristic is used to guess a suitable number of state nodes, and only if those nodes are unable to learn the given problem, are more added. No attempt is made to find a minimal number of nodes. Note that the new architecture does not reuse any of the weight parameters of the previous architecture; all weight values are re-initialized when the state vector size is changed. (In Chapter IV, we will explore the relationship between the number of state nodes and the computational power of a network. This will serve as a guide to choosing an initial network size.)

Automatically Incrementing Nodes (AIN)

The inadequacy of the trial-and-error method described led Fahlman (1991) and Giles et al. (1995) to devise automatic incremental architecture adjustment techniques. These techniques create new processing units in the network to represent memory as required. If the network is unable to learn by weight adjustment, then new state nodes are individually added. Each state node is connected to compute the desired state function used by the STCN. Unlike the trial and error approach, existing weights are maintained,

while new nodes to and from the new node are initialized to small values. In this sense, the short-term memory is increased until it is large enough to solve the problem. Figure 2-12 illustrates the growth of the memory in a LRSI memory, growth in other types of memory can be done similarly.

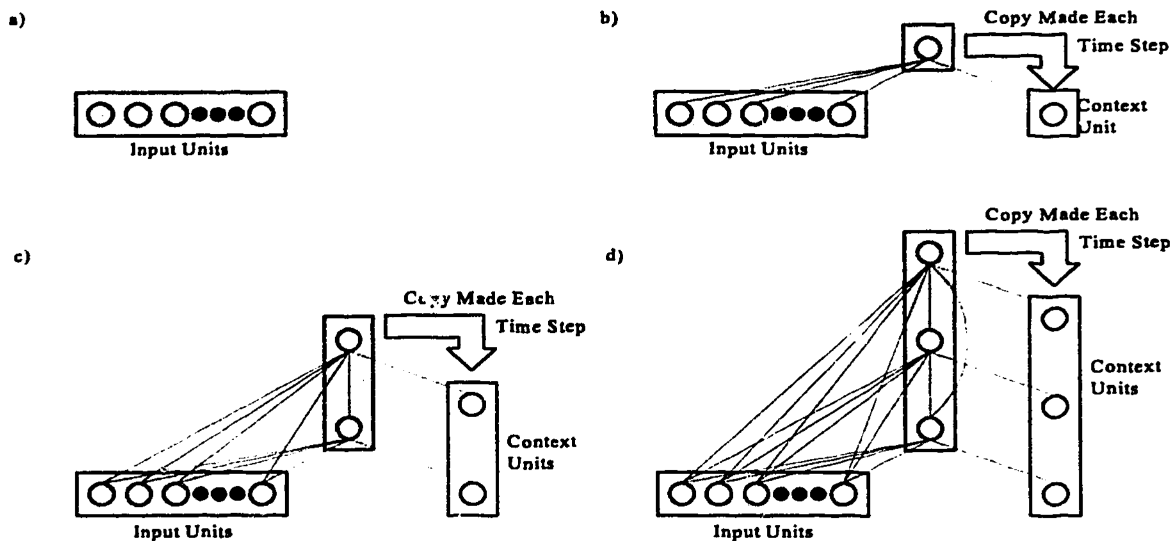

Figure 2-12: Adding nodes to a LRSI Memory. (a) Before any nodes are added. (b) After one node is added. (c) After two nodes are added. (d) After the addition of a third node.

2.6 SPECIFIC DESIGNS

We have now developed a relatively simple taxonomy based on four dimensions: (1) how the state vector is computed, $f_{\vec{s}}(\ \vec{s}(t-1),\ \vec{x}(t),\ W\ )$, (2) how the output vector is computed, $f_{\vec{y}}(\ \vec{s}(t),\ W\ )$, (3) how the change in weights is computed, $f_{\Delta W}(\ \vec{E}(t),\ W,\ \vec{x}(\cdot)\ )$, and (4) how the change in state vector size is determined, $f_{\Delta d(\vec{s})}(\ \vec{E}(t)\ )$. We also identified points along each of the dimensions. For the state vector computation, we presented seven alternatives: Window In Time (WIT) memories, Connectionist Infinite Impulse Response Filter (CIIR) memories, Single-Layer First-Order Context Computation (SLFOCC) memories, Single-Layer Second-Order Context Computation (SLSOCC) memories, Connectionist Pushdown Automata (CPA) memories, Connectionist Turing Machine (CTM) memories, and Locally Recurrent State and Input (LRSI) memories. For the output vector computation, we identified three specific functions: Zero Layer, One Layer, and Two Layer. We also formally described five methods for updating the weights in a STCN: Full Gradient Descent (FGD), Teacher Forcing (TF), Truncated Gradient Descent

(TGD), Auto-Associative Gradient Descent (AAGD), Stack Learning (SL). Finally, we identified two methods for computing the state vector size: Manual Architecture Changes (MAC) and Automatically Incrementing Nodes (AIN).

Since the dimensions are independent, it is possible to implement a STCN using any one of the seven state vector functions, any one of the three output functions, any one of the five weight update functions, and any one of the two state vector size functions. This yields a total of $(7\times3\times5\times2)$ 210 possible STCN designs. Of these designs, eleven have been reported in the literature. We now present these specific STCNs in a table indicating where along each of the four dimensions they lie. The STCNs in the table represent the leading approaches in the literature. There are clearly far more STCNs that could be added to such a table, but our goal is not to enumerate all STCNs. Instead, we present this table so that readers familiar with the more common architectures can quickly identify their place in the taxonomy and thus become more familiar with the taxonomy itself. If necessary any given STCN could be added to this table by answering the four design decision questions described in Section 2.5.

Because of its ability to describe all leading STCN approaches to grammatical induction, this taxonomy exhibits descriptive adequacy. Table 2-7 depicts the relationships between a number of STCNs. Those STCNs which share points along one or more of the dimensions answer one or more of the fundamental design decision questions in the same manner. In the following chapters, we will discover that the answers to the design decisions have important implications on the types of problems to which a grammatical induction system can be applied. Thus, those STCNs having matching entires under each of the dimension headings in the table will also have similar effectiveness for various grammar induction tasks. In this sense, the new taxonomy exhibits predictive power.

Since only eleven of the 210 possible STCN designs have been studied so far, the taxonomy also identifies 199 new STCNs which have not been previously proposed. By

focusing the consequences of the four basic design decisions which must be made for any grammatical induction system, the analysis in the following chapters provides insights into how these as yet unimplemented designs might perform for various types of problems. The ability to perform these types of formal analyses is another example of the predictive power of the new taxonomy.

Network	*Reference*	*State Function*	*Output Function*	*Weight Change Function*	*Architecture Change Function*
Window in Time	Sejnowski & Rosenberg (1986)	WIT	2-Layer	FGD	MAC
Neural Network Infinite Impulse Response Filter	Narendra & Parthasarathy (1990)	CIIR	2-Layer	TF	MAC
Simple Recurrent Network	Elman (1989, 1990)	SLFOCC	1-Layer	TGD	MAC
Recursive Auto-associative Memory	Pollack (1989, 1990)	SLFOCC	1-Layer	AAGD	MAC
Auto-associative Recurrent Network	Maskara & Noetzel (1992)	SLFOCC	1-Layer	AAGD	MAC
Real Time Recurrent Learning	Williams & Zipser (1989)	SLFOCC	0-Layer	FGD	MAC
Second Order Network	Giles et. al. (1991)	SLSOCC	0-Layer	FGD	MAC
Recurrent Cascade Correlation	Fahlman (1991)	LRSI	1-Layer	FGD	AIN
Connectionist Pushdown Automaton	Giles et. al. (1990)	CPA	0-Layer	SL	MAC
Connectionist Turing Machine	Williams & Zipser (1989)	CTM	0-Layer	FGD	MAC
Second Order Constructive Learning	Giles et. al. (1995)	SLSOCC	0-Layer	FGD	AIN

Table 2-7: Specific STCN Designs.

2.7 CONCLUSIONS

In this chapter we described the architectures and operation of a wide variety of spatio-temporal connectionist networks (STCNs). Rather than adopting an architecture-by-architecture approach, we described the networks by developing a taxonomy describing four fundamental design decisions. This permitted us to present many architectures more efficiently since there is considerable overlap in design decisions across different networks.

Prior to developing the taxonomy we identified three properties that make taxonomies effective. The properties are: descriptive adequacy, simplicity, and predictive power. We examined three taxonomies developed by Mozer (1993), Horne and Giles (1994), and Tsoi and Back (1994), and found that the recent flurry of research into STCNs has made them obsolete in terms of both predictive power and descriptive adequacy.

In response, we developed a new taxonomy based on four dimensions: how the state vector is computed, how the output vector is computed, how the change in weights is computed and how the change in state vector size is determined. These four dimensions were chosen because they represent the fundamental design decisions that any STCN grammatical induction system must address.

Next we identified specific points along these four dimensions. Along the state-vector dimension we identified: Window In Time (WIT) memories, Connectionist Infinite Impulse Response filter (CIIR) memories, Single-Layer First-Order Context Computation (SLFOCC) memories, Single-Layer Second-Order context computation (SLSOCC) memories, Connectionist Pushdown Automaton (CPA) memories, Connectionist Turing Machine (CTM) memories and Locally Recurrent State and Input (LRSI) memories. Along the output dimension we identified Zero Layer, One Layer and Two Layer functions. On the weight-change dimension we defined Full Gradient Descent (FGD), Teacher Forcing (TF), Truncated Gradient Descent (TGD), Auto-Associative Gradient Descent (AAGD) and Stack Learning (SL). Finally, we described Manual Architecture

Changes (MAC) and Automatically Incrementing Nodes (AIN) as points on the change-in-state-vector-size dimension. With each point on each dimension we referenced some of the networks which employed each approach.

Finally, we presented a table summarizing where along each dimension the major STCN approaches lie. Because the new taxonomy is the only one developed around the fundamental design decisions which must be addressed by any grammatical induction system, it has superior predictive power when used to compare and analyse how different STCNs might perform on various grammatical induction problems. Furthermore, the fact that the taxonomy is based around the principles of grammatical induction systems, rather than specific existing STCN designs, implies that it will easily accommodate future STCN designs as well. In the following chapters, we shall re-examine the points along the dimensions of our taxonomy in relation to the problem domain of grammatical induction. This will allow us to draw conclusions and make predictions about the performance of many different existing networks as well as future designs on grammatical induction problems based on their positions within the taxonomy.

Chapter III: The Problems of Grammatical Induction

3.1 INTRODUCTION

This chapter provides a formal description of the grammatical induction problem from the perspective of a search through an hypothesis space. It defines much of the terminology for the chapters which follow, and describes, in general, how STCNs can be applied to grammatical induction. Additionally, this chapter identifies the inherent difficulties of the problem and the techniques which have been proposed to overcome these difficulties. By providing a comprehensive survey of grammatical induction results relevant to the design of new grammatical induction systems, this chapter identifies the important issues that connectionist approaches to the problem must address. In the past, the designers of connectionist grammar induction systems have typically relied on empirical data to develop or select STCN for various problems. By contrast, this chapter

identifies important theoretical principles which can be applied to connectionist grammar induction systems to make the induction problem tractable and efficient.

This chapter is organized as follows. Section 3.2 examines grammatical induction from the perspective of a search algorithm. Section 3.3 identifies the nature of the algorithm's search space by defining formal languages, grammars and computing machines. Section 3.4 present's Chomsky's hierarchy of languages, grammars and machines. Section 3.5 relates connectionist grammar induction systems to the symbolic systems which the hierarchy was designed to classify. Section 3.6 compares the hypothesis spaces of connectionist induction systems to their symbolic counterparts. And, Section 3.7 examines the inherent difficulties of grammar induction.

3.2 GRAMMATICAL INDUCTION AS SEARCH

It has been argued that all of the problems of Artificial Intelligence (AI) can be solved within a search paradigm: "heuristic search, based on a sound understanding of the underlying structure of the problem domain forms the core of A.I. programs" (Rich, 1983). "Problem solving takes place by search in the problem space—i.e., by considering one knowledge state after another until (if the search is successful) a desired state is reached" (Newell and Simon, 1972, p.811). Under this interpretation, any task in AI can be reduced to a three step process. First, define a hypothesis space containing all possible configurations of the relevant objects. Second, identify a decision procedure for determining whether an arbitrary point in the hypothesis space constitutes the goal of the task. And, third, define a procedure for traversing the hypothesis space until a goal state is discovered.

If one adopts this view, then the domain of AI commonly referred to as game playing becomes a search of the space of possible game moves for the optimal move, given the current state of the board. Theorem proving becomes a search for a path from a set

of axioms to a lemma which is to be proven. Planning becomes a search for a set of actions that will transform the current state of the world into a desired state.

Grammatical induction can also be described in the context of search. However, this task is complicated by the fact that evidence accumulates over time. That is, throughout the induction process new information (typically strings labelled as either legal or illegal) is continually supplied to a classification mechanism which in turn is continually required to produce a "best guess" for the source grammar, consistent with the entire history of information presentations.

When interpreting grammatical induction as search, the *hypothesis space* is the class of all possible grammars which can be represented and therefore learned. In order to be successful, a grammatical induction system should eventually converge to a single grammar in this hypothesis space which is consistent with all input string presentations. This single consistent grammar represents the *goal* of the search. The *traversal algorithm* takes the form of some enumeration process which explores possible grammars sequentially until the goal grammar is found. This defines the hypothesis space, goal detection procedure and space traversal algorithm for the grammatical induction problem. Gold (1967) termed this task *language identification in the limit*.

Since this thesis uses spatio-temporal connectionist networks (STCNs) as an implementation mechanism for grammars, when we view grammatical induction as search, our hypothesis space is the set of connection weights which can be induced for a given STCN. In this chapter we show that the size of this space makes brute force search either impossible or intractable for all interesting problems, but suggest two alternatives: (1) reducing the size of the hypothesis space, and (2) adjusting the order in which the space is traversed (this corresponds to an heuristic search). Techniques to effectively implement both of these alternatives based on available a priori knowledge are presented and discussed.

3.3 LANGUAGES, GRAMMARS AND MACHINES

Before delving into the subject of grammatical induction in STCNs, we first examine some important theoretical results which apply to both symbolic and connectionist grammatical induction systems. Gold (1967) has shown that it is possible to perform some preliminary analyses of the complexity of the induction problem by examining only hypothesis spaces without discussing particular search algorithms. These analyses give some insight into the overall difficulty of grammatical induction and provide motivation for improved induction techniques which are the focus of this thesis. In order to understand the complexity of grammatical induction by STCNs, we must first describe some hypothesis spaces used by important classical (symbolic) grammatical induction algorithms. Then, we can compare the hypothesis spaces of STCNs to those employed in the symbolic paradigm. Having done this, it will then be possible to apply some of the conclusions reached by Gold to STCNs.

We adopt Hopcroft and Ullman's (1979, p. 2) definition of a language:

> An *alphabet* is a finite set of symbols. A *(formal) language* is a set of strings of symbols from some one alphabet.

The symbol Σ is commonly used to represent the alphabet. In the study of formal languages it is common to consider the simplest possible alphabets: e.g. $\Sigma = \{1\}$ or $\Sigma = \{0,1\}$. A simple language on the latter alphabet might be the language of all strings of even parity: $L = \{ \epsilon, 0, 00, 11, 000, 011, 101, 110, 0000, 0011, 0101, 0110, 1001, 1010, 1100, 1111, \ldots \}$. Here ϵ represents the null string. Note that this is an infinite language. That is, there are an infinite number of strings of even parity.

Grammars can now be viewed as descriptions of languages. As Pinker (19[illegible], p. 221) points out "it is a celebrated observation that natural and computer languages are infinite, even though they are used by beings with finite memory. Therefore the languages

must have some finite characterization, such as a recipe or program for specifying which sentences are in a given language." We shall thus use the following definition:

A *grammar* is a finite characterization of a potentially infinite language.

In particular, we shall use *Backus-Naur Form* (BNF) to represent all grammars. As per Hopcroft and Ullman (1979), a BNF grammar is a 4-tuple, $G=(V, T, P, S)$. Here, V represents a set of symbols, known as *variables,* which are used as intermediate results in the derivation of grammatical strings. Similarly, T represents a set of symbols, called *terminals*, which defines the alphabet of the language represented by the grammar. P is a finite set of rules, called *productions*, defining how strings of variables and terminals can be rewritten as other strings of variables and terminals in the process of deriving a grammatical string. Specifically, productions take the form $\alpha \rightarrow \beta$, where α and β are strings of symbols from the *Kleene closure* of the union of variables and terminals: $(V \cup T)^*$. Lastly, S is a special variable called the *start symbol.*

The process of deriving a legal string for a given grammar can now be formalized: First, the current string is initialized to be the start symbol. Second, strings of symbols within the current string matching the left-hand-side of one of the productions are replaced by the right-hand-side of the production. The second step is repeated until only terminal symbols remain in the current string, at which point, the current string represents a legal string. Formally, we define the rewrite operator, $\Rightarrow$, by asserting that $\gamma\alpha\delta \Rightarrow \gamma\beta\delta$ if and only if the production $\alpha \rightarrow \beta$ is a member of P. This operator represents one application of step two in the process above. Multiple applications can then be represented by the reflexive-transitive rewrite operator, $\overset{*}{\Rightarrow}$, which is defined as the reflexive and transitive closure of $\Rightarrow$. Applying the latter operator to the start symbol, the language described by the grammar, G, is defined as:

$$L(G) = \{ w \mid w \text{ is in } T^* \text{ and } S \overset{*}{\Rightarrow} w \}$$

The parity language presented above could be characterized according to the following re-write rules:

$S \rightarrow A$

$A \rightarrow \epsilon$

$A \rightarrow 0A$

$A \rightarrow 1B$

$B \rightarrow 0B$

$B \rightarrow 1A$

Having defined languages and grammars, we now define computing machines as follows:

> A *formal computing machine* is a logical mechanism which partitions strings of symbols from some alphabet into a finite number of categories.

For the parity example above, we can imagine a computational device with two states or categories which we shall call *even* and *odd*. The device shall start in the *even* state. As a string is presented to the device, one symbol at a time, the device switches state each time a 1 symbol is encountered, and remains in the same state each time a 0 symbol is encountered. This is illustrated in Figure 3-1. Clearly, after presenting the device with a string from the language L it will find itself in the *even* state, whereas for any string not in L, it will find itself in the *odd* state. If we label the *even* state as an accepting state (and the *odd* state as a rejecting state), then this device is an acceptor of the language L. Similarly, any *formal computing machine* which classifies strings according to whether or not they belong to a particular language, is said to be an acceptor for that language. Of course other types of computing machines with categories other than grammatical and non-grammatical are also possible.

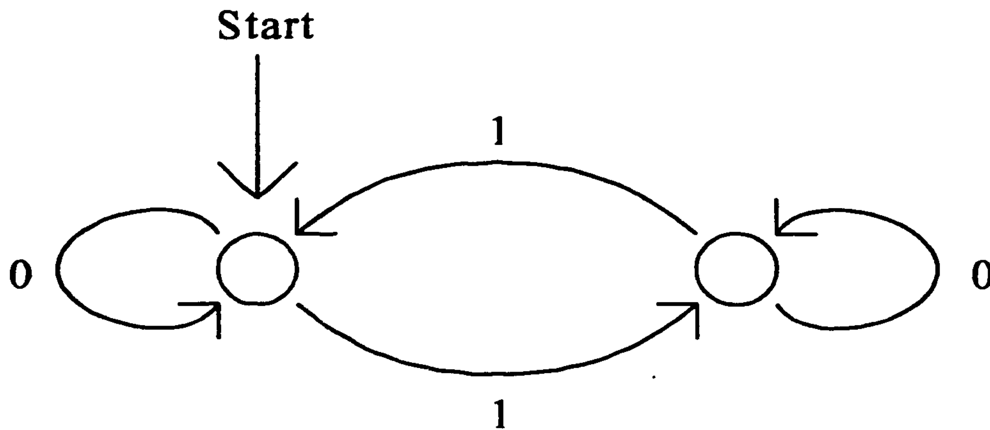

Figure 3-1: A logical mechanism to compute the parity of a string of 0's and 1's.

3.4 THE CHOMSKY HIERARCHY

The most influential person in the field of formal language theory has without a doubt been Noam Chomsky. In attempting to describe natural languages, Chomsky has suggested four broad classes of grammars. These are: the regular grammars (Chomsky type-3), the context free grammars (Chomsky type-2), the context sensitive grammars (Chomsky type-1) and the unrestricted grammars (Chomsky type-0). We now briefly describe three of these classes of grammars, the languages they describe, and the machines which act as acceptors for these languages. We omit the context sensitive grammars (Chomsky type-1) since these play no important role in the discussions which follow. The summary, here, is intended to serve as a brief introduction to the ideas and notation which will be used throughout this chapter. Readers requiring a more detailed discussion of automata theory, and formal grammars are referred to Hopcroft and Ullman (1979).

3.4.1 Regular Grammars, Regular Sets, Finite State Automata

The regular grammars represents the simplest class in the Chomsky hierarchy. All grammars in this class can be written in terms of rewrite rules in the form:

$A \rightarrow wB$ or $A \rightarrow w$

where A and B are non-terminal symbols, and w is a (possibly empty) string of terminals. The class of languages defined by regular grammars are called regular sets. A class of formal computing machines, called finite state automata, accept exactly the regular sets. A finite state automaton can be defined, in the notation of Hopcroft and Ullman (1979), by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$. Here, Q is a finite set of states, Σ is the alphabet of the language, δ is a transition function mapping $Q \times \Sigma$ to Q, $q_0 \in Q$ is the initial state and $F \subseteq Q$ is a set of accepting states. The automaton then renders grammaticality judgements as follows: First, the current state is set to q_0. Next, symbols are presented to the automaton, one at a time. The automaton uses the current state and the current input symbol as inputs to the transition function δ, and thus computes its next state. This process repeats until all symbols have been presented. Finally, the membership of the current state in the set of states, F, is determined. If the current state is in F, then the string is accepted, otherwise it is rejected.

The parity language, presented earlier, is accepted by a finite state automaton whose 5-tuple is given by the following values:

$$Q = \{ q_0, q_1 \}$$

$$\Sigma = \{ 0, 1 \}$$

$$\delta(q,s) = \begin{cases} q_0 & \text{if } (q \equiv q_0 \wedge s \equiv 0) \vee (q \equiv q_1 \wedge s \equiv 1) \\ q_1 & \text{if } (q \equiv q_0 \wedge s \equiv 1) \vee (q \equiv q_1 \wedge s \equiv 0) \end{cases}$$

$$F = \{ q_0 \}$$

Note that this automaton's behaviour corresponds to that previously illustrated in Figure 3-1.

3.4.2 Context Free Grammars, Context Free Languages, Pushdown Automata

The next class of grammars in the Chomsky hierarchy is the context free grammars (Hopcroft and Ullman, 1979). Grammars in this class can be represented by rewrite rules of the form:

$A \rightarrow \beta$

where A represents any single non-terminal and β represents any string of terminals and/or non-terminals. Clearly this definition implies that all regular grammars are context free (but not vice-versa). The form of the rewrite rules implies that for all non-terminal symbols, A, the way in which A is rewritten does not depend on any symbol occurring to the left of A. The languages described by context free grammars are context free languages. In order to accept or reject these languages, a more complicated device than a finite state automaton must be used. One such device is a Pushdown automaton.

A Pushdown automaton (Hopcroft and Ullman, 1979) is a finite state machine with an additional unbounded memory. This memory takes the form of a stack of symbols. Access to the stack is in a last-in first-out (LIFO) order. Formally, the Pushdown automaton can be represented by a 7-tuple: $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$. Here, once again, Q is a finite set of states, Σ is the alphabet of the language, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of accepting states. Additionally, Γ is the alphabet of symbols which are placed on the stack, $Z_0 \in \Gamma$ is the stack's initial symbol. Finally, δ is a mapping from $Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma$ to finite subsets of $Q \times \Gamma^*$.

The automaton then operates as follows: First, the current state is set to q_0 and the stack contents are set to Z_0. Then, the current state, q, the current input symbol, s, and the current symbol at the top of the stack, γ, are used as the parameters of the function δ. This function computes a set of state/stack-symbol-string pairs The automaton then selects any one of these pairs. The selected pair's state becomes the automatons next state, and the selected pair's stack-symbol-string is placed onto the stack (left-most symbol

highest). Note that if the function δ returns a non-empty set when ϵ is used as the input symbol, then the automaton can change state and add symbols to the stack without "consuming" input symbols. The computation of the function δ continues in this manner until all symbols have been used. Finally, the current state of the automaton is compared to the set F. If F contains the current state and the stack is empty, then the string is accepted.

Note that this is a non-deterministic process. The function δ can provide more than one possible next state which is selected by the automaton. This implies that grammatically judgements made by such a device vary from application to application. The overall grammaticality of a string is determined by examining whether or not the Pushdown automaton **ever** accepts the string. As Hopcroft and Ullman (1979, p. 107) point out, the equivalence of the languages accepted by Pushdown automata and defined by context free grammars "is somewhat less satisfactory [than the equivalence of the languages accepted by finite state automata and defined by regular grammars], since the pushdown automaton is a non-deterministic device, and the deterministic version accepts only a subset of all CFL's".

As an example of a context-free language we consider the language of all strings chosen from the alphabet {'(',')'}, in which the parenthesis are balanced: $L = \{ \epsilon, (), (()), ()(), ((())), (())(), ()(()), ()()(), \dots \}$. This language could be characterized by the following set of re-write rules:

$S \rightarrow A$,

$A \rightarrow \epsilon$,

$A \rightarrow (A)A$.

Note that, for all rules, the left-hand size is always a single non-terminal. It is also possible to define a pushdown automaton which accepts exactly this example language:

$Q = \{q_0, q_1\}$,

$\Sigma = \{\text{'('}, \text{')'}\}$,

$\Gamma = \{\text{'('}\}$,

$$\delta(q,s,\gamma) = \begin{cases} (q_0,\gamma'(') & \text{if } q=q_0 \wedge s='(' \\ (q_0,\varepsilon) & \text{if } q=q_0 \wedge s=')' \wedge \gamma='(' \\ (q_1,\varepsilon) & \text{if } q=q_0 \wedge s=')' \wedge \gamma=\varepsilon \\ (q_1,\gamma) & \text{if } q=q_1 \end{cases},$$

$Z_0 = \epsilon$,

$F = q_0$.

This pushdown automaton works as follows: States q_0, and q_1, represents accepting and rejecting states respectively (this is implied by the definitions of Σ, and F). The automaton starts in an accepting state (this follows from the definition of F). Case one of the definition of the transfer function, δ, implies that whenever an opening bracket is presented to the automaton, it is pushed onto the stack above whatever symbol happens to be there. Similarly, case two implies that whenever a closing bracket is presented, one of the opening brackets already on the stack is popped. If there are no opening brackets available on the stack the automaton enters the rejecting state (this is case three). Once in the rejecting state, the automaton never leaves it (case four).

3.4.3 Unrestricted Grammars, Recursively Enumerable Languages, Turing Machines

The final class of grammars in the Chomsky hierarchy is the class of unrestricted grammars (Hopcroft and Ullman, 1979). As the name implies, grammars in this class can be represented by rewrite rules of the form:

$\alpha \rightarrow \beta$

where α and β are arbitrary strings of terminals and/or non-terminals with the only provision being that $\alpha \neq \epsilon$. This implies that regular grammars and context-free grammars

are both subsets of the class of unrestricted grammars. The languages that are defined by unrestricted grammars are called recursively enumerable, and can be accepted by Turing machines (Turing, 1936). A Turing machine is a finite state automaton with an additional memory in the form of an infinitely long tape of connected cells into which symbols are written. The tape has a left-most cell, but no right-most cell. Unlike the automata previously discussed, the input symbols for a Turing machine are coded on the tape (as opposed to being presented to the machine by a specific input mechanism). Formally, a Turing machine can also be described by a 7-tuple: $(Q, \Sigma, \Gamma, \delta, q_0, B, F)$. Here, again, Q is a finite set of states, Σ is a set of input symbols, $q_0 \in Q$ is the start state, and $F \subseteq Q$ *I* is a set of accepting states. For the Turing machine, Γ is the set of allowable tape symbols; Σ must be a subset of Γ. The transition function δ is a mapping from $Q \times \Gamma$ to $Q \times \Gamma \times \{L,R\}$. The special symbol, B, represents a blank symbol, $B \subset \Gamma$, $B \not\subset \Sigma$.

The Turing machine operates as follows. First, the current state of the machine is set to equal q_0, and the string whose grammaticality is to be judged is written on the tape in consecutive cells starting with the left-most cell. The remaining infinity of cells on the tape all contain the blank, B, symbol. The tape-head of the machine is placed over the left-most cell. Then, the machine's current state and the symbol under the tape-head are used as parameters for the function δ. The function returns a state, a tape symbol and either an L or an R. The state represents the next state of the machine. The tape symbol is written into the cell under the tape-head (replacing the current symbol). Then, the tape head is moved one cell left or right as indicated by the L and R respectively. The function δ is continuously recomputed until the automaton enters an accepting state. This formulation implies that, for any legal string, the Turing machine will halt in an accepting state. Conversely, for illegal strings, the machine will never halt.

According to the Church-Turing hypothesis, Turing machines are believed to represent the ultimate in computational power. If this hypothesis is correct, then there can

Grammar	regular grammar, Chomsky type-3	context free grammar, Chomsky type-2	unrestricted grammars, Chomsky type-0
Rewrite rule description	all productions are of the form $A \rightarrow wB$ or $A \rightarrow w$ where A and B are variables and w is a (possibly empty) string of terminals	$A \rightarrow \beta$, where A is a single variable	$\alpha \rightarrow \beta$, where α and β are arbitrary strings of grammar symbols with $\alpha \neq \epsilon$
Language	regular sets	context-free languages	recursively enumerable languages
Machine	finite state automata	Pushdown automata	Turing Machines

Table 3-1: Grammars, languages and machines of the Chomsky hierarchy.

be no machines capable of rendering more powerful grammaticality judgements than a Turing machine. As for an example of a Turing machine, the reader is referred to Example 7.1 from Hopcroft and Ullman (1979). In the interest of brevity, we do not reproduce this example here. We have thus described a set of three mutually inclusive classes of grammars. These three grammars, their languages, and formal computing machines are summarized in Table 3-1.

3.4.4 Other Formalisms

It might seem strange that throughout this discussion, we have considered a language to be merely a set of legal strings. Many would argue that there is more to language than "mere" syntactic validity. For example there is the matter of attaching meaning to legal sentences. But, just like grouping strings as legal or illegal, attaching meaning can also be viewed as a type of categorization. In fact, it is possible to design a grammar in which only those strings with a certain meaning, say A, are considered legal. An automaton implementing this grammar would serve as an "A detector". If such a system were coupled with other automata to identify meanings B, C, D, etc., then the

result would be a system which assigns meaning to strings. Of course, rather than relegating these individual categorizations to separate computing devices, it might be more efficient to integrate them in the same device. This is the principle behind a Moore machine.

A Moore machine is a device whose output is not limited to a binary signal (accept/don't accept), but rather to an alphabet of output signals. Formally, a Moore machine can be represented by a six-tuple $(Q, \Sigma, \Delta, \delta, \lambda, q_0)$. As usual, Q is a finite set of states, Σ is the input alphabet, δ is the transition function mapping $Q \times \Sigma$ to Q, and q_0 is the start state. Additionally, Δ represents the output alphabet (or the categories into which all strings are classified), and λ is a mapping from Q to Δ which gives the output associated with each state. It is apparent that the formulation of a Moore machine is predicated upon the finite state automaton in the sense that both assume a finite number of internal states. It would of course be possible to similarly expand pushdown automata and Turing machines to devices which use a more complex output alphabet.

Since a Moore machine is not restricted to a binary output, it can perform more complex computations than the grammaticality judgements of *acceptors*. For example, a Moore machine can **generate** legal strings based on an enumerating input. Such a device is referred to as a *generator* by Gold (1967). Alternatively, a Moore machine is able to implement a mapping from the sequence of input symbols seen up to a particular point in time to an arbitrary output symbol at that time. Gold (1967) calls this type of device a *black box*. The use of a connectionist equivalent to a Moore machine is discussed in Section 6.4.

Another type of formal computing machine that uses an arbitrary output alphabet is a Mealy machine. Unlike a Moore machine, which computes its output based solely on its current state, a Mealy machine computes its output based on its previous state and the input symbol. Hopcroft and Ullman (1979, Theorem 2.7) have shown the equivalence of

Mealy and Moore machines. For this reason, and in the interest of consistency, we will not discuss Mealy machines (focusing instead on Moore machines) for the remainder of this thesis.

Another important class of computing machines which has not yet been discussed is stochastic automata. We delay discussion of these until Chapter VI where we will deal with STCNs that capture frequency information. Finally, we should note that many other types of formal computing devices have been proposed; we have discussed only a small subset of computational formalisms, here. A comprehensive discussion would lie beyond the scope of this dissertation.

3.5 SPATIO-TEMPORAL CONNECTIONIST NETWORKS AND THE CHOMSKY HIERARCHY

Classical machine induction systems are typically based on the Chomsky hierarchy in the sense that the hypothesis spaces searched by these systems represent grammatical classes from the hierarchy. Having identified the hypothesis spaces of classical machine induction systems, we must now consider how STCNs fit within the hierarchy of formal machines proposed by Chomsky.

3.5.1 Finite State Automaton Equivalence

It has long been known (Kleene, 1956; McCulloch and Pitts, 1943; Minsky, 1967) that networks of threshold units are capable of computing arbitrary boolean functions. If connected recurrently with delays, networks of such units are thus able to implement arbitrary finite state automata (and Moore machines). Since the logistic units used in modern connectionist networks are able to approximate threshold units to arbitrary degrees of precision, it must also be the case that STCNs are able to approximate finite state automata (and Moore machines). (A more formal proof of the ability of STCNs to

implement (rather than approximate) finite state automata (using a SLFOCC memory as an example) can be found in Section 4.4.2.) Thus, STCNs can be considered at least as powerful as finite state automata.

Note that, at this point, we are investigating only STCNs in general. Specific network designs must, of course, conform to certain connectivity schemes and node numbers. These restrictions can limit the computational power of these networks to subsets of finite state automata. The restrictions on particular network designs will be examined in Chapter IV.

3.5.2 Unbounded Memory

The next question to ask is whether or not STCNs are **more** powerful than finite state automata. The intuitive answer to this question is "no", since STCNs are simulated using digital computers with finite memory resources and thus could never represent the unbounded-length stack of a Pushdown automaton, or the unbounded-length tape of a Turing machine. Of course, the same argument can be applied to any digital computer. Yet, Hopcroft and Ullman (1979, p. 147) point out that "the Turing machine is equivalent in computing power of the digital computer as we know it today" (Hopcroft and Ullman, 1979, p. 147). This apparent contradiction arrises due to the fact that the notion of a digital computer does not place bounds on the amount of available memory (even though individual digital computer all have limited memory). The difference in computational power between a specific digital computer and digital computers, in general, represents a competence/performance distinction.

In the same way that Turing machines can be viewed as equivalent in computing power to the general class of digital computers, they can be also viewed as equivalent to STCNs as well. In order to achieve the computational power of either a Turing machine, or a Pushdown automaton, the system must posses some form of unbounded storage

resource. In digital computers, this takes the form of memory circuits. In an STCN, there are two forms of unbounded storage resources: state nodes and activation values. While individual STCNs all require specific numbers of nodes and finite sets of activation values that these nodes can assume, STCNs in general have no such limitations. Either or both of these unbounded resources can be used to prove Turing machine equivalence.

Specifically, Franklin and Garzon (1988), and Sun et al. (1990b, 1991) have proven that STCNs with infinite numbers of state nodes are (like digital computers infinite amounts of memory) Turing machine equivalent. These approaches all use increasing state node numbers as the unbounded resource; and it comes as no surprise that an infinite number of nodes can provide unbounded memory. There is one important drawback to implementing unbounded memory in this manner: when nodes are added to a network, connections must also be added so that the new nodes can interact with the rest of the network. The added connections, of course, must be carefully weighted in such a way as to allow the new nodes to operate in a manner consistent with the existing nodes.

A less intuitive, but simpler way of providing STCNs with unbounded memory resources is to increase the set of activation values that the state nodes can assume. Thus far, we have assumed that the nodes in STCNs act as threshold elements. That is, they compute their activation based on the weighted sum of connected node activations. If the sum is greater than a certain threshold, the nodes activation is 1 otherwise it is 0. While units with logistic activation functions are capable of operating in this fashion (i.e. when weight values are large), this is by no means their only mode of operation. The range of the logistic activation functions encompasses all real numbers between 0 and 1. This implies, that a single state node can encode far more than a simple boolean value. In fact, if we assume that activation values are real numbers (and not just finite precision approximations to real numbers), then a single node has the representational power of the

infinite stack in a Pushdown automaton. In the following section, we explore how this is possible.

3.5.3 Implementing a Stack in a Connectionist Network

The stack of a Pushdown automaton (see Hopcroft and Ullman, 1979) can be viewed as a potentially infinite sequence of symbols. Each symbol must be chosen from a finite set of symbols, called the stack alphabet of the automaton, Γ. If the alphabet size is finite, then every symbol can be represented as a binary vector of finite width. For example, consider the stack alphabet: $\Gamma = \{a, b, c, d, e\}$. These five symbols can be encoded as binary vectors of length 3. E.g.: a=(0,0,0), b=(0,0,1), c=(0,1,0), d=(0,1,1), e=(1,0,0). Having encoded the symbols in this fashion, the sequence of symbols on the Pushdown automaton's stack can then be represented by a sequence of binary vectors. Thus, if the symbols on the stack are, (b, a, d, c, a, c, e, ...), then these symbols would be encoded as: ((0,0,1), (0,0,0), (0,1,1), (0,1,0), (0,0,0), (0,1,0), (1,0,0), ...). Note that these representations can be of unrestricted length. If the punctuation in the previous binary sequence is removed, then the binary string, 001000011010000010100..., is formed. If we now precede the binary sequence with a decimal point, e.g. 0.001000011010000010100..., then a real valued number in the range [0,1) is created. Since nodes with sigmoidal activation functions are capable of representing arbitrary numbers in this range, these nodes must also be able to represent Pushdown automata stacks containing arbitrary numbers of symbols. Of course, the premise here is that activation values are continuous.

