ABSTRACT

This dissertation addresses the issues surrounding the computational capabilities of recurrent neural networks. My results apply not only to simple recurrent networks, Jordan networks, and higher order recurrent networks, but many other networks implemented as input-parameterized iterated functions.

The following reasons have driven my efforts to understand their computational capabilities. First, the question of knowledge content arises whenever we attempt to understand how a given network produces its behavior. Second, knowing the range of what is computable by a recurrent network can guide us in their intelligent application. Finally, this knowledge may also help us to develop new training strategies which bias the network towards desirable solutions.

While we already know that recurrent networks can perform complex computation by simulating machine tapes and stacks, one problem still remains: someone designed each universal-computing network by hand. We know the function decomposition because the designer can tell us what they intended each part to do. Unfortunately, weak learning methods, like back-propagation, that discover operable network weights cannot explain the internal functionality of final product. Thus, we are forced to externally determine the recurrent network's computation process by observing its structure and behavior.

To this end, I identify three facets of recurrent networks that directly affect their emergent computational descriptions: system dynamics, input modulation of state dynamics, and output generation. System dynamics, the mapping of current state to next state, have been traditionally considered the source of complex behavior. Input modulation occurs as a finite set of input vectors and induces beiterated function system-like behavior from the recurrent network. This selection creates state space representations for information processing states which display recursive structure. I show that the mechanism producing discrete outputs has dramatic effects on the resulting system complexity by imposing information processing regularities in the output stream strong enough to manipulate both complexion (number of states) and generative class of the observed computation.

As for new training methods, I outline a method of network training called entrainment learning which offers a novel explanation of the transmission of grammatical behavior structures between agents.

Very few beings really seek knowledge in this world. Mortal or immortal, few really ask. On the contrary, they try to wring from the unknown the answers they have already shaped in their own minds—justification, confirmation, forms of consolation without which they can't go on. To really ask is to open the door to a whirlwind. The answer may annihilate the question and the questioner.

Anne Rice, **The Vampire Lestat** (p. 332)

EXPLORING THE COMPUTATIONAL CAPABILITIES OF RECURRENT NEURAL NETWORKS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for

the Degree Doctor of Philosophy in the Graduate

School of The Ohio State University

By

John F. Kolen, B.A., M.S.

* * * * *

The Ohio State University

1994

Dissertation Committee:

J. B. Pollack

B. Chandrasekaran

D. Wang

Approved by

Adviser
Department of Computer and
Information Sciences

To Mary Jo

ACKNOWLEDGEMENTS

While a dissertation has a single author by definition, many people are responsible for its existence and should be recognized for their effort and contributions. Dr. Jordan B. Pollack, my advisor for six years, has had the most effect on the content of this document. Although he insisted on solving the big problems and I demanded feasibility in the projects that I worked on, we eventually settled upon a collection of results that satisfied both constraints. The other members of my dissertation committee, Drs. B. Chandrasekaran and DeLiang Wang, provided useful suggestions and comments in preparing the final version.

As Dr. Pollack's first student at OSU, I was in a position to watch his research team grow over the years. The other post-connectionist pre-docs, Dr. Peter Angeline, Edward Large, Viet-Anh Nguyen, Gregory Saunders, and David Stucki, have been an incredible group of people to work with. I can not thank them enough for helping me sift good ideas from bad hallucinations, sharpen salient arguments from incomprehensible drivel, and

I also wish to thank the other members of the Laboratory for Artificial Intelligence Research, both past and present: Drs. Ashok Goel and Dean Alemang for advising me during my first years at OSU, Barbara Becker for reading my papers even though they had nothing to do with systemic grammars, Arun Welch and Bryan Dunlap for keeping nervous running, and finally Dr. Dale Moberg for his enlightening discussions on just about everything.

I should not forget to mention the emotional support of the Dead Researchers Society (you know who you are). Our many meetings have helped soothe the anguish inflicted by troublesome advisors, collapsing research fields, and upheaving job markets.

I officially express gratitude to my two sources of financial support during my graduate education. My first five quarters at OSU were supported by the Department of Computer and Information Science through a graduate teaching assistantship. The remainder of my support as a graduate research assistant came from the Office of Naval Research by way of grant numbers N00014-89-J-1200 and N00014-92-J1195.

These acknowledgments would not be possible without the early efforts of my parents. Their hard work and sacrifices gave me the opportunity to choose the path that led here.

Last but not least, I thank Mary Jo for everything all that she has done over the last five years. Her love carried me though the dark times. Her companionship offset the loneliness of finishing the dissertation. Her understanding helped me survive graduate school.

VITA

December 17, 1965	Born - San Diego, California
1987	B.A., University of California at San Diego, La Jolla, California
1987-1988	Graduate Teaching Assistant, The Ohio State University, Columbus, Ohio
1988	M.S., The Ohio State University, Columbus, Ohio
1989-Present	Graduate Research Assistant, The Ohio State University, Columbus, Ohio

PUBLICATIONS

Kolen, J. F. (1994) Fool's Gold: Extracting Finite State Machines From Recurrent Network Dynamics. In J. D. Cowan, G. Tesauro, and J. Alspector (eds.), *Advances in Neural Information Processing Systems 6*, 501-508. San Mateo, CA:Morgan Kaufmann.

Kolen, J. F. (1994) Recurrent Networks: State Machines or Iterated Function Systems?. In M. C. Mozer, P. Smolensky, D. S. Touretzky, J. L. Elman, & A. S. Weigend (Eds.), *Proceedings of the 1993 Connectionist Models Summer School*, 203-210. Hillsdale, NJ: Erlbaum Associates.

Kolen, J. F., and J. B. Pollack. (1993) The apparent computational complexity of physical systems. In *The Proceedings of The Fifteenth Annual Conference of the Cognitive Science Society,* 617-622. .Hillsdale, NJ:Erlbaum.

Large, E. W., and J. F. Kolen. (1993) A dynamical model of the perception of metrical structure. In *Proceedings of the Society for Music Perception and Cognition*. Philadelphia, PA. May 1-3, 1993.

Saunders, G., J. F. Kolen, P. J. Angeline, and J. B. Pollack. (1992) Additive modular learning in preemptrons. In *The Proceedings of The Fourteenth Annual Conference of the Cognitive Science Society*, 1098-1103. Hillsdale, NJ:Erlbaum.

Kolen, J. F. and A. K. Goel. (1991) Learning in Parallel Distributed Processing networks: Computational complexity and information content. *IEEE Transactions on Systems, Man, and Cybernetics*, 21, 359-367.

Kolen, J. F. and J. B. Pollack. (1991) Multiassociative Memory. In *The Proceedings of The Thirteenth Annual Conference of the Cognitive Science Society,* 785-790. Hillsdale, NJ:Erlbaum.

Kolen, J. F. and J. B. Pollack. (1991) Back Propagation is Sensitive to Initial Conditions. In R. P. Lippman, J. E. Moody, and D. S. Touretzky (eds.), *Advances in Neural Information Processing Systems 3*, 860-867. San Mateo, CA:Morgan Kaufmann.

Kolen, J. F. and J. B. Pollack. (1990) Back Propagation is Sensitive to Initial Conditions. *Complex Systems*, 4, 269-280.

Kolen, J. F. and J. B. Pollack. (1990) Scenes from Exclusive-Or: Back Propagation is Sensitive to Initial Conditions. In *The Proceedings of The Twelfth Annual Conference of the Cognitive Science Society,* 868-874. Hillsdale, NJ:Erlbaum.

Kolen, J. F. (1989) Review of "Fast learning in artificial neural systems: Multilayer perceptron training using optimal estimation". *Neural Network Review*, 3.

Goel, A. K., J. F. Kolen, and D. Allemang. (1988) Learning in connectionist networks: Has the credit assignment problem been solved?, In *The Proceedings of the SIGART Aerospace Applications of Artificial Intelligence Conference*, II:74-80.

Kolen, J. F. (1988) Faster learning through a probabilistic approximation algorithm. In *The Proceedings of the Second IEEE International Conference on Neural Networks*, I:449-454.

Uht, A. K., C. D. Polychronopoulos, and J. F. Kolen. (1987) On the combination of hardware and software concurrency extraction methods. In *The Proceedings of the Twentieth Annual Workshop on Microprogramming*, 133 - 141.

FIELDS OF STUDY

Major Field: Computer and Information Sciences, Prof. J. B. Pollack

Specialization: Artificial Intelligence

Minor Field: Theory of Computation, Prof. T. Long

Minor Field: Computational Geometry, Prof. K. Supowit

TABLE OF CONTENTS

LIST OF TABLES

TABLE **PAGE**

LIST OF FIGURES

FIGURE **PAGE**

LIST OF SYMBOLS

SYMBOL	DEFINITION
x_t	Value of scalar x at time t
x^T	Vector or matrix transpose
$X^{(t)}$	Value of vector/matrix X at time t
$X_{\mathtt{I}}^{(t)}$	Value of vector/matrix X at time t indexed by input `I`
η	Scaling parameter for logistic function
ω	Winding number, an IFS transform
Ω	Oscillator frequency, a set of IFS transforms
`R`	A symbol
$\mathtt{R}^*$	A finite, but unbounded, sequence of `R`'s
$\mathtt{R}^\infty$	A infinite sequence of `R`'s
`L|R`	Either `L` or `R`
`LR`	`L` followed by `R`
$f^{\circ n}(x)$	Iterate function f on x n times (i. e. $f^{\circ 3}(x) = f(f(f(x)))$)
λ	Langton's cellular automata complexity measure

CHAPTER I

INTRODUCTION

1.1 Introduction

The problem of understanding the information processing capabilities of neural networks spans many fields. As such, researchers in both AI and cognitive science consider this an important issue. This concern also extends to biologists trying to understand the computational aspects of the brain and engineers copying such processes to silicon. While many researchers have successfully organized neural networks into structures displaying universal computational capability, most have ignored the more daunting endeavor of identifying ongoing computation as it occurs. My thesis addresses this problem as it relates to the understanding of the information processing capabilities of *recurrent neural network models*. Using these models as a cognitive *e. coli*, this dissertation is a step toward understanding the nature of cognition and intelligence.

Over the years, it has become clear that feedforward networks fail to provide the full repertoire of computational abilities essential to the construction of cognitive models. Neural networks, in the eyes of it's detractors, fail to support necessary and sufficient features of computation, such as compositionality and systematicity (Fodor & Pylyshyn, 1988). Although some researchers have displayed the breadth of functional approximation capabilities of these networks (Hornik, Stinchcombe & White, 1989), their responses have ignored the basic complaint. We can trace this monumental lack of computational prowess to the way that feed-forward networks have been applied, namely as transformers of representations. Representation transformers cannot display compositionality and systematicity because there is no way for them to *reuse* the representations they have generated.

Recurrent networks, however, utilize an internal state that is constantly updated according to an input stream. As such, they stand ready to combat claims of computational inadequacy of neural networks as cognitive models (e.g., RAAM (Pollack, 1990) and simple recurrent networks (Elman, 1990)). Recurrent networks appear to posses all the requisite mechanisms to support computation. Many proofs already exist that outline the construction recurrent networks implementing universal computation devices. While theoretically pleasing, no neural network learning mechanism has generated anything comparable to the neural stack and tapes specified by the proofs. The results often resemble simple state machines. As such, the neural network community appears content to interpret recurrent networks merely as deterministic finite state machines (DFSMs). Many researchers maintain this view, despite obvious performance and competence discrepancies, such as the difficulty of inducing trap-states in a recurrent network and the appearance of infinitely complex state spaces. DFSMs and recurrent networks share similar organizations and one could say the DFSMs are a special form of recurrent network utilizing a discrete step activation function that visits a finite number of locations in state space. This interpretation arises because the discrete states and next state mappings approximate the continuous nature of the states and next state mappings.

But why *this* approximation? DFSMs have become entrenched as an engineering tool, one that strongly colors the interpretation of new network architectures. Such interpretations, however, obscure the computational capabilities of recurrent networks, since they have unique natural dynamics that clearly differ from those of DFSMs. We need to find problems that match these dynamics, rather than attempting to squeeze recurrent networks into systems better served by DFSM.

I challenge this state of affairs by asking the following question: how can we *identify* which computation a given recurrent network is performing? I will show that when we examine networks with the finer tools of dynamical systems analysis, they can display very rich and surprising behaviors. Dynamical systems theory suggests that recurrent networks are more closely related to *iterated function systems* than DFSMs. In fact, the properties of IFSs predict many state clustering phenomena seen in recurrent network state analyses (e.g., (Servan-Schreiber, Cleeremans, & McClelland, 1988; Elman, 1990)). Also, we cannot lose track of the fact that automata, the mechanistic substructures of computation, are dynamical systems just like iterated maps and systems governed by differential equations. Limiting the behavioral competencies for recurrent networks to that of finite state machines overconstrains the models, given that we know how to *design* networks to do universal computation.

From the dynamical system vantage, the right level of abstraction for a given system is unclear. Is language recognition behavior of a recurrent network governed by finite set of discrete rules or by an iterated mapping? One could ask similar question regarding intelligence, production systems, and differential equations. To address these questions coherently, I have become interested in the relationship between the act of measurement and observed computational complexity. Specifically, observing the continuous state space trajectory of a recurrent network as a trajectory of discrete states affects the resulting computational explanation of that trajectory by introducing information processing regularities inseparable from those we would want to attribute to the dynamics. The effect is large enough to induce shifts both within and between computational complexity classes such as the Chomsky hierarchy.

I finally argue for new methodologies of design and exploitation that explicitly acknowledge the existence of several sources of complex computational behavior. The original anticonnectionist arguments demanded correspondence between neural networks and a particular form of computation that exhibited the crucial properties of compositionality and systematicity. Other forms of computation exist, however, each with its own set of necessary and sufficient properties. This begs the question: Are there any properties common to *all* forms of computation? I propose that computation arises from the interaction of three contributors: internal dynamics, input modulation of those dynamics, and interpretation (i. e., observation) of those dynamics. The last condition implies that all computation is an emergent phenomenon. This conclusion opens new routes for inquiry. For instance, I will suggest that coupled continuous dynamical systems could explain the grammatical inference known as language learning. While the observed behavior of these systems can be described in terms of grammatical competencies, these competencies are not explicitly implemented or manipulated. Rather the observed competencies, and other computational phenomena, are emergent properties of the system. I will argue that questions like "what kind of computation is going on in dynamical systems such as recurrent networks?", or "what kind of computation is going on in the brain", for that matter, are ill-formed questions.

1.2 Summary

The neural network revolution has been fueled by successes in feed-forward network research. Many tasks exist, however, in that feed-forward networks are clearly inapplicable. For instance, any time-dependent task in that input history affects current decisions demands a mechanism capable of storing and processing representations. The neural network incarnation of this mechanism is the recurrent network. Once we train a recurrent network for a particular task, as happens with feed-forward networks, we must question its epistemic content. In other words, what type of information processing is this recurrent network performing? Since extracting combinatorial logic from a feed-forward network has been fruitful, one would expect similar results from the extraction of computational descriptions from recurrent networks. Many have begun to exploit this approach, specifically by imposing sequential logic interpretations on the dynamics of recurrent networks although we know that recurrent networks can do universal computation (McCulloch & Pitts, 1943; Franklin & Garzon, 1988; Pollack, 1987b; Siegelmann & Sontag, 1991).

This dissertation reports on my explorations of the computational behavior of recurrent networks. I begin my examination of recurrent networks by showing how deterministic finite state machine interpretations of their dynamics can break down. Because of this finding, I suggest that recurrent networks are best described as iterated function systems. This approach provides explanations for two phenomena found in recurrent network behavior, namely *state clustering* and the appearance of *infinite state spaces*. Recurrent networks are not merely iterated transformers of state vectors. The output of the network is an observation of the state dynamics, an observation that contributes its own form of complexity to the behavior of the network. Most recurrent networks also transduce a sequence of input vectors. These sequences serve to induce virtual state transformations in the recurrent network that can select complex behaviors from a set of low complexity transformations. For example, two transformations with fixed point limit behaviors can be composed to form a chaotic generator. Only once these components are understood, we will be able identify the information processing abilities of recurrent networks.

The research presented in this dissertation contributes to our understanding of recurrent networks by identifying the effects of state dynamics, input modulation, and observation. In my attempt to characterize the computational behavior of recurrent networks, I have discovered new interest in a question that most computer scientists take for granted: What is computation and how does it arise in nature? While others argue about the underlying causal nature of computation, my work clearly shows that computation is subjective and emerges from the underlying dynamics through observation.[1] In other words, any computational story is purely descriptive and not necessarily prescriptive of the observed behavior. For instance, the failure modes of the computational description might be quite different from those of the target mechanism.

The subjective stance initially sounds very negative, however, it does shed light on many problems that currently plague AI. The pragmatists will argue that in order to solve the “right” problems we must ignore inconsequential system aspects, like failure modes. It is not enough to succinctly and conveniently define tasks: the planning task, the learning task, the understanding task. The poetic brevity of these definitions may actually contribute to their computational complexity. These apparently extraneous aspects provide a rich source of clues regarding how a system operates. Consider the case of modeling natural language learning as the acquisition and manipulation of explicit grammars. While we could describe language development as a trajectory through grammar space, it does not help us build language learners. The generative approach fails to offer an implementation theory (i.e. what the universal grammar is versus why that particular grammar exists). As an example of the type of theory that I believe that AI and cognitive science should explore, I will describe, in Chapter VII, a method of language learning called *entrainment learning*.

1.3 Preview

Chapter II outlines the historical and social context that spawned this dissertation. Artificial intelligence, as the study of formal models of cognitive abilities, has benefited very little from the neural network revolution. Many “advances” in connectionism often boil down to neural network implementations of existing AI techniques. Analysis of the internal representations generated by feed-forward network applications through clustering or principal component extraction support this position: feed-forward networks merely transform vector representations. Since the popularization of feed-forward networks by the publication of the two volume Parallel Distributed Processing books by McClelland and Rumelhart (1986), many shortcomings have been discovered. Some neuroscientists have attacked the biological implausibility of the models, citing discrepancies between model assumptions and neurological data (Grossberg, 1987). Certain linguists have focused on certain bad applications of the theory and extrapolated to the entire class of connectionist models (Pinker & Prince, 1988). My own research called attention to the lack of task-dependent information processing abstractions (Kolen & Goel, 1992). The most articulate opponents focused on the weak computational power of feed-forward networks (Fodor & Pylyshyn, 1988).

Of the all the criticisms leveled at connectionism, the most substantial is the lack of computational power of the feed-forward networks. One way to respond to these attacks is to turn the networks in on themselves: use connection graphs with cycles. The addition of recurrent connections to networks introduces *information processing state* to the network, much like the self-connections that promote combinatorial circuits to sequential circuits (McCulloch & Pitts, 1943). In Chapter III, I will review several recurrent network architectures, from Elman’s Simple Recurrent Networks to Pollack’s Sequential Cascaded Network. Besides describing the connectivity and processing flow of these models, I will also present various approaches to learning in recurrent networks.

---

1. Specifically, computation causally emerges from a system containing both the object in question and a measurement mechanism. Or as Searle (1992) might phrase it: observation plus emergent1.

Chapter IV will provide a brief introduction to the dynamical system terminology used throughout the remainder of the dissertation. The first section of this chapter reviews such concepts as attractors and repellors, periodic and chaotic regimes, and sensitivity to initial conditions. In particular, the later sections focus on the notions of symbolic dynamics and the theory of iterated function systems (IFSs) (Barnsley, 1988). These ideas will prove to be crucial in the following chapters. Symbol dynamics refers to the collapsing of continuous state space trajectories onto discrete symbol sequences. This characterization dispenses with details of the trajectory and focuses on the gross behavior of the system. The limit behavior of many systems can be described in terms of symbol generators spouting symbol sequences. In periodic behaviors, these generators repeat the same pattern over and over. If the underlying system displays chaos, the generator will have certain stochastic components for constructing aperiodic sequences. Iterated function systems explore the fractal structures resulting from the iteration of unions over affine transforms, much like the Cantor middle third set. An important aspect of these systems is that they can be input driven. For instance, the chaos game takes a probablistically generated sequence of input symbols, one for each transform in an IFS, and uses the sequence to construct an attractor approximation for the IFS. After laying the dynamical system groundwork, I then set out to explain the origins of complex behavior in recurrent networks in Chapters V, VI, and VII.

While several attempts have been made toward understanding the information processing capabilities of recurrent networks (e.g., (Servan-Schreiber, Cleeremans, & McClelland, 1988; Giles et al., 1992a; Watrous & Kuhn, 1992)), I believe that nearly all attempts to date have fallen short of understanding the nature of their processing. Each has been misled by the apparent similarity between recurrent networks and digital sequential circuits. I will argue in Chapter V (and (Kolen, 1994a)) against digital circuit and finite state machine analogies. In their place, I suggest the theory of iterated function systems, an approach that points us to an understanding of the clustering behavior seen across multiple recurrent network models (Kolen, 1994c). It also gives us concrete evidence of infinite state space dynamics in many recurrent networks, despite claims to the contrary (Kolen, 1994b). The main result is that recurrent networks can be thought of as indexed state space transformations that can interact and produce emergent properties.

Many researchers in AI and cognitive science believe that the complexity of a behavioral description reflects the underlying information processing complexity of the mechanism producing the behavior. In Chapter VI, I will explore the foundations of this complexity assumption (Kolen & Pollack, 1993). First, I distinguish between two types of complexity judgments applied to these descriptions and then argue that neither type can be an intrinsic property of the underlying physical system. These two measures are subjective. Changes in the method of observation can radically alter both the number of apparent states and the apparent generative class of a system's behavioral description. From these examples I conclude that the act of measurement can suggest misguiding computational explanations of physical phenomena, recurrent networks, and cognition.

In Chapter VII, I identify three contributors to the emergence of computational behavior: internal dynamics, input sensitivity, and, of course, measurement of state. For years, cognitive scientists have believed that the internal dynamics was either the true source of computationally complex behavior (e.g., (Chomsky, 1965)) or indistinguishable from interaction with a complex environment (e.g., (Simon, 1969)). Input sensitivity refers to the magnitude of environmental effects upon the system’s internal computational dynamics.

Besides internal dynamics and input sensitivity, I claim that the measurement of behavior has an effect equal to these contributors. The final sections of Chapter VII argue that all computation emerges as a subjective property of the time-varying system. As an application of this approach, I will propose an alternative model of language acquisition that views grammatical inference as the emergence of observed computational entities.

The problem of understanding the information processing capabilities of neural networks is an important issue to many people in artificial intelligence, cognitive science, and neurobiology. Some thought the key to justifying connectionist models as a valid approach to cognitive modeling lay in proving their ability to perform universal computation. Despite the results gained by this strategy, it allowed these fields to avoid the more daunting task of identifying ongoing computation as it occurs in nature. I have taken this problem as the central theme of my thesis: understanding the computational abilities of recurrent networks. Because of my explorations, I feel compelled to argue that the subjective nature of discrete observation shapes our computational explanations of physical phenomena independent of those systems. With this stance in mind, this dissertation reports on an important advance in our understanding of the nature of cognition, intelligence, and computation.

CHAPTER II

CONNECTIONISM
AND
FEED-FORWARD NETWORKS

2.1 Introduction

The reappearance of neural network systems as an alternative to traditional artificial intelligence techniques has produced an abundance of claims. Several years have passed since the coming of the currently popular neural network techniques and little has changed since the arrival of Hopfield networks (Hopfield, 1982; 1984), back-propagation (Rumelhart, Hinton & Williams, 1986a), and adaptive resonance theory (e.g. (Grossberg, 1987)). Many connectionist researchers have unknowingly succumbed to the "merely implementational" view (Fodor & Pylyshyn, 1988) by implementing representations and operations believed to be crucial by the AI and cognitive science communities (e.g. BoltzCONS (Touretzky, 1986)). As such, too much effort has been wasted by connectionists in reinventing the symbolic wheels of cognition. Before continuing the development of neural network systems, we must decide up front what role connectionist models will play in the effort to understand cognitive behavior: the models can either serve as explanatory devices or exist as mere implementational entities.

2.2 The Connectionist Revolution

Feldman and Ballard (1982) coined the term *connectionism* to encompass networks of interconnected processing units that locally perform simple operations, like neurons, yet globally produce interesting behavior. What has been recently called the Connectionist Revolution (Minsky & Papert, 1988) is a return to philosophies and approaches predating AI as a discipline, rather than an entirely new research paradigm. Many concepts put forth by James (1890) regarding the association of ideas and thoughts are still circulating in current research. The landmark paper of McCulloch and Pitts (1943), in which they prove that networks of neuron-like processing units can implement logical statements, looks more like a paper one would find in a modern day theoretical computer science journal rather than a journal of mathematical biophysics. Even the term "Hebbian synapse," a phrase common in many recent neural network papers, refers to the work of Donald Hebb in the 1940's (Hebb, 1949). Consequently, historians will likely describe the current flurry of connectionist research as a neural network revival, not a revolution.

Several models have served as rallying points for the revival. Besides coining the name of this emerging field, Feldman and Ballard suggested the use of several building blocks, such as Winner-Takes-All and Conjunctive Connections, for constructing systems of interacting processing units (Feldman & Ballard, 1982). McClelland and Rumelhart exploited multiple layers of perceptron-like units with soft thresholds to bypass some of Minsky and Papert's (1969) theoretical objections to multiple layered perceptrons in their *parallel distributed processing* (PDP) networks (Rumelhart, Hinton & Williams, 1986a). Hopfield introduced spin glass physicists to the neural network community by showing how to build content addressable memories from systems of locally interacting stochastic processes through *constraint optimization* networks (Hopfield, 1982). Finally, Grossberg and Carpenter's work on *adaptive resonance* networks (Carpenter & Grossberg, 1987) stands out due to both its insightful mathematical achievement and breadth of application. Each model has served to both promote the work of the individual research teams and the increase the awareness of the neural network approach as a viable research methodology.

Although the functional capabilities of the models themselves have proven very interesting, learning has played a key role in the rise of the popularity of these systems. In traditional environments, such as lambda calculus or stored program machines, programming is a difficult and time consuming task requiring human expertise. Although many automatic programming (i.e., learning) systems have been proposed, most are *ad hoc* collections of strategies (see (Michalski et al., 1983)). The learning systems built upon these representations are *ad hoc* in the sense that development of learning method was not a direct consequence of choosing the representation. Typical machine learning systems must intelligently manipulate discrete symbolic knowledge structures such as decision trees, boolean formula, frames or scripts. Selecting first-order predicate logic as a knowledge representation, for instance, does not provide any insights in prescribing a corresponding learning method. In fact, most symbolic weak methods apply to a variety of representation schemes.

The new connectionist systems, however, provided a new set of constraints for the machine learning community. Unlike their symbolic brethren, these models required the tuning of many numerical parameters before a system would exhibit the desired functionality, so automatic parameter tuning was necessary. Unlike most symbolic representations, the knowledge structures (the networks of connections) do not *explicitly* implement the knowledge contained within the weights. Parameter tuning for a desired functionality, often described by a training set, was usually performed automatically by a learning mechanism. A training set listed behavioral input/output parings that capture the essence of the desired operation. Because of the mathematical foundations of connectionist models, their learning methods were often derived from the mathematical definition of the model and performance criteria. Back-propagation, for instance, used gradient descent to reduce the sum of squared performance error by taking the derivative of this value with respect to each weight (Rumelhart, Hinton & Williams, 1986b). In other words, the resulting learning, or weight update, rule is *analytically* determined by this approach.

This is not to say that *ad hoc* neural network learning methods do not exist. For instance, one of the first machine learning systems implemented Hebb's famous learning hypothesis (Rochester et al., 1956). The search techniques known collectively as genetic algorithms (Schaffer et al., 1992) approach credit assignment from a statistical direction. The constructive method of cascade correlation (Fahlman & Lebiere, 1990) heuristically recruits new hidden units. Despite this collection of *ad hoc* methods, most network representations afford theoretical constructive analysis leading to learning methods such as back-propagation of error. Traditional symbolic knowledge representations, on the other hand, do not allow such constructive analyses.

2.3 The Broken Promises of the Revolution

Just like their predecessors, these new models and learning mechanisms arrived with an entourage of new promises. Several researchers held that connectionist models paralleled many neurologically plausible operations, although they did not map directly to neurons (McClelland et al., 1986; Rumelhart, Hinton, & McClelland, 1986). Modeling the brain, they contended, would provide valuable insights into how humans process information and how researchers can exploit those rules for artificial systems. These models would exploit parallel processing with large number of simple computational units. The neural networks would tolerate noisy inputs and still generate appropriate responses (also known as graceful degradation) because they would take advantage of distributed representations (Hinton et al., 1986). These systems would exhibit rule-like behavior without explicit symbolic rule interpretation (Rumelhart & McClelland, 1986). Tasks such as pattern completion and decision making would arise from the massively parallel constraint satisfaction driving most neural network mechanisms (Hopfield, 1982). Best of all, these systems would automatically program themselves by observing a few behavioral examples. Once the network learned the training set, they would correctly generalize from the previously encountered examples to unfamiliar situations (Sejnowski & Rosenberg, 1987). These systems could self-organize during learning by recognizing regularities in the training examples without explicit warnings of upcoming regularities (Knapp & Anderson, 1984).

Several years have passed since the arrival of the neural network techniques that are popular today. Despite the excitement surrounding their arrival, little has changed. What about the promises held high by the proponents of neural networks in the late 1980's? While neural networks have been successfully applied to a variety of pattern recognition tasks, they simply do not fare well in tasks requiring cognitive skills. Many experiments and analyses have revealed that neural networks have their own forms of brittleness (such as catastrophic interference (McCloskey & Cohen, 1989)). Generalization has proved to be a problem for connectionist systems, as well as traditional AI systems. In short, connectionist models have added little to the strategies of AI. Though Tesauro's backgammon system (Tesauro & Sejnowski, 1989) was the only machine learning entry to win first place at the First Computer Game Playing Olympiad, Samuel's checker player is still the epitome of machine learning systems. Despite its connectionist origins, Neurogammon merely embodied the crucial philosophies discovered over twenty years ago by Samuel. Towell and Shavlik (1993) use connectionist methods to buttress rule-based expert systems. Shastri (1988) showed us that probabilistic reasoning in semantic networks could be reduced to the connectionist framework. Unwittingly, these researchers played into the hands of their opposition by implementing existing AI mechanisms with connectionist architectures. Some observers of AI and cognitive science (e.g. (Fodor & Pylyshyn, 1988)) have fervently argued that connectionism could only serve as an implementational tool in the task of constructing intelligent agents and understanding intelligent behavior.

Despite the apparent similarities between neural network models and biological neurons, the biologists have yet to be satisfied by the validity of connectionist mechanisms as models of cognitive neurobiological processes. These models, according to most detractors, violate general principles of organization entailed by a long history of experimental findings. This biological complexity stance is indisputable, yet such arguments raise questions of relevance. While modeling the brain will no doubt provide valuable insights into how we process information, it is still unclear how we can exploit those rules for artificial systems. Duplicating every brain dynamic is overkill. This claim is especially true if the details of brain function are not the roots of brain behavior. For instance, the minute details may be washed out by principles that have their own behavioral "chemistry." As a mathematical example, consider the universality of symbol dynamics in unimodal iterated mapping (i.e., a single bump, like the logistic function $\lambda x(1-x)$). As long as the mappings meet some general qualifications[1], the iterated systems based on those mappings share the qualitative behavior, namely the bifurcation structure, despite the quantitative differences between the individual mappings (Feigenbaum, 1983). One could say that the dynamics of the symbol level emerges from the underlying system. Since cognitive behavior may also be an emergent property of the underlying neural dynamics, understanding the emergence of computation in neural network models is important for cognitive science and is the central theme of this dissertation.

Computer Science has not been satisfied with the products of the Connectionist Revolution. The promised panacea of automatic programming by machine learning has turned out to be more like traditional programming. In short, the programming of neural networks involves many explicit and implicit design decisions when setting up the learning environment that directly affects the acquired behavior. As a result, training neural networks is often regarded as a "black art" full of parameter settings, initial weight ranges, update rules, and training set organizations. While this condition may be inevitable, it contributes nothing to reducing the intractability of the learning task.

2.4 Indirectly Programming Neural Networks

It has been shown that, at least in principle, it is possible to design neural networks that can support universal computation (McCulloch & Pitts, 1943; Pollack, 1987b; Franklin & Garzon, 1988, 1991; Siegelmann & Sontag, 1991). Strictly speaking, the design of Turing-Universal neural networks requires constructs such as *pi* units, cycles, etc. that are not available in feed-forward, hidden layer, PDP networks. It is possible, however, to talk about families of feed-forward networks where each network has a different number of input units and has a computable specification (circuit families, e.g. (Balcazar & Gabarro, 1988)). Designing a neural network to compute a given function can be very difficult in practice. Even if we have at our disposal a neural network capable of universal computation, programming it will be at least as difficult as programming the target machine in the proof of universality.

To illustrate the scope of the network loading problem, consider the design of an n input, one output network capable of learning an arbitrary binary function without regards to the architecture of the network, i.e. number and connectivity of hidden units. The focus

1. See Chapter IV for a discussion of the universal properties of unimodal transfer functions.

of this demonstration will be on the flexibility that will be needed by such a device. That is, there should exist a many-to-one mapping from configurations to function. Assuming binary inputs and outputs, a robust network should be able to compute any one of 2^{2^n} functions. Each function capable of residing in the network demands at least one weight configuration. If m is the number of connections in the network, and W is the set of possible weight values for each connection in the network, then there are $|W|^m$ weight configurations, where $|W|$ is the size of W. To cover the entire function space, W must contain on the order of $2^{2^n/m}$ distinguishable weight configurations. Unless, m grows exponentially as n grows, the number of possible weight values gets quickly out of hand. For simplicity, I assume here that each weight configuration specifies a unique function. Some functions, however, may be specified by more than one weight configuration and thus make the situation more demanding.

Obviously, every architecture cannot support all possible binary functions. This is the problem of representational bias that Minsky and Papart (1969) found in single layer perceptrons. While the distinction between linearly separable and nonlinearly separable straddles the single/multi-layered architectural decision, the question of representational adequacy is still open for the case of nonlinearly separable mappings. The result that proves three-layered neural networks to be universal function approximators (Hornik, Stinchcombe & White, 1989) is a finding that is as useful to neural network modelers as the Church-Turing hypothesis is to the AI community. Both describe maximally competent representational mechanisms where additional bells and whistles fail to increase the representational adequacy of the system. Yet, neither provides any prescriptive advice in constructing systems. Without such assistance, the models are useless.

Given this state of affairs, how does one set about designing a neural network? The obvious critical choices one makes in this task are the number of layers, the number of units in each layer, and the connectivity between the layers and the units. Discussion of such decisions comprises much of the art, magic, and folklore of neural network construction. In this section, I will focus on not so obvious design decisions that also affect neural network training, i.e., the network loading problem. The first of these will be task-specific network design in which certain information processing abstractions are added to the network before training. The second indirect programming method involves the selection and refinement of the training set. Finally, how the network engineer sets the performance goals of the network will be shown to have profound effects on the learned behavior of the network.

2.4.1 Task-specific Network Design

I reported earlier on a set of experiments involving networks learning to play tic-tac-toe (Kolen & Goel, 1992). In this work, we were forced to draw two conclusions. First, without the high-level abstractions, our network only learned to mimic the situation-specific responses of its symbolic opponent. It could not generalize to the high-level abstractions needed to win against the opponent. In contrast, given the needed abstractions, the network of Rumelhart *et al.* (1986), appears to have learned the strategy for playing and winning tic-tac-toe. Thus, the content of what is learned by the method of back-propagation strongly depends upon the initial abstractions present in the network. This merely repeated the findings of Rosenblatt (1962) during his studies of perceptrons. The class-C perceptrons had a layer of processing units that extracted a set of useful features from the inputs. These features were hand-tuned for the problem at hand. The output layer, the one with the adjustable weights, was left with the task of finding a set of weights that partitioned the features into the desired output classifications.

Second, these high-level abstractions are a major source of power for learning in PDP networks. Chandrasekaran et al. (1988) suggested elsewhere that the success of PDP networks comes from the "programming" of these abstractions in the networks in a "compiled" form, and the results of the above experiment support this claim. These abstractions provide a domain model to the network, which decomposes the learning space into smaller and simpler spaces, and guides the network in navigating these spaces. In fact, it is these abstractions that form the computational theory (Marr, 1982) of the given information processing task.

Though we may freely talk about internal abstractions, how do we know for sure that the observed abstractions are, in fact, the ones being used? The abstractions must be implemented on a stratum of internal representations. The organization of these representations varies from local to distributed, and biases our representational stories of network behavior. Local representations allow external agents to tag units with semantic labels: the opponent-two-in-a-row-upper-horizontal, for instance. These labels provide the raw materials for constructing explanatory stories of the network's behavior. Unless explicitly built-in (e. g. competitive learning), local representations are a minority in the set of possible representation schemes. Distributed representations, in which many units collaborate in representing a feature, are more common, yet more difficult to observe than their local counterparts. They require principal components and cluster analysis to uncover their structure. It is unclear, however, when to base an account of a network's information processing on one method (cluster analysis or principal components) and not the other.

Shifting one's attention from the internal representations of feed-forward networks to those of recurrent networks only exacerbates this quandary. I will demonstrate in Chapter V the difficulties associated with identifying the abstractions used by recurrent networks. The attribution of internal information processing states based purely on output behavior will be shown to be the only effective means of constructing information processing explanations of recurrent network behavior.

2.4.2 Selection and Refinement of Training Sets

Another way to program the emerging behavior of a neural network is through the selection and refinement of the training set. Selection and refinement can occur in three different ways: representation selection, presentation, and coverage. The problem of representation must be solved first. That is, how well the input and output patterns correspond to the intended function of the neural network. Once a representation has been established, the network designer must decide the method of exemplar presentation. Finally, the designer must determine how much reliance will be placed on generalization: will the training set exhaustively cover all possible cases or will the network see only a subset of such cases? Neglecting any of these training set properties may produce a range of effects including failure to learn a mapping, improper generalization, or misunderstanding of the behavior of the resulting network

Choosing the right representation is very important not only for traditional AI systems, but also neural networks. Since feed-forward networks merely transform representations, selecting a proper representation is crucial to the operation of the network. This selection process may have unexpected effects, as demonstrated by the NETtalk system (Sejnowski & Rosenberg, 1989). After training a feed-forward network on seven character sliding windows of text, this text-to-speech generator was found to use internal representations (i.e. patterns of activation among the hidden units) which clustered into linguistically interesting groups: The network could easily divide between consonants and vowels at the hidden units. After a careful review by Verschure (1991), however, he explained this phenomenon as the selection of output activation patterns. He carried out cluster analysis on the representations for the behavioral responses and found that they had the same shape as the ones reported in the NETtalk paper. This example illustrates the danger lurking behind any extrapolation from behavioral evidence generated by neural networks to the information processing they allegedly perform.

The ordering of the exemplars on which the network is trained can also play an important role in the training of a network. For the back-propagation method, there exist at least two approaches to updating the weights. Either the weights can be updated after each presentation of a new exemplar, or the gradient can be accumulated over many presentations. In the latter case, called batch updating, all the training exemplars are presented before a weight update. When batch updating is used, the training patterns establish a set of fixed constraints that the learning algorithm eventually satisfies with a set of connection strengths. Network training, in this context, can be likened to nonlinear regression for a fixed model.

When the weights are updated more often, either per exemplar or group of exemplars, the ordering of the exemplars can have an impact on training. Cycling through a list of exemplars often leads to oscillations in network weights as the training algorithm alternates between two, or more, good short-term solutions. Such behavior can be prevented by presenting training exemplars in random order; thus providing the necessary symmetry breaking. This update method is most often used for the acquisition of mappings defined by large training sets. It is often prohibitive in terms of resources to do batch updating for very large lists of patterns.

For large networks, it is impractical to present every possible input/output pairing during training. The number of possible patterns grows exponentially with respect to the number of input and output units when considering binary values. Yet, we expect the network to learn the "right" generalization from an impoverished training set. What is the right generalization? From an engineering view, it is the one that does the task that we are interested in. This task, however, may not be representable in the network. In this case, the engineer is out of luck and should try a new architecture.

Even if the task is representable, it may not be reachable given the architecture and training set. The network may converge on weight sets that implement the input/output mapping of the training set, yet perform terribly on the problem as a whole. Theories of generalization cast the learning problem as a weight selection problem (Hertz et al., 1991) and examine the probability of selecting a weight set that both accounts for training set and its possible extensions. This approach identifies training sets that unambiguously (or with high probability of) specifying a single function that will also perform the desired task. These theories typically treat the learning factor as an *a priori* probability over weight configurations and ignore the dynamics of the process. In earlier work, I have shown that the probability density function for network weights after back-propagation training does not necessarily agree with traditional probability density function assumptions (uniform, gaussian, etc.).

2.4.3 Setting Performance Goals

Once the training is underway, the network engineer must explicitly provide a means to stop training. I will call this setting the performance goal of the network. In general, we want to stop training when the network has either "learned" the task or will no longer improve with additional exemplar presentations. Performance goals are intimately related to the representation, or activation patterns, of the target outputs and the context in which the network is to be used. They dictate the criteria used to identify that a network has learned the mapping. The most straightforward way to tell when back-propagation is finished is when the error measure (e. g., sum of squared error over all training vectors) is zero. More often than not, this simple termination criterion fails as a signal that indicates acquisition. If the training set is noisy or "ill-bred," the error may never reach zero because the same input vector maps to more than one output. The network might get stuck in a local minimum or trench because not enough hidden units are provided or the initial conditions were inconvenient. Another possibility is that the proper metric for the output representation space differs from that assumed by the back-propagation derivation.

For instance, a vector of real numbers between zero and one is not of much use if the environment is expecting a binary digit or a symbol. There are several ways to convert the real-valued vectors generated by a feed-forward network to a symbolic form. The most common of these is thresholding, where a real value is interpreted as zero if it falls below a particular threshold value, or one if it is above. Thresholding is useful for strict categorization, i.e., either the input pattern is in the class or it is not. Sometimes there are multiple mutually-exclusive classes from which the network must select the correct class. Here, the network engineer may decide to use a one-in-n encoding for the classes. A winner-takes-all post processor can decide which class the network believes to be the correct assignment (e.g. (Feldman & Ballard, 1982; Sejnowski & Rosenberg, 1989)). This postprocessor finds the unit with the largest activation and interprets it as a one and interprets the rest as zeros. An alternative to winner-take-all is to search for the nearest pattern in the training set. A content-addressable-memory can be inserted between the network and its environment to align the output of the network with the "expected" outputs. In fact, any decomposition from convex to fractal can partition the output vector space into regions of symbolic similarity--each adding their own form of inductive bias.

A parallel issue is that of overtraining, or letting the network modify its weights though it adequately captures the desired input/output mapping. Much has been written about the effects of overtraining. For instance, Chauvin (1989) reported a network, trained for a phonemic classification task, which exhibited performance degradation after prolonged training. Chauvin attributed this increase in error to the limited number of degrees of freedom (weights) in the network. According to many, overtraining produces networks that overfit their training sets and fail to distinguish between noise and the underlying mapping. These networks pass through a series of approximations to the training set during the learning phase. If we assume that the training algorithm will reduce the error measurement, then one would expect the error would be decreasing for each successive network. Unfor-

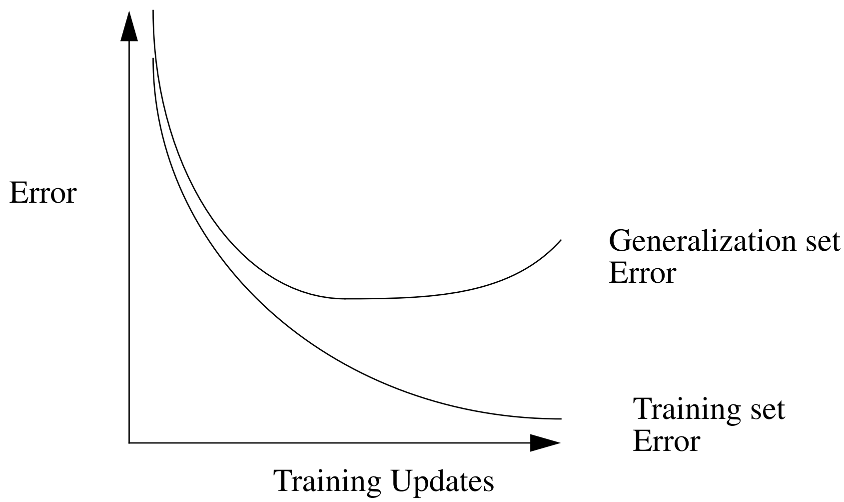


Figure 1: The error-generalization trade-off.

tunately, the error function does not address the issue of good generalization; it penalizes smooth interpolations over bumpy "memorizations" of training exemplars. The results of memorizing are shown in Figure 1. As the learning algorithm continues to reduce the error across the training set, the error with respect to a generalization, or cross-validation, set will continue to decrease until it reaches a minimum. At this point, the generalization error will increase. While there are many reported "fixes" to this problem (e.g., weight decay (Hinton & Sejnowski, 1986; Hinton, 1987; Krogh & Hertz, 1992; MacKay, 1992; Moody, 1992)), they are all task independent. In order to bias the network toward finding a good generalization, one must have enough knowledge of the task to decide what useful generalizations look like.

2.4.4 Summary

This section has addressed the topic of indirect programming in neural networks. In contrast with direct approach of traditional programming languages, these systems involve significant amounts of indirect programming. Although neural network learning algorithms can remove much of the burden from a system designer through automatic acquisition of parameter values, they cannot be expected to carry out the general task of automatic programming. Many explicit and implicit design decisions were shown to affect the behavior of the network after training. For instance, task-specific knowledge can be added to the network by designing in architectural biases that provide the network with abstractions. These abstractions can potentially make the networks' task easier by identifying salient features in the input patters which will be used by later processing. The selection of representations and presentation of training patterns also biases the resulting behavior of the network. The designer can introduce regularities into the hidden layer processing this way. Finally, I dis-

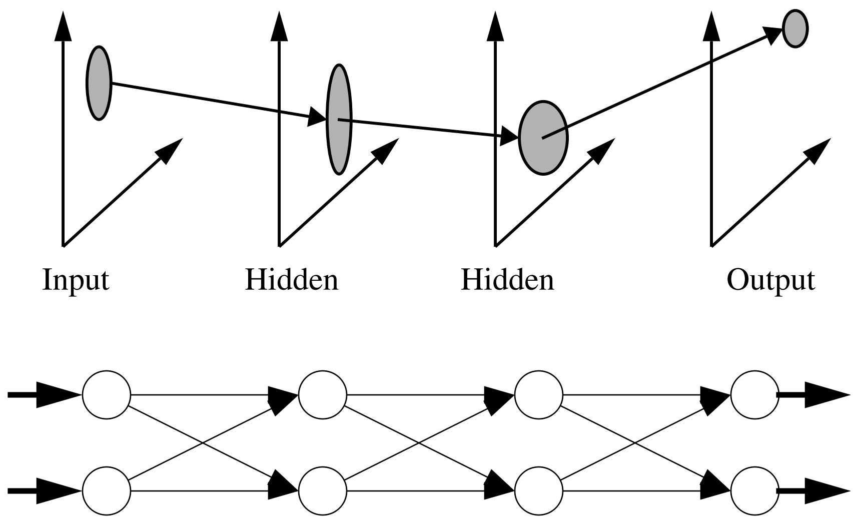


Figure 2: Transformations of input representations in a feed-forward network.

cussed how setting the performance goals and overtraining can affect the training of the network.

2.5 Representations and Transformations

This section reviews some techniques developed to understand the information processing of feed-forward networks. In the seventies and early eighties, many connectionist researchers praised their models' lack of internal representations. Pattern association was performed by direct mappings (Anderson, 1972; Kohonen, 1972; Hopfield, 1982; Hopfield, 1984). These systems lacked the explicit rules applied to explicit data prevalent in most knowledge-based approaches. By 1986, this stance began to erode as the development of learning mechanisms for networks with hidden units (i.e., non-input and non-output units (Hinton & Sejnowski, 1986; Smolensky, 1986; Rumelhart, Hinton & Williams, 1986b)). While they still performed pattern association, the patterns now had to "flow" through the feed-forward network, transformed along the way by each layer of processing units and their associated weights. While not the product of explicit rule-based transforms, the activation patterns had all the philosophical trappings of representations (Hinton et al., 1986).

Because of their structure, we can view feed-forward networks as continuous transformations on continuous representations. This is not surprising, given that these networks consist of multiple layers of processing units performing continuous transforms. To illustrate this property of feed-forward networks, I have drawn four representation spaces in Figure 2: the first one is the space of input activations, the second and third correspond to hidden unit layers, and the final space is range of output activations. Because of the continuous nature of the activation and sigmoidal output functions, these layers preserve neighborhoods. Any set of network inputs landing within the gray spot on the left will come out

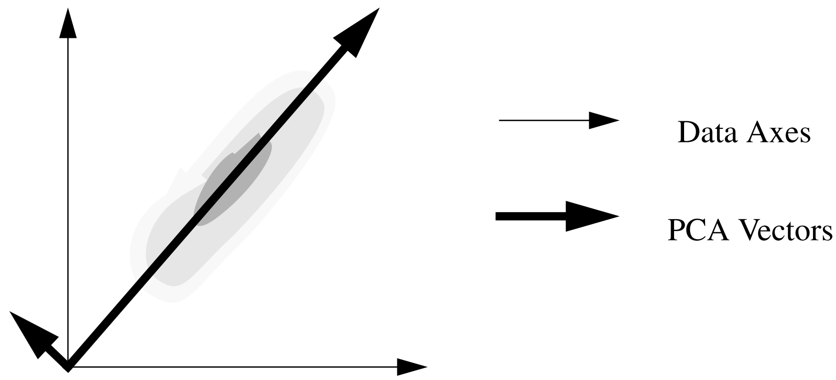


Figure 3: An example of principal components analysis (PCA).

the other end of the network in the gray spot on the right. The continuous nature of the transformation is one reason that neural networks appear insensitive to noisy inputs. If the signal-to-noise ratio is greater than a network specific level for a particular input, the noise will be bounded on the output.

Because of this topological property, certain analytic tools are useful in identifying the information processing carried out by a feed-forward network. Determining the *principal components* of a representation space is one such tool. A principal components analysis of a set of points in a vector space results in a set of M orthogonal vectors that reduce variance to an acceptable level. Figure 3 illustrates two principal component vectors extracted from a set of planar points. The longer of the two principal component vectors maximally reduces the variance of the data set. Another vector (the shorter one) can be included to account for even greater variance. In higher dimensional systems, the principal components are ordered by the amount of variance they account for in the data. Often the data can be explained with just a few vectors. Since the set of principal components vectors is smaller than the dimension of the data, the operation is often thought of as dimensionality reduction.

From a conceptual viewpoint, principal components analysis may identify independent concepts as a reduced-dimension normal basis of the representation space. For instance, the concept “apple” may require discriminations along “redness” and “roundness” dimensions, but discriminations along a “loudness” dimension would not be useful for identification purposes. If the neural network found and exploited these concepts, principal component analysis of the hidden unit representations of various apple and nonapple input can identify the network’s abstraction strategy.

Although orthogonal representations have a certain “localist” flavor to them, the assumption that PDP networks employ distributed representations does not prevent the development of such representations. The orthogonal basis may discriminate mutually exclusive features or compositional features. The basis for a hidden unit representation scheme could be translated, rotated, and scaled with respect to the hidden unit space. For instance, the layer-to-layer transformation may obscure a local representation scheme by rotating its 1-in-n code. The three representation transformations in Figure 2, and their many compositions, are within the grasp of most feed-forward networks with full connectivity between layers of processing units. As stated before, principal components analysis is an excellent tool for discovering these regularities.

We can also use principal components to determine the effective number of hidden units for a given network. Activation flowing through a hidden layer must carry information about the input activation pattern. Sometimes, there are more hidden units than necessary. Consider the case of a feed-forward network whose hidden unit activations lie upon a line for all possible input patterns. Here, we could replace the hidden layer by a single unit whose activation parameterized this line. Weigend (1994) argued empirically that back-propagation and other error minimization techniques provide no incentive for the establishment of orthogonal hidden unit representations. If a network did discover an orthogonal representation scheme, principal components analysis would help to detect it.

By contrast, many neural network learning systems attempt to extract principal components from vector representations of objects. It is interesting that this is a side effect of unsupervised Hebbian learning (Oja, 1982). By interpreting the output of a single unit as a measure of similarity with respect to a given distribution, one can construct a learning rule that performs principal components analysis. The Hebbian rule (increase the weight between units i and j if both are active) must be slightly altered to prevent weights from growing without bound. A simple constraint, weight decay, may be added to ensure that the sum of the weights remains bounded during training. Others have built upon this idea and proposed mechanisms that implement similar ideas (e.g. (Linsker, 1986; Linsker, 1988; Yuille et al., 1989)).

The focussed capabilities of principal components analysis forced the hidden unit analysts to look for other tools. Orthogonality, the strength of principal components analysis, fails to cover the wide variety of possible hidden unit representation schemes. Another statistical method known as *clustering* provides a way of grouping nearby representations. We often interpret these clumps as categories. In the analysis of neural network representations, the hidden unit activations of feed-forward networks are clustered as a way of examining the salient internally generated categories used by the network to perform its desired mapping.

Many researchers have used clustering to explain the emergence of useful hidden unit representations for many connectionist systems. The application of hierarchical clustering to this problem was originally suggested to Sejnowski and Rosenberg by Stephan Hanson (as reported in (Hanson & Burr, 1989)). Clustering of internal representations of the NETtalk system provided evidence of vowel/consonant discrimination in the hidden units (Sejnowski & Rosenberg, 1987). Elman also used this tool to illustrate the internal state clustering in his simple recurrent network (Elman, 1990). The wide acceptance of clustering representations must be some measure of its utility.

Some researchers have even combined the two methods, although for illustrative purposes only. Several researchers (e.g. (Meeden et al., 1993; Crucianu, 1994; Cummins & Port, 1994)) have used principal components analysis for dimensionality reduction. Their work often requires the display of high dimensional hidden unit vectors from various recurrent network models. While it is impossible to display a collection of points from a fifty-dimensional space, the first three principal components of such a set are displayable. These points can then be visually clustered into regions with semantic importance.

2.6 Repeating the Past

With all of the ways to interpret the hidden unit activity of neural networks, it comes as no surprise that their epistemology is unclear. Besides questionable knowledge content, many neural network applications also suffer from a lack of originality. Much of the current neural network research focuses on the implementation of representations and operations already playing a crucial role in AI theories of intelligent action. Shastri (1988) eliminated the semantic interpreter with his connectionist implementation of a semantic network. Touretzky (1986) developed a scheme for representing *tag/data/ptr* triples in an auto-associative memory system called BoltzCONS that could store arbitrary Lisp S-expressions. Pollack (1990) constructed compositional representations in recursive autoassociative memories. Various researchers have implemented production and inference systems (Ballard, 1986; Sun, 1989), parsers (Fanty, 1985; Selman, 1985), variable binding mechanisms (Dolan & Dyer, 1987; Smolensky, 1987; Mani & Shastri, 1991), and schemata systems (Rumelhart, Smolensky, McClelland & Hinton, 1986). This is hardly an exhaustive list; each year produces dozens of similar attempts to implement traditional AI models in connectionist substrates. As the field's most ardent detractors granted connectionist models mere implementation-level status (Fodor & Pylyshyn, 1988), too many researchers rushed to fulfill their prophecy.

The implementational view of neural networks and connectionist modeling is self-defeating. While some may feel compelled to argue from the Turing universality high ground that all we need is one general purpose computational system to study and model cognition, this is the wrong direction to go. We are not at a loss for models that may serve this purpose: first-order logic, SOAR, or even semantic networks. Such a view is narrow minded. The Computer Science field has long recognized the positive influence of a cornucopia of programming languages, each emphasizing different problem solving approaches. Procedural languages like Pascal, Modula, and C, force the programmer to think in terms of sequential operations carried out by machine level primitives. Functional languages based upon the lambda calculus, like Lisp, emphasize processing at a level of functions operating on arbitrary recursive data structures. Stack languages, like Forth and PostScript, force programmers to think about the locality of their data by streamlining stack access. Good programmers will not focus on one or two related languages, but will expose themselves to a variety of language paradigms.

Programming in a single language, or developing theories under a unified approach, is like putting on blinders; you become accustomed to applying the same tools and approaches on every problem. Having a diverse background of language skills allows the programmer to find the right abstractions that will lead to better solutions. C programmers cannot afford to forget recursion and Lisp hackers cannot abandon iteration. The same can be said about symbolic and connectionist approaches to cognition. AI researchers cannot afford to ignore either the symbolic or connectionist paradigms in their quest to build intelligent systems.

Connectionist waste too much effort in reinventing the symbolic wheels of cognition. The connectionist community must decide what place neural networks will have in the process of developing theories of intelligence. As I see it, there are only two choices: the models can serve either as explanatory devices or implementational mechanisms. Why resort to a system of phase-locked oscillators to do dynamic symbol binding (Mani & Shastri, 1991) when a simple assignment operator in C will do the same job? If phase-locked oscillators can perfectly implement the process of symbol binding, then they will have little influence on higher level theories. On the other hand, the implementation may not fulfill the specification and certain properties will "leak" through the levels of implementation (Chandrasekaran & Josephson, 1993). The problem is that AI as a whole suffers from a bad case of inductive bias as it elevates explanation and demotes mere implementation.

When we observe instances of intelligent behavior, we must build descriptions of that behavior that have the right generalizations. But what is the "right" generalization? Occam's razor, an overused and ill-applied philosophical tool, suggests that a minimal description will afford the right generalization (Blumer et al., 1987). This approach breaks down immediately once we realize that there is no objective complexity judgment for descriptions. Consider the plethora of computationally-universal processing models in existence today: Turing machines, lambda calculus, cellular automata, rewrite systems, register machines, ... Each model is reducible through a finite process to each other through a given simulator. Yet, there is no objectively canonical system. While some may argue that Turing machines occupy this preferred position, the elevation of this automaton as representative of all mechanisms embodying effective process is a social issue, not a reflection of the model's superiority over its alternatives.

Inductive bias affects AI in the other direction. Familiarity with a particular model, or world view, a priori defines the world that one is observing. In *Lila* (Pirsig, 1988), the title character argues that her persona is defined by the questions the philosopher asks of her: "I'm whatever your questions turn me into. You don't see that. It's your question that make me who I am."[2] Just as the questions put forth by Phædrus shaped Lila, so do the questions AI researchers ask about intelligent behavior. Preconceptions of learning, planning, understanding, and other "intelligent" behaviors can blind us from understanding the underlying mechanisms of truly intelligent action. Like Phædrus, AI researchers fail to realize the effects of preconceptions on their objects of study. A similar blindspot hindered the development of powered flight as scientists and inventors focused on the flapping of the bird's wings and ignored the issues of aerodynamics and control (Pollack, 1993).

The commonality of the types of difficulties encountered by both connectionist systems and symbolic systems may not be coincidence. The problem might not lie in the models, but in the *applications* of the models. As I have pointed out, many connectionist researchers are using their models in exactly the same ways as their symbolic counterparts. Since the universality of different mechanisms is not the issue, connectionists should focus their research energies on discovering and enumerating what neural networks do naturally. Only then should exploitation begin.

---

2. Page 220.

CHAPTER III

RECURRENT NETWORKS

3.1 Introduction

I argued in the last chapter that feed-forward networks merely transform representations (Figure 2 on page 16) and that the real power of PDP networks comes from selecting vector representations that embody the desired topological relationships. The success of an application based upon feed-forward neural networks rests upon the designer knowing these constraints ahead of time. The designer, for instance, should select representation vectors that ensure that all "noun" vectors should occupy a small region of word vector space. This constraint, and others like it, has guided researchers in constructing representations and is one of many programming aspects of neural networks. The problems to which we apply feed-forward networks have one constraint in common. These tasks are temporally independent: the "what" of current input unambiguously determines the current output independent of "when" it occurs. Unfortunately, many problems in AI and Cognitive Science are context dependent and thus demand that neural network architectures encode, store, and process context for later use.

A class of neural networks known as recurrent networks is often brought to bear in these situations. In recurrent networks, the current activation of the network can depend upon input history of the system and not just the current input. These networks have the potential to dynamically encode, store, and retrieve information. This chapter will provide a review of recurrent network architectures and their various learning techniques. While many applications have been built that exploit recurrent networks, there is still no clear understanding of how they process information. The next chapter defines the terminology that will be necessary to understand their behavior, namely the theory of dynamical systems. While some researchers have put forth explanations of recurrent network operation in terms of finite state machines, a simple demonstration in Chapter V will show how such descriptions can fail to capture the potential information processing power of a recurrent neural network.

3.2 State Dependent Problems

Before introducing the architectures, it will be constructive to consider the types of tasks to which they will be applied. Recurrent networks are usually applied to time dependent problems. Our world is full of time dependent processes. The daily cycle of light and dark, the

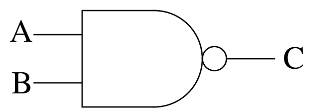
Table 1:A finite state automata transition table for a 35-cent vending machine. (A/B means go to state A after releasing B cents in the coin return)

State	Nickel	Dime	Quarter	Select
0	5	10	25	0
5	10	15	30	5
10	15	20	35	10
15	20	25	35/5	15
20	25	30	35/10	20
25	30	35	35/15	25
30	35	35/5	35/20	30
35	35/5	35/10	35/25	0

changing of seasons, the periodic beats of our hearts, are all examples of systems that change over time. Almost any real system has some notion of temporal changes, although we often do not think of them in that way. Consider a light switch. Although there is minuscule delay between closing the switch and the light turning on, we can ignore the signal propagation delay due to the speed of light and consider the system to be time independent. These simplifications allow us to reason about complex systems and devices without bogging down in detail.

There are systems in which the time dependent behavior is clearly important, however. A vending machine must release both your can of pop and your change after you deposit enough coins. Since the coin slot usually prevents simultaneous depositing of multiple coins, the machine must remember how much money had been deposited since the last sale. The behavior is context dependent since the machine will react differently to a coin depending upon the previous coinage. In this situation, a simple finite state automata can store the current total coinage (Table 1). This pop machine has eight distinct states, one for each possible increment of five cents it may receive. The representation of the coinage is the state of the finite state machine. It will accept nickels, dimes, and quarters as “input” until the total is thirty-five cents. If more than thirty-five cents has been fed to the machine, it will automatically release enough coinage to maintain the correct amount of money for a pop. After entering this state, the user selects a pop (via the Select input), the beverage is released, and the state of the machine returns to its initial state.

The light switch and vending machine represent two disjoint classes of machines defined by the time dependence of their behavior. In digital circuit theory, this distinction also underlies the division between combinatorial and sequential circuits. In combinatorial circuits, the current input totally determines the behavior of the mechanism. A NAND gate (Figure 4), for instance, should respond to only to the voltage levels present at response time. Sequential circuits, on the other hand, are sensitive to the temporal history of their inputs. The prime example of a sequential circuit is the fundamental memory unit, the flip flop (Figure 5). To meet its specification, the flip flop contains an internal state that changes



A	B	C
0	0	1
0	1	1
1	0	1
1	1	0

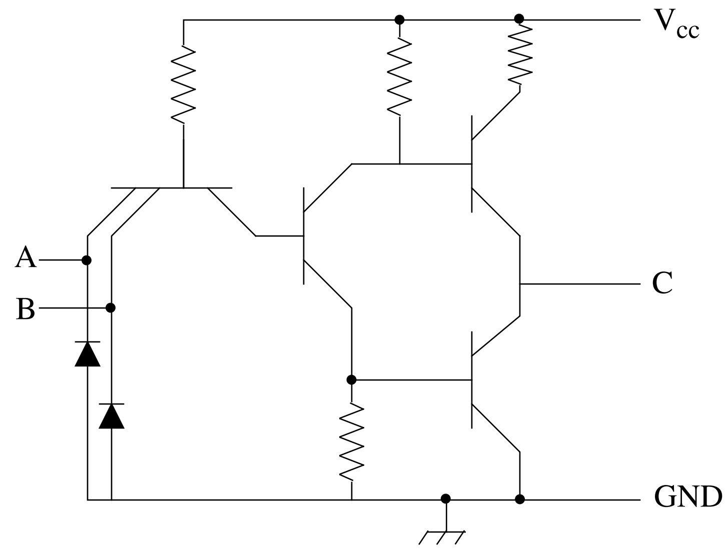


Figure 4: A NAND gate and an underlying integrated circuit implementation (schematic from (Texas Instruments, 1985)).

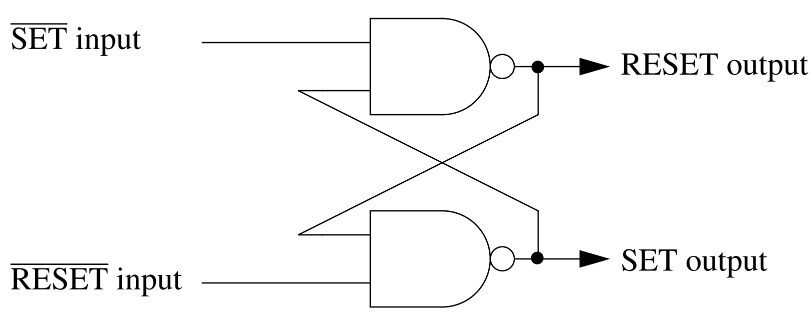


Figure 5: A simple $\overline{\text{S}}$-$\overline{\text{R}}$ flip flop built up from two NAND gates.

according to the inputs it receives. Internal state allows input signals to affect future output behavior.

This definition, unfortunately, is not broad enough; one could argue that the physical implementation of an NAND gate has internal state. The electric signals cannot travel faster than the speed of light, so any output behavior must be in the future of the input event. The same can be said for multilayered feed-forward neural networks--activations of the hidden units constitute the internal state of the network. If one looks at the behavior in the limit (i.e., the action of the device if the inputs were to remain constant for a very long time) we do see a difference. The NAND gate will experience short-lived transients and will settle upon a stable output. The network will also produce its own transients due to the pipelined nature of its processing, but once the input exits the pipeline, the network will generate the same behavior over time. Usually the transients are considered a side effect of the implementation, and not considered part of the functional specification. Those systems with very short transients (compared with the time course of interest) can be considered to lack a temporal component like the NAND gate. In other words, the schematic on the rights side of Figure 4 has temporal dependence, while the logic symbol on the left side does not. Chapters IV and V will discuss several systems in which transients do play an important role in their functional specification.

Sequential circuits fall into two architectural categories named after the engineers who pioneered them. Mealy and Moore machines are sequential systems that differ on the type of output function they employ (Figure 6). In a Mealy machine, the output at time t depends upon the state at time t and on the input at time t. The output of a Moore machine at time t, on the other hand, depends only upon the state at time t. This does not imply that the output is independent of the input sequence, but that any influence must flow through the state. While the set of Moore machines is clearly included in the class of Mealy machines, a simple proof can be given to show that any Mealy machine has a Moore machine equivalent. Thus, the two formalisms express the same functionality. We can extend this equivalency to regular grammars. If the output symbol is interpreted as a signal indicating the grammaticality/nongrammaticality of the preceding input string, we can use the Mealy and Moore machines to recognize regular languages.

3.3 Recurrent Networks

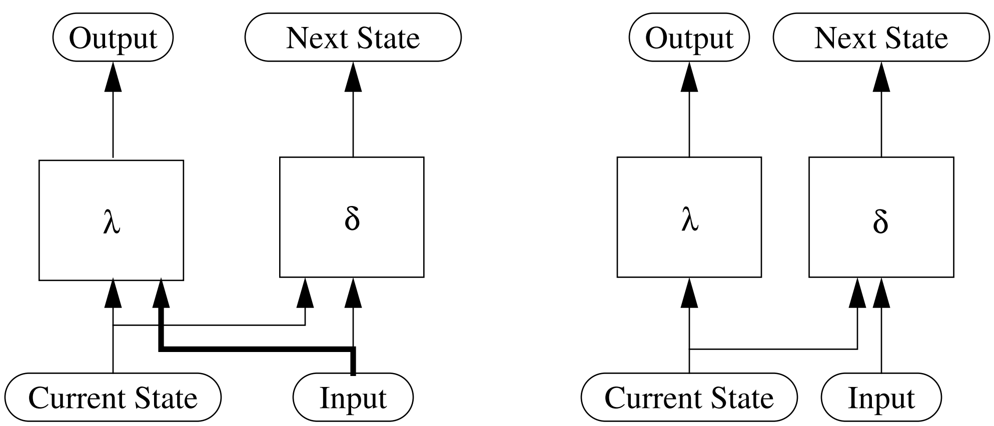
Feed-forward networks share many properties with combinatorial digital circuits. Both combinatorial circuits and feed-forward networks display directed acyclic connectivity graphs. The signal propagation time in both systems is always a constant proportional to the depth of the circuit. The fundamental discrete logic gates, such as AND, OR, NOT, and NAND, are implementable as linear combinations subjected to a high-gain transfer function[1]. Such similarities have prompted some researchers to extract logical circuit approximations to feed-forward networks.

On the other hand, we can think of recurrent networks as neural state machines much like sequential digital circuits. The recurrent connectivity of the network produces cycles of activation within the network. This connectivity allows the network to have a short-term memory of previous experiences, as these experiences may have some effect on the cycling activation and may later affect the processing of the network well after the stimuli have passed. This idea, known as reverberation, can be traced to the work of McCulloch and Pitts (1943) and is also the underlying idea behind the flip-flop. The memory compo-

1. Either a threshold or a function like $g(x) = \tanh ax$ when a is very large.

$M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$
Set of states Q
Start state q_0
Input alphabet Σ
Output alphabet Δ
Transition function
$\delta : Q \times \Sigma \rightarrow Q$
Output Function
$\lambda : Q \times \Sigma \rightarrow \Delta$

$M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$
Set of states Q
Start state q_0
Input alphabet Σ
Output alphabet Δ
Transition function
$\delta : Q \times \Sigma \rightarrow Q$
Output Function
$\lambda : Q \rightarrow \Delta$



Mealy Machine Moore Machine

Figure 6: Mealy and Moore machines

nent of recurrent networks suggests that they would serve as excellent candidates for solving problems involving temporal processing.

I devote the remainder of this section to describing the details of several recurrent network architectures. The cyclic vs. acyclic connectivity distinction separates recurrent networks from feedforward networks. These connections allow activation to cycle, or reverberate (McCulloch & Pitts, 1943), through the network. The first section will describe several full-connected networks where every processing unit is connected to every other unit. These networks could theoretically support any possible recurrent network configuration by setting any unneeded connection to zero. Most neural network designers, however, do not wish to wait for learning algorithms to prune away the unnecessary resources when they already have an idea of the proper processing organization. They bias their networks with an *a priori* connectivity structure. These organizations are generally inspired by Mealy and Moore machines from digital sequential circuit theory. Besides the traditional recurrent networks, I will also discuss a recursive application of a feed-forward network that constructs compositional vector representations. While this mechanism is not strictly a recurrent network, the analysis tools developed in Chapter V also apply to the study of its operation.

3.3.1 Fully-Connected Networks

An early use of a recurrent fully-connected network can be found in the work of Anderson (1977). Called Brain State in a Box (BSB), they used the system to model psychological effects seen in probability learning. Anderson selected an activation function that was zero for net inputs less than zero, one if the net input was greater than one, otherwise it was equal to the net output (Equation 1).

$$S^{(t+1)} = g\left(W \cdot \begin{bmatrix} S^{(t)} \\ 1 \end{bmatrix}\right) \qquad g(x) = \begin{cases} 0 & x < 0 \\ x & 0 < x < 1 \\ 1 & 1 < x \end{cases} \tag{Eqn 1}$$

Activations of the processing units, $S^{(t)}$, were updated synchronously. The activation of the network meanders within the interior of the activation space driven by the weight matrix, W, and is prevented from growing too large by the activation function, g. Incremental learning, based upon Equation 2, determines the direction to update the weights after each training pattern. This technique relies on auto-association to create basins of attraction in activation space. Thus, an input-output mapping can be implemented by clamping the "input" units to some fixed values and letting the rest of the units seek a harmonious set of activations. The learning parameter η modulates the outer product of the error and current activation of the network, a. The prediction error is the difference between the training output, t, and the stabilized activation, a.

$$\Delta W = \eta\,(t - a)\,a^T \tag{Eqn 2}$$

While BSB exhibits interesting behaviors, by far the most popular fully-connected network is the architecture advanced by Hopfield in the early eighties. While each processing unit is connected to every other unit in the network, there are no self connections. Each connection has a weighting associated with it, which assists in calculating the current state of the unit. The activity of a unit is measured by the inner product of the incoming outputs from other units and weights along those connections. The output of a unit is mapped through a nonlinear function, such as a linear threshold, hyperbolic tangent, or a logistic sigmoid. These activations were updated either asynchronously for discrete outputs from the linear threshold, or continuously (i. e., integrated) for the hyperbolic tangent case. While the network weights play a definitive role in determining the behavior of the network, after training one could view the connection weights as reflecting the statistical relationships between the firing behavior of the two units in the presence of the target patterns. If the unit's behavior is positively (negatively) correlated (i.e. the signs of the states are the same, the weight will be large and positive (negative)). If the units' behavior is not correlated, the weights will be near zero. The processing units update their state according to the weighted sum of the states of their neighboring units.

Content addressable memory, according to Hopfield (1982), could be viewed as minimization of an energy function where memories corresponded to local minima in the energy spaces. Hopfield's initial model was a network of fully-interconnected processing units whose output was computed using a linear threshold. To ensure that the network would behave as a content addressable memory, Hopfield ensured the satisfaction of two important constraints. First, the weight matrix of the system was symmetric and had a zero diagonal. Second, processing units were updated asynchronously with a fixed probability. Symmetry square ensures that a matrix is positive definite. A positive definite matrix has positive or zero eigenvalues that indicate that only point attractors will emerge in the limit behavior during asynchronous updates. These point attractors, in theory, correspond to the content addressable memories. If a particular memory pattern is corrupted by noise, the dynamics of the network would flip bits until the correct pattern stabilized.

The memorized patterns had to be learned by, or loaded in to, the network weights. Unlike many other neural network learning techniques, the Hopfield network relies on single shot learning. The learning algorithm sums the auto-correlations for each pattern. These auto-correlations become the weight matrix. Entries in the weight matrix are given by Equation 3, where x_{pi} is the i^{th} element of the p^{th} pattern. The weight matrix, along with the activation function of the processing units, specifies a system for minimizing the energy function in Equation 4.

$$w_{ij} = \sum_p (2x_{pi} - 1)\ (2x_{pj} - 1) \qquad \text{(Eqn 3)}$$

$$E = \frac{1}{2}\sum_p \sum_i \sum_j w_{ij} o_{pij} \qquad \text{(Eqn 4)}$$

Hopfield empirically observed the capacity to be approximately 0.15N orthogonal patterns, where N is the number of units in the network. Theoretical work by others suggests that the number of uncorrelated patterns that can be stored in a network be proportional to the mean number of synapses per neuron (Amit, 1989).

Later, Hopfield developed a continuous version (Hopfield, 1984). The new model used a sigmoid as the activation function for the processing units and units were updated continuously according to the differential equation described in Equation 5.

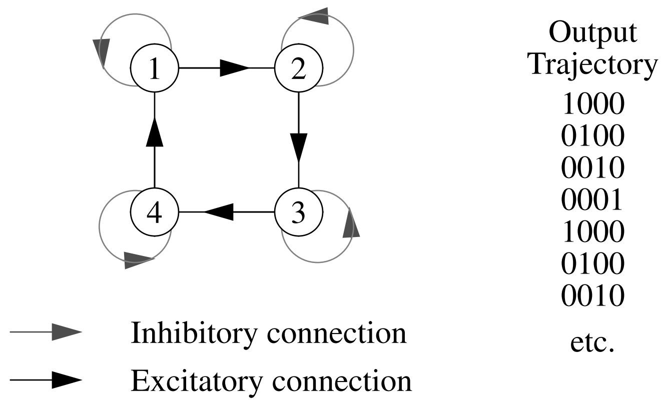


Figure 7: A Hopfield/BSB-like mechanism with asymmetrical weights for producing a period-four trajectory.

$$\tau_i \frac{dV_i}{dt} = -V_i + g\left(\sum_j w_{ij} V_j\right) \qquad \text{(Eqn 5)}$$

Hopfield also showed that the stable extreme states (states where all of the processing units had activations of zero or one) were identical to the discrete version with the same weight matrix. This formulation of the content addressable memory systems appears more amenable to hardware implementations in that the network could be built with operational-amplifiers for processing units and a resistor-capacitor array for the weight matrix. As such, many hardware implementations, electronic, optoelectronic, and all-optical, exist for this model. Graf et al. (1986), for instance, reported constructing custom CMOS integrated circuits with 256 fully connected nodes. Farhat et al. (1985) implemented a Hopfield network of 32 nodes using LEDs and photodiodes for the nodes and an optical mask for the weight matrix. For all-optical implementations, some nonlinear medium or device must be found to perform the node transfer function, such as strongly pumped phase-conjugate mirrors (Soffer et al., 1986).

One problem with Hopfield's model is the symmetry of the connections. Synapses, the biological equivalent to connections, are not even bidirectional. If a model is going to claim the emergent properties of real neurons, it should capture the observation that neural connections can be antisymmetric and unidirectional. Many researchers have studied networks where these implausible constraints are relaxed, networks with antisymmetric weights and self-feedback. Breaking the symmetry assumption is critical if one is interested in temporal sequence generation (e.g. (Amit, 1989)). Antisymmetric connections allow the network to produce very limit cycles, quasiperiodicity, and chaos.

3.3.2 Jordan Networks

While the BSB and Hopfield networks display time dependent behavior, most applications of these systems did not take advantage of this feature beyond pattern completion (e.g. (Hopfield, 1982)) or constraint satisfaction (e.g. (Hopfield & Tank, 1985)). Some systems

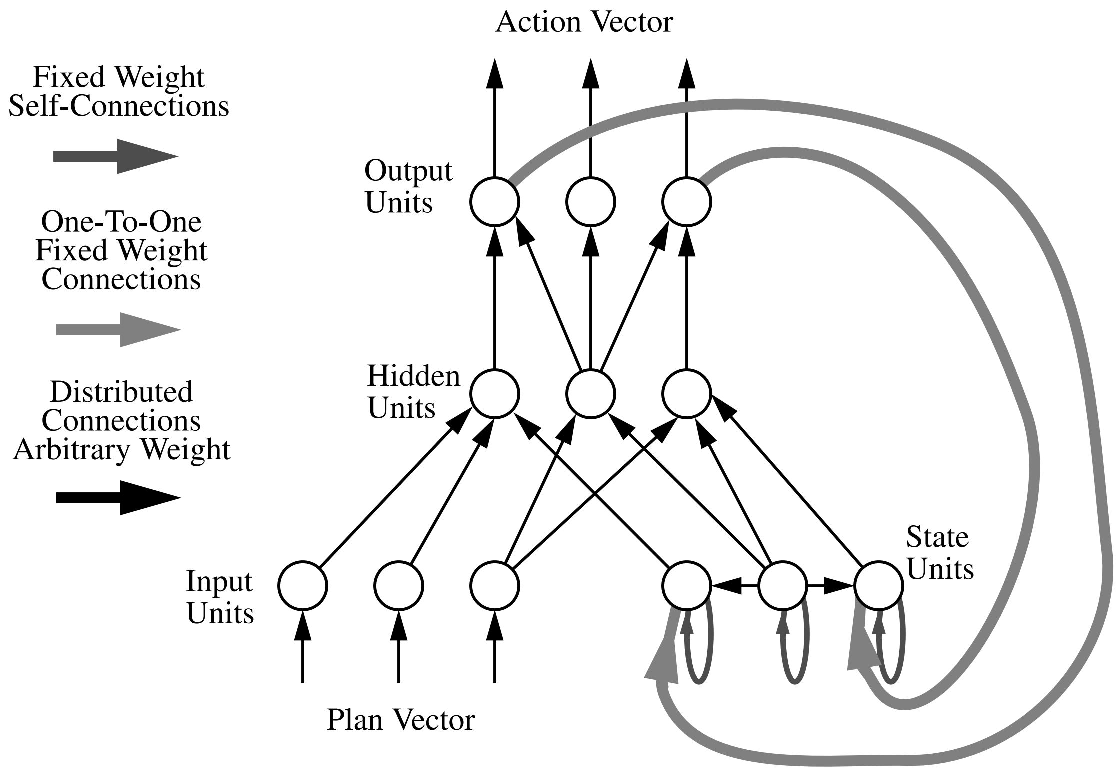


Figure 8: Jordan's recurrent network.

could generate trajectories, but they often simplified this generation by using local representations of states. For example, a four-step periodic pattern would require the use of four units connected as in Figure 7. The excitatory connections (dark arrows) increase the activation of the next node in the cycle. Once a node turns on, the inhibitory self-connection will cause it to turn off. The result is a single pulse of activity cycling around the network. As stated earlier, we are interested in systems displaying temporal variation and temporal context dependance. One member of the PDP group, Michael Jordan, developed a network model capable of such behavior (Jordan, 1986).

To encode several sequences into a single system, Jordan utilized the concept of plan vectors. A plan vector "selects" the sequence the network should be generating. The network associates an input pattern ("the plan") with a sequence of output activations ("the actions"). Plan vectors differ from traditional AI plans involving frames (Minsky, 1975) or scripts (Schank & Abelson, 1973). It is merely a vector of activations that were present during training of the output sequence. Jordan assumed that the "plan" would remain constant as the network followed its trajectory. In other words, the activation pattern of the input would select one of many output trajectories stored within the network. Since motor control could be viewed as a form of trajectory selection, this network served as useful role in experimenting with such theories. This type of network model differed from traditional views of motor control in that it emphasized that the processor did not store and retrieve output vector sequences in a linked list or any other abstract data structure. Rather, the trajectories were computed at run-time as the result of a dynamic process.

The units calculating these trajectories could be classified into four types: plan units, state units, hidden units, and output units. The hidden and output units calculated a weighted sum of the incoming activations. Then, a sigmoidal activation function is applied to the net input to produce an output bounded within a fixed range. The state units differed from these units in that they possessed inputs connected to themselves and other units within the state layer in addition to the standard connections to the output units. Connections between the output units and state units have a weight of one, providing a way to copy activations from the output units to the state units. In addition, each state unit is provided self-feedback through a fixed weight connection between the output and input of a unit. The self-connections are weighed by a recency constant, μ, which helps the units in calculating exponentially weighted sum of past outputs. This connection allows each state unit to construct a limited state history by exponential decay of previous states. The intralayer connections within the state units have strengths of 1 also. The state function must provide enough temporal context to avoid ambiguities, however. These ambiguities can occur in repeated subsequences and common subsequences between plans. Jordan placed no restriction on the output function that mapped the recurrent state to the output vector. The four types of units, plan, state, hidden and output, each implement important facets of the Jordan network.

The networks he prescribed were closely related to Mealy machines in that the output of the network was fed back as input. Training Jordan networks is similar to training a feed-forward network: the recurrent state is treated as just another input. Because of this assumption, methods like back-propagation were readily applied to the search for weight sets capable of generating desired trajectories. While some tasks are fully specified, like maintaining an oscillation, other tasks are not so fully defined. Take, for instance, the task of specifying all the moving pieces of the vocal tract during an utterance. There are many situations when placement is very critical, while other times there is little or no effect. In order to explore systems like this, Jordan introduced the concept of training with constraints. Rather than prespecifying the entire mapping the network is to implement, one can specify don't-care values when the output of a particular unit is irrelevant. Other constraints (Jordan, 1986) included inequalities ($o_a < c$), ranges composed of inequalities ($c_1 < o_a < c_2$), linear constraints ($c_1 o_a + c_2 o_b = c_3$), and nonlinear constraints (such as $c_1 o_a^2 + c_2 o_b^2 = c_3$). Using constraints, rather than specific values, was useful to model sequential behaviors like coarticulation in speech.

3.3.3 Simple Recurrent Networks

Another approach, developed by Elman (1990), borrows its structure from Moore machines. Rather than use the outputs of the network as input to the network, the activations of the hidden units are fed back as inputs. Context units were initially set to activations of 0.5. In the forward pass through the network, the hidden layer linearly combines the activations of the input and context units. The output of the network is then calculated by the output layer from the activations of the hidden layer. Next, the context units' activations are updated according to the one-to-one connections from the hidden unit layer. These connections have a weight of one. Mathematically, the SRN dynamics reduce to the iterated mapping of Equation 6.

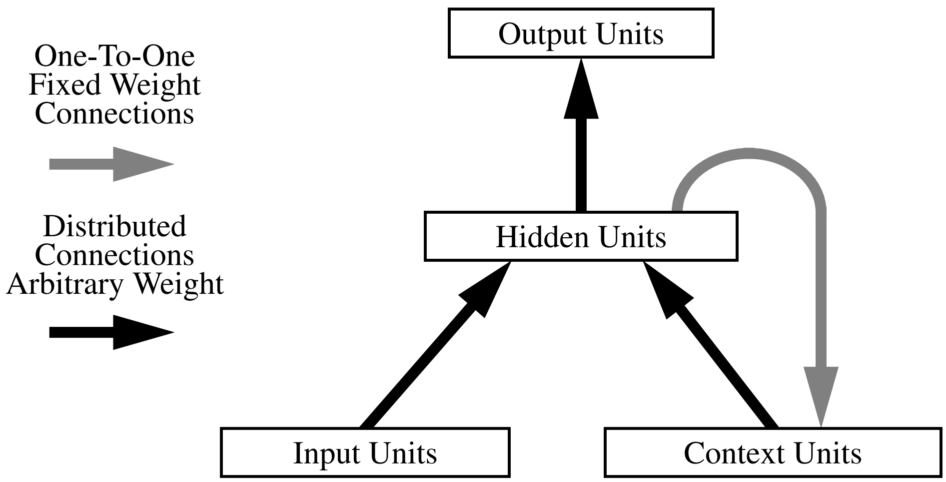


Figure 9: Elman's simple recurrent network (SRN).

$$S^{(t+1)} = g(W \cdot \begin{bmatrix} S^{(t)} \\ I^{(t)} \\ 1 \end{bmatrix}) \qquad O^{(t)} = g(V \cdot \begin{bmatrix} S^{(t+1)} \\ 1 \end{bmatrix}) \qquad \text{(Eqn 6)}$$

These networks, like Jordan's, were trained as if they were feed-forward networks and no error signals were back propagated through the recurrent connections. While Jordan-like networks have appeared in a variety of control applications, SRNs have been often applied to the problem of symbolic sequence prediction. As a sequence of input vectors is fed to an SRN, we can interpret the outputs of the SRN as expectations of the next symbol in the sequence. The networks trained under this paradigm rarely achieved a perfect level of performance. It is difficult for the system to report multiple predictions, i.e. if *abc* and *abd* are subsequences in the training set, the network must have difficulty reporting that after seeing *ab*, either *c* or *d* will appear next. (Unless, of course, the context preceding *ab* is enough to disambiguate the decision.) Rather than abandon his mechanism, Elman points to the RMS error of the network at interesting boundaries, like word boundaries in a letter sequence. Before these boundaries, the network will adequately predict next symbols because enough discriminating inputs have been collected to make a reasonable determination of the ending of words. Beginning of words, on the other hand, are very ambiguous and the networks react to the ambiguity by producing high RMS errors for ambiguous regions of input sequences.

While Elman trained his networks as if they were feed-forward networks, others explored the utility of using the full gradient in recurrent network learning systems. An early discussion appeared within the Rumelhart et al. (1986a) exposition of back-propagation. In this paper, they suggested that recurrent networks could be trained by "unrolling the loop" and viewing the network at time t as a t-layered network in which each layer is a copy of the recurrent network. The error signal could then be back-propagated using the same equations defined earlier in the paper. There was at least one significant problem with this approach. In order to properly unroll the loop, every input pattern must be stored off-line to reconstruct the unrolled network. Some considered this an extravagant waste of resources and thus resorted to truncated gradients that calculated one Δw for each weight, as opposed to all the Δw's that must be accounted for in the full gradient (Pollack, 1991).

Fortunately, there was another way of calculating the full gradient. Williams and Zipser (1989) discovered a way to express the gradient as a recurrence relation. This relation can be calculated during runtime and used whenever corrective feedback is provided. Thus, the learning system need not maintain a history of previous input events as they have been incorporated in the accumulated partial derivative. The network updates the table of partial derivatives on every input. Let $S^{(t)}$ be the current state and $I^{(t)}$ be the current input. Given the error, $E^{(t)}$, on the output units, the partial derivatives of the weights can be found by using Equations 7 and 8.

$$\frac{\partial S_i^{(t)}}{\partial w_{pq}} = g'(h_i^{(t)})\left[\delta_{ip}S_q^{(t-1)} + \sum_j w_{ij}\frac{\partial S_j^{(t-1)}}{\partial w_{pq}}\right] \qquad \text{(Eqn 7)}$$

$$\Delta w_{pq}^{(t)} = -\eta\frac{\partial E^{(t)}}{\partial w_{pq}} = \eta\sum_k E_k^{(t)}\frac{\partial S_k^{(t)}}{\partial w_{pq}} \qquad \text{(Eqn 8)}$$

The indices p and q determine the row and column of an element from the weight matrix w. Similarly, the index i determines the element of the state vector S. The vector h is the inner product of the weight matrix and the state vector. The back-propagated error, δ, and state vector combine multiplicatively to produce the current error's contribution to the gradient. The summation implements a feedforward propagation of accumulated error.

The full gradient does not come free: A space penalty is accrued by this method. Beyond the weight change scalar, the learning system must also store the partial derivative vector, $\frac{\partial S^{(t)}}{\partial w_{pq}}$, for each weight for a storage requirement of $|S||W|$. Equation 7 shows how to update this value after each iteration of the network.

3.3.4 Recursive Auto Associative Memories

Neural networks as a representation and processing models on their own have weathered many attacks from well-meaning philosophers. In particular, Fodor and Pylyshyn (1988) and Pinker and Prince (1988) simultaneously raised questions concerning the observations of and conclusions drawn by Rumelhart and McClelland regarding their feed-forward network model of past tense acquisition (Rumelhart & McClelland, 1986). While Pinker and Prince argued against plausibility of a language acquisition theory based upon this model, Fodor and Pylyshyn directly confronted the connectionist models' apparent lack of representational adequacy. Specifically, they point out that neural networks lack both compositional syntax and semantics. Without these two properties, Fodor and Pylyshyn argued, no model of cognition could account for both the compositionality and the systematicity of thought. Compositionality refers to humans' ability to think novel thoughts composed of existing ones. Without compositionality, we are forced to presume that concepts like "jet engine" must have been hard wired into the human cognitive system. The other feature, systematicity, refers to the fact that because a cognitive agent can think particular thoughts, it must have the potential to think others. A classic example of this phenomenon refers to the co-occurrence of sentences such as "John loves Mary" and "Mary loves John." The principle of systematicity says that an agent capable of thinking the first sentence is also capable of thinking the second. At the time, Fodor and Pylyshyn claimed that the only way a connectionist system could possibly exhibit both compositionality and systematicity would be as an implementational layer of a larger, traditional symbolic system.

Fodor and Pylyshyn failed to realize that there was more to compositionality than what occurs in traditional models of computation. In these vehicles, compositionality is concatative--the representation grows by concatenating smaller representational pieces into larger syntactic entities as happens with Turing machine tapes, parse trees, and semantic networks. This method is a specific case of a more general notion of compositionality in that only the *functionality* of composition is maintained (Gelder, 1990). Under this broader definition, any system is functionally compositional if it can combine representations into a new representation and can later extract the original representations from the composed one. This property is independent of the method of composition and can be implemented in a variety of ways, as I will show below.

The recursive auto-associative memory (RAAM) provides a solution to both the problems of compositionality and systematicity (Pollack, 1990). The RAAM network, inspired by the behavior of three-layer encoder networks, implements functional compositionality and was . Encoder networks autoassociate[2] patterns through a smaller, intermediate representation (Figure 10). The intermediate representation is thought of as a condensed representation for the input vector. For instance, the inputs and outputs could hold three letter patterns. In Figure 10 three registers hold the letters "C", "A", "T". These patterns are then auto-associated during training. The distributed representation viewed at the bottleneck of the network could then be used as a condensed representation for "CAT". The trick then is to realize that the autoassociative memory is really two devices.

Conceptually, both the encoder network and RAAM network can be divided into two parts, the encoder and the decoder. The encoder combines two or more representations into a single representation so that the decoder can later reconstruct the original constituents from the composed representation. The powerful aspect of the network is the reuse of the internal representation generated at the hidden layer. The activations of the hidden layer can recursively encode complex data structures in trees. The weights between the input layer and hidden unit comprise the encoder. They reduce the dimensionality of the input vector from kn to n, where k is the number of fields and n is the number of units in the hidden layer. The rest of the network implements the decoder, which transforms the reduced representation to the larger vector. Vectors for each field can be reused and decoded themselves (Figure 11).

The RAAM network is not a standalone device, unlike the other recurrent networks described in this chapter. In order to take advantage of its ability to combine and extract recursively constructed representations, an external controller must be added to the RAAM.

2. A network is autoassociative if the output of the network is identical to the input.

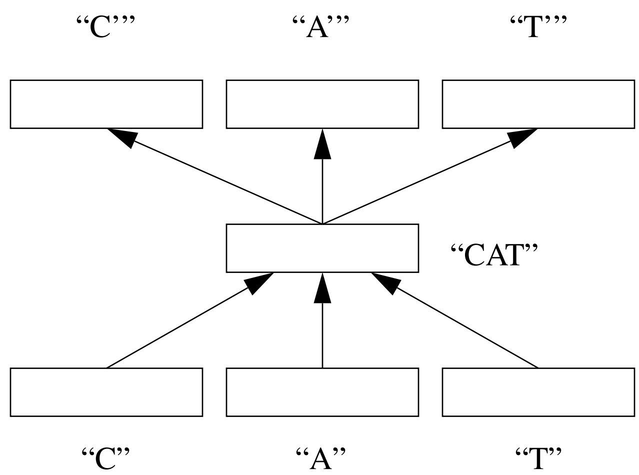


Figure 10: An auto-associative memory.

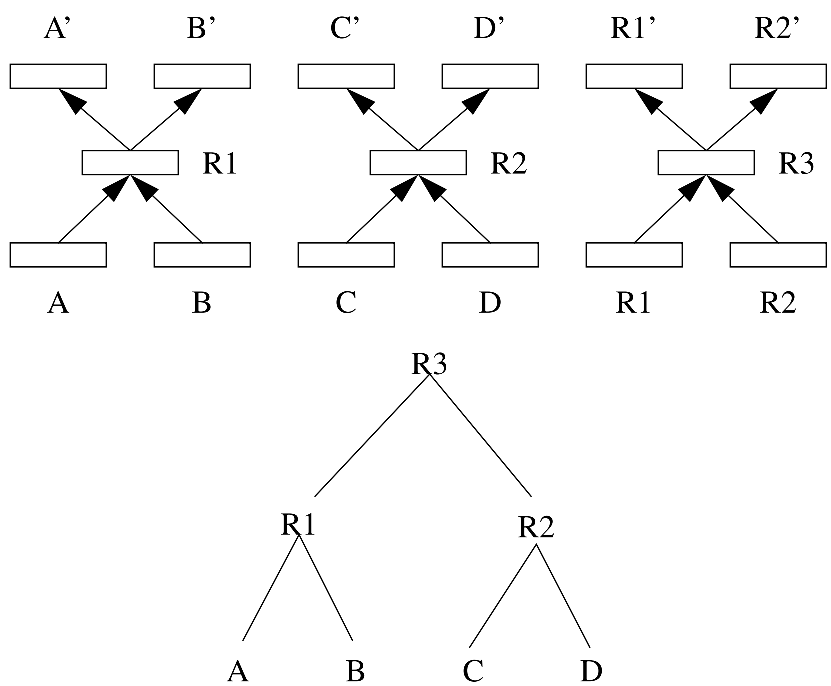


Figure 11: Encoding a tree data structure in a recursive auto-associative memory.

This controller must know when to stop expansion of nonterminal representations and store intermediate representations. In other words, the controller must have access to the structure of the entity stored within the state vector of a RAAM. Nonterminal expansion was originally controlled by a terminal test, a predicate that decides the terminal/nonterminal nature of a representation. Pollack used ε-balls surrounding a finite set of terminal vectors as the terminal. I have suggested the use of a *terminal bit* in the representation that is thresholded can indicate the terminal/non-terminal status of the representation. Large (Large, 1991) separated the terminal test from the content of the representation. In his model of musical sequence memory and improvisation, an external agent decides whether or not to expand a representation depending upon the desired depth of improvisation.

Even with these control mechanisms in place, the RAAM cannot face time dependent sequence processing alone. An external agent (i.e. programmer) must parse input sequences into valid tree structures in order to train the RAAM. In order to parse an incoming sequence, the RAAM requires external storage for intermediate results. The beauty of the RAAM model does not necessarily rest on its applicability, but upon its utility as a conceptual tool for understanding other recurrent networks. This topic will be expanded in Chapter V as I begin to explore the details of the state dynamics of RAAMs and recurrent networks.

3.3.5 Higher Order Recurrent Networks

The architectures specified by Jordan and Elman employed first-order connections between units. That is, the activation flowing from one unit to another is merely scaled ($w_{ij}o_j$) by the connection strength. High-order connections have been proposed (Rumelhart, Hinton & Williams, 1986b; Lee et al., 1986; Pollack, 1987a) which multiplicatively combine ($w_{ijk}o_jo_k$) incoming activations. In other words, the linear net input function ($\sum w_io_i$) is replaced with a higher order polynomial (such as $\sum w_{ij}o_io_j$) where the coefficients of the polynomial are implemented as connection strengths. One benefits of switching to higher order networks is representational: more functions can be loaded into a network with fewer resources. For instance, the exclusive-or problem becomes trivial when one has a quadratic net input function. The function $f(i_1, i_2) = g(-i_1^2 + 2i_1 - 2i_1i_2 - i_2^2 + 2i_2)$, where $i_j \in \{0, 1\}$ and g is a sigmoid or linear threshold returns the exclusive-or of i_1 and i_2 when 1 and 0 are interpreted as true and false.

Just as first order connections underlie Jordan networks and the SRN, multiplicative connections have formed the foundations of several recurrent networks. Pollack's Sequential Cascaded Networks (SCNs) (Pollack, 1991) and the Higher Order Recurrent Neural Networks (HORNNs) of Giles et al. (1990) have the appearance of Mealy machines.

The SCN has served as the workbench for many of my studies of recurrent networks. Figure 12 shows a schematic of the SCN. It is a single layer network with multiplicative connections running from the input units to the output and state units. The weights are stored in a three-dimensional array (Context Network) that is multiplied by the current state to yield a weight matrix. This matrix, called the Function Network, is multiplied by the input vector resulting in a net input vector. A sigmoid function is applied to elements of this vector, yielding the next state and output vectors.

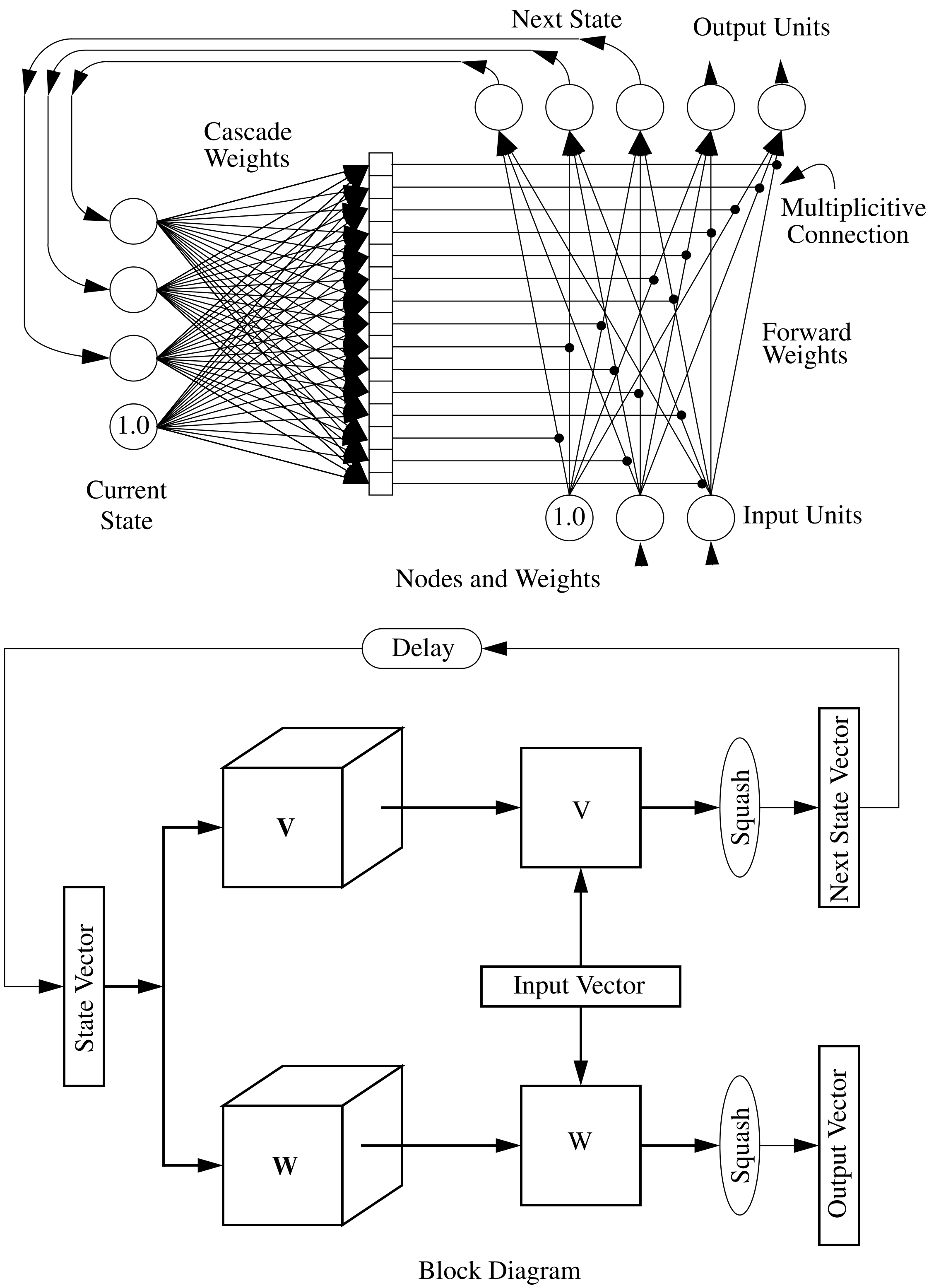


Figure 12: The sequential cascaded network.

Occasionally, when the output units are separate from the next state units, it is best to consider the cascade weights as two three-dimensional matrices. One matrix determines the mapping from the input units to the output units. The other matrix determines the mapping from the input units to the next state units. To calculate the output or next state, take the inner product of cascade weights and the state vector (the appended one is a bias). Another inner product is then formed with the resulting two dimensional matrix and the network's input. Finally, the result is then "squashed" with a sigmoid, hyperbolic tangent, or similar bounded monotonically increasing function. These operations are summarized in Equation 9.

$$O^{(t)} = g\left(\left(W \cdot \begin{bmatrix} S^{(t)} \\ 1 \end{bmatrix}\right) \cdot \begin{bmatrix} I^{(t)} \\ 1 \end{bmatrix}\right) \qquad S^{(t+1)} = g\left(\left(V \cdot \begin{bmatrix} S^{(t)} \\ 1 \end{bmatrix}\right) \cdot \begin{bmatrix} I^{(t)} \\ 1 \end{bmatrix}\right) \qquad \text{(Eqn 9)}$$

The SCN is usually trained using a truncated form of back propagation that saves the trainer from storing the activation history of the network. The error function derivative with respect to the weights feeding into the state output nodes is given by Equation 10 and the error derivative for the weights leading to the output units is Equation 11.

$$\frac{\partial E}{\partial V_{ijk}} = \left(\sum_a (O_a^{(t)} - T_a^{(t)})\, g'(O_a^{(t)}) \left(\sum_l I_l^{(t)} W_{alk}\right)\right) \left(g'(S_i^{(t-1)}) I_j^{(t-1)} S_k^{(t-2)}\right) \qquad \text{(Eqn 10)}$$

$$\frac{\partial E}{\partial W_{ajk}} = (O_a^{(t)} - T_a)\, g'(O_a^{(t)}) I_j^{(t)} S_k^{(t-1)}. \qquad \text{(Eqn 11)}$$

In both equations, $O_i^{(t)}$ is the output of unit i at time t, $I_i^{(t)}$ is the i^{th} input at time t, $g'(x)$ is the derivative of the nonlinear squashing function in terms of the activation of the unit, $T_a^{(t)}$ is the desired output for output unit a of the output nodes, and E is the standard sum of squared error function. E is assumed to be the standard sum of square error of the output with respect to a training target. This equation is not the true derivative of the error function, however. The recurrence introduces many extra terms that complicate the calculation of the gradient that can be safely ignored due to the derivative of squashing function (Pollack, 1991). As error is propagated backward through the network, the magnitude of the signal is reduced by at least a factor of $1/4$ when the traditional sigmoid function is used as the activation function, $g(x)$, since $g'(x) = x(1-x)$ for this particular choice of g. The scaling factor corresponds to the maximum value of the activation function derivative at zero net input. For most net inputs, those with large magnitudes, the derivative is much lower than the derivative at zero. The simulations reported in (Pollack, 1991; Kolen & Pollack, 1991) used this truncated form of gradient descent with momentum to modify the weights in the network.

Despite the loss in magnitude of the error correcting signal, others have used the full gradient in recurrent network learning systems. Originally described by Williams & Zipser (1989) and expanded to include second-order recurrent networks by Giles et al (1990), the gradient is expressed as a recurrence relation. This relation can be calculated during runtime and used whenever corrective feedback is provided. Thus, the learning system need not maintain a history of previous input events as they have been incorporated in the accumulated partial derivative. The full gradient does not come for free: A space penalty is accrued by this method. In addition to the weight change scalar, the learning system must also store the partial derivative vector for each weight. This increases the storage requirements from $O(|W|)$ to $O(|W||S|)$, which may be unwieldy for large networks.

The update equation for weight W_{ijk} at time t appears in Equation 12 (see (Giles et al., 1990)). The partial derivative of the state with respect to a given weight can be represented by the recurrence relation in Equation 13. Notice that only the partials from the previous time step are necessary for calculation of the current derivative. This fact allows the implementor to forward propagate the partial as the network is processing an input sequence. When training feedback is available, the accumulated partials are then used to calculate the total gradient at the current timestep without back-propagating error back through time.

$$\Delta W_{ijk} = -\eta \frac{\partial E}{\partial W_{ijk}} = -\eta \left(T_0 - S_0^{(t)}\right) \frac{\partial S_0^{(t)}}{\partial W_{ijk}} \qquad \text{(Eqn 12)}$$

$$\frac{\partial S_l^{(t+1)}}{\partial W_{ijk}} = g'(\Xi_i^{(t)}) \left[\delta_{li} S_j^{(t)} I_k^{(t)} + \sum_{m,n} W_{lmn} I_n^{(t)} \frac{\partial S_m^{(t)}}{\partial W_{ijk}} \right] \qquad \text{(Eqn 13)}$$

$$\Xi_i^{(t)} = \sum_{j,k} W_{ijk} S_j^{(t)} I_k^{(t)} \qquad \text{(Eqn 14)}$$

The weight update procedure defined in Equations 12, 13, and 14 operates on a slightly different network than the one described above. Rather than employ a separate output system, the output of the network is presumed to be the activation of S_0, the first state unit. The error signal originates from the difference between this unit and its target value. In all three equations, $S_i^{(t)}$ is the output of state unit i at time t, $I_i^{(t)}$ is the i^{th} input at time t, $g'(x)$ is the derivative of the nonlinear squashing function in terms of the activation of the unit, $T_0^{(t)}$ is the desired output for output unit a of the output nodes, and E is the standard sum of squared error function.

SCNs were originally applied to the task of deciding formal languages. In this task, the network observed a sequence of symbols from a two symbol alphabet (0 and 1) and calculated a single output value (plus the obvious next state vector). The desired output value depended upon the grammaticality of the string: 1 if the input string seen so far was contained in the language, and 0 otherwise. This mechanism mirrored the grammaticality judgment expected of the Mealy and Moore machines in the proof of their regular generative capacity. These networks clearly have a *perceptual* similarity to the finite state machines of Mealy and Moore in terms of connectivity. Yet in Chapter V I will show that this similarity does not extend to functionality: SCNs and other recurrent networks can display *infinite* state dynamics.

3.4 Summary

In this chapter, I have catalogued a collection of recurrent neural network models. Several of these mechanisms have information processing capabilities that are currently misunderstood. Although information processing in feed-forward networks is easy to grasp, the iteration of state transforms introduces many properties that researchers analyzing feed-forward networks do not have to contend with. What separates recurrent networks from their feed-forward brethren is internal memory that affects later processing. Yet it is precisely this difference that allows recurrent networks to be applied to problems involving temporal context.

The networks of Jordan, Simple Recurrent Networks of Elman, and the Sequential Cascaded Networks and Recursive Autoassociative Memories of Pollack share the overall internal organizations of traditional digital sequential circuits. Each network calculates the next state vector from the current state vector and current input vector. The current output can either be calculated by using only the next state vector, or calculated from both the current state and input to the network. While it is easy to describe the processing occurring in feed-forward networks without referring to their dynamical properties, recurrent networks are intimately tied to such descriptions. The next chapter will introduce dynamical system terminology critical to the understanding of these systems. While digital Mealy and Moore machines are computationally equivalent as finite state machines, in a later chapter I will show that the analogy does not hold for the analog state machines described above. Finite state machine interpretations may actually inhibit our understanding of the processing occurring in recurrent networks.

CHAPTER IV

DYNAMICAL SYSTEMS
AND
ITERATED FUNCTION SYSTEMS

4.1 Introduction

The last chapter reviewed several recurrent network architectures and their various learning techniques. In recurrent networks, the current activation of the network can depend upon input history of the system and not just the current input. Thus, they have the potential to encode, store, and retrieve information dynamically. The remainder of this dissertation will describe a new approach to understanding their behavior in terms of the routes, strategies, and schemes that recurrent networks use to manipulate information. While some researchers have put forth explanations of recurrent network operation of these systems in terms of finite state machines, a simple demonstration in Chapter V will show how such descriptions can fail to capture the potential information processing power of a recurrent neural network. Before heading into that discussion, a few definitions will be needed. This chapter will provide an introduction to dynamical systems, iterated function systems, and symbolic dynamics. The ideas described below will support the foundations necessary for understanding of the dynamical properties of recurrent networks. Only then will we be able to comprehend their information processing abilities.