The fact that a single node in a STCN is able to represent the contents of a Pushdown automaton's stack by no means implies that STCNs have the computational power of a Pushdown automaton. In order to be Pushdown automaton equivalent, a STCN must also have a control mechanism capable of popping symbols from, and pushing

symbols onto the stack. This involves computing the next value of the state node responsible for storing the stack contents based on the nodes previous value as well as the STCN input vector. Since connectionist networks are known to be able to compute functions in L_2 (the space of functions whose coordinate functions are square-integrable on the unit cube) over the n-dimensional vector space $[0,1]^n$ to an arbitrary degree of precision, and since L_2 contains all "discontinuous functions that are piecewise continuous on a finite number of subsets of $[0,1]^n$" (Hecht-Nielsen, 1990, p. 133), they can approximate the function to update the stack node to an arbitrary degree of precision. This implies that STCNs in general are as powerful as Pushdown automata. Of course, specific network designs with limited (finite) node numbers are not capable of universal function approximation and thus are not capable of performing arbitrary pop and push operations on the simulated stack. The restrictions on particular network designs will be examined in Section Chapter IV.

3.5.4 Turing Machine Equivalence

Having argued that STCNs are capable of representing the contents of Pushdown automata stacks, we now turn our attention to the most powerful computing device in the Chomsky hierarchy: the Turing machine. In order to represent the state of a Turing machine, a STCN (which is not equipped with an external tape) must be able to represent not only the infinite tape, but also the Turing machine's tape head's position on the tape. Perhaps the easiest way to represent both is to use a two-stack machine. As noted by Hopcroft and Ullman (1979), a two stack machine can simulate a Turing machine by using one stack for the contents of each side of the tape relative to the tape head. If the tape head moves left, then a symbol is popped from the left stack and pushed onto the right stack. Similarly, if the tape head moves right, then a symbol is popped from the right stack and pushed onto the left stack. Thus two stacks can be used to represent a Turing

machine and its infinite tape. As noted above, one stack's contents can be represented by one node in a STCN. Therefore, STCNs must also be capable of simulating Turing machines. This was first shown by Pollack (1987).

We have now identified the overlap between the languages, grammars and machines in the Chomsky hierarchy and those which can be implemented or approximated by connectionist networks under various assumptions (e.g. fixed precision vs. real-valued activation values, finite vs. infinite numbers of nodes). Specifically, we have seen that STCNs with finite node numbers and node precisions are computationally as powerful (in general) as finite state automata, and thus, can implement regular grammars and accept regular sets. By contrast, STCNs with either unbounded node numbers or real-valued node activations are Turing machine equivalent, being able to implement unrestricted grammars and accept recursively enumerable languages. The inability of specific STCNs to provide unbounded memory resources will cause us to focus primarily on the relationship between STCNs and finite state automata, in the chapters which follow.

3.5.5 Hypothesis Spaces in Connectionist vs. Symbolic Systems

Despite the common representational powers of STCNs and classical computational devices, there are important differences between the hypothesis spaces used in connectionist grammatical induction systems as compared to symbolic ones. The traditional search space for the grammatical induction task is discrete. That is, there exist a countable number of distinct machines which can be induced. This is due to the fact that there are only a finite number of output symbols which can be generated, and only a countable number of states which such a machine could assume. By contrast, connectionist models use real numbers to represent output symbols and states. Since real numbers are not countable, the set of all possible spatio-temporal connectionist systems must also not be countable. Even if connection weights are represented using finite

precision, the vast number of possible connectionist networks of a given size makes their enumeration infeasible in practise.

This critical difference between traditional and connectionist systems results in widely different methods for searching the hypothesis space of STCNs. In the traditional system, it is possible to systematically search the space, machine by machine, always ruling out those machines which cannot produce the grammatical output strings and those which can produce non-grammatical strings. Note that, as Gold (1967) has shown, this systematic search procedure does not guarantee that the correct solution will be found after a finite length of time.

In a STCN, a systematic search which explores all options is not possible due to the uncountable number of possible STCNs. Instead, the space of inducible machines is sampled at various points until a STCN which produces an error less than a given tolerance with respect to the grammatical and non-grammatical strings is found. For this reason, enumerative hypothesis space exploration procedures cannot be applied to STCNs.

3.5.6 Training Networks and Grammatical Induction

Since the weights, W, of a STCN define the language that it can accept (or generate), they can be considered a grammar for that language in the sense that they represent a finite characterization of a potentially infinite language. Similarly, an STCN which performs a pattern classification based on a string of input symbols is a formal computing machine since it partitions strings of symbols into a finite number of categories. We, thus view the process of adapting the weights of a network to suit some problem as a type of grammatical induction.

We will not examine the issue of translating a grammar which is represented as a set of weights into a grammar represented as, for example, rules of Backus-Naur form in this dissertation. Nor will we examine how a given STCN might be translated to a formal

computing device of the types described in Section 3.4. The translation of connectionist representations of languages to traditional ones has been extensively studied by Das, Giles and Sun (1993), Giles et al. (1991), Miller and Giles (1993), Omlin, Giles and Miller (1992), Sun et al. 1993, and Watrous and Kuhn (1992), among others.

3.6 THE GRAMMAR INDUCTION PROBLEM

3.6.1 The Importance of Grammatical Induction

The term grammatical induction is easily associated with learning human languages, but grammars can be used to describe far more than the intuitive notion of a language. It is perhaps this misleading connotation that led King Sun Fu, one of the foremost grammatical induction researchers in the classical paradigm, to use the term "syntactic pattern recognition" to describe this field instead (Fu, 1982). We will continue to use the words grammatical induction, but also point out that grammars can be used to describe many different things.

Specifically, grammatical induction has been applied to: modelling natural language learning, process control, signal processing, phonetic to acoustic mapping, speech generation, robot navigation, nonlinear temporal prediction, system identification, learning complex behaviours, motion planning, prediction of time ordered medical parameters, and speech recognition, to name but a few. In fact, grammatical induction can be used to induce anything that can be represented by a language. In this sense, it represents a prototypical form of induction.

3.6.2 The Difficulty of Inducing Chomsky Grammars

Having identified some fundamental hypothesis spaces for grammar induction in the symbolic paradigm, and having shown that STCNs can use these same hypothesis spaces, we now examine the consequences of using the Chomsky grammar classes as

hypothesis spaces for grammatical induction. We begin by considering the types of information available to guide the search. Gold (1967) has identified two basic methods of information presentation, "text" and "informant". A *text* is a sequence of legal strings containing every string of the language at least once. Typically, texts are presented one symbol after another, one string after another. Since most interesting languages have an infinite number of strings, the process of string presentation never terminates. An *informant* is a device which can tell the learner whether any particular string is in the language. Typically the informant presents one symbol at a time, and upon a string's termination supplies a grammaticality judgement.

Gold (1967) investigated the problem of language identification in the limit. He asked the question: Which classes of languages are learnable with respect to a particular method of information presentation? A class of languages is learnable if there exists an algorithm which repeatedly guesses languages from the class in response to example strings, and "Given any language of the class, there is some finite time after which the guesses will all be the same and will all be correct" (Gold 1967, p.447). The algorithm does not keep guessing forever or, more precisely, it settles on a particular language and that language is correct.

Gold showed that this is a surprisingly difficult task. For example, if the method of information presentation is a *text*, then only finite cardinality languages can be learned. Finite cardinality languages consist of a finite number of legal strings, and are a small subset of the regular sets (the smallest set in the Chomsky hierarchy). In other words, **none** of the language classes in the Chomsky hierarchy are text learnable.

The situation is somewhat more promising if both positive and negative examples from the language are available. Under *informant learning*, the regular sets, and context-free languages are both *identifiable in the limit*, however, the recursively enumerable languages remain unlearnable.

The fact that regular sets and context-free languages are learnable under the *informant learning* paradigm by no means implies that such learning is practical. Pinker points out that "in considering all the finite state grammars that use seven terminal symbols and seven auxiliary symbols (states), which the learner must do before going on to more complex grammars, he must test over a googol (10^{100}) candidates" (Pinker 1979, p.227). This reveals that even for tiny computational machines (seven states) the machine induction problem is often intractable if no a priori knowledge is available to remove some of the machines from consideration.

The conclusion which must be drawn from Gold and Pinker's observations is that grammatical induction is an exceptionally complex task. So complex, in fact, that it cannot be solved as originally posed by Gold. In this and the following chapters, we shall present a number of modifications to the original problem which overcome the inherent difficulties implied by Gold and Pinker's conclusions and thus allow the problem to be solved by connectionist networks.

3.6.3 Tractable Learning

The difficulty of any search depends on the size of the hypothesis space that must be explored as well as any available evidence to narrow down that space (in addition to the difficulty of generating and testing new hypotheses). In this chapter we focus on how a priori knowledge about a particular problem can be used to make the search more efficient. We begin by considering a simple example: Suppose you are unable to find your car keys. We shall assume that the keys are somewhere in the house. A simple search algorithm might involve searching the house from top to bottom—starting on the upper floor and moving down to the basement. This represents an exhaustive brute-force search. Now suppose you know for a fact that you have not been upstairs or downstairs since you last used the keys. In this case, it would be sensible to reduce the search space

from the entire house to just the ground floor. This would no doubt lead to a more efficient search and you would expect to find your keys sooner. Thus, reducing the search space increases search efficiency. Of course, there is a drawback to a reduced search space. Suppose you had forgotten that you had in fact travelled upstairs, and left the keys there. Now your search of only the ground floor would be guaranteed to fail. A reduced hypothesis space is useful only if does not exclude the goal.

Another way to speed search is to order or bias the hypothesis space based on some heuristic. Suppose you are an habitual car key loser and that you keep track of where your keys turn up after each search. The results of such record keeping might be something like: coat pocket: 53%, hallway shelf 27%, kitchen table: 16%, beside telephone: 3%, in refrigerator 1%. If you know that most of the time the missing key has been located in your coat pocket, then it makes sense to begin your search there. That is, it is logical to order your hypothesis space and bias it in favour of the coat pocket. But just like a bad hypothesis space reduction can hinder search, a bad ordering can also impede an effective search. For example, using the hypothesis ordering designed for your car keys to find a pitcher of orange juice would clearly be very inefficient.

3.6.4 Reducing and Ordering in Grammatical Induction

Naturally, the techniques of hypothesis space reduction and ordering described in the previous section are applicable to search in general—not just car key searches. As such, they can be used to make the task of grammatical induction solvable or tractable. The notion of hypothesis space reduction in the context of grammatical induction refers to searching for a grammar consistent with the training data in a class which is smaller than the class of unrestricted (Chomsky type-0) grammars. Symbolic grammar induction systems have used the class of context-free grammars (Chomsky type-2) and the class of regular grammars (Chomsky type-3) as reduced hypothesis spaces. However, the fact that

Gold showed that even the smallest of these classes is not learnable based solely on *text* training data, combined with the fact that most interesting grammars belong to the larger classes, have made these restrictions unpopular techniques for hypothesis space reduction.

A more useful technique is to devise a class of grammars which lies tangential to Chomsky's hierarchy. Such a tangential class contains some grammars which are not regular, and some grammars which are not context-free but contains only a subset of the unrestricted grammars. By using a class tangential to the Chomsky hierarchy as one's hypothesis space it will be possible to represent some of the grammars which only fall into the unrestricted class, while at the same time reducing the size of the hypothesis space so as to identify members of the space based on input data more rapidly. Of course, as with the car key example, it is critical to choose an hypothesis space which contains those grammars which are to be learnable.

Restricted hypothesis spaces in symbolic grammatical induction systems are typically described in terms of restrictions on the type of grammar rules they employ. These restrictions on rules are similar in nature to those provided in Row 2 of Table 3-1. Figure 3-2 illustrates the Chomsky hierarchy of languages and a tangential class of languages representing a reduced hypothesis space. Ovals and their contents represent classes of languages.

This type of hypothesis restriction was first suggested by Chomsky (1965). While working on the problem of human language acquisition, he proposed that only those grammars possessing the basic properties of natural languages should be considered as candidates for grammatical induction. By weighting the naturalness of languages based on a specific set of properties, he proposed an induction algorithm which considered only those languages which were both consistent with the training sample, and had a sufficiently

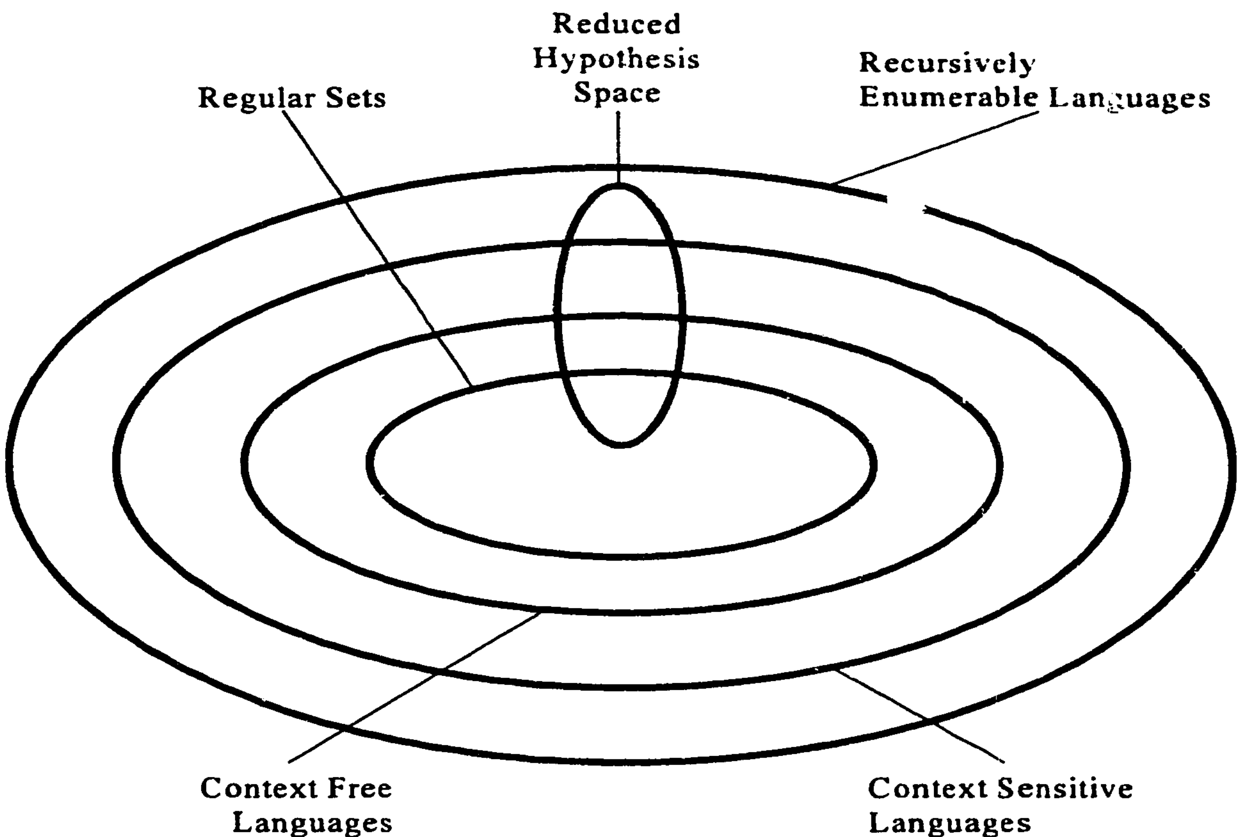

Figure 3-2: The Chomsky hierarchy and a reduced hypothesis space.

high weight.

Another popular technique for restricting the space is to employ the universal base hypothesis. Under this hypothesis different grammars are defined by means of a two-step process. First, a universal base grammar which all different grammars use is defined. Then, a restricted class of rewrite rules are employed to translate from the symbols of the universal base grammar to a variety of derived grammars. The grammars derived in this fashion form a reduced hypothesis space which can then be used to define a grammatical induction algorithm. This approach is fundamental to Wexler and Culicover's (1980) model of human language learning and will be discussed in greater detail in Sections 4.2 and 4.3.

Just as hypothesis space restriction can be used to simplify the search for a grammar, hypothesis space ordering has also been applied to grammatical induction within the symbolic paradigm. In this case, a working hypothesis about the grammar (from the hypothesis space) is used as a starting point. Then, as new evidence about the grammar is presented in the form of training data, a change to the hypothesis is made. The nature of this change is defined by some heuristic. That is, certain types of hypothesis changes will be favoured over other changes even if both are consistent with the training data. The chosen hypothesis change results in a new working hypothesis, and the process is repeated. Typically all of the possible hypothesis changes are evaluated and the resulting hypotheses are evaluated according to some weighting scheme. Then only the highest valued new hypothesis is selected as the new working hypothesis. This is analogous to a best first search algorithm.

A weighting scheme could be based on complexity, for example, by assigning a weight inversely proportional to the number of auxiliary symbols (states) used by each grammar. This weighted selection process effectively orders the grammars of the hypothesis space. While searching for a grammar which is consistent with the training data, this ordering favours certain solutions over others. Ideally, good solutions to the problem to which the grammatical induction system is applied, would be considered first, and thus, learning would be speeded. We will give a specific example of a symbolic weighting scheme in Section 6.4.4 and compare it to a similar connectionist hypothesis space ordering scheme.

3.7 CONCLUSIONS

In this chapter we provided a description of the grammatical induction problem from the perspective of a search through a space of candidate grammars called an hypothesis space. In doing so, we have defined much of the terminology which will be

used in the following chapters. Additionally, we have explored the inherent difficulties of grammatical induction, and reviewed two broad solutions that Gold and others have proposed to overcome them: hypothesis space reduction and hypothesis space ordering. Since the difficulties of grammatical induction identified by Gold (1967) are theoretical, rather than implementation-specific, any connectionist solutions to the grammatical induction problem must also address these issues. By identifying important theoretical principles which can be applied to connectionist systems to make the induction problem tractable and efficient, this chapter gives the designers of STCNs a new set of principles around which to design their systems. The theoretical approach is particularly significant since, in the past, many connectionist researchers have focussed on empirical performance data to select or develop their networks. The following chapters apply the theoretical principles identified here to specific network designs.

Chapter IV: A Priori Knowledge and the Selection of Appropriate State and Output Functions

4.1 INTRODUCTION

While the previous chapter related formal languages and automata to STCNs in general, this chapter examines the restrictions that particular STCN designs place on the languages and automata which they can represent. This is a critically important issue since an hypothesis space which is too small can rule out ideal solution grammars, while an hypothesis space which is too large can easily make a problem unsolvable or at the least intractable. Thus, a connectionist's choice of STCN design directly influences induction speed and success. In the past, most connectionist networks have been designed based on principles like ease of implementation or extension from existing work, rather than on the classes of languages and automata that they can induce.

This chapter presents new formal proofs describing the representable languages and automata for five of the seven possible memory functions identified in Chapter II (for the remaining two functions, previous results are applied to draw conclusions about computational power). Specifically, we prove a new result describing the types of languages implementable by window-in-time memories. Then, we prove that single-layer first-order context computation memories can implement arbitrary finite state automata and that they can do so using $n \cdot p$ nodes (where n is the number of states, and p is the number of input symbols). We also prove that single-layer second-order context computation memories are incapable of implementing arbitrary automata using binary state encodings. Next, we prove that a locally recurrent state and input memory is incapable of representing finite state automata whose state transitions form cycles of length greater than two under an input signal consisting of two alternating symbols. We also prove that if the input signal to a LRSI memory oscillates with period, n, then the LRSI memory can represent only those finite state automata whose state transitions form cycles of length l, where $n \bmod l = 0$ if n is even and $2n \bmod l = 0$ if n is odd. Finally, we prove that a one-layer output function, together with a single-layer first-order context computation memory can compute output of any automaton.

In addition to discovering and proving these new results, this chapter presents a table relating memory-types, output functions, formal-computing machines and number of nodes. This is the first synthesis of this kind and can serve as an essential guide to anyone intending to use a STCN for grammatical induction. By using the table rather than principles like ease of implementation or extension from existing work, researchers will be able to choose and design networks based on the classes of languages and automata that they wish their system to be able to learn (i.e. the hypothesis space of their search algorithm). Since a judicious choice of hypothesis space can make an otherwise unsolvable problem solvable or an otherwise intractable problem tractable, the

relationships between STCN designs and representable languages is clearly essential to any connectionist grammar induction system.

Choosing a particular state function may seem like an unusual way of restricting the hypothesis space. Perhaps this is because many authors use only one state function in their research (often a function designed by themselves) and give little or no justification for the choice. Yet as we shall see below, the choice of function critically influences the representational power of the STCN and hence the hypothesis space available to the learning algorithm. This chapter discusses the effect on hypothesis space of each state and output function presented in the taxonomy of the previous chapter.

4.2 WINDOW-IN-TIME (WIT) MEMORIES

4.2.1 WIT Machines

We begin with the Window in Time (WIT) memory function. The Window in Time (WIT) approach constrains the machines which can be induced by using a state vector which is fixed in the sense that it is always formed by the vector concatenation of the current input symbol and the previous n-1 input symbols:

$$\vec{s}(t) = \vec{x}(t) \oplus \vec{x}(t-1) \oplus \cdots \oplus \vec{x}(t-n+1)$$

Here, n represents the size of the temporal window on previous inputs. This function is very restrictive in the sense that the weights of the network cannot be adjusted to influence the content of the state vector, $\vec{s}(t)$. This state function is similar to the symbolic approach which uses a fixed base grammar to generate non-terminals (described earlier). Both define a fixed grammar which generates symbols (in this case, concatenated input vectors) which represent the non-terminals of a base grammar.

More formally, let us assume that the machine uses p input symbols denoted by x_0, x_1, x_2, ... , x_{p-1}, plus a blank symbol denoted b. In the notation of Hopcroft and Ullman (1979), the input alphabet is:

$$\Sigma = \{ x_0, x_1, x_2, \ldots, x_{p-1}, b \}$$

Each of these input symbols is represented by a vector of input activations, in the STCN. If the input vector at time, t, is denoted $\vec{x}(t)$, then the state of the WIT memory,

$$\vec{s}(t) = \vec{x}(t) \oplus \vec{x}(t-1) \oplus \cdots \oplus \vec{x}(t-n+1),$$

can be symbolically represented by labelling states with the strings of previous inputs. In Hopcroft and Ullman's (1979) notation:

$$Q = \{ X(t)X(t\text{-}1)X(t\text{-}2) \ldots X(t\text{-}n+1) \mid X(t')\in\Sigma \}$$

where $X(t)$ represents the input symbol at time t. The state transition function, which is fixed throughout training simply consists of right-shifting the previous state $(X(t)X(t\text{-}1)X(t\text{-}2) \ldots X(t\text{-}n+1))$, and appending it to the new input symbol $X(t+1)\in\Sigma$. This is represented as:

$$\delta(X(t)X(t\text{-}1)X(t\text{-}2) \ldots X(t\text{-}n+1), X(t+1)) = X(t+1)X(t)X(t\text{-}1)X(t\text{-}2) \ldots X(t\text{-}n+2)$$

This transition function is implemented by shifting the internal memory vector in the WIT memory. Finally, the initial state of the WIT memory is simply the state whose string representation consists of n blank symbols:

$$q_0 = bbb\ldots b$$

The WIT memory is then simply a finite automaton represented as the four-tuple:

$$M = (Q, \Sigma, \delta, q_0)$$

We purposely omit any references to accepting states or output functions since these are specified by the STCNs output function, not the memory function. The restrictions presented here present a formal specifications of the types of machines which can be implemented in a WIT memory. Giles, Horne and Lin (in press) were first to recognize that Kohavi (1978) had previously called this subclass of finite state automata *definite machines*. We have rewritten Kohavi's formulation of definite machines in order to be consistent with the notation of Hopcroft and Ullman (1979) used throughout this dissertation.

4.2.2 WIT Languages

We now consider the languages which an STCN with a WIT memory can accept. Specifically, we will identify a set of restrictions on grammatical rewrite rules such that the resulting language can be accepted by a WIT memory. Since we already know that these memories are a subset of finite state automata, we can restrict ourselves to grammatical rewrite rules of the form:

$A \to wB$ or $A \to w$

where A and B are non-terminals and w is a (possibly empty) string of terminals. If the input window is of width n, then it is possible to place the following additional restrictions on the rewrite rules. We restrict ourselves to rules which satisfy one of the following two forms:

(1) $S \to w$, w is a string of at most n terminals.

(2) $S \to aS$, where a is a single non-terminal symbol.

The restrictions on rewrite rules can be described by the logical formulae:

(1) $V \equiv \{S\}$

(2) $\forall w \in T^* \; (((S \to w) \in P) \Rightarrow (\|w\| \leq n))$

(3) $\forall w \in T^* \; (((S \to wS) \in P) \Rightarrow (\|w\| = 1))$

(4) $\forall p \in P \; (\exists w \text{ s.t. } p = (S \to w) \vee p = (S \to wS))$

Together, these four rule restrictions guarantee that strings in the language have a temporal dependence of at most n symbols. In other words, if a grammar satisfies all three rules above, then it can be implemented by a WIT memory. We state the relation between the

grammars representable by a WIT memory and the restrictions on their rewrite rules in our first theorem:

***Theorem 4-1*:** Any grammar, $G=(V,T,P,S)$, satisfying the conditions, $V \equiv \{S\}$, $\forall w \in T^*$ $(((S \rightarrow w) \in P) \Rightarrow (\|w\| \leq n))$, $\forall w \in T^*$ $(((S \rightarrow wS) \in P) \Rightarrow (\|w\|=1))$ and $\forall p \in P$ $(\exists w$ s.t. $p=(S \rightarrow w) \vee p=(S \rightarrow wS))$ identifies a language which can be accepted by a STCN with a WIT memory.

***Proof*:** We prove this theorem by showing that any grammar satisfying the four criteria must be implementable by a definite machine, and hence a network with a WIT memory. This is done by noting that the second and third rules imply that the grammaticality of any string depends only on (at most) the last n symbols of the string. Since (at most) the last n symbols presented to a *definite machine* define its state, an appropriate function to map state to output will always be able to accept exactly the set of grammatical strings specified by a grammar conforming to the four rules. This proves Theorem 4-1.□

Conversely, any grammar defining a language accepted by a WIT memory can be represented in such a way that it conforms to all four rules above. This is our second theorem:

***Theorem 4-2*:** For every language accepted by a STCN with a WIT memory, there exists a grammar, $G=(V,T,P,S)$, that describes the language and satisfies the conditions: $V \equiv \{S\}$, $\forall w \in T^*$ $(((S \rightarrow w) \in P) \Rightarrow (\|w\| \leq n))$, $\forall w \in T^*$$(((S \rightarrow wS) \in P) \Rightarrow (\|w\|=1))$, and $\forall p \in P$ $(\exists w$ s.t. $p=(S \rightarrow w) \vee p=(S \rightarrow wS))$.

Proof: Since the output function of a *definite machine* computes a grammaticality judgement based on the value of its state, and since the state encodes the last n symbols presented to the machine, it must be the case that the last n symbols in a string define its grammaticality. If every set, w, of n symbols that can terminate a legal string is included in a rule of the form $S \rightarrow w$, and if rules of the form $S \rightarrow aS$ are added for all symbols, a, in the language, then the grammaticality of every string will depend only on its last n symbols.□

These two new proofs identify, the languages which networks with WIT memories can accept.

For a finite state vector size, n, the WIT memory clearly places great restrictions on the types of grammars which can be induced. Most importantly, a WIT memory is incapable of classifying two input strings into different categories if they end in n identical symbols. An example of such a task might be to classify strings consisting of the digits "0" and "1" according to parity (as originally described in Section 3.3). Since it is always possible to devise two strings of length $n+1$ which differ only in their first symbol, no window size will ever be adequate for this task. This reveals that, for any finite state vector size, the WIT memory limits the types of grammars which can be induced. Of course, reducing the size of the state vector further limits the types of state functions which can be represented since state vector size dictates how far back in input history the memory is allowed to look. Alternatively, if the number of state nodes in increased to infinity, a WIT memory's representational power encompasses not only that of a finite state automaton, but also that of a Turing Machine.

4.3 CONNECTIONIST INFINITE IMPULSE RESPONSE FILTER (CIIR) Memories

The connectionist infinite impulse response filter (CIIR) memory is more powerful than the WIT memory since the latter is a special case of the former. The content of a CIIR memory also constrains the classes of machines which can be induced by using a state vector which is fixed. More formally, let us assume that the machine uses p input symbols denoted by $x_0, x_1, x_2, \ldots, x_{p-1}$, plus a blank symbol denoted b. In the notation of Hopcroft and Ullman (1979), the input alphabet is:

$$\Sigma = \{ x_0, x_1, x_2, \ldots, x_{p-1}, b \}$$

Each of these input symbols is represented by a vector of input activations, $\vec{x}$, in the STCN, with $\vec{x}(t)$ representing the input vector presented to the network at time, t. Unlike the simple WIT memory, however, the CIIR memory uses both previous input and previous output symbols to compute its state. If there are q output symbols denoted y_i, then the output alphabet is:

$$\Delta = \{ y_0, y_1, y_2, \ldots, y_{q-1} \}$$

Each of these output symbols is represented by a vector of output node activations, $\vec{y}$, with $\vec{y}(t)$ representing the output vector presented to the network at time t. The state of the CIIR memory:

$$\vec{s}(t) = \vec{x}(t) \oplus \vec{x}(t-1) \oplus \cdots \oplus \vec{x}(t-n-1) \oplus \vec{y}(t-1) \oplus \vec{y}(t-2) \oplus \cdots \oplus \vec{y}(t-m)$$

can then be symbolically represented by labelling states by the strings of previous inputs and outputs. In Hopcroft and Ullman's (1979) notation:

$$Q = \{ X(t)X(t\text{-}1)X(t\text{-}2) \ldots X(t\text{-}n\text{-}1)Y(t\text{-}1)Y(t\text{-}2)Y(t\text{-}3) \ldots Y(t\text{-}m) \mid X(t')\in\Sigma,\ Y(t')\in\Delta \}$$

The state transition function, which is fixed throughout training simply consists of adding the new input symbol to the previous n-1 input symbols which are added to the previous m output symbols.. This is represented as:

$$\delta(X(t)X(t\text{-}1)X(t\text{-}2)\ldots X(t\text{-}n\text{-}1)Y(t\text{-}1)Y(t\text{-}2)Y(t\text{-}3)\ldots Y(t\text{-}m), X(t+1)) =$$

$$X(t+1)X(t)X(t\text{-}1)X(t\text{-}2)\ldots X(t\text{-}n\text{-}2)Y(t)Y(t\text{-}1)Y(t\text{-}2)Y(t\text{-}3)\ldots Y(t\text{-}m\text{-}1),$$

This transition function is implemented by right-shifting the internal memory vector in the CIIR memory. Finally, the initial state of the CIIR memory is simply the state whose string representation consists of exactly $n+m$ blank symbols:

$$q_0 = bbb\ldots b$$

The CIIR memory is then simply a finite automaton represented as the five-tuple:

$$M = (\, Q, \Sigma, \delta, \Delta, q_0\,)$$

The restriction presented here present a formal specifications of the types of machines which can be implemented in a CIIR memory. Once again, Giles et al. (in press) were first to recognize the relation between CIIR memories and what Kohavi (1978) has called *finite memory machines*. We have rewritten Kohavi's formulation in order to be consistent with the notation of Hopcroft and Ullman (1979) used throughout this dissertation.

Finally, it should be noted that the WIT memory is a special case of a CIIR memory where $m=0$. Thus, the conclusions about FSA and Turing Machine equivalence as the number of nodes is increased to infinity apply.

4.4 SINGLE-LAYER FIRST-ORDER CONTEXT COMPUTATION (SLFOCC) MEMORIES

4.4.1 SLFOCC Memories Cannot Compute Arbitrary State Functions

Single-Layer First-Order Context Computation memories are far more powerful than CIIR memories. This is because the state update function uses a matrix of weights as a parameter. This weight matrix gives the state update function flexibility to compute a wide variety of functions. The matrix, however, is not powerful enough to represent arbitrary mappings from input and context to state. Goudreau et al. (1994) proved this by considering the following example which we now present in a notation consistent with the rest of this thesis.

Suppose a SLFOCC memory is to switch between two states based on whether or not strings of "0"s and "1"s presented to the memory have an even number of "1"s or an odd number of "1"s. Without loss of generality, we let $\vec{s}_0$ and $\vec{s}_1$ represent the states assumed by the SLFOCC memory in response to strings with even and odd numbers of "1"s respectively. We also let $\vec{x}_0$ and $\vec{x}_1$ represent the vectors corresponding to input symbols "0" and "1" respectively. Now, in order to compute its next state correctly, the state function of the SLFOCC memory would have to be defined by:

$$f_{\vec{s}}(\ \vec{s}_0,\ \vec{x}_0,\ W\ )\ =\ \vec{s}_0$$

$$f_{\vec{s}}(\ \vec{s}_0,\ \vec{x}_1,\ W\ )\ =\ \vec{s}_1$$

$$f_{\vec{s}}(\ \vec{s}_1,\ \vec{x}_0,\ W\ )\ =\ \vec{s}_1$$

$$f_{\vec{s}}(\ \vec{s}_1,\ \vec{x}_1,\ W\ )\ =\ \vec{s}_0$$

Combining these equations gives:

$$f_{\vec{s}}(\ \vec{s}_0,\ \vec{x}_0,\ W\ )\ =\ f_{\vec{s}}(\ \vec{s}_1,\ \vec{x}_1,\ W\ )$$

$$f_{\vec{s}}(\ \vec{s}_0,\ \vec{x}_1,\ W\ )\ =\ f_{\vec{s}}(\ \vec{s}_1,\ \vec{x}_0,\ W\ )$$

But, since the state function is computed by multiplying the weight matrix by the concatenation of the input and state vectors and passing the result through a monotonic squashing function, the squashing function can be eliminated to give:

$$W\times\{1\oplus\vec{x}_0\oplus\vec{s}_0\}\ =\ W\times\{1\oplus\vec{x}_1\oplus\vec{s}_1\}$$

$$W\times\{1\oplus\vec{x}_1\oplus\vec{s}_0\}\ =\ W\times\{1\oplus\vec{x}_0\oplus\vec{s}_1\}$$

We now let W^1 represent the left-most column of the matrix W (the part that is multiplied by 1), W^x represent the middle part of the matrix W (the part which is multiplied by components of the vector $\vec{x}$, and W^s represent the remainder of the matrix. It is now possible to express the two equations as:

$$W^1\times 1\ +\ W^x\times\vec{x}_0\ +\ W^s\times\vec{s}_0\ =\ W^1\times 1\ +\ W^x\times\vec{x}_1\ +\ W^s\times\vec{s}_1\ ,$$

$$W^1\times 1\ +\ W^x\times\vec{x}_1\ +\ W^s\times\vec{s}_0\ =\ W^1\times 1\ +\ W^x\times\vec{x}_0\ +\ W^s\times\vec{s}_1\ .$$

Subtracting the second equation from the first yields:

$$W^{x}\times\vec{x}_0 - W^{x}\times\vec{x}^{1} = W^{x}\times\vec{x}_1 - W^{x}\times\vec{x}_0,$$

which implies:

$$W^{x}\times\vec{x}_0 = W^{x}\times\vec{x}_1.$$

Similarly, adding the equations results in:

$$W^{s}\times\vec{s}_0 = W^{s}\times\vec{s}_1.$$

But, since

$$\vec{s}_0 = \vec{\sigma}(\ W^{1}\times 1 + W^{x}\times\vec{x}_0 + W^{s}\times\vec{s}_0\ ) = \vec{\sigma}(\ W^{1}\times 1 + W^{x}\times\vec{x}_1 + W^{s}\times\vec{s}_0\ ) = \vec{s}_1,$$

it is impossible to distinguish between even and odd states. In this manner, Goudreau et al. showed that no SLFOCC memory (regardless of state vector size) can assume one state for all even strings and a different state for all odd strings.

4.4.2 SLFOCC Memories Can Represent Arbitrary Finite State Automata (Moore Machines)

At first, the inability to compute arbitrary state functions might seem like a serious drawback to SLFOCC memories, were it not for the fact more than one automaton can compute a given state function. In other words, different automata can be functionally equivalent. If a connectionist network design is to be as powerful as the class of finite state automata, then only one automaton out of each set of equivalent automata must be representable. Goudreau et al. (1994) have argued that, for every automaton that cannot be directly implemented in a SLFOCC memory, there exists another automaton whose behaviour is identical to the original, but which uses a different representation of internal states and can thus be realized by a SLFOCC memory. They call this technique "state splitting" because it assigns several states in the new automaton the same role that was played by a single state in the original device.

The Parity Problem Revisited

Goudreau et al. (1994) give an example of how "state splitting" can be used to solve the parity problem. We now present this result in a notation that is consistent with the remainder of this chapter. It is possible to split the "odd" state $\vec{s}_1$ into two new "odd" states according to whether or not the last symbol presented was a "0" or a "1". We shall use $\vec{s}_{10}$ and $\vec{s}_{11}$ to represent odd strings ending in "0" and "1" respectively. Then the state function must compute:

$$f_{\vec{s}}(\ \vec{s}_0,\ \vec{x}_0,\ W\ ) = \vec{s}_0$$
$$f_{\vec{s}}(\ \vec{s}_0,\ \vec{x}_1,\ W\ ) = \vec{s}_{11}$$
$$f_{\vec{s}}(\ \vec{s}_{10},\ \vec{x}_0,\ W\ ) = \vec{s}_{10}$$
$$f_{\vec{s}}(\ \vec{s}_{10},\ \vec{x}_1,\ W\ ) = \vec{s}_0$$
$$f_{\vec{s}}(\ \vec{s}_{11},\ \vec{x}_0,\ W\ ) = \vec{s}_{10}$$
$$f_{\vec{s}}(\ \vec{s}_{11},\ \vec{x}_1,\ W\ ) = \vec{s}_0$$

Clearly there are many possible solutions to this set of equations. One solution is:

$$\vec{x}_0 = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \qquad \vec{x}_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

$$\vec{s}_0 = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \qquad \vec{s}_{10} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \qquad \vec{s}_{11} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

$$W = k \cdot \begin{pmatrix} +5 & -2 & -2 & -2 & +5 \\ -1 & +2 & +2 & +2 & -2 \end{pmatrix}$$

where k is a large constant.