4.2 Dynamical Systems

Computation is often taken for granted. We can point to a variety of artifacts that carry out computation: personal computers, calculators, coin sorters, etc. Yet it is unclear how one goes about recognizing computation occurring in natural systems such as cells, neurons, and animals. An integrated circuit with transistors, capacitors, and resistors can be configured to implement the logical NAND behavior. Thus, we naturally consider this organization of silicon to be a NAND gate. Another organization of similar parts could yield an operational amplifier. Specifically, how can we identify computation in artifacts in which the goal of computing played no guiding role in their development? What logical functions are an *in vivo* cortical neuron, a flowing stream, or a glass prism, computing? Each has a particular causal organization, but it is unclear that a computational teleology exists for any of these examples.

We could apply a similar line of questioning to recurrent networks. In order to understand the roots of information processing, it is necessary to understand the underlying mechanisms upon which they are based. Before directly answering this question, it will be necessary to review another explanatory framework known as dynamical systems theory. Dynamical systems theory attempts to describe systems in motion, those that change over time. In the previous chapters, I repeatedly acknowledged that the difference between feed-forward networks and recurrent networks was one of static versus dynamic behavior—behavior that changes over time. When studying real systems, such as swinging pendulums, dripping faucets, or circadian rhythms, motion can depend upon both observable and unobservable variables. According to Casti (1988), “the study of natural systems begins and ends with the specification of observables belonging to such a system, and a characterization of the manner in which they are linked."[1] In short, dynamical systems theory is the mathematical characterization of those observables that allows us to describe, classify, and predict behaviors of natural and artificial systems. Computation, as I will clearly demonstrate, is but one aspect of dynamical systems theory and is best understood together with other, equally viable, ways of describing dynamic behavior.

4.2.1 Definition

The first order of business is to define what a dynamical system is. A dynamical system consists of two parts, a state and a dynamic. A state describes the current condition of the system as a vector of observable quantities, like position, mass, and temperature. The collection of possible states is known as the state space (or phase[2] space) of the system. State captures those features a researcher holds relevant. In a sense, the state delimits the system, separating the interesting from the noninteresting. A dynamic describes how the system’s state evolves over time. The dynamical systems described below are deterministic in that the underlying dynamic is deterministic; for each state in state space, the dynamic uniquely specifies a single next state or direction of motion. While many nondeterministic systems exist, this overview specifically avoids the issues surrounding dynamical systems with stochastic components.

A simple example of a dynamical system is a freely moving pendulum. The state of the pendulum describes the current angular position, θ, and current velocity, ω. The dynamic is in form of differential equation describing the relationship between the rates of change of these two quantities, θ, and ω, as shown in Equation 15. The pendulum also exemplifies continuous dynamical systems. Notice that our description of the pendulum makes two assumptions. First, the state space is continuous. Second, the equations of motion are continuous in time.

---

1. (Casti, 1988) page 2

2. A phase space is a special kind of state space, but for the present discussion the difference is insignificant.

Table 2:A taxonomy of dynamical systems.

		State Space	
		Continuous	Discrete
Time	Continuous	Differential Equations	Spin Glasses
	Discrete	Iterated Mappings	Automata

$$\frac{d^2\theta}{dt^2} + c^2 \sin\theta = 0$$
$$\omega = \frac{d\theta}{dt} \quad \text{(Eqn 15)}$$

Continuity of time or state, however, is not a necessary feature of a dynamical system. Ecological population models are an example of a family of discrete time dynamical systems, for instance. The number of animals alive, x, at a given time, t, is the state and the dynamic relates the current population with a future population. The logistic model (Equation 16), is often used for this purpose (May, 1976). The value x_t is the proportion of animals alive at time t with respect to some maximal population. The iterated mapping characterizes growth and demise of populations, such as all the rabbits that inhabit a particular stretch of forest. When the population is small, growth is fed by available resources. When the population is large, a lack of resources and overcrowding can hamper growth. The parabolic bump of Equation 16 captures these two extremes. The parameter η controls the growth rate of the population. Environmental factors, such as birth rate and availability of food for reproduction, determine the value of this parameter.

$$x_{t+1} = \eta x_t (1 - x_t) \quad \text{(Eqn 16)}$$

Taken together, the examples of the pendulum and the rabbit population model illustrate that we can characterize dynamical systems two independent features that differentiate the granularities of the state space and the time course of the dynamic. The result is a taxonomy of dynamical systems presented in Table 2. We can describe both time and space in terms of continuous or discrete quantities.

In discrete time models, the time course of the system has been abstracted from the model. Consider the sequence of locations obtained by measuring the pendulum with a strobe light. The pendulum's bob will appear to jump from position to position. If the frequency of the strobe matches the frequency of the pendulum, the bob will appear to stand still. The state space sampling need not be periodic—the strobe could be driven by a very arbitrary time sampling. Therefore, there is no way to objectively compare durations in these models because they preserve ordering but not duration. Thus, it is not surprising to find that many attempts to produce rate invariance in discrete time systems have suffered difficulties.

Continuous time models preserve event orderings, also. Like its discrete counterpart, the continuous time reference frame allows one to assign strict event orderings. One may think that we can make objective comparisons of durations in continuous time models. In both the discrete and continuous case, however, time is an abstraction. Time in the model may be a nonuniform scaling of the true time course of events, whatever that may be. For example, we may model a fast/slow oscillator with a fixed-rate oscillator with the appropriate phase-dependent scalings of the time axis.

We can make a similar distinction with continuous versus discrete space models regarding the granularity of state space. Discrete space systems have arbitrary topologies as induced by the dynamic. Continuous state space systems, however, take advantage of the underlying continuity of the state space. We describe their dynamics in terms of continuous functions that maintain neighborhood relations.

These differences combine to produce four unique system classifications. Before delving into definitions and examples of dynamical system properties, here are a few examples of models and their place in the taxonomy. Systems defined by differential equations, like the pendulum, are both continuous in time and state. A researcher can integrate these dynamical systems over time in order to observe their behavior or to predict the behavior of the modeled system. Integration can be computationally expensive when simulations are implemented on a computer. Thus, many researchers work with discrete time dynamics over continuous state spaces. The result is an iterated map, such as the logistic map. Continuous time and discrete states occur in the modeling of certain physical processes such as spin glasses and percolation systems. We can characterize these systems by large numbers of objects with a discrete property, such as magnetic orientation. Finally, the traditional automata of computer science are examples of systems in which both time and states are discrete. The definitions and descriptions presented below will work with any of these systems, although some are more amenable to the certain definitions than other. The best strategy for understanding the properties of dynamical systems is to project these properties upon different models from the taxonomy.

4.2.2 Trajectories, Transients, and Attractors

The dynamic of a system encapsulates the process of change over time. The sequence of states exhibited by a dynamical system during this evolution is called a trajectory. Although a system can experience radically different initial trajectories, these trajectories often approach a characteristic behavior in the limit known as an attractor. An attractor is a region in state space where all trajectories converge from a larger subset of state space. The region of state space from which the system will evolve to the specified attractor is called a basin of attraction. As the system evolves toward the attractor, the initial part of the trajectory is considered a transient. One way of finding attractor approximations is to pick an initial condition and let the dynamic propel the state of the system through state space. After stripping away the initial transients of the trajectory, what is left is a good approximation to the underlying attractor. This is especially true if the goal is to plot the attractor on some display device (or printer) since any discrepancy between the real attractor and its approximation is often lost in the finite resolution of the display device. This assumes, of course, there is a single attractor. Since many multiple-attractor dynamical systems exist, the iterative method will only approximate a single attractor for the basin from which the initial conditions were selected.

Although there is an apparent qualitative difference between discrete and continuous systems, both exhibit the same types of dynamic behavior. In the temporal limit, only four types of qualitatively different attractors, or more correctly, regimes, can be identified: fixed points, limit cycles, quasiperiodicity, and chaos. Some refer to a chaotic attractor as a strange attractor for historical reasons. These behaviors can be found across all combinations of discrete or continuous time and space.

4.2.2.1 Fixed Points

Many systems appear stuck in a particular state, such as a pendulum bob at rest. We characterize this lack of motion as a *fixed point*, the simplest of the three dynamic regimes. This resting state can be the final state of a system as, in the pendulum's case, the system dissipates potential energy. Thus, a fixed point is thought of as the limit of a sequence of states resulting in a changeless state. Fixed points occur in both continuous and discrete state spaces when the dynamic fails to move the state away from its current value. A trajectory whose state does not change is a finite fixed point, like the resting pendulum, otherwise we call it an infinite fixed point (Zak, 1988). Fixed points appear in two varieties: stable and unstable. Small perturbations to a stable, or attracting, fixed point results in a trajectory that returns to the original attractor after the initial transients die away. Consider the iterated map $x_{t+1} = \frac{1}{2}x_t$ that has a fixed point at zero, the solution to the equation $x = \frac{1}{2}x$. The fixed point is attracting since for every value $c > 0$ there exists a time t_o such that for all possible states x_t at time $t > t_0$, $|x_t| < c$. In other words, the distance between the system's state at time t and the fixed point will become arbitrarily small, given enough time.

Not all fixed points attract their surrounding states, some actually repel their neighbors (Figure 13). If the system starts in one of these states, it will remain forever in that state, hence the label of "fixed." Any perturbation, however, will dislodge the system from its perch and send it rolling away from its once peaceful home. Consider the iterated map $x_{t+1} = 2x_t$. It has a fixed point at zero. Although a system started at zero will stay there forever, small deviations from this point will rapidly accelerate the system state away from the fixed point. Compare this behavior with the previous mapping, $x_{t+1} = \frac{1}{2}x_t$. This mapping clearly has an attracting fixed point at zero. Yet the state space trajectories of these two mappings near the origin are radically different. The difference in behavior lies in the slope of the mapping at the fixed point. If the absolute value of the slope of the mapping at the fixed point is greater than one then the iteration will escape from the fixed point. Otherwise, when the slope is between -1 and one, the fixed point attracts its neighbors in state space.[3]

Between the attractive and the repelling fixed points lie the saddle points. These points in state space exhibit both attractive and repelling dynamics depending upon the direction of approach. Saddle points occur in systems with more than one dimension. Imagine a hyperbolic energy surface in three-space. The dynamics attempt to minimize this energy in direction of surface gradient. Like the other fixed points, the saddle point can be found where the derivative of the surface is zero. This point is special in that along one axis a marble will roll toward the saddle point and in another axis of motion it will roll away from it, though the marble could conceivably be balanced at the saddle point and not roll away.

4.2.2.2 Limit Cycles

While fixed points epitomize the denial of motion, limit cycles indicate a system in constant motion. The term *limit cycle* refers to the limit of a trajectory that periodically revisits a set of states during an oscillation. For instance, a frictionless pendulum has a periodic trajectory once it begins swinging. A computer program caught in an infinite loop is also "experiencing" a limit cycle. The ideal pendulum endlessly cycles through a set of angle and velocity pairs determined by the initial conditions of the system although a real pendulum undergoes frictional forces that will eventually damp its motion until it reaches a fixed point. Similarly, the program counter endlessly cycles through a fixed sequence of instruction addresses. The process of repeating a set of state values over time can be described by the temporal relationship $x(t+p) = x(t)$. The term p is the period of oscillation. To illustrate some general properties of limit cycles, consider the iterated mapping, $f(x) = -x^3$. This mapping has a limit cycle of period two that can be found by solving the equation $f(f(x)) = x$. This equation has three solutions, -1, zero, and one. The solutions -1 and one are a cycle: the system will alternate between the two values, $f(-1) = 1$ and $f(1) = -1$. The zero solution is a subharmonic of the period-two oscillation, it has a period that is an integer divisor of the target period. In general, a cycle of period n, and its subharmonics, for a given mapping can be found as a solution to Equation 17.[4]

$$f^{\circ n}(x) = x \qquad \text{(Eqn 17)}$$

Limit cycles, like fixed points, can either attract, repel, or display some combination of the two. Attracting limit cycles produce oscillations which are stable with respect to minor perturbations. Repelling limit cycles can be found in which perturbations send the system away from limit cycle in order to approach another attractor. As in the fixed point case, the slope of the dynamic near the oscillation will determine the attracting or repelling nature of the limit cycle.

Some systems, such as the van der Pol oscillator, exhibit saddle effects with oscillations. The van der Pol oscillator has two attractors, a fixed point at the origin and a cycle surrounding the origin. If the oscillator starts outside the cycle, the dynamics will eventually send the system toward the oscillation. If started on the interior of the cycle, the state of the system will evolve toward the fixed point at the origin.

---

3. A slope of 1 makes everything a repelling fixed point. A slope of -1 makes every point, except the origin, a period two repellor.
4. The degree-n notation indicates nesting of f. Thus, $f^{\circ 3}(x) = f(f(f(x)))$.

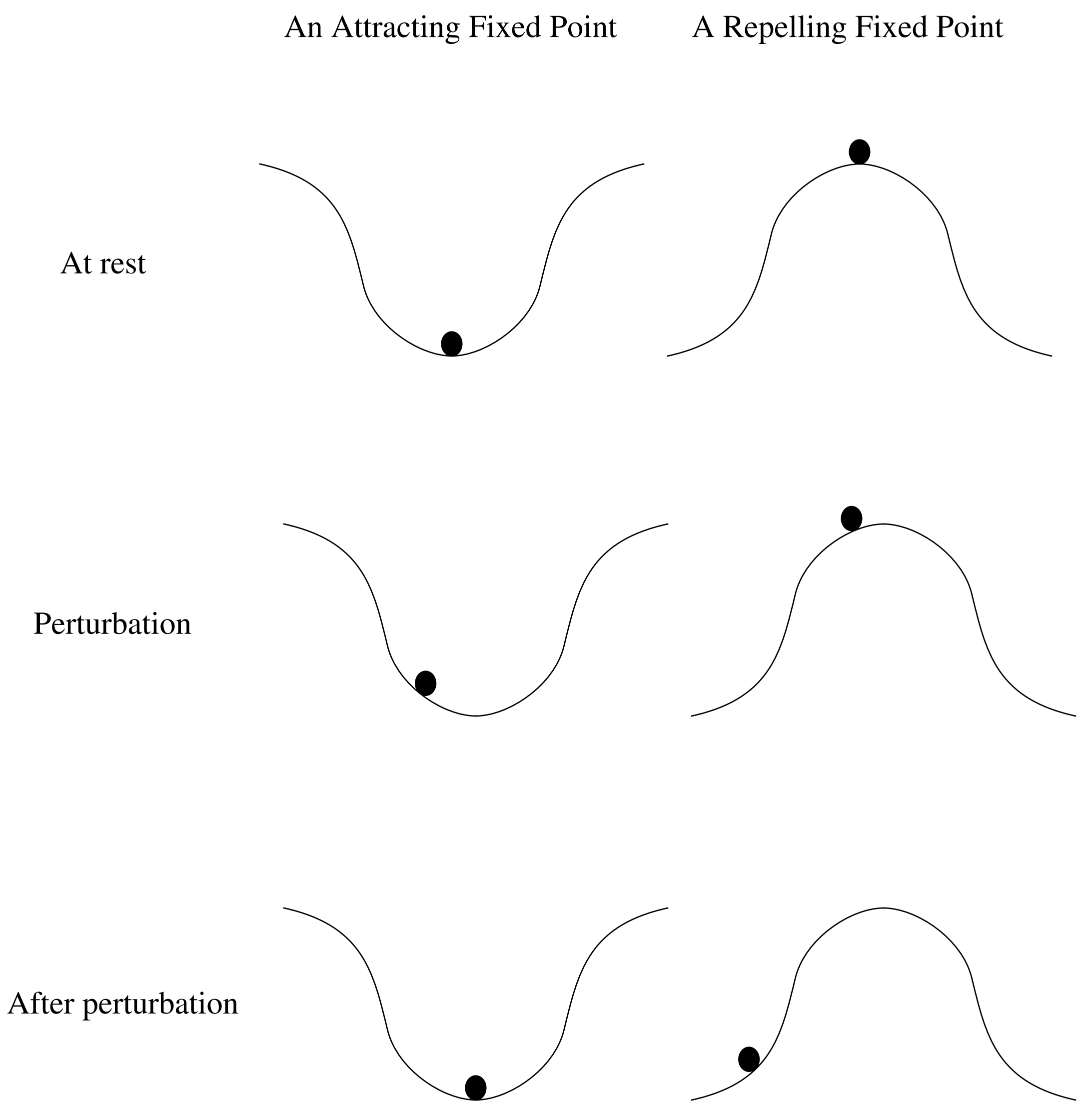


Figure 13: Attracting and repelling fixed points.

4.2.2.3 Quasiperiodic Behavior

Limit cycle behavior is not restricted to a single periodicity. Systems can possess attractors and repellors with multiple constituent periodicities. Sometimes the behavior of such compositions can be unexpected. Consider the case of the system defined by two oscillators. In Equation 18, ϕ measures the phase of the oscillators and Ω is the period of oscillation.

$$\begin{aligned}\phi_1 &= \sin\Omega_1 t \\ \phi_2 &= \sin\Omega_2 t\end{aligned} \qquad \text{(Eqn 18)}$$

The *winding number*, $\omega = \frac{\Omega_1}{\Omega_2}$, is important for determining the behavior of the system. If ω is rational, then the system will be strictly periodic with period of

$$\frac{\Omega_1\Omega_2}{\gcd(\Omega_1, \Omega_2)}. \qquad \text{(Eqn 19)}$$

Thus, two relatively prime periods combine to form a limit cycle with period equal to the product of the original periods. In Figure 14, the top two graphs plot Equation 18 for a pair of oscillators with winding ratios of 2:3 and 2:4. Notice that the rational winding number produced one-dimensional periodic trajectories. Compare the trajectories of these rational winding numbers with the irrational ratio of $2{:}\pi$ at the bottom of Figure 14. When ω is irrational, the behavior of the system as a whole is quasiperiodic. Although frequency analysis of the trajectory will reveal the underlying constituent periodicities, the system will never revisit any point in phase space. In the limit, the trajectory of the quasiperiodic system will become a two-dimensional area.

4.2.2.4 Chaos and Strange Attractors

The third type of dynamic regime, and arguably the most interesting, is chaos. The trajectory of a system following a chaotic regime is aperiodic since no point state space is ever visited twice. If the trajectory did cross itself, the deterministic nature of the evolution functions would cause the system to behave exactly as it did after the previous same state. This crossing would result in an oscillation. When the chaotic motion is attracting, it also called a strange attractor.

Chaotic attractors can only arise when a system's dynamic has nonlinear components. If the dynamic is linear, then only fixed points and periodic cycles are possible. Linear dynamics can only stretch, compress, and rotate state space. Imagine a piece of pizza dough that someone maliciously added a drop of blue food coloring. Stretching will only elongate the blotch of blue dough and no amount of twirling will affect the spot. More important, neither of these operations will spread the food coloring to the rest of the dough. These operations, in a formal sense, map, or transform, lines in state space into lines. These operations generate state space trajectories and produce the dynamic phenomena discussed in the previous sections. Fixed points result from the repeated application of stretching or

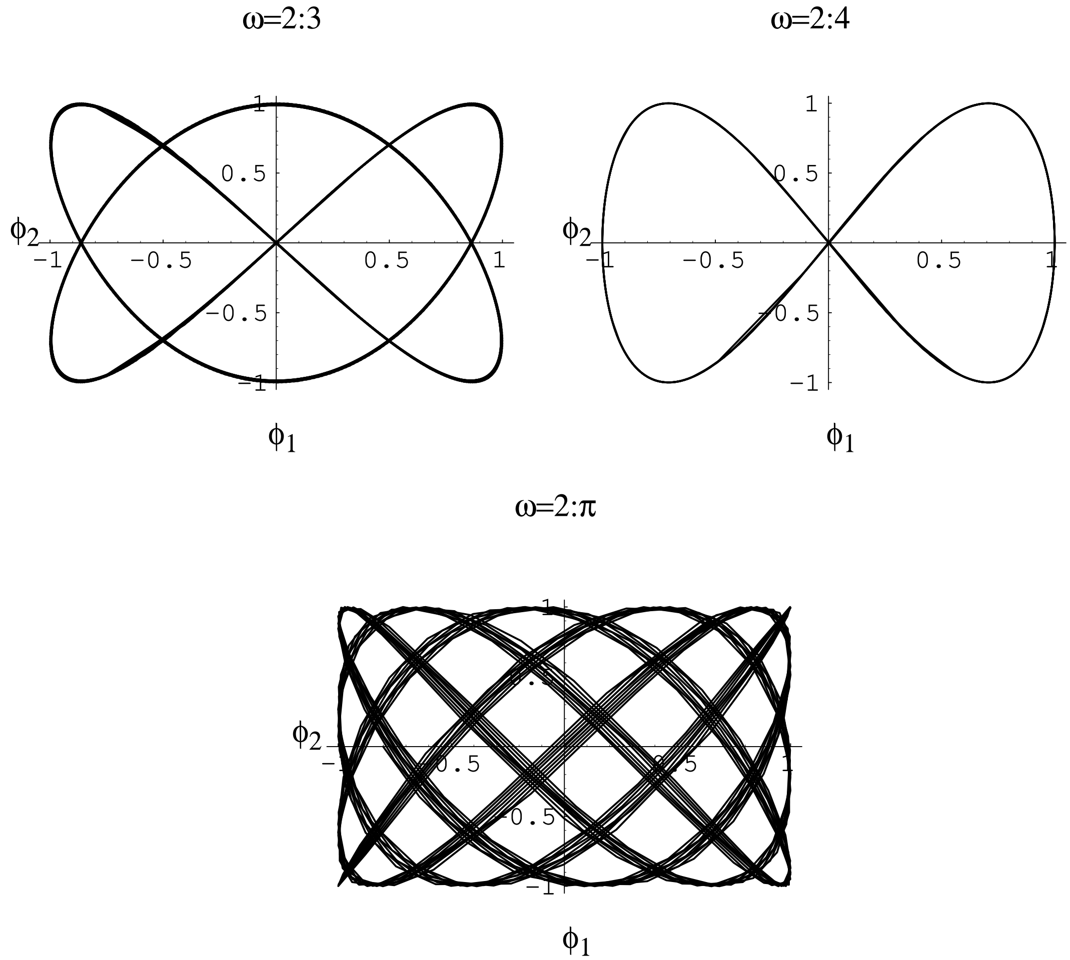


Figure 14: A demonstration comparing periodic and quasiperiodic trajectories.

compressing operators. In the limit, these operations reduce subsets of the state space to a single point. Limit cycles, on the other hand, originate from rotations of state space.

When the dynamic repeatedly stretches, compresses, and folds the state space, strange attractors emerge. Figure 15 demonstrates the effect of these operations. Stretching occurs when a dynamic scales states space in a particular direction with a scale factor greater than one. In the figure, the state space (the left most line) is stretched by a factor of two. The state space is then folded by way of a nonlinear transformation, like the quadratic shown in Figure 15. Folding is one method of reorganizing the topology of the state space, as it can create new neighborhoods. The combination of stretching and folding can have devastating effects on the original topology of the state space. The dynamic stretches neighborhoods and eventually straddles them across the bend in the horseshoe. Given enough time, the dynamic will spread the neighborhood across the entire attractor. This makes

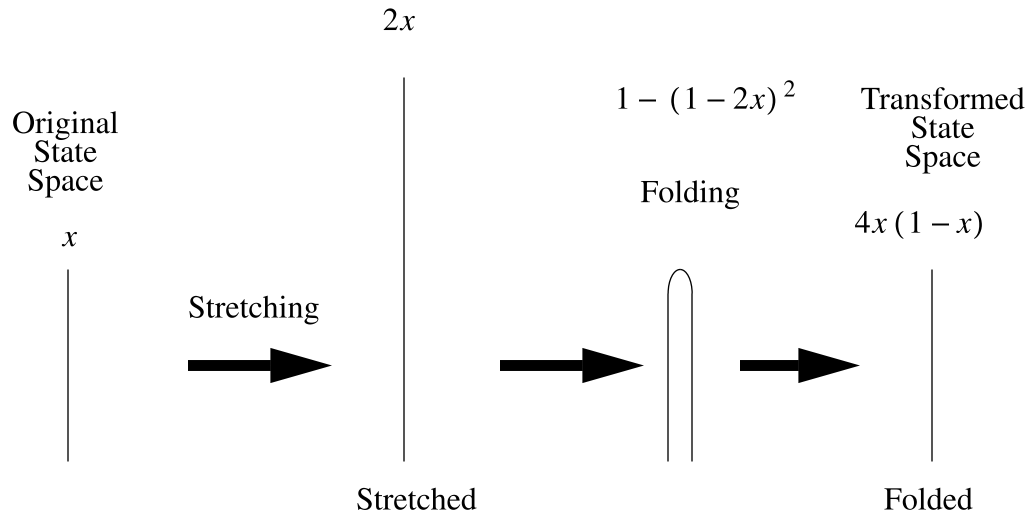


Figure 15: Stretching and folding of state space in the logistic mapping.

long-term prediction in such systems impossible: since any measurement of position is guaranteed to be incorrect, the difference between the measured state and the true state will exponentially diverge as the system evolves. Even though detailed long term predictions are impossible, this fact does not prevent the possibility of making detailed short term or gross long term predictions. The former is true because the dynamic of the system is deterministic. The later holds because the trajectory is bound by the extent of the attractor.

One curious property of chaotic system involves the dimensionality of state space. For continuous systems, three dimensions are necessary for chaotic behavior to emerge. One would not expect to see chaos in the trajectory of a single frictionless pendulum since its phase space is two-dimensional. The dynamic folds and stretches the state space manifold. When this happens in a two-dimensional space it causes trajectories to intersect, which leads to limit cycle behavior when the dynamics are deterministic. In three dimensions, the attractor manifold can be compactly rolled into a spacefilling curve that never terminates (in a fixed point) nor intersects itself (a limit cycle). If the pendulum is externally driven or is actually a double pendulum, then chaos can arise in the systems' attractor due to the increase in the system's degrees of freedom.

I mentioned earlier that deterministic chaos is characteristically different from random noise. Chaotic systems can be described by a set of deterministic equations, while a truly random process can only be characterized in terms of statistical properties. We often assume that random noise originates from a stochastic process with an inherent probabilistic element (e.g. Markov processes). The deterministic equations of motion were assumed for many years to leave no room for explicit stochastic generators. Yet, the apparently random variations exhibited by deterministic systems must come from some place. The noise source for deterministic systems is the initial conditions.

As I alluded to earlier, long-term prediction of the motion of these systems is impossible due to sensitivity to initial conditions. Sensitivity to initial conditions means: given two arbitrarily close starting points after some amount of time the system rapidly evolves the two points away from each other. In many systems this expansion cannot go on indefinitely, the extent of the attractor bounds the maximal distance between two separating points. Upon reaching the attractor, however, the trajectories will no longer correlate.

An alternative way to understand the apparently random behavior of chaotic dynamical systems is to view them as shift mechanisms on the numeric representations of their state variables. Sensitivity to initial conditions is a direct effect of the stretching and folding of state space by the dynamic. Stretching, or scaling by a factor greater than one, shifts the digits of the state representation to the left. Folding, which occurs when the state space is mapped through a function like x^2, throws away the most significant digits. Since a real number is an infinite source of digits, the shift and truncate operation can create an endless aperiodic sequence totally specified by the initial conditions. If two initial states are identical for the first k digits of their decimal expansions and the dynamics are shifting at the rate of one digit per unit time, the two trajectories will rapidly diverge after $k + 1$ time units. The accuracy of the state description at the beginning of the prediction, compared with the size of the attractor, determines how long a prediction will hold. Thus, long term quantitative prediction is impossible. Qualitative predictions could be made, however, if the structure of the underlying attractor is known. Even chaotic systems possess certain structural regularities, such as the attractor manifold in state space, that could be used for general long term predictions. For instance, knowing whether the solar system will eventually fly apart differs from predicting the location of Jupiter 50,000 years from now.

The metaphors of stretching and folding are tools for a dynamical perspective of the behavioral explanation. Alternative perspectives exist for describing the behavior of chaotic systems. For instance, some consider the state a reservoir of information. From this information theoretic perspective, chaotic systems are producers of information, or at least unpackers of the information contained in the low order digits of their state. They trade space for time. The only way to observe the very low order digits of a continuous state variable is to wait around long enough for them to be shifted into the digits accessible by our measuring devices. The following example will illustrate how we can shift from one perspective to another with minimal effort.

Consider the bakers' mapping $z = 2z \bmod 1$. A trajectory of this system is entirely dependent upon the selection of the initial state due to the continual left shift of the binary encoding of the first value of x. This focus on the binary encoding of state is typical of the information perspective. Since there are many more irrational numbers in the range $(0, 1)$ than rational numbers, the dynamics will most likely generate an aperiodic trajectory that appears random (due to the sensitivity to initial conditions), but is none the less deterministic. The logistic map, with $\eta = 4$, also exhibits aperiodic trajectories indicative of chaos. In fact, the logistic map at this parameter setting shares the same shift behavior as the baker's map. The proof of this statement hinges upon an isomorphism between the state spaces

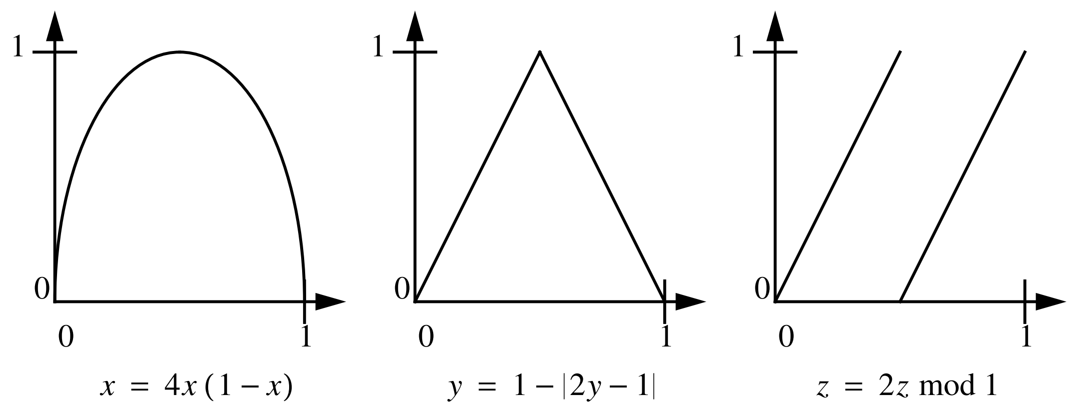


Figure 16: From logistic map to tent map to bakers' map

of the bakers' map and this logistic function. The isomorphism involves a simple coordinate transformation ($y = \frac{1}{2} - \frac{1}{\Pi}\operatorname{asin}(1-2x)$ and flipping the right side of the resulting tent map) as shown in Figure 16. While the bakers' map interpretation emphasizes bit shifting, the original logistic mapping supports folding and stretching explanations.

In the preceding paragraphs, I have provided a general account of chaotic behavior. Yet, the mathematical requirements for proving a system to be chaotic are concise and three fold. The system must display:

- sensitivity to initial conditions,
- mixing,
- and a set of repelling periodic points densely covering the attractor.

First, the system must exhibit sensitivity to initial conditions, which suggests stretching and folding of state space. Second, the dynamic must continually mix points within the attractor. This implies that for any two subsets A and B of the strange attractor, there exists a t such that for dynamic f after t iterations (or integrating for time t) using set A as initial conditions, the resulting image intersects B. In other words, the dynamic eventually spreads any subset of the attractor through the entire attractor and thus the dynamic produces topological transitivity. The last condition requires that periodic points in the state space will densely cover the attractor. The existence of a chaotic attractor does not preclude the existence of repelling periodic and fixed points in the same dynamical system. Chaos would not exist without an infinity of periodic repellors. The repellors must be extremely close to the chaotic attractor: if these periodic points were the centers of balls with small radii, the entire attractor would be covered by the balls, even as the radii go to zero.

While a system may exhibit a chaotic attractor, it simultaneously possesses an infinite set of unstable periodic attractors. This fact has led several researchers to develop "chaotic controllers" that select periodic regimes from an otherwise chaotic system (Grebogi et al., 1987; Li & Yorke, 1975). They accomplish this feat by deforming the state space near a repellor in order to balance the system near that repellor. The deformation is dynamic and is accomplished solely by manipulating the control parameter (i.e. the η in $\eta x(1-x))$). The controllers have worked well in both computer simulation and laboratory experiments. Grebogi et al. has used this scheme to select periodic regimes from the Henon map. Ditto et al. (1990) subjected an amorphous magnetoelastic ribbon to a magnetic field. Under certain field strengths, the strip would vibrate chaotically. Yet they could select low period regimes from the strip through minute manipulations of the magnetic field strength. Singer et al. (1991) performed similar experiments involving a thermal convection loop. Both research groups experimentally demonstrated that periodic behavior regimes could be "selected" by minute variation in the system control parameters.

While both chaotic and quasiperiodic trajectories appear both never revisit points in state space, there are important differences that separate these two behaviors. First, quasiperiodic systems are not sensitive to initial conditions. State perturbations will not cause exponential divergence. Rather, the system will continue to "carry" a perturbation until it is wiped out by another perturbation. Second, the frequency spectra are different. Quasiperiodic trajectories will exhibit frequency peaks at the constituent frequencies. Chaotic trajectories, on the other hand, display wide-band spectra with the power decreasing with the inverse of frequency ($\frac{1}{f}$). These spectra differentiate chaos from noise: white noise displays a uniform distribution of frequencies across the spectrum.

4.2.3 Attractor Dimensionality

Many chaotic attractors with a variety of shapes and sizes can satisfy the definition of above. Given that periodic attractors can be differentiated by their period, how can we differentiate between two different chaotic attractors? Unfortunately, we cannot resort to a property such as periodicity or volume to classify these infinite sets. We can, however, resort to another property of point sets, namely dimensionality. Attractors, like any other subsets of vector spaces have a characteristic dimension. A fixed point, since it is a point, has a dimension of zero. Limit cycles in discrete time systems also have dimensions of zero. The limit cycle in a continuous time system, such as the one exhibited by a frictionless pendulum, is a one dimensional object, a line segment joined at the ends. Other limit cycle attractors, like tori, have an integer dimensions. But what is the dimensionality of a strange attractor? To answer this question, the whole notion of dimensionality had to be rethought to account for the space filling curves associated with chaos.

Before describing the dimensionality of strange attractors in general, first consider the dimensionality of a seemingly innocuous collection of points known as the Cantor set. This set is defined recursively over a closed interval of the reals, such as $X = [0, 1]$. Now, remove the open middle-third of this set, i.e. $(\frac{1}{3}, \frac{2}{3})$. This leaves two pieces, $\left[0, \frac{1}{3}\right)$ and

$(\frac{2}{3}, 1]$. Recursively repeat this operation of middle-third exclusion on the resulting intervals. The limit of this operation is an uncountable set of points known as the Cantor set. What is the dimensionality of this set? Intuitively, the dimension is clearly less than one, as the limit of a particular sequence of subdivisions is always a disconnected point. Likewise, one would not want to classify the set as dimension zero since it appears to have more structure than a single point. To satisfy both intuitions, one must introduce the notion of noninteger, or fractal, dimension that somehow captures the recursive structure underlying the generative production of the set (Mandelbrot, 1982). One way of describing this generative production is as recursive similarity: The local shape of an object is a scaled copy of the global shape of the object. The similarity (or fractal) dimension is given by Equation 20 where N is the number of parts scaled and r is the scaling factor used to scale those parts.

$$D = \frac{\log N}{\log \frac{1}{r}} \qquad \text{(Eqn 20)}$$

Let us apply the similarity dimension to the Cantor set. In the construction described above, we can recursively decompose each region into two parts, the left and right regions. Each region is scaled to be $\frac{1}{3}$ the length of the original line. Thus, the fractal dimensionality of the Cantor set is

$$\frac{\log 2}{\log 3} = .63092975\ldots$$

Another way to identify a chaotic system is to calculate the Hausdorff dimensionality (Mandelbrot, 1982) of the underlying attractor.[5] This is related to the similarity dimensionality of fractal sets defined above. For instance, slices through the Lorenz attractor (Lorenz, 1963) show bands isomorphic to the Cantor set (Mandelbrot, 1982). A conservative estimate of the Hausdorff dimension, can be obtained by calculating the integral autocorrelation function of Grassberger and Procaccia (1983), shown in Equation 21.

$$C(r) = \frac{1}{N^2} \sum_{\substack{i,j=1 \\ i \neq j}}^{N} \vartheta(r - |X_i - X_j|) \qquad \text{(Eqn 21)}$$

The Heaviside function, $\vartheta(x)$, is 1, when $x > 0$, and 0, otherwise. X is the sequence of points being analyzed embedded in a higher dimensional space: $X_t = [x_t, x_{t+1}, \ldots, x_{t+n-1}]$. This function simply counts the number of point pairs whose distance is less than r and normalizes for the number of distance measures (N^2). In other

5. Other ways of identifying chaotic systems include proving that the dynamics have the three properties of a chaotic system and examination of the Lyapunov exponents.

words, $C(r)$ is the average number of pairings separated by less than r. For small r, $C(r)$ scales with r^{υ}, where υ is the correlation dimension. Thus, the correlation dimension υ can be found with Equation 22, where n is the dimension of the embedding space X. When the dimensionality is calculated this way, many refer to it as the box-counting dimension.

$$\upsilon = \lim_{n \to \infty} \frac{\log C(r)}{\log r} \qquad \text{(Eqn 22)}$$

Other methods of approximating the Hausdorff dimension exist. For instance, one could replace the set-dependent boxes with arbitrary grids. One problem with the grid method is that a poor choice of grids can give an inaccurate measurement of dimensionality, as such it is important to evaluate grid-based dimensions with several different grid origins. The original similarity dimension formula also has its own caveats. Specifically, it only applies to similitudes, transformation which scale all directions by the same factor. There is no known way to empirically estimate the dimensionality of an attractor generated by the interaction of multiple, nonaffine, transformations.[6]

4.2.4 Bifurcations and Phase Transitions

Some systems, like the logistic function described above (Equation 16), are sensitive to parameter adjustment. Many parameterized systems exhibit changes in their attractors as the control parameter is varied. The logistic mapping is interesting since there exist parameter values that will produce attractors of all periods, including many aperiodic attractors. Based upon the value of the parameter, η, the logistic function can produce the three different attractor classes: fixed points, limit cycles, and aperiodic sequences. Figure 17 shows how the attractor of the logistic function changes as the control parameter increases. The diagram was created by iterating the logistic function with η set to a particular value and collecting the sequence of states generated by the system. The initial transients were dropped, and the union of the remaining points were plotted on the abscissa. For values of η less than 3.0, the system evolves to a fixed point. As η increases, it reaches a point that the system evolves to a period-two orbit. The sudden doubling of the number of periods is called pitchfork bifurcation (Figure 18). While it may appear that fixed point splits, this is not so. The fixed point still exists, however, it is no longer stable after the bifurcation. The act of bifurcation transfers stability from the fixed point to a pair of newly created fixed points for the *second* iterate of the mapping. These “fixed points” appear as period-2 oscillations in state space. As the parameter increases, the attractor bifurcates again to produce a period-four attractor. Period doubling theoretically can continue indefinitely; a period-n limit cycle bifurcates into a period-$2n$ cycle.

Does the period doubling cascade ever end? If the distance between each bifurcation was constant, the answer would be no. This is usually not so, however. Feigenbaum (1983) discovered a self-similarity in the cascade that he used to prove that the distance (in

---

6. This state of affairs prevents application of the analytical methods determining dimension of recurrent network state space (Chapter 5).

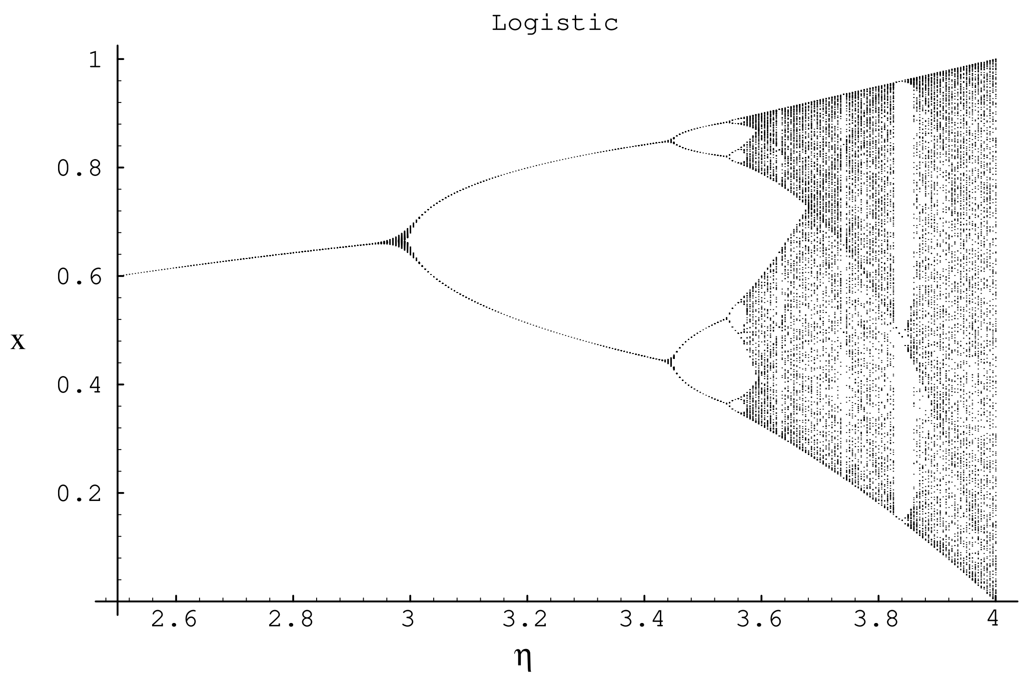


Figure 17: Logistic Function Bifurcation Diagram

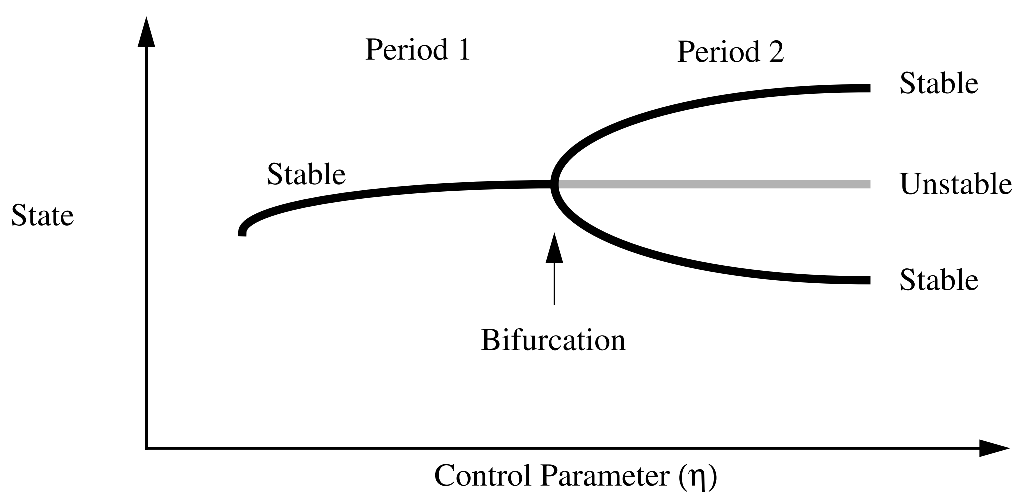


Figure 18: A pitchfork bifurcation.

η space) between two orbits[7] decreases geometrically with the size of the period. The $\Delta\eta$ necessary to go from period-2 to period-2^{n+1} is less than what is needed to go from 2^{n+1} to 2^{n+2}. At first the system bifurcates slowly. After many bifurcations, very small changes in η will cause period doublings. For the logistic function, the limit of these small changes occurs at $\eta_{2^\infty} = 3.5699\ldots$ In fact, for all odd fixed m, the sequence of parameters $\{\eta_{m2^n}\}$ has a limit. At these parameters, the mapping no longer exhibits a periodic oscillation. The system is at *criticality*, a condition marked by the onset of chaotic behavior from periodic. The critical values for this system is the set $\{\eta_{m2^\infty}\}$.

While the critical values are important because they are the limits of bifurcation cascades, they also suggest other interesting mechanisms at work. As mentioned earlier, the dynamic can stretch and fold state space. Sometimes these operations will attain some measure of alignment--periodicity occurs when iterates of the dynamic map on to themselves. Criticality is another such accident. Period doubling occurs when the dynamic of a period-n attractor is similar to dynamic of the period-$2n$ mapping (Feigenbaum, 1983). Similarity in this context suggests that one mapping is a scaled and translated copy of the other mapping. This similarity does not extend to $4n$ iterate, except in the case of criticality. At criticality the similarity holds for all iterates of the form $n2^m$. Thus, the trajectories for logistic mappings and other unimodal maps at criticality parameter never repeat except a countable number of repelling exceptions with periods of $n2^m$.

If a system that initially exhibits fixed point behavior and then goes through a cascade of period-doubling pitchfork bifurcations, are period-2^n limit cycles and chaos the only behaviors this system can exhibit? The answer is no. Sometimes the dynamics of a system will produce a saddle-node bifurcation. In the logistic mapping, a saddle-node can create a period-n attractor when the peak of a hill or bottom of a valley of the n^{th} iterate crosses the diagonal. This crossing creates two orbits, one stable, the other unstable. These bifurcations are responsible for the appearance of odd-period attractors.

Recall that chaotic trajectories are surrounded by a dense cloud of unstable periodic repellors. The pitchfork bifurcation cascade offers an explanation for this property. The unstable fixed points and limit cycles are the instabilities created by the bifurcation cascade; every period-n bifurcation leaves behind $n-1$ unstable trajectories.

An entire cascade of pitchfork bifurcations yields a collection of attractors with periods of the form 2^n. These long period regions eventually give way to saddle-node bifurcation creating much shorter periods such as seven, five, and three. It is clear from the bifurcation diagram that the periods do not appear in increasing order of period size as the control parameter increases, i.e.$\eta_{n-1} < \eta_n < \eta_{n+1}$ for all n, where η_n is a parameter producing a period n attractor. Sharkovskii (Sharkovski, 1964) discovered that a nonintuitive

7. Specifically, orbits containing the critical point of the mapping, e. g. the top of the logistic parabola.

ordering of the parameters underlies the bifurcation tree. He focused on the first occurrence of a period in the tree[8] and found that the minimal period parameter values are ordered as shown in Equation 23.

$$\begin{gathered}3 > 5 > 7 > 9 > \ldots > 2 \cdot 3 > 2 \cdot 5 > 2 \cdot 7 > 2 \cdot 9 > \ldots \\ 2^n \cdot 3 > 2^n \cdot 5 > 2^n \cdot 7 > 2^n \cdot 9 > \ldots > 2^n > \ldots > 4 > 2 > 1\end{gathered} \quad \text{(Eqn 23)}$$

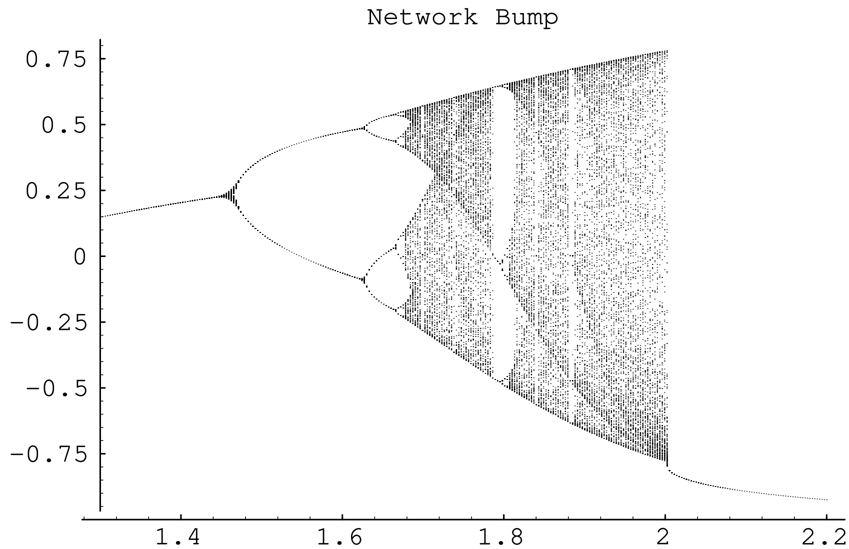
The ordering starts with the powers of two. This captures the initial period doubling cascade that follows a fixed point. The ordering ends with period three preceded by remainder of the odd periods. Between these two extremes one finds a sequence of periods of the form $m2^n$. For a fixed exponent n, m is filled with the increasing odd numbers. This peculiar ordering implies the existence of a period-six attractor before and after the appearance of the map with period three. Before the first appearance of a period-three attractor, a saddle node bifurcation will create a period-six attractor. The second period-six attractor will occur after the period three as the result of a period doubling pitchfork bifurcation. One useful application of the Sharkovskii theorem is searching for specific periods in one dimensional mappings. If a period-four oscillation is isolated (we know the parameter for it by way of analysis (solving $x = f^{\circ 4}(x)$) or experiment (bifurcation diagram)), the ordering tells us that there exists parameters for period-2 and fixed point attractors. Likewise, if a cycle of period three is ever encountered in a one dimensional system, then for any n there exists a parameter value that produces a period n orbit. From this one can conclude that any period is possible. York (1975) originally described this condition as chaos. However, chaos actually occurs much earlier, after the 2^n bifurcation limit.

One of the most interesting features of the bifurcation diagram is its universality. Many other iterated functions produce similar diagrams. Feigenbaum, in addition to the contributions described earlier, proved that this behavior is universal for all unimodal, i.e. single bump, functions. Period doubling is independent of the specific function as long as certain conditions regarding the gross shape of the mapping are met (Feigenbaum, 1983). The results of this theoretical work suggest that other universals exist for other families of mappings.

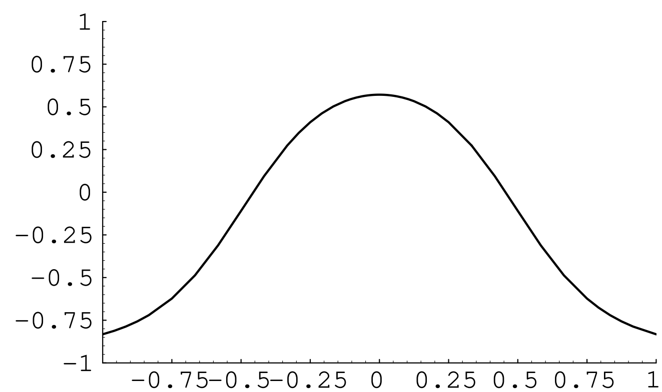
As a demonstration of the universality of period doubling in the context of neural networks, I provide a recurrent neural network with the properties specified by Feigenbaum (1983). The bifurcation diagram in Figure 19 illustrates the period doubling cascade of a one input, two hidden units, one output recurrent neural network that provides the single scalable bump through a modulation of the gain parameter of the hyperbolic tangent transfer function. This iterated mapping exhibits the same period doubling cascade as the logistic function and displays the full range of dynamical behaviors. In Chapters V and VII, I will extensively use this mapping as a demonstration tool.

Bifurcations come in several varieties according to the effects of parameter changes on fixed points. A pitchfork bifurcation occurs when a stable fixed point splits into a repelling fixed point and a limit cycle. This event is the crucial to Feigenbaum's description of

8. For all values of r less than r_n, there does not exists an r which generates a period of length n.



a



a=1.8

$$x' = \tanh\left(a\tanh\left(ax+\frac{a}{2}\right) - a\tanh\left(ax-\frac{a}{2}\right) - a\right)$$

Figure 19: A recurrent network with an unimodal transfer function with bifurcation diagram.

what happens during a period doubling bifurcation. The same process occurring in a continuous time system is known as a Hopf bifurcation. When a stable fixed point meets an unstable one, the two attractors annihilate each other in a saddle-node bifurcation. Itermittency, a short turbulent phase followed by a longer period of stability, suggests a saddle-node bifurcation. The wealth of dynamical behavior encapsulated in a system as simple as the logistic function can be traced to the interplay of the period-doubling pitchfork bifurcations and the destroying saddle-node bifurcations.

4.2.5 Summary

Dynamical systems theory provides a framework for describing systems whose state changes over time. The internal dynamic, a differential equation or iterated mapping, governs the evolution of the system. While there are many dynamical systems and many ways to encode their internal dynamic, there are only four general types of behavior they can produce. Fixed points, limit cycles, quasiperiodicity, and chaos exhaust the behavioral repertoire of dynamical systems. These regimes may either attract or repel state space trajectories. The process of bifurcation allows systems to shift from one regime to another or from a stable regime to an unstable one. While bifurcations can either affect existing periodicities or create new ones, chaos occurs at the limit of an infinite cascade of bifurcations.

So far, one important aspect of dynamical systems has been avoided. If we are interested in modeling natural processes, clearly no system stands alone. Systems must interact with other systems. Therefore, the problem of coupling, or input/output, must be acknowledged. Since the goal of this dissertation is to understand the computational abilities of recurrent networks, I will now focus on a class of systems with discrete couplings.

4.3 Iterated Function Systems

The preceding section outlined several basic ideas from the theory of dynamical systems theory. These concepts are broad in scope and application. It is now time to focus on particular pieces of mathematical knowledge that will directly help us understand the behavior of recurrent networks.

This section will provide a brief overview of a relatively new branch of mathematics known as iterated function theory. Barnsley origninally developed the foundational work (1988) as a method of describing the limit behavior of systems of transformations. The limit behavior of linear or affine transformations can be determined by examining eigenvalues of the transformations. The trajectories can be either fixed points or limit cycles.[9] While the effects iterating linear systems had been fully mapped out by this time, only recently did anyone consider the case of multiple linear transformations in parallel, i.e., $f(x) = \bigcup_{i=1}^{n} f_i(x)$. Such systems have now been shown to be a generalization of Cantor's "middle third" sets. This section will also describe an alternative method of approaching

9. Another regime exists, known as quasiperiodicity, occurs in systems composed of multiple limit cycles with irrational pairwise winding ratios.

the attractors defined by IFSs. Known as the random iteration algorithm, or the chaos game, it can produce sequences of points dense in the attractor by creating a random sequence of transformations. Finally, this section addresses the effects of noncontractive transformations on the IFS theory.

4.3.1 Basic Iterated Function Systems Theory

Recurrent networks share many similarities with the iterated function systems of Barnsley, but they also possess several radical differences. The iterated function systems responsible for the fractal structures seen in the Sierpinski triangle and other fractals (Figure 20) are defined as a finite set of contractive mappings over a metric space. What makes IFSs so fascinating is that although the limit behavior of a single transformation is just a point, the limit set over the union of the transformations can be an extremely complex set with recursive structure.

A metric space (X, d) is the combination of a set X and distance metric $d: X \times X \to \Re$. Consider, for illustrative purposes, the unit square and Euclidean distance function as our metric space. The three functions in Equation 24, ω_1, ω_2, and ω_3, map the unit square into three of the four quadrants of the unit square.

$$\begin{aligned} \omega_1((x, y)) &= (0.5x, 0.5y + 0.5) \\ \omega_2((x, y)) &= (0.5x, 0.5y) \\ \omega_3((x, y)) &= (0.5x + 0.5, 0.5y) \end{aligned} \qquad \text{(Eqn 24)}$$

If we take any one of them, ω_1 for instance, we find the limit of iterating this function on any point in the unit square exists and is the point (0,1). You can verify this by applying ω_1 to (0,1) and showing that it returns (0,1). This is a fixed point. It is an attracting fixed point since iteration draws other points closer to it over time. The other transforms have limits of (0,0) and (1,0).

Given that each transform, on its own, will go to a fixed point, what will the system of transforms, as a whole, do over time? In other words, we are interested in the limit behavior of iterating the function $\Omega(x) = \bigcup_{i=1}^{3} \omega_i(x)$. A system of three fixed-point transformation does not necessarily collapse into a single point. For instance, assume that the fixed point attractor A is $\{(x, y)\}$. Since A is a fixed point, $\Omega(A) = A$. But, $\Omega(A) = \{\omega_1((x, y)), \omega_2((x, y)), \omega_3((x, y))\}$, which only equals A in degenerate cases of ω. In general, it can have attracting set with infinite recursive structure. A single transformation shrinks the entire limit set into a one-fourth sized copy of the original, while the union operation pastes the different copies together to form the original image again. This process is illustrated in Figure 22.

The attractor structure depends upon the transients of the transformation. In other words, the limit behavior of the individual transformations plays only a minor role in determining the emergent shape of the attracting set. A simple example will demonstrate this

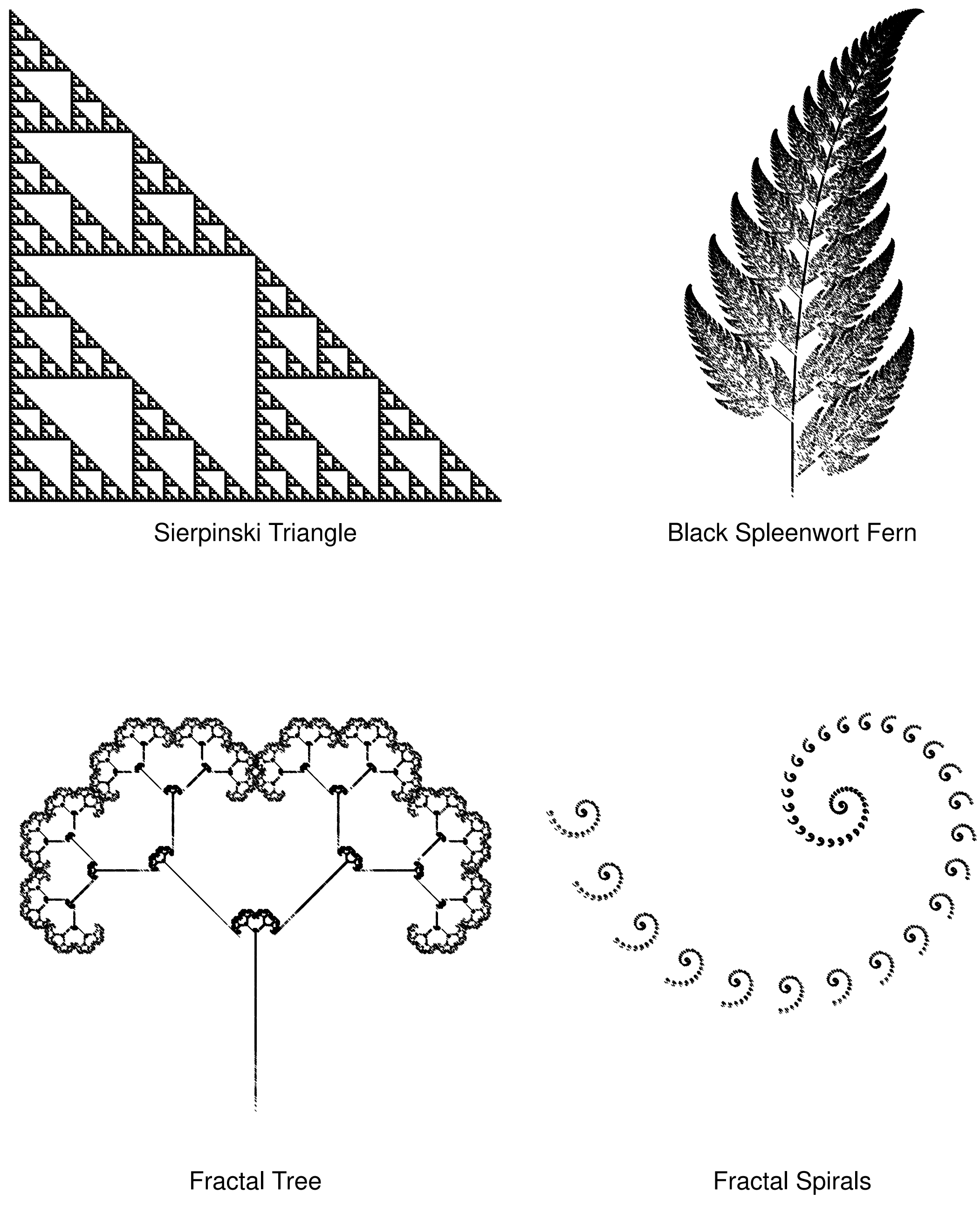


Figure 20: Several fractal sets generated by iterated function systems.

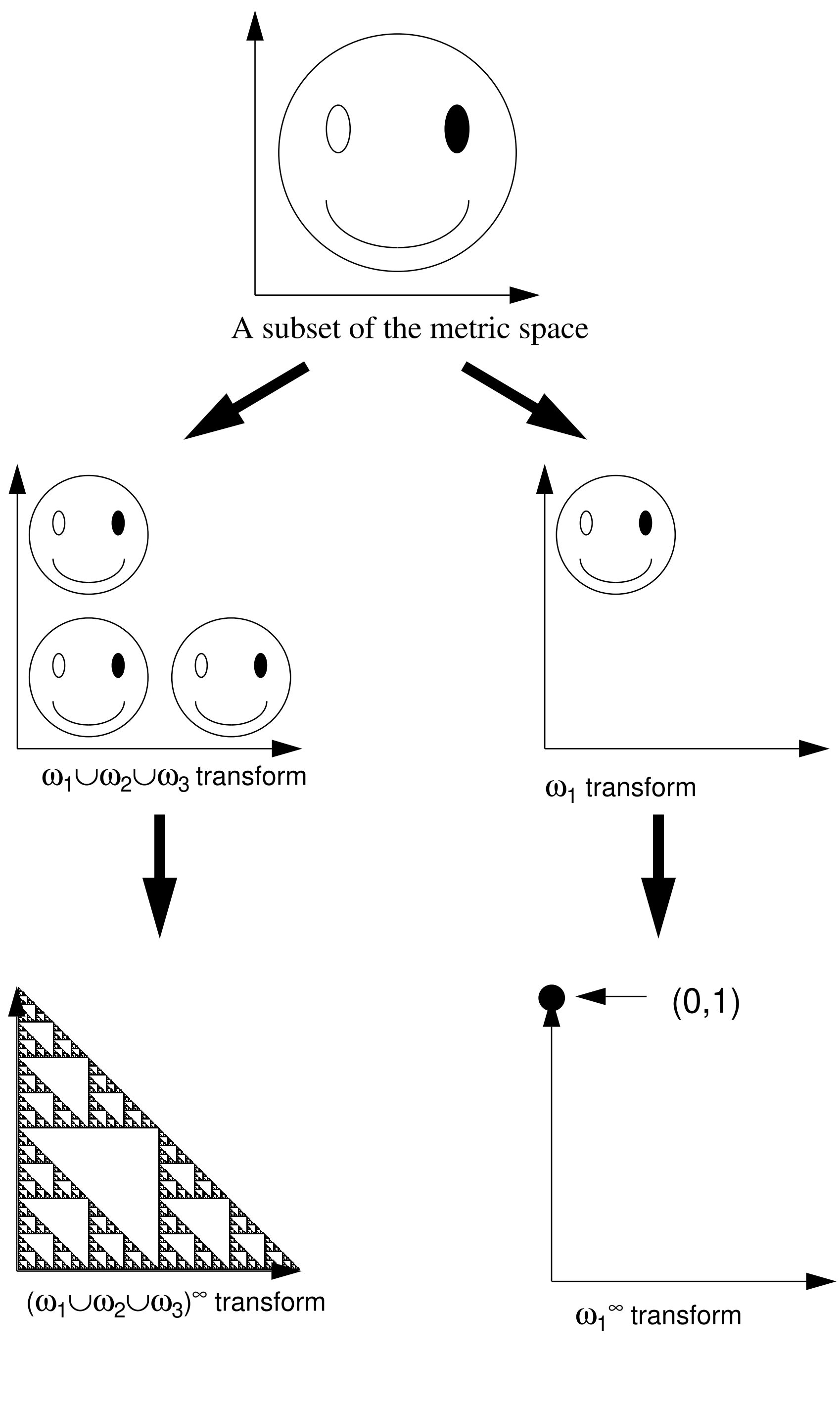


Figure 21: The difference between the limit of a single transformation (a point) and the limit of a collection of transformations (a Sierpinski triangle). The actual affine transformations are defined in Equation 24. The infinity superscript denotes the limit of an infinite sequence of function compositions.

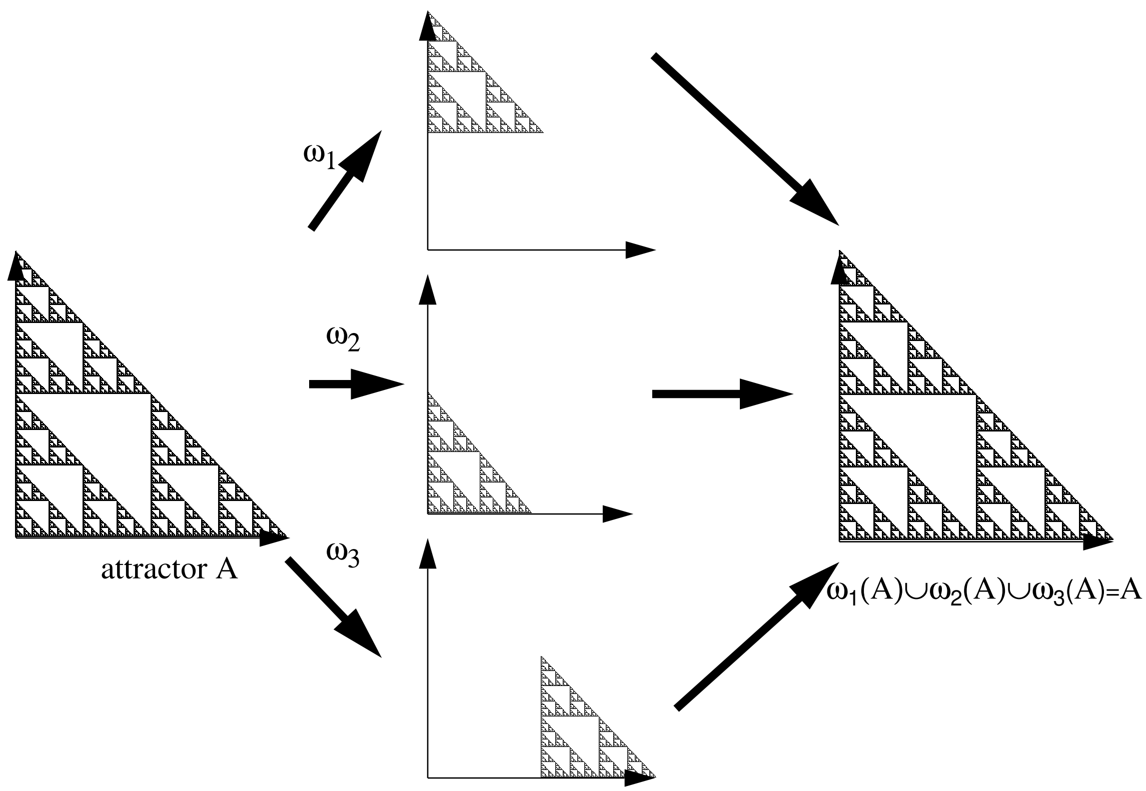


Figure 22: The individual transformations make three reduced copies of the attractor. Taking the union of the individual copies pastes together a the original attractor.

fact. Each of the three transformations that make up the IFS for the Sierpinski triangle has a limit point. These points, (0,1), (0,0), (1,0), are also the limits of the transformations in Equation 25.

$$\omega_1((x, y)) = (0, 1) \qquad \omega_2((x, y)) = (0, 0) \qquad \omega_3((x, y)) = (1, 0) \quad \text{(Eqn 25)}$$

The limit set resulting from iterating this set of transformations is the finite set {(0,1),(0,0),(1,0)}. These three points fall short of the uncountable number of points comprising the Sierpinski triangle.

Since an IFS is a dynamical system, one could ask questions about its stable and unstable behaviors. It turns out that IFSs have a single basin of attraction. An important concept for the understanding this fact of iterated function systems is the Hausdorff space. The Hausdorff space of a complete metric space (X, d) is the set of all compact subsets of X, excluding the empty set. A compact set is both closed and totally bounded. The distance between two points in Hausdorff space is given by Equation 26.

$$h(d)(A, B) = \sup \{ \inf \{ d(x, y) | x \in A \} | y \in B \} \quad \text{(Eqn 26)}$$

This metric measures the distance between two sets as the maximal distance between a point in A and the nearest point in B with respect to the original metric space. This function supports all of the properties of a metric such as if $A = B$, then $h(d)(A, B)$ is zero and the triangle inequality, $h(d)(A, B) < h(d)(A, C) + h(d)(C, B)$. Hausdorff space allows us to consider subsets of (X, d) as single points, a shift that lets us view the set of iterated transformations over the metric space as a single transformation in Hausdorff space. The complicated mapping and pasting can then be analyzed like any other dynamical system.

The theory laid out by Barnsley shows that there exist collections of functions whose iteration produces simple, fixed-point, dynamics in Hausdorff space. For instance, it can be shown that the Sierpinski triangle is a point in $((X, d)$ and that each of the three Sierpinski transformations reduce the distance between any starting subset of X and the Sierpinski triangle. This fixed point is also unique; there are no other attractors in $H((X, d))$. These conclusions can be proved without considering the individual transformations as long as certain properties are true (e.g. contraction).

4.3.2 Random Iteration

An interesting side effect of the unique attractor for Barnsley's IFSs is that approximations to the attractor are easy to construct. Random iteration, also known as the chaos game, produces attractor approximations very rapidly. Before describing the process of random iteration, I will first show how to play the chaos game. For this demonstration, that is needed is a piece of paper, a pencil, a six-sided die, and a ruler. First, mark three points on the piece of paper and label them one, two, and three. Now, place a fourth point inside the triangle formed by the three points. This will be your current point. The rest of the chaos game is simple: roll the die and make a new point one-half the distance between the current point and triangle vertex with the same label as the die (for die rolls greater than three, interpret four as one, etc.) Repeat these steps for many iterations. The result of this iterative process should be a figure with the same topological structure as the Sierpinski triangle described above. That is, the center triangle will be clear, the center of the smaller triangles will be clear, etc.

The chaos game can be formulated in terms of iterated function systems. Moving halfway between the current point and a vertex is a contractive affine transformation. The transformations used in the Sierpinski triangle in Equation 24 will transform a given point to a new point midway between one of three vertices at $(0, 0)$, $(1, 0)$, and $(0, 1)$. The chaos game, in other words, takes an initial point in the metric space and calculates a sequence of points by randomly selecting *one* IFS transformation at each iteration. It is easy to show that after each iteration the distance between the current point and the attractor geometrically decreases and approaches zero. This is due to the contractivity of the affine transformations, as the scaling factor for each successive iteration is less than one. Some points in this sequence do not lie anywhere near the attractor, although this sequence of points quickly settles down to a "stable" emergent pattern. Thus, the initial portion of the sequence is ignored as a transient because these points exponentially converge on the attractor. The resulting union of the sequence remainder is a finite approximation to the infinite point set of the attractor.

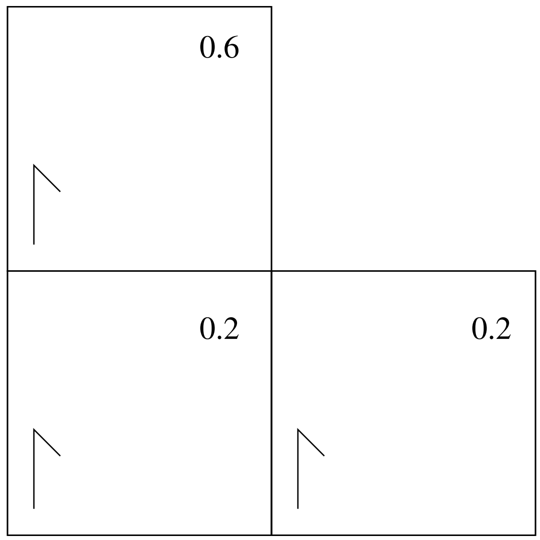
The description above is enough to generate useful approximations to many IFS attractors. Unfortunately, there are many IFSs that random iteration will fail to generate an approximation. The problem lies not in the algorithm of random iteration, but in the method for randomly selecting the transformations. Random selection of transformations results in coverage of the attractor, i.e., an infinite sequence of points generated this way will be dense in the attractor. While the system eventually reaches an attractor cover for any fixed ε, it may take a long time. The problem is that one part may be covered more than another. One way to ensure uniform coverage is to, as suggested by Barnsley, set the probability of selecting a transformation be proportional to the relative volume of transformation, otherwise the resulting volume of mapping the entire metric space by the transformation. In the Sierpinski triangle, each transformation compresses the original area of the unit square by one fourth. Thus, each transformation has a probability of one-third of being selected. The famous black spleenwort fern IFS (Figure 20) has four transformations, including a transform that maps the unit square into line forming part of the stem. The probability of this measure zero transformation is assigned a very small volume solely for the determination of their probabilities. Otherwise, these transformations will have a zero probability of being selected.

These probabilities are important since they affect the relative weighting of the resulting attractor. Figure 23 shows the effects of changing the probabilities of the transformations. The top pair of diagrams illustrates this effect on the Sierpinski Triangle IFS. On the left are the results of mapping the unit square[10] with each of the three transformations. The right diagram of the pair shows how an uneven distribution of transformations can "weight" some regions of the attractor more than others. Note that the more visited points do not occupy a connected region, they are distributed in a self-similar manner throughout the attractor. For the black spleenwort fern IFS, the proper selection of transformation probabilities is crucial to the emergence of the final form. By not weighting the large transformation more than the others, the infinite regress of subsystem is much more difficult to construct.

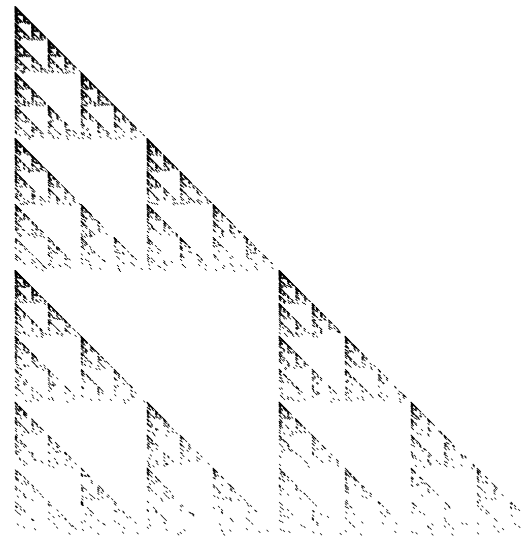
4.3.3 Summary

This section provided a brief overview of a relatively new branch of mathematics known as iterated function theory. This theory attempts to describe the limit behavior of systems of transformations. Mathematics has known the limit behavior of linear or affine transformations for many years. The trajectories can be either fixed points or limit cycles as determined by an examination of transformation's eigenvalues. OatOnly recently has anyone considered the case of multiple transformations. Specifically, the theory of iterated function systems applies to iterated mappings composed of multiple linear transformations, for example $\Omega(x) = \bigcup_{i=1}^{n} \omega_i(x)$. Such systems generalize the process that yields Cantor's "middle third" set, a set generated by recursively removing the middle-third of an interval and all of its subintervals. They are also responsible for the fractal structures seen in the

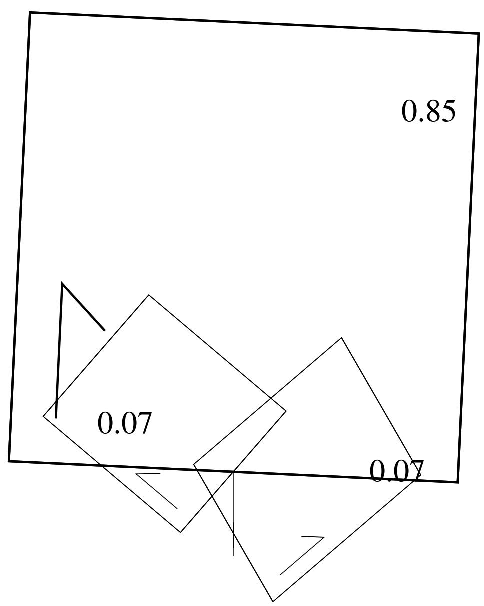
10. Actually, the unit square and a small arrow in the lower left corner. The arrow was added to indicate orientation.



Sierpinski Triangle
Transform Outlines
(Probabilities in upper right corner)



Sierpinski Triangle
Skewed Transform Probabilities
(Compare with Figure 20)



Black Spleenwort Fern
Transform Outlines

Black Spleenwort Fern
Equiprobable Transformation
(Compare with Figure 20)

Figure 23: Examples of attractors generated with different transformation probabilities.

Sierpinski triangle and other fractals (Figure 20). While each transformation of an IFS has very simple limit behavior, the limit set over the union of the transformations displays highly complex recursive structure. The key to understanding the limit behavior of IFSs was the Hausdorff space. Rather than concentrating on the state space, IFS theory tells us to look at the set of all closed sets of the original space. In this new space, IFSs display fixed point attractors whose basin of attraction is the entire state space. The proof of this theorem assumes that the transformations are contractive.

By directly implementing the definition of IFSs, one can construct images of the attractor. Repeated applications and unions of the contractive mappings will result in the attractor image, independent of the initial set. Because this method can consume an overwhelming amount of computational resources, an alternative method of approaching the attractors was discussed. Known as the random iteration algorithm, or the chaos game, it can produce sequences of points dense in the attractor by creating a random sequence of transformations. The random sequence is generated by assigning each transformation a fixed selection probability. The chaos game is not foolproof. The selection probabilities can influence the weighting of the resulting image. Likewise, the method of selection can affect the resulting image.

4.4 Symbolic Dynamics

While it is helpful to characterize systems in terms of their dynamics, other methods exist for describing the structure and regularities of dynamical systems. One way of examining an attractor of a system is through the state values it generates. However, this may obscure underlying regularities, especially for high dimensional state spaces. An alternative approach is through symbol dynamics. By reducing a trajectory from a sequence of state values to a sequence of discrete symbols many regularities can be found in symbol sequences of different dynamical systems (Bai-lin, 1989). These symbol sequences correspond to labels of state space regions. This is a very useful tool in that it allows one to abstract away from the details of the trajectories and focus on the gross dynamic behavior of the system.

The best way to introduce symbol dynamics is an example. Consider the one dimensional logistic map $x^{(t+1)} = \eta x^{(t)}(1 - x^{(t)})$ from population dynamics (May, 1976). This mapping has been studied extensively and will serve as an example throughout this section. Selecting values of η in $[0, 4]$ ensure that a trajectory started within the state space $[0, 1]$ will remain there. Values of η outside this region will produce systems whose attractors are at infinity. Note that the maximum next value of the logistic mapping occurs at $x = 1/2$. This value of x is very important. To the left of this value, the logistic mapping has a positive slope. Similarly, the slope is negative to the left of $x = 1/2$.

A symbol sequence can replace a trajectory evolved by this system in the following manner. If the state value is less than $1/2$, replace it with an **L** for left of center. If the state value is greater than $1/2$, replace it with an **R** for right of center. Otherwise, the state value is equal to $1/2$, so it is replaced it with a **C**. This scheme for the conversion of states to symbols is easily extended to other mappings. Simply partition the state space into regions

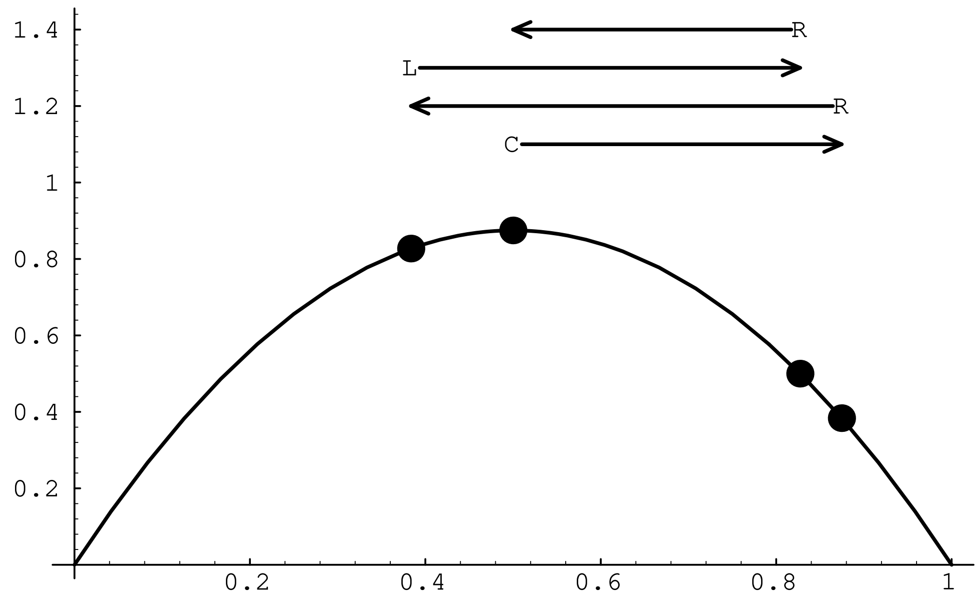


Figure 24: A superstable period four orbit (CRLR)$^\infty$ in the logistic mapping (r=3.49856). The trajectory starts at 0.5 (labeled C). The arrows indicate the resulting sequence of states and their symbolic labels.

with boundaries defined by the zero first derivatives of the mapping variable. In the logistic function, the first derivative of $f(x)$ is zero when $x = 1/2$.

The partitioning and symbolic dynamics of a system are best understood in terms of the following examples involving the logistic function. Consider the parameter setting $r = 5/2$; a parameter setting which causes the system to eventually settle on a fixed point attractor at $3/5$. This fixed point is to the right of the maximum. Thus, the symbol sequence for this attractor is **RRRRR**..., or more compactly[11] **R**$^\infty$. At the higher value r=3.3, the trajectory oscillates between 0.823603... and 0.479427..., generating a symbol sequence of (**RL**)$^\infty$ (or equivalently (**LR**)$^\infty$).

Sometimes the boundary points, those points whose slopes are zero, are part of the underlying attractor. When this occurs, the periodic attractor is *superstable*[12]. The sequence (CRLR)$^\infty$ indicates a superstable orbit of period 4 that starts at $x = 1/2$ and returns to this point after four iterations. In Figure 24, the periodic points are plotted on the logistic parabola for r=3.49856. The winding arrow at the top of the graph indicates the trajectory. During this cycle, the trajectory will visit one point to the left of center and two points to

11. I use $^\infty$ rather than the Kleene star (*) to emphasize that the attractor generates an infinite sequence.

12. Since the maximum of the logistic function is at $x = 1/2$, the superstable orbits of the logistic function must include $1/2$.

Table 3:Attractor Symbol Sequences for the Logistic Map

Period	Regular Expression
1	$\mathbf{C}^{\infty}$
2	$(\mathbf{CR})^{\infty}$
4	$(\mathbf{CRLR})^{\infty}$
8	$(\mathbf{CRLRRRLR})^{\infty}$
2^{n}	$(A)^{\infty}$
2^{n+1}	$(AA')^{\infty}$

the right of center. The convention of spliting the state space at the maximum often aids analysis, but is hardly a requirement. The symbol mapping simply must partition the state space into disjoint regions. Feigenbaum used this particular decomposition to calculate the ratio of successive period doubling parameters. The superstable orbit was a convenient landmark (just solve $f^{\circ n}(\frac{1}{2}) = \frac{1}{2}$) found in each period region.

Now consider the Bakers' mapping $x_{t+1} = 2x_t \bmod 1$. The trajectory of this system is entirely dependent upon the selection of the initial state. Multiplying by two repeatedly left-shifts the binary encoding of the first value of x, while the modulus throws away any bits to the left of the decimal point. Since there are many more irrational numbers in the range $(0, 1)$ than rational numbers, the trajectory is most likely aperiodic and appears random (due to the sensitivity to initial conditions), but is nonetheless deterministic. Therefore, any symbol subsequence is possible with the right choice of initial conditions, a property compactly described as $(R|L)^{\infty}$. This symbol dynamic is common as it is at the root of most chaotic systems: the Bakers' map is isomorphic to the logistic function $4x(1-x)$ and shares the same symbolic dynamics ((Schroeder, 1991) and see Figure 16). A summary of the supercritical attractor sequences for low power of two periods is given in Table 3. The underlying pattern is simple: double the sequence of the previous stage, A, and replace the **C** at the beginning of the second half, A', with an **R** if the number of **R**s in A is even, otherwise transform it into a **L**.

The first part of this chapter discussed the merits of characterizing systems in terms of their dynamic flows or iterated mappings over continuous spaces. Symbol dynamics, however, provides another method for describing the structure and regularities of dynamical regimes. This method reduces the trajectory through continuous space into a sequence of discrete symbols. While one may believe that such discretizations would obscure trajectory complexities, this approach allows an observer to abstract away from the details of the trajectories and focus on the gross dynamic behavior of the system. Fixed points show up as infinite sequences of single symbols. Limit cycles produce repeated patterns. Chaotic systems generate "random" sequences that unpack the symbols composing their initial conditions. Period-doubling bifurcations affect superstable periodic symbol sequences by appending a modified copy of the periodic subsequence: the **C** is replaced according to the parity of **R**s in the original subsequence. In later chapters, these properties will help us understand the computational capabilities of recurrent networks.

4.5 Summary

This chapter reviewed key concepts from dynamical systems theory. The concepts of attractor and repellor were introduced to characterize the long term behavior of dynamical systems. The long term behavior could also be described either as fixed point, periodic, or aperiodic (chaotic). Two methods were described for identifying the type of attractor: Hausdorff dimensionality and period doubling routes to chaos as seen in bifurcation diagrams. In addition, I discussed Iterated Function Theory, a set of theoretical results applicable to understanding the dynamic behavior of sets of transformations. The final section of the chapter briefly introduced symbolic dynamics, a tool that will become useful in Chapter VI.

While it is easy to describe the processing occurring in feed-forward networks without referring to their dynamical properties, recurrent networks are intimately tied to such descriptions. In the next chapter, I will return to recurrent networks and show how finite state machine interpretations may inhibit our understanding of the processing occurring in recurrent networks.

CHAPTER V

RECURRENT NETWORKS AS ITERATED FUNCTION SYSTEMS

5.1 Introduction

From Chomsky's generative grammars to Newell and Simon's physical symbol systems, models of human intellectual competence have possessed the potential to store and act upon any one of an infinite collection of internal representations. The construction of a working computer to implement these representations appears to require the cooperation of memory and logic units with temporally stable states. The theme of this chapter, and the following chapters, could be summarized as the following question: Do other sources of computation exist that do not rely on explicit and expensive state storage? My search for computational models grounded in physical devices has uncovered this behavior in recurrent neural networks. While many claimed to have tackled and solved the problem of universal computation in recurrent networks, the results seem too contrived, often exploiting trivial features of the recurrent network models. I claim that this body of work has addressed a different question altogether. These configurations for computation merely solved the problem of *designing* for computation. This line of inquiry has neglected the more difficult problem of identifying computations as they occur, without the intentional intervention of a designer.

I will show that deterministic models of computation, like deterministic finite state machines, can fail to capture the internal processing regularities of recurrent networks. This will be especially true if those state machine explanations arise from discretization of internal states. A new method of describing recurrent network behavior is necessary. I believe that the theory developed for the analysis of iterated function systems (IFS) (Barnsley, 1988) provides a solid framework for the understanding of recurrent networks. This mathematical theory, introduced in Chapter IV, describes and predicts the properties of iterated systems of transformations. Others have explored the connection between recurrent networks and IFSs (e.g. (Stark, 1991) and (Stucki & Pollack, 1992)), yet this work has mainly focused on the implementational issues associated with mapping IFSs into recurrent networks. Such implementational issues are not the subject of this dissertation. Rather, the IFS formalism will serve as an explanatory tool for understanding both the dynamics and information processing behaviors of recurrent networks.

The goal of this chapter is to highlight the relationship between recurrent neural network models and iterated function systems. First, I will introduce the notion of information processing (IP) state and use it to motivate an examination of the regularities occurring in recurrent networks. I will then explain how the common notion of deterministic information processing fails to hold for certain deterministic recurrent neural networks whose dynamics are sensitive to initial conditions. Iterated function systems theory, as developed by Barnsley and others, is a method of describing the dynamic behavior of systems of transformations. This chapter centers on the link between recurrent networks and iterated function systems. To apply IFS theory to recurrent networks, we must identify the effects of noncontractive transformations on the IFS theory. The final section will address the application of IFS theory to the problem of understanding the internal state dynamics of SCNs. This link points to *model independent constraints* on recurrent network state dynamics that explain universal behaviors of recurrent networks such as internal state clustering which have shown up in many models (e.g. (Elman, 1990; Pollack, 1990)). Although this work specifically focuses on a particular recurrent network architecture, SCNs, the results apply to other forms of recurrent networks, like SRNs, Jordan networks, and RAAMs.

5.2 Recurrent Networks as State Machines

Feedforward neural networks process information by applying fixed transformations from one representation space to another (Chapter I). Recurrent networks, on the other hand, process information quite differently. While thinking of the recurrent network as an instantaneous feedforward network may help in developing learning schemes (Chapter 3), this conceptualization fails to help us understand the information processing carried out by the network. To understand recurrent networks one must confront the notion of state since recurrent networks perform *iterated transformations on state representations*.

Inspired by the pioneering work of McCulloch and Pitts (1943), several simulation proofs of universal computation in recurrent networks have appeared over the last few years. McCulloch and Pitts linked simple models of neural function to first-order logic. Franklin and Garzon (1988) constructed a Turing machine tape and control from an infinite pool of neurons. The "neuring machine" network (Pollack, 1987b) consisted of two units: a multiplicative unit and a step unit. The tape was stored in the binary expansion of the continuous state employed by the processing units. Siegelman and Sontag (1991) describe a network of approximately one thousand processors that computes a universal partial-recursive function by combining the two operations described by Pollack into a single threshold unit with a linear response for a given input range. MacClellenan (1992) showed how an infinite state linear system can perform universal computation. While each of these projects supports the representational adequacy of recurrent networks in general, they are insufficient in one meaningful way. They support this claim by showing that recursive computation can arise by *design*. These projects, however, provide little insight in the *detection* of computation occurring in a system. In other words, they do not tell us how to recognize computation when it is happening before our eyes nor how it could arise through evolution.

5.2.1 What Are States?

The first step in ascertaining the computational power of a system is to identify its information processing (IP) states. The attribution of computational behavior to a system traditionally rests on the discovery of a mapping between states of the physical (or dynamical) sys-

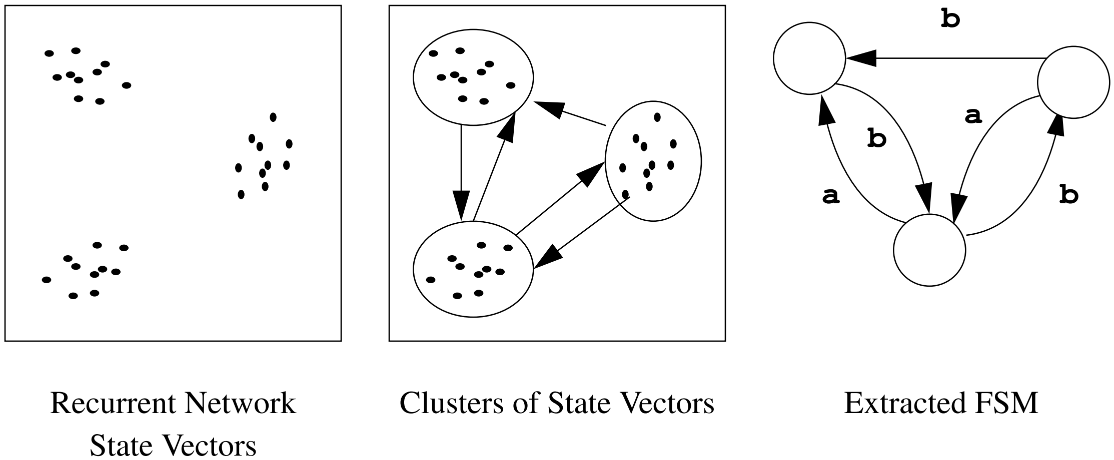


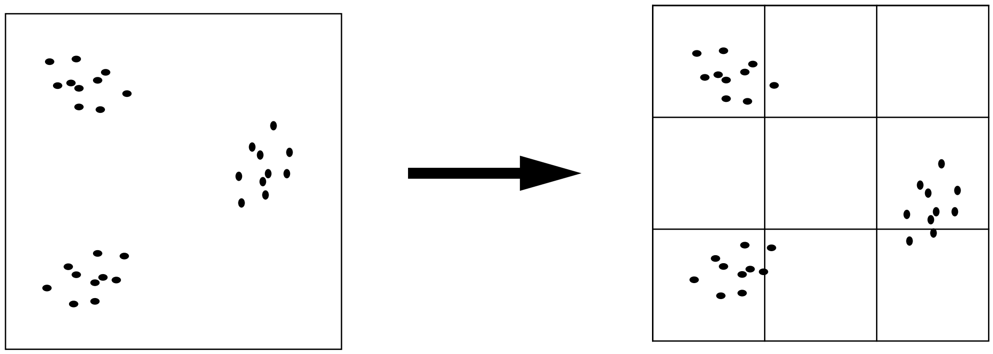
Figure 25: Finite state automata extraction from recurrent neural networks using statistical clustering.

tem to the IP states of the computational model (Ashby, 1956). Consider the (trivial) case of identifying computation in an electronic computer. Certain voltage levels of particular wires in a CPU, for instance, are mappable across time and context to the bits processed by the idealized random access machine.

Many researchers studying the computational power of recurrent networks have followed this tradition. Some researchers have recognized this difference and have suggested parallels between recurrent networks and various deterministic automata (Servan-Schreiber, Cleeremans, & McClelland, 1988; Giles et al., 1992; Giles et al., 1992a; Watrous & Kuhn, 1992). Servan-Schreiber et al. used hierarchical clustering to analyze the recurrent state representations their networks found while learning a regular language. The work of Giles et al. (1992b) described the behavior of second and higher order recurrent networks trained on positive and negative examples of unknown grammars with finite state machines. Watrous and Kuhn have reported a similar mechanism for finite state machine extraction. Giles et al. (1992b) extended this work to the induction of a finite state stack controller for a neural network pushdown automaton.

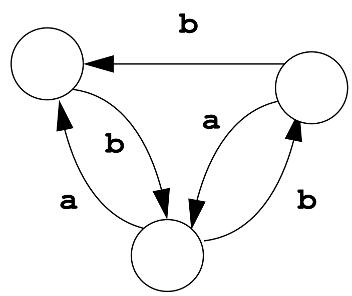
These researchers clustered internal state vectors and drew correspondences between these clusters and the deterministic state machine generating the training data (e.g. (Servan-Schreiber, Cleeremans, & McClelland, 1988)). Servan-Schreiber et al. trained an SRN to predict the next symbol from a Reber grammar generator. After training the network and collecting state vectors, they examined cluster diagrams extracted from the recurrent activations. The cluster divisions appeared to distinguish the five states of the Reber generator. (In the last section of this chapter, I will return to this experiment and present an explanation for this phenomenon.)

This work was extended under the title of dynamic state partitioning (Giles et al., 1992a). Dynamic state partitioning consists of four steps (see Figure 26). First, the state space trajectory of the network must be partitioned into clusters of states. These partitions



State	Input: a	Input: b
11	13,23	31,32
12	-	-
13	-	11,12
21	13,23	31,32
22	-	-
23	-	11,12
31	11,12	13,23
32	11,12	13,23
33	-	-

FSM Transitions From Discrete
Quantized State Vectors



Minimized FSM

Figure 26: Finite state automata extraction from recurrent neural networks using quantized state vectors.

are generated by arbitrary quantizations of the state space, such as the nine regions in Figure 26. Next, a transition table is constructed by observing alphabetic transitions as the recurrent network dynamic propels the current state vector from one discrete region to another. Then the clusters and transition table are combined to form a state graph. This graph is then minimized using standard finite state machine minimization algorithms (e.g. (Hopcroft & Ullman, 1979)). The result is a minimized finite state machine interpretation of the recurrent network's internal dynamics.

While Giles and his collaborators use arbitrary partitions of state space to build clusters, others (Watrous & Kuhn, 1992) adapted their partitions during the extraction process. They start by collecting the network's state response to a collection of strings. For each hidden unit, a zero-width histogram[1] is computed and then partitioned to bisect intervals between adjacent peaks. State labels are created by concatenating histogram interval numbers for each hidden unit. If the procedure records a transition from the same state on the same input symbol to two different states, it collapses the two states by deleting or moving histogram boundaries. Accept and reject states are marked according to the output of the network. Like Giles et al. (1992), Watrous and Kuhn applied traditional state minimization techniques to the resulting state transition table.

Each method shares the same problem of multiple inductions; a single network can produce many state machine interpretations. First, a poor initial partition can affect the extraction process. Both research teams have sidestepped this issue by rerunning the dynamic state partitioning procedure several times under different parameter settings in search of the minimal interpretation. Second, if hierarchical clustering is applied, the order of clustering can affect the resulting cluster relationships.

The traditional approach of drawing correspondences between physical states and IP states is not without its problems, especially if the modeler is looking for complex computation. As the complexity of the computation increases, the number of IP states also increases. The modeler must measure the current state of a process with sufficient resolution to justify the mapping from physical device to IP state. The problem explodes when the modeler switches mechanism classes, such as changing from finite state machines producing regular languages, to push-down machines with the ability to generate and accept context-free languages. In models capable of recursive computation, the IP state can demand unbounded accuracy from the modeler's measurements. According to the traditional approach, identifying nontrivial computation might require partitioning the state space into an infinite number of cells.

The researchers above implemented automatic search mechanisms whose goal was the discovery of an isomorphism between network state behavior and the simplest automaton that mimics its behavior. Recent work by Crutchfield and Young (1989) suggested that IP states can be assigned without examining the internal dynamics. Their work focused on the problem of finding models for physical systems based solely on measurements of the systems' dynamical state. Rather than assuming a stream of noisy numerical measurements, they constructed models from periodic samplings with a single decision boundary. Unlike mathematically describable numerical measurements, the binary sequence they collected requires a computational description. In other words, the description should take the

1. Essentially a union operation that keeps track of the number of identical elements.

form of an automaton that can generate the observed sequence. They claimed that the minimal finite state automaton induced from this sequence of discrete measurements provides a realistic assessment of the intrinsic computational complexity of the physical system under observation. To test this claim, Crutchfield and Young generated binary sequences from nonlinear dynamical systems such as the iterated logistic map and the tent map. These systems have the property that manipulation of a single global parameter cause qualitative behavioral changes known as period doubling bifurcations (Chapter 4). These bifurcations appear when slight changes in the control parameter force a period n oscillation into a period $2n$ oscillation. A period n oscillation can be described with an n state automata.

An important distinction can be drawn between the methods of Crutchfield and Young with those used by the multitude of researchers studying recurrent networks. Dynamical systems theory, particularly the study of systems producing chaotic behavior, shows that significant "state" information can lay buried deep within the system's initial conditions. This sensitivity to initial conditions implies that often best way to "measure" a system's state is to simply observe its behavior over long periods and retroactively decide the state. Other methods of system identification employ this approach. For instance, Takens' (1981) method of embedding a single dimensional measurement into a higher dimensional space can provide a judgment of the underlying system's dimensionality. Crutchfield and Young extended a similar idea of their own to handle the computational understanding of systems by reconstructing the computational dynamics of the system from the sequence of boolean measurements. The difference between these approaches rests upon the availability of state information.

5.2.2 The Counterexample

Two methods of assessing the computational ability of a system were described above. One relies on the identification of an isomorphism between the IP states of a computational model and the observables of a system. The other method induces a computational model from a sequence of behaviors. The difference between these two approaches rests upon the assumed availability of state information. The following example shows how clustering of observable states can lead to the extraction of nonexistent deterministic states from recurrent networks. The basis of this claim rests upon the definition of information processing state.

Information processing (IP) state is the foundation underlying automata theory. Two IP states are the same if and only if they generate the same output responses for all possible future inputs (Hopcroft & Ullman, 1979).

This definition is the fulcrum for many proofs and techniques, including finite state machine minimization. Any FSM extraction technique should embrace this definition. It also grounds the standard FSM minimization methods and the physical system modeling of Crutchfield and Young (Crutchfield & Young, 1989). Clustering methods, as we shall see, cannot establish the existence of IP states when dealing with systems whose dynamics are sensitive to initial conditions.

Some dynamical systems exhibit exponential divergence for nearby state vectors, yet remain confined within an attractor. This behavior is known as chaos. If this divergent behavior is quantized, it appears as nondeterministic symbol sequences (Crutchfield &

$$x \leftarrow 4x(1-x) \qquad O(x) = \begin{cases} A & x < 0.5 \\ B & x > 0.5 \end{cases}$$

$$x \leftarrow 2x \bmod 1 \qquad O(x) = \begin{cases} A & x < \frac{1}{3} \\ B & \frac{1}{3} < x < \frac{2}{3} \\ C & \frac{2}{3} < x \end{cases}$$

$$x \leftarrow 3.68x(1-x) \qquad O(x) = \begin{cases} A & x < 0.5 \\ B & x > 0.5 \end{cases}$$

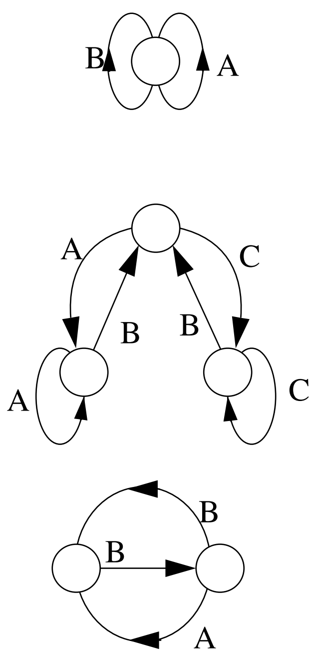


Figure 27: Examples of deterministic dynamical systems whose discretized trajectories appear nondeterministic.

Young, 1989) although the underlying dynamical system is completely deterministic (Figure 27). In the case of the logistic parameter with $\lambda = 4$, the chaotic dynamics produces random symbol-dynamics sequences. At $\lambda = 3.68$, the mapping prevents subsequences of the form AA from occurring. The resulting generator reflects this restriction.

Sensitivity to initial conditions can also occur in recurrent networks. Figure 19 illustrates the bifurcation diagram of a particular 1-3-1 feedforward network with weights scaled by a control parameter. Because of the unimodal shape of the network's transfer function, iteration yielded the universal bifurcation scheme. For purposes of illustration, consider a recurrent network with one output and three recurrent state units. The output unit calculates a threshold at zero activation for state unit one. That is, when the activation of the first state unit of the current state is less than zero then the output is A. Otherwise, the output is B. Equation 27[2] presents a mathematical description. $S(t)$ is the current state of the system $O(t)$ is the current output.

$$S(t+1) = \tanh\left(\begin{bmatrix} 2 & -2 & 0 & -2 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & 2 & -1 \end{bmatrix} \cdot \begin{bmatrix} S(t) \\ 1 \end{bmatrix}\right) \qquad O(t) = \begin{cases} A & S_1(t) < 0 \\ B & S_1(t) \geq 0 \end{cases} \qquad \text{(Eqn 27)}$$

Figure 28 illustrates what happens when the network starts from a large set of initial conditions. The point in the upper left-hand state space is actually a thousand individual points all within a ball of radius 0.01. In one iteration, these points migrate down to the lower corner of the state space, elongating along one dimension. After ten iterations the original ball shape is no longer visible. After twenty, the points are spread along a two dimensional sheet within state space. By fifty iterations we see the set of network states reaching their extent in state space. The divergence of the three thousand points is summarized in Figure 29. The graph shows how the maximum pairwise distance between points increases exponentially over many iterations. Leveling off occurs because the points achieved coverage of the attractor, a common phenomena in chaotic systems. This demonstration would reproduce the same results for neighborhoods around every point on the attractor. The vast set of unique output sequences that follow from this "state" without any assistance from disambiguating inputs clearly invalidate the common assumption that all points in a state space neighborhood within a *deterministic* system share (or implement) the same IP state. Specifically, the IP states of finite state machines do not necessarily map onto neighborhoods in recurrent network state space though those networks have deterministic dynamics.

5.3 Recurrent Networks as Iterated Function Systems

In this section, I will discuss the relative merits of several recurrent network architectures in applications involving finite input pattern sets. For many applications this is really no restriction at all. Recall that in the earlier formal language acquisition example (Section 5.2.1 on page 74) we assumed a finite alphabet. Any time we use a recurrent network in a symbolic processing setting, the resulting system will fall under this characterization. Each symbol from the alphabet will have one and only one vector associated with it. For instance, the formal language system usually operates on binary symbol sequences. These sequences are implemented as two vectors **`0`** $= \{1, 0\}$ and **`1`** $= \{0, 1\}$ and the network will never see any other vectors on its input buffer. The ubiquity of these applications forces us to understand the processing capabilities of recurrent networks in this context.

Obviously, the network fails to use the entire n-dimensional vector space over the state units. By defining state space as the set of reachable internal state vectors, the state space can be constructed by applying every finite input sequence to the network's initial

2. $A \cdot \begin{bmatrix} B \\ 1 \end{bmatrix}$ indicates the inner product between A and the vector B with a 1 appended to that vector.

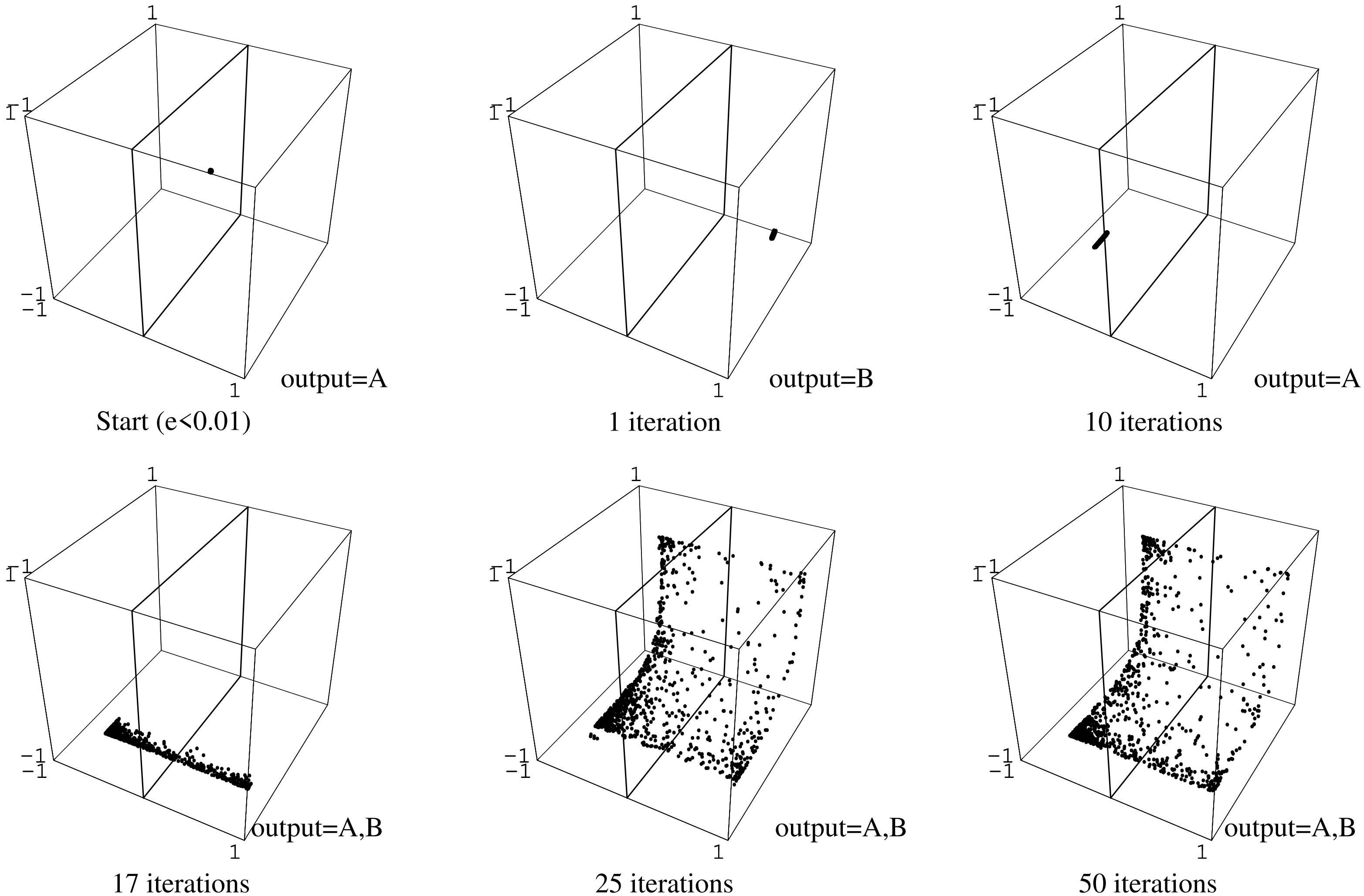


Figure 28: The state space of a recurrent network whose next state transitions are sensitive to initial conditions.

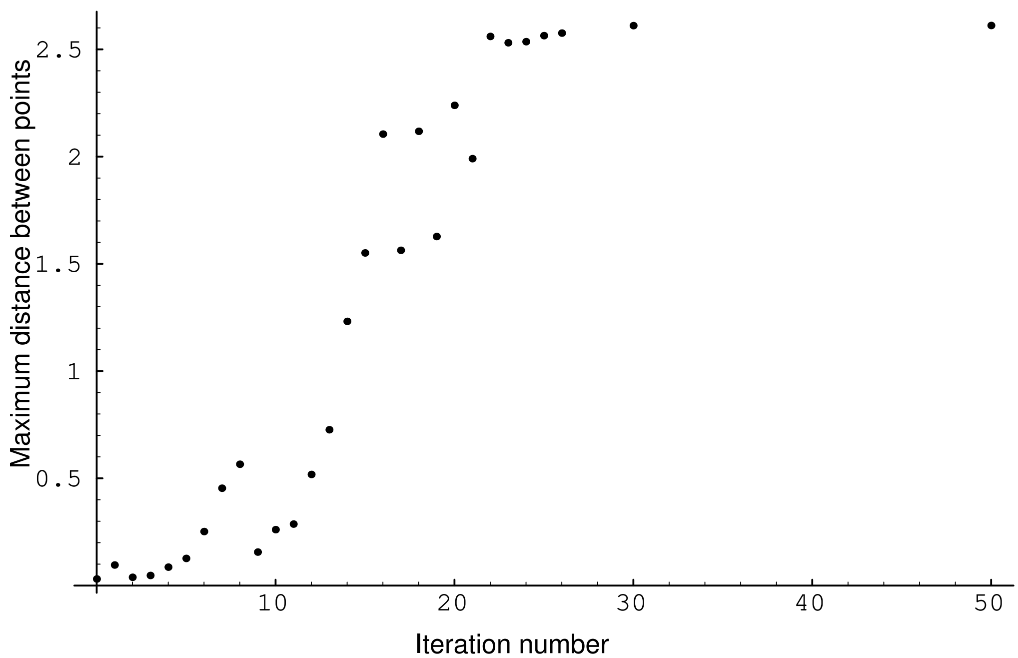


Figure 29: The exponential divergence of a 3000 points across the recurrent network attractor.

state and taking the union of the resulting vectors. One would hope that regularities in state space suggest the information processing carried out by the network. To understand these regularities I will describe the state space behavior of recurrent networks using the vocabulary of iterated function systems.