Proof:

$$f_{\vec{s}}(\vec{s}_0,\vec{x}_0,W) = \vec{\sigma}(k \cdot \begin{pmatrix} +5 & -2 & -2 & -2 & +5 \\ -1 & +2 & +2 & +2 & -2 \end{pmatrix} \times \{1 \oplus \begin{pmatrix} 1 \\ 1 \end{pmatrix} \oplus \begin{pmatrix} 0 \\ 1 \end{pmatrix}\}) = \vec{\sigma}(\begin{pmatrix} +6k \\ +k \end{pmatrix}) \approx \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \vec{s}_0$$

$$f_{\vec{s}}(\vec{s}_0,\vec{x}_1,W)=\vec{\sigma}(k\cdot\begin{pmatrix}+5&-2&-2&-2&+5\\-1&+2&+2&+2&-2\end{pmatrix}\times\{1\oplus\begin{pmatrix}1\\1\end{pmatrix}\oplus\begin{pmatrix}1\\0\end{pmatrix}\})=\vec{\sigma}(\begin{pmatrix}-1k\\+5k\end{pmatrix})\approx\begin{pmatrix}0\\1\end{pmatrix}=\vec{s}_{11}$$

$$f_{\vec{s}}(\vec{s}_{10},\vec{x}_0,W)=\vec{\sigma}(k\cdot\begin{pmatrix}+5&-2&-2&-2&+5\\-1&+2&+2&+2&-2\end{pmatrix}\times\{1\oplus\begin{pmatrix}1\\0\end{pmatrix}\oplus\begin{pmatrix}0\\1\end{pmatrix}\})=\vec{\sigma}(\begin{pmatrix}+8k\\-k\end{pmatrix})\approx\begin{pmatrix}1\\0\end{pmatrix}=\vec{s}_{10}$$

$$f_{\vec{s}}(\vec{s}_{10},\vec{x}_1,W)=\vec{\sigma}(k\cdot\begin{pmatrix}+5&-2&-2&-2&+5\\-1&+2&+2&+2&-2\end{pmatrix}\times\{1\oplus\begin{pmatrix}1\\0\end{pmatrix}\oplus\begin{pmatrix}1\\0\end{pmatrix}\})=\vec{\sigma}(\begin{pmatrix}+k\\+3k\end{pmatrix})\approx\begin{pmatrix}1\\1\end{pmatrix}=\vec{s}_0$$

$$f_{\vec{s}}(\vec{s}_{11},\vec{x}_0,W)=\vec{\sigma}(k\cdot\begin{pmatrix}+5&-2&-2&-2&+5\\-1&+2&+2&+2&-2\end{pmatrix}\times\{1\oplus\begin{pmatrix}0\\1\end{pmatrix}\oplus\begin{pmatrix}0\\1\end{pmatrix}\})=\vec{\sigma}(\begin{pmatrix}+8k\\-k\end{pmatrix})\approx\begin{pmatrix}1\\0\end{pmatrix}=\vec{s}_{10}$$

$$f_{\vec{s}}(\vec{s}_{11},\vec{x}_1,W)=\vec{\sigma}(k\cdot\begin{pmatrix}+5&-2&-2&-2&+5\\-1&+2&+2&+2&-2\end{pmatrix}\times\{1\oplus\begin{pmatrix}0\\1\end{pmatrix}\oplus\begin{pmatrix}1\\0\end{pmatrix}\})=\vec{\sigma}(\begin{pmatrix}+k\\+3k\end{pmatrix})\approx\begin{pmatrix}1\\1\end{pmatrix}=\vec{s}_0$$

Thus, the given values for $\vec{x}_0$, $\vec{x}_1$, $\vec{s}_0$, $\vec{s}_{10}$, $\vec{s}_{11}$, and W satisfy the requirements placed on the state equations for the parity problem, proving that the parity problem can be solved by a SLFOCC memory.□

Arbitrary Automata

Minsky (1967) argued that any finite state automaton could be implemented by this type of state splitting approach in a recurrent network of threshold elements. In order for a non-temporal connectionist network with continuous activation units to represent a function operating on sets of discrete symbols, an encoding relating the discrete symbols to points in the continuous vector space is required. Specifically, there must exist a mapping from input symbols to input vectors, as well as a mapping from the output vectors generated by the network to the output symbols generated by the function.

Similarly, for a STCN with logistic units to represent the state transition function of a finite state automaton, there must exist two functions mapping: (1) the input symbols to input vectors, and (2) the state vectors to a state symbols. Additionally, the state

function, $f_{\bar{s}}$, must correspond to automaton's state function, encoded according to the two functions. Formally, we say that:

Definition: A SLFOCC memory, defined by the weight matrix W, "represents" the state transition function $\delta : Q \times I \rightarrow Q$ of an automaton, if there exists a function $f_{i \rightarrow \bar{x}} : I \rightarrow [0,1]^p$, mapping input symbols to input vectors, and if there exists a function $f_{\bar{s} \rightarrow q} : [0,1]^n \rightarrow Q$, mapping state vectors to state symbols, whose inverse is denoted $f_{q \rightarrow \bar{s}}$, such that $\delta(q,i) = f_{\bar{s} \rightarrow q}(f_{\bar{s}}(f_{q \rightarrow \bar{s}}(q), f_{i \rightarrow \bar{x}}(i), W))$.

We can now prove:

Theorem 4-3: A SLFOCC memory can represent the state transition function of any finite state automaton.

Proof: The proof relies on a state splitting technique based on input patterns. That is, if a finite automaton has n states, $q_0, q_1, q_2, \ldots, q_{n-1}$, and p input symbols, $i_0, i_1, i_2, \ldots i_{p-1}$, then an equivalent automaton can be created with $n \cdot p$ states, which we shall label using a pairing of one of the state symbols of the original automaton with one of the input symbols, i.e. $Q=\{(q_0, i_0), (q_0, i_1), (q_0, i_2), \ldots (q_0, i_{p-1}), (q_1, i_0), (q_1, i_1), (q_1, i_2), \ldots (q_1, i_{p-1}), (q_2, i_0), (q_2, i_1), (q_2, i_2), \ldots (q_2, i_{p-1}), \ldots (q_{n-1}, i_0), (q_{n-1}, i_1), (q_{n-1}, i_2), \ldots (q_{n-1}, i_{p-1})\}$. As shown below, this state-split automaton can then be implemented by a SLFOCC memory. The key to constructing this new automaton is to translate the state transition function of the original automaton, $\delta(q,i)$, which computes the new state of the automaton based on the old state and the current input symbol, to a new function: $\delta'((q,i'), i) = (\delta(q,i),i)$ which incorporates input with the state.

The new automaton created in this fashion can now be considered equivalent to the original automaton in so far as its state can be used to compute the same output function as the original automaton. In order to do this, the output function must be capable of recombining the split states. As we shall see in Section 4.9.1 below, a 1-Layer STCN output function is capable of performing such a recombination. Of course, a 0-Layer output function cannot.

It now remains to be proven that a SLFOCC memory is capable of implementing the modified automaton. First, it will be necessary to chose a representation of the automaton's states in the STCN's state vector and input symbols in the input vector. We select the simplest possible representation for the input symbols, assigning the set of input symbols to unit normal vectors coded in the activations of the input units. More specifically, the kth component of the input vector used to represent input symbol i_j is equal to 1 if and only if $k=j$:

$$f_{i\to\vec{x}}(i_j)_k = \begin{cases} 0 & \text{if } j \neq k \\ 1 & \text{if } j = k \end{cases}$$

Similarly, state symbols can be represented by unit normal vectors encoded in the activations of state units. However, since the logistic function which computes the components of the state vector only approaches the values of 0 and 1, an error tolerance, $\epsilon < 0.5$, must be incorporated into the mapping between state symbols and state vectors. Specifically, each component of the state vector, $\vec{s}$, is mapped to one of the state symbols (q_j,i_k) in the split-state automaton. If the state vector's $(q_j,i_k)^{th}$ component is approximately 1, and all of its other components are approximately 0, then the state vector represents state symbol (q_j,i_k) of the split-state automaton. Since this state symbol can be further mapped to state symbol q_j in the original automaton, the function mapping state vectors to state symbols (in the original automaton) is described by:

$$f_{\vec{s}\to q}(\vec{s}) = \begin{cases} q_j & \text{if } \exists k \text{ s.t. } \vec{s}_{(q_j,i_k)} > 1-\epsilon \wedge \forall l \neq (q_j,i_k)\ \vec{s}_l < \epsilon \\ \text{undefined} & \text{otherwise} \end{cases}.$$

Note that this defines a partial mapping from the vector space of the state unit activation values since there are state vectors which have no state symbol interpretation. Also, note that the inverse of this function, $f_{q\to\vec{s}}$, is not a function, but rather a relation, since it maps individual state symbols to regions in the state vector's space.

This encoding now implies that each component, (q,i), of the state vector $\vec{s}$ must be computed according to the logical function:

$$\vec{s}_{(q,i)}(t) \stackrel{\epsilon}{=} \vec{x}_i(t) \wedge \left\{ \bigvee_{q'\in Q} \bigvee_{i'} \vec{s}_{(q',i')}(t-1) \right\},$$

where:

$$Q = \{ q' \mid \delta(q',i)=q \},$$

and $\stackrel{\epsilon}{=}$ denotes equality under the assumption that values in the range $[0,\epsilon]$ are interpreted as being equal to 0, and values in the range $[1-\epsilon,1]$ are interpreted as 1. In order to compute the value of the appropriate state vector component, each node must thus be able to compute the logical *AND* of the appropriate input node's activation and the value of the logical *OR* of a number context unit activations.

We show that a single logistic node is capable of computing such a function. Suppose that the weight of the connection from the input node to the state node is:

$$W_{\vec{x}_i(t),\vec{s}_{(q,i)}(t)} = c$$

where c is a positive constant (to be computed later). And, suppose that the weights of the connections from all of the context nodes $\vec{s}_{(q',i')}(t-1)$ are:

$$W_{\vec{s}_{(q',i')}(t-1),\vec{s}_{(q,i)}(t)} = \frac{c}{\|Q\|\cdot p}$$

(recall that p represents the number of input symbols). And finally, suppose that the bias value of the state node is:

$$W_{1,s_{(q,i)}(t)} = -c\left(\frac{1-\epsilon}{2\|Q\|\cdot p} + 1 \right)$$

We assume that the weights of all other connections to the state node are zero. The three equations then imply that the activation value of the state node is given by:

$$\vec{s}_{(q,i)}(t) = \sigma\left(c\cdot \vec{x}_i(t) + c\cdot \sum_{q'\in Q} \sum_{i'} \frac{1}{\|Q\|\cdot p} \vec{s}_{(q',i')}(t-1) - c\cdot\left(\frac{1-\epsilon}{2\|Q\|\cdot p} + 1 \right) \right).$$

We now evaluate this equation under three conditions: (1) the activation value of input node i is less than or equal to ϵ (i.e. $\cong 0$), (2) the activation value of all state nodes (q',i') is less than or equal to ϵ (i.e. $\cong 0$), and (3) the activation value of input node i is greater than or equal to 1-ϵ (i.e. $\cong 1$) and the activation value of at least one of the state nodes (q',i') is greater than 1-ϵ (i.e. $\cong 1$). Mathematically, these three conditions can be expressed as:

(C1) $\vec{x}_i(t) \le \epsilon$,

(C2) $\displaystyle\sum_{q'\in Q} \sum_{i'} \frac{1}{\|Q\|\cdot p} \vec{s}_{(q',i')}(t-1) \le \epsilon$,

(C3) $\vec{x}_i(t) \ge (1-\epsilon)$ and $\displaystyle\sum_{q'\in Q} \sum_{i'} \frac{1}{\|Q\|\cdot p} \vec{s}_{(q',i')}(t-1) \ge \frac{(1-\epsilon)}{\|Q\|\cdot p}$.

We can now evaluate the function that computes the state node's activation value for all three conditions:

(1) $\vec{s}_{(q,i)}(t) \le \sigma\left(c\cdot\epsilon + c\cdot\left(\frac{\epsilon-1}{2\|Q\|\cdot p} \right) \right)$,

(2) $\vec{s}_{(q,i)}(t) \leq \sigma\left(c \cdot \epsilon + c \cdot \left(\frac{\epsilon - 1}{2\|Q\| \cdot p} \right) \right),$

(3) $\vec{s}_{(q,i)}(t) \geq \sigma\left(-c \cdot \epsilon - c \cdot \left(\frac{\epsilon - 1}{2\|Q\| \cdot p} \right) \right).$

Recall that we wish to satisfy the equation:

$$(4)\quad \vec{s}_{(q,i)}(t) \overset{\epsilon}{=} \vec{x}_i(t) \wedge \left\{ \bigvee_{q' \in Q} \bigvee_{i'} \vec{s}_{(q',i')}(t-1) \right\}$$

Combining either equation (1) or (2) with equation (4) yields:

$$\sigma\left(c \cdot \epsilon + c \cdot \left(\frac{\epsilon - 1}{2\|Q\| \cdot p} \right) \right) \leq \epsilon < 0.5 .$$

Solving this equation for ϵ yields:

$$\epsilon < \frac{1}{2\|Q\| \cdot p + 1} .$$

And then solving the same equation for c yields:

$$c \geq \frac{\sigma^{-1}(\epsilon)}{\left(\epsilon + \frac{(\epsilon - 1)}{2\|Q\| \cdot p} \right)},$$

where σ^{-1} is the inverse of the squashing function. Similarly, combining equation (3) with equation (4) yields:

$$\sigma\left(-c \cdot \epsilon - c \cdot \left(\frac{\epsilon - 1}{2\|Q\| \cdot p} \right) \right) \geq 1 - \epsilon > 0.5 .$$

Solving for ϵ and c again gives:

$$\epsilon < \frac{1}{2\|Q\|\cdot p + 1}$$

and

$$c \geq \frac{\sigma^{-1}(\epsilon)}{\left(\epsilon + \frac{(\epsilon - 1)}{2\|Q\|\cdot p}\right)},$$

By restricting the values of c and ϵ in this manner, we have constructed a SLFOCC memory which computes the components of the state vector according to Equation (4). This in turn implies that whenever the input symbol and state symbol of the state transition function which is to be represented by the SLFOCC memory are encoded according to the functions $f_{i \to \bar{x}}$ and $f_{q \to \bar{s}}$ respectively, the SLFOCC memory will compute a new state vector which can be decoded using the function $f_{\bar{s} \to q}$ to yield the same state as the original state transition function would compute. Thus, if constructed as detailed here, a SLFOCC memory can represent the state transition function of any finite state automaton. This proves Theorem 4-3.□

Automaton Complexity

It is important to recognize that the use of state splitting increases the number of states required. Since the number of states determines the size of the state vector, which in turn determines the number of state nodes required, state splitting increases the number of nodes in the SLFOCC memory. Goudreau et. al. (1994) have pointed out that the state splitting scheme used here can be inefficient. They use the parity example described above to point out this inefficiency. If the parity automaton were to be implemented in a SLFOCC memory using state splitting, it would require four state nodes. But in the example above, we developed a SLFOCC memory which required only two state nodes.

This leads one to ask the question: how many state nodes (how large a state vector) are required to implement an automaton with n states? If we assume that state units function as threshold elements, then the lower bound is clearly $\log_2(n)$. Similarly, our proof above pegs the upper bound at $n \cdot p$ where p represents the number of input symbols.

Horne and Hush (1994a, 1994b) and Alon, Dewdney and Ott (1991) have both examined how many neurons are required to compute worst-case state transition functions. However, unlike our previous proof, these authors assumed that arbitrary numbers of computing layers are available between the input and context nodes and the state nodes. This is not the case for any of the memory functions discussed in this thesis other than LRSI memories. None-the-less, since SLFOCC memories are a special case of the arbitrary computing layer networks discussed by these authors implies that lower bounds derived by these authors still apply.

In particular, Alon, Dewdney and Ott (1991) show that $\Omega((n\log n)^{1/3})$ nodes are required to implement the most complex n state automata. The authors proceed by comparing the number of functionally different automata with n states, $L(n)$, to the number of functionally different automata which can be implemented by STCNs with m nodes, $U(m)$. Clearly, if an STCN is to be able to represent all n state machines, then it must be able to implement at least $L(n)$ functionally different automata. Thus, $U(m) \geq L(n)$ which implies: $m \geq U^{-1}(L(m))$. By deriving equations for $L(n)$ and $U(m)$ the authors are then able to compute a minimum number of nodes required to be able to implement arbitrary automata with n states. Specifically the authors prove:

$$U(m) \leq 2^{(m+1)^3}$$

$$L(n) \geq 2^{n\log n}$$

and conclude:

$$m \geq \Omega((n\log n)^{1/3})$$

This proves that the ideal case in which n states are implemented by a SLFOCC memory with $\log_2(n)$ nodes is not realizable using threshold nodes.

Combining Alon, Dewdney and Ott's (1991) result with that of Kremer (1995a), implies that the state transition functions of all automata with n states can be implemented by SLFOCC memories with m nodes where:

$$\Omega((n\log n)^{1/3}) \leq m \leq n \cdot p$$

The exact value for the minimum size of SLFOCC memory required to be able implement all automata with n states remains an open question.

4.4.3 Turing Machine Equivalence

To date, there are no results relating the computational power of a SLFOCC memory to that of a Pushdown automaton. However, there are results regarding Turing machine equivalence. We will consider only equivalence with finite numbers of units, since equivalence with infinite numbers of processors is trivial for a network capable of emulating a finite state automaton. Siegelmann and Sontag (1991) were the first to show the existence of a finite connectionist network, made up of sigmoidal nodes which simulates a universal Turing machine. Previous work had focussed only on higher-order connections. However, Siegelmann and Sontag's proof used only first order connections (thus making it applicable to SLFOCC memories). The authors use a simplified "sigmoid", called a saturated linear function, as their transfer function. This function is described by the following equation:

$$f(x) = \begin{cases} 0 & \text{if } x<0 \\ x & \text{if } 0 \leq x \leq 1 \\ 1 & \text{if } x>1 \end{cases}$$

The authors start by noting that any Turing machine can be simulated by a push-down automaton with three unary stacks. They then proceed to prove that one unit can be used to encode the value of each unary stack and that such a unit is capable of performing "pop" and "push" operations on the simulated stack using only first order recurrent connections. Lastly the authors prove that another node can "read" the value of the stack. Using these constructions, Siegelmann and Sontag are able to place an upper bound on the number of processors in a SLFOCC memory required in order to obtain the behaviour of a universal Turing machine. This upper bound is $N = 10^5$.

In a later paper (Siegelmann and Sontag, 1992), the authors extend their result to smaller networks which represent the infinite tape by simulating two stacks using two nodes in a SLFOCC memory (this representation is the same as that discussed in Section 3.5.4, above). Once again, only first order connections are used. However, in this proof, the authors also show that the number of computational steps required by the SLFOCC memory to compute an arbitrary computation is linear with respect to the number of steps required by a Turing machine to perform the same computation. The authors further show that a 2-stack machine can be implemented in a $12s+50$ node SLFOCC memory, where s is the number of states in the finite state stack controller. The authors further argue that Minsky's well known (1967) 4-symbol, 7-control state universal Turing machine can be simulated by a 2-symbol, two-stack, 84-state Pushdown automaton. This implies, that the universal Turing machine can be implemented in a 1058 processor SLFOCC memory. This order-of-magnitude difference from their previous result is due to a more efficient encoding of the tape in the state units of the network. The authors also point out that the

1058 processor network is also a conservative estimate on the number of units required to implement a universal Turing machine, and that it is likely that there exists an even better bound.

Both papers of Siegelmann and Sontag (1991, 1992) use a simplified sigmoid function, the saturated linear function. The derivative of this function is given by:

$$f'(x) = \begin{cases} 0 & \text{if } x<0 \\ 1 & \text{if } 0<x<1 \\ 0 & \text{if } x>1 \end{cases}$$

but is undefined at $x=0$ and $x=1$. This makes the function unsuitable for the standard gradient descent learning algorithm which requires an error function which is continuous with respect to connection weights. All practical SLFOCC memories use a continuously differentiable function instead. Recently, Kilian and Siegelmann (1993) have examined SLFOCC memories using such a function. They provide a rather complex proof of the universality of SLFOCC memories with, what the authors call, "valid sigmoids." A sketch of the proof follows: First, the authors show that SLFOCC memories are capable of simulating alarm clock machines. Next, they show that alarm clock machines can simulate adder machines which in turn are equivalent to counter automata. But, counter automata have previously been shown to be Turing machine equivalent. Thus, SLFOCC memories must also be Turing machine equivalent. (The details of the proof are omitted here. The interested reader is referred to Kilian and Siegelmann (1993).) The results presented by these authors indicate that "Turing universality is a relatively common property of recurrent neural network modes" (Kilian and Siegelmann, 1993, p. 137).

The computational equivalence of SLFOCC memories and Turing machines is a mixed blessing. On one hand, one can rejoice in the fact the computational power of a SLFOCC memory rivals that of any computing device. On the other hand, however, Gold's (1967) results imply that it will be impossible to train a SLFOCC memory using

only positive and negative example strings. As Horne and Hush (1994, p.2) point out, "These results are significant since they show that basic convergence questions concerning these architectures is undecidable even for fixed-size networks".

4.5 SINGLE-LAYER SECOND-ORDER CONTEXT COMPUTATION

4.5.1 Single-Layer Second Order Context Computation Memories Can Encode Any State Transition Function without State Splitting

Second order connections are more powerful than first order connections, so it should come as no surprise that SLSOCC memories can represent state more easily than SLFOCC memories. In particular, Goudreau et. al. (1994) have shown that if input symbols and states are encoded using unit normal vectors, then any finite state automaton can be encoded in a SLSOCC memory without state splitting. That is, an arbitrary finite state automaton's inputs, i_0, i_1, i_2, ... i_p,can be represented by input vectors, (1,0,0,...,0), (0,1,0,...,0), (0,0,1,...,0), ... , (0,0,0,...,1), respectively. The automaton's states, q_0, q_1, q_2, ... , q_n, can be represented by state vectors, (1,0,0,...,0), (0,1,0,...,0), (0,0,1,...,0), ... , (0,0,0,...,1), respectively. And it is possible to implement the automaton's state transition function, $\delta(q,s)$ by setting each of the second order weights, w_{ijk}, connecting input node i and context node j to state node k according to the function:

$$W_{ijk} = \begin{cases} -c & \text{if } \delta(q_j,i_i)=q_k \\ +c & \text{otherwise} \end{cases},$$

where c is a large constant. With these weights, the activation value of state node k will approach 1 if:

$$\sum_i \sum_j W_{ijk}\bar{x}_i(t)\bar{s}_j(t-1)>0\,,$$

and will approach 0 otherwise. This implies that the state node will assume a value of one if and only if:

$$\exists i,j \text{ s.t. } \delta(q_j, i_i) = q_k.$$

In other words, the state units of the SLSOCC memory will assume the value of the state vector q_k if and only if the previous state and current input are q_j and i_i respectively, and there is a legal transition from the previous state to the new state under the given input symbol. Thus, a SLSOCC memory is capable of implementing any finite state automaton using a unit normal vector encoding scheme for input symbols and states. Furthermore, this encoding of state is more efficient than the state splitting approach which must be used in a SLFOCC memory, requiring a number of state nodes equal to the number of states in the automaton. It should be noted, however, that this approach requires more weights than a SLFOCC memory.

4.5.2 Single-Layer Second-Order Context Computation Memories Cannot Encode Every State Transition Function with Binary Encoding

Ideally, of course, one would like to implement any n state automaton by a STCN using $\log_2 n$ state nodes. We now prove, however, that this cannot be accomplished by a SLSOCC memory:

Theorem 4-4: SLSOCC memories cannot encode all state transition functions using a binary state encoding.

Proof: To see that this is the case we consider one specific automaton with two input symbols and four states:

$$\Sigma = \{ s_0, s_1 \}$$

$$Q = \{ q_0, q_1, q_2, q_3 \}$$

and a transition function defined as follows:

$\delta(q_0, s_0) = q_1$ $\delta(q_0, s_1) = q_1$

$\delta(q_1, s_0) = q_2$ $\delta(q_1, s_1) = q_3$

$\delta(q_2, s_0) = q_3$ $\delta(q_2, s_1) = q_0$

$\delta(q_3, s_0) = q_0$ $\delta(q_3, s_1) = q_2$

In the ideal case an automaton with four states should be implementable in a STCN with $\log_2 4 = 2$ state nodes. If we assume binary state encodings then there must be a mapping from the states in Q to the set of vectors { (0,0), (0,1), (1,0), (1,1) }. We now note that the state transition function of this particular automaton implies that whenever the input symbol remains constant, the states cycle with a period of four. There exist six possible state vector cycles of length four which are illustrated in Figure 4-1.

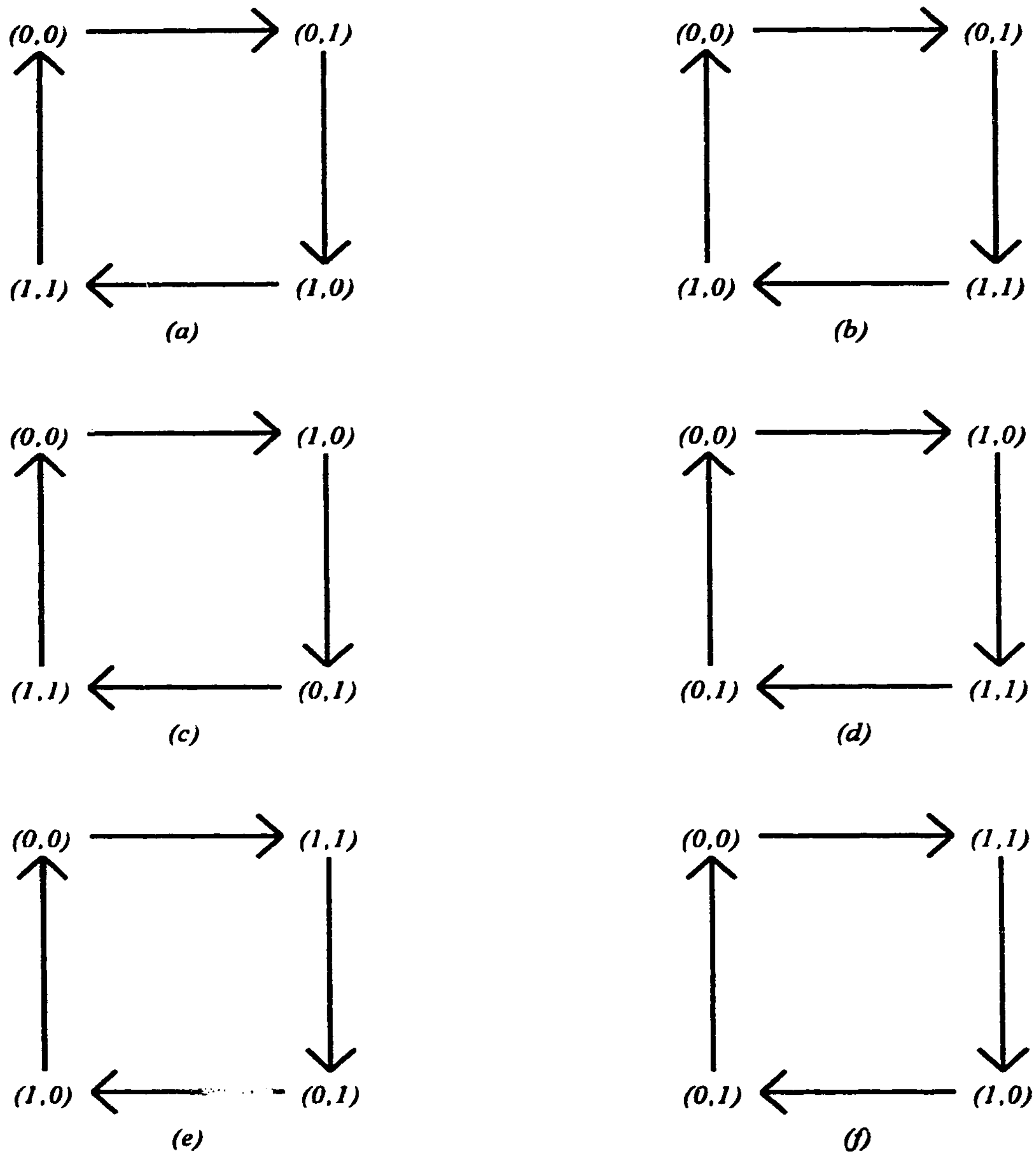

Figure 4-1: Possible cycles in a two-dimensional binary state space.

The fact that the input symbol is constant throughout the cycles implies that the input vector $\vec{x}(t)$, regardless of how it is encoded, must also remain constant. This, in turn, implies that the matrix product of the three-dimensional weight matrix and the input vector is a constant two-dimensional matrix for all input vectors; i.e.:

$$W' = W \times \vec{x}(t)$$

We now consider the activation vector of the state nodes, $\vec{s}(t)$. Since we assume that the new state does not depend on the input symbol:

$$\vec{s}(t) = f(W' \times \vec{s}(t-1))$$

But, this equation is now the activation equation for a single-layer, first-order network. And it is well known that such networks are not capable of computing functions which are not linearly separable.

Our next step is to create a table for each of the six cycles illustrated in Figure 4-1. These tables will relate the new state vector, $\vec{s}(t)$, to the previous state vector, $\vec{s}(t-1)$. By examining these six tables it will then be possible to determine that four of the cycles in Figure 4-1 cannot be implemented in a SLSOCC memory. The state transition tables for the cycles in Figure 4-1 are presented in Table 4-1. In Table 4-1(a), the first component of the next state vector forms a non-linearly separable pattern with respect to the two components of the previous state vector. Similarly, in Tables 4-1(c) and 4-1(e), the second component of the new state vector forms a non-linearly separable pattern. Finally, in Table 4-1(f), the first component of the new state vector is not linearly separable. Since the SLSOCC memory can only compute linearly separable state transitions under a constant input vector, neither of these four cycles can be used in the implementation of our automaton. This leaves only cycles (b) and (d) for consideration.

Since there are two different cycles in the automaton which we wish to implement, we must have two different cycles in the SLSOCC memory. Combining both of the

$\vec{s}(t-1)$	$\vec{s}(t)$
(0,0)	(0,1)
(0,1)	(1,0)
(1,0)	(1,1)
(1,1)	(0,0)

(a)

$\vec{s}(t-1)$	$\vec{s}(t)$
(0,0)	(0,1)
(0,1)	(1,1)
(1,0)	(0,0)
(1,1)	(1,0)

(b)

$\vec{s}(t-1)$	$\vec{s}(t)$
(0,0)	(1,0)
(0,1)	(1,1)
(1,0)	(0,1)
(1,1)	(0,0)

(c)

$\vec{s}(t-1)$	$\vec{s}(t)$
(0,0)	(1,0)
(0,1)	(0,0)
(1,0)	(1,1)
(1,1)	(0,1)

(d)

$\vec{s}(t-1)$	$\vec{s}(t)$
(0,0)	(1,1)
(0,1)	(1,0)
(1,0)	(0,0)
(1,1)	(0,1)

(e)

$\vec{s}(t-1)$	$\vec{s}(t)$
(0,0)	(1,1)
(0,1)	(0,0)
(1,0)	(0,1)
(1,1)	(1,0)

(f)

Table 4-1: Transition tables for the cycles depicted in the previous figure.

remaining legal cycles in one SLSOCC memory results in an automaton with the following transition function:

$\delta((0,0), \vec{x}_0) = (0,1)$ $\delta((0,0), \vec{x}_1) = (1,0)$

$\delta((0,1), \vec{x}_0) = (1,1)$ $\delta((0,1), \vec{x}_1) = (0,0)$

$\delta((1,0), \vec{x}_0) = (0,0)$ $\delta((1,0), \vec{x}_1) = (1,1)$

$\delta((1,1), \vec{x}_0) = (1,0)$ $\delta((1,1), \vec{x}_1) = (0,1)$

Clearly there exists no mapping from the states of the automaton we desire to implement, $\{ q_o, q_1, q_2, q_3 \}$, to the states a SLSOCC memory can implement with two binary state nodes, { (0,0), (0,1), (1,0), (1,1) }, which along with a mapping from the input symbols of the automaton $\{ s_0, s_1 \}$ to the input vectors of the SLSOCC memory $\{ \vec{x}_0, \vec{x}_1 \}$, preserves the original state transition function $\delta(q,s)$. This proves Theorem 4-4.□

Of course, it should be noted that while it is impossible, in general, for a SLSOCC memory to implement a state function using $\log_2 n$ nodes, this does not preclude individual automata implementations from achieving this level of efficiency. In general, however, the upper bound of at least n state nodes remains, using the unit normal vector encoding described above.

4.5.3 Turing Machine Equivalence

Finally, we turn our attention to SLSOCC memories and Turing machine equivalence. Pollack (1987) has shown that it is a relatively simple task to create a network which is Turing equivalent by using two units with continuous activation units which act as two binary stacks. His implementation is very similar to that presented in Section 3.5.4. Each stack node's activation value is computed by adding a decimal point before a binary string representing the stack contents. Pollack's STCN Turing machine relies on multiplicative (second-order) weights to gate the old value of each stack node during the computation of its new value. When a symbol is popped from the stack, the stack node's activation value must be doubled (after subtracting the symbol at the top of

the stack). When a symbol is pushed onto the stack, the stack node's activation value must be halved (after adding the new top symbol).

Thus, the stack node's old value must either be multiplied by 2 or by ½. Using second order weights it is possible to connect two control nodes to two connections from old stack node to new stack node. A "pop" node is connected with the context node containing the stack value to the state node containing the stack value with a weight of 2. The "pop" node's activation value is multiplied by the context node's value and by 2. Thus, when the "pop" node's activation value is 1, the state node receives a signal equal to twice the context node's activation. When the "pop" node's activation value is 0, the state node receives a signal of 0. Similarly, a "push" node is used to gate one half of the context node's activation value. Together, the "pop" and "push" node are thus able to perform any stack operations.

In the SLSOCC memory, the only second-order connections are from one context node and one input node to one state node. Since the "pop" and "push" nodes in Pollack's system must be context nodes, and since the stack node must be a context node, the SLSOCC memory cannot implement a Turing machine in this fashion. No other results regarding the computational power of SLSOCC memories of finite size have been published. Thus, their universality remains an open question. It is possible that the proofs in section 4.4.3 regarding the universality of SLFOCC memories could be adapted to SLSOCC memories as well, but the fact that SLSOCC memories have no biases or first order connections makes this translation non-trivial.

4.6 CONNECTIONIST PUSH-DOWN AUTOMATA (CPA) MEMORIES

4.6.1 Stack Limitations

Despite its name the CPA (connectionist Pushdown automaton) memory does not automatically have the computational powers of a Pushdown automaton. This is due to the fact that, as noted in Section 3.2.2, a Pushdown automaton is a non-deterministic device whereas the CPA memory is deterministic. Deterministic Pushdown automata (Hopcroft and Ullman, 1979) are computational devices whose power lies properly between finite state automata and (non-deterministic) Pushdown automata. The languages which can be represented by deterministic Pushdown automata are called deterministic context-free languages and contain the regular sets while being contained within the set of (non-deterministic) context-free languages. Deterministic context-free languages have proven to be a more useful set of languages than (non-deterministic) context-free languages for many applications. Virtually all computer languages belong to the set of deterministic context-free languages.

The fact that CPA memories use the same alphabet for their input symbols as for their stack does not limit their powers beyond the limitations of deterministic Pushdown automata. This is due to the fact that it is possible to translate a deterministic pushdown automaton $M=(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, to another deterministic automaton $M'=(Q', \Sigma, \Gamma', \delta', q_0', Z_0', F')$ which accepts the same strings, and has the property that $\Gamma'=\Sigma$. However, as Das et al. (1993) point out, the fact that only the current input symbol can be pushed onto the stack at any time, i.e.:

$$\delta(q,s,\gamma) = (q',s)$$

and the fact that there are no epsilon transitions does limit the power of CPA memories. Thus, such a memory is limited to implementing the same classes of languages that can be implemented by a deterministic Pushdown automaton which always pushes the same

symbol onto the stack as the input symbol (whenever a push action is performed) and has no epsilon transitions. To date, these languages have not been described in terms of grammatical rewrite rules.

4.6.2 Controller Limitations

We must now consider whether the languages which a CPA memory can represent are further restricted. The CPA memory consists of two components: a third-order network and an external analog stack. The network is responsible for computing whether a symbol should be popped from or pushed onto the stack as well as recomputing its own internal state. Thus, in order to represent arbitrary languages of the type which can be accepted by the specialized automaton described above, the network control mechanism must be able to issue arbitrary pop, push and no-operation instructions and make arbitrary state transitions. We now prove that this is in fact the case:

Theorem 4-5: A CPA memory is capable of implementing any stack operations which can be performed by a deterministic Pushdown automaton with no epsilon transitions and which only pushes the current input symbol onto the stack during a push operation.

Proof: The controlling network in a CPA memory is very similar to a SLSOCC memory. However, instead of using second order connections (connecting two nodes to a third), the network uses third order connections (which connect three nodes to a fourth). In particular, every input node is connected with every context node, and every read node to every state node. Similarly, every input node is connected with every context node and every read node to the action node. We assume that the deterministic Pushdown automaton's inputs, i_0, i_1, i_2, ... , i_p, (and hence stack symbols) can be represented by vectors that lie within ϵ along each dimension of the unit normal vectors (1,0,0,...,0),

(0,1,0,...,0), (0,0,1,...,0), ... , (0,0,0,...,1), respectively. The controller's states, q_0, q_1, q_2, ... , q_n, can be represented by state vectors that lie within ϵ along each dimension of the unit normal vectors (1,0,0,...,0), (0,1,0,...,0), (0,0,1,...,0), ... , (0,0,0,...,1), respectively. And, it is possible to implement the automaton's state transition function, $\delta(q,s,\gamma)$ by setting the each of the third order weights, W_{ijkl}, connecting input node i, context node j, and read node k to every state node l according to the function:

$$W_{ijkl} = \begin{cases} +c & \text{if } \exists\gamma \text{ s.t. } \delta(q_j,s_i,\gamma_k)=(q_l,\gamma) \\ -c & \text{otherwise} \end{cases},$$

where c is a large constant. With these weights, the activation value of state node l will exceed or match 1-ϵ if:

$$\sum_i \sum_j \sum_k W_{ijkl}\vec{x}_i(t)\vec{s}_j(t-1)\vec{r}_j(t) \geq \sigma^{-1}(1-\epsilon),$$

and will lie below or at ϵ if:

$$\sum_i \sum_j \sum_k W_{ijkl}\vec{x}_i(t)\vec{s}_j(t-1)\vec{r}_j(t) \leq \sigma^{-1}(\epsilon),.$$

This implies that the state node will assume a value of greater than or equal to 1-ϵ if and only if:

$$\exists i,j,k \text{ s.t. } \delta(q_j,i_i,\gamma_k)=q_l.$$

and if ϵ is sufficiently small, and c is sufficiently large. In other words, the state units of the CPA memory will assume the value of the state vector q_l if and only if the previous state, current input and symbol at the top of the stack are q_j, i_i, γ_k respectively, and there is a legal transition from the previous state to the new state under the given input and top

of stack symbols. Thus, a CPA memory is capable of implementing any state transition sequence using a unit normal vector encoding scheme for symbols and states.