By adopting the IFS formalism, it is easy to see that recurrent networks can produce state spaces of infinite diversity. All that is necessary is that the input-selected transformations cover sufficiently measurable regions of state space. Otherwise the input will simply "load" current state prestored next state. The infinite collection of states is a product of the transients associated with leaving one transformed space into another. We can also represent a finite state automaton, like a recurrent network, as a set of input-selected state transformations. These transformations lack transients, however. They directly map to the destination state, like the transformations in Equation 25 that mapped directly to the limit points of the Sierpinski triangle transforms.

With recurrent networks, a generative force, rather than transients, is at work. The noncontractivity of the transformations allows the existence of transformations with multiple basins of attraction. Depending upon initial conditions, such as initial state or input sequences, the network can find itself selecting one attractor from a collection of possible attractors. External and noninput-related environmental effects could affect these systems, thus altering their behavior without changing the dynamics.

5.3.1 Effects of Noncontractivity

To simplify the mathematics of IFS theory, Barnsley assumed that the individual IFS transformations are contractive. That is, the distance between two transformed points is always less than the distance between the original points. The benefit of this assumption lies in his proof of the existence of a limit point in Hausdorff space for the IFS. The contractivity of the underlying transformations implies that the union of these transformations will reduce the size of any set in Hausdorff space.

The transformations we see in recurrent networks are, unfortunately, not always uniformly contractive as in an IFS. It was the noncontractivity of the earlier network that produced the sensitivity to initial conditions (Figure 28 and Equation 27). Consider the simplest case of a recurrent network that transforms state vector s with weight matrix W and a sigmoid transfer function, g. Here, for inputs near the solution of $Ws = 0$, the derivative of g is maximized. If the eigenvalues of W are greater than four ($= \frac{1}{g'(0)}$), then the dynamic in this region of state space is expansive. As Ws moves away from zero, the squashing nature of the sigmoid takes over to force points closer together, rather than separating them. I showed a recurrent network, earlier in this chapter, with a mapping that is both expansive and contractive, a combination that produces an effect known as sensitivity to initial conditions. One might argue, that network was a contrived example to display sensitivity and is not indicative of typical behavior. The following example addresses this issue by looking at the transformations actually learned by recurrent networks.

Contractive and noncontractive iterated function systems exhibit some important differences. The most significant is the nature of the fixed points. Contractive IFSs always possess a unique attractor and no other fixed points. The number and variety of fixed points are unlimited in noncontractive IFSs. They can have many attracting and repelling fixed points. Even chaos is possible. While the original iterated function systems are not sensitive to initial conditions[3], we have already seen a noncontractive IFS that is sensitive to its initial conditions (Figure 28). In addition, the random iteration algorithm produces trajectories dense in the attractor for Barnsley's formulation. The possibility of multiple basins of attraction prevents us from making the same claim in the noncontractive case. These basins are the same attractors that Hopfield, and many others, used to implement content addressable memory (Hopfield, 1982; Hopfield, 1984).

5.3.2 Internal States in Recurrent Networks

The recurrent network selected for this demonstration is the sequential cascaded network (SCN) (Figure 12 on page 36), a recurrent connectionist architecture (Pollack, 1991) that

3. As witnessed by the effectiveness of the chaos game in producing attractor approximations.

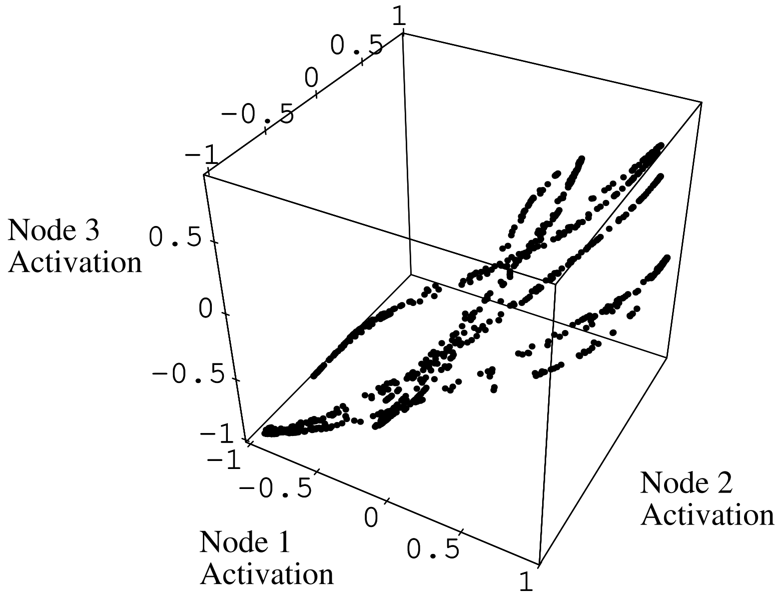


Figure 30: Internal states of the ABCD multiassociative memory

extends the cascaded network of Pollack (1988). An SCN can be described by two dynamic equations

$$O^{(t)} = g\left(\left(W \cdot \begin{bmatrix} S^{(t)} \\ 1 \end{bmatrix}\right) \cdot \begin{bmatrix} I^{(t)} \\ 1 \end{bmatrix}\right) \qquad S^{(t+1)} = g\left(\left(V \cdot \begin{bmatrix} S^{(t)} \\ 1 \end{bmatrix}\right) \cdot \begin{bmatrix} I^{(t)} \\ 1 \end{bmatrix}\right) \qquad \text{(Eqn 28)}$$

Where $O^{(t)}$ is the output vector at time t, $S^{(t)}$ is the state vector, $I^{(t)}$ is the input vector, and W and V are three-dimensional weight matrices describing state-to-state and state-to-output transformations performed by the SCN.

Pollack initially applied SCNs to the task of formal language recognition where the networks apparently induced regular languages (Pollack, 1991). The internal states of these dynamical recognizers, however, were *not* finite. Dimensional analysis revealed a fractal dimension of approximately 1.4 for the set of all possible internal states, clearly indicating an infinitely structured set (Mandelbrot, 1982). The SCN also served as the foundation of multiassociative memory (MM) (Kolen & Pollack, 1991), a system of storing multiple associations to a single stimulus by generating sequences of associations. In one MM task, a simple mapping of four inputs to two different outputs each yielded collections of internal states similar to those produced by the dynamical recognizers (Figure 30).

What properties of the SCN produce these state spaces? To answer this question, I will rewrite the SCN's dynamic equations to resemble IFS transforms. Assuming a finite set of input vectors corresponding to the predefined input set, we may write the SCN equations in terms of transformation selected by the input (Equation 29).

$$O^{(t)} = g(W_{\mathbf{I}} \cdot \begin{bmatrix} S^{(t)} \\ 1 \end{bmatrix}) \qquad S^{(t+1)} = g(V_{\mathbf{I}} \cdot \begin{bmatrix} S^{(t)} \\ 1 \end{bmatrix}) \qquad \text{(Eqn 29)}$$

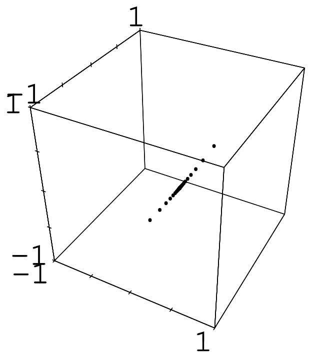
In Equation 29, $W_{\mathbf{I}}$ and $V_{\mathbf{I}}$ are two dimensional matrices selected by input pattern **`I`**. Dropping the sigmoid on the state units and insisting on a finite set of acceptable inputs, the SCN is mathematically equivalent to the affine transformations in an IFS. A single contractive affine transformation from an IFS, when repeatedly applied to some starting point, yields a fixed point attractor. When an external selection mechanism can choose from a collection of such transformations, the attractor of all possible composed transformation sequences can become a complex fractal set.

The SCN adds a sigmoid to "squash" the internal state into a small subregion, preventing states from flowing to infinity and allowing nonfixed point attractors, such as limit cycles and chaos, in the individual transformations. This sigmoid complicates the analysis since the expansion and contraction of the transform is no longer uniform over the domain, a property held by both linear and affine transformations. To visualize an attractor formed by the selection of transformations, one can play the "chaos game" (Barnsley, 1988) by plotting a sequence of randomly transformed state points. In Figure 31, the internal states of the **`ABCD`** MM have been plotted.[4] This plot was created by presenting a sequence of one thousand inputs selected at random from the set {**`A`**, **`B`**, **`C`**, **`D`**}, plotting the internal states vectors as they were generated, and dropping the first 100 to eliminate transients.

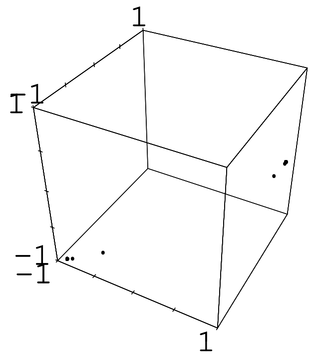
The attractors associated with each input of the SCN appear in the state space plots of Figure 31. The individual transformation associated with input **`A`** produces a fixed point trajectory; the remaining inputs generate period two attractors in the limit. Unlike the group product of FSM states, the SCN generates a fractal set of states. We can now attribute the "ravines" in the state space trajectory of Figure 30 and those observed by Pollack (1991) to the SCN switching between the individual attractors of its transformations. In the previous chapter, I referred to the same mechanism in the construction of the Sierpinski triangle. The transients, not the limit behavior, of the individual transformation are responsible for self-similar or self-affine structures that arise from their combined iteration.

Pollack trained an SCN to recognize regular languages from a set of positive and negative examples.Figure 32 shows a state space dissection of the SCN trained on examples from the language **`0`*****`1`*****`0`*****`1`***. While the network never learned the target language, it did exhibit this beautiful state space (marked as SCN states in Figure 32). By breaking the SCN into two transformations, one for each input symbol, we can examine the individual transformations that produced the "magic mushroom." The first iteration of the **`0`** transformation maps the state space onto a thin sheet along the diagonal while the first iteration of the **`1`** transformation pushes the state space to the ceiling with a one-sixth twist. In the limit, the **`0`** transformation produces a fixed point, while the **`1`** transformation gets closer and

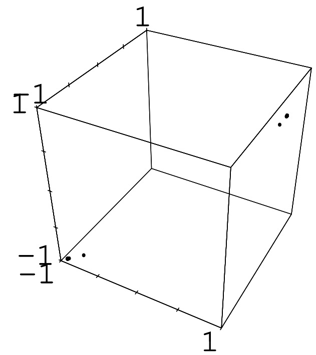
4. The SCN used three internal state nodes.



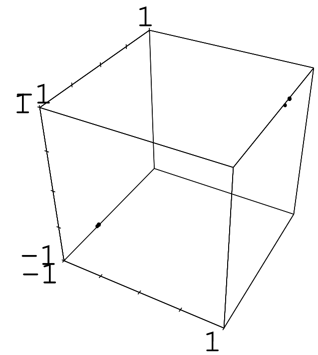
Input **A**: Iterations 1 through 50



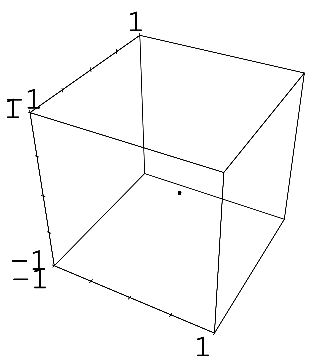
Input **B**: Iterations 1 through 50



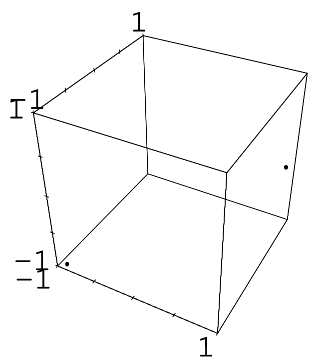
Input **C**: Iterations 1 through 50



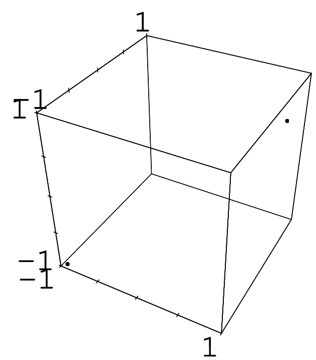
Input **D**: Iterations 1 through 50



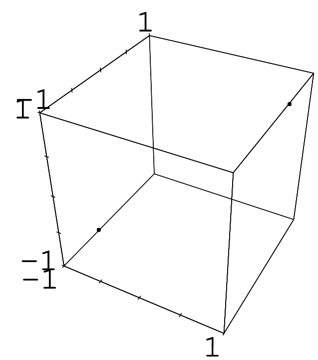
Input **A**: Iterations 51 through 100



Input **B**: Iterations 51 through 100



Input **C**: Iterations 51 through 100



Input **D**: Iterations 51 through 100

Figure 31: Attractors of the individual transformations of the ABCD MM

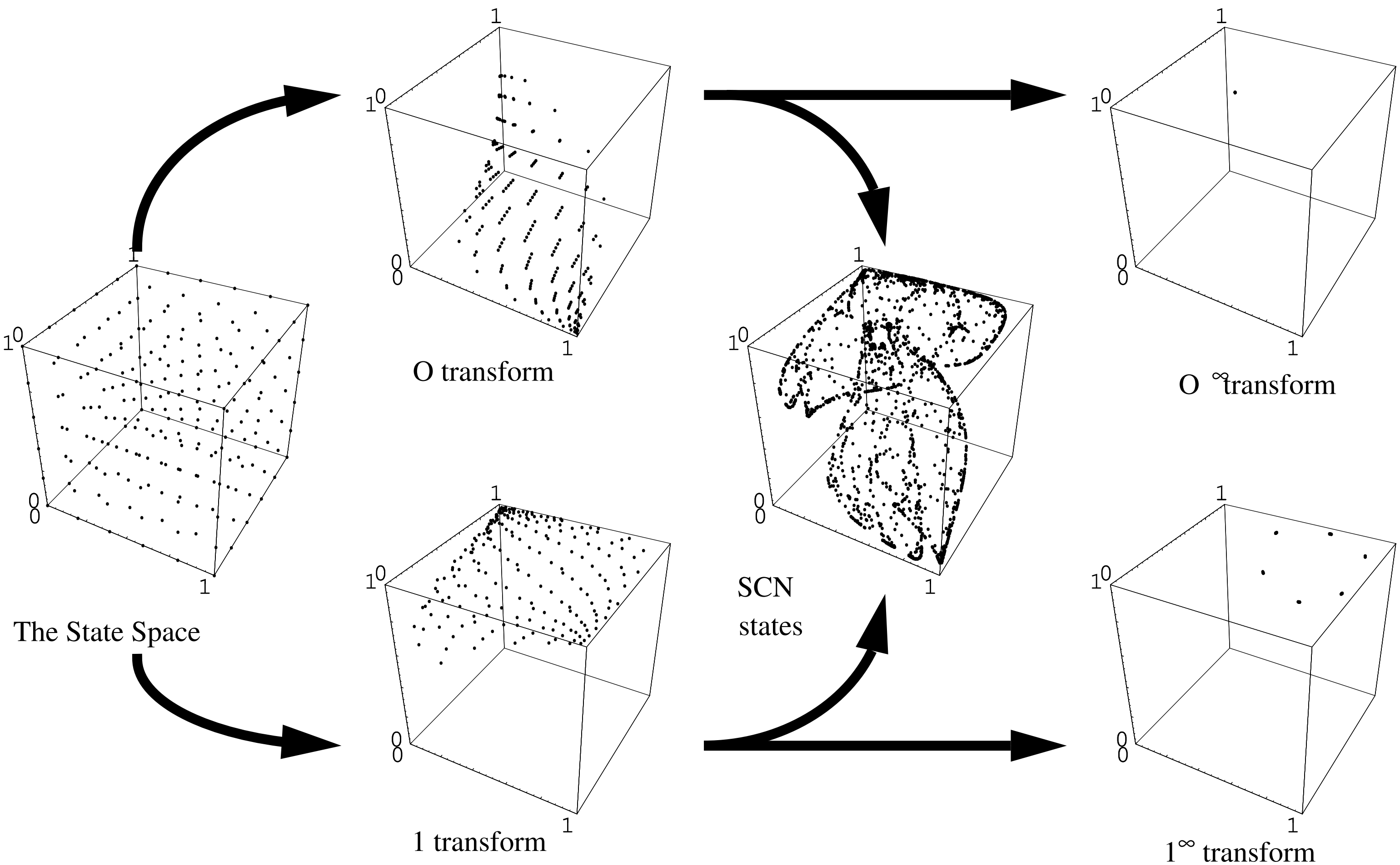


Figure 32: Dissection of the sequential cascaded network trained to accept 0*1*0*1*.

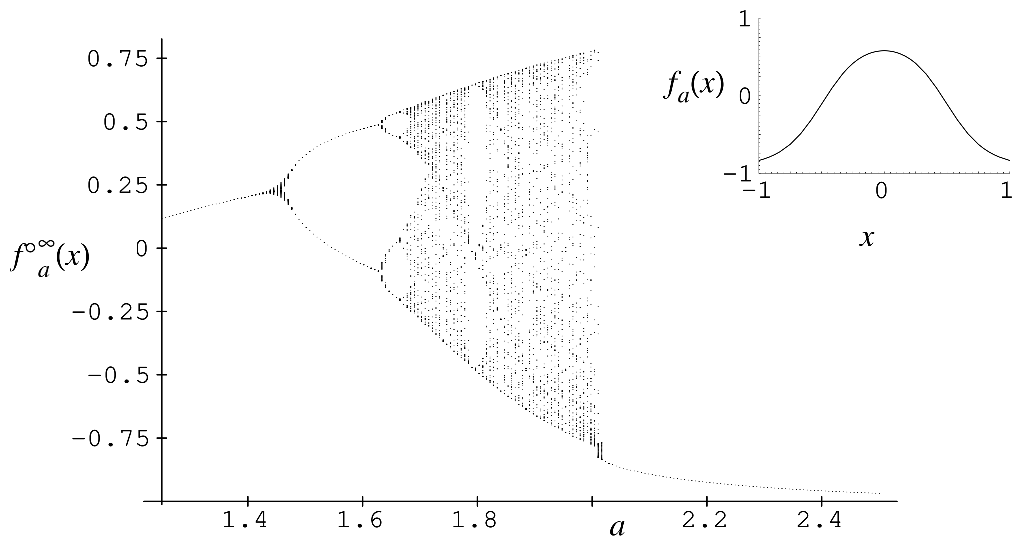


Figure 33: Sample mapping function and the resulting bifurcation diagram.

closer to this period-six attractor. Each of the 3-dimensional state spaces was generated by mapping the appropriate transformations on a 15x15x15 lattice in state space.

The interaction of transforms with limit point and period six attractors produced the mushroom attractor, clearly explaining the existence of fixed point and limit cycle attractors for individual inputs. Chaotic, or aperiodic, attractors are also possible in SCN state space. Figure 33 illustrates the effect of increasing parameter a on the asymptotic trajectories of the function f_a as shown in Equation 30.

$$f_a(x) = \tanh(a\tanh(ax+0.5a) - a\tanh(ax-0.5a) - a) \qquad \text{(Eqn 30)}$$

The resulting plot, known as a bifurcation diagram, captures the underlying structure of the period doubling phenomena. At the limits of period doubling (such as $a \approx 1.7$), the system's attractor is chaotic, thus implying infinite state behavior in the limit. The function f_a can be expressed as a SCN-like input-selected transformation by spreading the iterated computation of over two time steps, as in Equation 27 where the third element of $S^{(t+1)}$ is $f_a(x_t)$ and the first and second elements of $S^{(t)}$ are $\tanh(ax+0.5a)$ and $\tanh(ax-0.5a)$.

In this section, I described a dissection method for studying SCNs as IFSs. The key to this method has been the indexing of state transformations by a finite set of input vectors. As with IFSs, the transient behavior of the indexed transformations plays a crucial role in the emergent fractal structure of the attractor. Unlike IFSs, the transformations induced in

SCNs possess nonlinearities that allow them to display nonfixed point attractors. The tasks that we train SCNs to perform, however, require nontrivial attractors. For instance, recognizing binary strings with even parity requires a period-two oscillation upon seeing strings of `1`s. Periodic attractors were also seen in the ABCD multiassociative memory and the mushroom attractor. Finally, I demonstrated the capability of an SCN's indexed transformation to produces a chaotic attractor.

5.3.3 Other Recurrent Network Architectures

Sequential cascaded networks are not the only recurrent network architecture. Recall that Chapter III reviewed several architectures with internal states that dynamically change over time. I will now examine three of those models, SRN, Jordan, and RAAM, in the same manner that the SCN was dissected in the previous section. It will be shown that each architecture, and thus all other recurrent networks, shares similar transfer mechanisms.

I will first turn my attention to Elman's simple recurrent networks (SRNs). A single layer of feed forward connections merge the input and current state vectors into the next state vector. By reformulating the dynamics of SRNs, we can describe them as sets of input-selected transformations. The target form of these transformations will be $A_{\mathbf{I}}x + b_{\mathbf{I}}$, where A represents the connection weights from the current state to the next state units and $b_{\mathbf{I}}$ represents the biases of the next state units for a particular input `I`. The input `I` *parameterizes* the state transfer function. With the SRN, the state transition dynamics are given by Equation 31.

$$S^{(t+1)} = g[W \cdot \alpha S^{(t+1)} + V \cdot I^{(t)}] \quad \text{(Eqn 31)}$$

By making the same assumptions about input indexing as in the SCN case, we get Equation 32.

$$S^{(t+1)} = g[W \cdot \alpha S^{(t+1)} + v_{\mathbf{I}}] \quad \text{(Eqn 32)}$$

The $v_{\mathbf{I}}$ is an input-selected bias applied to the state units. Note that SRN transforms are of the form $Ax + b_{\mathbf{I}}$. In other words, only the bias is affected by inputs and the multiplicative part of the transformation remains constant. This is a significant difference between first and second order recurrent systems. In networks with second order connections, the inputs can specify entirely new transformations, while first order networks can only change the bias. Note that for each SRN, the common use of the term in the dynamics restricts the class of transformations$A$. There is no way for the input to select an arbitrary transformation. In any SRN, the current state is transformed into the current state modulated by the input pattern. The output of the network can be calculated in any number of ways and is independent of this discussion. This mirrors a computational difference between first and second order recurrent networks identified by Goudreau et al. (to appear). This work showed that single layer recurrent networks require an extra layer of processing to implement arbitrary finite-state recognizers. This layer performs state-splitting (Minsky, 1967). The states required splitting because the transfer function could not effectively encode all possible state transitions. Therefore, new auxiliary states had to be postulated.

The recurrent networks specified by Jordan suffer from a similar problem. Elman's networks were a simplification of the ones studied by Jordan. They incorporated multiple layers of processing and state activation decay. Although the transformations can be arbitrarily complex by adding more processing layers, the input only affects the biases of the first layer of processing. The rest of the network is unaffected by the input, as it still carries out the same state transformation. Thus, the behavior of Jordan networks can be described as the composition of a fixed transformation specified by the layers not directly connected to the input and the state decay with the input selected transformation occurring at the first level of processing.

Recursive autoassociative memories (RAAMs) also fall under the IFS explanatory umbrella. Recall that a RAAM is merely an encoder-decoder network in which the hidden unit activations are recursively used as representations. Consider the 2n-n-2n RAAM that is typically used to encode binary trees of terminal and nonterminal "symbols." These symbols are actually n-vectors. We can decompose the decoder for this RAAM into a LEFT decoder and RIGHT decoder for reproducing the left and right children of a node. These operations are recursively applied to the resulting vector representations—if they are nonterminals, to produce terminals or more nonterminals. Yet recall that the terminal/nonterminal distinction is one of observation: the controlling process must decide if a representation is atomic or not. Since we can apply these operations *ad infinitum* and they are discretely selected, the IFS framework applies again.

A finite state machine can be thought of as a special type of recurrent network (Equation 33). The vector $S^{(t)}$ represents the current state of the machine as a one-in-n code of zeros and ones. The weight matrix, $W_{\mathbf{I}}$, implements the machine's transition table: if a transition exists from state i to state j, then $W_{\mathbf{I},ij} = 1$, otherwise $W_{\mathbf{I},ij} = 0$.

$$S^{(t+1)} = W_{\mathbf{I}} \cdot S^{(t)} \qquad \text{(Eqn 33)}$$

Note that this implementation requires n state units and n^2 weights for an n state machine. For large machines, the necessary resources could quickly get out of hand. In these situations, one would expect to see resource savings by switching to binary representation. This reduction in state resources would require an increase in the complexity of the state mapping. This mapping must perform arbitrary n-bit to n-bit binary associations. Since two-layer networks are universal approximators (when given sufficient hidden units (Hornik, Stinchcombe & White, 1989)), the mapping in Equation 33 can implement such arbitrary associations. The δ function performs a zero/one step at zero. Unlike the previous implementation, the weight matrices $W_{\mathbf{I}}$ and $V_{\mathbf{I}}$ are not easily defined in terms of the target state machine.

$$S^{(t+1)} = \delta\left(V_{\mathbf{I}} \cdot \begin{bmatrix} \delta\left(W_{\mathbf{I}} \cdot \begin{bmatrix} S^{(t)} \\ 1 \end{bmatrix}\right) \\ 1 \end{bmatrix}\right) \qquad \text{(Eqn 34)}$$

In this section, I have extended the IFS interpretation to several recurrent network models. The SRN and Jordan networks were shown to implement indexed transformations in which only the bias changed from input to input. Recall that with the SCN, the input affected the weights between the current state units and the next state units. This difference arises because of the first order connections in SRNs and second order connections in SCNs. RAAMs, while not strictly a recurrent network, could also be characterized in terms of indexed transformation. If we consider the "state" of the RAAM to be the compressed representations created at the encoder/decoder bottleneck, then repeated decoding of these representations into the left or right subtrees exhibited an indexed mapping. Finally, I showed how a finite state machine could be implemented as a simple matrix multiplication where the weight matrix is indexed by the current input symbol.

In many dynamical systems, such as iterated mappings and cellular automata, the only way the outside world can affect these systems is through their initial conditions. We can, however, couple IFSs and recurrent networks to their environment. For IFSs, the chaos game requires external selection of transformation. Recurrent networks connect to the outside world by means of an input vector. Just as the chaos game is sensitive to the distribution of the transformation selections, so are the state spaces of recurrent networks. This sensitivity can be exploited to implement language recognizers with recurrent networks. I will return to the question of how complex computation can emerge in an input driven recurrent network in Chapter VII.

5.3.4 Why Clustering Occurs

At the beginning of this chapter, I introduced two methods of extracting finite state machine descriptions from recurrent networks. The basis of one of these methods was a state-vector clustering algorithm. The popularity of clustering analysis of internal representations occurring in neural networks suggests that there is some merit to this approach to analysis. Chapter II reviewed the successful application of this technique to understanding feed-forward hidden unit representations. The results of this section illustrate an application of clustering which fails to provide any useful insights into the underlying mechanism. The assumption confronted below is that the relationships between internal state vectors of a recurrent network will help us understand the processing performed by the network at a cognitive level. Many studies have reported the emergence of various "concept" clusters during recognition and production tasks (e.g., (Servan-Schreiber, Cleeremans, & McClelland, 1988; Cleeremans, Servan-Schreiber, & McClelland, 1989; Pollack, 1990; Elman, 1990; Crucianu, 1994)). In fact, we can explain the clustering phenomena without resorting to information processing accounts: the key is IFS theory.

Before I start the explanation, I must return to the domain of iterated function systems. Every IFS attractor, according to Barnsley, has an addressing scheme defined by the set of transformations. Every point on the attractor serves as the limit of some sequence of transformations. Thus, an address of a point on an attractor is the infinite sequence of transformations whose limit is that point. In the case of the Sierpinski triangle, the first entry of the addresses of all the points in the upper left corner are the same. This process continues recursively within each mapping of the state space. Because the IFS relies on contractive mappings, the regions shrink with each transformation application, the limit of which is our target point. Figure 34 illustrates the addressing scheme for the Sierpinski triangle. The numbers refer to the transformation number. Starting with the final attractor (not shown),

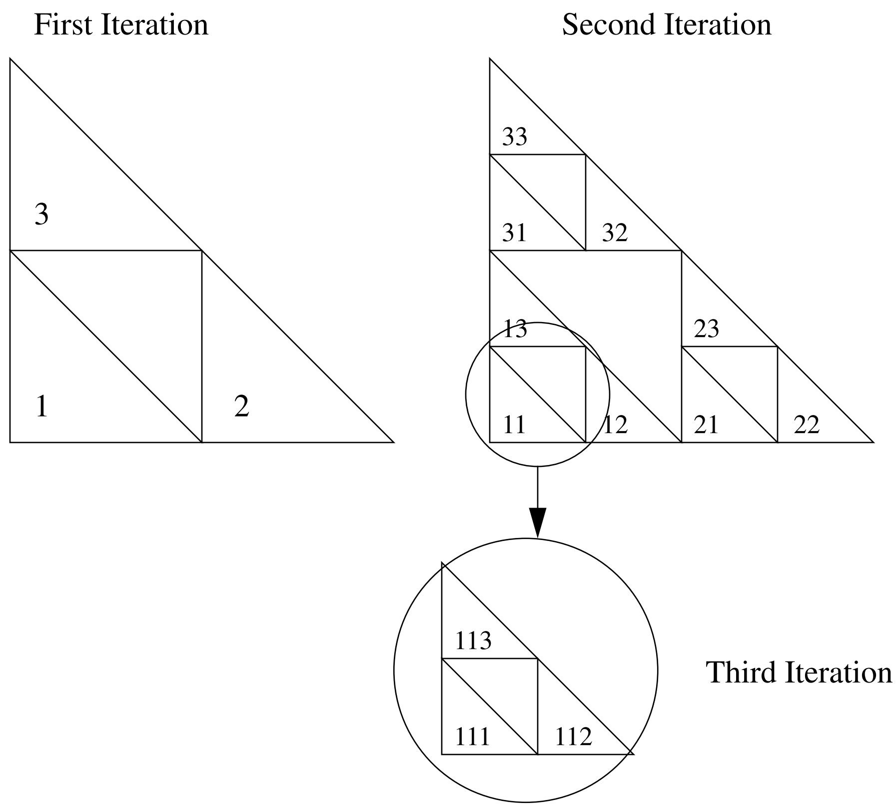


Figure 34: The addressing scheme for the Sierpinski triangle.

each transformation copies the original image into three regions (see First Iteration). Each copy has been labeled by the transformation that placed it. This is the address. On the second iteration, the copying process continues. This time, however, the transformation number is prepend to any existing label. This labeling process will continue indefinitely as each region becomes smaller and smaller. In the limit, the regions will be points. At this time, the address will be an infinite sequence of transformation labels.

The IFS address is the reverse of the input string to a recurrent network. The first element of the address was the most recent input symbol. The second element of the address was the next most recent input. If the transformations in the recurrent network are contractive (as they always are due to the sigmoid), *one could predict clustering on recency solely from the mathematics of the state transformations and independent from the learning task.*

Consider the experimental findings encountered during grammar learning using SRNs (Servan-Schreiber, Cleeremans, & McClelland, 1988; Cleeremans, Servan-Schreiber, & McClelland, 1989). In both papers, they took a simple recurrent network and trained it in the prediction task for the Reber grammar. The finite state machine underlying the Reber grammar and the network used to predict it is displayed in Figure 35. It is a six-

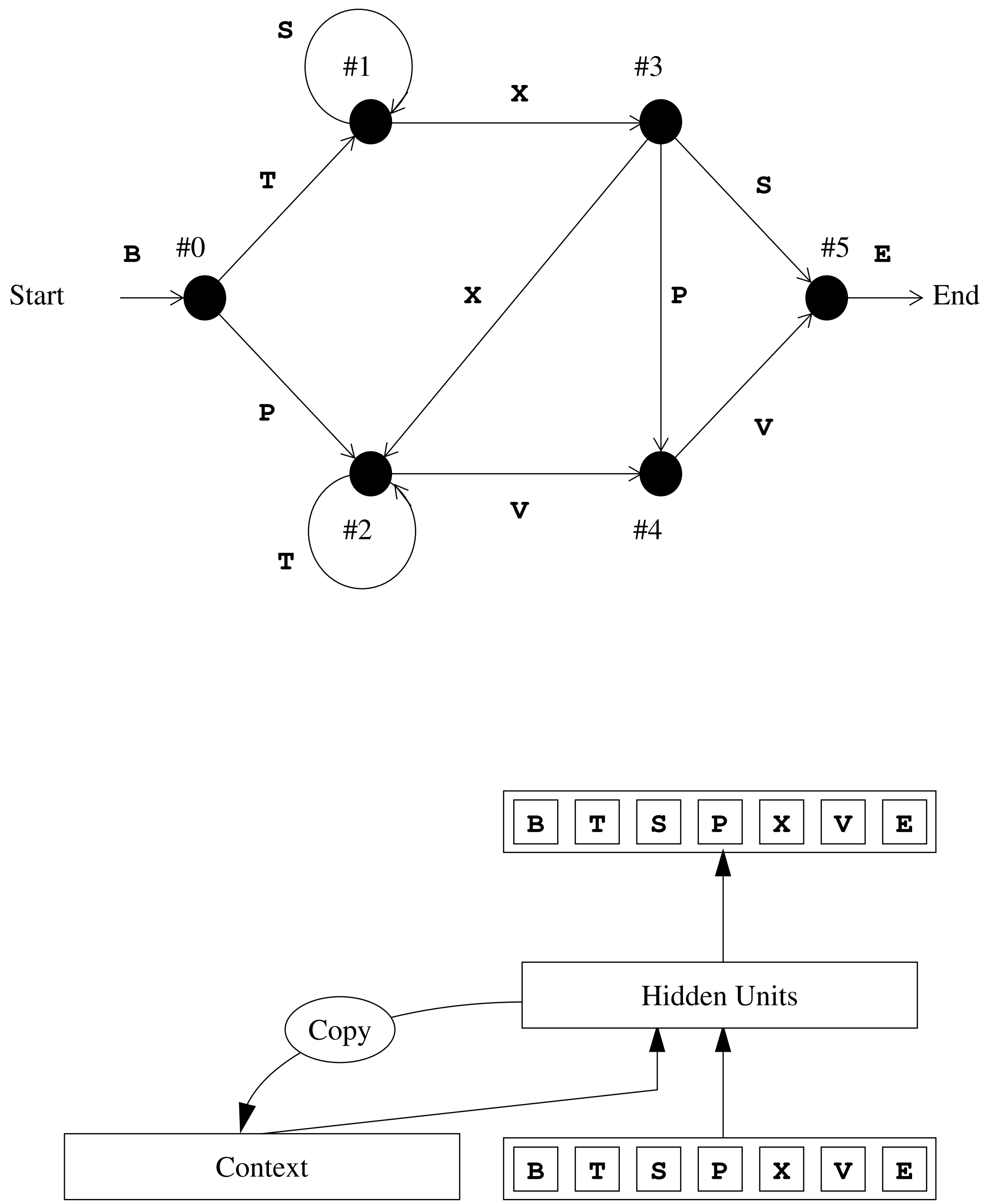


Figure 35: The Reber grammar and the simple recurrent network described in (Servan-Schreiber, Cleeremans, & McClelland, 1988).

state machine with an alphabet of seven characters. The rationale behind selecting this grammar over others is its historical position of being used in several psychological experiments on implicit learning (Reber, 1967). The training and evaluation environment of the network modeled these experimental setups. The goal of the task is to predict the next symbol, or symbols, given the current context. Figure 35 also contains a schematic diagram of the recurrent network used for this task. The network received input as a one-in-seven encoding of the seven characters of the Reber alphabet. The output consisted of seven units in a similar encoding scheme.

To justify the validity of my claim, I will generate a set of indexed transformations and perform a cluster analysis of the states encountered during the processing of the Reber grammar. Recall that different recurrent network architectures have different IFS interpretations. To implement the behavior of a SRN, the individual transformations must only differ in their additive factors. SCNs, on the other hand, can have independent transformations. In this demonstration, I have constructed SRN-like transformations with a state (or context) dimension of two (Equation 35).

$$S^{(t+1)} = g\left(\begin{bmatrix} a & b \\ c & d \end{bmatrix} S^{(t)} + \begin{bmatrix} e \\ f \end{bmatrix}\right) \qquad \text{(Eqn 35)}$$

The motivation for the number of hidden units was presentational: it is easier to view planar transformations. Similar arguments will hold for higher dimensional representation spaces. Table 4 lists the parameters of the five transformations, one for each symbol in the grammar (excluding **B** and **E**). Because the SRN architecture precludes input modification of the multiplicative parameters, the first four columns of Table 4 (corresponding to the parameters a, b, c, and d of Equation 35) are equal across each transformation. The transformation parameters were selected from a normal distribution with a mean of zero and standard deviation of one.

Analysis of this recurrent network will parallel the dissection of the SCN mushroom (Figure 32). The five plots (labeled **t**, **p**, **s**, **x**, and **v**) in the top half of Figure 36 show how the individual transforms map a 15x15 grid of equally spaced points in the plane back onto the plane representing the activations of the two state units. Also in the top half of Figure 36 is a plot labeled “Chaos game” that illustrates the attractor resulting from applying the chaos game approximation method. Each transformation had 1/5 probability of selection. The first 100 points were ignored as transients, and the last 500 points were plotted. This plot provides an approximate picture of the state representations of Σ^* (cf. (Pollack, 1991)) and corresponds to the mushroom plot in the center of Figure 32.

The bottom half of Figure 36 illustrates the SRN state representation of all Reber strings of length eight or less. The graphs labeled #1, #2, #3, #4, and #5 report the activation of the state vector when the generator was in the corresponding state. The final graph, labeled “All Reber States," is the union of the other five state graphs. Notice the similarity between the attractor approximation and the set of valid Reber states. Any decision mechanism using hyperplanes will have a difficult time differentiating between Reber and non-Reber strings solely from the state activation.

Servan-Schreiber et al. (1988) produced a cluster diagram of hidden unit activations before any training has occurred. This diagram clearly shows that the internal representa-

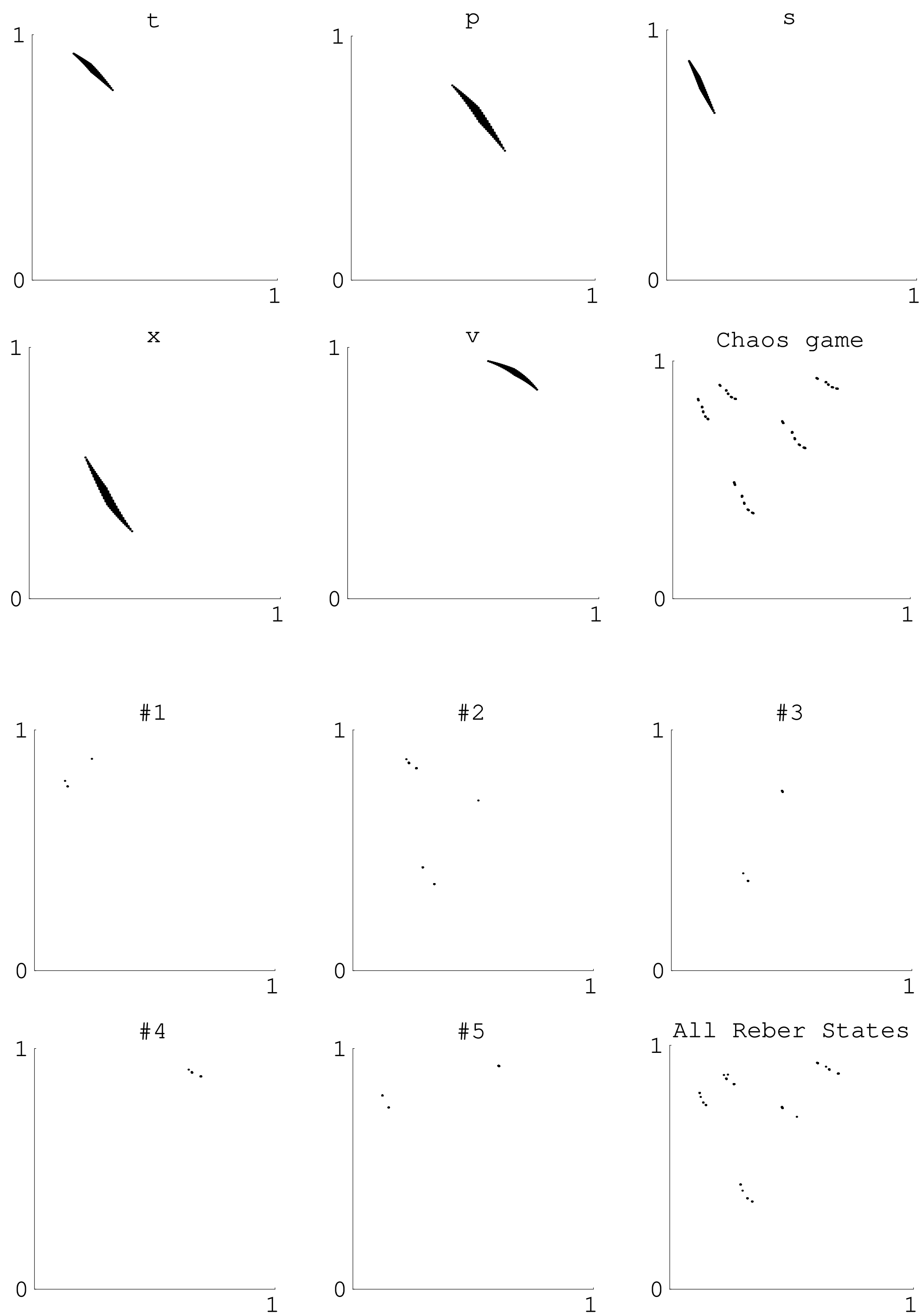


Figure 36: A collection of SRN representation spaces. The axes measure the activations of state nodes one and two. See text for explanation of individual graphs.

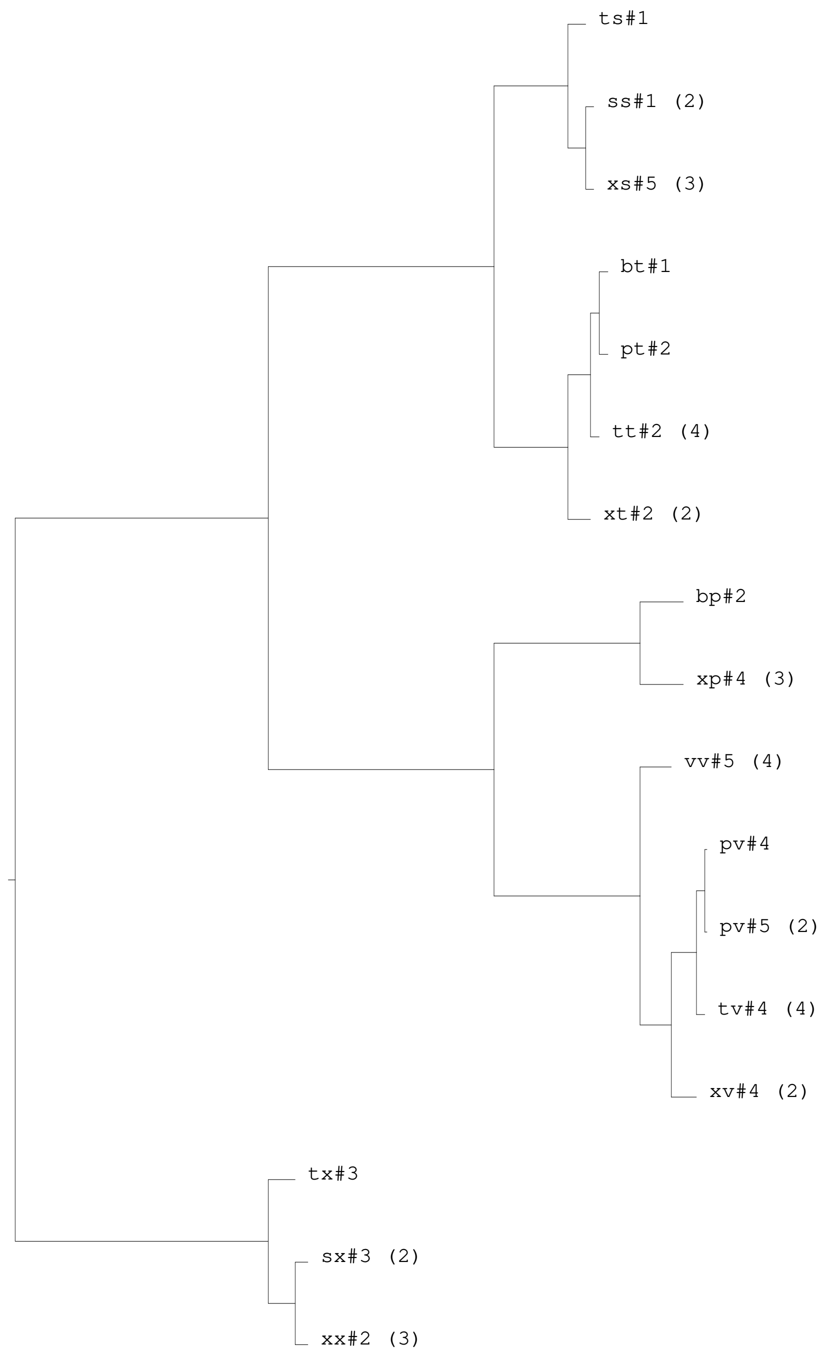


Figure 37: The resulting cluster diagram of all Reber strings of length eight or less on a random set of SRN transformations. The tree has been simplified, the numbers in parentheses is the total number of strings at that node with the same last two symbols.

Table 4:The SRN Transformations

SYM	a	b	c	d	e	f
t	-0.444983	-0.433067	0.759095	0.492625	-0.719507	1.23284
p	-0.444983	-0.433067	0.759095	0.492625	0.528296	0.121272
s	-0.444983	-0.433067	0.759095	0.492625	-1.44764	0.700865
x	-0.444983	-0.433067	0.759095	0.492625	-0.369454	-1.00214
v	-0.444983	-0.433067	0.759095	0.492625	1.11458	1.59991

tions of the network were clustering by the most recent symbol. Cleeremans' et al. claimed that the clustering of hidden unit representations was a product of training the network in the prediction task. These representations, in their eyes, captured regularities in the previous symbols. I disagree with this explanation. Since the Reber grammar states are uniquely decided by the last two symbols, the clustering comes free once the transformations no longer overlap. Figure 36 ("All Reber States") shows the state vectors for all strings up to length 8 that still lie on the Reber state graph. Clearly, the states clump together to a significant degree *without any training*.

A cluster diagram of these vectors, in Figure 37, emphasizes this observation. Rather than list every string still on the Reber state machine of length eight or less, I have abbreviated the string with the last two symbols and the current state. Thus, the string **`TSSXP`** will appear as **`XP`**#4. Numbers in parentheses indicate the number of other strings with the same suffix and state[5]. While this cluster diagram reveals an interesting amount of structure, that structure clearly does not correspond to the Reber state machine.

A similar expansion was performed on random SCN transformations. The transformation parameters were generated as before, but because the SCN architecture allows the input to affect the multiplicative factor in Equation 35, every cell in the table is different. Figure 38 shows the effects of the transformations listed in Table 4. Notice that the multiplicative parameters differ for each transformation, creating more variability in the shapes and orientations of the transformations. The relative differences in volume between the SRN transformations of Figure 38 result from the sigmoid activation function, while the difference in the SCN transformations can be traced to the multiplicative parameters. The bottom half of Figure 38 shows the state vectors for all strings up to length 8 that still lie on the Reber state graph. As before, the states display some amount of structure, even with random weight assignment. A cluster diagram of these vectors, in Figure 39, emphasizes this observation.

---

5. The cluster analysis was performed on the full strings. The abbreviations merely simplify the cluster tree and emphasize the distribution of state among the clusters.

Table 5:The SCN Transformations

SYM	a	b	c	d	e	f
t	-0.633273	-0.558169	-1.73999	0.431231	0.259375	-0.89197
p	-0.025588	-0.677765	0.962798	-0.277900	1.317470	1.12233
s	1.300490	-0.697644	1.21811	0.243118	-0.262879	-0.76881
x	1.171490	0.866640	0.111505	-0.496449	-0.464076	2.37686
v	-1.220900	-1.254000	0.09564	-0.82854	-0.105562	-0.36281

No learning is taking place in either demonstration. *Any emerging structure in the state space was independent of any task or learning mechanism.* Since most learning systems start with small weight assignments, one can conclude that learning time is spent:

- waiting for the recurrent state transformation to separate input symbol history encodings and
- finding an output function that separates them into relevant classes.

There is, however, no need for the network to actively cluster the states. The mathematics of the state transformations ensure that clustering will occur independent of training set and learning algorithm.

In this section, I explained the mechanics behind the emergence of state space clusters in recurrent network state space. I examined the clusters generated by randomly selected networks to avoid the regularities introduced by either tasks or learning mechanisms. By viewing a recurrent network as a set of input-indexed transformations, the true structure underlying state space attractors become evident. An input symbol causes the network to collapse its full state space into a subset of the original state space. This makes it more likely that state space representations of symbol strings ending with the same symbol will lie near each other. IFS theory shows that such mappings display recursive structure, implying that this argument holds for the remaining symbols in reverse order.

5.4 Conclusion

Formal language learning (Gold, 1969) has been a topic of concern for cognitive science and artificial intelligence. Neural information processing approach to this problem involves the use of recurrent networks that embody the internal state mechanisms underlying automata models (Pollack, 1991; Giles et al., 1992a; Servan-Schreiber, Cleeremans, & McClelland, 1988; Watrous & Kuhn, 1992). Unlike traditional automata based approaches, learning systems relying on recurrent networks carry a difficult burden; it is still unclear what these networks are processing, let alone what they learning. Many have assumed that the networks

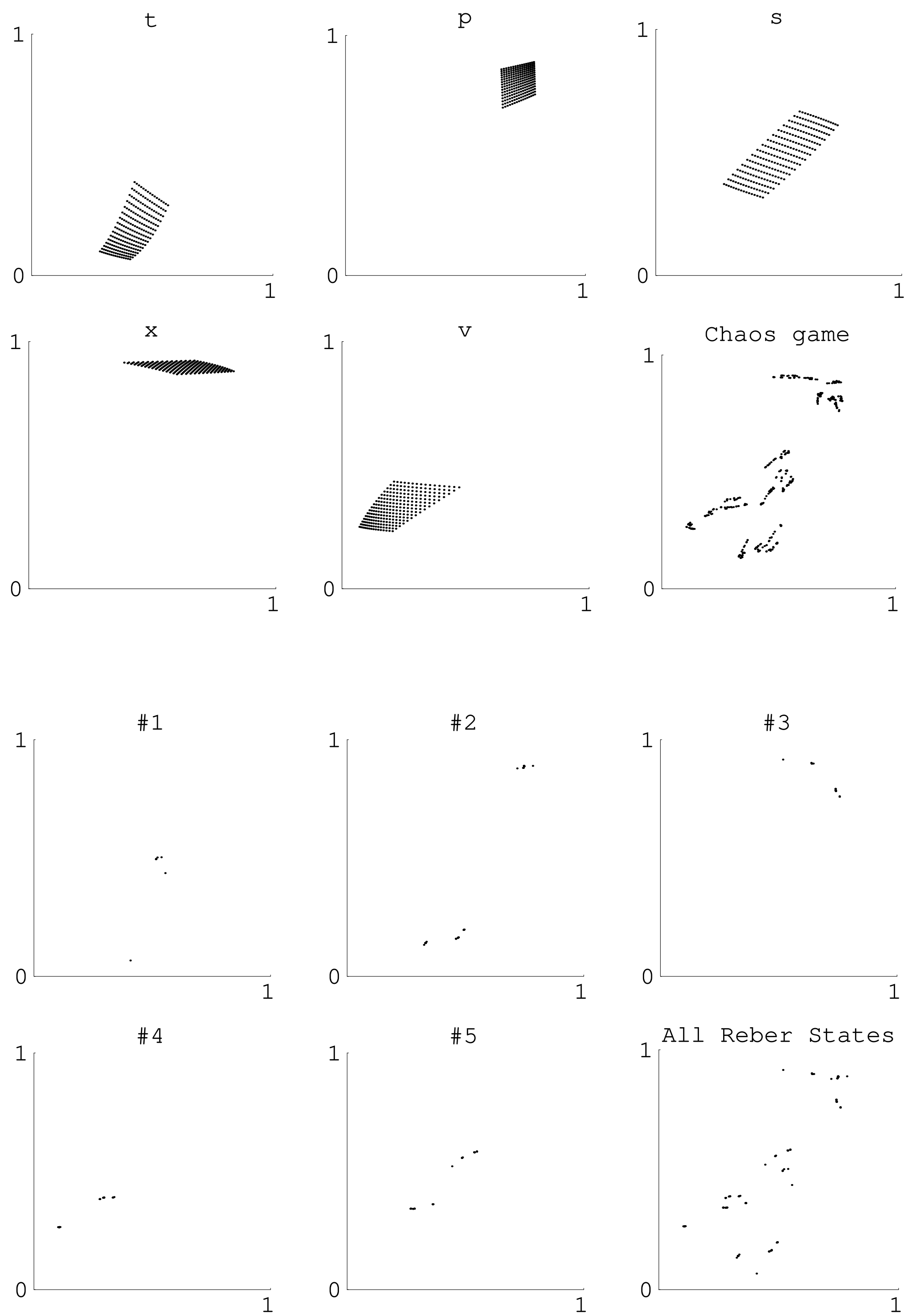


Figure 38: A collection of SCN representation spaces. The axes measure the activations of state nodes one and two. See text for explanation of indi-

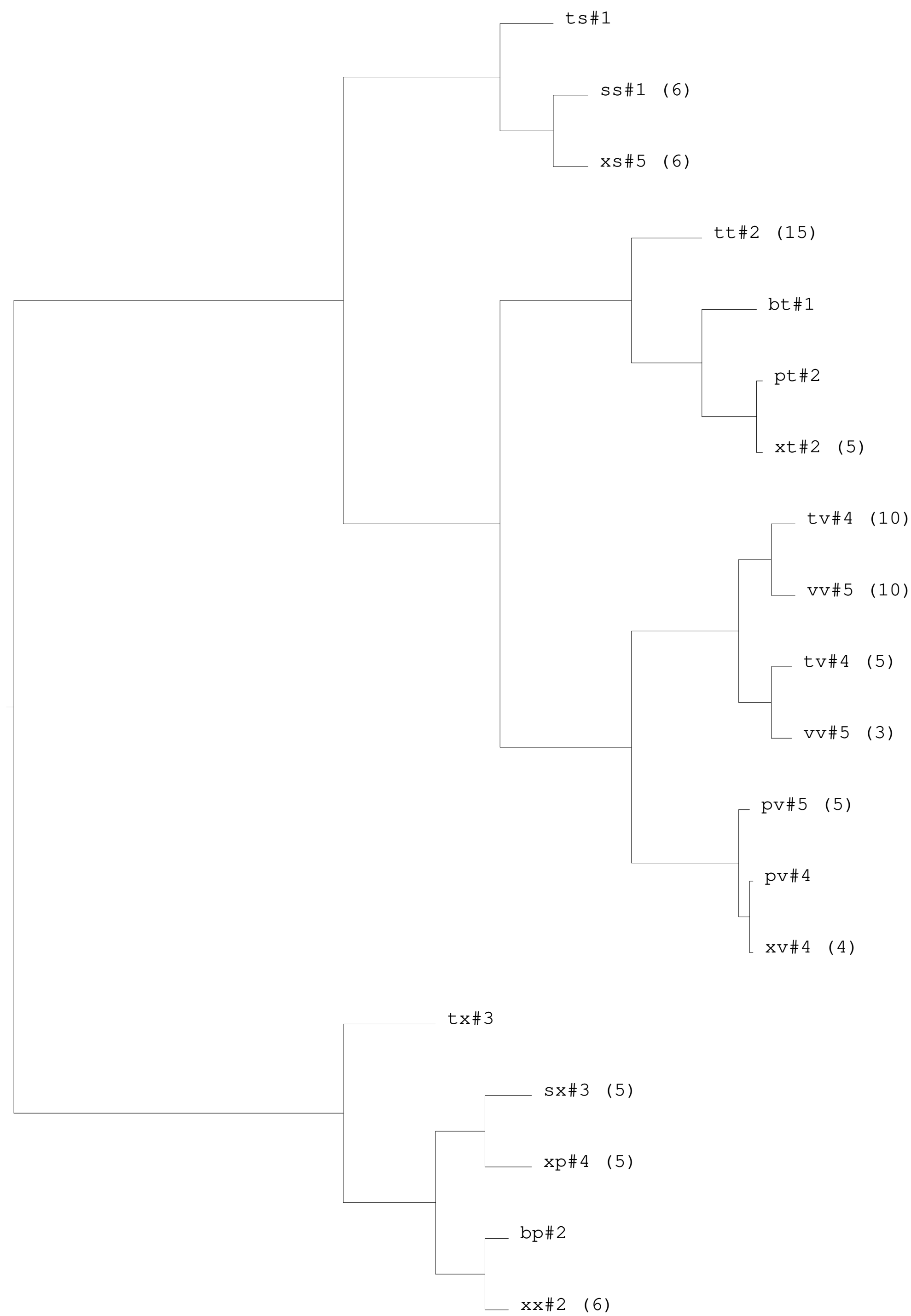


Figure 39: The resulting cluster diagram of all Reber strings of length eight or less on a random set of SCN transformations. See Figure 37.

are learning to simulate finite state machines in their state dynamics and have begun to extract finite state machines from the networks' state transition dynamics (Giles et al., 1992a; Watrous & Kuhn, 1992; Cleeremans, Servan-Schreiber, & McClelland, 1989). The extraction methods employ various clustering techniques to partition the internal state space of the recurrent network into a finite number of regions corresponding to the states of a finite state automaton. As demonstrated in the first part of this paper, clustering of hidden unit activations, or recurrent network state space, provides incomplete information regarding the IP state of the network. IP states determine future behavior and encapsulate input history. The network's state transformations can exhibit sensitivity to initial conditions and generate disparate futures for state clusters of all sizes.

The second part of the chapter illustrated how IFS theory can help explain recurrent network state dynamics. The limit behavior of a linear or affine transformation can be determined by examining eigenvalues of the transformations. Similar proof methods are used in understanding the dynamics of Hopfield networks. While the effects of iterating linear systems are fully understood, only recently did anyone consider the case of multiple linear transformations that occur in IFSs. The first hurdle in applying IFS theory to recurrent networks was understanding the effects of noncontractive transformations on the IFS theory. While there are many differences between IFSs with contractive and noncontractive mappings, many IFS properties are shared by systems that contract the volume of the state space but may not be uniformly contractive. The most important of these properties is the multiplication of transients that occur when the attractor is constructed by the union of a set of transformations.

The linkage between IFSs and recurrent networks has revealed existing constraints on network dynamics independent of network models. By assuming a finite set of inputs, which is often the case in symbolic domains, one can describe recurrent network models as a finite collection of nonlinear state transformations. The interaction of several transforms produce complex state spaces with recursive structure. The limit behavior of the collection of transformations, and thus of recurrent networks in symbolic applications, is more complex than the union of the individual transformations. An input driven recurrent network behaves like the random iteration algorithm. Infinite input sequence generates sequences of points dense in the state space attractor when the transformations are contractive. While the demonstration in this paper used the SCN, other models produce similar IFS-like behaviors as long as the network's input selects transformations (Kolen, 1993).

The IFS approach also explains the phenomena of state clustering in recurrent networks. Servan-Schreiber et al. reported significant clustering in simple recurrent networks both before and after training in the Reber grammar prediction task. A set of random transformations will normally reduce the volume of the recurrent network's state space, and place an upper bound on the distance between two transformed points. The upper bound has a significant effect on the clustering, especially when the transformations map to very small regions of state space. The task required that the clusters arranged themselves in a particular structure constrained by the single layer network generating predictions.

One response to the results presented in this chapter is to eliminate any source of nondeterminism from the mechanisms. In fact, Zeng et. al (1993) modified the SCN model by replacing the continuous internal state transformation with a discrete step function. (The continuous activation remained for training purposes.) This move was justified by their focus on regular language learning, as these languages can be recognized by finite state machines. This work is questionable on two points, however. First, tractable algorithms already exist for solving this problem (e.g. Angluin, 1987). Second, they claim that the network is self-clustering the internal states. Self-clustering occurs only at the corners of the state space hypercube because of the discrete activation function, in the same manner as a digital sequential circuit “clusters” its states. Das and Mozer (1994), on the other hand, have relocated the clustering algorithm inside the network. Their work focused on recurrent networks that perform internal clustering during training. These networks operate much like competitive learning in feed-forward networks (e.g. Rumelhart & Zipser, 1986) as the dynamics of the learning rules constrain the state representations such that stable clusters emerge.

The shortcomings of finite state machine extraction must be understood with respect to the task at hand. The actual dynamics of the network may be inconsequential to the final product if one is using the recurrent network as a pathway for designing a finite state machine. In this engineering situation, the network is thrown away once the FSM is extracted. We can view neural network training as an “interior” method to finding discrete solutions. It is interior in the same sense that linear programming algorithms can be classified as either edge or interior methods. The former follows the edges of the simplex, much like traditional FSM learning algorithms search the space of FSMs. Internal methods, on the other hand, explore search spaces that can embed the target spaces. Linear programming algorithms employing internal methods move through the interior of the defined simplex. Similarly, recurrent neural network learning methods swim through mechanisms with multiple finite state interpretations. Some researchers, specifically those discussed above, have begun to bias recurrent network learning to walk the edges (Zeng et al., 1993) or to internally cluster states (Das & Mozer, 1994).

There are situations, however, when the recurrent network *is* the target mechanism. The resulting recurrent network may assume many roles. For example, they may play an explanatory role in a cognitive model, or a mechanistic role in a control device, or an algorithmic role in image production. When these applications are explained in terms of finite states and transitions, something is lost. The discrete observations, necessary to develop such an explanation, ignore incremental changes that can contribute to long distance interactions. We gain something through these observations, too. In Chapter VI, I will address the effects of observation on the apparent complexity of systems such as recurrent networks.

To understand the behavior of recurrent networks, these devices should be regarded as dynamical systems (Kolen, 1994c). In particular, most common recurrent networks are actually iterated mappings, nonlinear versions of Barnsley’s iterated function systems (Barnsley, 1988). While automata also fall into this class, they are a specialization of dynamical systems, namely discrete time and state systems. Unfortunately, information processing abstractions are only applicable within this domain and do not make any sense in the broader domains of continuous time or continuous space dynamical systems.

The utility of network states, despite the mechanisms changing them over time, is directly related to the complexity of the mechanism measuring the state, which I call the observer. The next chapter illustrates the effects of observation on the apparent complexity of systems, including recurrent networks.

CHAPTER VI

THE OBSERVERS' PARADOX

6.1 Introduction

This chapter examines the effects of observation in the determination of complexity in physical systems. As we try to ascertain the information processing abilities of recurrent networks, it will become clear that the measurement of internal state plays a crucial role in the creation of complex behavior, including cognition. In his dynamical recognizer paper, Pollack (1991) introduced the following hypothesis regarding the computational complexity of dynamical recognizers.

> The state-space limit of a dynamical recognizer, as $\Sigma^* \rightarrow \Sigma^\infty$, is an Attractor, which is cut by a threshold (or similar decision) function. The complexity of the generated language is regular if the cut fall between disjoint limit points or cycles, context-free if it cuts a "self-similar" (recursive) region, and context-sensitive if it cuts a "chaotic" (pseudo-random) region.[1]

Pollack's hypothesis relates computational complexity to a combination of observer (slice) and dynamics. Since state dynamical systems can have separable, fractal, or chaotic areas, Pollack intended dynamical recognizers, an application of recurrent networks, to provide an alternative explanation of the complex syntactic structure of natural language. Various dynamical systems combined with a decision region for determining grammaticality could account for, he believes, the generative complexity of natural language, currently describable by context-free or context-sensitive grammars. These new systems could judge the grammaticality of symbol sequence just like their discrete time/space counterparts. Despite the novelty of his approach, the spirit of his hypothesis mirrored traditional accounts of grammaticality: different types of automata are responsible for the differences in grammatical complexity. For instance, push down automata can accept context-free languages, yet context-sensitive languages require a different mechanism, that of the linear bounded automaton. Although the "cutting" of the attractor indicates a measurement, Pollack, like many others, assumed that complexity arises purely from mechanism. This assumption directly implies that the implementation of complex behavior requires complex mechanisms.

1. Pollack (1991, p. 144).

While Pollack's hypothesis points the way to alternative mechanisms, such as a recurrent network displaying various dynamic regimes, it avoids the issue of source. Where does the computational complexity come from? This chapter argues that observation, besides mechanism, can introduce complexity in our descriptions of behavior. Concluding that observation is the sole source of complexity, however, would entail the same categorical errors described above. Chapter VII will attempt to unify the results presented in this chapter with several results presented in previous chapters to produce a coherent account of emergent computation in recurrent networks and physical systems.

6.2 Cognitive Science and Observation

Cognitive science, like many other fields, has worked under the general assumption that complex behaviors arise from complex computational processes. Computation lends a rich vocabulary for describing and explaining cognitive behavior in many disciplines, including linguistics, psychology, and artificial intelligence. It also provides an objective method for evaluating models by comparing the underlying generative capacity each model. The generative enterprise in linguistics, for example, maintains that the simplest mathematical models of animal behavior–as finite state or stochastic processes–are inadequate for the task of describing language. The complexity argument maintains that descriptions (or explanations) of language structure require at least a context-free or context-sensitive model:

> There are so many difficulties with the notion of linguistic level based on left-to-right generation, both in terms of complexity of description and lack of explanatory power--it seems pointless to further pursue this approach.[2]

Chomsky's complexity argument is not alone in cognitive science. Even Newell and Simon's Physical Symbol System Hypothesis (Newell & Simon, 1976) identifies recursive computation of a physical symbol system as both necessary and sufficient condition for the production of intelligent action. Newell and Simon emphasize that:

> The Physical Symbol System Hypothesis clearly is a law of qualitative structure. It specifies a general class of systems within which one will find those capable of intelligent action.[3]

Such claims are important as they focus our research attention on particular classes of solutions that we know *a priori* to have necessary mechanistic power to perform in our desired contexts. Since the publication of this hypothesis, the consensus of cognitive science has held that the mind/brain is computing something; identifying exactly what it *is* computing has emerged as the goal of the field.

Many have used computational complexity to separate cognitive (or intelligent) behaviors from other types of animal behavior. It will be shown that these judgments depend upon the observation mechanism, as well as the process under examination. In related work, Putnam (1988) has proved that all open physical systems can have *post hoc* interpretations as arbitrary abstract finite state machines and Searle (1990) has claimed that Word-

2. Chomsky (1957, p. 24).
3. Newell and Simon (1976, p. 116).

Star must be running on the wall behind him (if only we could pick out the right bits). Neither considered the effects of the observer on the complexity *class* of the behavior, the cornerstone of all complexity arguments in cognitive science and AI.

The remainder of the chapter is organized as follows. I will first examine the role that discrete measurements play in our studies of complex systems. Over the years, methods of complexity judgment have separated into two orthogonal approaches, namely *complexion* and *generative class*. The former is a judgment related to the number of moving parts (or rules, or lines of code) in a system, while the later may be viewed as a measure of the generative capacity of the chosen descriptive framework. Then I review research on the problem of identifying symbolic complexity in physical systems, emphasizing the recent work of Crutchfield and Young (1989). Once we recognize that computational descriptive frameworks apply to *measurements* of a system's state rather than the state itself, I then show how simple changes of the observation method or measurement granularity can affect either the system's complexion or its generative class. For instance, a shift in measurement granularity can promote the apparent complexity of a system from a context-free language generator to a context-sensitive language generator. Finally, I discuss the meaning of these results as they pertain to the cognitive science community.

6.2.1 Measurements and Complexity

Cognitive science expends most of its effort describing and explaining human and animal behavior. To construct these behavioral descriptions, one must first collect measurements of the observed behavior. Descriptions take many forms. A list of moves describes the behavior of the chess player, a transcript records linguistic behavior of a conversation, a protocol of introspected states during problem solving describes deliberative means-ends analysis, and a sequence of (x,y) locations over time records eye movement in a study of reading. In each of these examples, a written record of measured events serves as a description of behavior. The measurement may be simple, as in the case of the Cartesian coordinates, or it may be more involved, like the transcript or protocol. I assume that these measurements are necessarily discrete since they could be written down.[4] To emphasize the creation of discrete symbolic representations of the physical events in the world, I will identify this process as *symbolization*. Transcription of continuous speech, for example, is a symbolization of speech production. It is impossible to avoid symbolization; there is simply too much information inherent in any physical process that is undeniably irrelevant to our needs. Imagine trying to describe the conversation between two people on a street corner. The information generated by such an encounter is infinite due to a large number of real dimensions of movement, sound, time, etc. We avoid these complications by symbolizing the physical action into sequences of measurable events, such as phonemes, words, and sentences.

Information is clearly lost during symbolization. A continuous real value is a "bottomless" source of binary digits, yet only a few distinctions are retained through the transduction of measurements. Of course, shuffling high precision real numbers is a waste of re-

---

4. This becomes crucial when trying to measure an apparent continuous quantity like temperature, velocity, or mass. Recording continuous signals simply postpones the eventual discretization. Rather than measuring the original event, one measures its analog.

sources when only a few bits suffice. It is wrong, however, to believe that the information loss is merely modeling error if, as I show below, it often confuses our efforts at understanding the underlying system.

One way of understanding a system is by gauging its complexity. We have some good intuitions about certain relative comparisons: a human being is generally considered more complex than a rock. What does this ordering entail? Although judgments of system complexity have no uniformly accepted methodology, the existing approaches are sharply divided into two groups. The first appeals to the common sense notion that judges the complexity of a system by the number of internal moving parts. Thus, a system is more complex if it has more unique states induced by the internal mechanisms generating its behavior. Others (e.g. (Aida et al., 1984)) have adopted the term *complexion*. I specifically use this term to refer to a measure of complexity based upon the number of unique moving parts within a system. For instance, the complexion of a five-state finite state machine is greater than that of a four-state machine. Or is it? One must be sure that the five-state machine is *minimal* and not just a three-state machine embedded in a five-state implementation.

The second approach to judging complexity is more subtle. Imagine a sequence of mechanisms, specified within a fixed framework, with ever increasing complexion. As the complexion of a device increases, its behavior eventually reaches a limit in complexity inherent in the framework itself. This limit, or generative class, can only increase or decrease with modifications to the framework. These classes are not unique; many frameworks share the same generative limit. Chomsky's early work (e.g. (1957; 1965)) in linguistics contributed to the foundations of computer science. Followers of this work have enshrined four classes of formal languages, each with a different computational framework. The regular, context-free, context-sensitive, and recursive languages are separated by constraints on their underlying grammars of specification, and form an inclusive hierarchy. In addition, they correlate quite beautifully with families of mechanisms, or automata, operating under memory constraints. Of course, it is now well known that many other classes are possible by placing different constraints on how the changeable parts interact (see many of the exercises in (Hopcroft & Ullman, 1979)). I use the term "generative class" out of respect to the fact that this theory of complexity arose in formal languages (automata) and the questions of what kinds of sentences (behaviors) could be generated.

6.3 Computation, Cognitive Science, and Connectionism

Computation offers a way to describe and manipulate the resulting measurements once we have symbolized a sequence of measurements. In this respect, computation can be thought of as one of the most powerful tools of cognitive science during its explosive growth over the last forty years. The rise of the generative enterprise in linguistics, information processing in computational psychology, and the symbolic paradigm of artificial intelligence—each firmly based on symbolization—all benefited greatly from the modern computer's ability to universally simulate symbolic systems.

The rise of the "generative enterprise" over other descriptive approaches to linguistics can be attributed in part to its affinity with computation, since it initially appeared computationally feasible to generate and recognize natural languages according to formal grammars. A formal grammar is a collection of rewrite rules involving terminal symbols (those appearing in target language) and nonterminal symbols (temporary symbols used during the rewrite process). Chomsky (1957) proved that various constraints on rewrite rules produced sets of strings that could not be produced under other constraints. For instance, a grammar composed of rewrite rules involving nonterminals on the left-hand side of the productions and no more than a single nonterminal on the rightmost part of the productions (i. e., $A \rightarrow aB$, where A and B are nonterminal symbols and a is a terminal symbol) has less generative capacity than a grammar with rules whose right-hand sides can contain arbitrary sequences of terminal and nonterminals (such as $A \rightarrow aBb$). Generative capacity refers to the ability to mechanically produce more strings. The generative capacity of regular grammars is strictly less than that of context-free grammars, and both are strictly less than that of context-sensitive grammars. English readily provides several examples of center embedding that eliminate regular grammars from the possible set of mechanical descriptions of natural language grammaticality. Descriptions of natural languages based upon varieties of context-free phrase structure grammars, while very easy to work with, could not stand as correct models under such phenomena as featural agreement or crossed-serial dependencies. From a linguistic standpoint, any system capable of understanding and generating natural language behavior must exhibit context-sensitive generative capacity, though it is widely held that a class known as "indexed context free" is consistent with the weak generative capacity of natural languages (Joshi et al., 1989).

Psychologists, discouraged by behavioristic accounts of human performance, could now turn discretized protocols into working models of cognitive processing driven by internal representations. For example, Newell and Simon's (1962) Generalized Problem Solver (GPS) implemented means-ends analysis and could model intelligent behaviors like planning. The operators used in GPS were nothing more than productions and the entire system could be viewed as a production system. Production systems are computationally equivalent to Turing machines and other universal computational mechanisms. Based upon this, Newell and Simon (Newell & Simon, 1976) concluded that intelligent action is best modeled by systems capable of universal computation. This claim is now known as the Physical Symbol System Hypothesis and specifically states that a physical system implementing a universal computer is both necessary and sufficient for the modeling of intelligent behavior. Thus, context-sensitivity is not enough for intelligence, as in the case of linguistic competence, but the full computational power of recursive systems must be engaged for the production of intelligent behavior.

These behaviors are easily generated by universal frameworks such as unrestricted production systems or stored program computers. This unconstrained flexibility fueled the explosion in AI, where bits and pieces of cognitive action, such as chess playing, or medical diagnosis, could be converted into effective symbolic descriptions simulated by algorithms in computers. While flexibility is an asset in the development of computer software, unconstrained computational descriptions cannot help us develop a theory of cognitive processing. More often than not, we are left with a fragile model that overfits the data and fails to generalize despite the noble effort behind unification projects like Soar (Newel, 1990).

The scientific problem regarding the lack of constraints offered by general information processing systems has fueled the recent "back to basics" reaction in using more "neurally plausible" means of modeling. Connectionism has been a vigorous research area expanding in its scope throughout the 1980's, bounded both by the underconstraints of artificial intelligence and information processing psychology and by the overconstraints of computational neuroscience. Connectionism seeks to understand how elements of cognition can be based on the physical mechanisms that can be in the brain, without being constrained by the overwhelming detail of neuroscience. As such, it is constantly battered both from below (e.g., (Grossberg, 1987)), on actual biological plausibility of the mechanisms, and from above (e.g., (Fodor & Pylyshyn, 1988)), on the adequacy of its mechanisms when faced with normal tasks of high-level cognition requiring structured representations (Pollack, 1990).

Pollack tried to address the generative capacity issue raised long ago by Chomsky. In this work he examined biologically plausible iterative systems and found that a particular construal, the "dynamical recognizer," resulted in recurrent neural network automata that had finite specifications and yet infinite state spaces (Pollack, 1991). From this observation he hypothesized that yet another mapping might be found between the hierarchy of formal languages and increasingly intricate dynamics of state spaces (implemented by recurrent neural networks). Crutchfield and Young's (1989) paper (summarized below) and similar conjectures regarding the emergence of complex computation in cellular automata as reported in the work of Wolfram (1984) and Langton (1990) supported this line of reasoning. After many attempts to reconcile his recurrent neural network findings with both the dynamical systems results and a traditional formal language view, I came to believe that the difficulty of our endeavor lay in the traditional view that a particular mechanical framework adheres to a particular generative class.

6.4 The Emergence of Complex Behavior in Physical Systems

The two notions of complexity—complexion and generative class—have been traditionally applied only to computational systems. However, recent work by Crutchfield and Young (1989) suggests that one could talk similarly about the generative class of a physical process. Their work focuses on the problem of finding models for physical systems based solely on measurements of the systems' state. Rather than assuming a stream of noisy numerical measurements, they explore the limitations of taking very crude measurements. The crudest measurement of state is periodic sampling with a single decision boundary: either the state of the system is above or below a threshold at every time step. Unlike numerical measurements that can be described mathematically, the binary sequence they collect requires a computational description (i.e. what kind of automaton could have generated the sequence?) They claim that the minimal finite state automaton induced from this sequence of discrete measurements provides a realistic assessment of the intrinsic computational complexity of the physical system under observation. To test this claim, Crutchfield and Young generated binary sequences from nonlinear dynamical systems such as the iterated logistic and tent maps. These systems have the property that infinitesimally small changes in a single global parameter can cause qualitative behavioral changes, known as period doubling bifurcations, where the behavior of the system in the limit moves from a period n oscillation to a period $2n$ oscillation. Besides the claim stated above, their paper provides three key insights into the problem of recognizing complexity as it arises in nature.

First, the minimality of the induced automaton is important. Crutchfield and Young propose that the minimal finite state generator induced from a sequence of discrete measurements of a system provides a realistic judgment of the complexity of the physical sys-

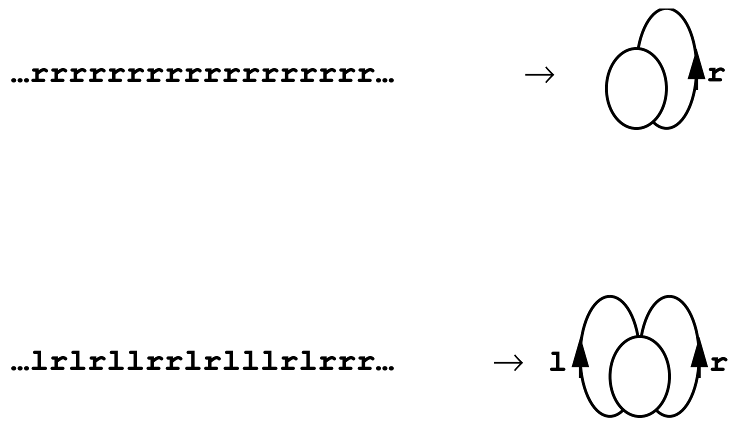


Figure 40: Finite state descriptions of equivalent complexity. The first subsequence is from the sequence of all **r**s. The second subsequence is from a completely random sequence. Both sequences could be generated by a single state generator since each new symbol is independent from all other preceding symbols.

tem under observation. Minimality creates equivalence classes of descriptions based on the amount of *structure* contained in the generated sequence. Consider two systems–the first constantly emits the same symbol, while the second generates a completely random sequence of two different symbols. Both systems can be described by one-state machines that can generate subsequences of the observed sequences (Figure 40). In the constant case, the machine has a single transition. The random sequence, on the other hand, has a single state but two stochastic transitions. The ability to describe these sequences with single state generators is equivalent to saying that any subsequence of either sequence will provide no additional information concerning the next symbol in the sequence. Thus, total certainty and total ignorance of future events are equivalent in this framework of minimal induced description.

Second, they show that physical systems with limit cycles produce streams of bits that are apparently generated by minimal finite state machines whose complexion increases with the period of the cycle. A system with a cycle of period two is held to be as complex as a two-state machine. Systems with constrained ergodic behavior exhibit similar levels of complexion; the number of induced states is determined by the regularities in the subsequences that cannot be generated. These are shown schematically in Figure 41.

Third, Crutchfield and Young proved that the minimal machines needed to describe the behavior of simple dynamical systems *when tuned to criticality* had an infinite number of states. At criticality, a system displays unbounded dependencies of behavior across space and/or time (Schroeder, 1991). The spread of forest fires at the percolation threshold of tree density (Bak et al., 1990) and sand pile avalanches (Bak & Chen, 1991) both produce global dependencies at critical parameters of tree density and pile slope. Even simple systems, such as the iterated logistic function $x_{t+1} = rx_t(1-x_t)$, exhibit criticality for an infinite set of r parameter values between zero and four. Crutchfield and Young proved that the

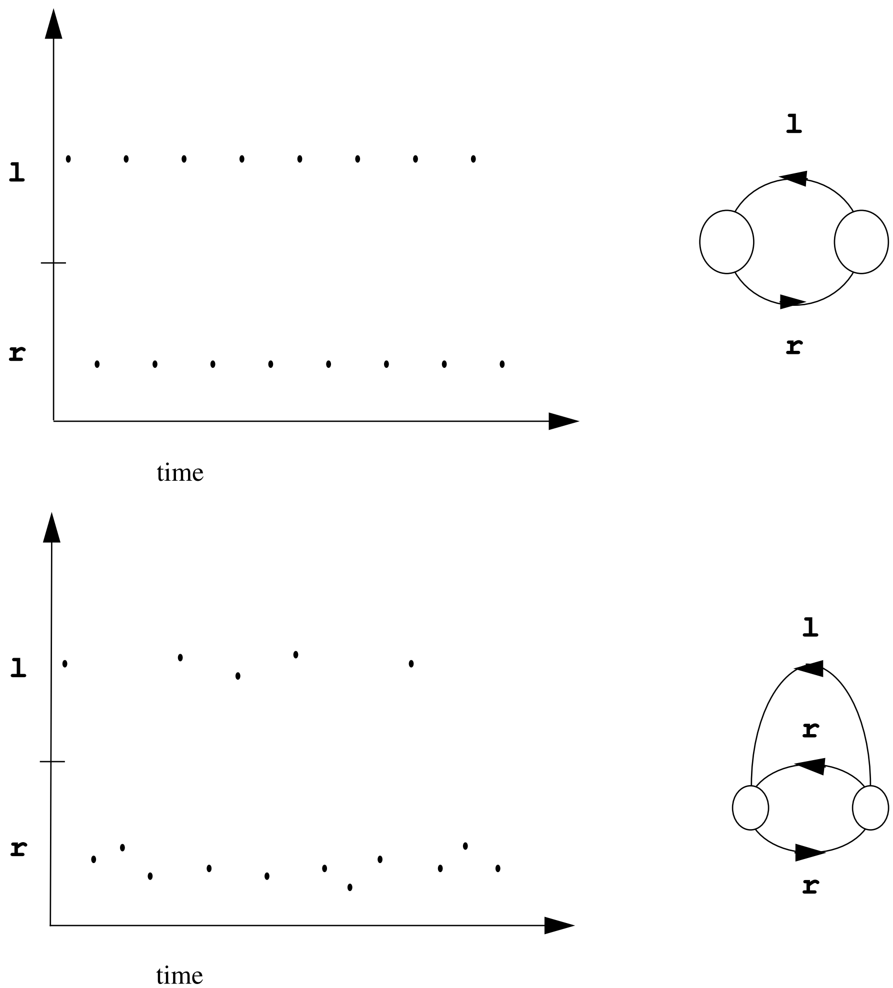


Figure 41: The state machines induced from periodic and chaotic systems. Note that the lower system does not produce `ll` pairs. This omission is the reason for the increase in number of states over the random generator in Figure 40.

computational behaviors at these parameter settings are not finitely describable in terms of finite state machines, but are compactly described by indexed context-free languages.

6.5 Apparent Complexion

The number of states in the systems studied by Crutchfield and Young can be selected by an external control parameter as the system bifurcates through various dynamical regimes. The task of merely increasing the number of apparent states of a system seems uninteresting because the simplest solution lies in being more sensitive to distinct states. Since we can arbitrarily zoom into any physical system, any object, including a rock, can simultaneously have a description requiring only a single state and descriptions with high complexion driven by atomic level motions. Figure 42 shows effects of increasing measurement granularity on the finite state machines induced from a dynamical system. I have selected the iterated mapping $x_{t+1} = 2x_t \pmod 1$, also known as the Baker's shift, for this demonstration. The behavior of this iterated system is to shift the bits of a binary encoding of the state, x_t at time t to the left by one place and then discard the bits to the left of the decimal point ($x_1 = 0.110101..._2$, $x_2 = 0.10101..._2$.) The first automaton was constructed by dividing the state space into two equal regions. This division results in a one-state machine that stochastically emits `0`s and `1`s with equal probability. The same trajectory, subjected to a measurement mechanism sensitive to three disjoint regions induces a three-state automaton. When four measurement regions are imposed on the state space, the resulting symbol sequence could be generated by the two-state machine at the bottom of Figure 42. Since an odd number of divisions will induce state machines with a corresponding number of states, an infinite number of finite state automata can be induced from the Baker's shift dynamical system.

Other scientists and philosophers have explored this route to complexity. Putnam (1988) has proved that an open system has sufficient state generative capacity to support arbitrary finite state interpretations. His core argument relies on *post hoc* labeling of state space to accommodate an arbitrary mapping between trajectories in state space and automaton states. Searle (1990) questions the relevance of computational models to cognition by claiming that a properly tuned perceptual mechanism could experience the WordStar word processing program in operation on the surface of the wall behind him. Based upon this observation, Searle concludes that causality, blatantly missing from the state transitions in his example, is the key component of cognition (Searle, 1992). Fields (1987) suggests that the arbitrary nature of state labellings is only a problem for classical systems, i.e., systems unaffected by observations of their state. He claims that when observations are made of nonclassical systems, the interaction between observer and system limits the granularity of observation and thus prevents the observer from drawing arbitrary computational inferences from the behavior of the system.

Finally, recall that Ashby (1956) points out that variable selection, which underlies the notion of a system, is critical since any physical process can provide an infinitude of variables that can be combined into any number of systems. This applies on the one hand to arguments of Searle and Putnam. On the other hand, the work in dynamical systems by Crutchfield and Young, relates the problem with unbounded accuracy to the issue of sensi-

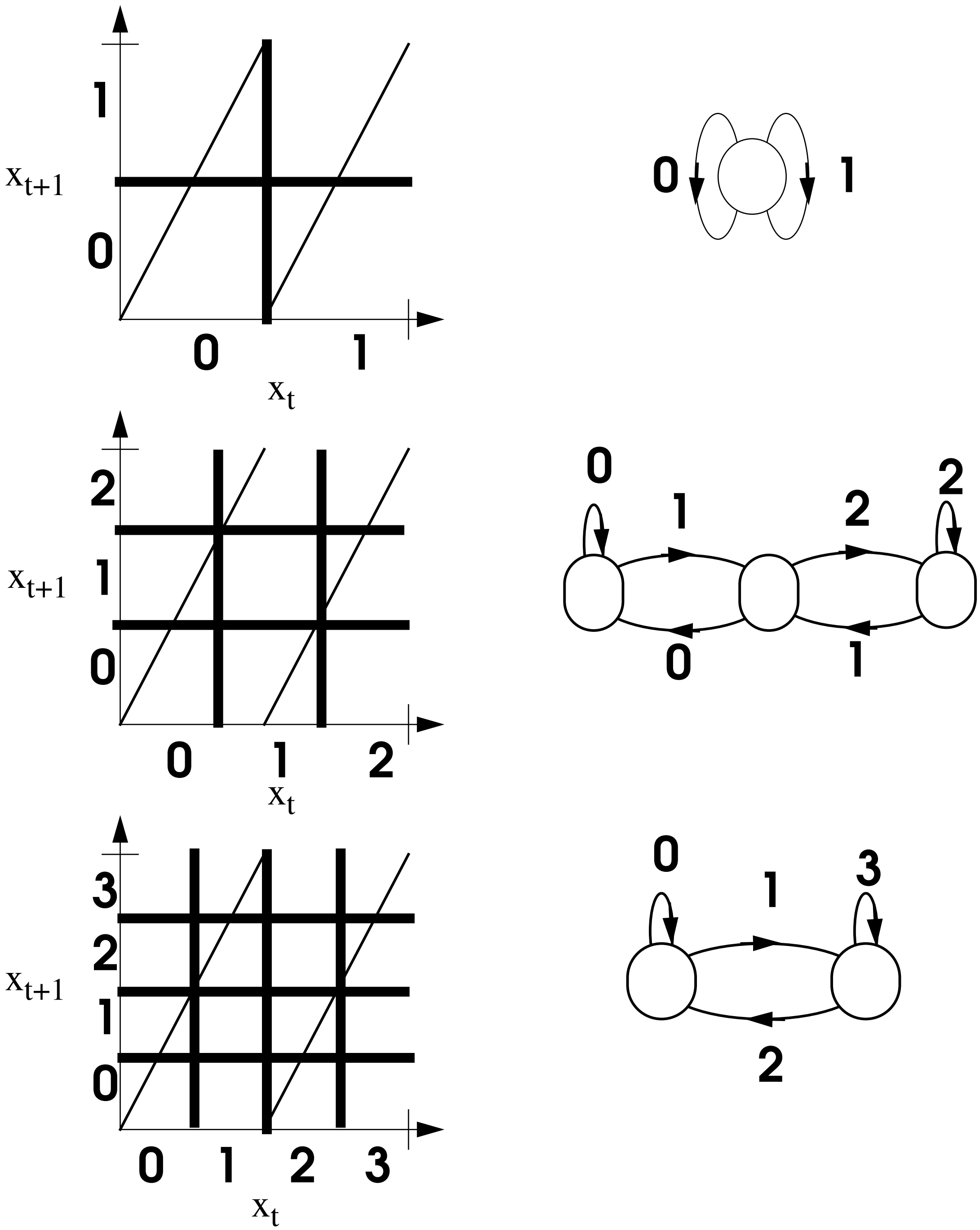


Figure 42: An illustration of an increase in the number of internal states due to explicit symbolization. The underlying mapping is $x_{t+1} = 2x_t \bmod 1$. The x_t and x_{t+1} axes in the graphs range from 0 to 1.

tivity to initial conditions. From Searle, Putnam, and Ashby's point of view, an attribution of computational behavior to a process rests on an isomorphism between states of the physical process and the information processing states of the computational model. Thus, the intended computational model guides the selection of measured quantities of the physical process. In addition, a modeler must measure the current state of a process with sufficient resolution to support the isomorphism. In models capable of recursive computation, the information processing state can demand unbounded accuracy from the modeler's measurements. As we shall see, the granularity of observation is not the issue here, rather it is the effects of discrete boundaries upon evoked complexity.

Relating the problem of unbounded accuracy to the issue of sensitivity to initial conditions may be a way to correctly characterize this issue. In other words, significant "state" information can be buried deep within a system's initial conditions, and become widely distributed in the state. This implies that often the best way to "measure" a system's complexity is to simply observe its behavior over long periods and retroactively determine the critical components of the state. For instance, Takens (1981) method of embedding a single dimensional measurement into a higher dimensional space can provide a judgment of the underlying system's dimensionality that is independent of the dimension of the observable. Crutchfield and Young extend this philosophy to the computational understanding of systems, and infer the generative complexity of a process from long sequences of measurements. This approach is also evident in traditional models of computation: the Turing machine can only react to the portion of tape, i.e. state, directly under the tape head. To fully understand the state of the Turing machine one must either know the contents of the tape or reconstruct the portion of the tape visited by the Turing machine during operation.

6.6 Apparent Complexity

Both Putnam and Searle avoided the bulwark engineered by Chomsky, namely the issue of generative complexity classes. Is generative complexity also sensitive to manipulation of the observation method? The answer, surprisingly, is yes. To support this claim, I will present some simple systems with at least two computational interpretations: as a context-free generator and a context-sensitive generator.

Consider a point moving in a circular orbit with a fixed rotational velocity, such as the end of a rotating rod spinning around a fixed center, a white dot on a spinning bicycle wheel, or an electronic oscillator. I will measure the location of the dot in the spirit of Crutchfield and Young, by periodically sampling the location with a single decision boundary (Figure 43). If the point is to the left of the boundary at the time of the sample, write down an "**`l`**". Similarly, write down an "**`r`**" when the point is on the other side.[5] In the limit, we will have recorded an infinite sequence of symbols containing long sequences of **`r`**s and **`l`**s.

The specific ordering of symbols observed in a long sequence of multiple rotations is dependent upon the initial rotational angle of the system. The sequence does, however,

5. The probability of the point landing on the boundary is zero and can arbitrarily be assigned to either category without affecting the results below.

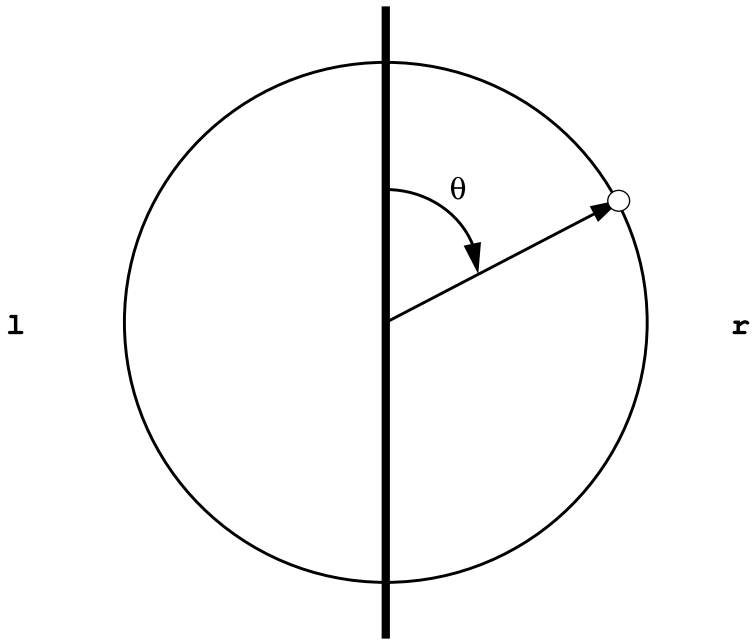


Figure 43: Decision regions that induce a context-free language. θ is the current angle of rotation. At the time of sampling, if the point is to the left (right) of the dividing line, an `l` (`r`) is generated.

possess certain recurring structural regularities that I call *sentences*. A sentence in this context is a run of **r**'s followed by a run of **l**'s. For a fixed rotational velocity (rotations per time unit) and sampling rate, the observed system will generate sentences of the form $\mathbf{r}^n\mathbf{l}^m$ $(n, m > 0)$. (The notation $\mathbf{r}^n$ indicates a sequence of n **r**s.) Each rotational velocity specifies at most three sentences in which the number of **r**s and **l**s in each sentence differ by at most one. These sentences repeat arbitrarily according to the initial conditions of the rotator. Thus, a typical subsequence of a rotator that produces sentences $\mathbf{r}^n\mathbf{l}^n$, $\mathbf{r}^n\mathbf{l}^{n+1}$, $\mathbf{r}^{n+1}\mathbf{l}^n$ looks like the line below. Individual sentences have been underlined for clarity.

$$\ldots\underline{\mathbf{r}^n\mathbf{l}^{n+1}}\,\underline{\mathbf{r}^n\mathbf{l}^n}\,\underline{\mathbf{r}^n\mathbf{l}^{n+1}}\,\underline{\mathbf{r}^{n+1}\mathbf{l}^n}\,\underline{\mathbf{r}^n\mathbf{l}^n}\,\underline{\mathbf{r}^n\mathbf{l}^{n+1}}\ldots \qquad \text{(Eqn 36)}$$

A language of sentences may be constructed by examining the families of sentences generated by a large collection of individuals, much like a natural language is induced from the abilities of its individual speakers. In this context, a language could be induced from a population of rotators with different rotational velocities where individuals generate sentences of the form $\{\mathbf{r}^n\mathbf{l}^n, \mathbf{r}^n\mathbf{l}^{n+1}, \mathbf{r}^{n+1}\mathbf{l}^n\}$, $n > 0$. The resulting language can be described by a context-free grammar and has unbounded dependencies; the number of **l**s is related

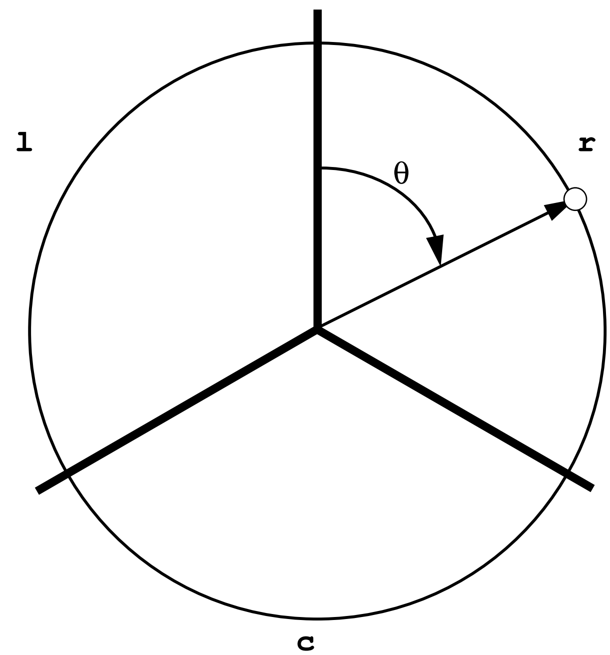


Figure 44: Decision regions that induce a context-sensitive language.

to the number of preceding **r**s. These two constraints on the language imply that the induced language is context-free.

Now I show that this complexity class assignment is an artifact of the observational mechanism. Consider the mechanism that reports three disjoint regions covering equal angles of rotation: **l**, **c**, and **r** (Figure 44). Now the same rotating point will generate sequences of the form

$$\mathbf{rr \ldots rrcc \ldots ccll \ldots llrr \ldots rrcc \ldots ccll \ldots ll \ldots} \qquad \text{(Eqn 37)}$$

For a fixed sampling rate, each rotational velocity specifies no more than seven sentences, $\mathbf{r}^{n}\mathbf{c}^{m}\mathbf{l}^{k}$, where n, m, and k can differ no by no more than one. Again, a language of sentences may be constructed containing all sentences where the number of **r**s, **c**s, and **l**s differs by no more than one. The resulting language is context-sensitive since it can be described by a *context-sensitive grammar* and cannot be context-free because it is the finite union of several context-sensitive languages related to $\mathbf{r}^{n}\mathbf{c}^{n}\mathbf{l}^{n}$.

Therefore, a single population of rotators with different rotational velocities exhibited sentential behavior describable by computational models from different generative classes, and the class depended upon the observation mechanism.

6.6.1 The Rotator

This section contains detailed derivations of the languages observed in the rotator. First, I show that the two region case produces a context-free language. The derivation of the three region case parallels this derivation.

The specific ordering of symbols in a long sequence of multiple rotations is dependent upon the initial, assumed random, rotational angle of the system. For a fixed rotational velocity (rotations per time unit), ω, and sampling rate, s, $(\omega < 2s)$ the observed system will generate sentences of the form $\{\mathbf{r}^{n}\mathbf{l}^{m-n}\}$, where $n = \left\lfloor \frac{s}{2\omega} - \phi \right\rfloor$, $m = \left\lfloor \frac{s}{\omega} - \phi \right\rfloor$,

and ϕ is a random variable such that $0 \le \phi < 1$. The value of ϕ embodies the slippage of n and m due to incommensurate rotational velocities and sampling rates. If s is an integer multiple of ω, then no matter the value of ϕ then values of n and m will be constant. If s is an irrational multiple of ω, then the current value of ϕ will produce minor (no more than 1) variation in the value of n and m.

For a fixed sampling rate, each rotational velocity specifies up to three sentences, $L(\omega) \subseteq \{\mathbf{r}^{n}\mathbf{l}^{n}, \mathbf{r}^{n}\mathbf{l}^{n+1}, \mathbf{r}^{n+1}\mathbf{l}^{n} | n = \left\lfloor \frac{s}{2\omega} - \phi \right\rfloor\}$ that repeat arbitrarily according to the divisibility of s and ω. We can then induce a language from a *population* of rotators with different rotational velocities, thus $L_2 = \bigcup_{\omega \in (0, s)} L(\omega)$. L_2 contains all sentences of the form $\{\mathbf{r}^{n}\mathbf{l}^{n}, \mathbf{r}^{n}\mathbf{l}^{n+1}, \mathbf{r}^{n+1}\mathbf{l}^{n}\}$, $n > 0$. The resulting language can be described by the *context-free grammar*

$$(\{S\},\{\mathbf{r},\mathbf{l}\},S,\{S\text{->}\mathbf{r},S\text{->}\mathbf{l},S\text{->}\mathbf{rl},S\text{->}\mathbf{r}S\mathbf{l}\}). \quad \text{(Eqn 38)}$$

No regular grammar can describe this language due to the unbounded dependency between the number of **r**s and **l**s. Therefore L_2 is a context-free language.

The three region case is similar. For a fixed rotational velocity, ω, and sampling rate, s, $(\omega < 3s)$ the observed system will generate sentences of the form $\{\mathbf{r}^{n}\mathbf{c}^{m-n}\mathbf{l}^{k-n-m}\}$, where $n = \left\lfloor \frac{s}{3\omega} - \phi \right\rfloor$, $m = \left\lfloor \frac{2s}{3\omega} - \phi \right\rfloor$, $k = \left\lfloor \frac{s}{\omega} - \phi \right\rfloor$ and $0 \le \phi < 1$. For a fixed sampling rate, each rotational velocity specifies up to seven sentences, $\mathbf{r}^{n}\mathbf{c}^{n}\mathbf{l}^{n}$, $\mathbf{r}^{n}\mathbf{c}^{n}\mathbf{l}^{n+1}$, $\mathbf{r}^{n}\mathbf{c}^{n+1}\mathbf{l}^{n}$, $\mathbf{r}^{n+1}\mathbf{c}^{n}\mathbf{l}^{n}$, $\mathbf{r}^{n}\mathbf{c}^{n+1}\mathbf{l}^{n+1}$, $\mathbf{r}^{n+1}\mathbf{c}^{n}\mathbf{l}^{n+1}$, $\mathbf{r}^{n+1}\mathbf{c}^{n+1}\mathbf{l}^{n}$. Let L_3 equal the union of all sentences generated from the rotational velocities in $(0, s)$. As before, $L_3 = \bigcup_{\omega \in (0, s)} L(\omega)$. L_3 contains all sentences of the form $\mathbf{r}^{n}\mathbf{c}^{n}\mathbf{l}^{n}$, $\mathbf{r}^{n}\mathbf{c}^{n}\mathbf{l}^{n+1}$, $\mathbf{r}^{n}\mathbf{c}^{n+1}\mathbf{l}^{n}$, $\mathbf{r}^{n+1}\mathbf{c}^{n}\mathbf{l}^{n}$, $\mathbf{r}^{n}\mathbf{c}^{n+1}\mathbf{l}^{n+1}$, $\mathbf{r}^{n+1}\mathbf{c}^{n}\mathbf{l}^{n+1}$, and $\mathbf{r}^{n+1}\mathbf{c}^{n+1}\mathbf{l}^{n}$, where $n > 0$. The resulting language can be described by the *context-sensitive grammar*

$$\begin{array}{c}(\{X, Y, Z\}\ ,\ \{r, c, l\}, X \\ \{X \to aXY, X \to aZ, ZY \to bZc, cY \to Yc, Z \to bc\}\ )\end{array} \qquad \text{(Eqn 39)}$$

Since L_3 is the finite union of several context-sensitive language related to $\mathbf{r}^n\mathbf{c}^n\mathbf{l}^n$, no context-free grammar can describe this language. Therefore, L_3 is a context-sensitive language.

6.6.2 Empirical Verification

The two languages observed in the family of rotators can also be observed in the dynamics of a single deterministic system. A slow-moving chaotic dynamical system controlling the rotational velocity parameter in a single system can express the same behavior as a population of rotators with individual rotational velocities. The equations below describe a rotating point with Cartesian location (x, y) and a slowly changing rotational angle θ controlled by the subsystem defined by w, z, and ω.

$$\begin{aligned}
x_{t+1} &= \tanh(x_t - \theta_t \tanh y_t) \\
y_{t+1} &= \tanh(y_t + \theta_t \tanh x_t) \\
w_{t+1} &= 4w_t(1 - w_t) \\
z_{t+1} &= \frac{4}{5}y_t + \frac{1}{5}w_t \\
\omega_{t+1} &= \omega_t\left(1 + \omega_t\left(w_t - \frac{1}{2}\right)\left(\frac{1}{2} - \frac{1}{2}\tanh 5z_t^2\right)\right) \\
\theta_{t+1} &= \frac{1}{5}\theta_t + \frac{4}{5}\omega_t
\end{aligned} \qquad \text{(Eqn 40)}$$

This system slowly spirals around the origin of the (x, y) plane. The value of w is a chaotic noise generator that is smoothed by the dynamics of z and rotational velocity of ω.

As before, I construct two measurement mechanisms and examine the structures in the generated sequences of measurements. The first measurement device outputs an **r** if x is greater than zero, and an **l** otherwise. From this behavior, the graph in Figure 45 plots the number of consecutive **r**'s versus the number of consecutive **l**'s. The diagonal line is indicative of a context-free language as a simple corollary to the pumping lemma for regular languages (Hopcroft & Ullman, 1979).

Determining if a language is context-free is hard enough given a mathematical description of the sentences, but what do you do when you have to answer this question about a set of strings defined by a set of examples? To solve this problem in the context of this chapter, I have focused on a particular structure found in a sample of strings, namely pairs of run lengths for symbols in the language. The regularities present in these strings allow us to rule out classes of automata capable of generating the observed strings.

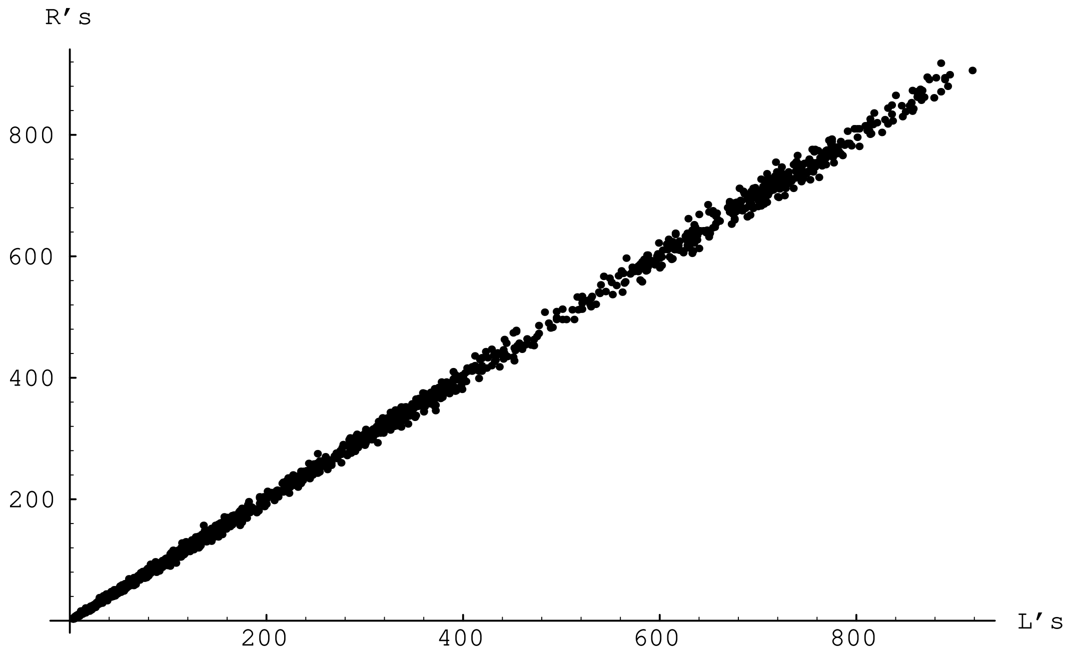


Figure 45: The two symbol discrimination of the variable rotational speed system.

The languages I examine in this paper possess two important properties:

Property 1: Each string has one run, or subsequence of the same symbol, for each symbol in the language.

Property 2: The runs are always in the same order.

Consider, for example, the string **`rrrrlll`**. This string is in the language due to Property 1 since it contains one run of **`r`**s and one run of **`l`**s, while **`rrrrllllrr`** is not in the language since it has two runs of **`r`**s. According to Property 2, if **`rrrrlll`** is in the language, then **`llllrrrr`** is not.