The required restrictions on ϵ and c can be computed in a manner analogous to that used in the proof of Theorem 4-3. We do not show the computation of these values, here, as it does not provide any new information about the computational powers of a CPA memory.

We now turn our attention to the values of the action units since they also must be able to represent arbitrary stack operations (chosen from *push*, *pop*, and *no-operation*). We assume that the weights to action node l are defined:

$$W_{ijkl} = \begin{cases} +c & \text{if } \exists q \text{ s.t. } \delta(q_j,i_i,\gamma_k)=\{(q,i_i\gamma_k)\} \\ -c & \text{if } \exists q \text{ s.t. } \delta(q_j,i_i,\gamma_k)=\{(q,\epsilon)\} \\ 0 & \text{otherwise} \end{cases},$$

where c is a large constant. With these weights, the activation value of state node l will match or exceed an arbitrary threshold 1-ϵ if:

$$\sum_i \sum_j \sum_k W_{ijkl} \vec{x}_i(t) \vec{s}_j(t-1) \vec{r}_j(t) \geq \sigma^{-1}(1-\epsilon),$$

and will lie on or below the threshold $-1+\epsilon$ if:

$$\sum_i \sum_j \sum_k W_{ijkl} \vec{x}_i(t) \vec{s}_j(t-1) \vec{r}_j(t) \leq \sigma^{-1}(-1+\epsilon),$$

and will lie between $-\epsilon$ and $+\epsilon$ if:

$$\sigma^{-1}(-\epsilon) \leq \sum_i \sum_j \sum_k W_{ijkl} \vec{x}_i(t) \vec{s}_j(t-1) \vec{r}_j(t) \leq \sigma^{-1}(+\epsilon)$$

approach 0 otherwise. This implies that the state node will assume a value of one if and only if:

$$\exists i,j,k,q \text{ s.t. } \delta(q_j,i_i,\gamma_k)=(q,i_i\gamma_k)$$

and ϵ is sufficiently small, while c is suitably large. In other words, the action node of the CPA memory will signal a push operation if and only if the previous state, current input and symbol at the top of the stack are q_j, i_i, γ_k respectively, and a push operation is

indicated for the given input, state and top of stack symbols. Similarly, the state node will assume a value of negative one if and only if:

$\exists i,j,k,q$ s.t. $\delta(q_j,s_i,\gamma_k)=(q,\epsilon)$

In other words, the action node of the CPA memory will signal a pop operation if and only if the previous state, current input and symbol at the top of the stack are q_j, i_i, γ_k respectively, and a pop operation is indicated for the given input, state and top of stack symbols. Again, precise bounds on ϵ and c can be computed as was done in the proof of Theorem 4-3. This proves Theorem 4-5.□

While Das et al. (1993) recognized that a CPA memory is capable of implementing any stack operations which can be performed by a deterministic Pushdown automaton with no epsilon transitions and which only pushes the current input symbol onto the stack during a push operation, they did not formally prove this result.

4.6.3 Turing Equivalence

Thus far, we have assumed that the CPA memory's external stack operates like the stack of a conventional Pushdown automaton. While a CPA memory's stack certainly can operate in this fashion (when all symbols on the stack are of length 1, for example), the CPA memory's stack is a continuous stack. Although the use of a continuous stack is strictly a consequence of the need for a differentiable error function, it is possible that, in creating such a modified device, additional computational power becomes available. This additional power could come from two sources. First, there is the fact that the CPA memory can read more than one symbol from its stack during any time step. This is possible because whenever the widths of the symbols on the stack are less than one, the activations of the read nodes are computed based on a weighted average of vectors corresponding to the first few symbols on the stack making a total width of one. The

second potential source of additional power is the fact that in addition to storing "just" symbols, the stack also stores analog widths. It is possible that the widths themselves could store some useful information which might be exploited by the CPA memory. These two differences between the operation of the CPA memory's stack, and the operation of the stack of a conventional Pushdown automaton are only potential advantages. To date, the consequences of the continuous stack have not been properly examined. It is certainly possible that they will offer no advantages.

Thus far, we have assumed that any power a CPA memory might have beyond that of a traditional finite state automaton must be derived from CPA memory's external stack. However, there is no known reason why a CPA memory could not exploit the continuous nature of the activation values of its state units to provide additional power in the same fashion as a SLFOCC memory can achieve Turing equivalence. Unfortunately, the fact that the CPA memory has only third order connections makes any proofs of computational power for SLFOCC memories (e.g. Kilian and Siegelmann, 1993) not directly transferable to this particular type of memory. It is possible that a translation of such a proof from SLFOCC memories to SLSOCC memories might shed some light on further translations to CPA memories.

4.7 CONNECTIONIST TURING MACHINE (CTM) MEMORIES

A CTM memory consists of a SLFOCC memory control mechanism connected to a discrete infinite tape mechanism. In order to perform any computation which can be performed by a Turing machine, the SLFOCC controller must be able to arbitrarily write symbols to the tape as well as move the position of the tape head relative to the tape. A SLFOCC memory is known to be able to implement any finite state automaton (including the one which properly controls the tape head for a given Turing machine), but only if state splitting is used. This implies that further computation maybe required to re-join the

split states. The extra computation can be performed using an extra time step. That is, if tape head writes and movements only occur every other time step, then the extra time step effectively provides an additional layer of state units which can be used to rejoin split states. Thus, a CTM memory must be capable of implementing arbitrary Turing machines using this two time step approach.

4.8 LOCALLY RECURRENT STATE AND INPUT

4.8.1 Constant Input

Although the LRSI memory has an adaptable state function like the SLFOCC and SLSOCC memories, the types of automata which can be represented have some surprising limitations resulting from the local-only recurrence and the monotonic activation function. Giles et al. (1995) have proven that a LRSI memory is incapable of implementing automata which have cycles of length greater than two under constant input. That is, given a sequence of constant input symbols over time $I(t) = I(t+1) = I(t+2) = \ldots$, a LRSI memory will always go through a sequence of states $q(t)$, $q(t+1)$, $q(t+2)$, ... where:

$$\forall t \;\; q(t) = q(t+2)$$

This conclusion follows from the fact that in an LRSI memory, whenever the input vectors remain constant, $\vec{x}(t)=\vec{x}(t+1)=\vec{x}(t+2)=\ldots$ the state vectors can never go through a sequence of values, $\vec{s}(t)$, $\vec{s}(t+1)$, $\vec{s}(t+2)$, ... where:

$$\forall t \;\; \vec{s}(t) \neq \vec{s}(t+2)$$

Giles et al. (1995) prove this result inductively. We now present this result in a notation consistent with the rest of this thesis. Specifically we prove:

Theorem 4-6: A LRSI memory is incapable of representing any FSA whose states cycle with a period of greater than two under constant input.

Proof: First, we consider the activation value of the first state node which is not an input node, $i=\|\vec{x}\|+1$. This noae s activation values is computed (see Section 2.5.1):

$$\vec{s}_i(t) = \sigma(\ W[1..(i+1)][i]\times\{1\oplus\vec{s}(t)[1..i-1]\oplus\vec{s}(t-1)[i]\}\ )$$

Since the input vector is constant, we can let $\Lambda=W[1..(\|\vec{x}\|+1)][i]\times\{1\oplus\vec{x}(t)\}$, and simplify the activation equation to:

$$\vec{s}_i(t) = f(\ \Lambda+W[i][i]\cdot\vec{s}_i(t-1)\ )$$

We now consider the points at which this equation intersects the diagonal:

$$\vec{s}_i(t)=\vec{s}_i(t-1)$$

The points of intersection represent the fixed points of the state node's activation function.

There are three ways in which the equations can intersect. If $W[i][i]>0$, then the straight line can intersect the curve at one point. We shall call this Case (a); it is depicted in Figure 4-2(a). Alternately, still assuming $W[i][i]>0$, the straight line can intersect the curve at three points. We shall call this Case (b); it is depicted in Figure 4-2(b). Finally, if $W[i][i]<0$, then the straight line must intersect the curve at one point. We call this Case (c), depicted in Figure 4-2(c).

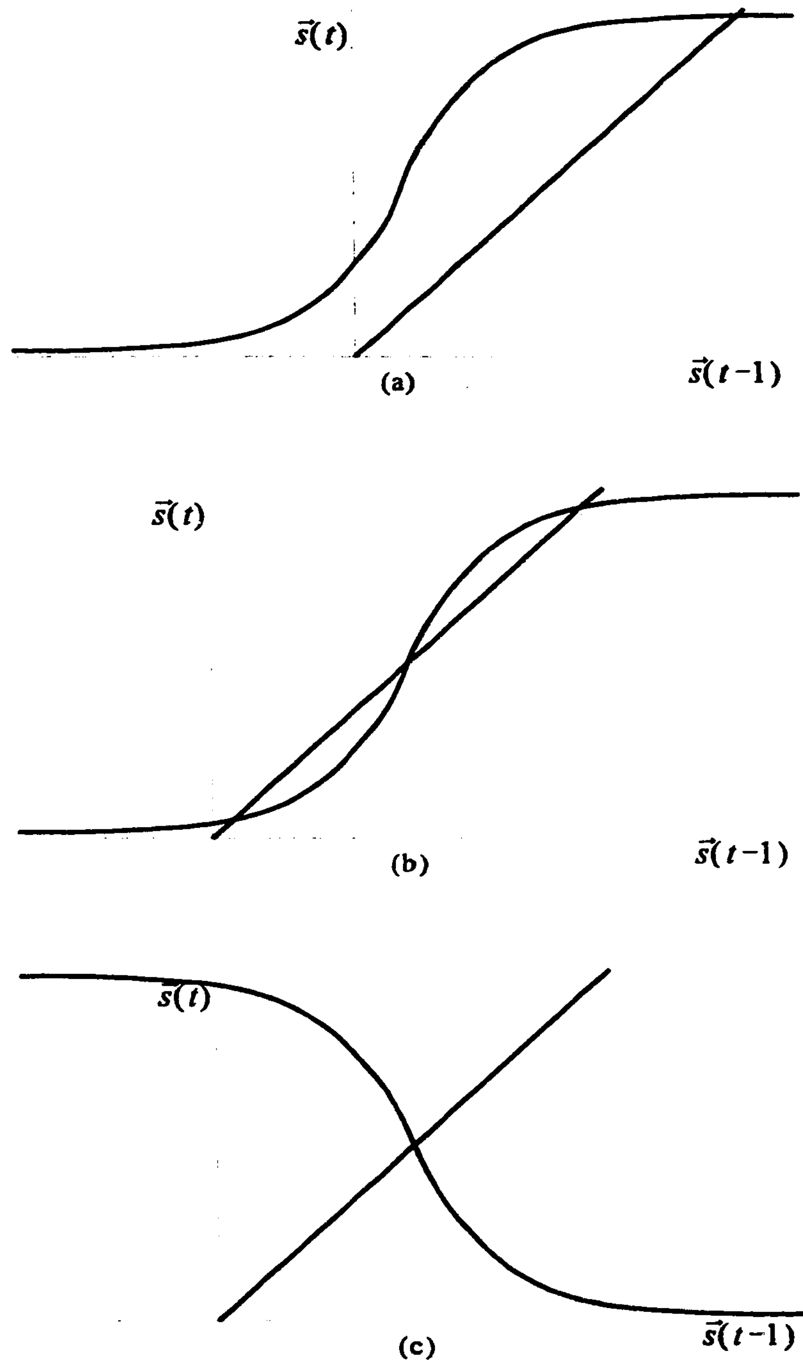

Figure 4-2: Intersections between node activation and diagonal functions. (a) For $W[i][i] > 0$, and one point of intersection. (b) For $W[i][i] > 0$, and three points of intersection. (c) For $W[i][i] < 0$.

In Case (a), $f(\ \Lambda+W[i][i]\cdot\vec{s}_i(t-1)\ )>\vec{s}_i(t-1)$ to the left of the fixed point, while $f(\ \Lambda+W[i][i]\cdot\vec{s}_i(t-1)\ )<\vec{s}_i(t-1)$ to the right of the fixed point. This, and the fact that the function is monotonically increasing, implies that the sequence $\vec{s}_i(t)$, $\vec{s}_i(t+1)$, $\vec{s}_i(t+2)$, ..., must converge monotonically towards the fixed point.

Similarly, for Case (b), $f(\ \Lambda+W[i][i]\cdot\vec{s}_i(t-1)\ )>\vec{s}_i(t-1)$ to the left of the first fixed point and between the second and third fixed points, while $f(\ \Lambda+W[i][i]\cdot\vec{s}_i(t-1)\ )<\vec{s}_i(t-1)$ between the first and second fixed points and to the right of the third fixed point. This, and the fact that the function is monotonically increasing, implies that the sequence $\vec{s}_i(t)$, $\vec{s}_i(t+1)$, $\vec{s}_i(t+2)$, ..., must converge monotonically towards the first fixed point whenever $\vec{s}_i(t)$ lies to the left of the central fixed point. Similarly, the sequence must converge monotonically towards the third fixed point whenever $\vec{s}_i(t)$ lies to the right of the central fixed point.

For Case (c), the sequence $\vec{s}_i(t)$, $\vec{s}_i(t+1)$, $\vec{s}_i(t+2)$, ..., can oscillate as it cycles around the single fixed point. This is due to the fact that, the curve is a monotonically decreasing function and to the left of the fixed point $f(\ \Lambda+W[i][i]\cdot\vec{s}_i(t-1)\ )>\vec{s}_i(t-1)$, while to the right of the fixed point $f(\ \Lambda+W[i][i]\cdot\vec{s}_i(t-1)\ )<\vec{s}_i(t-1)$. In order to examine the nature of these oscillations we must consider the fixed points of the equation:

$$\vec{s}_i(t)\ =f(\ \Lambda+W[i][i]\cdot f(\ \Lambda+W[i][i]\cdot\vec{s}_i(t-2)\ )\ )$$

This function is monotonically increasing (even for negative $W[i][i]$). Thus, it can have fixed points like to those illustrated in Figures 4-2(a) and 4-2(b). As argued above, this implies that the sequence $\vec{s}_i(t)$, $\vec{s}_i(t+2)$, $\vec{s}_i(t+4)$, ..., (note the increments of two, this time) must converge monotonically to a fixed point. Since the values of $\vec{s}_i(t)$ converge to one fixed point on even time steps, and converge to another fixed point on odd time steps, the activation value of state node i will oscillate with period two.

We have now shown that the activation value of state node $i=\|\vec{x}\|+1$, under constant input, can either converge to a fixed value or oscillate with a period of two. The

activation value can never oscillate with a period greater than two. We must now prove that the same holds for all other state nodes, $i > \|\vec{x}\| + 1$. We do this by means of an inductive proof. First, we use the case $i = \|\vec{x}\| + 1$ as a basis. Next, we assume that all state nodes, i, oscillate with a period of at most two as our inductive hypothesis. Finally, we must prove that this implies that node $i+1$ oscillates with a period of at most two. This node's activation value is computed (see Section 2.5.1):

$$\vec{s}_{i+1}(t) = f(\ W[1..i+1][i+1] \times \{\vec{s}(t)[1..i] \oplus \vec{s}_{i+1}(t-1)\}\ )$$

Using the inductive hypothesis, we let:

$$\lambda_0 = W[1..i][i+1] \times \vec{s}(t)[1..i] \quad \text{for even } t$$

and

$$\lambda_1 = W[1..i][i+1] \times \vec{s}(t)[1..i] \quad \text{for odd } t$$

And simplify the state node's activation equation to:

$$\vec{s}_{i+1}(t) = f(\ \lambda_0 + W[i+1][i+1] \cdot f(\lambda_1 + W[i+1][i+1] \cdot \vec{s}_{i+1}(t-2)\ )\ ) \qquad \text{for even } t$$

Again, this is a monotonically increasing function (even for negative $W[i][i]$), so the sequence $\vec{s}_{i+1}(t)$, $\vec{s}_{i+1}(t+2)$, $\vec{s}_{i+1}(t+4)$, ... for even t must approach a fixed point. A similar argument can be made for the same sequence for odd t. Thus, the activation value of state node i can cycle with period at most two. This completes our re-presentation of Giles et al.'s proof of Theorem 4-6.□

Since none of the nodes in the state vector can oscillate with a period of greater than two under constant input, the state vector $\vec{s}(t)$, must also oscillate with a period of at most two under these conditions. Clearly this imposes some strong restrictions on the types of automata a LRSI memory can implement. Nonetheless, Giles et al. (1995) discovered that there appear to be other automata, not accounted for by this theoretical result, that LRSI memories appear unable to learn in practise. The authors noted that, in numerical simulations, a LRSI memory was unable to learn a simple automaton which

accepts strings defined over the alphabet of 0s and 1s which have even parity. That is, LRSI could not learn the automaton illustrated in Figure 3-1. The authors were careful to note that their theorem about cycles of length greater than two does not apply to the parity automaton, since it cycles through one state under constant "0" input, and through two states under constant "1" input, but never through three, or more states under constant input.

4.8.2 Oscillating Input[1]

Giles et al.'s (1995) observations lead to two possible conclusions. First, the difficulty with parity could be caused by *in practise* limitations of the particular learning algorithm used by the authors. That is, even though the hypothesis space of the LRSI memory architecture contains a solution to the parity problem, the learning algorithm which explores that space does not find it. Second, the LRSI memory architecture could provide an *in principle* reason why this particular automaton cannot be represented. We shall now prove the latter to be the case. More specifically, we shall prove that there exists another large class of automata (besides those already identified by Giles et al.) which cannot be represented within the constraints of a LRSI memory.

Theorem 4-7: A LRSI memory is incapable of representing finite state automata whose state transitions form cycles of length greater than two under oscillating input.

Proof: We use a proof by contradiction. First, we assume that there exist LRSI memories capable of representing finite state automata whose state transitions form cycles of length greater than two under oscillating input. We then prove that such a memory can be easily

[1]A version of this section has been submitted for publication. Kremer 1995b.

augmented to form a larger LRSI memory which oscillates with period of two under constant input. Since such an LRSI memory cannot exist (Giles et al.'s (1995) proof), the premise must be false, and hence the contradiction of the premise (the theorem above) must be true.

We assume that the weights $W[1..n][1..n]$ define a LRSI memory whose state vector, $\vec{s}(t)$, oscillates with a period of greater than two, when the input vector, $\vec{x}(t)$, to the memory oscillates with a period of two. Next we create a second LRSI memory, with a constant input vector $\vec{x}\,'$, whose weights W' are defined as follows:

$$W'[i][j] = \begin{cases} 1 & \text{if } i \le \|\vec{x}\,'\| \text{ and } j > \|\vec{x}\,'\| \\ -1 & \text{if } \|\vec{x}\| < i \le \|\vec{x}\,'\| + \|\vec{x}\| \text{ and } i = j \\ 0 & \text{if } \|\vec{x}\| < i \le \|\vec{x}\,'\| + \|\vec{x}\| \text{ and } i < j \le \|\vec{x}\,'\| + \|\vec{x}\| \\ W[i - \|\vec{x}\,'\|][j - \|\vec{x}\,'\|] & \text{if } i > \|\vec{x}\,'\| \text{ and } j > \|\vec{x}\,'\| + \|\vec{x}\| \text{ and } i \le j \\ \text{undefined} & \text{otherwise} \end{cases}$$

Note that the definition of these weights does not violate any of the constraints on LRSI memories. We now consider the operation of this constructed LRSI memory. First, we consider state nodes in the range $i < \|\vec{x}\,'\|$. The activation values of these nodes are computed:

$$\vec{s}_i(t) = \vec{x}_i$$

and thus remain constant.

Next, we consider the activation values of state nodes, i, in the range $\|\vec{x}\,'\| < i \le \|\vec{x}\,'\| + \|\vec{x}\|$. The activation values of these nodes are computed:

$$\vec{s}_i(t) = f(\ W'[i] \times \{\vec{s}(t)[1..i-1] \oplus \vec{s}_i(t-1)\}\ )$$

Using the values of W' defined above, this equation can be simplified to:

$$\vec{s}_i(t) = f(\ \lambda - \vec{s}_i(t-1)\ )$$

where $\lambda = W[i][1..\|\vec{x}'\|]\times\vec{x}'$. As described above, an activation equation of this form will cause the node's activation to oscillate between two values. Thus, all state nodes, *i*, in the range $\|\vec{x}'\|<i\leq\|\vec{x}'\|+\|\vec{x}\|$ will oscillate with period two.

Finally, we turn our attention to state nodes, *i*, in the range $\|\vec{x}'\|+\|\vec{x}\|<i$. Based on our construction of the matrix *W'* above, these state nodes will behave identically to the state nodes in the original LRSI memory defined *W* when its inputs, $\vec{x}(t)$, are defined:

$$\vec{x}(t) = \vec{s}(t)[\|\vec{x}'\|+1...\|\vec{x}'\|+\|\vec{x}\|]$$

But, we know (from the previous paragraph) that this input vector will oscillate with period two. And, we also know (by the premise of our proof by negation) that the original LRSI memory oscillates with a period of greater than two when presented with an input vector which oscillates with a period of two. Thus, the constructed LRSI memory must oscillate with a period of greater than two, under constant input. But, Giles et al. (1995) already proved that no such LRSI memory can exist. This implies that our premise must be incorrect, and hence that our original theorem must be true. In other words: A LRSI memory is incapable of representing finite state automata whose state transitions form cycles of length greater than two under oscillating input.□

This new theoretical result applies, among many others, to the parity automaton discussed by Giles et al.. While those authors were only able to show **empirically** that parity appears not the be learnable by a LRSI memory, we were able to prove that there exist **theoretical** reasons which make it impossible for LRSI memories to implement parity. Interestingly, our proof by negation uses their result concerning cycles under constant input to show that a similar proof exists for cycles under oscillating input.

4.8.3 Arbitrary Cycles[1]

Giles et al.'s (1995) proof can be viewed as defining the types of state cycles which cannot be represented by a LRSI memory when the input oscillates with a period of one (i.e. remains constant). The new proof in Section 4.8.2 can be viewed as defining the types of state cycles which cannot be represented when the input oscillates with a period of two. It is natural to ask whether further proofs for state cycles can be developed for input cycles of greater than two. In particular, it would be desirable to define a function mapping the period of the input signal to the types of automata cycles which can be represented. We now develop such a generic proof. We begin by proving the following Lemma:

Lemma 4-1: If $\lambda(t)$ oscillates with even period, n, or if $W[i][i]>0$, then state node i must oscillate with a period l, where $n \bmod l = 0$.

Proof: First, we define the activation equation for state node i as:

$$\vec{s}_i(t) = f(\ \lambda(t) + W[i][i] \cdot \vec{s}_i(t-1)\ )$$

where:

$$\lambda(t) = W[1..i-1][i] \times \vec{s}(t)[1..i-1]$$

we now assume that $\lambda(t)$ oscillates with a period of n. Thus, we can define: $\lambda(t) = \lambda_{t \bmod n}$ and rewrite the activation equation of state node i as:

[1] A version of this section has been accepted for publication. Kremer, in press.

$$\vec{s}_i(t) = f(\ \lambda_{t \bmod n} + W[i][i] \cdot f(\ \lambda_{(t-1) \bmod n} + W[i][i] \cdot f(\ \lambda_{(t-2) \bmod n} + W[i][i] \cdot \ddots f(\ \lambda_{(t-n+1) \bmod n} + W[i][i] \cdot \vec{s}_i(t-n)\ ) \cdots)\ )\ )$$

If $W[i][i] > 0$, or if n is even, then $\vec{s}_i(t)$ is monotonically increasing with respect to $\vec{s}_i(t-n)$. This implies (as argued above) that the sequence $\vec{s}_i(t)$, $\vec{s}_i(t+n)$, $\vec{s}_i(t+2n)$, ..., converges to a single fixed point. Thus, $\vec{s}_i(t)$ oscillates with period at most n. Furthermore, its period of oscillation must be a factor of n since every n^{th} value is the same. Thus, $\vec{s}_i(t)$ must oscillate with a period, l, where $n \bmod l = 0$. This proves Lemma 4-1.□

Next we consider the case for $\lambda(t)$ oscillating with odd period. Specifically, we prove:

***Lemma 4-2*:** If $\lambda(t)$ oscillates with odd period, n, and if $W[i][i] < 0$, then state node i must oscillate with a period l, where $2n \bmod l = 0$.

***Proof*:** If $W[i][i] < 0$ and n is odd, then we must expand the activation equation for state node i further:

$$\vec{s}_i(t) = f(\ \lambda_{t \bmod n} + W[i][i] \cdot f(\ \lambda_{(t-1) \bmod n} + W[i][i] \cdot f(\ \lambda_{(t-2) \bmod n} + W[i][i] \cdot \ddots f(\ \lambda_{(t-2n+1) \bmod n} + W[i][i] \cdot \vec{s}_i(t-2n)\ ) \cdots)\ )\ )$$

Under this expansion, $\vec{s}_i(t)$ is monotonically increasing with respect to $\vec{s}_i(t-2n)$. This implies (as argued above) that the sequence $\vec{s}_i(t)$, $\vec{s}_i(t+2n)$, $\vec{s}_i(t+4n)$, ..., converges to a single fixed point. Thus, $\vec{s}_i(t)$ oscillates with period at most $2n$. Furthermore, its period of oscillation must be a factor of $2n$ since every $(2n)^{th}$ value is the same. Thus, $\vec{s}_i(t)$ must oscillate with a period, l, where $2n \bmod l = 0$. This proves Lemma 4-2.□

Lemmas 4-1 and 4-2 relate the rate of oscillation of $\lambda(t)$ to that of state node i. However, we wish to know the relation between the rate of oscillation of the LRSI memory's input signal and the entire state vector. To do this, we again consider both even and odd cases, even first:

Lemma 4-3: If the input symbol to a LRSI memory oscillates with even period, n, then the activation of all state nodes must oscillate with a period of l, where $n \bmod l = 0$.

Proof: We use a proof by induction on the state node number. For state node $i = \|\vec{x}\| + 1$:

$$\lambda(t) = W[1..i][i] \times \vec{s}(t)[1..i-1] = W[1..i][i] \times \vec{x}(t).$$

This implies that $\lambda(t)$ oscillates with a period of n, and, by Lemma 4-1, that $\vec{s}_i(t)$ oscillates with a period of l, where $n \bmod l = 0$. This forms the basis for our induction. Next we assume that Lemma 4-3 holds for all state nodes less than i. This implies that $\lambda(t)$ must oscillate with a period of exactly n (since the input vector is a component of $\lambda(t)$). Theorem 4-1 then implies that the activation value of state node i must oscillate with a period of l, where $n \bmod l = 0$, which completes the proof.□

We now prove the following lemma for the odd case:

Lemma 4-4: If the input symbol to a LRSI memory oscillates with odd period, n, then the activation of all state nodes must oscillate with a period of l, where $2n \bmod l = 0$.

Proof: Again, we use a proof by induction on the state node number. For state node $i = \|\vec{x}\| + 1$:

$$\lambda(t) = W[1..i][i] \times \vec{s}(t)[1..i-1] = W[1..i][i] \times \vec{x}(t).$$

This implies that $\lambda(t)$ oscillates with a period of n, and, by Lemma 4-2, that $\vec{s}_i(t)$ oscillates with a period of l, where $2n \bmod l = 0$. This forms the basis for our induction. Next, we assume that Lemma 4-4 holds for all state nodes less than i. This implies that $\lambda(t)$ must oscillate with a period of either n or $2n$. Lemma 4-1 then implies that the activation value of state node i must oscillate with a period of l, where $2n \bmod l = 0$, since n is odd, and $2n$ is even. This proves Lemma 4-4.□

We can now combine Lemma's 4-3 and 4-4 to conclude:

Theorem 4-8: If the input signal to a LRSI memory oscillates with period, n, then the LRSI memory can represent only those finite state automata whose state transitions form cycles of length l, where $n \bmod l = 0$ if n is even and $2n \bmod l = 0$ if n is odd.

Giles's original proof for input cycles of length $n = 1$ (constant input) and the new proof presented here for input cycles of length $n = 2$ can now be viewed as special cases of this theorem. It is interesting to note that Fahlman's (1991) original empirical experiments with LRSI memories did not reveal these important shortcomings of his STCN design. This adds weight to the central argument of this thesis that formal analyses of grammatical induction in a connectionist paradigm are important. Of course, Giles et al.'s (1995) observations regarding the parity automaton precipitated the formal analyses that

led to the proofs described here, but while these results only concerned a single automaton, the new proofs cover an infinite number of classes of automata (for each cycle size) each of which in turn contains an infinite number of individual automata that have been proven unrepresentable in a LRSI memory.

4.9 OUTPUT FUNCTIONS

Having examined the types of states transitions which can be implemented by various state functions, we now turn our attention to the output function. Showing that a state function can implement arbitrary automata is of little use if one cannot also show that a particular STCN design is also capable of mapping the state to a desired output. The latter task is implemented by the output function of the STCN. In the previous chapter we defined three possible output functions: 0-Layer, 1-Layer and 2-Layer.

4.9.1 0-Layer Output Function

A 0-Layer output function (see Section 2.5.2) is nothing more than a filter which is applied to the state vector. Thus, any outputs computed by an STCN employing a 0-Layer output function must already be present in the state vector. For this reason, a 0-Layer output function lends itself to be used in conjunction with only the most representationally powerful memory functions. In the previous sections we discovered that only SLSOCC and CPA memories allow arbitrary finite state transitions to be represented without state splitting. Goudreau et al. (1994) showed that a SLSOCC memory can be given a special "output" node which can be used to render a grammaticality judgement. This node receives second order connections from the input and context nodes of the STCN. If the input and context nodes use a unit normal vector encoding scheme, then the output node can realize arbitrary mappings from input symbol and context to values 0 and 1 (just as the state nodes can realize arbitrary mappings). Interestingly, the special

"output" node can be one of the state nodes without violating any of the connectivity constrains of the SLSOCC memory (assuming that all connections leading out from the "output" node of the context units have a weight of zero).

Of course there is no reason why only one such output node can be employed. If multiple output nodes are used, then output are no longer restricted to boolean grammaticality judgements, but rather can implement the types of output functions of Moore Machines (see Section 3.4.4). This implies that the state vectors of a SLSOCC memory are not only able to compute arbitrary state transition functions, but also arbitrary output functions. Therefore, it is possible to use a 0-Layer output function to extract the activation values of the "output" nodes from the activation values of those state nodes which actually represent state. An analogous argument can be made for CPA memories.

A 0-Layer output function is also appropriate for LRSI memories (although in practise 1-Layer output functions have been used—see Fahlman, 1991; Giles et al., 1995). This is because the cascaded connectivity scheme of the state nodes in an LRSI memory just like the non-recurrent version of RCC is capable of performing the same computations as any other given network (regardless of node connectivity pattern, number of layers, etc.). Thus, adding additional layers does not provide any advantage in computational power.

4.9.2 1-Layer Output Function

STCNs employing less powerful memory functions must use more powerful output functions in order to implement arbitrary automata. We have seen in Section 4.2 that SLFOCC memories can only implement arbitrary state transitions if state splitting is employed. This restriction makes the computation of arbitrary output functions much more difficult. We shall first prove that:

Theorem 4-9: A 0-Layer output function cannot be used to compute arbitrary output functions based on the states computed by a SLFOCC memory.

Proof: The proof relies on showing that it is impossible to add special state nodes (which would be selected by the 0-Layer output function) to a SLFOCC memory to compute arbitrary output functions. We consider the example of the parity automaton discussed in Section 4.4.2. This time we assume that the automaton is to output one symbol for all even strings and a different symbol for all odd strings. Without loss of generality, we assume that there exists one node, i, for which the output vectors differ. This implies that node i must assume one value whenever the input signal is "0" and the state signal is in an "even" state or whenever the input signal is "1" and the state signal is in an "odd" state and node i must assume a different value for the other two cases. However, this relation is a linearly inseparable function (OR). This implies that the monotonic activation function employed by node i will not be powerful enough to compute the desired function, which in turn implies that a 0-Layer output function will not be powerful enough to map the state computed by a SLFOCC memory to an arbitrary output, which completes the proof.□

We now prove that:

Theorem 4-10: A 1-Layer output function, together with a SLFOCC memory can emulate any Moore Machine (and hence any finite state automaton).

Proof: Recall that a Moore Machine consists of a six-tuple $(Q,\Sigma,\Delta,\delta,\lambda,q_0)$. We assume that whenever the Moore Machine which is to be emulated outputs symbol b_k the output vector $\vec{y}(t)$ has all its components less than or equal to ϵ with the exception of the kth component whose value is at least 1-ϵ (i.e. we use a unit normal encoding scheme for the

output vector, with a tolerance of ϵ). We let $\Xi(j)$ represent the set of states in the automaton whose state have been split which correspond to state j of the original automaton. Note that due to state splitting, more than one state in the split automaton may be used to represent a single state in the original. In order to correctly compute the output symbol the output function must compute the correct vector $\vec{y}(t)$ for all state vectors $\vec{s}(t)$. Assuming unit normal encodings for states (after splitting) within ϵ, this can be done by computing the kth component of the output vector according to the equation:

$$y_k(t) \stackrel{\epsilon}{=} \bigvee_{j\in J} \bigvee_{i\in\Xi(j)} s_i(t) \quad \text{where } J=\{j\,|\,\lambda(q_j)=b_k\},$$

where $\stackrel{\epsilon}{=}$ denotes equality under the assumption that values in the range $[0,\epsilon]$ are interpreted as being equal to 0, and values in the range $[1-\epsilon,1]$ are interpreted as 1. We now assume that the weights from all the nodes in $\Xi(j)$ to node k are all equal to a positive constant c, and that the bias of the node is $-c(1-\epsilon-\|\Xi(j)\|\epsilon)/2$. There are to cases to consider for the values of the state nodes: (1) the activation values of all state nodes in $\Xi(j)$ are less than or equal to ϵ and (2) at least one of the activation values of all state nodes in $\Xi(j)$ is greater than or equal to $1-\epsilon$. In case 1, the activation of output node k is computed:

$$y_k(t) \le \sigma\left(\|\Xi(j)\|\cdot c\cdot\epsilon - \frac{c\cdot(1-\epsilon-\|\Xi(j)\|\cdot\epsilon)}{2} \right), \qquad [1]$$

while in case 2, it is computed

$$y_k(t) \ge \sigma\left(c\cdot(1-\epsilon) - \frac{c\cdot(1-\epsilon-\|\Xi(j)\|\cdot\epsilon)}{2} \right). \qquad [2]$$

In order to conform with our encoding scheme, equation 1 must yield a value less than or equal to ϵ, whereas equation 2 must yield a value of greater than $1-\epsilon$. Solving for ϵ and c under these conditions gives:

$$\epsilon < \frac{1}{\|\Xi(j)\|-1}$$

and:

$$c \le \frac{2\sigma^{-1}(\epsilon)}{\|\Xi(i)\|\cdot\epsilon-1+\epsilon}$$

This implies that a 1-Layer output function can compute the output of an arbitrary Moore Machine given the state computed by a SLFOCC memory and proves Theorem 4-10.□

4.9.3 2-Layer Output Function

The representationally weakest STCN memory functions are WIT and CIIR, so it should come as no surprise that these functions require the most powerful of output functions to maximize their descriptive adequacy. Since both of these memory functions allow for no flexibility (other than adjusting window sizes) in defining what is computed in the state vector, output functions used with these memories must be able to compute arbitrary boolean functions in order to realize the automata and languages described in Sections 4.2 and 4.3 as the limits of their computational power. Two layer networks (not counting the input layer) are known to be universal function approximators and able to implement arbitrary boolean functions with finite numbers of nodes. This implies that 2-Layer output functions will be able to realize the maximum potential of WIT and CIIR memories.

4.10 SUMMARY

In this chapter we have seen that choosing one particular type of memory for a STCN rather than another influences the types of computational machines which can be emulated and hence the types of formal languages which can be implemented. We have also discovered that the representational capacity of a fixed size state vector is limited, and depends on the particular memory design. It follows that a judicious choice of memory system and state vector size can be used to limit the hypothesis space of a grammatical induction task. We have also shown that the representation of state varies from memory to memory and that certain types of output functions can maximize the representational power of the resulting STCN.

Table 4-2 summarizes the number of state nodes and the type of output function required by each STCN memory function to implement each formal computing machine. Question marks represent open research problems. This table can now be used to select the most appropriate memory function, and output function, for a given grammatical induction problem, provided some knowledge of the classes of solutions which are required is available. Furthermore, by relating the number of states in the automata which are to be represented to the number of nodes in various STCN designs, the issue of choosing a network size is simplified. This is especially useful to researchers using the Manual Architecture Changes (MAC) method for computing the change in state vector size (see Section 2.5.4) since it can be used to provide an educated guess as to the number of nodes an STCN should have.

The results presented in the table follow from numerous old and new proofs presented in this chapter. In particular, we proved a new result describing the types of languages implementable by a window in time memory. We proved that a single layer first-order context computation memory can implement arbitrary finite state automata and that they can do so using $n \cdot p$ nodes (where n is the number of states, and p is the number of input symbols). This chapter also presented a proof showing that single-layer second-order context computation memories are incapable of implementing arbitrary automata using binary state encodings, and two new proofs describing the limitations of LRSI memories.

In the past, most connectionist networks have been designed based on principles like ease of implementation or extension from existing work, rather than on the classes of languages that can actually be implemented and hence induced. By proving that architectures designed in this way can have unexpected limitations, this chapter suggests that a formal analysis of computational power may be a more sound method for making decisions about STCN designs. Similarly, since efficient grammatical induction requires

a search space which is as small as possible, it is also important to select an architecture which is not too powerful. In the chapters which follow, we shall continue to examine grammatical induction in a connectionist paradigm, but focus on more specific techniques of restricting and ordering the hypothesis spaces of these systems.