The first language I am interested in comes from measurements based on two regions. The strings in this language are from the strings defined by $\mathbf{r}^n\mathbf{l}^m$. One way of representing this language is by plotting points on a Cartesian grid where the (x,y) location is determined by the number of **`r`**s and the number of **`l`**s in each string. With help from the pumping lemma for regular languages, a few predictions can be made about this graph when $n + m$ is large, if the underlying language is regular. (The quantity $n + m$ is large when it is greater than the number of states in the minimal finite state generator for the language.) Here, one would expect to find pumped revisions of $\mathbf{r}^n\mathbf{l}^m$; there exists some as-

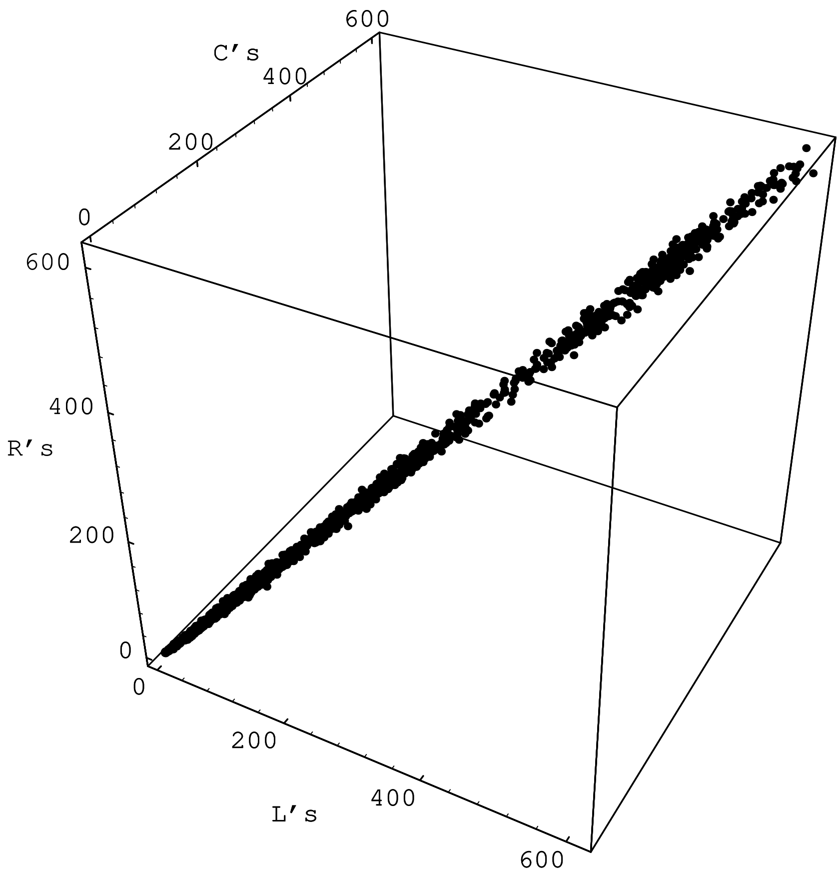


Figure 46: The three symbol discrimination of the variable system.

signment of u, v, and w such that $uvw = \mathbf{r}^n\mathbf{l}^m$ in that the set of strings uv^iw, for $i > 0$, is also in the language. Since the graph plots number of consecutive **r**s versus the number of consecutive **l**s, the uv^iw relationship constrains straight lines in the graph to be either vertical, as in the case of v being all **l**s, or horizontal, as in the case of v being all **r**s. The partition v cannot be a string of the form $\mathbf{r}^a\mathbf{l}^b$, because uv^iw would violate the general properties of our languages. Even in the general case, such a partition would lead to graphs without straight lines.

The straight line seen in the empirical data is *diagonal*. Because neither horizontal nor vertical lines are present in the graph, we are forced to conclude that the underlying finite state generator for this language either has a very large number of states, or it simply does not exist. The former is ruled out if we assume that the diagonal structure extends to strings of every length.

When the granularity of the measurement device changes from two regions to three, a parallel change occurs in the class of the measurement sequence from context-free to context-sensitive. Figure 46 shows the relationship between the number of consecutive **r**'s, consecutive **c**'s, and consecutive **l**'s. As in the previous case, one can interpret the diagonal line in the graph as the footprint of a context-sensitive generator.

A similar argument can be made against the context-freeness of the language produced by the three-region measurement system. As before, the elements of the language are represented on a lattice according to the run lengths of their substrings. A three-dimensional lattice, however, replaces the two-dimensional lattice of the previous case. Similarly, constraints on straight lines emerge from the intersection of the set of strings predicted by the pumping lemma for context-free languages and set of strings described by $\mathtt{r}^{n}\mathtt{c}^{m}\mathtt{l}^{k}$. If the language is the product of a push down automata, one could see straight lines in which one or two dimensions are varied. But the major diagonal exhibited by the data from the rotator system implies that there is either a large number of states in the push-down automata or that the PDA does not exist. The former is ruled out again, if the diagonal structure extends to strings of every length.

If the underlying language is regular then one would expect, according to the pumping lemma, to find pumped revisions of $\mathtt{r}^{n}\mathtt{l}^{n}$, (there exists some assignment of u, v, and w such that $uvw = \mathtt{r}^{n}\mathtt{l}^{n}$ that indicates that the set of strings $uv^{i}w$, for $i > 0$), also in the language. Since the graph records the number of consecutive $\mathtt{r}$s versus the number of consecutive $\mathtt{l}$s, the $uv^{i}w$ relationship constrains straight lines in the graph to be either vertical, as in the case of v being all $\mathtt{l}$s, or horizontal, as in the case of v being all $\mathtt{r}$s. If v is a string of the form $\mathtt{r}^{a}\mathtt{l}^{b}$, then the graph would not contain any straight lines. A formal proof appears in the appendix.

6.7 Discussion: The Observers' Paradox

The preceding example suggests a paradox: the variable speed rotator has interpretations both as a context-free generator and as a context-sensitive generator, depending upon measurement granularity. Yet how can this be if the computational complexity class is an inherent property of a physical system, like mass or temperature? What attribute of the rotator is responsible for the generative capacity of this system? There is no pushing and popping nonterminal symbols from an explicit stack. The rotator, however, does have a particular invariant: equal angles are swept in equal time. By dividing the orbit of the point into two equal halves I have ensured that the system will spend almost the same amount of time in each decision region. This constraint "balances the parentheses" in our measurement sequence. One may argue that the rotational velocity and the current angle together implement the stack and therefore claim that the stack is really being used. Such an argument ignores the properties of the stack, namely the ability to arbitrarily push and pop symbols. Stories involving internal Turing machine tapes are similarly misguided.

The decision of the observer to break the infinite sequence of symbols into sentences can also affect the complexity class. Similar arguments for sentences of the form $\mathtt{r}^{n}\mathtt{l}^{n+a}\mathtt{r}^{n+b}$, $(|a|, |b| < C)$ produces a context-sensitive language. From this perspective, Crutchfield and Young clearly biased the languages they found by assuming closure under substrings. Their assumption (Crutchfield & Young, 1989), if string x is in language L then all substrings of x are also in L, affected the induced minimal automata and criticality languages.

Because the variable speed rotator has two disjoint interpretations, *the computational complexity class cannot be an intrinsic property of a physical system;* it emerges from the interaction of system state dynamics and measurement as established by an observer. The complexity class of a system is an aspect of the property commonly called computational complexity, a property defined as the union of observed complexity classes over all observation methods. This definition undermines the traditional notion of system complexity, namely that systems have unique well-defined computational complexities. Consider the case of a sphere painted red on one side and blue on the other. Depending upon the viewing angle, an observer will report that the ball is either red, blue, or dual colored. It is a mistake, however, to claim that "redness," in exclusion of "blueness," is an intrinsic property of the ball. Rather, "color" is a property of the ball and "redness" and "blueness" are mere aspects of this property.

An observation such as the one described in this chapter should not be surprising considering the developments in physics during the first half of this century. The observation methods described above can select computational complexity in the same manner that observation of a quantum phenomenon collapses its wave function of possible values. Specifically, the wave/particle duality of radiation is an example of how observation can affect the apparent properties of a system. Depending upon experimental setup, a beam of light can either display wave-like diffraction or particle-like scattering.

As demonstrated above, strategic selection of measurement devices can induce an infinite collection of languages from many different complexity classes. For a single physical system, the choice of method and granularity of observation also "selects" the computational complexity of a physical system.

6.8 Conclusion

The goal of most modelers in cognitive science has been to build computational models that can account for the discrete measurements of input to and output from a target system. The holistic combination of the organism and symbolizing observer can create apparent computational systems independent of the actual internal behavior producing processes. Our examples show that the resulting computational model embodies an apparent system that circumscribes the target processes and the measurement devices. The apparent system has apparent inputs and outputs given by the symbolic inputs to and outputs from the computational model. For both input and output, the one-to-many mappings entailed by symbolization are not unique and can have simultaneous multiple symbol reassignments depending upon the observer. As the multiple interpretations of the rotator shows, these reassignments can change the computational complexity class of this apparent system.

I believe the results described above have relevance for cognitive science. Recall that both the Physical Symbol System Hypothesis and generative linguistics rest on an underlying assumption of the intrinsic nature of computational complexity classes. It suggests, on the surface, the irrelevancy of the hierarchy of formal languages and automata as accounts of complexity in physical systems. At a deeper level, it implies that we cannot know the complexity class of the brain's behavior without establishing an observer since the brain itself is a physical system. Thus the Physical Symbol System Hypothesis relies on an unmentioned observer to establish that an ant following a pheromone trail is not computational while problem-solving by humans is. The necessary and sufficient conditions of universal computation in the Physical Symbol System Hypothesis provide no insight into cognitive behavior; rather, it implies that humans can write down behavioral descriptions requiring universal computation to simulate.

Even the computational intractability of models of linguistic competence (e. g., (Barton et al., 1987)) is dependent on a particular symbolization of human behavior, not an underlying mechanical capacity. This highlights the groundless nature of rejections of mathematical models solely on claims of insufficient computational complexity. This work suggests that alternative mechanisms and formalisms that exhibit apparent complexities of the sort attributed to the "language module" should also be explored. In fact, the generative capacity of natural language may reflect the observational process rather than any linguistic mechanism:

> The daily warmth we experience, my father said, is not transmitted by Sun to Earth but is what Earth does in response to Sun. Measurements, he said, measure measuring means.[6]

As our ability to establish good measurements has increased, we now know that there are many areas in nature where unbounded dependencies and systematic forms of recursive structuring occur. The genome code, the immunological system, and botanical growth are but a few examples that are proving as complex as human languages. Physics could accept wave/particle duality as a product of observation. It is only cognitive science that presumes the "specialness" of language and human mental activity to justify a different set of scientific tools and explanations based upon the formal symbol manipulation capacity of computational models. To truly understand cognition, we may have to stop relying on symbolic models with fixed complexity classes and turn to explanations whose apparent complexity matches our observations.

6. Cage ((1969), p. 7).

CHAPTER VII

COMPUTATION IN RECURRENT NEURAL NETWORKS

7.1 Introduction

This chapter marks the return to the general theme of this dissertation: understanding the computational aspects of recurrent neural network models. In Chapter V, I provided an alternative explanation for the internal state behavior of recurrent networks. It was shown that discrete selection of continuous state space mappings induces a finite set of nonlinear state space transformations. This induction occurs whatever the underlying architecture. The study of iterated function systems (Barnsley, 1988), composed of finite collections of affine transformations, pointed to several phenomena common to both systems. First, we could describe IFSs and recurrent networks as unions over multiple transformations, which is a process capable of producing fractal state space attractors. Despite their overall attracting dynamics, these infinitely complex limit sets arise from the interaction of attractor transients belonging to the individual transformation. Second, I argued that compositions of these state transforms, selected by input, produced the often reported phenomenon of state clustering (Servan-Schreiber, Cleeremans, & McClelland, 1988). Finally, finite discrete state models were shown to be incapable of capturing the processing regularities of these recurrent networks. Thus, iterated function systems, when compared to finite state automata, were found to serve as a better explanadum of recurrent network state evolution.

Explaining recurrent network behavior via IFS's, however, does not directly answer the original question that lies at the heart of my investigation: How does computation arise in recurrent networks? The IFS theory provides a useful tool for explaining state space trajectories and regularities seen across many models. Yet it fails to characterize the computations produced by the models. Thus, I detoured from the discussion of recurrent networks in Chapter VI and considered the effects of observation on the *ascription of computational abilities* to physical systems. Although many scientists attribute the complexity of a behavior to the mechanism producing the behavior, I demonstrated how changes in the method of observation can radically alter both the number of apparent states and the apparent generative class of a system's behavioral description. Specifically, the method of dividing the state space and the identification of salient "sentences" are "observables" contributing to the attributed complexity of the physical system's behavior in two ways: they affect both the apparent complexion, as measured by the number of states, and the apparent generative complexity, also known as complexity class. Complexion changes because the granularity of discrete observation induces a different set of permissible state changes. The choice of boundary class that induces sets of sentences, either by subset or segment, can also change the apparent grammatical class of the observable sequence.

With these preliminaries aside, I will return to the topic of recurrent neural networks and discuss the implications of IFS theory and the Observers' Paradox to the understanding of computational abilities of recurrent networks. This chapter will provide a coherent account of the emergence of computationally complex behavior of recurrent neural networks from three different sources: internal dynamics, input modulation, and observation. Before tackling this unification theory, I will remind the reader how the notion of computation developed through the beginning of the Twentieth Century.

7.2 Computation and Effective Process

The rapid growth of computational science and engineering over the last six decades can be traced to the development of explicit symbol processing by means of an "effective process." The Church/Turing thesis claims that any effective process can be described in terms of a Turing machine. Turing based his original hypothesis about the nature of computation on the assumption that his abstract machine captured the important intuitive aspects of effective process (Turing, 1936). But what is an effective process, and how does it relate to physical systems?

To answer the first question, consider the academic climate preceding the invention of Turing's mechanism. Almost twenty years before Turing entered Oxford College, a famous mathematician Hilbert initiated a plan to exorcize many philosophical difficulties haunting the foundations of mathematics. He communicated this concern to his peers in his famous twenty-three problems for mathematics lecture to the International Congress of Mathematicians in 1900. Many of these problems defined the research agendas of mathematicians around the world and succumbed quickly to their Herculean efforts. The tenth problem, however, resisted the efforts of even the brightest researchers of the time. Hilbert asked: does there exist an effective process for determining whether a given Diophantine equation is solvable in whole numbers? Hilbert's attempt to clear up other puzzles like this led him to invent the notion of formal systems. These systems relied upon symbols to unambiguously represent mathematical concepts, as in the case of the symbol "$\sqrt{-1}$" refers to the square root of one. Mathematics, in his view, involved the manipulations of finite sets of such symbols by a finite set of rules. Formal systems, as envisioned by Hilbert, would avoid many ambiguities introduced by natural language and the irreproducibility resulting from the unconscious use of intuition.

The Tenth Problem remained unsolved until 1936. Alonzo Church, a Princeton University professor, and Alan Turing, a student at Oxford College, provided separate proofs that answered the question negatively: no effective procedure could exist which decided the solvability of Diophantine equations. Church relied upon his newly developed notion of recursive functions to prove his theorem. This theory posited the existence of a finite set of characters, known as an alphabet, and a finite set of functions defined over the set of strings formed by the alphabet. Church realized that he could model the art of proving theorems with these primitives. Starting from a string of alphabet symbols representing a set of axioms, a sequence of functions could be applied to the initial sequence. This application would generate a sequence of new strings that one could interpret as theorems. In fact, one could represent the functions as sequences of symbols describing the necessary operations to carry out the function. Because of this representational capability, the lambda calculus functions could manipulate descriptions of other lambda functions. A miracle occurred and the immense power of simulation via description allowed Church to demonstrate the undecidability of certain questions, including many of Hilbert's challenges.

Turing, unaware of Church's developments in the United States, searched for a direct analogy to theorem proving. Since a mathematician must write down his proofs for the benefit of others, he imagined using a special kind of typewriter for a hypothetical theorem prover, a typewriter in which the operator could only see the current location of a large piece of paper. While annoying, this constraint possessed some level of realism; real typewriters present only a fixed amount of type available for immediate examination and/or correction. Unfortunately, a typewriter cannot prove theorems by itself. Thus, Turing began to consider the role of the typist/operator in his theorem prover. Following Hilbert's lead, Turing clearly wanted to avoid relying on the intuition of typist. Despite the many counterexamples, intuition is often wrong and difficult to communicate and replicate. To avoid this well-known pitfall, Turing decided to provide the operator, aka the mechanical theorem prover, with a finite set of simple instructions. The controlling symbols, now known collectively as a program, could unambiguously specify the behavior of the typewriter. Turing foresaw no problems with misinterpretation—each part of the process was simple, discrete, and deterministic. From his standpoint, anybody interested in proving theorems could read and carry out instructions like move forward, backspace, and write a 1. By following these instructions, the operator could solve complicated problems and perform intricate calculations. Like Church, Turing unleashed the true power of this system when he realized that he could write the instructions with the same symbols used by the typewriter. The theorem prover, like the lambda calculus, could manipulate and simulate the operation of other theorem provers.

7.2.1 Emergent Universality

Church and Turing introduced a critical aspect of effective process that Hilbert missed. While Hilbert focused on eliminating the problems of ambiguity, others recognized the power of interpreting descriptions of effective process. The models proposed by Turing and Church could simulate their own execution substrates. One could write, for instance, a Turing machine program that effectively executes a Turing machine description on a given input string. This machine has a name: the Universal Turing Machine. It is universal because it can simulate any finitely described Turing machine operating on a finite input tape. Similarly, Universal Lambda Functions exist that operate on other lambda functions and their arguments. The observation that effective processes could manipulate their own descriptions, the notion of universality, has proved to be one of the most influential discoveries of the Twentieth century.

Not only can we design these mechanisms to simulate their own operation and systems described under the same formalism, like the universal Turing machine, they can provide the executive basis for other models of computation. Simulation can transform a Turing machine into a stored program computer, a rewrite system, or a lambda function evaluator. The most important feature of these models that makes them so amenable to simulation is explicit control mechanisms. The explicitness of these mechanisms has allowed us to understand and exploit computational systems of many varieties through the power of simulation.

The universal power of simulation suggests that the essence of computation does not lie in the details of the mechanism. Arbitrary head movements and Turing machine control state are no more necessary features of computation as are lambda functions, production rules, and registers. The explicit nature of these control systems, and many others, allows for infinite regression of implementation: the Turing machine implemented on a stack machine implemented on a register machine implemented on a random access machine, and so on. Every computational formalism has sufficient features that allow exhibition of potentially-universal computational behavior. Each sufficient feature, no matter the model, is describable in terms of another set of sufficient features. Therefore, no single necessary feature drawn from a single computational formalism can exist that applies across the complete set of computational mechanisms. In other words, the incomprehensible range of implementations—and the reductions they allow—will foil any attempt to draw substantial mechanistic preconditions for deterministic computation. This situation only gets worse when we allow computation to encompass other cornerstones of theoretical computing: oracles, stochastic state, and nondeterministic choice.

One consequence of having a toolbox full of computational mechanisms is that it allows us to characterize *everything* as effective process. Once we symbolize our observations, effective process emerges. Computational mechanisms merely describe this emergent process in a convenient notation. Without a set of necessary conditions to distinguish between computational and noncomputational processes, the notion of effective process deflates as we recognize its vacuousness. The symbolization detaches the effective process from it original causal medium and transplants it into a universe of unbounded multiple realizability. Searle (1992) offers similar arguments denying the utility of computation in understanding phenomena, such as consciousness, emerging from biological substrates.

Most traditional mechanisms employ explicit control of information processing states. Symbol processing may also occur through implicit control, without the obvious use of explicit mechanisms. Rather than manipulate structural compositions of atomic symbols, implicit systems perform functional compositions with patterns of activation. An external agent must decree the manipulated objects as atomic or not. For instance, the field of neural networks has claimed to offer a subsymbolic foundation for cognitive models (Smolensky, 1988). The term implicit symbol processing correctly characterizes the nature of neural network processing. The input and output vectors employed by NETtalk represent input/output symbols, but the semantics of the internal symbols depend upon the granularity of observation: a collection of hidden unit representations may indicate a single symbol or a set of symbols. The recurrent autoassociative memory (RAAM) (Pollack, 1990) exploits this distinction also. The RAAM encodes trees and other data structures in fixed width vectors with the help of encoding and decoding functions implemented as feed-forward networks. A significant difference between traditional data structures and those stored in a RAAM is that terminal/nonterminal (node/leaf) distinction is arbitrary and does not necessarily depend upon the content of the current vector (node). Only an outside observer[1] knows that (0.9,0.3,0.8) is the nonatomic symbol for “dog” and (0.1,0.9,0.1) is the atom “d”. The network understands these symbols as well as the Chinese Room operator knows Chinese (Searle, 1990).

Are the observed patterns merely interpretations, or are they inherent in the phenomena? This question has crossed the minds of some people in the emergent computation field. I have reviewed Crutchfield and Young's work in previous chapters and Chapter VI presented my own work on observed complexity classes. We can draw a single overwhelming conclusion from this body of work. A trained eye can find computation anywhere: in a spinning wheel, in a Tinker Toy contraption, or in the mind.

Considering the histories of effective process and theorem proving, rational thought could be considered the original target of computational descriptions. From Aristotle to Boole, Hilbert to Turing, rational thought has begged for a formal mechanism beyond the faulty, unreliable implementations we deal with every day. Yet rationality is a property we instinctively attribute to an individual only after observing their behavior. No one seriously suggests that we judge the rational/irrational distinction by an *in vivo* examination of brain states. Many philosophers of computation have proposed this method with respect to identifying ongoing computation. Boyle (1992) defines computation as a physical process that causes change through the fitting of spatially-extended structures. Chalmers (Chalmers, 1994) claims that a system implements a given computation when its causal structure is isomorphic to the formal structure of the computation. As demonstrated above, deciding whether a system is computational should not rely on the causal pathways of the mechanisms, but as a qualitative judgment based upon its behavior. Do the finite state machines from the logistic function indicate computation or hallucination? Since the first step of ascription is observation, and I have already shown that discrete observation can introduce apparent computational complexity, clearly the act of isolating computation is just as subjective as identifying rationality.

These facts suggest that the essence of computation lies not in the details of the mechanism, but in the *subjective* account of the behavior of the mechanism. Central to the notion of computation is the fact that we can ascribe the property of discrete effective processing to a system, much in the same way we ascribe color as a property of an object or rationality as a property of an agent.

An alternative explanation of this type of symbol processing exists in the literature, namely emergent computation. The boldest statement is by Forrest (1991) where she identified three constituents for emergent computation:

- A collection of agents, each following explicit instructions.
- Interactions among the agents (according to the instructions), which form implicit global patterns at the macroscopic level, i. e. epiphenomena.
- A natural interpretation of the macrobehavior as computations.

The types of models explored by Forrest have biased her definition of emergent computation. These models included cellular automata, genetic algorithms, and neural networks. Researchers describe these systems as collections of interacting agents following simple instruction sets: the cellular automaton has a next state lookup table, neural net-

1. This observer may take the form of a person, a program, or another network with discrete output.

works perform simple weighted sums of input vectors, and genetic algorithm populations live and die according to an evaluation function. Just as her predecessors were blinded by the features of the computational systems they were familiar with, Forrest drew upon surface regularities and failed to explore the deeper relationships between these systems. This collection-of-agents viewpoint presumes a difference between standard computation (e. g. Turing machines) and emergent computation (e. g. cellular automata). She quickly acknowledges, however, that emergent computation is no more representationally powerful than any other model of computation. By the Church-Turing thesis, there exists Turing machines capable of generating any emergent computation. The real benefit of emergent computation over standard approaches, in her view, is its efficiency, flexibility, and groundedness.

As will be discussed below, I do not subscribe to a division between emergent and standard computation. Consider the case of a Turing machine with a thousand tape heads operating on the same tape. By all of Forrest's requirements, this mechanism displays emergent computation. If we remove a tape head, is it still emergent? If so, when does the k-tape Turing machine no longer exhibit emergent computation? While Forrest acknowledges the fact that there is a question to what extent epiphenomena are "in the eye of the beholder," from my vantage point I can only conclude that *all* computation is emergent.

Forrest's view of emergent computation differs from implicit processing in that the information processing is "epiphenomenal" to the underlying dynamics. In this sense, computation emerges from those dynamics. Implicit processing exploits some system regularity, like the feedforward neural network's nearly boolean operations or a Tinker Toy shift register. In emergent computation, however, the atomic entities of computation (e. g., variables and procedures) have no relationship with the agents producing the system's dynamics. A simple example can be found in the class of iterated unimodal functions. This class includes the logistic, cosine, and network bump functions discussed in earlier chapters. The universal period doubling bifurcation cascade, it turns out, has little to do with the numeric details of the map, but only with the qualitative shape of the map (Feigenbaum, 1983). Crutchfield and Young's studies of the logistic function and observed sequences of symbols that appeared produced by finite state generators with varying complexity. Where are these machines that have been induced from the data stream? It cannot be the logistic function since it does not act like a finite state machine. It cannot be the logistic function plus observation method either because two observers measuring the same system can extract different computational explanations.

Rather than try to locate *the* state machine for a system, we should consider computation an intentional property of the system (Dennett, 1971). Dennett defines intentional systems in the following manner: "A particular thing is an intentional system only in relation to the strategies of someone who is trying to explain and predict its behavior."[2] In people, we try to explain their behavior in terms of beliefs and desires. The folk psychological terms, such as "know" and "remember," reflect the possession of information and goal directedness according to Dennett. We can draw a similar mapping to computational systems. Computational systems possess information as data and programs that direct their actions. While one may bristle at the thought of ascribing goal-directedness to a computer program

2. (Dennett, 1971) page 221.

blindly executing instructions, consider that it is the hardware, and not the software, to which we ascribe goal-directedness. The computer tries to meet our expectations of its behavior. From this viewpoint, I like to think of computation as the folk psychology of physical systems in motion.

This section began by acknowledging the fulcrum used by Church and Turing to topple Hilbert's Tenth problem. They recognized that effective processes can manipulate descriptions of effective processes, as well as prove theorems or find roots. Explicit control mechanisms, such as the finite state control of a Turing machine, facilitate the simulation of one effective process mechanism by another. While the ability to compound layer upon layer of simulation points to many interesting proofs about the limitations of computation, the lack of a distinct bottom—it's turtles all the way down—suggests that there are no necessary features of computation that can differentiate between a computational and noncomputational system. For instance, some mechanisms, such as neural networks, lack explicit control of their computational mechanisms, but do possess implicit control. Recall the subsymbolic nature of NETtalk and the terminal/nonterminal distinction in RAAM. These mechanisms, along with the rotator languages of the previous chapter, suggest that computation is a subjective account of behavior. In the next section, I will describe how three different facets of recurrent networks contribute to the computations that emerge from these systems.

7.3 Origins of Computationally Complex Behavior

In this section I will provide a coherent picture of the emergence of computationally complex behavior of recurrent networks and many other systems. I propose that three different sources conspire in this emergence: internal dynamics, input modulation, and observation. In dynamical systems terms, internal dynamics refers to the laws of motion that guide the trajectory of the system. Input channel effects generalize the effects shown in Chapters IV and V regarding IFSs and the chaos game. Chapter IV detailed the effects of subjective observation. These three sources collaborate to spawn the emergence of computation.

7.3.1 The Internal Dynamics

Intuitively, a computational device should not be static: it should possess observables that (normally) change over time. In dynamical systems, this change is governed by an internal dynamic. Many scientists expect to find the roots of complex behavior within the internal dynamics. For instance, the finite state control of a Turing machine internally controls the evolution of the system through time. Such views permeate the foundations of the generative approach to linguistics, the cognitive approach to psychology, and the symbolic approach to artificial intelligence. The internal representation, as data and operations, lies at the center of this view. *It is a very enticing view as it is the internal processing, or dynamics, that we have most control over when we build models of cognitive processes*. Committing to this approach frees researchers to focus their energies on issues of representation, organization, and communication within the context of the dynamic.

Many abstract machines offer explicit control mechanisms. Identifying aspects of their design crucial to the generation of computationally complex behavior is fairly straight forward. Finite state automata, push-down automata, linear-bounded automata, and Turing machines all share finite state control but exploit different storage mechanisms that, in turn, produce differences in their behaviors. Alternatively, the same automata can come from a Turing machine by assuming restrictions on the control program:

- FSAs from unidirectional head movement,
- PDAs whose heads move one direction, but always write a blank on moving in the other direction,
- and LBAs never move past the ends of the input string.

The same can also be said for the grammatical frameworks corresponding to these automata.

Yet many abstract machines do not have an explicit control mechanism. Even if they do have an explicit internal control, sometimes emergent computational behavior has little to do with this mechanism. Crutchfield and Young (1989) examined the finite state generator (FSG) descriptions of symbol sequences produced by the logistic and other functions. The generators culled by their algorithm captured not only the sequence of the attractor but also all possible routes of starting the attractor. Consider the FSG descriptions of dynamical systems with control parameters near their critical values[3]. Approaching the critical parameter from one side, the system will exhibit an increase in deterministic structure. For instance, increasing the control parameter in the logistic function from 3.0 to 3.67, the logistic function goes through many period doubling bifurcations. Similarly, the number of observed states in the generator also doubles along the bifurcation cascade. The logistic function, and other systems, will produce highly complex random symbol sequences. The complexity is relative, though. If we assume a stochastic automaton the same sequence will have a one state generator. On the other hand, if deterministic automata are the proper modeling mediums, the same symbol sequence results in a generator with infinite state. Both generators emerge from the dynamics of the logistic mapping.

Dynamical systems with their control parameters tuned to critical values have another description. At criticality, the two forces of structure and randomness produce what appears as complex computation. Deterministic machines, capable of describing periodicities, lie to one side of the critical parameter. On the other side of the critical parameter, stochastic behavior drives the state machine accounts of the system. Thus, both the number of deterministic and nondeterministic states increase near these critical areas. A highly complex description, in terms of a minimal stochastic or deterministic automaton, of the symbolic dynamics is necessary at the criticality point. This boundary region is also known as the edge of chaos.

Criticality phenomena are not limited to iterated maps like the logistic function or tent map. Other researchers have come across the same long distance correlations indicative of criticality in a variety of systems. The most notable of these is Wolfram's study of the qualitative behavior of cellular automata (Wolfram, 1984). Cellular automata are discrete-state/discrete-time dynamical systems. A cell can store one state from a finite set of possible states and is connected to a neighborhood of cells. A cell examines its own state and the current states of its neighbors in calculating its next state. All cells update their

---

3. See page 109.

states synchronously with each time step. The cellular automaton differs from other computational models in that the composite state of the system changes globally rather than locally. The tape head of a Turing machine ensures that only one tape square can be changed per time step, while every cell of a cellular automaton could change state in a single time step. This difference does not affect the computational power: the ability to make infinite changes to the state is counterbalanced by the problem of communication. Global notification of local changes is limited by the CA's "speed of light," the maximum range of a cell's neighborhood.

Given even a simple cellular automata rule, such as Life[4] (Berlekamp et al., 1982), it is impossible to predict the limit behavior of the automata without running the automaton. The game of Life, beyond the beautiful pattern sequences it generates, provides a universal computational substrate. One can organize the initial states of the lattice to implement various logic gates and "wires" connecting them (Berlekamp et al., 1982). The existence of this one computational organization suggests the existence of many other configurations that also perform computation, but operate outside our ability to perceive them as such.

In order to understand the behavior of cellular automata, Wolfram identified four classes of cellular automata (CA) rules. Class I automata evolve to a homogeneous state, or limit points. Class II evolves to simple separated periodic structures, or limit cycles. Class III yields chaotic aperiodic patterns, or strange attractors. Class IV yields complex patterns of localized structures, or systems with extremely long transients to their attractors. These classes exhibited an interesting relationship: class IV rules seem to emerge from between the periodic class II rules and the stochastic class III rules. Class IV systems exhibit the long distance correlations indicative of criticality. As a test of his taxonomy, Wolfram examined the behavior of various one-dimensional cellular automata rules and classified them according to the conditions above.

Such descriptive systems always beg the question: how do I generate a rule from a particular class? Langton (1990) found a way of producing CA rules that exhibited the properties of a desired class. He created the λ parameter as a distribution measure for CA rules. Langton defined this parameter as the ratio of the number of times a particular state appeared on the right hand side of a transition rule to the number of transition rules. Assuming that we wish to calculate the λ measure of a CA with K states, neighborhood of n, and N transitions to an arbitrary quiescent state, q, Equation 41 shows how.

$$\lambda = \frac{K^n - N}{K^n} \qquad \text{(Eqn 41)}$$

By using this parameter to constrain CA rule construction algorithms, Langton produced periodic, chaotic, and indeterminate CA behavior. To generate a CA that fits a given λ, merely fill the rule table with N state q's and randomly, with a uniform distribution, as-

4. Each cell in a two-dimensional lattice has two states, on and off. At each time step, a cell is turned off unless there are either two or three cells in the immediate neighborhood of eight cells that are on. The cell is turned on in this case.

sign the remaining rule table slots with the other states. The likelihood of randomly generating Class I, II, III, or IV CA with a properly chosen λ is high.

There are no guarantees, however, that the rule generator will always produce rules within a class or that an arbitrary rule set will have an class-interpretable λ. While λ is useful in the random generation of CA rule sets with class IV behavior, the parameter is all but useless in *predicting* the behavior of a given set of rules (Mitchell et al., 1993). The parameter λ merely controls the rule set's distribution of states, much like the mean and standard deviation controls the sample generation of a random variable with a normal distribution. For example, a random variable with mean zero and standard deviation of one can generate a sequence that looks like it has a mean of 0.5. A CA possess multiple λ parameters, one for each state.

While Langton searched for the edge of chaos in cellular automata, the work of Sompolinsky and Cristiani (1988) provides some help in locating the edge of chaos in neural networks. They analyzed the effects of random weights in fully connected recurrent networks. These networks are similar to the networks described by Hopfield, however, the symmetric weight restriction is removed. Their results focus on the limit behavior of very large networks (i. e. as the number of nodes, N, increases toward infinity). In addition, they assume that the weights are selected from a normal distribution with a mean of zero. When the standard deviation of this distribution is greater than $\frac{1}{N}$, the network will display chaotic behavior. Below this cutoff, the network follows a periodic regime. Since recurrent network state transformations have the same structure as the networks Sompolinsky and Cristiani examined, we can use the $\frac{1}{N}$ standard deviation as a guideline for constructing recurrent networks with computationally complex behavior at the edge of chaos. Like Langton's λ method, this approach only works when generating neural networks with critical behavior and provides no analytical leverage in discovering the behavior class of any given recurrent neural network.

Changes in the internal dynamics affect the complexity of their behaviors. The grammatical complexity of Turing machines can be altered by enforcing constraints on the finite control. Cellular automata rules can exhibit a variety of behavior classes. Iterated maps and recurrent networks can undergo parameter-controlled bifurcation cascades that increase the number of apparent states they posses. In the following sections, it will become clear that these are not the only routes to create complex computational behaviors.

7.3.2 The Input

The previous section examined the role that internal dynamics plays in the expression of computation in recurrent networks. Recall that in Chapter V, I described how the transient behaviors of the individual transformations interact to produce the fractal state spaces. In this section, now I will focus on how input sequences can create virtual transformations whose complexity is greater than any of the individual iterations. It is these virtual transformations that give dynamical systems, such as recurrent networks, their input dependent computational power.

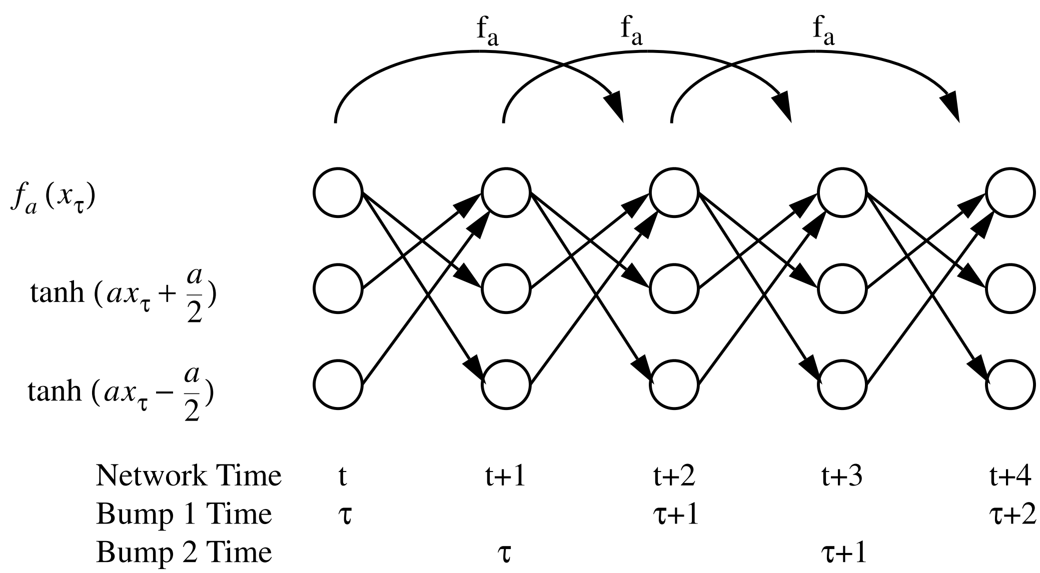


Figure 47: An SCN implementation the iterated unimodal function.

One can develop a better understanding of an iterated mapping's dynamics by examining the iterated compositions of the mapping. For instance, this method is useful for finding periodic behavior in functions like the logistic map. To find a period n orbit in the mapping f, just look at the composite function $g = f^{\circ n}$. The degree-n notation indicates the iteration of function$f$ n times. The points that satisfy $x = g(x)$ either have a period of n or have a subharmonic period of n ($\frac{n}{2}$, $\frac{n}{3}$, ...). One can also find out if these fixed points are attracting or repelling depending upon the derivative of g in the neighborhood of x.

Compositions are also useful in the analysis of iterated function systems like recurrent networks. The composition operation for IFS's differs from that of traditional iterated functions in that decisions can be made at each iteration as a transformation selection. For instance, consider a recurrent network whose input sequence may be selected from three different symbols, **a**, **b**, and **c**. If this network receives the sequence, **abc**, then the composition, $f_{\mathbf{abc}}(x) = f_{\mathbf{c}}(f_{\mathbf{b}}(f_{\mathbf{a}}(x)))$, describes the transformation of the network's internal state from what is was before encountering the sequence to the internal state it will be afterward.

The transform $f_{\mathbf{abc}}$ is a virtual transform: it exists only in the presence of a particular input sequence. An IFS with m transformations can have m^n virtual transformations of length n. Recall the constraint that the affine mappings occurring in traditional IFSs must be uniformly contractive. Whatever the sequence of compositions, the resulting composition for an IFS will also contract the state space. By relaxing the contractivity constraint, as

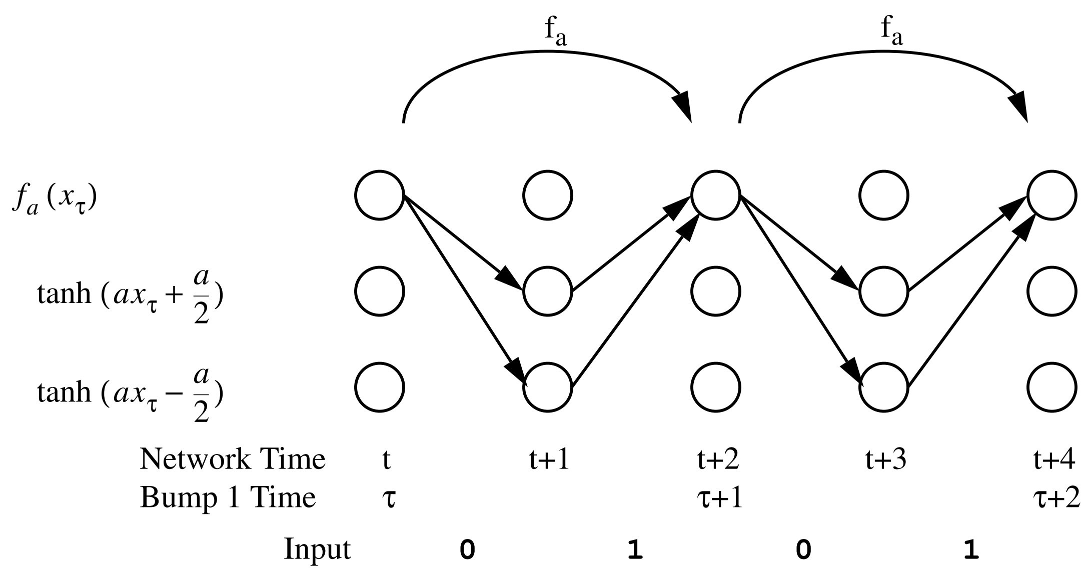


Figure 48: Spreading the bump over two iterations

in the case of recurrent networks, transformations and their resulting compositions can contract and/or expand the state space. In Chapter V, I described a recurrent network that displayed sensitivity to initial conditions. This recurrent network implements the iterated mapping found in Equation 27. This function can be expressed as a SCN transformation by spreading the iterated computation over two time steps. This is diagramed in Figure 47 and the SCN transformation is provided in Equation 42.

$$S^{(t+1)} = \tanh\left(\begin{bmatrix} a & -a & 0 & -a \\ 0 & 0 & a & \frac{a}{2} \\ 0 & 0 & a & \frac{-a}{2} \end{bmatrix} \cdot \begin{bmatrix} S^{(t)} \\ 1 \end{bmatrix}\right) \qquad \text{(Eqn 42)}$$

This network implements two iterated bumps: one during the even timesteps, the other during odd timesteps. The first element of $S^{(t+1)}$ is $f_a(x_{t-2})$, while the second and third elements of $S^{(t)}$ are $\tanh(ax_{t-2} + \frac{a}{2})$ and $\tanh(ax_{t-2} - \frac{a}{2})$. The network operates at three different time scales: that of the recurrent network, that of the first bump, and that of the second bump. The initial conditions of the three state units determine the initial state of the bump systems. Figure 28 on page 81.

There are many ways to implement Equation 27 in a recurrent network. For instance, it can be expressed as two SCN transformations by spreading the iterated computation of over two time steps. By relying on external stimulation, the network can alternate between two transformations given in Equation 43.

$$\omega_{\mathbf{0}}(S^{(t+1)}) = \tanh\left(\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & a & \frac{a}{2} \\ 0 & 0 & a & -\frac{a}{2} \end{bmatrix} \cdot \begin{bmatrix} S^{(t)} \\ 1 \end{bmatrix}\right) = \begin{bmatrix} 0 \\ \tanh\left(ax_\tau + \frac{a}{2}\right) \\ \tanh\left(ax_\tau - \frac{a}{2}\right) \end{bmatrix} \qquad \text{(Eqn 43)}$$

$$\omega_{\mathbf{1}}(S^{(t+1)}) = \tanh\left(\begin{bmatrix} a & -a & 0 & -a \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} S^{(t)} \\ 1 \end{bmatrix}\right) = \begin{bmatrix} f_a(x_\tau) \\ 0 \\ 0 \end{bmatrix}$$

The first transformation, $\omega_{\mathbf{0}}$, calculates $\tanh\left(ax_\tau + \frac{a}{2}\right)$ and $\tanh\left(ax_\tau - \frac{a}{2}\right)$. The second transformation, $\omega_{\mathbf{1}}$, combines these values into the desired output. This network exhibits several important differences from the previous implementation. First, this network is implementing a single sequence of bump iterations instead of the two entwined sequences of the original network. Second, the network requires a particular input sequence to evoke the desired behavior. These inputs will selectively compose the proper function from the primitives defined by the input symbols. Given an infinite sequence of **01**'s, one could view the resulting recurrent network dynamics as an iterated mapping without reference to the underlying individual transformations. This approach translates the computational force from the recurrent network to the input stream; a translation that avoids the language of representation transformation (Chapter II) underlying most computational descriptions.

The information processing approach assumes that input directs useful information into the system. Therefore, effective computational systems should have some method of acquiring information for internal processing. By connecting the inputs of a system to an environment, the system becomes an embedded system. One could even argue that the embeddedness of a system is a criterion for identifying physical computational systems. An embedded system must have inputs; otherwise, it cannot interact with the environment. I characterize the effects that input has on behavior with three levels of interaction. I do not believe these levels are discrete categories, but they provide useful labels in describing a continuum of effects.

First, the recurrent network could modulate the complexity of the input symbol sequence. Networks like this could copy the input symbol to the output symbol with little regard for the sequence history. For instance, combinatorial circuits always perform the same input to output mapping. These networks could also delay the input/output copying, by storing previous sequences in their internal state. Substitutions may occur between copying input to output, ranging from homomorphisms that replace substrings with single strings to general substitutions nondeterministically selected from a list of possibilities. At the other extreme, there exist networks whose output behavior complexity is independent of the input it receives.

Between these two extremes of copying and ignoring input reside in networks whose output complexity depends upon the input history of the network. These networks can amplify the complexity of the input sequence by allowing the input to construct virtual functions whose behavioral complexity is both greater than the input sequence and the complexity of other iteration composition function. The input sequence acts as a key to select the correct behavior from the set of possible behaviors. For instance, alternating the input symbols fed to the network portrayed in Figure 48 select the iterated bump behavior from the rest of the virtual transformations that produce the attractor.

Recall that complexity of symbolic dynamics arises from the competition between structure and disorder (Crutchfield & Young, 1989). From this viewpoint, complex systems do not exhibit either fixed point or random behavior. We can draw a similar conclusion after considering the effects of input on the dynamics of a system. Systems that either copy the input verbatim or entirely ignore the input stream are less complex than those whose behavior nontrivially depends upon the input. Hallucination and mimicry are not nearly as interesting as interaction.

These concepts extend the views of Herbert Simon (Simon, 1969). His ant parable is now the classic argument supporting the importance of environment to the expression of complex behavior. In Simon's gedanken experiment, he observes an ant crawling across the sand dunes of a beach. This observation moves him to contemplate the mechanisms governing the particular trajectory the ant takes during its journey. In Simon's words: "The apparent complexity of [the ant's] behavior over time is largely a reflection of the complexity of its environment."[5] Simon goes on to qualify this statement by postulating both an external environment (the world) and an internal environment (memory).

Two important differences separate my views from Simon's. First, Simon assumes two environments, the external environment of the outside world and the internal environment of the agent's "information packed memory." While this may simplify Simon's discourse, it creates an unnecessary dichotomy in the agent. Does the control of a given behavior originate from the environment or the information packed memory? If the information packed memory controls behavior, there is no way to detect if this memory is actively controlling the system, or providing data for another controller. Now the dilemma lies in discerning between data and control, yet we regularly recognize the interchangeability of the two. Rather than keep them separate, why not merge both data and control into a single mechanism? We have already seen this in other systems; Jordan (1986) describes his recurrent networks as motor program generators without explicit storage of a trajectory.

The second difference addresses the passive nature of the environment. Simon argues that the peaks, valleys, and slopes of the sand dunes gently nudge the ant's trajectory across the beach. In other words, the agent merely reflects characteristics of the environment. As Simon's demonstrations showed, the environment can interact with an agent by modulating the control systems of the agent. Reflection, or copying, indicates very little interaction between the environment and the agent. In this situation, the agent is no more than a communications channel sensitive to some aspect of the environment. If we view agents and communication channels, it is unclear whether we are watching internal information processing or environmental noise. In other words, one person's noise is another's computation.

5. (Simon, 1969) page 64.

In this section, I explored the effects that discrete input streams can have on recurrent networks. Rather than view the recurrent network as a transformer of input representations, I have suggested that the form of input has little to do with the operation of the network. The crucial aspect of the input is its permeance from encounter to encounter. That is, the vector for the symbol **`a`** remains constant whenever the network observes **`a`**. This fixed vector induces a fixed mapping within the network, which in turn transforms the current state to the next state. I gave a demonstration of a recurrent network where certain key input sequences can unlock complexities hidden in the individual transformations. These input sequences induce virtual transformations in the recurrent network and represent the emergence of new behaviors that no single internal state transformation could possibly produce.

Virtual transformations unlock a new approach to understanding the processing abilities of recurrent networks. Previous attempts to understand neural networks have focused on the input transformation properties of the systems. By treating the input as an active, rather than a passive, processing component expands our descriptive lexicon for neural network behavior.

7.3.3 The Act of Observation

So far I have outlined the effects of internal dynamics and input modulation on the computational abilities of systems. This section will address the effects of observation on those abilities.

In the previous chapter, I introduced the Observers' Paradox and explained its effects on the induction of grammatical competence of physical systems. The centerpiece of the chapter was a variable speed rotator that had interpretations both as a context-free generator and as a context-sensitive generator. The resulting interpretation depends upon measurement granularity. This holistic interaction between system and observer is an example of emergent computation: the observed system behaves as if it is governed by computational forces. By sweeping nearly equal angles in equal time, the system's dynamic ensured that the point spent almost the same amount of time in each decision region, a regularity the measurement device perceived as a sequence of nearly balanced parentheses. One may claim that the rotator employs a hidden stack and argue that the rotational velocity and the current angle together implement that stack. An argument such as this ignores the stack's ability to *arbitrarily* push and pop symbols. Simiarly, claims regarding an internal Turing machine tape are misguided.

Because the variable speed rotator has two disjoint interpretations (in this case $\mathbf{l}^n\mathbf{r}^n$ and $\mathbf{l}^n\mathbf{c}^n\mathbf{r}^n$), computational complexity classes emerge from the interaction of system state dynamics and observer-established measurement. The complexity class of a system is an aspect of the property commonly called computational complexity, a property I defined as the union of observed complexity classes over all observation methods. While this definition undermines the traditional notion of system complexity, namely that systems have unique well-defined computational complexities, it accounts for our intuitions regarding the limitations of identifying computation in physical systems.

This argument shows how observing the output of a system can lead to multiple hypotheses of internal complexities. Observations can also affect our descriptions of input/output behavior. The diagram in Figure 49 illustrates the inappropriateness of using sym-

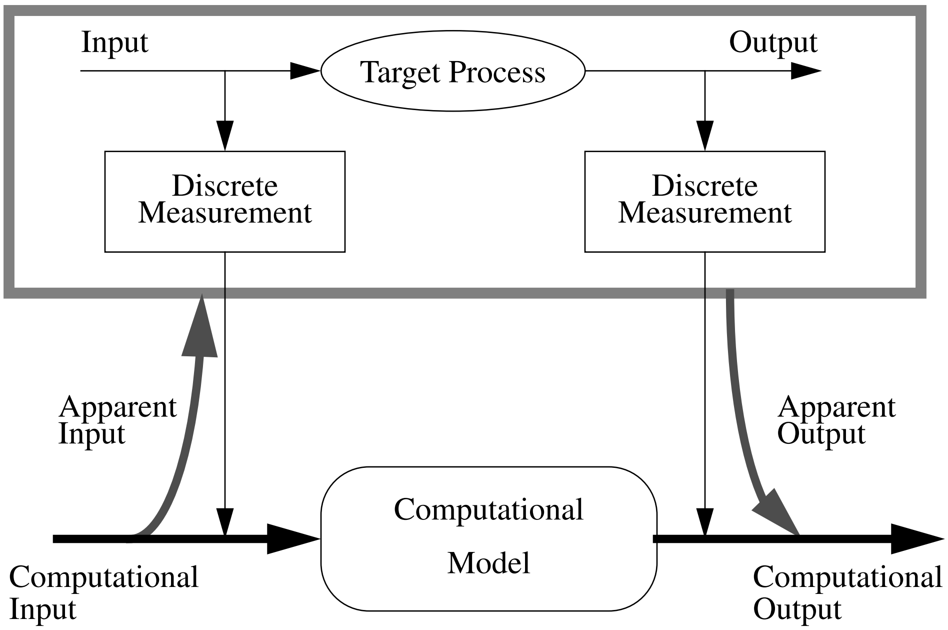


Figure 49: The relationship between a computational model and the process it models. The computational model must account for the behavior of the process, plus the apparent computation that emerges through discrete measurement. The grey box surrounding the target and measurements circumscribes the apparent system described by the computational model, while the grey arrows identify apparent input/output flow through the apparent system.

bolized behaviors as the basis of our models of natural phenomena. The computational model, a theoretical account of the relationships between observed symbols, describes the input/output behavior of an apparent system as observed by the measurement devices. While trying to capture the behavior of a target system, the model must account for both the discrete measurements of input to and output from the process, besides the processing internal to the target system.

The apparent system has an apparent output that are the symbolizations of the process' output. The outputs of the apparent system are symbolizations of quantities that most interest the modeler: an utterance, a decision, or a reaction. As demonstrated in Chapter VI, the operation of discrete measurement can either inflate or deflate the apparent complexity of the target process and thus corrupt the understanding of this process.

The apparent system also has apparent inputs that we measure from the inputs to the target process. A good computational theory describes the generation of output symbols in terms of these input symbols. As a symbolization process, the same hazards that threaten the act of collecting process inputs also affect the act of collecting system outputs. Yet there is another problem that emerges after a computational model is induced to explain an input/

output relationship. The phenomenon of apparent input arises when we assume that the input to the computational model is identical to the input entering the process. These inputs are not the result of a measurement, as were the apparent outputs. The computational inputs originate from outside the system, usually from a modeler interested in making predictions about the target process. The problem occurs when we assume that causality flows from the computational input through the target process. Notice in the diagram of Figure 49 how the apparent input flows against the direction of discrete measurement from the target process inputs. If there is any causality in this system, it flows from the input source, through a discrete measurement, through the computational model, and affects other systems by means of the computational output.

Let's assume, for the moment, that we have a useful computational model of our target system. In addition, we are using this computational description as a replacement of the target process. This is not as farfetched as it sounds: such replacements are the *modus operandi* of computer engineering. The relationship between apparent input and actual input is uncertain due to the many-to-one mapping of the measurement system that can show up as a nondeterministic or historical dependence of behavior. Consider the Bakers' mapping. Despite the mapping's determinism, any finite, discrete state description of its behavior will involve nondeterministic state transitions. Therefore, the input observation system is inseparable from the computational model because the observation mechanism transducing the apparent input sequence muddies the original complexity, given that there is any there in the first place.

Now consider the reverse situation: replacing the computational description with the target process. Now the many-to-one observation of the input must be inverted to convert the incoming symbols into some quantity that can drive the target process. Of all the possible alternatives for a given symbol, which one should we pick? This is the problem. Stochasticity enters at this point with us selecting one value from a vast collection of alternatives.

A similar situation occurs in the study of computational complexity in computer science: given a formal task, what is its complexity? The traditional method of proof is to reduce a problem of unknown complexity to one of known complexity. The complexity of the transformations involved in proofs of this sort are limited a priori to prevent them from overwhelming the complexity of the reduction. For instance, we allow deterministic polynomial time transformations of representations when we assume that the target process operates in nondeterministic polynomial time. Most of cognitive science tries to do the same thing. Here, the cognitive scientist tries to reduce the tasks of cognition to computational tasks. Unfortunately, the cognitive scientist cannot make this assumption since the processes required to observe and describe the target process are often identical. For instance, the act of transcription requires that the scribe understand the original utterances.

Other non-computational systems exhibit complexity amplification from observation. Consider the operation of two coupled oscillators. The first oscillator, the driver, emits a periodic output completely independent of the second oscillator. The driven oscillator reacts in a fixed manner (the coupling) to the driver. The resulting behavior of the driven oscillator could exhibit fixed, periodic, or chaotic motion. The selection of these regimes depends upon the nature of the linkage and the dynamics of the driven oscillator. The linkage could be viewed as an observation. Thus, the complexity of the driven oscillator depends upon its observation of the driving oscillator.

In this section, I reviewed the observers' paradox argument from Chapter VI and showed how it applies to the construction of computational models. Emergent computation arises from the holistic interaction between system and observer. The observed system behaves as if it is governed by computational forces, though we know that no such processes are involved. While the rotator example applies to the problem of characterized output behavior, the paradox can also affect computational descriptions of input/output behavior. While computational models can account for mappings from apparent input to apparent output, a computational model unaccompanied by its observation mechanisms is useless.

7.3.4 Synthesis

In the previous section, I described the three sources of emergent computation. These sources include the ability to constrain internal dynamics capable of supporting complex behavior, sensitivity to input in constructing virtual state transformations, and finally the output is an observation of the internal states of the network. In dynamical systems terms, internal dynamics refers to the laws of motion guiding the trajectory of the system. Input channel effects generalize the effects shown in Chapters IV and V regarding IFS's and the chaos game. The subjective effects of observation, detailed in Chapter VI, suggest that all systems lack a unique system complexity. These three sources collaborate to provide proper environmental conditions for the emergence of computation within the context of recurrent neural networks.