	Definite Machine	Finite Memory Machine	Finite State Automaton	Deterministic Pushdown Automaton†	Turing Machine
WIT memory	$n_i \cdot w_i$ 2-Layer	∞	∞	∞	∞
CIIR memory	$n_i \cdot w_i$ 2-Layer	$n_i \cdot w_i + n_o \cdot w_o$ 2-Layer	∞	∞	∞
SLFOCC memory	?	?	$n_s \cdot n_i$ 1-Layer	?	$12n_s+50$†† 0-Layer
SLSOCC memory	?	?	n_s 0-Layer	?	?
CPA memory	?	?	?	n_s 0-Layer	?
CTM memory	?	?	$n_s \cdot n_i$	?	$n_s \cdot n_i$ 0-Layer
LRSI memory	∞††† 0-Layer	∞††† 0-Layer	∞††† 0-Layer	∞††† 0-Layer	∞††† 0-Layer

n_s = number of states, n_i = number of input symbols, n_o = number of output symbols, w_i = input window width, w_o = output window width

† With no epsilon transitions and which always pops the current input symbol onto the stack during a pop operation.

†† Requires infinite precision units.

††† Only certain machines in this class can be represented with a finite number of state nodes (see Section 4.8).

Table 4-2: Number of state units and types of output functions required by STCNs to implement various automata.

Chapter V:
Fixing and Initializing Weights

5.1 INTRODUCTION

In the previous chapter we have seen that the choice of a particular STCN memory and output function can restrict the classes of computing machines which can be implemented by a STCN. We have also seen that fixing the number of state nodes restricts the size of the machines which can be realized. The types of restrictions imposed in these manners are very general restrictions in the sense that they limit the overall size or classes (in Chomsky hierarchy) of automata that can be represented. Sometimes, however, more specific information about the problem domain is available.

This chapter examines how specific knowledge about the classes of grammars to be induced can be incorporated into connection weights in an STCN either by choosing initial weights or by hard-wiring (fixing weights and not allowing them to be trained) prior

to training. This type of knowledge is called a priori knowledge. While numerous authors have provided empirical evidence showing that chosen initial and hard-wired weights improve the performance of their networks, this chapter focuses on explicating their results by proving how chosen and hard-wired weights both limit and order the hypothesis space of inducible grammars.

This chapter is organized as follows: Section 5.2 defines the notions of fixing and choosing initial connection weights and their effects. Section 5.3 examines how fixing weights in a STCN limits the space of grammars which a STCN can represent. Section 5.4 describes how choosing a network's initial weights affects the order in which the hypothesis space is explored. Section 5.5 explains how the chosen initial weights can also restrict the hypothesis space of STCNs. Section 5.6 addresses the issue of how specific *a priori* knowledge can be mapped into the weights of connections in a STCN and gives some example of the types of a priori knowledge other researchers have assumed might be available. Section 5.7 summarizes the results of this chapter.

5.2 FIXING AND INITIALIZING CONNECTION WEIGHTS TO RESTRICT AND ORDER HYPOTHESIS SPACES

The most natural way to incorporate specific *a priori* information about a problem to be solved by a STCN is to encode it in the weights of the network, since it is the weights that define the computation a network performs. We distinguish between two fundamentally different approaches: fixing weights and choosing initial weights. Fixing weights refers to keeping the weights of some of the connections in a network at a constant pre-specified value throughout the training process. Choosing initial weights refers to setting the weight values of some or all of the connections in a network to a pre-specified value but allowing these weights to be adjusted during training.

Note that choosing weights can be applied to all of a network's weights, whereas fixing can only be applied to some weights. This is due to the fact that if all of the weights in a network are fixed, the network is hard-wired and no learning can occur. While hard-wiring networks may be an interesting problem, it does not constitute a grammatical induction task, and hence lies beyond the scope of this thesis. Another approach to incorporating *a priori* knowledge in networks is to selectively prune some of the connections of the network before training. In the discussions which follow, we consider pruning to be a special case of fixing weights since any weights which are fixed at zero can effectively be considered pruned.

Since we are studying grammatical induction from the viewpoint of a search through an hypothesis space, we examine the effects which fixing and choosing initial weights can have on the size of the hypothesis space, and on the order in which it is explored. Thus we have two "causes" (fixing weights and choosing initial weights) and two possible "effects" (hypothesis space reduction and hypothesis space ordering). This results in four possible combinations. Of these, only fixing weights and hypothesis space ordering cannot be causally related, i.e. fixing the weights in a STCN does not affect the order of the exploration of the (reduced) hypothesis space. In the following three sections, we discuss each of the other three cause and effect combinations.

5.3 FIXING WEIGHTS TO LIMIT HYPOTHESIS SPACE

Perhaps, the most natural way to reduce the hypothesis space is to fix some of the weights in the STCN. Clearly, choosing weight values for some of the connections in a STCN and not allowing these connections to be trained, restricts the degrees of freedom available to the learning algorithm. This restricts the formal computing machines which can be represented, and hence the grammars which can be induced.

Fixing weights, however, cannot guarantee that the supplied a priori knowledge will actually be incorporated in the grammar induced by a STCN. This is due to the fact the trained weights in the network can overpower or nullify the contributions of the fixed weights. Suppose that the trained weights of the connections leading into node i in some STCN are much larger than the fixed weights leading into the same node. Since the signal transmitted though each connection are multiplied by the connection's weight and then summed together by node i, the effect of the fixed weights will be negligible compared to the effects of the larger trained weights. In this situation, the trained weights overpower the fixed weights.

Now suppose that all the connections leading out of node j are trained and have a very small weight after the training process. In this case any fixed weights leading into node j will affect the activation value of the node, but this activation value will be ignored by the rest of the STCN due to the small outgoing weights. In this sense, the trained weights nullify the effect of the fixed weights.

Of course, a network which ignores a priori knowledge in either of these two ways will further limit its representational capacity. That is, a STCN with n nodes will be able to represent a large class of automata. An STCN of the same size which has some fixed weights, and uses those fixed weights to compute its behaviour will be able to represent a smaller class of automata. Finally, a STCN with n nodes which has some fixed weights, but does not use these weights in its computation (either because they are overpowered, or because they are nullified) will be able to represent the smallest class of automata.

Frasconi, Gori, Maggini, and Soda (1991, in press) have explored fixing network weights based on *a priori* knowledge about an isolated word recognition task to be solved. Specifically, they develop a network consisting of two separate SLFOCC memories and a 1-Layer output function. One of the SLFOCC memories (called "K") consists entirely of fixed weights whose values are assigned based on the available knowledge. Details

concerning the types of available knowledge and how this knowledge is encoded in weights will be discussed in Section 5.6.1. The other SLFOCC memory (called "L") has adaptive weights whose values are learned based on training data. By using this modular approach, these authors are able to prevent the trained weights from overpowering the fixed weights. However, the output layer can still ignore the values of the state nodes in "K", by setting all weights originating from the K-memory to small values. We defer the discussion of the type of *a priori* knowledge used by the authors and how this knowledge is encoded into connection weights to Section 5.6 which is devoted to the encoding issue.

Frasconi, Gori, Maggini, and Soda's (1991,1994) networks are able to achieve a recognition rate as high as 92.3% in empirical performance tests. The authors indicate that this is a significant achievement due to the fact that the task of isolated word recognition is complicated by the fact that the words used are composed only of vowel and nasal sounds. They further argue that their approach is more efficient that ones which do not use *a priori* knowledge. Unfortunately, the authors do not provide any empirical data comparing networks with *a priori* data to networks without *a priori* data.

5.4 CHOOSING INITIAL WEIGHTS TO ORDER HYPOTHESIS SPACE

Just as fixing weights is the most intuitive way of limiting the hypothesis space of a STCN, choosing initial weights is the most natural way to order it. Obviously the initial weights define the first potential grammar which is explored by the induction algorithm. The exploration of subsequent grammars is governed by the learning algorithm. When the learning rate used by the gradient descent algorithm is small, each grammar considered will lie close to the previous candidate grammar in the STCNs weight space. Since the output and state of a STCN are governed by functions continuous in the connection weights of the network, a small change in a connection weight will tend to result in a small

change in output and state. This means that the exploration of the hypothesis space will proceed via similar grammars.

One advantage of using an ordering technique as opposed to a hypothesis space restriction technique is that there is often some uncertainty associated with a priori knowledge about a task. This implies that an irreversible decision, like eliminating certain grammars from consideration, is less desirable than an approach which can eventually ignore incorrect information. Setting the initial weights of a network can operate in this fashion since even if the first weights are wrong, and the network updates weights in small steps, the network will still be able to eventually explore other regions of the hypothesis space. This conclusion has been empirically verified by Giles and Omlin (1993a). They initialized the weights of STCNs to implement one automaton, A, and then trained the STCNs to represent another (different automaton), B. Despite the fact that this imposed an ordering on the hypothesis space which caused the network to explore automaton A first, the network was still able to eventually find and learn automaton B.

Specifically, Giles and Omlin (1993a) trained a STCN to implement a randomly generated 10-state finite state automaton. Then, they initialized the weights of the automaton to encode a different randomly generated 10-state automaton. The authors discovered that, so long as the assigned weight values assigned were not too large (>2), the networks were able to learn the correct automaton in spite of the "malicious" information provided by weight initialization. Of course, the authors also found that learning times were significantly longer for "malicious" information than for correct information. Giles and Omlin's results indicate that even if a priori knowledge is incorrect, an ordering scheme such as initializing weights can sometimes still find the correct solution.

5.5 CHOOSING INITIAL WEIGHTS TO RESTRICT HYPOTHESIS SPACE

While it is obvious that the initial weights used in a STCN order the hypothesis space, it is less apparent that initial weights can also restrict the space. To recognize the latter fact, we must realize that the search of the hypothesis space in STCNs is governed by a gradient descent algorithm. This implies that each candidate grammar considered during the search must have a smaller error value than the previous. But, since the initial weights of a network define a grammar, and since that grammar is assigned an error value, it must be the case that all grammars with higher error values than the initial grammar are omitted from the search. Thus, the initialization of weights can serve to both order the hypothesis space (as discussed in the previous section) and also to restrict the hypothesis space by causing all grammars with higher error values to be rejected outright. Figure 5-1 illustrates an initial set of weights (i.e. a point in weight space), a fictional error function, and those grammars which are not explored during the search algorithm.

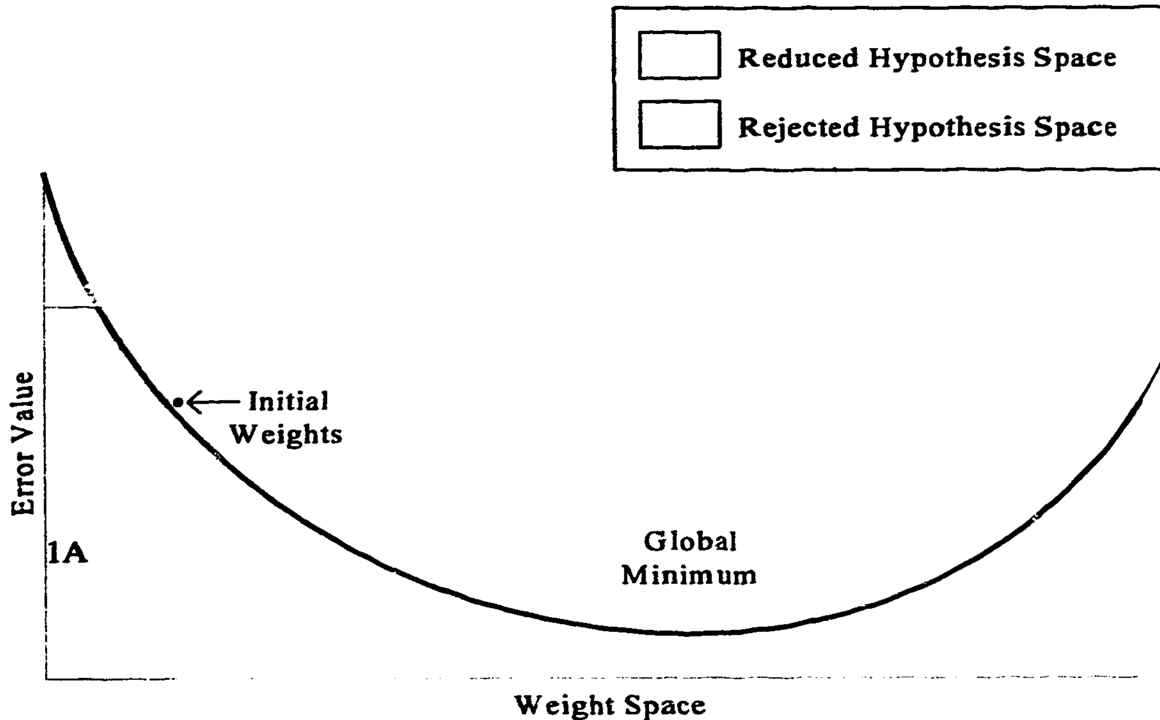

Figure 5-1: Initializing weights to limit space.

It is interesting to note that "good" a priori knowledge will tend to significantly reduce the hypothesis space, while "bad" knowledge tends not to reduce the hypothesis space as much. This is due to the fact that good a priori knowledge will tend to result in a STCN having a small error value. Since only those STCNs and grammars with even smaller error values are explored, the hypothesis space will tend to be greatly reduced. Conversely, bad a priori knowledge will tend to result in a STCN with a large error value. In this case there will be many STCNs and grammars having smaller error values and hence the hypothesis space will tend to remain large. This is an extremely useful property since it implies that good information will tend to have a large (positive) effect while bad information will tend to have very little effect.

There is, however, one serious drawback to choosing initial weights to restrict the hypothesis space: local minima in the error function. If the function mapping weight

values to STCN error is non-monotonic, then it may be the case that to get to a smaller error value one must first travel though a region (in weight space) of larger error. Since the gradient descent algorithm travels only down the error gradient, such smaller error values can never be achieved. That is, the initial weights do not limit the hypothesis space to all STCNs with smaller error values, but rather to those STCNs lying within the current basin of attraction. If the attractor at the bottom of this basin represents a local minimum (as opposed to a global minimum), then the hypothesis space will be unduly restricted to exclude the best solutions. This is illustrated in Figure 5-2.

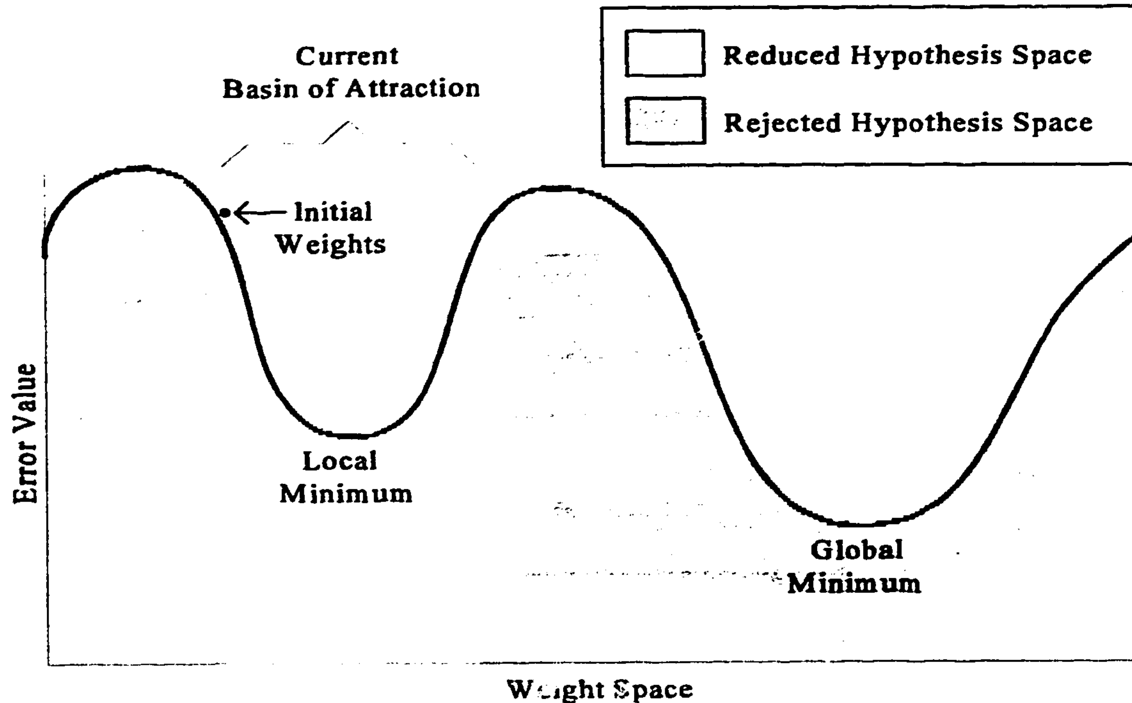

Figure 5-2: How initial weights can reduce the hypothesis space to exclude optimal solutions.

5.6 ENCODING A PRIORI KNOWLEDGE (WIRING NETWORKS)

Thus far, we have examined only the effects of initializing or fixing weights in a STCN. But in order to initialize and fix weights in a useful manner, it will be necessary to translate a priori knowledge into connection weights. Clearly, the encoding of knowledge will depend greatly on the particular architecture used by a STCN as well as the types of knowledge available. In particular, the nature of the memory function used by a STCN determines the interpretation which can be given to a weight and hence the types of a priori knowledge which can be encoded by fixing weights.

5.6.1 Single Layer First-Order Context Computation Memories

In a SLFOCC memory[1], the weight $W[i][j]$ reflects the correlation between node i (which is either an input, or context node) and node j (which is a state node). If $i<\|\vec{x}\|$, then i is an input node. In this situation, a large positive weight value indicates that input i tends "push" the STCN into a state in which state node j has a high activation value. Similarly, if the weight value is near zero, this indicates that input i does not significantly influence whether or not the next state is represented by a vector whose j^{th} component is high. Finally, if the weight value is large and negative then this implies that input i tends to "push" the STCN into a state in which state node j has a low activation value.

If node i is a state node, then a large positive value for weight $W[i][j]$ indicates that input states for which node i assumes a high value tend to have transitions to states for which node j assumes a high activation value. At the same time, a weight value near zero indicates that states for which node i assumes a high value may or may not have transitions to nodes where the activation of node j is high, and a large negative weight value indicate that states for which node i assumes a high value tend to have transitions to states for which the activation of node j is low.

Of course, these are only general trends which can be violated by other large weight values with contrary effects. As with most connectionist networks, the activation value of any node depends on the interaction of the activation values of all nodes connected to it. This implies that many such systems do not implement hard rules, but rather simultaneously consider numerous soft constraints.

---

[1]We begin with the SLFOCC memory and not the WIT or CIIR memory because the latter two memories do not make use of the weight matrix in their computation of state. Hence, fixing weight values has no effect on the state function for WIT and CIIR memories.

The disadvantage of interaction between encoded and learned weights is that it may be difficult to pre-wire connections, since the encoded knowledge may never actually be used by the final network. On the other hand, it also allows for uncertain *a priori* information to be encoded in a safe manner. If the magnitude of a pre-wired connection is small, then it will be relatively easy to override its effect (i.e. a small amount of training will suffice), while if it is large, then it will be difficult or even impossible (i.e. large amounts of training will be required). This implies that it will be useful to set connection weight magnitudes in proportion to the certainty of the knowledge which they encode. Of course, the degree to which it is possible to evaluate the certainty of knowledge depends on the application to which a grammatical induction system is applied. That is, rules which are very certain will be encoded with high connection weights (making them hard to override), while rules which are not as certain will be encoded with low connection weights (making them easy to override).

The interaction between wired and trained weights can also be avoided by modularizing a network. Frasconi et al. (1991, 1994) have used modularity to incorporate specific a priori knowledge into a STCN. They employ two separate SLFOCC memories: one hard wired (called 'K'), and the other adaptive (called 'L'). All available input signal are supplied to both networks and both compute their own states independently. The two states are then passed through an output function which unifies them and computes a common output signal. Using this approach, Frasconi et al. were able to show that a priori knowledge made learning more efficient for a language recognition task.

Specifically, Frasconi et al. (1991,1994) used a STCN to perform isolated word recognition. The input to their network consisted of phoneme vectors. Phoneme vectors encode the likelihood that a set of phonemes is detected by an acoustic preprocessing system. Each component of each vector indicates the certainty that one particular phoneme has been detected. When supplied with a recording of individual spoken words

from a dictionary of Italian, the preprocessor computes the likelihood of each phoneme being present at a sampling rate of 8192Hz, and encodes these values as a sequence of phoneme vectors. The phonemes detected were: /a/, /e/, /i/, /o/, /u/, /m/, and /n/.

Frasconi et al. used a separate STCN for each word that could be detected. It was each network's task to induce a mapping from a string of phonemes to the actual word which was spoken. In each case, the weights in the pre-wired component of the network 'K', were initialized to encoded a specific automaton, using a linear programming algorithm. In each case the encoded automaton used was a chain graph representation of the phonemes in the word. A chain graph is a graph with a tree structure but with an additional loop from each vertex to itself. The non-recurrent arcs in this graph are labelled with the phonemes of the word which is to be represented. Figure 5-3 illustrates the chain graph which the authors constructed for the Italian word /Numa/; the recurrent connections are not illustrated in this figure. The initial weights of the network 'K' were initialized to encode the automaton represented by this graph. Frasconi et al. note that the encoded automaton is capable of ignoring multiple occurrences of the same phoneme and of dealing with phoneme skips.

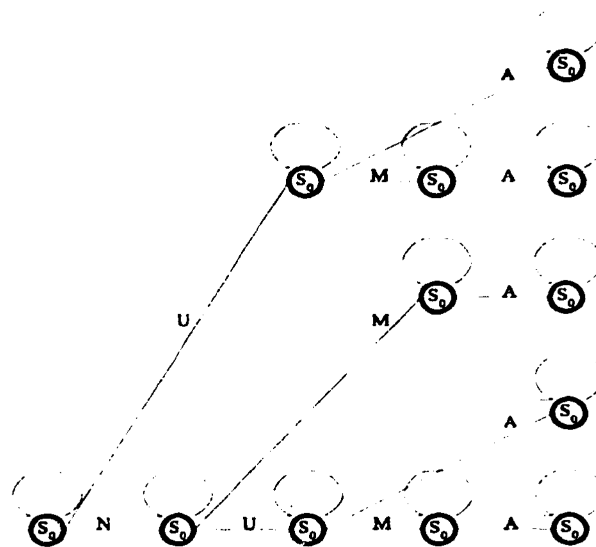

Figure 5-3: Automaton devoted to detect the Italian word /Numa/.

5.6.2 Single-Layer Second-Order Context Computation Memories

In SLSOCC memories, each weight connects three nodes. This allows for a different type of modularity. Suppose *W*[*i*][*j*][*k*] is the weight connecting input node *i* and context node *j* to state node *k*. If this weight value is large and positive then input signal *i* occurring during a state when the jth node's activation value is high will tend to "push" the network into a state where the kth node's activation value is also high. If we assume that states are encoded as unit normal vectors, then a large positive weight value will indicate a transition from state *j*, to state *k*, under input symbol *i*. Furthermore, under the unit normal vector assumption, all other weights must transmit a signal of zero.

This prevents the type of interference discussed for the SLFOCC memory. It is important to realize that this insulation is accomplished due to the fact that one single weight in the STCN uniquely defines one transition in the encoded automaton. In this sense, each weight can be considered a module. Omlin and Giles (1992, in press), Omlin, Giles and Miller (1992), Giles and Omlin (1992c, 1993a, 1993b), have used this form of

a priori knowledge encoding in SLSOCC memories and found that learning speed increased by up to one order of magnitude.

Specifically, Giles and Omlin (1993a) trained networks to induce 10-state automata. They encoded partial representations of the automata to be induced by initializing the second order connections as described above. The partial representations included: the target automaton with self-loops (from one state to itself) removed, the target automaton with 4 states (an the associated connections) pruned, the state transitions for a subset of the language implemented by the target automaton, several disjointed state transitions selected from the target automaton, the state transitions for a single legal string, the state transitions for a sequence of symbols that do not form a complete string, and the state transitions for two short strings. For each of these partial representations, the authors observed performance improvements ranging from reducing the learning time by one half to reducing the learning time by an order of magnitude (for more details, the reader is referred to Giles and Omlin, 1993a). This shows that initializing weights can result in significant performance improvements for grammatical induction tasks, even when relatively little prior knowledge (e.g. a single legal string) is available. Similar encodings can be performed in CPA memories and CTM memories in an analogous manner. Das et al. (1993) examined the former case using similar partial knowledge and also found increased learning speeds.

5.6.3 Locally Recurrent State and Input Memories

Fixing weights in LRSI memories has never been studied. Since LRSI uses first order connections, results are expected to be similar to those for SLFOCC memories. An interesting possibility would be to combine two heterogenous networks one hard wired, and one adaptive (as described by Frasconi, Gori, Maggini & Soda, 1991). If the hard wired network is capable of representing automata with cycles, then the other component

could be a LRSI memory. This would result in a LRSI memory hybrid which would not be forced to suffer the inherent limitations of LRSI memories described in the previous chapter.

5.6.4 Other Types of A Priori Knowledge

A final issue which should be considered in this discussion of fixing connection weights is the type of a priori information available to decide on appropriate values for the fixed weights. The method used to hard wire the second order connections in a SLSOCC memory relies on the assumption that a priori knowledge is in (or can be easily converted into knowledge in) the form of labelled state transitions. This may not always be the case. To see that this is so, we consider a psychology experiment proposed by Reber (1967). This author required human subjects to learn a simple grammar. He then created a new grammar identical to the grammar learned by the subjects with a different, but corresponding, set of terminal symbols. Reber found that subjects were able to learn the second grammar more quickly than the original.

One way of interpreting this result is that there exists some a priori knowledge which is applied to make learning the second grammar faster. This a priori knowledge is gained during the learning of the original grammar. If one wanted to model the second component of Reber's experiment, it would be necessary to encode information about the transitions of the target automaton in a STCN. However, unlike the encodings discussed earlier which assume that labelled transitions are available, in this situation only the fact that a transition exists and not the conditions under which such a such a transition would occur can be encoded in the STCN. In a SLSOCC memory it would be impossible to encode this type of knowledge since each individual weight represents a labelled state transition, and it is impossible to represent an unlabelled transition. This suggests that for

certain types of knowledge (i.e. unlabelled transitions) a SLSOCC memory's granularity may be too large.

We have indicated here that first order connection weights reflect general trends regarding the relationship between states and/or inputs. At the same time, second order connection weights indicate specific transitions. In some cases the types of a priori knowledge may not be specific enough to allow for its encoding in second order weights. If the a priori knowledge is "fuzzy" it might be more amenable to a first-order encoding. For example, if a probabilistic grammar which approximates the deterministic target grammar is known, knowledge might take the form of fuzzy grammatical rules such as "state A tends to be the predecessor to state B especially when input symbol s_l is presented". Such a rule could readily be translated to large positive weights on the first-order connections between the context node representing state A and the state node representing state B and between the input node representing symbol s_l and the state node representing state B.

5.7 CONCLUSIONS

In this chapter we have focused on how specific *a priori* knowledge about a grammatical induction task can be used to order and reduce the hypothesis space of inducible grammars. Specifically, we have stated that fixing some of the weights in a STCN limits the hypothesis space. We also argued that initial weights can both order and restrict the hypothesis space. An additional proof described how good *a priori* knowledge encoded in initial weights tends to limit the hypothesis space much more than bad *a priori* knowledge. While previous work in this area was empirical in nature, the results presented here give theoretical reasons for previous observations. While this chapter focused on prior knowledge about the grammar induction problem, in the next chapter we examine the use of information that becomes available during the induction process.

Chapter VI:
Using a Posteriori Knowledge to Find Target Grammars

6.1 INTRODUCTION

In the previous chapters, we examined how we could use prior knowledge about the grammars to be induced to accelerate learning or make it tractable. In this chapter we examine how a different type of knowledge can be used to achieve the same purpose. The three techniques explored here all provide additional information to the learner during the learning process rather than before induction begins. These techniques represent an alternative to the performance improvements which can be gained according to the previous chapters.

This chapter is divided into five Sections. Following this introduction, Section 6.2 describes how ordering the presentation of grammatical (and optionally non-grammatical) strings can be used to convey additional knowledge to the induction system. It examines

the use of input ordering by connectionists in STCNs. The ordering techniques used are proven to be equivalent to ordering techniques used in a symbolic paradigm. Next we present a theoretical explanation of how input ordering affects learning and accounts for the empirical results observed in STCNs. Section 6.3 describes using specific information about the states of the automaton to be induced, provided to the learner during induction, to guide the exploration of the hypothesis space. Section 6.4 discusses how the frequency with which strings are presented can be used to induce a new type of grammar: a stochastic grammar, and how stochastic grammar induction can overcome the limitations suggested by Gold. Specifically, we present a new proof that shows that a popular training scheme used for STCNs involves training these networks to identify the most probably correct stochastic grammar for a given sample. The training scheme is also compared to existing stochastic grammar induction algorithms in a symbolic paradigm, and the advantages of the connectionist approach are discussed. Finally, Section 6.5 summarizes the main results of the chapter and presents some conclusions.

6.2 INPUT ORDERING

One a posteriori t[illegible] for making the grammar induction problem tractable and more efficient is input ordering. In the traditional grammar induction paradigm, the learner is required to identify a grammar based on a set of positive (and optionally a set of negative) example strings. Under input ordering, the data available to the learner consists not of a set of strings, but of a sequence of strings. That is, there is an order associated with the input data. If input strings are presented in a non-random order, then the position of a string within the sequence can represent an additional source of information about the grammar to be induced.

For an input ordering to be advantageous, two criteria must be met: (1) The presentation of a string s at time t must encode some information other than the mere fact

that the string is a member (or not a member) of the language. (2) The learning system must be informed of the import of this encoded information and use it to limit or order the exploration of the remaining search space. Only when both of these criteria are met can a computational advantage be realized.

In this section, we examine the possibilities of input ordering in training STCNs for grammatical induction. Specifically, Subsection 6.2.1 presents some previous cases of input ordering in a symbolic paradigm where learning was improved. These serve as examples of how input ordering works and what it can achieve. Subsection 6.2.2 explains how the training regimens, used by various STCN researchers, encode additional information in the form of string ordering. This satisfies the first criterion for a performance-enhancing input ordering scheme. Subsection 6.2.3 explains why STCNs are sensitive to the order in which their input is presented and how of their training regimens exploit that sensitivity to improve performance. This satisfies the second criterion for a performance enhancing input ordering scheme. Subsection 6.2.4 concludes by noting that STCN training regimens constitute input ordering, and are thus a form of a posteriori performance enhancement.

6.2.1 Previous Results

We begin our discussion of input ordering by examining how input ordering works, and what it can achieve. Gold (1967) proposed two types of input string orderings which can improve the classes of grammars that can be induced using only positive input strings (text learning). The first of Gold's ordering schemes uses indirect negative information to learn languages which cannot be learned from positive information alone. This is done by using the absence of a string at a particular point in a sequence to infer that the string is illegal.

Suppose an order on all possible strings (grammatical and ungrammatical) is known to the learning system and this order defines how the environment provides the input data (e.g. alphabetical order). Note that there is an important distinction between knowing the order in which a sequence of strings are presented and the actual sequence of strings. An ordering defines a relation between all possible strings for a given alphabet, i.e. Σ^*, and thus defines where each string should belong (if it were legal), whereas the input sequence generally consists of only a subset of Σ^* and defines the actual set of legal strings.

Now assume that the induction environment presents all the grammatical strings to the learner according to the given order. Then, by omitting a string at the appropriate time, the environment essentially informs the learning mechanism that the given string is ungrammatical. Since this implies that the training set effectively contains both grammatical and ungrammatical strings, it is equivalent to informant learning (as described in Section 3.2.5). Since Gold has already shown that primitive recursive languages are identifiable in the limit under informant learning, this class of languages must also be learnable under ordered text learning. While this strict sense of ordering is obviously an unrealistic idealization for practical grammar induction systems, Gold's work does point out the power that an ordering scheme can provide. Later in this section, we will explore a less stringent, and thus more realistic ordering technique.

A second ordering technique proposed by Gold (1967) involves an even more elaborate transfer of information from the environment to the learner. Gold proved that if each sentence, s_t, in the sequence of input strings, s_0, s_1, s_2, ... , is computed according to a primitive recursive function on the sentence's position in the sequence, t, then it is possible to learn any Turing machine which maps integers into strings chosen from any arbitrary formal language L. Note that since only positive strings are presented, this is a version of text learning which can learn **any** Turing machine. It is very important to note, however, that the assumption that the ordering is computed according to a primitive

recursive function, and that the learner knows this function is a very strong one. It is so strong, in fact, that Gold concludes that this model of language learning "is of no practical interest" (Gold, 1967, p.452). Nonetheless, it does show that "restrictions on the order of presentation of elements of the text can greatly increase the learnability power of this method of information presentation" (Gold, 1967, p.453).

A third and less stringent ordering scheme has been proposed by Feldman (1972). He showed that even an effective approximate ordering of the input strings could be used to convey indirect negative information. If there exists a point in time by which every grammatical sentence of a given length or less has appeared in the sample, then a learner capable of computing this point in time can also compute which sentences are not in the language (this could be the case in human language learning if children were spoken to in short sentences). Once again this is equivalent to a learner which is provided with both grammatical and non-grammatical strings, appropriately labelled.

The common thread to all three of these techniques is the fact that the learner reacts differently to the same set of strings presented in differing orders. More specifically, strings which are presented early cause the learner to make certain assumptions about remaining strings which affects the order in which potential grammars are considered, or the size of the hypothesis space which is explored. More efficient and tractable learning can be accomplished by tailoring the learning algorithm and the input sequence to each other.

While all three of these techniques are designed to allow context-free and arbitrary grammars to be learnable in the limit given text input, input ordering can also be used to improve the *performance* of a grammatical induction system. If the learner can infer that certain strings are legal, or illegal, based on the order in which other strings are presented, then less of the hypothesis space must be explored in order to identify a grammar. Thus,

input ordering can be applied in both text and informant learning to improve the efficiency of learning.

6.2.2 Input Ordering in STCN Training

Having seen some examples of input ordering in a symbolic paradigm, and having examined the advantages therein, we now turn our attention to connectionist grammar induction systems. Recall that every input ordering system requires two components: an environment that orders input strings, and a learner which uses that order. We begin by studying the first of these two components. More specifically, we focus on training regimens for STCNs and examine whether or not additional information (beyond the grammaticality of individual strings) is conveyed in their associated input orderings. We identify four forms of input ordering used by STCN researchers: Lengthening Input Data, Alphabetical Input Data, Multi-Phase Uniform Complete Input Data, and Multi-Phase Non-Uniform Complete Input Data.

Lengthening Input Data

A simple ordering scheme which can be placed on strings is to sort them in order of increasing length. Das et al. (1993) have used a STCN training scheme whereby short simple strings are presented first, and progressively longer strings are presented as learning proceeds. They contend that "incremental learning is very useful when (a) the data presented contains structure, and (b) the strings learned earlier embody simpler versions of the task being learned" (Das et al., 1993, p.69), a well known concept in machine learning theory. In this situation, the fact that short strings are presented early, together with the fact that these strings embody simple versions of later strings, implies that it is possible to use the strings which have already been presented to make certain implicit logical inferences about strings which have not yet been presented. A grammar

induction system can be designed to use these inferences to dynamically reduce or re-order the space grammars it can induce.

For example, when a string of length n is presented as input to Das et al.'s system, it is possible to conclude that all strings which are shorter than n and have not yet been presented must be ungrammatical. This implies that these shorter strings will not be presented at a later point in time. In this sense, additional information about future strings (i.e. that they will not contain certain short strings) is transmitted by the ordered data. We will see later how a STCN learning system could function in this fashion.

Alphabetical Input Data

Giles et al. (1991, 1992a, 1992b) and Miller and Giles (1993) have used another simple ordering scheme: alphabetical ordering. If input strings are presented in strict lexicographic order, then the presentation of a string, s, implies that all lexicographically preceding strings which have not been presented must be ungrammatical, in the case of text learning, and must be of irrelevant (don't care) grammaticallity, in the case of information learning. In this sense, an alphabetical presentation order can convey additional information (regarding the grammaticality of unpresented strings). Once again, a learning system which is tuned to this type of ordering in the sense that it restricts or orders the space of inducible grammars dynamically could perform better than a system in which input is not ordered. (Empirical data describing Giles et al.'s and Miller and Giles' results will be provided in Section 6.2.3.)

Multi-Phase Uniform Complete Input Data

Both lengthening and alphabetical input orderings are very restrictive in the sense that they precisely prescribe the order of presented strings. For practical applications, it is often more desirable to use a less stringent ordering. Consider a case where input

strings are presented in phases. In the first phase, all short strings, and only short strings, are presented. In later phases other strings are presented. We shall refer to this type of partial ordering as *multi-phase uniform complete*[1] since the strings presented in the first phase are uniformly short and completely represented. A multi-phase uniform complete input ordering can provide additional information in the same sense that a lengthening input ordering does, with the exception that assumptions about strings which have not been presented can only be made at the end of a phase, as opposed to after each string. Giles et al. (1990) have used a multi-phase uniform complete ordering to train STCNs.

A similar ordering technique has been used by Elman (1991a, 1991b) who used a form of ordering to train his networks. While Elman considers his technique to be different than ordering (he describes the techniques used by Gold—which include ordering—and then states, "in this paper, I want to pursue what may be a third factor" (Elman, 1991b, p.4)), we shall prove that it is actually a form of multi-phase uniform complete ordering. Each phase consists of training a network on five passes through a specific corpus. In Elman (1991a), the corpora are described as follows:

> Phase 1: The first training set consisted exclusively of simple sentences. This was accomplished by eliminating all relative clauses. The result was a corpus of 34,605 words forming 10,000 sentences (each sentence included the terminal '.').
>
> Phase 2: The network was then exposed to a second corpus of 10,000 sentences which consisted of 25% complex sentences and 75% simple sentences (complex sentences were obtained by permitting relative

[1] The term *uniform complete* is based on Trakhtenbrot and Barzdin (1973) and Angluin (1976) who discussed the complexity of learning subsets of languages containing all strings not exceeding a given length and only those strings.

clauses). Mean sentence length was 3.92 (minimum: 3 word, maximum: 13 words).

Phase 3: The third corpus increased the percentage of complex sentences to 50%, with mean sentence length of 4.38 (minimum: 3 words, maximum: 13 words).

Phase 4: The fourth consisted of 10,000 sentences, 75% complex, 25% simple. Mean sentence length was 6.02 (minimum: 3 words, maximum: 16 words). (Elman, 1991a, p. 204)

In Elman (1991b), a fifth phase with 100% complex sentences is added.

We now prove that this training scheme corresponds to multi-phase uniform complete ordering because there are only a finite number of simple sentences and it is highly likely that all simple sentences are presented during the first phase of training. In fact, there are only 440 simple sentences in Elman's language. This is figure is computed as follows: Table 6-1 depicts the grammar for simple sentences. It is based on the grammar presented in Elman (1991a, p. 202), however, all relative clauses have been eliminated to allow only simple sentences. Further, the two "Additional restrictions" given in the original table have been incorporated directly into the grammatical rules. This involved distinguishing between singular and plural cases for noun phrases, verb phrases, nouns and verbs (NP1 vs. NP2, VP1 vs. VP2, N1 vs. N2, VI1 vs. VI2, VT1 vs. VT2) and distinguishing between intransitive and transitive verbs (VI1 vs. VT1, VI2 vs. VT2). Note that these changes do not alter the set of sentences (i.e. the language) defined by the grammar. They merely allow the representation of the language by BNF grammar rules alone, as opposed to a representation using both BNF grammar rules and "Additional restrictions". This will greatly simplify the computation of the number of simple sentences.

Rule	Number of Possible Combinations
S → NP1 VP1 "." | NP2 VP2 "."	$6\times44 + 4\times44 = 440$
VP1 → VI1 | VT1 NP1 | VT1 NP2	$4 + 4\times6 + 4\times4 = 44$
VP2 → VI2 | VT2 NP1 | VT2 NP2	$4 + 4\times6 + 4\times4 = 44$
NP1 → PropN | N1	$2 + 4 = 6$
NP2 → N2	$4 = 4$
N1 → *boy* | *girl* | *cat* | *dog*	4
N2 → *boys* | *girls* | *cats* | *dogs*	4
PropN → *John* | *Mary*	2
VI1 → *sees* | *hears* | *walks* | *lives*	4
VI2 → *see* | *hear* | *walk* | *live*	4
VT1 → *chases* | *feeds* | *sees* | *hears*	4
VT2 → *chase* | *feed* | *see* | *hear*	4

Table 6-1: Grammar for simple sentences (adapted from Elman, 1991a, Table 1, p.202).

The computation of the number of simple sentences goes as follows: There are a total of four singular nouns (N1), four plural nouns (N2), two proper nouns (PropN), four intransitive singular verbs (VI1), four intransitive plural verbs (VI2), four transitive singular verbs (VT1), and four transitive plural verbs (VT2), as indicated in Table 6-1. This implies that there are six singular noun phrases (NP1) and four plural noun phrases (NP2). For singular/plural verb phrases (VP1/VP2), the singular/plural intransitive verbs (VI1/VI2) are added to each combination of a singular/plural transitive verb (VT1/VT2) with a noun (singular or plural) phrase (NP1/NP2), giving a total of 44 singular/plural verb phrases (VP1/VP2). Sentences are formed by pairing singular/plural noun phrases (NP1/NP2) with singular/plural verb phrases giving a total of 440 sentences (S).

If we assume that all sentences are generated independently with equal probability (Elman does specify which probability distribution he uses), then the probability, P(*s*), of

generating any one simple sentence, s, is 1/440. We now compute the expected number of samplings required to see all 440 sentences, $E(n)$. We break the computation into steps, letting $E(n_i)$ represent the number of samples required before a new (unseen) sentence is found, given that i sentences have already been seen. If $i=0$, then the first string sampled is guaranteed to be a new string, so the expected number of samples required is exactly one:

$$E(n_0) = 1.$$

If $i=1$, then there is a 439/440 chance that the first string sampled is a new string. The chance that the second string sampled is the first new string is equal to the chance that the first string sampled is not new (1/440), multiplied by the chance that the second string sampled is new (439/440). The chance that the third (etc.) string sampled is the first new string can be computed in a similar fashion. Therefore, the expected number of samples required before a new string is discovered, given that one of the 440 strings has already been seen, is:

$$E(n_1) = \frac{439}{440}(1) + \frac{1}{440}\frac{439}{440}(2) + \frac{1}{440}\frac{1}{440}\frac{439}{440}(3) + \cdots$$

$$= \sum_{j=1}^{\infty} \frac{439}{440^j}(j)$$

If $i=2$, then the chance of not sampling a new string the first time is 438/440. The chance that the second string sampled is the first new string is equal to the chance that the first string sampled is not new (2/440), multiplied by the chance that the second string sampled is new (438/440). The chance that the third (etc.) string sampled is the first new string can be computed in a similar fashion. Therefore, the expected number of samples required before a new string is discovered, given that two of the 440 strings have already been seen, is:

$$E(n_2) = \frac{438}{440}(1) + \frac{2}{440}\frac{438}{440}(2) + \frac{2}{440}\frac{2}{440}\frac{438}{440}(3) + \cdots$$

$$= \sum_{j=1}^{\infty} \frac{2^{(j-1)}}{440^j} 438(j)$$

It is now possible to generalize this pattern for all values of i, where i is the number of already-seen strings. Specifically,

$$E(n_i) = \sum_{j=1}^{\infty} \frac{i^{(j-1)}}{440^j} (440-i)(j).$$

Rearranging gives:

$$E(n_i) = \frac{(440-i)}{i} \sum_{j=1}^{\infty} \left(\frac{i}{440} \right)^j (j).$$

Since the summation is now an arithmetic-geometric series of the form:

$$\sum_{n=0}^{\infty} nr^n = \frac{r}{(1-r)^2},$$

with $r=i/440$. The equation can be rewritten as:

$$E(n_i) = \frac{(440-i)}{i} \left(\frac{i}{440} \right) \left(\frac{440}{440-i} \right)^2,$$

which is equivalent to:

$$E(n_i) = \frac{440}{440-i}.$$

Recall that $E(n_i)$ represents the expected number of samplings required to discover a new string, given that i distinct strings have already been discovered. To discover all 440 strings, an expected

$$E(n) = \sum_{i=0}^{439} E(n_i)$$

samplings must be made. Substituting for $E(n_i)$ reveals that

$$E(n) = \sum_{i=0}^{439} \frac{440}{440-i}.$$

Solving this equation gives:

$$E(n) \approx 2933,$$

which implies that it will take an average of 2933 samples before all 440 simple strings are discovered by random sampling. Recall that Elman (1991b), uses 10000 simple string samples in his first phase of training.

Another interesting question to ask about Elman's multi-phase training regimen is: What is the probability,

$$P\left(\overline{s_1 \wedge s_2 \wedge s_3 \wedge \cdots \wedge s_n} \mid samplings = 10000\right)$$

that all 440 simple strings have **not** been presented after the first phase? The probability that one or more of the 440 strings will **not** be generated in 10000 samples is one minus the probability that **all** 440 strings will be sampled:

$$P\left(\overline{s_1 \wedge s_2 \wedge s_3 \wedge \cdots \wedge s_n} \mid samplings = 10000\right) = 1 - P\left(s_1 \wedge s_2 \wedge s_3 \wedge \cdots \wedge s_n \mid samplings = 10000\right)$$

The latter probability, in turn, is equal to the probability that **one** of the 440 stings is generated taken to the power of 440:

$$P\left(s_1 \wedge s_2 \wedge s_3 \wedge \cdots \wedge s_n \mid samplings = 10000\right) = P\left(s_i \mid samplings = 10000\right)^{440}.$$

The probability of generating one string in 10000 samples is equal to one minus the probability of not generating the string raised to the 10000th power:

$$P\left(s_i \mid samplings = 10000\right) = 1 - P\left(\overline{s_i}\right)^{10000}.$$

Combining all of these equations:

$$P\left(\overline{s_1 \wedge s_2 \wedge s_3 \wedge \cdots \wedge s_n} \mid samplings = 10000\right) = 1 - (1 - (439/440)^{10000})^{440}.$$

Using the binomial approximation:

$$P\left(\overline{s_1 \wedge s_2 \wedge s_3 \wedge \cdots \wedge s_n} \mid samplings = 10000\right) < 1 - (1 - 440(439/440)^{10000}) \approx 5.8 \times 10^{-8}.$$

This clearly indicates that it is extremely unlikely that any of the 440 simple strings have not been presented by the end of the first phase of training. Since it is very likely that no new simple strings are presented after the first phase of learning, the data presented are complete in addition to being uniform.

In addition to Giles et al. (1990) and Elman (1991a, 1991b), Stolcke (1990c) also uses a multi-phase training scheme but does not provide sufficient details to determine whether or not the phases are uniform and complete. If strings are presented according to multi-phase uniform complete input scheme, a learner that can infer that certain strings are illegal after the first training phase would be able to gain a computational advantage over a learner which made no such assumptions.

Multi-Phase Non-Uniform Complete Input Data

It is interesting to ask what would occur if the sample size of Elman's first training phase had been so small that not all of the simple strings could be included in the first phase. Could input order still make learning easier? In this situation, it would take a second (or third, etc.) phase before all simple strings were presented. The data presented in the two (three, etc.) phases would be complete, because all simple strings are presented, but non-uniform, because some complex strings are presented in the second (and higher) phase. Complete non-uniform ordering has also been used by Sun et al. (1990b) who uses a multi-phase training technique in which all short strings, and some mid-length strings are presented during the first phase of training. Both of these presentation techniques convey additional information because there exists a point in time at which all strings of a certain type have been presented (i.e. all short strings, or all simple strings). This is just like Feldman's ordering, and once again, a learner tuned to this ordering can use this additional information to make the learning process tractable or more efficient.

6.2.3 Input Order Sensitivity in STCNs

We have now seen that the training environments used for STCNs sometimes contain additional implicit information (beyond the grammaticality of individual strings) in the form of string ordering. This represents one of the two components required for

a more efficient learning system. The second component is a learner which uses the additional information. In this section we examine how input ordering affects the solutions explored and found by STCNs, thereby addressing this second component. Specifically, we examine two types of order sensitivity: engineered sensitivity and natural sensitivity.

Engineered Sensitivity

One way of ensuring that a learning system makes use of input ordering is to specifically design an induction algorithm around an ordering scheme. Since every symbolic algorithm "is equivalent to and can be 'simulated' by some neural net" (Minsky, 1967, p.55) it is not at all surprising that it is possible to realize such a hand crafted algorithm in the form of a connectionist network. As an example, Porat and Feldman (1991) have designed a connectionist network which implements an algorithm which induces FSA based on alphabetically-presented input strings. In order to implement the algorithm; however, the connectionist network requires a extremely complex control structure (compared to typical connectionist networks) and has both hardwired and mutable connections. Thus, the resulting learning system seems more like a connectionist-iterative learning hybrid than a purely connectionist architecture.

Natural Sensitivity

An alternative to designing the learning system to accommodate a particular input order is to design the input order to accommodate the learning system. This is typically done in STCNs, where network design is based on principles like simplicity, homogeneity, local processing, etc.. Having designed the network according to these principles, the researcher can only ensure cooperation between input order and learner by adjusting the input order to suit the network's own natural sensitivities to this order. In a sense, the researcher has assumed part of the burden of learning the language. It turns out that the

order of pattern presentation affects STCN (and spatial-network) learning greatly. This is due to the fact that initial weight changes in a network can draw solutions toward a certain local minimum, from which the STCN cannot later escape. This phenomenon is due the fact that STCNs do not perform true gradient descent.

Recall that, in order to efficiently approximate the gradient, weight adjustments, $\Delta W(t)$, are made piecewise over time. This implies that the component of the gradient caused by a pattern presented at time t is computed after the weight adjustment caused by the pattern at time t-1 has been made. This, in turn, means that successive weight adjustments are not commutative. To better understand the implications of this fact, we consider a simple example. Suppose we have a language consisting of only two training strings. Suppose also that the network error, $\left\{\frac{1}{2}\left\|\vec{E}(t)\right\|^2\right\}$, for each of these two strings is given in Figure 6-1 parts a) and b), and the total error for both strings, ε, is given in part c). Now suppose that the network's initial weights and corresponding errors are represented by the point labelled "B" in all three graphs. Clearly if the network is first trained only on the string whose error function is depicted in (a), then the network's weights will move to the point labelled "C". Subsequent training with the second string will keep the network's weights at "C" since it represents a local minimum in the second string's error space. By contrast, if the network is first trained using only the second string, (b), then the network will converge to point "A". Again, subsequent training will not change the weights in the network. Thus, it is clear that the order of string presentation during training limits the hypothesis space (range of weights) which is considered in searching for an error minimizing solution during later string presentations.

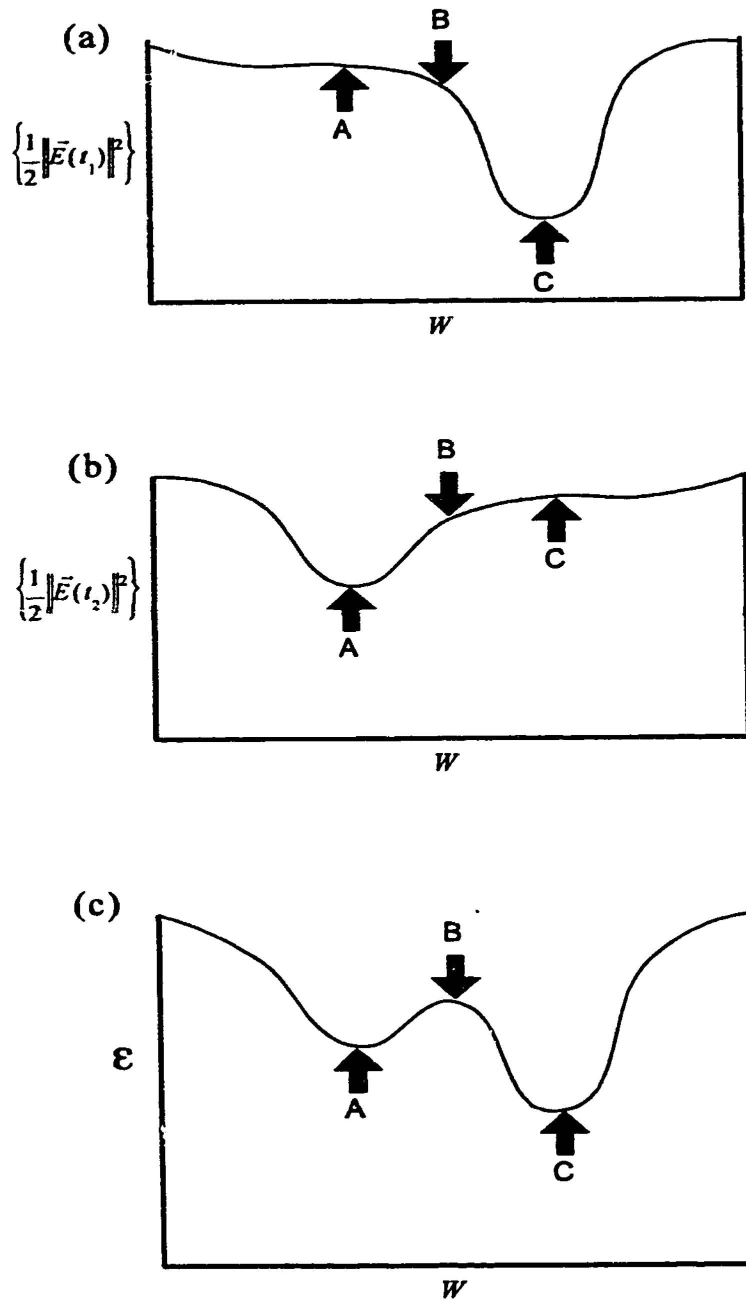

Figure 6-1: Error spaces for an STCN learning a two-string language. (a) Error spac for string 1. (b) Error space for string 2. (c) Combined (total) error space.

While it is easy to see that input ordering affects hypothesis space search during learning, it is much more difficult to identify the ideal ordering scheme for an STCN. In the example above, presenting the string (a) before string (b) will restrict the range of weights to include the global minimum of the error space. By contrast, presenting string (b) prior to string (a) also restricts the range of weights, but the restricted range does not include the global minimum of the error space. Thus, in the simple example, it is important to present string (a) first.

In more general terms, it is always best to present strings whose error functions have local minima at the same points in weights space as the total error function has global minima. Since the presentation of a string adjusts the weights of the network so that the string's error is reduced, the network's weights will approach a local minimum in the presented string's error function. Ideally, this local minimum in the string's error function will correspond (or lie close to) a global minimum in the total error. Since the specific strings satisfying this condition depend entirely on the language to be learned by the STCN, we cannot identify a general ideal ordering scheme. Instead we turn to the empirical evidence to show that the ordering schemes described above do in fact correspond to the natural sensitivities to input order in STCNs.

Das et al. (1993) compared training STCNs with a lengthening input ordering to training the same STCNs with a random ordering of strings. They observed a 50% reduction in training time for the lengthening ordering scheme (see Das et al., 1993, Table 1, p. 68). Giles et al. (1992b) also observed an improvement in training times when they presented strings in alphabetical order, and concluded that, "the sequence of strings presented during training is very important and certainly gives a bias in learning" (Giles, 1992b, p.320) and that "training with alphabetical order ... is much faster and converges more often than random order presentation" (Giles, 1992b, p.320).

While the performance improvements realized by the ordering schemes of Das et al. (1993) and Giles et al. (1992b) took the form of accelerated learning, Elman (1991a, 1991b) used ordering to learn an otherwise unlearnable grammar. In two learning experiments, Elman's multi-phase consistent complete ordering approach was used after previous attempts to train the network on the entire data set (complex and simple sentences) failed. In both cases, Elman found that, "when the network was permitted to focus on the simpler data first, it was able to learn the task quickly and them move on successfully to more complex patterns" (Elman 1991a, p. 204). This evidence clearly shows that input ordering can be used in the connectionist domain (just as it has in the symbolic paradigm) to improve learning efficiency and tractability.

6.2.4 Conclusions

In this section, we have reviewed how input ordering can be used to accelerate learning as well as make it tractable. To do this, sample strings must be presented in an order that conveys useful information to the learning system in the sense that the system restricts or orders its hypothesis space based on the presentation order. It must also be the case that the learner is tuned to the presentation order in the sense that it can extract the information from it. While some authors have designed a learner to accommodate a particular input ordering, grammatical induction by STCNs typically involves tailoring the input scheme to suit the learner. Four input ordering schemes, lengthening, alphabetic, multi-phase uniform complete, and multi-phase non-uniform complete, have been used. Empirical evidence suggest that all of these ordering schemes exploit the natural sensitivities to input order exhibited by gradient descent learners and result in improvements in learning efficiency and tractability.

6.3 AUTOMATON INFORMATION

Another approach to using additional information during the search for a target grammar involves providing the learner, at certain points in time, with very specific, but incomplete, knowledge about the automaton to be induced. This section explores STCN approaches to providing automaton information. Subsection 6.3.1 explains how automaton information can been used to learn an otherwise unlearnable grammars. Then, Subsection 6.3.2 describes how knowledge of dead states in the automaton can be used to accelerate and simplify the problem of grammar induction in STCNs, while Subsection 6.3.3 describes an approach to learning context-free grammars which requires knowledge of when symbols are pushed and popped onto the stack of a pushdown automaton implementing the grammar. Finally, Subsection 6.3.3 summarizes this section.

6.3.1 Using Automaton Information to Make Learning Tractable

It has been known since Gold (1967) that only finite cardinality languages can be learned from text input. Yet, there are arguments that children do not receive direct negative examples (or at least pay no head to them) and arguments (see Chomsky, 1956) that natural languages are, at least, context-sensitive (if not arbitrary). Additionally, most children are able to learn natural languages. This has led researchers to search for explanations to reconcile this apparent paradox.

One of the most popular explanations is that only part of any given language is learnt and the remainder of the language is available to the learner from other sources. Two proposals have been offered. First, Chomsky proposed that there is an innate pre-wiring to learn language. This is known as the "modularity" position. Alternatively, if one rejects the modularists' position, then one believes that the context—including semantic referents, goals, etc.—provides additional information about a language's structure.

For our purposes, we do not take a position on this debate. Instead, we break both proposed mechanisms into two components: the first provides additional knowledge (from innate pre-wiring in the case of a modularist position, or context in the non-modularist case), and the second, performs grammatical induction based upon both the presented strings and the additional knowledge supplied by the former component. If we are interested only in the "learning" component of a language learner, then only the content, and not the source of the available knowledge about a language is important. Therefore, we study only the second component, while making certain assumption about the information content supplied by the first component.

Specifically, it is useful to assume that the knowledge about a language provided takes the form of an underlying grammar whose terminal symbols are rewritten by a learned grammar to represent the desired language. The underlying grammar is called a "universal base", because it is assumed that whatever language the learner eventually learns is predicated on this base. The existence of a universal base grammar is called the "universal base hypothesis". If one accepts the universal base hypothesis, then language learning can be reduced to learning a rewrite grammar which maps the universal base to the desired language. Note that we have not made any assumptions about the source of the information contained in the universal base—it could come from pre-wiring; or it could come from context as has been suggested by Wexler and Culicover (1980).

Typically, there are relatively strong restrictions placed on the rewrite grammars which can be induced. These restrictions can make the rewrite grammar easier to learn than the combined grammar for the language, defined by applying rules from the rewrite grammar to symbols generated by the base. For example, if the universal base is a context sensitive (or unrestricted) grammar, and the rewrite grammar is a regular grammar, then the combined grammar is context sensitive (or unrestricted). Yet, since only the rewrite grammar component is learned, and since regular grammars can be induced from positive

and negative example strings (as shown by Gold, 1967), the combined system will be capable of inducing context sensitive (or unrestricted grammars). This is significant, because Gold also showed that grammatical induction of context sensitive (unrestricted) grammars cannot be induced based only on positive and negative example strings.

In the context of the induction mechanism, whose job it is to induce an automaton which implements a desired grammar, the universal base grammar translates to a universal base automaton (e.g. a specific Turing machine) and the rewrite grammar translates to a second (simpler) automaton which performs a mapping from the output of the universal base to the output of the grammar learner. It is the job of the induction system to develop a combined mechanism which can then implement the desired grammar.

For example, Wexler and Cullicover (1980) developed a model of human language learning based on a non-modularist approach. In their model, the context in which sample sentences are provided is represented by a context sensitive grammar. The terminal symbols of this base grammar (base phrase markers) are then supplied to the learning component of their system. This tells the learner about the state of the base automaton, it must then learn only the rewrite automaton. Wexler and Cullicover showed that the learner is capable of identifying, in the limit, an automaton combining of a pre-specified component and an induced component, which is capable of implementing certain classes of context free grammars. Since their theory is both complex (their entire book is devoted to its development and explanation) and specific to natural languages, it will not be reproduced here. Instead, we focus only on the highest level principle of their theory: that intractable language learning tasks can be made tractable if specific information about the operation of a part (i.e. the universal base) of the total machine is available to the learner.

6.3.2 When Strings Become Illegal

Providing base automaton information to the learner is also an approach which has been used in a connectionist paradigm. One technique involves informing the learner, not just *that* a string is illegal, but at what point the string became permanently illegal. The manner in which this is related to base automaton information will become clear shortly. To make things more clear, we consider a simple example. Suppose we wish to teach a STCN the language 1^n0^n (i.e. the language consisting of all strings having *n* 1s, followed by *n* 0s, for all *n*). This language can be described by the grammar: S→1S0. Rather than presenting the string "111101011001" labelled illegal, we could inform the learner that the string became permanently illegal upon presentation of the prefix "111101". This is equivalent to telling the learner that after presentation of "111101", there are no suffixes which can be applied to make the string "111101" conform to the grammatical rules of the language to be induced. This effectively conveys more information than just labelling the stings as illegal, because it identifies a whole class of illegal strings (i.e. all strings starting with "111101"). Of course, this assumes that there exists a *teacher* or an *oracle* in the training environment having the ability to identify precisely at what point the string becomes illegal.

It is important to realize that any automaton for which it is possible to identify strings with an illegal prefix string must have a "dead state" (a state which is non-accepting, and has connections only to itself). In the example above, presentation of the string "111101" would place the minimal automaton into the dead state since no further input symbols can ever make the string legal again. By presenting strings along with information regarding when the strings become illegal, the learning system is told when the automaton to be induced should be in the dead state. In a sense, than can be thought of as proving information about the state of a base automaton upon which an induced automaton is built.

Das et al. (1993) have used dead state information to train STCNs on grammatical induction tasks. This is done by defining a dead state output-node and training this node to assume a value of 1 if and only if the prefix string makes all strings starting with the prefix illegal. While the use of dead states appears to have little effect on learning for simple, context-free grammars, empirical results show that several more complex context-free grammars proved to be unlearnable without dead state information, yet readily learnable with this form of additional knowledge about the target grammar (automaton) (Das et al., 1993, Table 2, p.68).

6.3.3 Push/Pop Information

We now describe another approach to using information about the automaton to guide a grammatical induction system during learning. This approach is specific to context-free grammar induction. It involves providing the learner with knowledge of when the induced automaton should push and pop symbols onto and from the stack. In many grammars, pushing and popping symbols from the stack corresponds to what we would intuitively call the beginnings and ends of phrases respectively. More formally, these operations correspond to entering and returning from recursive procedures.

For example, sentence fragments like "the bump on the log in the hole at the bottom of the sea" are usually represented grammatically as recursive structures of the form illustrated in Figure 6-2. In this figure, NP represents a noun phrase, Art an article, N a noun, PP a prepositional phrase, and Prep a preposition. The structure depicted can also be represented by the following context free grammar:

$S \rightarrow$ NP

NP $\rightarrow$ Art N PP

PP $\rightarrow$ Prep PP

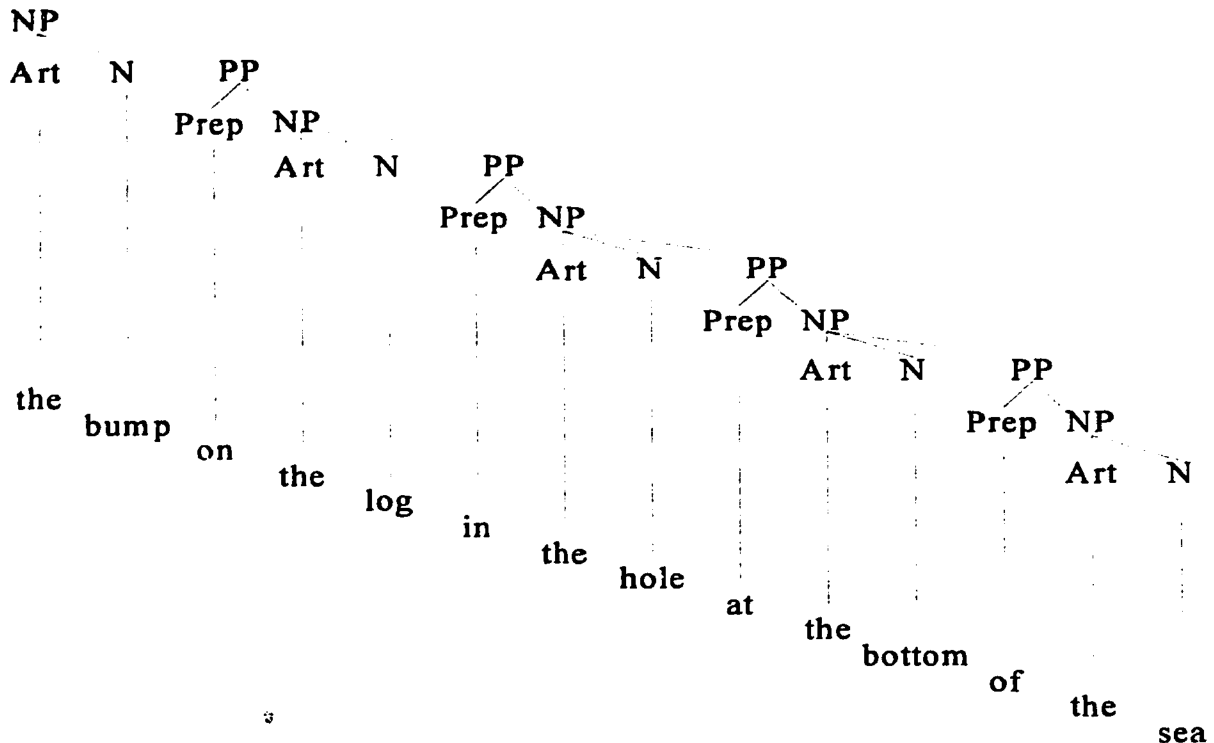

Figure 6-2: Recursive phrase structure.

When implemented in a pushdown automaton, such a grammar would result in one push operation every time a new PP is rewritten and one pop operation after the rewrite is completed. If this type of push and pop information is supplied to a grammatical induction system it will be able to identify when recursive phrases begin and end. In this sense, this type of information is analogous to the base phrase markers used by Wexler and Culicover (1980).

Das et al. (1993) have also used this approach to train STCNs on a difficult grammatical induction task: learning the palindrome language for two symbols. This language consists of all strings which consists of the same sequence of symbols when read either forwards or backwards. Das et al. note that the palindrome grammar represents an especially difficult language induction task because the automaton implementing it must precisely remember the first half of every string which is presented. Unfortunately, they

do not present any empirical comparisons of learning with, versus learning without push/pop information. It is clear, however, that this information represents important additional knowledge about the computational machine that is the goal of the induction process.

6.3.4 Conclusions

This section has shown that specific information about the automaton which is to be induced can be indirectly presented to a learning system to simplify the learning task. In effect, this involves presenting more information than the grammaticality of only one string. Two approaches were discussed: dead state information and push/pop (phrase structure) information. Empirical results confirm superior learning for both types of information.

6.4 FREQUENCY CODED STRINGS

In this section we prove that the frequency with which training strings are presented during grammatical induction can be used as an additional source of information to induce a new type of grammar: a stochastic grammar. Further we explain why this new learning scenario can offer a tractable solution to the problem of grammar induction described by Gold (1967). In Subsection 6.4.1 we give a detailed description of the new learning paradigm, including formal specifications of stochastic grammars, stochastic languages, and stochastic automata. Subsections 6.4.2 and 6.4.3 examine some of the complications involved with stochastic grammar induction by focusing on the representations of solutions and in what sense they might be considered optimal. Subsection 6.4.4 presents previous conclusions, made by Horning (1969), about the tractability of grammar induction in the context of the new paradigm. And in Subsection 6.4.5 we prove that a common STCN training technique is an alternate implementation of

stochastic grammar induction in the new learning paradigm. We are thereby able to conclude in Section 6.4.6 that STCNs are able to exploit the advantages of stochastic grammar induction to avoid the difficulties associated with deterministic grammar induction as suggested by Gold.

6.4.1 Stochastic Grammars, Languages, and Automata

In previous chapters, we have considered languages simply as sets of grammatical strings, and STCNs as systems which learn to render grammaticality judgements on strings. The grammaticality judgement procedure executed by STCNs is strictly deterministic. That is, for each sequence of input vectors, $\vec{x}(0)$, $\vec{x}(1)$, ..., $\vec{x}(t)$ there exists exactly one target vector $\vec{y}^*(t)$ representing the ideal network response. While this paradigm works well when networks are trained to render grammaticality judgements as in Gold's *acceptor* problem, it cannot be easily applied to Gold's *generator* problem.

Recall that in Gold's generator problem, the objective of learning is to be able to generate grammatical strings (and not just recognize them). Any system doing this must implicitly make a decision regarding the frequency with which respective strings are to be generated. Previously we defined a *(formal) language* as a set of strings of symbols from some one alphabet. In the context of string generation, this definition will no longer be suitable. In its place, we use the following definition:

> A *(formal) stochastic language* is a frequency distribution on strings of symbols from some one alphabet.

Henceforth, we shall used the word *language* to refer only to non-stochastic languages and the words *stochastic language* to refer to stochastic languages. As before, the symbol Σ

is used to represent the alphabet. *Stochastic Grammars* are viewed as descriptions of stochastic languages. Formally:

A *stochastic grammar* is a finite characterization of a stochastic language.

Once again, we shall distinguish between (non-stochastic) grammars and stochastic grammars. We use *Backus-Naur Form* (BNF) as a meta-grammar to describe stochastic grammars. While Hopcroft and Ullman (1979) do not discuss probabilistic automata, other authors (e.g. Salomaa, 1969; Huang and Fu, 1971) do. We choose, however, to develop a new formalism to describe these devices in order to be consistent with the previous notations (which were based on Hopcroft and Ullman).

Recall that a BNF grammar is a 4-tuple, $G=(V,T,P,S)$ where: V represents the set variables, T represents the set of terminals, P the set of productions, and S the start symbol. Unlike for (non-stochastic) grammars, where productions took the form $\alpha\rightarrow\beta$, for stochastic grammars productions are described with an additional parameter, $p_{\alpha i}$: $\alpha \overset{p_{\alpha i}}{\rightarrow} \beta_i$. Here, $p_{\alpha i}$ represents the probability of applying the rule i to sting α to generate a new string β_i. In this thesis, we assume that all stochastic grammars are normalized in the sense that:

$$\sum_i p_{\alpha i} = 1 , \ \forall i \text{ s.t. } p_{\alpha i} \in P.$$

This assumption implies that for any string, α, occurring on the left size of a production, the probability of applying a production is unity.

The rules of a stochastic grammar are applied randomly according to the given probabilities in order to generate a sample of strings from the language which the stochastic grammar describes. An example of a Stochastic grammar is given in Table 6-2. Note that this particular stochastic grammar is a *regular* stochastic grammar since all rewrite rules satisfy the requirements for regular grammars given in Section 3.4.1.

Rule	Probability	Rule	Probability
S→tA	50%	B→vD	50%
S→pB	50%	C→xB	50%
A→sA	50%	C→s	50%
A→xC	50%	D→pC	50%
B→tB	50%	D→v	50%

Table 6-2: Simple stochastic grammar.

While grammars are used to test whether or not a given string is a member of a particular language, stochastic grammars can be used to generate sample strings of a stochastic language. Since stochastic languages are frequency distributions, the strings generated by stochastic grammars are sampled from the given distribution. This is done as follows: First, the current string is initialized to be the start symbol. Second, the left-most non-terminal symbol in the current string is matched to all production rules having the same symbol on the left-hand side. Third, one of the production rules is selected according to the rule probabilities and applied. Steps two and three are repeated until only terminal symbols remain in the current string, at which point the current string represents a sample string.

String, β	Frequency, $\mu(\beta)$	String, β	Frequency, $\mu(\beta)$
pvv	1/8	pttvv	1/32
txs	1/8	ptvps	1/32
ptvv	1/16	tssxs	1/32
pvps	1/16	txxvv	1/32
tsxs	1/16		

Table 6-3: Frequently generated strings and their probabilities.

We have assumed here that all production rules have only a single non-termina symbol on the left-hand side. This applies only to regular and context-free grammars Generation of strings from stochastic context-sensitive and stochastic unrestrictec grammars is a complex and only partially resolved issue which has been studied b Sankoff (1978) and Labov (1969). We limit consideration to stochastic grammars that ar either regular or context free.

Formally, we define the stochastic rewrite operator, $\overset{p_{Ai}}{\Rightarrow}$, by asserting tha $uA\beta \Rightarrow u\alpha_i\beta$ with probability p_{Ai} if and only if the production $A \overset{p_{Ai}}{\rightarrow} \alpha_i$ is a member of P. Thi operator represents one application of steps 2 and 3 in the string sampling process Multiple applications can be represented by the stochastic reflexive-transitive rewrit operator, $\overset{p}{\Rightarrow}*$, which is defined as the reflexive and transitive closure of the stochasti rewrite operator. In the case that it is necessary to go though a sequence of direc derivations before generating β from α, for example,

$$\alpha \overset{p_1}{\rightarrow} \xi_1 \overset{p_2}{\rightarrow} \xi_2 \overset{p_3}{\rightarrow} \dots \overset{p_n}{\rightarrow} \xi_n \overset{p_{n+1}}{\rightarrow} \beta,$$

we say that β is derived from α with probability p,

$$\alpha \overset{p}{\Rightarrow}* \beta,$$

where

$\delta(q_0,T,q_1) = 0.5$	$\delta(q_2,V,q_4) = 0.5$
$\delta(q_0,P,q_2) = 0.5$	$\delta(q_3,X,q_2) = 0.5$
$\delta(q_1,S,q_1) = 0.5$	$\delta(q_3,S,q_5) = 0.5$
$\delta(q_1,X,q_3) = 0.5$	$\delta(q_4,P,q_3) = 0.5$
$\delta(q_2,T,q_2) = 0.5$	$\delta(q_4,V,q_5) = 0.5$

Table 6-4: Transition function for a stochastic finite state machine.

$$p = \prod_{i=1}^{n+1} p_i .$$

Finally, when α represents the start symbol, S, and β is a string consisting entirely of terminal symbols, then the probability, p, is the probability of generating string, β. To be more specific, we say that strings are generated according to the probability law μ, where $\mu(\beta)$ is the probability of terminal-string β being generated from the start symbol. I.e.:

$$\mu(\beta) = \prod_{i=1}^{n+1} p_i$$

when

$$S \overset{p_1}{\rightarrow} \xi_1 \overset{p_2}{\rightarrow} \xi_2 \overset{p_3}{\rightarrow} \ldots \overset{p_n}{\rightarrow} \xi_n \overset{p_{n+1}}{\rightarrow} \beta ,$$

Since all rewrite rules in the above example grammar all occur with probability 0.5 (this need not be the case), the frequency of occurrence of each legal string is defined by the number of steps in each string's derivation. The number of steps is equal to the strings length, l, so the frequency of occurrence of each string is governed by the formula: 2^{-l}. Some of the more frequently generated strings, β, and their respective probabilities, $\mu(\beta)$, are illustrated in Table 6-3.

We now turn our attention to stochastic automata, specifically we define stochastic generators to be automata which generate strings sampled from stochastic languages.

Using the notations for finite state automata and Moore machines defined in Sections 3.4.1, and 3.4.4 respectively, we describe a stochastic finite generator by a five-tuple $(Q,\Delta,\delta,q_0,q_T)$. Here, Q represents a finite set of states, $\Delta=\Sigma$ the output alphabet of the automaton and the alphabet of the language, and q_0 the start state. In non-stochastic automata, δ was a function mapping current state and input symbol to next state. In contrast, in stochastic finite generators, $\delta(q,a,q')$ is a function mapping the current state, q, the output symbol, a, and the next state, q', to the probability of entering state q' from state q while generating output symbol a. It is assumed that the function δ is normalized in the sense that:

$$\sum_{q'} \delta(q,a,q') = 1 ,$$

for all states, q, except the final state, q_T, for which:

$$\sum_{q'} \delta(q_T,a,q') = 0 .$$

The stochastic finite generator operates as follows: First, the current state is set to q_0. Then, the automaton's next state and output symbol are generated according to the current state, and the probability function, δ. This step is repeated until the final state q_T is reached. The sequence of output symbols generated then represents a generated string.

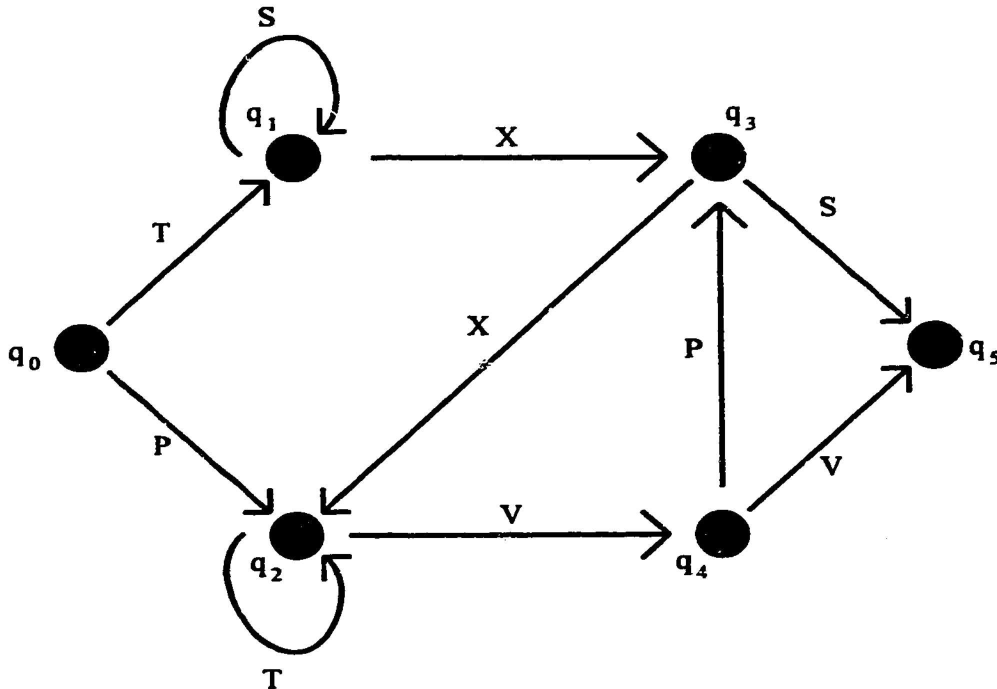

Figure 6-3: Stochastic finite generator automaton (adapted from Reber, 1967)

Our example grammar and language from above can be implemented by a 6-state stochastic finite generator with alphabet, $\Delta=\{P,S,T,V,X\}$, and transition probability function, δ, defined in Table 6-4. All values for δ not explicitly defined are assumed to equal zero. This stochastic finite generator is illustrated in Figure 6-3. An arrow from state q to state q' with the label a indicates that $\delta(q,a,q')=0.5$. That is, all transitions occur with 50% probability. Though not explicitly proven, it should be clear from this example how any stochastic regular grammar can be mapped into a stochastic finite generator.

Stochastic pushdown generators are defined similarly. Using the notations for pushdown automata and Moore machines defined in Sections 3.4.2, and 3.4.4 respectively, we describe a stochastic pushdown generator by a four-tuple $(Q, \Delta, \Gamma, \delta, q_0, q_T, Z_0)$. Here, Q represents a finite set of states, $\Delta=\Sigma$ the output alphabet of the automaton and

the alphabet of the language, and q_0 the start state. For stochastic pushdown generators, $\delta(q,a,\gamma,q',\gamma')$ is a function mapping the current state, q, the output symbol, a, the current stack symbol, γ, the next state, q', and the next stack symbol, γ', to the probability of entering state q' from state q when the stack symbol is γ while simultaneously replacing the stack symbol with γ', generating symbol a. Normalization implies that:

$$\sum_{q',\gamma'} \delta(q,a,\gamma,q',\gamma') = 1 ,$$

for all states, q, except the final state, q_T, for which:

$$\sum_{q',\gamma'} \delta(q_T,a,\gamma,q',\gamma') = 0 .$$

The stochastic pushdown generator operates as follows: First, the current state is set to q_0 and the stack contents to Z_0. Then, the automaton's next state, new stack contents, and output symbol are generated according to the current state, stack symbol, and the probability function, δ. This step is repeated until the final state q_T is reached. The sequence of output symbols generated then represents a generated string.

The stochastic automata presented in this section are but one example of formal computing models of probabilistic processes. There are many others. Perhaps the most note-worthy are hidden Markov models. While we have assumed that our devices do not accept input, but rather generate strings, hidden Markov models are capable of producing outputs in response to specific inputs. In this sense they are capable of representing mappings from one (input) language to another (output) language in a similar fashion to Moore machines. By contrast, the automata presented here "merely" represent the contents of one language. Clearly, the relationship between Hidden Markov Models and

the STCNs described in the following section represents an interesting and expansive area to be explored in future work.

6.4.2 Stochastic Grammar Induction by Non-Stochastic Automata

The problem of stochastic grammar induction can now be described as the process of using a finite sample of some stochastic language to develop a finite representation of the entire language. More specifically, we assume that the sample of the stochastic language is generated by some unknown stochastic automaton, and that the representation of the language also takes the form of an induced automaton. This induced automaton need not be stochastic, but it must be capable of representing the original language. In particular, we shall assume that the automaton implements a function mapping partial strings of symbols, denoted by the variable U, to the expected value of the next symbol to be generated by the unknown automaton. If we further assume that the symbols in the alphabet of the language, $b_1,b_2,b_3,\ldots,b_n \in \Delta$, are encoded as vectors, denoted by variable Y, in general, and the vectors $\vec{y}_1,\vec{y}_2,\vec{y}_3,\ldots,\vec{y}_n$, in particular, then the expected value for the next symbol, given the partial string U, would be denoted: $\vec{E}(\vec{Y}|U)$. The conditional expectation is computed by summing the vector encodings, $\vec{y}_j$, of the output symbols weighted by their respective a posteriori probabilities for the given string, $\mathrm{p}(b_j|U)$, over all output symbols , $b_1,b_2,b_3,\ldots,b_n \in \Delta$, according to the formula:

$$E(\vec{Y},U) = \sum_{j=1}^{n} \vec{y}_j \, \mathrm{p}(b_j|U).$$

If we assume a simple encoding scheme whereby each output symbol is encoded by a unit normal vector, then

$$\vec{y}_j = \hat{i}_j,$$

and

$$\vec{E}(\vec{Y},U) = \sum_{j=1}^{n} \hat{i}_j \, p(b_j | U).$$

In other words, the expected value of Y is equal to a vector whose components represent the probabilities of generating each of the possible output symbols. For the example used throughout this section, some sample (partial) input strings, U, and the probability distribution vectors which the induced automaton would be required to compute are given in Table 6-5.

We refer to an automaton which computes expected values for Y generated by some specific stochastic automaton as that stochastic automaton's *predictor*. It is important to notice that a predictor using a unit normal output vector encoding encodes just as much information as the original stochastic automaton, only in a deterministic form. We have now identified the goal of a stochastic grammar induction system: to find a predictor automaton. The next section explains why this goal is harder to achieve than might be expected.

6.4.3 Simplicity vs. Similarity

The degree of success of the induction process can be measured in terms of the similarity between the grammars represented by the source automaton and the induced automaton. Note that complete success cannot be guaranteed since there are always an infinite number of potential unknown automata, and the finite sample of sentences only provides a limited amount of information about the unknown automaton. Instead, we would like to develop an induction algorithm which identifies a language which has a maximum likelihood of being the original stochastic automaton. In this sense, we are searching for a "most probably" correct language.

Note that the approach presented here shares some common features with Valiant's (1984) probably approximately correct (*PAC*) learning paradigm. The goal of *PAC*

learning is to identify an object that lies reasonably close (according to some pre-specified metric) to the ideal solution to a given problem. To be considered a *PAC* learning algorithm, an induction procedure must achieve the given level of approximation with reasonable certainty. Much of *PAC* learning theory is devoted to proving that *PAC* learning algorithms can operate in polynomial (as opposed to exponential time).

While Valiant's goal was to identify an object that lies reasonably close to the ideal solution, the goal of probabilistic grammar induction is to identify the most probably correct grammar (in a Bayesian sense). Since STCNs are typically trained to reduce the mean squared error between output and target vectors to a given threshold, ε, STCN training can be viewed as a technique to find an approximately correct solution to a problem. We do not compute the probability with which an STCN finds a solution with a small error (i.e. the probability that an STCN convergence), nor do we examine the complexity of the weight update procedure. The likelihood of convergence and the complexity of weight adaption represent important open issues for connectionism and cannot be properly discussed within the scope of this dissertation. For an introduction to these issues, the interested reader is referred to McInerney, Haines, Biafore and Hecht-Nielsen (1989) and Judd (1991). If the probability of STCN convergence could be calculated for polynomial running times, then the algorithm presented here could be considered a *PAC* learning algorithm.

An Example

As an example, we randomly generate 100 strings based on the stochastic automaton of Figure 6-3 or equivalently the stochastic grammar of Table 6-2 to illustrate the difficulty of inducing a stochastic automaton based on a finite sample. Later, we shall show that the same conclusions reached here can apply to larger sample sizes. The frequencies of the strings which were generated are presented in Table 6-6. All strings

not illustrated in this table occurred with frequency zero. Additionally, the probability of generating each of these strings was calculated. When multiplied by the sample size (100), this probability represents the expected frequency of occurrence of each string; this frequency was rounded off to the nearest whole number.

Note that the observed frequencies and the expected frequencies are different; this difference is a form of noise. The occurrence of this noise is a property of random sampling from a binomial distribution and should come as no surprise. However, in the context of stochastic grammar induction, this noise complicates the induction task. This is because it is not possible to determine which components of the sample are due to noise and which are due to the source automaton.

In our example, we generated 52 strings beginning with the letter "*p*" and 48 strings beginning with "*t*". We know from the description of the source automaton that the probability of generating either a *p*, or a *t* as the first letter is exactly 50 per cent. A learner which has no a priori knowledge of the source automaton, however, must conclude, using Bayes' theorem, that it is most likely that the source automaton generates a *p* as the first symbol with probability 52 percent, and a *q* as the first symbol with probability of 48 percent. This is because, of all possible stochastic automata, the ones most likely to generate 52 strings beginning with *p* and 48 beginning with *t* are those with have probabilities of 52 and 48 percent. While this small discrepancy may not seem very significant in terms of the accuracy of the induced automaton and while it is also true that the accuracy will increase with increasing sample size, the difference remains a serious problem when it comes to minimizing the inducing automata. We illustrate this by continuing to explore our example case.

Partial String, U	Probability Distribution, $\vec{y}=$ (P(*p*), P(*s*), P(*t*), P(*u*), P(*v*), P(*x*))
ϵ	(0.5, 0.0, 0.5, 0.0, 0.0, 0.0)
p	(0.0, 0.0, 0.5, 0.0, 0.5, 0.0)
t	(0.0, 0.5, 0.0, 0.0, 0.0, 0.5)
pt	(0.0, 0.0, 0.5, 0.0, 0.5, 0.0)
pv	(0.5, 0.0, 0.0, 0.0, 0.5, 0.0)
ts	(0.0, 0.5, 0.0, 0.0, 0.0, 0.5)
tx	(0.0, 0.5, 0.0, 0.0, 0.0, 0.5)
ptt	(0.5, 0.0, 0.0, 0.0, 0.5, 0.0)
ptv	(0.5, 0.0, 0.0, 0.0, 0.5, 0.0)
pvp	(0.0, 0.5, 0.0, 0.0, 0.0, 0.5)
pvv	(0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
tss	(0.0, 0.5, 0.0, 0.0, 0.0, 0.5)
tsx	(0.0, 0.5, 0.0, 0.0, 0.0, 0.5)
txs	(0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
txx	(0.0, 0.0, 0.5, 0.0, 0.5, 0.0)
pttt	(0.0, 0.0, 0.5, 0.0, 0.5, 0.0)
pttv	(0.5, 0.0, 0.0, 0.0, 0.5, 0.0)
ptvp	(0.0, 0.5, 0.0, 0.0, 0.0, 0.5)
ptvv	(0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
pvps	(0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
pvpx	(0.0, 0.0, 0.5, 0.0, 0.5, 0.0)
tsss	(0.0, 0.5, 0.0, 0.0, 0.0, 0.5)
tssx	(0.0, 0.5, 0.0, 0.0, 0.0, 0.5)
tsxs	(0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
tsxx	(0.0, 0.0, 0.5, 0.0, 0.5, 0.0)
txxt	(0.0, 0.0, 0.5, 0.0, 0.5, 0.0)
txxv	(0.5, 0.0, 0.0, 0.0, 0.5, 0.0)

Table 6-5: Deterministic representation of stochastic grammar.

String	Sample Frequency	Expected Frequency
ptttttvpxvpxtttttvps	1	0
ptttvpxvps	2	0
pttvpxvpxttvv	1	0
pttvv	4	3
ptvps	3	3
ptvpxttvv	1	0
ptvv	7	6
pvps	7	6
pvpxttvps	1	0
pvpxtvps	1	0
pvpxtvv	1	1
pvpxvps	1	1
pvpxvpxvv	1	0
pvpxvv	1	2
pvv	20	13
tsssssssxxvpxvv	1	0
tsssssxxvv	1	0
tssssxs	1	1
tsssxs	1	2
tsssxxtvv	3	0
tsssxxvps	1	0
tsssxxvv	2	0
tssxs	1	3
tssxxttttttttttvpxtvv	1	0
tssxxttvpxvps	1	0
tssxxvps	1	0
tssxxvv	1	1
tsxs	4	6
tsxxtttttvpxtttvpxt-tttvv	1	0
tsxxttvv	2	0
tsxxvpxttvps	1	0
tsxxvpxvps	1	0
tsxxvv	1	2
txs	10	13
txxttvv	2	1
txxtvps	2	1
txxtvv	3	2
txxvps	2	2
txxvpxvps	1	0
txxvpxvv	1	0
txxvv	2	3

Table 6-6: Sample strings generated by probabilistic grammar/automaton.

Minimization of Stochastic Automata

We generated a tree whose arcs are labelled with the symbols from the strings in Table 6-6 and their relative frequencies. This tree represents a stochastic generator for the sample data and is illustrated in Figure 6-4. Note that the re-created generator is much more complex that the source of Figure 6-3. However, just as FSA can be minimized, this stochastic generator in Figure 6-4 can also be simplified.

The minimization algorithm for FSA (Hopcroft and Ullman, 1979) relies on collapsing equivalent states into one single state. Two states p and q are considered equivalent, $\equiv$, if and only if, for every input string x, the automaton starting in state p accepts the string if an only if the automaton starting in state q also accepts the string. If $\hat{\delta}(q,w)$ is the state of the FSA after reading the string w starting in state q, then equivalence can be mathematically expressed as:

$$p \equiv q \text{ iff } \forall x \in \Sigma^* \quad \hat{\delta}(p,x) \in F \text{ } iff \text{ } \hat{\delta}(q,x) \in F .$$

We now develop a new minimization algorithm for stochastic automata. Here, the situation is more complicated since two states, p and q, can only be considered equivalent if their string generation probabilities are equal. That is, if the stochastic automaton in state p can generate the string x with probability of $\hat{\delta}(p,x)$, then if state q is equivalent to state p, it too must generate the string x with probability $\hat{\delta}(q,x)$, exactly equal to $\hat{\delta}(p,x)$. Mathematically, this can be expressed as:

$$p \equiv q \text{ iff } \forall x \in \Delta^* \quad \hat{\delta}(p,x) = \hat{\delta}(q,x),$$

where $\hat{\delta}(q,w)$ is the probability of generating string w starting in state q.

A minimization algorithm can now be formulated as follows: First, since no strings can be generated from a terminal state, q_T, all leaf nodes, p, in the tree of Figure 6-4 must be equivalent and can be collapsed. Mathematically, it must be the case that:

$$\forall x \in \Delta^* \quad \hat{\delta}(p,x) = \hat{\delta}(q_T,x) = 0 .$$

Second, if any two nodes p and q have all links to the same nodes with the same probabilities, they are equivalent and can be collapsed. Mathematically,

$$p \equiv q \text{ iff } \forall a \in \Delta, \forall r \in Q, \delta(p,a,r) = \delta(q,a,r)$$

Step 2 can be repeated until no further minimizations can be achieved. The resulting automaton is minimal in terms of number of states.

Applying Minimization to the Example

This process was applied to the stochastic finite generator of Figure 6-4 to yield the somewhat simplified automaton of Figure 6-5. Note that the simplification is fairly insignificant in the sense that the automaton of Figure 6-5 is still far more complex than that of Figure 6-4. It is also critical to note that this difference is not attributable to the algorithm used to create the automaton of Figure 6-5, but rather to the random noise which enters the system during the generation of the sample data. In fact, the automaton of 6-5 represents the sample data perfectly and minimally. This data, however, does not completely specify the original automaton (language).

One might wonder if the fact that more states in the generator of Figure 6-5 cannot be collapsed is just an artifact of the small sample size (100 strings) used in the example. While it is true that more data would drive the transition probabilities of the arcs in 6-4 closer to the values of the corresponding arcs in 6-3, it would also allow for a greater resolution in probability scores. This would have the effect that while probabilities of corresponding arcs would approach each other, they would also be less likely to be exactly equal, which implies that their origin states could not be collapsed, and the stochastic automaton could not be minimized. Furthermore, if more sample data were available then new strings (not present in the original 100 string sample set) would have to be incorporated into the automaton. Thus, more sample data would result in an even larger minimized automaton.

Specifically, let us assume that q_i and q_j represent two different states in a non-minimized stochastic automaton like the one illustrated in Figure 6-4. Further, let us assume that both q_i and q_j correspond to a single state in the original source automaton. We would like to know the probability that the frequencies, f_i, and f_j of the transitions from q_i and q_j do not correspond, since this probability determines whether or not q_i and q_j will be collapsable into a common state using our minimization algorithm.

If the number of strings generated from q_i and q_j are different, then the difference in granularity of the possible frequencies than can be represented will preclude equal transition frequencies. Specifically, if N_i is the number of strings generated from q_i and if N_j is the number of strings generated from q_j, then the observed frequencies of symbols generated from q_i can achieve an accuracy of at most $1/2N_i$. Similarly, the observed frequencies of symbols generated from q_j will have an accuracy of at most $1/2N_j$. Thus the accuracy of the frequency difference between the transitions from state q_i and state q_j will be limited to $1/2N_i + 1/2N_j$ or:

$$\frac{N_i + N_j}{2N_i N_j}.$$

For this reason, the probability that states q_i and q_j can be collapsed is equal to:

$$P\left(|f_i - f_j| > \frac{N_i + N_j}{2N_i N_j} \right).$$

We now prove:

***Theorem 6-1*:** As N_i and N_j increase, the probability that the frequencies of the transitions from states q_i and q_j will correspond, and hence that q and q can be collapsed into one state decreases.

Proof: We begin with the equation for the probability that the two states can be collapsed and observe that since f_i-f_j is symmetrical about the origin:

$$\mathrm{P}\left(|f_1 - f_2| > \frac{N_1 + N_2}{2N_1N_2} \right) = 2\mathrm{P}\left(f_1 - f_2 > \frac{N_1 + N_2}{2N_1N_2} \right).$$

Rearranging the inequality gives:

$$\mathrm{P}\left(f_i - f_j > \frac{N_i + N_j}{2N_iN_j} \right) = \mathrm{P}\left(f_i - f_j - \frac{N_1 + N_2}{2N_iN_j} > 0 \right).$$

Since the values of f_i and f_j both assume binomial distributions, they can be approximated by normal distributions as sample size is increased. Further, since the difference between two normally distributed variables also assumes a normal distribution, the probability is equivalent to that of a z-test, $\mathrm{P}(z > z_0)$, where:

$$z_0 = (0 - \mu)/\sigma,$$

and μ and σ represent the mean and standard deviation, respectively, of the distribution of:

$$f_i - f_j - \frac{N_1 + N_2}{2N_iN_j}.$$

Since the mean values of f_i and f_j are both zero,

$$\mu = \frac{N_1 + N_2}{2N_iN_j}.$$

Furthermore, since f_i and f_j are both normally distributed and independent, their variances, σ_i^2 and σ_j^2, can be added to give the variance of:

$$f_i - f_j - \frac{N_1 + N_2}{2N_i N_j}.$$

Computing the square root of the combined variance gives the standard deviation:

$$\sigma = \sqrt{\sigma_i^2 + \sigma_j^2}.$$

The variance of f_i is given by:

$$\sigma_i^2 = 1/4N_i,$$

so:

$$\sigma = \frac{1}{2}\sqrt{\frac{N_1 + N_2}{N_1 N_2}}.$$

This implies that:

$$z_0 = \frac{N_i + N_j}{2N_i + N_j} \Bigg/ \frac{1}{2}\sqrt{\frac{N_i + N_j}{N_i N_j}} = \sqrt{\frac{N_i + N_j}{N_i N_j}}.$$

We now note that if N_i and N_j are both greater than 2, then the value of z_0 decreases as either N_i or N_j or both increase. Further, since $P(z > z_0)$ increases as z_0 decreases we can conclude that as N_i and N_j increase, the probability that the frequencies of the transitions from states q_i and q_j will correspond, and hence that q_i and q_j can be collapsed into one state, decreases. This proves Theorem 6-1. □

Figure 6-4: Recreated stochastic generator automaton.

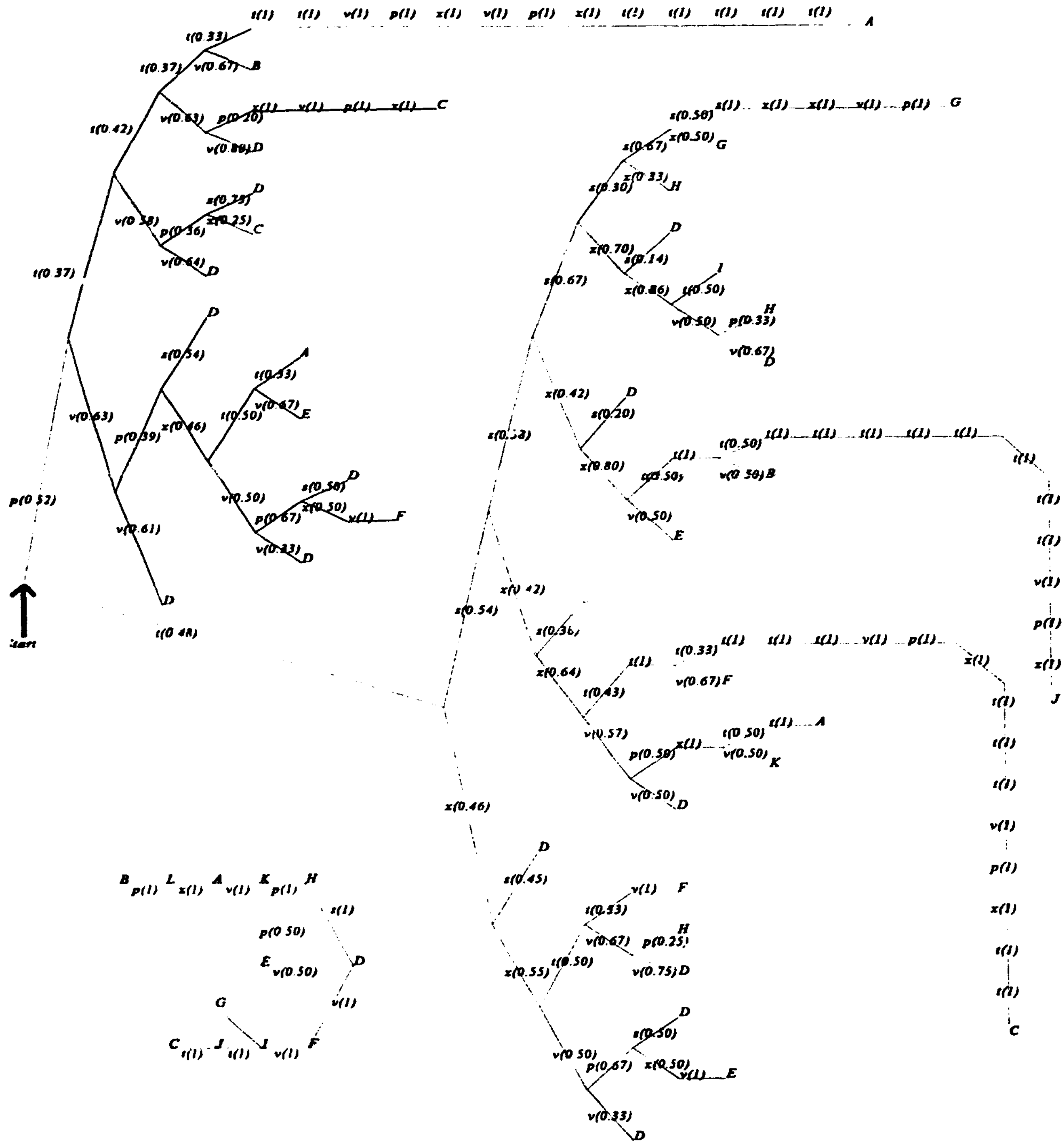

Figure 6-5: Minimized stochastic generator automaton.

The lesson to be learned from this exercise is that an exact representation of the sample data is unfeasible. Instead, a stochastic grammar induction algorithm must find a balance between accuracy with respect to the sample data and simplicity of the induced grammar (automaton); how this can be accomplished will be explained in the following sections. Note that the issue of exact representation differs from the problem of overtraining a network to the point at which it can no longer generalize for two reasons. First, this technique applies not just to STCNs, but rather all stochastic grammar induction systems. Second, the problem here does not concern misclassification of input patterns, but rather the complexity of the internal representations developed.

6.4.4 Previous Results Concerning Stochastic Grammar Induction

Having defined stochastic grammar induction and identified its ultimate goal (to find a simple stochastic grammar which has a high probability of matching the source), we now turn our attention to previous work in this paradigm. This will give us some insight into how stochastic grammar inducers operate and how well they perform. Horning (1969) examined the difficulties of learning stochastic grammars. He developed a learning algorithm based on Bayes's Theorem which states that the probability of a cause, c, given an observed effect, e, is equal to the probability of the cause multiplied by the probability of the effect given the cause, divided by the probability of the effect. Mathematically this is expressed as:

$$P(c|e) = \frac{P(c)P(e|c)}{P(e)}.$$

In the context of grammar induction, the cause can be interpreted as a stochastic grammar, while the effect is a sample of strings generated from the grammar. Under this interpretation, it is possible to compute the probability of a stochastic grammar, G, given

a set of sample strings, S. This is done by multiplying the *a priori* probability of the grammar, P(G), by the probability that the grammar would generate the sample, P(S | G), and dividing by the probability of the sample stings, P(S). Mathematically this is expressed as:

$$P(G|S) = \frac{P(G)P(S|G)}{P(S)}.$$

In this equation, the value of $P(S|G)$ can be thought of as a measure of how well the grammar, G, matches the sample. It is then natural to treat $P(G)$ as the simplicity of the grammar in order to give maximal values of $P(G|S)$ to those grammars which both match the sample and are relatively simple.

Horning assumes that various grammars have differing *a priori* probabilities ($P(G)$) according to their complexity. He then compares the grammars by computing values of $P(G)P(S|G)$ since the denominator in the equation for $P(G|S)$ remains constant for a given sample. The value of $P(S|G)$ is easily computable by multiplying the probabilities of generating the constituent strings of S, each of which, in turn, can be computed by multiplying the probabilities of the rules of G used to generate the given string (just as the probabilities for the strings in Table 6-6 were computed). Starting with the grammar with highest *a priori* probability, Horning computes the value $P(G)P(S|G)$ until the *a priori* probability for the grammar drops below the greatest value of $P(G)P(S|G)$ discovered so far, i.e. $P(G) < P(G^*)P(S|G^*)$. Since $P(S|G)$'s maximum value is 1 (for the case of a grammar which always generates the same string), a grammar whose *a priori* probability lies below the maximum value of $P(G)P(S|G)$ can never produce a higher $P(G)P(S|G)$ value. In this manner, Horning's algorithm can converge on the most probably correct grammar for any text.

Recall that according to Gold (1967) the task of converging on the correct grammar for a text which is drawn from a regular set is impossible. Thus, Horning's result is somewhat surprising, and a promising solution to the difficulty of grammatical induction identified by Gold. Unfortunately, Horning's algorithm has one important limitation: it is enumerative. The algorithm functions by examining individual stochastic grammars in a predetermined order. Enumerative induction algorithms work well for discrete search spaces of limited size. Yet, the search space of stochastic grammars is continuous since the rule probabilities are real values. Horning's algorithm can only explore an infinitesimal portion of this space.

In practise, the effectiveness of the algorithm in selecting the most probable grammar hinges on a good choice of *a priori* probability function. This is because the probability function defines both how the continuous search space is discretized and the order of the exploration of the space. It is important that the a priori probability function be well chosen because it defines a discrete set of points in the continuous space of stochastic grammars which will be searched by the algorithm. In this sense, the probability function defines the quality of any solution which can be found. If there are no points in Horning's discrete search space which lie close to the optimal grammar for a given problem, then only poor solutions will result. Of course, without exact knowledge of where appropriate solutions can lie no discrete search algorithm can function as well as one designed find a solution in a continuous space.

The second important effect of the a priori probability function is that it determines the order in which the space is explored and thus the volume of space which must be examined before a solution can be found. Specifically, suppose that G_1 and G_2 are two successive grammars in order of decreasing *a priori* probability, i.e.: $P(G_1) > P(G_2)$. Since Horning's algorithm only terminates when $P(G_1)P(S|G_1) > P(G_2)$, it is desirable that the inequality $P(G_1)P(S|G_1) > P(G_2)$ hold for as many successive grammars, G_1 and G_2, as

possible. In other words, under ideal conditions $P(S|G_1)$ should be greater than the ratio of probabilities, for successive values of G, i.e.:

$$P(S|G_1) > \frac{P(G_2)}{P(G_1)}.$$

In this situation, Horning's algorithm will converge on a most probable value of G after considering only a small number of other values of G.

Problems occur when $P(S|G)$ is less than the relative rate of change in $P(G)$. In this case, successive values of G will have $P(S|G_1)P(G_1) < P(G_2)$. This implies that the algorithm used to find the most probable grammar for a given sample will never converge. This worst case scenario will occur whenever $P(S|G)$ is small or whenever $P(G_1) \approx P(G_2)$ (in which case $P(G_2)/P(G_1)$ is maximized). When might these conditions be satisfied?

We first examine under what conditions $P(S|G)$ is likely to be small. Recall that $P(S|G)$ is computed by multiplying the a posteriori probabilities of the constituent strings, s, of the sample, S. This is expressed by the formula:

$$P(S|G) = \prod_{s \in S} P(s|G).$$

In this equation, $P(S|G)$ will be minimized as $\|S\|$ is increased and as the individual string probabilities, $P(s|G)$ are decreased. The former occurs for larger sample sets, while the latter will occur for grammars, G, which are more complex in the sense that many diverse sets of strings can be generated. We now consider the second condition under which Horning's algorithm will perform poorly (in the sense that it takes a long time to converge): i.e. when $P(G_1) \approx P(G_2)$. This scenario is realized whenever there are many, relatively equally probable a priori candidate grammars for the induction. In other words, when there is no existing bias to favour one grammar over another independent of example

strings. In summary Horning's algorithm cannot be expected to perform well when there is a large set of sample strings, when the candidate grammars are complex, or when there are many good candidates—in other words, when the problem is scaled up.

These theoretical predictions are supported by Horning's empirical results. Horning implemented his algorithm and tested in on several very small (no more than 3 non-terminal symbols) regular grammars. The algorithm successfully induce grammars with 2 non-terminals, but had to be halted before it was able to induce a 3 non-terminal grammar. This, no doubt, was in large part due to the hardware available in 1969. An implementation of his system on today's machines would surely be able to perform much more successfully. Nonetheless, even Horning believed that his algorithms were too slow to be of practical use, describing them as, "disappointing in terms of computational efficiency, and it is not claimed that in their present form they are economically justifiable for practical applications" (Horning, 1969, p.119). Horning also recognized that the inefficiency of his induction algorithm was due to the enumerative nature of his technique, stating that, "the enumerative problem is immense" (Horning, 1969, p.120) and, "Even when restricted to reduced grammars, however, the procedure was rather slow, due to the voluminous nature of the enumeration" (Horning, 1969, p.121).

It is vital to realize the difference between Horning's scaling and discretization problems, and the problem of grammatical induction identified by Gold (1967). Gold showed that, under text learning (positive training data only), it is impossible to ever converge upon one grammar which remains constant for all subsequent example strings. By contrast, Horning showed that it is possible to weaken Gold's learning criterion such that for any given sample of stochastically generated strings (positive training data only), and for a specific set of a priori grammar probabilities, a most probably correct grammar can be identified. Unfortunately, the algorithm used by Horning to select the ideal grammar for any weighted set of sample strings can only converge in reasonable time for

small problems (small sets of sample strings, simple candidate grammars, and few a priori good candidates) and is limited in the quality of solutions which can be found.

A different algorithm from the technique proposed by Horning could be used to successfully converge on most probable grammars for given sets of strings. Such an algorithm would ideally need to be able to explore any point in the continuous, stochastic-grammar hypothesis-space and not in a pre-specified enumerative order. In the next section we will discover that a popular STCN training technique implements precisely this type of algorithm.

6.4.5 Predictor STCNs are Most Probably Correct Stochastic Grammars

We now extend White's (1989) proof to show that a popular training methodology for STCNs minimizes the Euclidean difference between the output of the STCN and the output of the predictor for the stochastic grammar which generated the training data. In other words, by minimizing its error, the STCN learns the most probably correct stochastic grammar. Furthermore, the STCN training algorithm does not involve an enumerative process or a pre-specified discretization of the continuous hypothesis space.

While STCNs are sometimes trained to render grammatically judgements on input strings, they can also be used in a predictive manner. The alphabet of the stochastic language to be induced, Δ, is encoded as a set of unit vectors encoded in the activations of the input and output units of the network. Sequences of input symbols are presented to the network, one symbol at a time. For each input symbol presented, the target value for the output nodes is the vector representing the next symbol in the input sequence. Thus, the network is trained to predict the next symbol. This approach has been used by Elman (1990, 1991a), Cleeremans, Servan-Schreiber, and McClelland (1989), Cleeremans and McClelland (1991), and Servan-Schreiber, Cleeremans and McClelland (1988, 1989, 1991).

We now develop a statistical interpretation of this training process. The derivation loosely follows that of White (1989) who has proven a similar (though more extensive) result for non-recurrent (spatial) networks and simple spatial input-output mappings. We represent the partial input strings by the variable, U, and the predicted symbol by the variable, Y. We next suppose that the partial sentences, U, are generated according to a probability law, μ, where the probability of partial sentence U is given by $\mu(U)$. In effect, μ represents the training environment. Since there is no one correct predicted symbol Y for each input U, the relation between the two variables must also be governed by a probability law. Specifically, we say that the probability of a symbol, Y, being the correct symbol for a specific input string u is $\gamma(Y|u)$. Here, γ represents the stochastic relationship between partial input strings and next output symbols.

If output symbols, Y, are encoded as unit normal vectors, $\vec{Y}$, then the expected output for any string, U, can be defined as a function computing an expected vector:

$$\vec{E}(\vec{Y}|U) = \int_{\vec{y}} \vec{y} \cdot \gamma(\vec{y}|U)\ dy .$$

Similarly, we define the output vector of a network with weights, W, in response to any input string, U, as $\vec{\Phi}(U,W)$. Here, the function $\vec{\Phi}(U,W)$ represents repeated applications of the state function $f_{\vec{s}}(\cdot)$, applied to the output function $f_{\vec{y}}(\cdot)$. Specifically:

$$\vec{\Phi}(U,W) = f_{\vec{y}}(\ f_{\vec{s}}(\ \cdots\ f_{\vec{s}}(\ f_{\vec{s}}(\ \vec{s}(0),\ \vec{x}(0),\ W\ ),\ \vec{s}(1),\ \vec{x}(1),\ W\ )\ \cdots\ ,\ \vec{x}(t),\ W\ ),\ W\ )$$

where U represents the input sequence $\vec{x}(0), \vec{x}(1), \vec{x}(2), \ldots, \vec{x}(t)$. Ideally, we would like the two equations $\vec{E}(\vec{Y}|U)$ and $\vec{\Phi}(U,W)$, to always produce the same result. More specifically, we would like the expected difference, between $\vec{E}(\vec{Y}|U)$ and $\vec{\Phi}(U,W)$ to be as small as possible. We define the difference, between the two vectors according to their squared Euclidean difference:

$$\left\| \vec{E}(\vec{Y}|u) - \vec{\Phi}(u,W) \right\|^2 ,$$

and the expected difference as:

$$E\left\{ \left\| \vec{E}(\vec{Y}|U) - \vec{\Phi}(U,W) \right\|^2 \right\} = \int_u \left\| \vec{E}(\vec{Y}|u) - \vec{\Phi}(u,W) \right\|^2 \mu(du) .$$

It is this value which should ideally be minimized. Note that, in order to minimize this function, the probability laws γ and μ must be discovered by the learner.

Learning algorithms for STCNs are not specifically designed to approximate the expected output function $\vec{E}(\vec{Y}|S)$ or equivalently to discover the probability laws γ and μ. Instead they are designed to perform acceptably well according to a predefined error measure. In Section 2.5, we defined the error of a STCN as:

$$\varepsilon = \sum_t \left\{ \frac{1}{2} \left\| \vec{y}^{\,*}(t) - \vec{y}(t) \right\|^2 \right\}$$

In the context of the training algorithm described above, this equation can be rewritten as an expectation function:

$$\varepsilon = E\left(\frac{1}{2} \left\| \vec{Y} - \vec{\Phi}(U,W) \right\|^2 \right)$$

The process of adapting the weights in a STCN is designed to minimize this equation. We now prove that it also minimizes the difference between the expected value of the output vector, $\vec{E}(\vec{Y}|u)$, and the actual network output, $\vec{\Phi}(u,W)$, for a given string u:

***Theorem 6-2*:** Minimizing $E\left(\frac{1}{2} \left\| \vec{Y} - \vec{\Phi}(U,W) \right\|^2 \right)$ also minimizes $E\left\{ \| \vec{E}(\vec{Y}|U) - \vec{\Phi}(U,W) \|^2 \right\}$.

Intuitively, this theorem can be expressed as: training the individual outputs of a network to match the individual next-symbols automatically trains the output of the network to match the best prediction for the next-symbol.

***Proof*:**

The equation for the error of a network can be rewritten:

$$\varepsilon = \frac{1}{2}E\left(\left\| \vec{Y} - \vec{E}(\vec{Y}|U) + \vec{E}(\vec{Y}|U) + \vec{\Phi}(U,W) \right\|^2 \right)$$

$$= \frac{1}{2}E\left(\left\| \vec{Y} - \vec{E}(\vec{Y}|U) \right\|^2 + \left\| \vec{E}(\vec{Y}|U) - \vec{\Phi}(U,W) \right\|^2 + 2\left\| \vec{Y} - \vec{E}(\vec{Y}|U) \right\| \left\| \vec{E}(\vec{Y}|U) - \vec{\Phi}(U,W) \right\| \right)$$

$$= \frac{1}{2}E\left(\left\| \vec{Y} - \vec{E}(\vec{Y}|U) \right\|^2 \right) + \frac{1}{2}E\left(\left\| \vec{E}(\vec{Y}|U) - \vec{\Phi}(U,W) \right\|^2 \right) + E\left(\left\| \vec{Y} - \vec{E}(\vec{Y}|U) \right\| \left\| \vec{E}(\vec{Y}|U) - \vec{\Phi}(U,W) \right\| \right)$$

The final term in this equation can now be expanded:

$$E\left(\left\| \vec{Y} - \vec{E}(\vec{Y}|U) \right\| \left\| \vec{E}(\vec{Y}|U) - \vec{\Phi}(U,W) \right\| \right)$$

$$= E\left(E\left\{ \left\| \vec{Y} - \vec{E}(\vec{Y}|U) \right\| \left\| \vec{E}(\vec{Y}|U) - \vec{\Phi}(U,W) \right\| \mid U \right\} \right)$$

$$= E\left(E\left\{ \left\| \vec{Y} - \vec{E}(\vec{Y}|U) \right\| \mid U \right\} E\left\{ \left\| \vec{E}(\vec{Y}|U) - \vec{\Phi}(U,W) \right\| \mid U \right\} \right)$$

$$= E\left(\left\{ 0 \right\} E\left\{ \left\| \vec{E}(\vec{Y}|U) - \vec{\Phi}(U,W) \right\| \mid U \right\} \right)$$

$$= 0$$

Therefore, the error is equal to:

$$\varepsilon = \frac{1}{2}E\left(\left\| \vec{Y} - \vec{E}(\vec{Y}|U) \right\|^2 \right) + \frac{1}{2}E\left(\left\| \vec{E}(\vec{Y}|U) - \vec{\Phi}(U,W) \right\|^2 \right).$$

This implies that minimizing the error of a STCN also minimizes:

$$E\left(\left\| \vec{E}(\vec{Y}|U) - \vec{\Phi}(U,W) \right\|^2 \right),$$

the expected difference between the expected output symbol and the actual output of the network.□

This result also allows us to address the issue of generalization:

Theorem 6-3: If W^* is a set of weights minimizing the network error ε, then W^* generalizes optimally *by construction* in the sense that: given an input string randomly selected from the probability distribution, μ, and a corresponding output randomly selected from the conditional probability distribution, γ, the network output, $\vec{\Phi}(U,W^*)$, has the best possible average performance; i.e. it generalizes optimally.

Note that Theorems 6-2 and 6-3 are based only on a specific error definition and are independent of network architecture (e.g. state function and output function). They can be applied equally effectively to first-order vs. second-order, recurrent vs. non-recurrent, multi-layered vs. single layered networks. The only necessary criteria are that the network is trained to predict next symbols encoded as binary vectors (which is done for Gold's generator problem, but not for Gold's acceptor problem), and that the error ε is minimized. Of course, the latter criterion may not be realized when local minima occur.

6.4.6 A New Stochastic Grammar Induction Algorithm that is not New

Since training networks to predict subsequent symbols for partial strings has now been formally proven to optimally induce stochastic automata, we can recognize this technique as a new stochastic automata induction algorithm. That is, while the algorithm is old, its identification as a stochastic automata induction algorithm is new. Recall that we noted in Section 6.4.3 that a stochastic grammar induction algorithm should not only attempt to fit its induced automata to the training data, but should also induce automata of limited complexity. Training STCNs to predict subsequent symbols according to the gradient descent techniques discussed in Chapter II, satisfies the latter constraint in the sense that the complexity of the network is limited by the size of the weight matrix W. The larger the matrix the larger the set of automata which can be represented. Chapter IV has described classes of deterministic automata which can be implemented by networks of various types and sizes. It would be similarly possible to describe specific types of stochastic automata which can be implemented by networks of various sizes.

The key to this newly recognized algorithm's potential advantages over that described by Horning lies in the fact that it does not use an exhaustive (iterative) search algorithm to find a solution grammar. Instead, it samples the hypothesis space at various points and "guesses" (based on the error gradient) where the most probably correct grammar might lie. This selective sampling approach is clearly to be preferred in a domain where the hypothesis space is continuous, and thus infinite. In a sense, the algorithm simultaneously considers multiple solutions since the error gradient reveals information about neighbouring points in the hypothesis space. By contrast, Horning's symbolic approach can only consider one potential grammar at a time. The difference between the two algorithms is similar to the difference between interior and exterior techniques in linear programming—while Horning's algorithm follows the edges of a

predefined simplex (whose corners represent the a priori probable grammars), the STCN approach can explore the search spaces that embed the target space used by Horning.

Of course, Horning's is not the only stochastic grammar induction algorithm. It was chosen for this discussion because of the natural way in which it could be compared and contrasted to the STCN error minimization technique. Future work should explore the relationship between other symbolic stochastic grammar induction techniques and STCN training.

6.4.7 Conclusions

In this section we have examined a different grammar induction paradigm. By focusing on the generation of a language (as opposed to its recognition), we have defined stochastic languages, grammars, and automata which define probabilities with which strings occur. It is possible to describe these stochastic formalisms by means of predictor automata which, for any string of symbols, predict the next symbol to be generated. It is important to address the issue of similarity vs. simplicity when judging the automata induced for various samples. If an automaton accurately represents a sample, it not only represents the original grammar from which the sample was generated, but it also represent the inherent noise in the generation of the sample. This can result in induced automata which are far mor complex than the original source of the sample. Thus, it is necessary to compromise between the similarity between the language represented by the induced automaton as compared to the sample and the simplicity of the induced automaton. Another complication of stochastic grammar induction results from the fact that it is infeasible to sample the solution space at predefined points and expect to efficiently find a reasonable solution grammar. We have seen how STCN grammar induction schemes overcome these difficulties by adjusting their sampling according to the error gradient of the current candidate grammar and the advantages that lie therein. Finally, we have

proven that a popular STCN grammar induction algorithm actually computes the most probably correct stochastic automaton for a given set of sample strings.

6.5 CONCLUSIONS

This chapter has examined the use of a posteriori knowledge for find target grammars for the grammar induction problem. Unlike a priori knowledge, the knowledge discussed in this chapter is provided to the learner throughout the learning process and guides its path through the hypothesis space. We have examined three forms of a posteriori knowledge: input ordering, automaton information, and frequency coding. We have provided formal proofs confirming the validity of empirical observations regarding the use of these forms of prior knowledge. Clearly, anyone working in the field of grammar induction must consider these alternatives if they wish to develop effective and efficient induction algorithms.

Chapter VII: Conclusions

7.1 A THEORY OF GRAMMATICAL INDUCTION IN THE CONNECTIONIST PARADIGM

Over the past 30 years there has been a great deal of theoretical and experimental work on symbolic grammatical induction in diverse fields, including artificial intelligence, machine learning, computer science, psychology, linguistics, philosophy and cognitive science. The theoretical work in these areas has led directly to better machine induction systems and hence superior experimental results. Despite these successes, effective grammatical induction systems remain a distant goal for many practical applications. In the past decade, a new paradigm within artificial intelligence called "connectionism" has been developed. Connectionism has shown itself most promising for those tasks that seem effortless for humans (such as language and perception) but which have been extremely

difficult to model within conventional computing systems. Recently, connectionist networks have been applied to grammatical induction problems with some success. While these efforts have provided significant empirical data on the applicability of connectionist networks to grammatical induction problems, a formal theory of grammatical induction in the connectionist paradigm was missing.

This dissertation represents the first attempt to develop a theory of grammatical induction for this new paradigm. Like any scientific theory, ours should not only attempt to account for existing empirical observations, but also make new predictions which can be empirically verified. In so far as the predictions made by the theory relate to the performance of connectionist grammar induction systems, the theory can be used to predict the performance of existing systems as well as to evaluate that of proposed systems. In this sense, a theory of grammatical induction in the connectionist paradigm can be used to design superior connectionist grammar induction systems.

In the introductory chapter several important issues were introduced. In particular, we claimed that the tractability and efficiency of training particular connectionist networks to implement certain classes of grammars can be formally determined by applying principles and ideas that have been explored in the symbolic grammatical induction paradigm. Furthermore, we argued that this formal analysis also allows networks to be tailored to efficiently solve specific grammatical induction problems. In this concluding chapter we shall re-examine these claims. Before evaluating the validity of these conjectures, however, we must review the results presented in this dissertation.

7.2 RESULTS

In Chapter II we described an new type of taxonomy for spatio-temporal connectionist networks based on a formal mathematical description of grammatical induction. The new taxonomy is the only one developed around the fundamental design

decisions which must be addressed by any grammatical induction system. Because of this, it has superior predictive power when used to compare and analyse how different STCNs might perform on various grammatical induction problems. Additionally, the taxonomy is general enough to accommodate all of the leading STCN designs described in the literature. In fact, the exact categorization of each leading design is precisely specified. Furthermore, the fact that the taxonomy is based on the principles of grammatical induction systems rather than specific existing STCN designs implies that it will be able to easily accommodate future STCN designs as well.

In addition to surveying the field of STCNs for grammatical induction, we provided a simple way of applying the results of the following chapters to many different STCN designs currently in use, as well as future STCN designs. By focusing on components of the grammatical induction task which are implemented in the same manner in different networks, the taxonomy complements the following chapters in allowing theoretical conclusions based upon the implementations of the components to be directly applied to multiple network designs. Furthermore, if future STCN designs embody the same solutions to components of the induction task as existing systems, then predictions about the performance of networks which have never been implemented can be made. In this manner, the taxonomy can serve as a reference for anyone needing to select or design a particular STCN to solve a given problem.

Chapter III provided a formal description of the problem of grammatical induction. It defined much of the terminology for the chapters which followed and described, in general, how STCNs can be applied to grammatical induction. Additionally, Chapter III described the inherent difficulties of the problem and two broad techniques which have been used to overcome them. By providing a comprehensive survey of grammatical induction results relevant to the design of new grammatical induction systems, this chapter identified the important issues that connectionist approaches to the problem must address.

In the past, connectionist researchers have paid little attention to formal language learnability results and to the approaches used by symbolic induction systems to make the induction problem tractable and efficient. This chapter explained why these formal results are critically important to the users of connectionist grammar induction systems.

The fourth chapter examined the restrictions that various STCN designs place on the languages and automata which they can represent. A clear understanding of the hypothesis space used by an induction system is vital since an hypothesis space which is too large can easily make a problem unsolvable or at the least intractable, whereas an hypothesis space which is too small may not include any acceptable solutions to the problem. Thus, a connectionist researcher's choice of STCN design directly affects induction speed and success. This observation is important because almost all connectionist grammar induction systems have, in the past, been designed on the basis of ease of implementation or expansion from existing networks.

In particular, Chapter IV presented new formal proofs describing the representable languages and automata for five of the seven possible memory functions identified in Chapter II. Specifically, we proved a new result describing the types of languages implementable by window in time memories. Then, we proved that single-layer first-order context computation memories can implement arbitrary finite state automata and that they can do so using $n \cdot p$ nodes (where n is the number of states, and p is the number of input symbols). We also proved, that single-layer second-order context computation memories are incapable of implementing arbitrary automata using binary state encodings. Next, we proved that a locally recurrent state and input memory is incapable of representing finite state automata whose state transitions form cycles of length greater than two under oscillation input. We also proved another even more dramatic result regarding the limitations of LRSI memories relating input cycles to state transitions cycles. Finally, we proved that a 1-layer output function can compute all mappings from split state to output.

In addition to discovering and proving these individual new results, Chapter IV presented a table relating memory-types, output functions, formal-computing machines and number of nodes. This is the first synthesis of this kind and can serve as a guide to anyone intending to use a STCN for grammatical induction in their selection of a particular STCN design and choice of size.

Chapter V described how the users of STCNs can reduce and order the space of potential grammars to suit a particular problem. This is critical for the design of any efficient grammar induction algorithm. We showed that fixing some of the weights in a STCN limits the hypothesis space, and that initializing weights can both order and restrict the hypothesis space. An additional proof described how good *a priori* knowledge, encoded in initial weights, tends to limit the hypothesis space much more than bad *a priori* knowledge. Previous work by connectionist researchers on encoding *a priori* knowledge into their networks has been exclusively empirical. This chapter gave a formal grounding to these empirical results.

Chapter VI examined the use of *a posteriori* knowledge, provided during training, to guide a grammar induction system's exploration of the hypothesis space. In particular, it related the work of connectionist researchers to known results for grammar induction. We proved that Elman's multi-phase training algorithm only uses a small number (440) of distinct simple strings during the first phase of training. This implies that it is highly likely that all simple strings are presented during the first phase of training. This is significant since it implies that Elman's presentation scheme corresponds to an input ordering scheme (something that Elman himself was unclear about) and in particular that the presentation scheme is identical to one which has already been studied and proven effective by Feldman. Another new result shown in Chapter VI was the fact that presentation order can affect which local minimum's basin of attraction is descended in a STCN. This was proven via a simple example.

Finally, Chapter VI presented a proof that a popular training scheme used for STCNs involves training these networks to identify the most probably correct stochastic grammar for a given sample. We started with an illustration and a proof that an exact representation of a stochastic training sample is undesirable and that stochastic grammar induction must balance similarity to the training set with simplicity of the induced automaton. Next we examined a popular approach to inducing stochastic grammars and presented a new proof that the approach will perform very poorly for large data sets, complex candidate grammars, or large numbers of candidate grammars. This theoretical result confirmed the empirical evidence previously gathered by Horning. Having motivated the search for a better stochastic grammar induction algorithm, we presented a new proof that a STCN's error is minimized when the network encodes the most probably correct stochastic grammar. This was a novel result because minimizing the average error between network output and the actual next symbol in a string is not necessarily equivalent to minimizing the average error between a network output and the expected next symbol in a string. We thus also proved that by defining a STCN's error as the average Euclidean difference between individual outputs and next symbols, we can ensure that minimizing the error will result in a network which generalizes optimally for the given probability laws: $\gamma:(y,u)\rightarrow\mathbb{R}$ and $\mu:u\rightarrow\mathbb{R}$. This result proved that a popular STCN training algorithm is in fact an optimal stochastic grammar induction algorithm. We also discussed the advantages of this newly recognized algorithm.

7.3 CONCLUSIONS

Together the chapters of this dissertation represent a thorough theoretical study of grammatical induction in a connectionist paradigm. The thesis has revealed, for the first time, that many of the techniques developed to make grammar induction tractable and efficient in a symbolic paradigm not only can be, but have been adapted for use in the connectionist domain. Additionally, the thesis provides numerous new formal results that form the basis of a first theory of grammatical induction in the connectionist paradigm. It is hoped that this theory will guide future connectionist researchers to develop more effective and more efficient induction systems.

In this dissertation, we have seen that grammatical induction is a very difficult problem in the sense that it is intractable under the most natural assumptions. Furthermore, the sheer numbers of grammars make brute force enumerative searches too inefficient to be of practical use for all but the simplest induction problems. We have shown that these observations made by researchers in the symbolic paradigm are just as applicable to those working with connectionist systems. In order to overcome the difficulties of grammatical induction in the symbolic paradigm, two broad approaches have been employed: (1) reducing the number of grammars which are considered, and (2) selecting an order of exploration of the space of grammars which favours the most likely candidates. Both of these approaches assume that there is some form of additional knowledge about the task which can be used to make informed choices about the hypothesis space of candidate grammars.

We have shown by example that it is possible to identify the hypothesis space used in connectionist systems and relate them to symbolic representations of languages. Further we have seen how fixing and initializing weights can be used to further restrict and order the hypothesis spaces used and thus change the tractability and efficiency of grammar induction. Finally, we discovered that if information other than the grammaticallity of

strings is available to a connectionist network, it too be used to help solve grammatical induction problems. Thus, we have formally determined the tractability and efficiency of training particular connectionist networks to implement certain classes of grammars, as we set out to do in our introductory chapter. The conclusions reached also allow the users of STCNs to select the various components of a STCN induction system in order to maximize its effectiveness for some given purpose. In this sense, our formal analysis allows networks to be tailored to efficiently solve specific grammatical induction problems. This was the second goal we introduced in the first chapter.

7.4 LIMITATIONS AND FUTURE WORK

Like any other scientific theory, the predictions made here will have to be empirically verified. No doubt such attempts at verification will inevitably result in the discovery of faults and limitations in this work. In turn, new theories will be developed to account for the shortcomings of this one, just as all scientific theories are continually tested and revised in the process known as the scientific method. In particular, we can identify a number of issues which future work will need to address.

The taxonomy presented in Chapter II incorporates the leading STCN designs for grammatical induction. No doubt, other connectionist grammar induction systems will be developed in the future. These systems should be analyzed according to the four basic components of any grammar induction system and added to the taxonomy. If a new system incorporates a different approach to one of the components, then a new point along one of the four dimensions of the taxonomy will need to be added, and of course the corresponding analyses for the following chapters.

Chapter IV identified the representational powers of various STCN memory and output functions. These results were summarized in Table 4-2. There are a number of open problems indicated in this table. Perhaps the most interesting of these is the issue

of whether or not Single-Layer Second-Order Context Computation memories are capable of universal computation. Future work should attempt to fill these gaps.

Throughout this dissertation, we assumed that various kinds of information about the grammatical problem to be solved is available. This information took the form of the classes of grammars which needed to be represented, non-optimal solutions to the problem, specific details about the automata to be induced, or information about the frequencies with which sample sentences occur. The assumption that this type of information is available is a necessity in the sense that without some additional source of knowledge, the grammatical induction problem is unsolvable (except for languages of finite size). We have not addressed whether or not the types of information assumed by the algorithms proposed in this dissertation are actually available in practical applications.

Of course, the validity of assuming that a given form of information is available depends entirely on the domain to which grammatical induction is applied. The problem of grammatical induction applies to much more than the learning of what we would normally call languages. In fact, it describes a process to learn relationships between information scattered across space and time. Specifically, grammatical induction has been applied to: modelling natural language learning, process control, signal processing, phonetic to acoustic mapping, speech generation, robot navigation, nonlinear temporal prediction, system identification, learning complex behaviours, motion planning, prediction of time ordered medical parameters, and speech recognition, to name but a few. Since this dissertation focuses only on general principles of grammatical induction (as opposed to an application specific approach), its scope does not include an analysis of the validity of assuming that certain types of information are actually available for a given application. None-the-less, future work will need to address this issue.

Bibliography

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

Angluin, D. (1976). *An application of the theory of computational complexity to the study of inductive inference.* Ph.D. dissertation, Department of Electrical Engineering & Computer Science, Univ. California, Berkeley.

Angluin, D. (1978). On the complexity of minimum inference of regular sets. *Information and Control,* **39**:337-350.

Back, A. D., & Tsoi, A. C. (1991). FIR and IIR synapses, a new neural network architecture for time series modelling. *Neural Computation,* **3**(3):375-385.

Chomsky, N. (1956). Three models for the description of language. *IRE Transactions on Information Theory,* **2**(3):113-124.

Chomsky, N. (1965). *Aspects of the Theory of Syntax.* Cambridge, Mass.: MIT Press.

Cleeremans, A., Servan-Schreiber, D., & McClelland, J. (1989). Finite state automata and simple recurrent networks. *Neural Computation,* **1**(3):372-381.

Cleeremans, A., & McClelland, J. L. (1991). Learning the structure of event sequences. *Journal of Experimental Psychology: General,* **120**(3):235-253.

Das, S., Giles, C. L., & Sun, G. Z. (1993). Using prior knowledge in a NNPDA to learn context-free languages. In S. J. Hanson, J. D. Cowan, & C. L. Giles (Eds.), *Advances in Neural Information Processing Systems 5*:65-72.

De Vries, B., & Principe, J. C. (1992). The gamma model - a new neural model for temporal processing. *Neural Networks,* **5**:565-576.

Elman, J. L. (1988). *Finding Structure in Time.* (CRL Tech. Rep. No. 8801). La Jolla, CA: Center for Research in Language, University of California at San Diego.

Elman, J. L (1989). *Representation and Structure in Connectionist Models.* (CRL Tech. Rep. No. 8903). La Jolla, CA: Center for Research in Language, University of California at San Diego.

Elman, J. L. (1990). Finding structure in time. *Cognitive Science,* **14**:179-211.

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

Elman, J. L. (1991b). *Incremental Learning, or the Importance of Starting Small.* (CRL Tech. Rep. No. 9101). La Jolla, CA: Center for Research in Language, University of California at San Diego.

Fahlman, S. E. (1991). The recurrent cascade-correlation architecture. In R. P. Lippman, J. E. Moody, & D. S. Touretzky, (Eds.), *Advances in Neural Information Processing Systems 3*:190-196.

Feldman, J. (1972). Some decidability results on grammatical inference and complexity. *Information and Control,* **20**:244-262.

Franklin, S., & Garzon, M. (1988). Neural network implementations of Turing machines. *The Proceedings of the Second Institute of Electrical and Electronics Engineers International Conference on Neural Networks.*

Frasconi, P., Gori, M., Maggini, M, & Soda, G. (1991). An unified approach for integrating explicit knowledge and learning by example in recurrent networks. *Proceedings of the International Joint Conference on Neural Networks,* **1**:811-816.

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

Frasconi, P., Gori, M., Maggini, M., & Soda, G. (in press). Unified integration of explicit rules and learning by example in recurrent networks". *IEEE Transactions on Knowledge and Data Engineering.*

Giles, C. L., Sun, G.Z., Chen, H.H., Lee, Y.C., & Chen, D. (1990). Higher order recurrent networks & grammatical inference. In D. S. Touretzky (Ed.), *Advances In Neural Information Processing Systems-2*:380-387. San Mateo: Morgan Kaufmann Publishers.

Giles, C. L., Chen, D., Miller, C. B., Chen, H. H., Sun, G. Z., & Lee, Y. C. (1991). Second-order recurrent neural networks for grammatical inference. *1991 IEEE INNS International Joint Conference on Neural Networks- Seattle,* 2:273-281.

Giles, C. L., Horne, B. G., & Lin, T. (in press). Learning a class of large finite state machines with a recurrent neural network, in *Neural Networks*.

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

Giles, C.L., Miller, C.B., Chen,D., Sun, G.Z., Chen, H.H., & Lee, Y.C. (1992b). Extracting and learning an unknown grammar with recurrent neural networks. In J.E. Moody, S.J. Hanson, & R.P. Lippmann, (Eds.), *Advances in Neural Information Processing Systems 4:*317-324. San Mateo: Morgan Kaufmann Publishers.

Giles, C.L., & Omlin, C.W. (1992c). Inserting rules into recurrent neural networks. In S. Y. Kung, F. Fallside, J. A. Sorenson, & C. A. Kamm, (Eds.), *Neural Networks for Signal Processing II, Proceedings of the 1992 IEEE Workshop:*13-22.

Giles, C. L., & Omlin, C. W. (1993a). Extraction, insertion and refinement of symbolic rules in dynamically-driven recurrent neural networks. *Connection Science.*

Giles, C. L., & Omlin, C. W. (1993b). Rule refinement with recurrent neural networks. *Proceedings IEEE International Conference on Neural Networks (ICNN'93)*, **2**:801-806.

Giles, C. L., Chen, D., Sun, G. Z., Chen, H. H., Lee, Y. C., & Goudreau, M. W. (1995). Constructive learning of recurrent neural networks: limitations of recurrent cascade correlation and a simple solution. *IEEE Transactions on Neural Networks*. **6**(4):829-836.

Gold, E. M. (1967). Language identification in the limit. *Information and Control*, **10**:447-474.

Goudreau, M. W., Giles, C. L., Chakradhar, S. T., & Chen, D. (1994). First-order Vs. second-order single layer recurrent neural networks. *IEEE Transactions on Neural Networks*, **5**(3):511-515.

Hopcroft, J. E., & Ullman, J. D. (1979). *Introduction to Automata Theory, Languages, and Computation*. Don Mills: Addison-Wesley.

Horne, B. G., & Giles, C. L. (1994). An experimental comparison of recurrent neural networks. *Advances in Neural Information Processing Systems 6*.

Horne, B. G., & Hush, D. R. (1994a). Bounds on the complexity of recurrent neural network implementations of finite state machines. In J. D. Cowen, G. Tesauro, & J. Alspector, (Eds.), *Advances in Neural Information Processing Systems 6*:359-366. New York: Morgan Kaufmann.

Horne, B. G., & Hush, D. R. (1994b). On the node complexity of neural networks. *Neural Networks*.

Horning, J. J. (1969). *A study of grammatical inference* (CS No. 139 & MEMO AI-98). Stanford, CA: Stanford University.

Huang, T., & Fu, K. S. (1971). On stochastic context-free languages. *Information Sciences*, 3:201-224.

Judd, S.J. (1990). *Neural Network Design and the Complexity of Learning*. Cambridge: MIT Press.

Kilian, J., & Siegelmann, H. T. (1993). On the power of sigmoid neural networks. *Proceedings of the Sixth ACM Workshop on Computational Learning Theory*:137-143.

Kleene, S. C. (1956). Representation of events in nerve nets and finite automata. In C. E. Shannon & J. McCarthy (Eds.), *Automata Studies*:3-42. Princeton: Princeton University Press.

Kohavi, Z. (1978). *Switching and Finite Automata Theory* (2nd ed.). New York:McGraw-Hill.

Kremer, S. C. (1995a). On the computational powers of Elman-style recurrent networks, *IEEE Transactions on Neural Networks*, 6(4):1000-1004.

Kremer, S. C. (1995b). *Comments on: constructive learning of recurrent neural networks: cascading the proof describing limitations of recurrent cascade correlation.* Manuscript submitted for publication.

Kremer, S. C. (1995c). *On the computational power of recurrent cascade correlation.* Manuscript submitted for publication.

Kremer, S. C. (in press). Finite state automata that recurrent cascade correlation cannot represent, In D. Touretzky, M. Mozer & M. Hasselmo (Eds.), *Advances in Neural Information Processing Systems 8.* Cambridge, MA: MIT Press.

Labov, W. (1969). Contraction, deletion and inherent variability of the English copula. *Language,* **45**:715-762.

Lang, K., Waibel, A., & Hinton, G. (1990). A time-delay neural network architecture for isolated word recognition. *Neural Networks,* 3(1):23-44.

Lapedes, A., & Farber, R. (1987). *Nonlinear signal processing using neural networks: prediction and system modelling.* (Tech. Rep. No. LA-UR87-2662). Los Alamos: Los Alamos National Laboratory.

Lippmann, R. P. (1987). An introduction to computing with neural nets. *IEEE ASSP Magazine,* (April):4-22.

Liu, Y. D., Sun, G. Z., Chen, H. H., Lee, Y. C., & Giles, C. L. (1990). Grammatical inference and neural network state machines. *Proceedings of the International Joint Conference on Neural Networks 1990,* 1:285-288.

Marr, D. (1982). *Vision*. San Francisco: W. H. Freeman.

Maskara, A., & Noetzel, A. (1992). Forced simple recurrent neural networks and grammatical inference. *Neural Networks for Signal Processing II.* Piscataway, NJ: IEEE.

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of ideas imminent in nervous activity. *Bulletin of Mathematical Biophysics,* 5:115-133.

McInerney, J.M., Haines, K.G., Biafore, S., and Hecht-Nielsen, R. (1989). Can backpropagation error surfaces have non-global minima? In *International Joint Conference on Neural Networks*, II, 627, New York: IEEE Press.

Miller, C. B., & Giles, C. L. (1993). Experimental comparison of the effect of order in recurrent neural networks. *International Journal of Pattern Recognition and Artificial Intelligence,* 7(4):849-872.

Minsky, M. (1967). Neural networks: automata made up of parts. *Computation: Finite and Infinite Machines*:32-66. Englewood Cliffs, NJ: Prentice-Hall.

Mozer, M. C. (1989). A focused backpropagation algorithm for temporal pattern processing. *Complex Systems,* 3(4):349-381.

Mozer, M. C. (1993). Neural net architectures for temporal sequence processing. In A. Weigend & N. Gershenfeld (Eds.) *Time series Prediction: Forecasting the Future and Understanding the Past:* 243-264, Redwood City, CA: Addision-Wesley Publishing.

Narenda, K., & Parthasarthy, K. (1990). Identification and control of dynamical systems using neural networks. *IEEE Transactions on Neural Networks,* 1(1):4-27.

Nerrand, O., Roussel-Ragot, P., Personnaz, L., Dreyfus, G., Marcos, S. (1993). Neural Networks and Non-linear Adaptive Filtering: Unifying Concepts and New Algorithms. *Neural Computation*, 5:165-197.

Newell, A., & Simon, H.A. (1972). *Human Problem-Solving.* Englewood Cliffs, NJ: Prentice Hall.

Omlin, C. W., & Giles, C. L. (1992). Training second-order recurrent neural networks using hints. In D. Sleeman & P. Edwards (Eds.), *Proceedings of the Ninth International Conference on Machine Learning*:363-368. San Mateo, CA: Morgan Kaufmann.

Omlin, C. W., Giles, C. L., & Miller, C. B. (1992). Heuristics for the extraction of rules from discrete-time recurrent neural networks. *Proceedings of the International Joint Conference on Neural Networks,*1:33-38. Piscataway, NJ, IEEE Press.

Omlin, C. W., & Giles, C. L. (1994a). Constructing deterministic finite-state automata in sparse recurrent neural networks. *IEEE International Conference on Neural Networks,* **1**: 1732-1737. Piscataway, NJ, IEEE Press.

Omlin, C. W., & Giles, C.L. (in press). Constructing deterministic finite-state automata in recurrent neural networks. *JACM.*

Omlin, C. W., & Giles, C. L. (in press). Stable encoding of large finite-state automata in recurrent neural networks with sigmoid discriminants. *Neural Computation.*

Pinker, S. (1979). Formal models of language learning. *Cognition,* **7**:217-283.

Poddar, P., & Unnikrishnan, K.P. (1991). Non-linear prediction of speech signals using memory neuron networks. *Neural Networks for Signal Processing: Proceedings of the 1991 IEEE Workshop*:395-404. Piscataway, NJ, IEEE Press.

Pollack, J. B. (1987). *On connectionist models of natural language processing.* Unpublished Ph.D. thesis, University of Illinois.

Pollack, J. B. (1989). Implications of recursive distributed representations. In D. S. Touretzky (Ed.), *Advances in Neural Information Processing Systems 1*: 527-536. San Mateo. CA: Morgan Kaufmann.

Pollack, J. B. (1990). Recursive distributed representations. *Journal of Artificial Intelligence*, **46**:77-105.

Pollack, J. B. (1994). *Limits of connectionism.* Unpublished manuscript.

Porat, S., & Feldman, J. A. (1991). Learning automata from ordered examples. *Machine Learning,* **7**(2-3):109-138.

Reber, A.S. (1967). Implicit learning of artificial grammars. *Journal of Verbal Learning and Verbal Behavior,* **5**:855-863.

Rich, E. (1983). *Artificial Intelligence.* New York: McGraw Hill.

Robinson, A., & Fallside, F. (1988). Static and dynamic error propagation networks with application to speech coding. In D. Z. Anderson (Ed.), *Neural Information Processing Systems:* 632-641. New York: American Institute of Physics.

Rumelhart, D., Hinton, G., & Williams, R. (1986). Learning internal representation by error propagation. In J. L. McClelland, D. E. Rumelhart, & the P. D. P. Group (Eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition,* (vol. 1). Cambridge, MA: MIT Press.

Salomaa, A. (1969). *Theory of Automata.* Oxford: Pergamon Press.

Sankoff, D. (1978). Probability and linguistic variation. *Synthese,* **37**:217-238.

Schmidhuber, J. (1992). A fixed size storage $O(n^3)$ time complexity learning algorithm for fully recurrent continually running networks. *Neural Computation*, **4**:243-248.

Sejnowski, T. J. , & Rosenberg, C. R. (1986). *NETtalk: a parallel network that learns to read aloud* (Tech. Rep. No. JHU/EECS-86/01). Baltimore: Johns Hopkins University, Department of Electrical Engineering and Computer Science.

Servan-Schreiber, D., Cleeremans, A., & McClelland, J. L. (1988). *Encoding sequential structure in simple recurrent networks* (Tech. Rep. No. CMU-CS-88-183). Pittsburgh: Carnegie-Mellon University, Department of Computer Science.

Servan-Schreiber, D., Cleeremans, A., & McClelland, J. L. (1989). Learning sequential structure in simple recurrent networks. In D. S. Touretzky, (Ed.), *Advances in Neural Information Processing Systems 1*. San Mateo, CA: Morgan Kaufmann.

Servan-Schreiber, D., Cleeremans, A., & McClelland, J. L. (1991). Graded state machine: the representation of temporal contingencies in simple recurrent networks. *Machine Learning,* 7(2-3):161-193.

Shaw, A., & Mitchell, R. A. (1990). Phoneme recognition with a time-delay neural network. *International Joint Conference on Neural Networks, San Diego* (vol. 2):191-195.

Siegelmann, H. T., & Sontag, E. D. (1991). Turing computability with neural nets. *Applied Mathematics Letters,* 4(6):77-80.

Siegelmann, H. T., & Sontag, E. D. (1992). On the computational power of neural nets. *Proceedings of the Fifth ACM Workshop on Computational Learning Theory*:440-449. New York: ACM.

Siegelmann, H. T., & Sontag, E. D. (1995). On the Computational Power of Neural Nets. In *Journal of Computer and System Sciences*, **50**(1):132-150.

Stolcke, A. (1990). *Learning feature-based semantics with simple recurrent networks* (Tech. Rep. No. TR-90-015). ICSI.

Stolcke, A., Omohundro, S.M. (1994). *Best-first Model Merging for Hidden Markov Model Induction.* (Tech. Rep. No. TR-94-003). International Computer Science Institute, Berkeley, CA.

Sun, G. Z., Chen, H. H., Giles, C. L. Lee, Y. C., & Chen, D. (1990). Connectionist pushdown automata that learn context-free grammars. In M. Caudill (Ed.), *Proceedings of the International Joint Conference on Neural Networks 1990:* 577-580. Hillsdale, NJ: Lawrence Erlbaum.

Sun, G. Z., Chen, H. H., Giles, C. L., Lee, Y. C., & Chen, D. (1990). Neural networks with external memory stack that learn context-free grammars from examples. *Proceedings of the 1990 Conference on Information Sciences and Systems:* 649-653 (vol. 2).

Sun, G. Z., Chen, H. H., Lee, Y. C., & Giles, C. L. (1991). Turing equivalence of neural networks with second order connection weights. *1991 IEEE INNS International Joint Conference on Neural Networks- Seattle*:357-362. Piscataway, NJ: IEEE.

Sun, G. Z., Giles, C. L., Chen, H. H., & Lee, Y. C. (1993). *The neural network pushdown automaton: model, stack and learning simulations.* (UMIACS-TR-93-77/CS-TR-3118) University of Maryland.

Trakhtenbrot, B. A., & Barzdin, Y. M. (1973). *Finite Automata.* Amsterdam: North-Holland.

Tsoi, A. C., & Back, A. (1994). Locally recurrent globally feedforward networks: a critical review of architectures. *IEEE Transactions on Neural Networks,* 5(2):229-239.

Turing, A. M. (1936). On computable numbers with an application to the Entscheidungsproblem. *Proceedings of the London Mathematics Society,* 2(42):230-265.

Valiant, L.G. (1984). A theory of the learnable. *Communications of the ACM,* 27:1134-1142.

Waibel, A., Hanazawa, T., Hinton, G., Shikano, K., & Lang, K. (1989). Phoneme recognition using time-delay neural networks. *IEEE Transactions on Acoustics Speech and Signal Processing,* 37(3):328-339.

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

Watrous, R. L., & Kuhn, G. M. (1992). Induction of finite-state automata using second-order recurrent networks. In J. E. Moody, S. J. Hanson, & R. P. Lippmann

(Eds.), *Advances in Neural Information Processing Systems*:309-316. San Mateo, CA: Morgan Kaufmann.

Wexler, K., & Culicover, P. (1980). *Formal Principles of Language Acquisition*. Cambridge, Mass.: MIT.

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

White, H. (1989). Learning in artificial neural networks: a statistical perspective. *Neural Computation*, **1**:425-464.

Zeng, Z., Goodman, R. M., & Smyth, P. (1993). Learning Finite State Machines with Self-Clustering Recurrent Networks. *Neural Computation*, **5**(6):976-990.

Zeng, Z., Goodman, R. M., & Smyth, P. (1994). Discrete recurrent neural networks for grammatical inference. *IEEE Transactions on Neural Networks*, **5**(2):320-330.