The preceding arguments show conclusively that these three conditions are sufficient for emergence, but are they necessary? Clearly, internal dynamics cannot be excluded. It provides the underlying energy that drives system change and computational systems must change state over time. Similarly, input dependency is also important because most of the time we identify computational systems as transducers of information. Even if a computational system ignored its input, as in the case of an enumerative generator, it still could be portrayed as utilizing input. Finally, the role of observation is inseparable from the notion of emergent computation in that the system must be observed doing *something*.

7.4 Entrainment Learning

In this section, I present an application of the philosophical stance described above. In cognitive science, the problem of learning spans many domains. One of the most interesting domains is language learning. The success of employing grammars as descriptions of linguistic knowledge has many researchers equating language learning with grammar acquisition. There are plenty of examples that suggest that natural language competency requires grammars of at least context-free, if not context-sensitive, generative capacity. While aesthetically pleasing, this assumption entails either intractability or uncomputability for all but the most restricted grammars.

The goal of entrainment learning is to model the apparent transfer of symbolic behavior without explicit transfer of structure. The current formulation requires both a teacher, whose internal structure remains constant during the transfer of information, and a student, whose internal structure changes over the course of the interaction. I have modeled these abstract entities as iterated one-dimensional maps, in particular the logistic function. This function was selected due to the rich nature of inherent dynamical complexity exem-

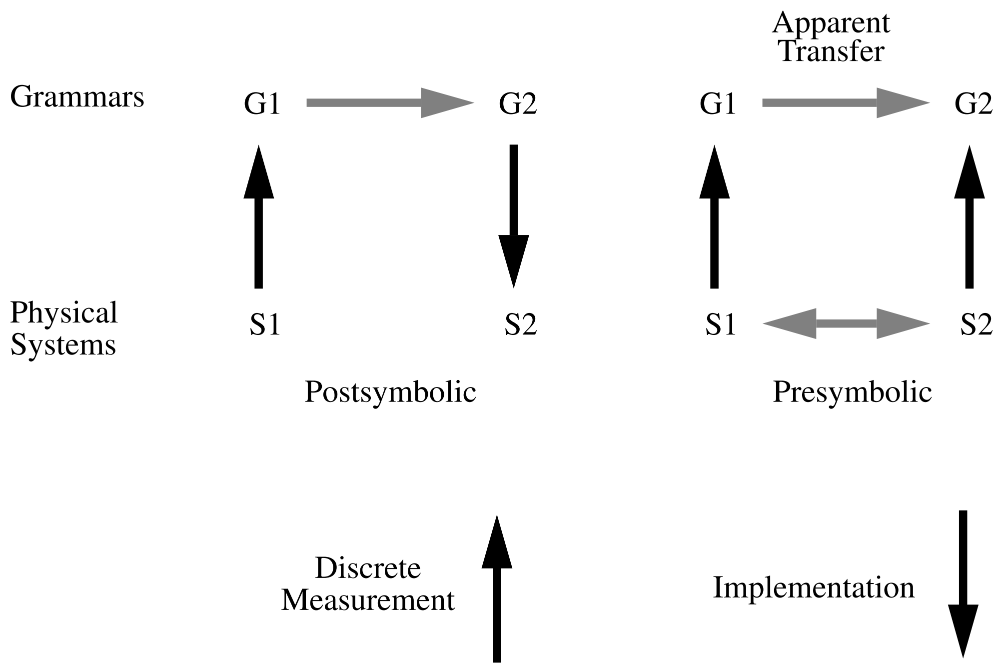


Figure 50: Postsymbolic and presymbolic learning

plified by the bifurcation diagram. Specifically, all unimodal iterated functions exhibit the same bifurcation route to chaos, implying that these function share the same symbol dynamics. Recall that symbolic dynamics, introduced in Chapter IV, is the study of dynamical systems by examining their trajectories after a discretization process. For example, a series of real numbers can be converted into a sequence of discrete symbols, say 0 and 1, depending upon which side of a threshold each value lies.

The underlying assumption of this work is that communication at a presymbolic level will accelerate the acquisition of ascribed symbolic behavior. Figure 50 shows a schematic of how postsymbolic and presymbolic learning differ. In each diagram we have two entities, S1 and S2. In post-symbolic learning, it is assumed that the entities exchange information only after a discrete measurement of the other's behavior. Presymbolic learning recognizes the physical reality of the situation: that the linkage between the two occurs before any measurement takes place. Thus, presymbolic learning produces an apparent transfer of information after discrete measurement.

One of the benefits of presymbolic learning is the ability to form multiple associations with a single symbol. Consider the case in which the teacher is trying to transfer the concept **`a`** before **`b`** or **`c`**. Such regularities are difficult to learn in the presence of noise without making a priori commitments about the number of such nondeterministic linkages. Since the presymbolic level provides an entire range of values for **`a`**, **`b`**, and **`c`** (due to the

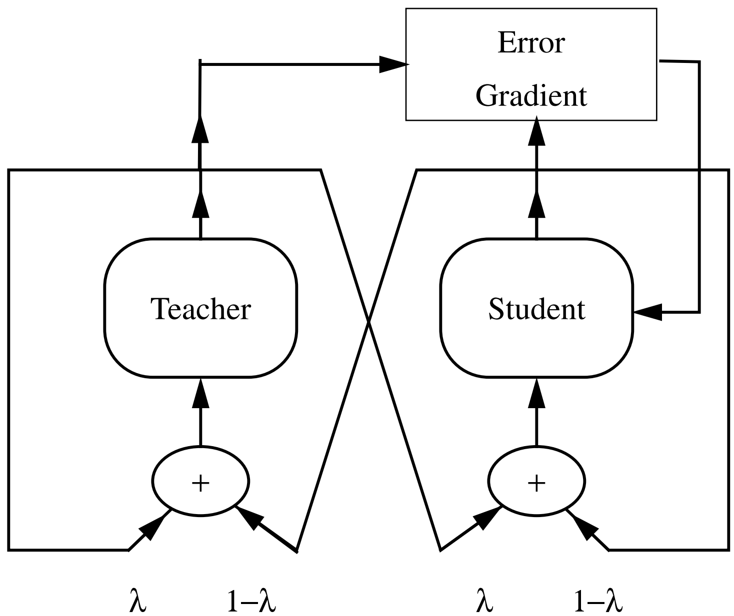


Figure 51: Schematic of entrainment learning system

discrete symbolization) the learning task is simplified because a subset of **a**'s values can map exclusively to **b**'s or **c**'s.

A second benefit provided by presymbolic learning emerges from the linking of the two entities. In the experiments that follow, linkage occurs as a linear interpolation of the inputs for each entity. Rather than let the teacher blather continuously and pay no attention to the student, the dynamics of the teacher also depends upon the dynamics of the student. For instance, if the student is currently approaching a fixed point, the teachers' trajectory is simplified. Because the student's internal structure is alterable, it can be shaped to generate the output of the teacher. When the student's external behavior has changed significantly, the dynamics of the teacher changes. This change appears as a bifurcation, namely the period of the teacher doubles. Again the student strives to imitate the teacher's output that leads to yet another bifurcation. The process continues until the system reaches an equilibrium in which the student's internal structure remains nearly constant. Figure 51 shows a caricature of the entrainment learning scheme illustrating coupling and structure modification.

Two sets of experiments to date have been carried out. In both experiments a logistic function ($\eta x (1 - x)$) has served as the teacher. The η parameter was fixed at 3.6 for the results reported below. In the first experiment, I compared the uncoupled learning case with the coupled case where the coupling constant was fixed at 0.9. The derivative of $(o_1 - o_2)^2$ with respect to η_2 was used to update the second logistic function's parameter in a gradient descent method (learning rate of 1.0). Without coupling, the student function slowly changed its η parameter, indicative of merely capturing the statistical regularity of

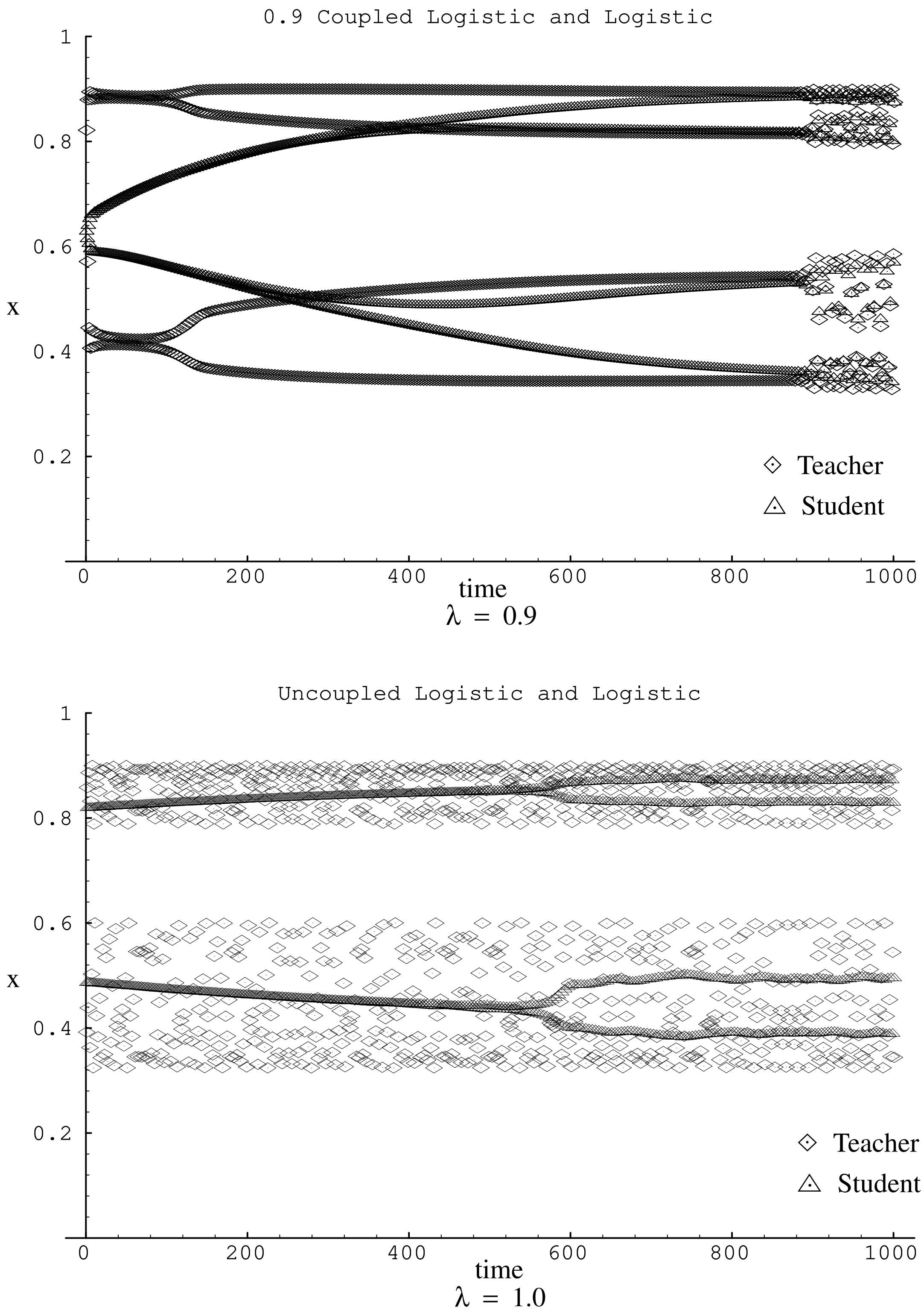


Figure 52: Entrainment data for logistic function teacher and student.

the high period signal. After 1000 iterations of the mappings, the student could only produce a period-four sequence. When the teacher was coupled to the student, a completely different dynamical regime appeared. The teacher started in a period-two attractor because of the student's fixed point behavior. A bifurcation in the student's trajectory then led to a period doubling in the teacher. Eventually, both settled into an identical complex trajectory.

Experiment two repeated this setup, except that the student was replaced by a 1-3-1 feed-forward neural network. This network was used iteratively and the output of the previous time step was the input for the current time step. A gradient descent method (back propagation) changed the weights of the network in the same manner as the learning system changed the logistic mapping's parameter in the previous experiment. The initial weights of the network were biased toward a bump transfer function. Again the same pattern appears (Figure 52). In the uncoupled case the network takes a long time to separate, while in the coupled case the network quickly tracks the changes in the teacher's attractor. This experiment shows how universality of unimodal functions' symbolic dynamics allows quantitatively different structures to evolve the same symbolic behavior.

To test the hypothesis that combining both structural and internal state changes will accelerate the apparent communication of symbolic function between two systems, I have performed some initial explorations. A neural network was constructed to learn the symbolic version of the logistic function $x_{t+1} = \eta x_t(1 - x_t)$. By performing back propagation with the network's current state, next state, and desired state, the network could not learn the mapping. However, when combined with the tracking of the state of the logistic function by the state of the network, the network was able to acquire the desired functionality, a symbol sequence identical to the symbol sequence generated by the logistic function.

This small demonstration makes several points. First, the network did not need to recreate an approximation, such as lest squares) to the logistic function. Due to the universality of the symbol dynamics of unimodal functions, of which the logistic function is probably the most studied, the network merely needed to construct a single hump transfer function. Second, this experiment demonstrates the inherent difficulties with learning in a vacuum, that is with a teacher insensitive to the abilities of the learner. Connectionist and symbolic machine learning systems suffer similarly due to batch or random sample learning. Finally, the computational complexity of symbol system problems such as learning, parsing, and searching may be reduced by focusing on the physical level. Additional work will include theoretical substantiation of the above claims and a larger examples of this phenomena. It is unclear whether this approach will scale to systems with more degrees of freedom and complex behavior.

One point this demonstration makes is the importance of the communication medium. Grammar learning from discrete symbol sequences is provably intractable for anything more difficult than regular languages. In this case, the fulcrum for reducing the complexity of induction problems is to correctly identify what is being transferred. Chomsky (1965) postulated the existence of a universal grammar as the parent of all natural language grammars. Children are born with knowledge of the universal grammar and the language acquisition process finds the parameters which tunes the universal grammar into a grammar for Japanese, English, or Turkish. While grammars are excellent descriptions of sentence syntax, grammars by no means exhaust the cornucopia of mechanisms with the appearance of

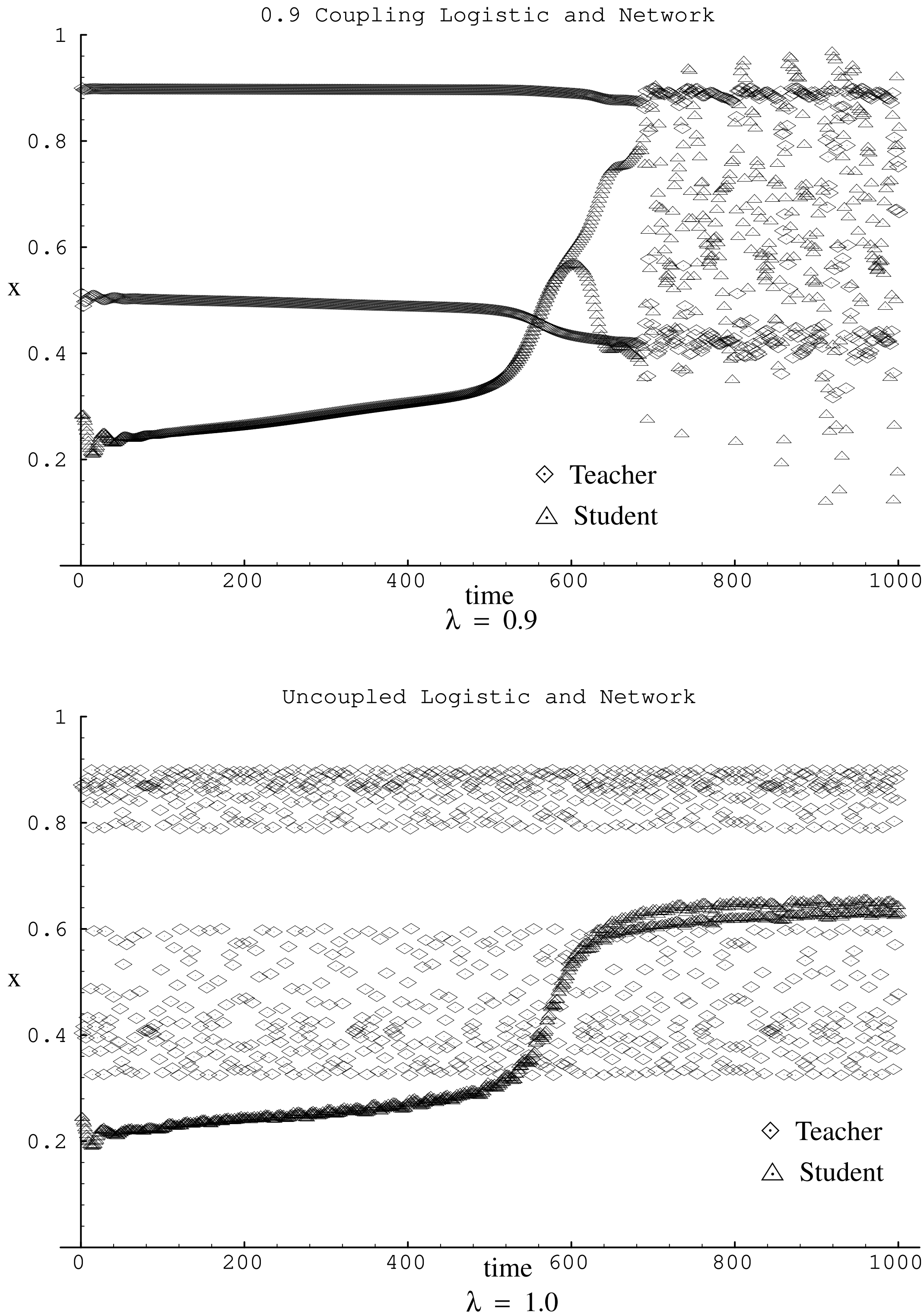


Figure 53: Entrainment data for logistic function teacher and network students.

generative systems. Entrainment learning suggests that grammar learning occurs epiphenomenally and emerges from interacting processes with no connection to the high level language acquisition. In this case, the medium is more important than the message.

7.5 Conclusion

In this chapter, I returned to the topic of neural networks and discussed the implications of the last two chapters to the understanding of recurrent networks. The apparent computational complexity of recurrent networks emerges through a combination of internal dynamics, input modulation, and observation of the underlying physical system. What are the substrates for the physical symbol system or universal grammar? There are none. Computation arises as a holistic combination of the organism and the observer since the observer symbolizes behavior, regardless of the actual nature of the internal processes producing that behavior. The necessary and sufficient conditions of universal computation in Newell and Simon's physical symbol system hypothesis provides little insight on the computational behavior of a system. As in the case of the observers' paradox, it merely implies that we can simulate, with any universal computational model, any written record of behavior.

Attempting to understand the computational aspects of recurrent neural network models has proved disappointing, but fruitful. It was shown in Chapter V that the discrete selection of continuous state spaces mappings induced a finite set of nonlinear state space transformations, regardless of the underlying architecture. Explaining recurrent networks behavior as IFS's, however, did not directly answer our original question. It merely explains the regularities found in state space attractors. In Chapter VI, I considered the effects of observation on the attribution of computational abilities to physical systems. Since many scientists attribute the complexity of a behavior to the mechanism producing the behavior, discovering that discrete observation measurements can wreck havoc on the observed computational complexity was an important clue in understanding the information processing mechanisms of recurrent networks.

The main conclusion to be drawn from this chapter focuses on the emergent nature of computation. Computation, a notion which originated earlier this century during the formalization of theorem proving, is capable of reliably describing systems whose state space is discrete and changes at discrete timesteps. The ambiguities which arise when one tries to ascribe computational intentionality to systems outside this dynamical system class, clearly identify computation as nothing more than a descriptive stance and in no way supports it as an organizational principle.

CHAPTER VIII

RETROSPECTIVE AND CONCLUSION

8.1 Retrospective

The contents of this dissertation mark a trail that leads to a better understanding of recurrent neural network behavior. Like the early American settlers, I had to explore unknown territory in search of bountiful lands. This journey offered many promising alternative paths that I reluctantly ignored as I blazed this trail. At times, the fractal intricacies of the trail drew my attention away from the original promise of enlightenment. Sometimes I could not extract myself from the breathtaking reverberation of recurrent nets. Chaotic storms, initiated by an unnamed Brazilian butterfly, repeatedly distracted me from my task. After many years of exploration, this dissertation maps out a path to the observations, arguments, and conclusions made in the Chapter VII. There I described an approach to understanding computation in recurrent networks that challenged traditional views of information processing in neural networks whose foundation spans a wide range of disciplines introduced in the earlier chapters. It is time to look back and review the events that brought us to this point.

Chapter II outlined both the historical and social context that surrounded the issues addressed in this dissertation. Artificial intelligence, as the study of formal models of cognitive abilities, has benefited very little from the neural network revolution. The reappearance of neural network systems as an alternative to traditional artificial intelligence techniques has produced an abundance of claims. Since the appearance of Hopfield networks, back-propagation, and adaptive resonance theory, little has changed. Despite the promises of new problem solving and modeling approaches, many connectionist researchers have unknowingly succumbed to providing a "merely implementational" contribution to the understanding of cognition (Fodor & Pylyshyn, 1988). While this might have some technical worthiness, it has played into the hands of the field's major detractors. Connectionists have wasted too much time and too many resources by reinventing the symbolic wheels of cognition. This should not be surprising: designing an artifact from a specification is much easier than finding new uses for some object. The connectionist community has reached a crossroad. We must now decide how connectionist models will contribute to cognitive science in its effort to understand cognitive behavior. These models can offer much more than an implementational substrate for knowledge level accounts of intelligence. I clearly believe that they provide an interesting explanatory service on their own that other traditional forms of explanation do not.

Chapter II also discussed some approaches used to understand the processing performed by feed-forward neural networks. These networks transform representations from one vector space to another, maintaining neighborhood relationships along the way. Two tools, principal component analysis and hierarchical clustering, can examine the resulting representations. The former looks for an orthogonal basis for hidden unit representations. Clustering categorizes neighboring hidden unit representations into clumps of neighboring points. Researchers assign meaningful labels to the analytical fallout of these methods to suggest the information processing abstractions the representations entail.

Since the publication of Parallel Distributed Processing (Rumelhart et al., 1986) many have discovered shortcomings of the models and approaches described within this influential collection of papers. Some detractors have attacked the biological implausibility of the models, citing discrepancies between model assumptions and neurological data (Grossberg, 1987). Others have developed pointed arguments against poorly executed applications of the theory and then extrapolated these deficits to the entire class of connectionist models (Pinker & Prince, 1988). The most articulate of the opponents have focused on the weak computational prowess of the models (Fodor & Pylyshyn, 1988).

Of the all the criticisms leveled at connectionism, the most substantial has been their models' lack of computational power. Response to these attacks has been to turn the networks in on themselves: to use connection graphs with cycles. The addition of recurrent connections to networks introduces informational processing state to the network, much like the self-connections that transform combinatorial circuits into sequential circuits. In Chapter III, I catalogued a collection of recurrent neural network models. It detailed many mechanisms whose information processing capabilities are currently misunderstood. Although information processing in feed-forward networks is relatively easy to grasp, the iteration of state transforms introduces many properties that researchers analyzing feed-forward networks do not have to contend. For instance, the presence of internal memory forces the network to generated different responses to the same input vector and separates recurrent networks from their feed-forward cousins. It is precisely this difference that allows us to apply recurrent networks to problems involving temporal context and clouds our understanding of their computational behavior.

The networks of Jordan, Simple Recurrent Networks of Elman, and the Sequential Cascaded Networks of Pollack share a general internal organization with traditional digital sequential circuits. Each network calculates the next state vector from the current state vector and current input vector. The current output can either be calculated from just the next state vector, or from a combination of the current state and input. While it is easy to describe the processing that occurs in feed-forward networks without referring to their dynamical properties, recurrent networks are intimately tied to such descriptions. Chapter IV introduced dynamical system terminology critical to the understanding of these systems. While digital Mealy and Moore machines are computationally equivalent as finite state machines, Chapter V, I showed that the analogy does hold for the analog state machines described above. Finite state machine interpretations, despite their utility, actually inhibit our understanding of the processing occurring in recurrent networks.

Before exploring the processing abilities of these networks, it was necessary to introduce certain terminology from the study of dynamical systems as this terminology appears throughout the dissertation. Chapter IV contained review material from the study of dynamical systems. I provided a taxonomy of dynamical systems that related computation and other dynamical system paradigms. The two dimensions of taxonomy divided the paradigms by their approach to representing state and abstracting time. Each dimension distinguished between discrete and continuous representations. I reviewed the notions of attractor and repellor to characterize stability of the long term behavior of dynamical systems. The long term behavior could also be described either as fixed point, periodic, or aperiodic (chaotic). I then described two methods for identifying the type of attractor: Hausdorff dimensionality and period doubling routes to chaos. The former recognizes recursive structure in the attractor, while the latter characterize limit behavior seen in bifurcation diagrams. In addition, I discussed iterated function theory, a set of theoretical results applicable to understanding the dynamic behavior of sets of transformations. Iterated function systems explore the fractal structures resulting from the iteration of unions over affine transforms, much like the Cantor middle third set. An important aspect of these systems is that they can be input driven. For instance, the chaos game takes a probablistically generated sequence of input symbols, one for each transform in an IFS, and uses the sequence to construct an attractor approximation for the IFS. The final section of the chapter briefly introduced symbolic dynamics, a tool for understanding dynamical systems that collapses continuous state space trajectories onto discrete symbol sequences. This characterization dispenses with details of the trajectory and focuses on the gross behavior of the system. The limit behavior of many systems can be described in terms of symbol sequence generators. In periodic behaviors, these generators continually repeat the same pattern of state symbols. If the generator has certain stochastic components that allow it to construct aperiodic sequences, the underlying dynamical system is chaotic. This conceptual tool became useful in Chapter VI when I discussed the effects of measurement on observed complexity.

Chapters II, III, and IV provided the necessary groundwork for constructing the argument developed in Chapters V, VI, and VII. In those chapters, I set out to explain the origins of complex behavior in recurrent networks.

Chapter V marked the beginning of the quest to understand the processing capabilities of recurrent networks. Several attempts have been made toward understanding the information processing abilities of these networks, each involving some sort of finite state machine. Although widely applicable, these traditional digital circuit analysis techniques have misled all previous attempts to characterize recurrent network information processing. I argued that digital circuit analogies did not provide any insight on the problem and suggested that the theory of iterated function systems provides a better foundation for understanding recurrent networks state dynamics. I then contrasted finite state machine interpretations of recurrent networks with the IFS approach. The main result showed that we can think of recurrent networks as indexed state space transformations that interact to produce emergent properties. The fractal infinite state spaces emerged through the interaction of the transients of the network's state transformations.

After laying this groundwork, I then set out to explain the origins of complex behavior in recurrent networks in Chapters VI and VII. Many researchers in AI and cognitive science believe that the complexity of a behavioral description reflects the underlying information processing complexity of the mechanism producing the behavior. In Chapter VI, I explored the foundations of this complexity assumption. I first distinguished two orthogonal judgments of system complexity, complexion and generative complexity, applicable to these descriptions. Complexion measures the number of observed states and generative complexity refers to the class of behaviors describable under a given formalism, like finite state or push-down automata. Then I argued that neither complexity judgment can be an intrinsic property of the underlying physical system. For instance, the number of FSM states and neither is a generative class like context-free languages are not intrinsic to the underlying physical system. Changes in the method of observation can radically alter both the number of apparent states and the apparent generative class of a system's behavioral description. I concluded from these examples that the act of observation could suggest frivolous computational explanations of physical phenomena, ranging from recurrent networks up to, and including, cognition.

In Chapter VII, I identified three contributors to the emergence of computational behavior in recurrent networks: internal dynamics, input modulation, and, of course, observation. For years, the internal dynamics have been argued as the true source of computationally complex behavior (Chomsky, 1965) or indistinguishable from a complex environment (e.g., Simon's ant (Simon, 1969)). Input modulation referred to the direct manipulation by the environment of the systems' internal computational dynamics. This notion first appeared in Chapter V as the indexed selection of transformations in recurrent networks. I extended the concept to cover the creation of virtual transformations through the regular repetition of periodic inputs. Thus, the effect of the input sequence **ABABABAB**... on a recurrent network with state transforms $f_{\mathbf{A}}$ and $f_{\mathbf{B}}$ is the same as the effects of iterating the composed function $f_{\mathbf{AB}}(x) = f_{\mathbf{B}}(f_{\mathbf{A}}(x))$. Besides internal dynamics and input modulation, I argued that the observation of behavior has an effect equal to the combined effects of the other contributors.

The final sections of Chapter VII suggested that emergent computation is not categorically separate from traditional notions of computation. Rather, all computation is emergent in that it is a subjective property of a system. It is impossible to objectively decide when one system is computing while another is not. Since discrete computation lies in the eye of the beholder, we can now feel free to apply other dynamical system paradigms to traditional symbol processing tasks. As a demonstration, I proposed an alternative model of language acquisition. The standard approach defines language acquisition as the successive refinement of grammatical models. We should view these grammatical models as emergent computational entities, like the finite state machines within the iterated logistic function. Language induction is then implemented as coupling between two dynamical systems with trainable parameters. For simplicity, I assumed a fixed (dynamic) teacher whose state changed according to the student system. After training, the student dynamical system approximates the behavior of the teacher. This approximation was enough to elicit the same symbolic dynamics. In other words, there was apparent transmission of grammatical competence from the teacher to the student.

8.2 Conclusion

Over several years of studying the problem of machine learning, I have patiently watched as artificial neural networks adjusted their internal parameters to "learn" approximations to simple symbolic functions. Given the intractability of neural network learning (Kolen, 1988) and other nontrivial machine learning problems, I must now ask the following question. How can the human brain deal with this intractability? While some researchers (e.g., (Penrose, 1991)) propose that computation on a quantum mechanics level bypass this issue, I cannot blindly follow in this direction. I maintain that the source of the intractability is ourselves: we are asking the wrong questions. By incorrectly judging the important features and processes of cognition, building computational models of those processes can lead to incorrect competence judgments. Computational models are often selected because they exhibit the desired *competence* rather than performance: planning as means-ends analysis, language understanding as context-free parsing, and reasoning as logical inference. In each of these cases, the AI community has defined the desired competence before identifying the underlying mechanism and evaluating *its* own competence.

Ironically, cognitive science traditionally describes performance limitations in terms of the competence model. Consider the sentence "The dog that the cat that the fly ... caught chased." We view failure to comprehend sentences like this one to arbitrary depth as a performance issue. The standard argument contends that we lack the computational resources to handle arbitrary depth, but given enough memory, time, patience, we could understand such sentences. This argument draws its resource assumption from the competence model, our picture of how things should be. What if the competence model is wrong? A very large finite state machine is usually easily described by a push-down automaton or Turing machine. Yet if we abstract a level of competence from the finite state machine that is describable in terms of a Turing machine, the types of available resources differ between the competence model and the actual implementation. Making claims about the implementation appears suspect when based upon the *virtual* resources of the competence model .

A concrete example of this way of thinking occurs in the field of connectionism. While trying to understand the processing abilities of neural nets, researchers often fall back to the perspectives of existing models. Such ideas permeate the AI and cognitive science: feedforward nets operate like boolean functions, nonlinear regressions, and principal component extractors. Following the establishment of these attitudes, people started to select certain modes of behavior from the networks in such a manner to impose their *a priori* images upon the network. While improving their conceptions, they often abstract away from the behavioral or computational repertoires, or just conveniently ignore other elements as "noise".

It has become clear that feedforward networks fail to provide the full repertoire of computational abilities essential to the construction of cognitive models. In their zeal to illustrate the representational breadth of feed-forward networks, researchers ignored the basic complaint. Neural networks fail to support necessary and sufficient features of computation, such as compositionality and systematicity.

Recurrent networks, however, employ a constantly updated internal state and appear adequately prepared to support computation. Many proofs already demonstrate the universal computational capabilities of recurrent networks. As such, claims against the computational adequacy of neural networks as models of cognitive phenomena now seem ridiculous. Despite these results, the neural network community appears perfectly comfortable interpreting recurrent networks as merely deterministic finite state machines (DFSMs). We use Occam's razor to convince ourselves that finite state automata are simpler than Turing machines and expect a similar ordering to hold in recurrent networks.

Although the breadth of DFSM entrenchment through a wide range of disciplines suggests that this interpretation has merit, it obscures the computational capabilities unique to recurrent networks. I have clearly shown that they possess their own unique natural dynamics. Before passing judgment on the utility of these models, we need to first find problems that match these dynamics. Unfortunately, there is little social impetus to follow this difficult path. In Chapter II, I pointed out that many instances of connectionist research merely implement traditional symbolic AI techniques. If connectionism expects to contribute to AI and cognitive science, this practice must stop.

The standard automata associated with these computational models and formal languages can trace their lineage back to a metaphor of writing. Automata theory defines the power and the limitations of an agent equipped with paper and pencil when guided by a finite set of rules, as exemplified by rigorous theorem proving. Thus, automata theory applies to the power of written descriptions interpretable with a finite set of rules. Most of the work in the field of artificial intelligence is founded on the idea of "animated descriptions". Once a cognitive faculty, like language, is described by rules, the power of programmed computing devices can elevate these passive descriptions in to active agents. For instance, an AI programmer might fashion a language user from a grammar of English or a problem solver from a generalization of verbal protocols.

Faced with the prospect of incorrect competence and the creation of animate descriptions, this dissertation reports upon my examination of the nature of computation and how it relates to foundational problems of artificial intelligence and cognitive science. The major result of my thesis has been the classification of computation as another observer-created scientific descriptive notion like "center of gravity". Centers of gravity exist solely to aid our understanding (i. e., there is no individual point mass that serves as the center of gravity, it is a convenient shorthand for describing the cumulative gravitational properties of a physical body). As a method of symbolic description, computation provides a similar service. Computation provides a means to concisely write down a smaller version of a longer, potentially infinite, description. For instance, linguistics uses context-sensitive grammars for compactly describing the set of grammatical sentences (Chomsky, 1957).

Many in the field of cognitive science equate computation with the processing of representations. This view is understandable given the appearance and propagation of Turing machines, production systems, and other symbol manipulation systems during the infancy of cognitive science. In this thesis, I have shown that computation is possible without manipulation of representation. Machines composed of a simple physical system with an observation mechanism, such as an oscillator paired with a threshold device, can display temporal behavior strictly afforded to infinite state computation engines. Computation, in other words, emerges from the observation process. Currently, several mechanisms have been shown both formally and empirically to generate such complexity. With these results in hand, I disagree with Fodor's view that representation is a prerequisite for computation (Fodor, 1982), and maintain that representation is merely a simplification for the benefit of our understanding. As Hans Reichenbach said, "Logic is the touchstone of thinking, not its propelling force; it formulates the laws by which we judge thought products correct, not laws we want to impose on thinking."[1] As a derivative of logic, computation is imposed upon behavior, not its propelling force.

If a computational system does not directly manipulate representations, as alluded to above, how can we program it? One major accomplishment of the twentieth century was the discovery of the power of simulation, the mechanical manipulation of descriptions of

1. (Reichenbach, 1947) page 1.

mechanisms. Through a long chain of events, von Neumann designed a computer architecture specifically designed to manipulate descriptions, or representations, of computing devices (von Neumann, 1946). We now call these descriptions computer programs. Early special purpose computers were programmed by physically rewiring the device. Now, thanks to von Neumann, a software level replaces the arduous task of rewiring; the computer provides an interface for changing the internal state of the device. These two approaches, hardware and software manipulation, have their respective strengths and weaknesses. I believe that merging the two approaches will result in a combination that is more akin to naturally occurring "programming", i.e. learning.

One aspect of programming is to understand what has been programmed. The question I now ask is: how can we *identify* which computation a given recurrent network is performing? Initial attempts to answer this question resulted in many researchers looking for, and finding, finite state machines in the state dynamics of recurrent networks. Limiting the behavioral competencies of recurrent networks to that of finite state machines is ludicrous, given that we know how to *design* such networks to carry out universal computation. I have shown in the preceding chapters that the finer tools of dynamical systems analysis reveal that networks display very rich and surprising behaviors. Dynamical systems theory suggested a closer relationship between recurrent networks and iterated function systems than DFSMs.

From the dynamical system vantage, it is unclear what is the right level of abstraction to apply to a continuous state system like recurrent networks. Similar questions can be asked about cognitive faculties, such as language production. Therefore, I have become interested in how shifts from one classification of states to another affect apparent complexity. Observing a trajectory of a recurrent network through a continuous state space and interpreting it as a trajectory of discrete states, for instance, will affect the resulting computational *explanation* of that trajectory. I have described several examples that illustrate how observation can affect both the number of observed states and the complexity class of the mechanism embodying those states.

Finally, I argued for a new design and implementation methodology that acknowledges the three sources of complex computational behavior. The original anticonnectionists based their arguments on a particular form of computation in which both compositionality and systematicity are crucial properties. Other forms of computation exist, however, each with its own set of necessary and sufficient properties. This begs the question: Are there any properties common to *all* forms of computation? I proposed that computation arises from three factors: internal dynamics, input modulation of those dynamics, and interpretation (i.e. observation) of those dynamics. In other words, a computational system must change over time, be affected by another "system", and be watched while this is occurring. The latter condition, observability, implies that all computation is an emergent phenomenon, a phenomenon with a permanent resting place in the eye of the beholder. Computation requires no causal underpinnings for its emergence, despite our emotional needs for such grounding.

While this conclusion may sound too negative, it does open new routes for inquiry. For instance, I suggested that the process of grammatical inference known as natural language learning could be explained in terms of coupled dynamical systems with continuous state spaces. While grammatical competencies can describe the observed behavior, nothing explicitly implements or manipulates these competencies. The observed competencies, and other computational phenomena, emerge as observed properties of the system.

This question of what kind of computation is going on in dynamical systems such as recurrent networks, or the brain, for that matter, is ill-formed. When a tree falls in the forest with no one around, does it make a sound? Does it make a noise? Of course the falling tree disturbs the air surrounding the tree, creating pressure waves we call sound. But is it a noise? This subjective judgment depends entirely upon the listener. Computation falls into a similar predicament. The internal dynamics propel the system through state space, but external observation discretizes this movement. The result is a subjective experience sculpted by the measurement process. Complex computational capabilities, those abilities demanded by linguistic and psychological accounts of cognitive behavior, are the result of the combination of a simple continuous state system with a discrete time symbolic observation. Entrainment learning suggests that the loss of direct manipulation of representation afforded by computational explanations is offset by the elimination of theoretical time complexity difficulties associated with representation-based descriptions of intelligence.

LIST OF REFERENCES

Aida, S., Allen, P. M. , Atlan , H. K., Boulding, E. , Chapman, G. P., Costa de Beuregard, O., Danzin, A. , Dupuy, J.-P., Giarini, O., Hägerstrand, T., Holling, C. S., Kirby, M. J. L. , Klir, G. J., Laborit, H., Le Moigne, J.L., Luhmann, N., Malaska, P., Margalef, R., Morin, E., Ploman, E. E., Pribram, K. H., Prigogine, I., Soedjatmoko, Voge, J., & Zeleny, M. (1984). *The Science and Praxis of Complexity.* Tokyo, Japan:United Nations University.

Amit, D. (1989). *Modeling Brain Function: The World of Attractor Neural Networks.* Cambridge, England:Cambridge University Press.

Anderson, J. A. (1972). A simple neural network generating an interactive memory. *Mathematical Biosciences, 14*, 197-220.

Anderson, J. A., Silverstein, J. W., Ritz, S. A. & Jones, R. S. (1977). Distinctive features, categorical perception, and probability learning: some applications of a neural model. *Psychological Review, 84*, 413-451.

Angluin, D. (1987). Learning regular sets from queries and counterexamples. *Information and Computation, 75*, 87-106.

Ashby, W. R. (1956). *An Introduction to Cybernetics.* London:Chapman and Hall.

Bai-lin, H. (1989). *Elementary Symbolic Dynamics and Chaos in Dissapative Systems.* New Jersey:World Scientific.

Bak, P., Chen, K. & Creutz, M. (1990). A forest-fire model and some thoughts on turbulence. *Physics Letters, 147*, 297-300.

Bak, P. & Chen, K. (1991). Self-organized criticality. *Scientific American, 264*, 46-53.

Balcazar, J. L. & Gabarro, J. D. J. (1988). *Structural Complexity.* Berlin:Springer-Verlag.

Ballard, D. H. (1987). Parallel logical inference and energy minimization. Report TR142, Computer Science Department, University of Rochester.

Barnsley, M. (1988). *Fractals Everywhere.* San Diego, CA:Academic Press.

Barton, J. G. E., Berwick, R. C. & Ristad, E. S. (1987). *Computational Complexity and Natural Language.* Cambridge, MA:MIT Press.

Berlekamp, E. R., Conway, J. H. & Guy, R. K. (1982). *Winning Ways for Your Mathematical Plays.* New York:Academic Press.

Blumer, A., Ehrenfeucht, A., Haussler, D. & Warmuth, M. K. (1987). Occam's razor. *Information Processing Letters, 24*, 377-380.

Boyle, C. F. (1991). On the physical limitations of pattern matching. *Journal of Experimental and Theoretical Artificial Intelligence*, *3*, 191-217.

Cage, J. (1969). *A Year From Monday: New Lectures and Writings by John Cage.* Middletown, Connecticut:Wesleyan University Press.

Carpenter, G. A. & Grossberg, S. (1987). ART2: self-organization of stable category recognition codes for analog input patterns. *Applied Optics, 26*, 4919-4930.

Casti, J. (1988). *Alternate Realities: Mathematical Models of Nature and Man.* Wiley.

Chalmers, D. (1994) Does a Rock Implement Every Finite-State Automata? Manuscript.

Chandrasekaran, B., Goel, A. K. & Allemang, D. (1988). Connectionism and information processing abstractions: the message still counts more than the medium. *Artificial Intelligence Magazine*, Winter, 24-34.

Chandrasekaran, B. & Josephson, S. R. (1993). Architecture of intelligence: the problems and current approaches to solutions. *Current Science*, *64*:6, 366-380.

Chauvin, Y. (1989). A Back-Propagation Algorithm With Optimal Use of Hidden Units. In D. Touretzky, (Ed.), *Advances in Neural Information Processing Systems 1*, 519-526. San Mateo, CA:Morgan Kaufman.

Chomsky, N. (1957). *Syntactic Structures.* The Hague:Mounton & Co..

Chomsky, N. (1965). *Aspects of the Theory of Syntax.* Cambridge, MA:MIT Press.

Cleeremans, A., Servan-Schreiber, D. & McClelland, J. L. (1989). Finite state automata and simple recurrent networks. *Neural Computation, 1*, 372-381.

Crucianu, M. (1994). Looking for structured representations in recurrent networks. In M. Mozer, P. Smolensky, D. Touretzky, J. Elman & A. Weigand, (Eds.), *Proceedings of the 1993 Connectionist Models Summer School*, 170-177. Hillsdale, NJ:Earlbaum.

Crutchfield, J. & Young, K. (1989). Computation at the Onset of Chaos. In W. Zurek, (Ed.), *Entropy, Complexity, and the Physics of Information*, 223-269. Reading, MA:Addison-Wesley.

Cummins, F. & Port, R. F. (1994). On the treatment of time in recurrent neural networks. In M. Mozer, P. Smolensky, D. Touretzky, J. Elman & A. Weigand, (Eds.), *Proceedings of the 1993 Connectionist Models Summer School*, 211-218. Hillsdale, NJ:Earlbaum.

Das, S. & Mozer, M. (1994). Self-clustering in recurrent networks. In J. D. Cowan, G. Tesauro & J. Alspector, (Eds.), *Advances in Neural Information Processing Systems 6*, 19-26. San Mateo, CA:Morgan Kaufman.

Dennett, D. (1971). Intentional systems. *The Journal of Philosophy, 68*, 87-106.

Ditto, W. L., Rauseo, S. N. & Spano, M. L. (1990). Experimental control of chaos. *Physical Review Letters, 65*, 3211-3214.

Dolan, C. & Dyer, M. G. (1987). Symbolic schemata, role binding and the evolution of structure in connectionist memories. In *IEEE First International Conference on Neural Networks*. San Diego, CA.

Elman, J. L. (1990). Finding structure in time. *Cognitive Science, 14*, 179-211.

Fahlman, S. E. & Lebiere, C. (1990). The Cascade-Correlation Learning Architecture. In D.S. Touretzky, (Ed.), *Advances in Neural Information Processing Systems*, Vol. 2, 524-532. San Mateo:Morgan Kaufmann.

Fanty, M. (1985). Context-free Parsing in Connectionist Networks. TR174. University of Rochester, Computer Science Department, Rochester, N.Y..

Farhat, N. H., Psaltis, D., Prata, A. & Paek, E. (1985). Optical implementation of the hopfield model. *Applied Optics, 24*, 1469-1475.

Feigenbaum, M. J. (1983). Universal behavior in nonlinear systems. *Physica, 7D*, 16-39.

Feldman, J. A. & Ballard, D. H. (1982). Connectionist models and their properties. *Cognitive Science, 6*, 205-254.

Fields, C. (1987). Consequences of nonclassical measurement for the algorithmic description of continuous dynamical systems. *Journal of Experimental and Theoretical Artificial Intelligence, 1*, 171-178.

Fodor, J. (1982). *The modularity of mind.* Cambridge:MIT Press.

Fodor, J. A. & Pylyshyn, Z. W. (1988). Connectionism and cognitive architecture: a critical analysis. *Cognition, 28*, 3-71.

Forrest, S. (1991). Emergent computation: self-organizing, collective, and cooperative phenomena in natural and artificial computing networks. *Physica, 42D*, 1-11.

Franklin, S. & Garzon, M. (1988). Neural Computability. In O. Omidvar (Ed.) *Progress in Neural Networks*. Norwood, NJ:Ablex.

Giles, C. L., Sun, G. Z., Chen, H. H., Lee, Y. C. & Chen, D. (1990). Higher order recurrent networks and grammatical inference. In D. Touretzky, (Ed.), *Advances in Neural Information Processing Systems 2*, 380-387. San Mateo, CA:Morgan Kaufman.

Giles, C. L., Miller, C. B., Chen, D., Sun, G. Z., Chen, H. H. & Lee, Y. C. (1992a). Extracting and Learning an Unknown Grammar with Recurrent Neural Networks. In J. E. Moody, S. J. Hanson & R. P. Lippman, (Eds.), *Advances in Neural Information Processing Systems 4*, 317-324. San Mateo, CA:Morgan Kaufman.

Giles, C. L., Das, S. & Sun, G. (1992b). Learning context-free grammars: Capabilities and limitations of a recurrent neural Network with an external stack memory. In *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*, 317-324. Hillsdale, NJ:Earlbaum.

Gold, E. M. (1969). Language identification in the limit. *Information and Control, 10*, 372-381.

Goudreau, M. W., Giles, C. L., Chakradhar, S. T. & Chen, D. (in press). First-order vs. second-order single layer recurrent networks. To appear in *IEEE Transactions on Neural Networks*.

Graf, H. P., Jackel, L. D., Howard, R. E., Straughn, B., Denker, J. S., Hubbard, W., Tennant, D. M. & Schwartz, D. (1986). VLSI Implementation of a Neural Network Memory with Several Hundreds of Neurons. In J. S. Denker, (Ed.), *Neural Networks for Computing*, 182-187. New York:American Institute of Physics.

Grassberger, P. & Procaccia, I. (1983). Characterization of strange attractors. *Physics Review Letters, 50*, 346-349.

Grebogi, C., Ott, E. & Yorke, J. A. (1987). Chaos, strange attractors, and fractal basin boundaries in nonlinear dynamics. *Science, 238*, 632-638.

Grossberg, S. (1987). Competitive learning: from interactive activation to adaptive resonance. *Cognitive Science, 11*, 23-63.

Hanson, S. J. & Burr, D. J. (1989). What connectionist models learn: learning and representation in connectionist networks. *Behavioral and Brain Sciences, 13*, 471-488.

Hebb, D. O. (1949). *The Organization of Behavior: A Neuropsychological Theory.* New York:John Wiley & Sons.

Hertz, J., Krogh, A. & Palmer, R. G. (1991). *Introduction to the Theory of Neural Computation.* Reading, MA:Addison-Wesley.

Hinton, G. E., McClelland, J. L. & Rumelhart, D. E. (1986). Distributed Representations. In D. E. Rumelhart, J. L. McClelland & the PDP Research Group, (Eds.), *Parallel Distributed Processing: Experiments in the Microstructure of Cognition*, Vol. 1: Foundations, 77-109. Cambridge, MA:MIT Press.

Hinton, G. E. & Sejnowski, T. J. (1986). Learning and Relearning in Boltzman Machines. In D. E. Rumelhart, J. L. McClelland & the PDP Research Group, (Eds.), *Parallel Distributed Processing: Experiments in the Microstructure of Cognition*, Vol. 1: Foundations, 282-317. Cambridge, MA:MIT Press.

Hinton, G. E. (1987). Learning translation invariant recognition in a massively parallel network. In G. Goos & J. Hartmanis, (Eds.), *PARLE: Parallel Architectures and Languages Europe*. Berlin:Springer-Verlag.

Hopcroft, J. E. & Ullman, J. D. (1979). *Introduction to Automata Theory, Languages, and Computation.* Reading, MA:Addison-Wesley.

Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences USA, 79*, 2554-2558.

Hopfield, J. J. (1984). Neurons with graded response have collective computational properties like those of two-state neurons. *Proceedings of the National Academy of Sciences USA, 81*, 3088-3092.

Hopfield, J. J. & Tank, D. W. (1985). Neural computation of decisions in optimization problems. *Biological Cybernetics, 52*, 141-152.

Hornik, K., Stinchcombe, M. & White, H. (1989). Multi-layer feedforward networks are universal approximators. *Neural Networks, 2*, 359-366.

James, W. (1890). *The Principles of Psychology.* New York:H. Holt and Company.

Jordan, M. (1986). Serial order: A parallel distributed processing approach. UCSD Technical Report, La Jolla, CA.

Joshi, A. K., Vijay-shanker, K. & Weir, D. J. (1989). Convergence of mildly context-sensitive grammar formalisms. In T. Wasow & P. Sells , (Eds.), *The Processing of Linguistic Structure*. Cambridge, MA:MIT Press.

Knapp, A. G. & Anderson, J. A. (1984). Theory of categorization based on distributed memory storage. *Journal of Experimental Psychology: Learning, Memory, and Cognition, 10*, 616-637.

Kohonen, T. (1972). Correlation matrix memories . *Institute of Electrical and Electronics Engineers Transactions on Computers, C-21*, 353-359.

Kolen, J. F. (1988). Faster learning through a probabilistic approximation algorithm. In *The Proceedings of the Second Institute of Electrical and Electronics Engineers International Conference on Neural Networks,* I:449-454. San Diego, California.

Kolen, J. F. & Pollack, J. B. (1991). Multiassociative Memory. In *Proceedings of the Thirteenth Annual Conference of the Cognitive Science Society*, 785-789. Hillsdale, NJ:Earlbaum.

Kolen, J. F. & Goel, A. K. (1992). Learning in parallel distributed processing networks: computational complexity and information content. *Institute of Electrical and Electronics Engineers Transactions on Systems, Man, and Cybernetics, 21*, 359-367.

Kolen, J. F. & Pollack, J. B. (1993). The apparent computational complexity of physical systems. In *Proceedings of the Fifteenth Annual Conference of the Cognitive Science Society*, 617-622. Hillsdale, NJ:Earlbaum.

Kolen, J. F. (1994a). Fool's gold: Extracting finite state machines from recurrent Networks Dynamics. In J. D. Cowan, G. Tesauro & J. Alspector, (Eds.), *Advances in Neural Information Processing Systems 6*, 501-508. San Mateo, CA:Morgan Kaufman.

Kolen, J. F. (1994b). The origin of clusters in recurrent network state space. In *Proceedings of the Sixteenth Annual Conference of the Cognitive Science Society.* Hillsdale, NJ:Earlbaum.

Kolen, J. F. (1994c). Recurrent networks: State machines or iterated function systems?. In M. Mozer, P. Smolensky, D. Touretzky, J. Elman & A. Weigand, (Eds.), *Proceedings of the 1993 Connectionist Models Summer School*, 203-210. Hillsdale, NJ:Earlbaum.

Krogh, A. & Hertz, J. A. (1992). A simple weight decay can improve generalization.. In J. E. Moody, S. J. Hanson & R. P. Lippman, (Eds.), *Advances in Neural Information Processing Systems 4*, 950-957. San Mateo, CA:Morgan Kaufman.

Langton, C. (1990). Computation at the edge of chaos: phase transitions and emergent computation. *Physica, 42D*, 12-37.

Large, E. (1991). A neural network model for recoding of musical stimuli. In *The Proceedings of the American Accoustical Society.*

Lee, Y. C., Doolen, G., Chen, H. H., Sun, G. Z., Maxwell, T., Lee, H. Y. & Giles, C. L. (1986). Machine learning using a higher order correleational network. *Physica, 22D*, 276.

Li, T. Y. & Yorke, J. A. (1975). Period three implies chaos. *American Mathematics Monthly, 82*, 985-992.

Linsker, R. (1986). From basic network principles to neural architecture. *PNAS, 83*, 7508-7512, 8390-8394, 8779-8783.

Linsker, R. (1988). Self-organization in a perceptual network. *Computer*, 105-117.

Lorenz, E. N. (1963). Deterministic nonperiodic flow. *Journal of Atmospheric Sciences, 20*, 130-141.

MacKay, D. J. C. (1992). A practical bayesian framework for backprop networks. *Neural Computation, 4*, 448-472.

MacLennan, B. (1992). Field computation in the brain. Technical Report.

Mandelbrot, B. B. (1982). *The Fractal Geometry of Nature.* San Francisco:Freeman.

Mani, D. R. & Shastri, L. (1991). *Proceedings of the Thirteenth Annual Conference of the Cognitive Science Society.* Hillsdale, NJ:Earlbaum.

Marr, D. (1982). *Vision.* San Francisco:Freeman.

May, R. M. (1976). Simple mathematical models with very complicated dynamics. *Nature, 261*, 459-467.

McClelland, J. L., Rumelhart, D. E. & Hinton, G. E. (1986). The Appeal of Parallel Distributed Processing. In D. E. Rumelhart, J. L. McClelland & the PDP Research Group, (Eds.), *Parallel Distributed Processing: Experiments in the Microstructure of Cognition*, Vol. 1: Foundations, 3-44. Cambridge, MA:MIT Press.

McCloskey, M. & Cohen, N. J. (1989). Catastrophic interference in connectionist networks: the sequential learning problem. In *The Psychology of Learning and Motivation*, Vol. 24, 109-165. .

McCulloch, W. P. & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics, 5*, 115-133.

Meeden, L., McGraw, G. & Blank, D. (1993). Emergent Control and Planning in an Autonomous Vehicle. In *Proceedings of the Fifteenth Annual Conference of the Cognitive Science Society*, 735-740. Hillsdale, NJ:Earlbaum.

Ryzard S. Michalski, Jamie G. Carbonell & Tom M. Mitchell, (Eds.) (1983). *Machine Learning: An Artificial Intelligence Approach.* Palo Alto, CA:Tioga.

Minsky, M. L. (1975). A Framework for Representing Knowledge. In P. H. Winston, (Ed.), *The Psychology of Computer Vision*, 211-280. New York:McGraw-Hill.

Minsky, M. L. & Papert, S. A. (1969). *Perceptrons.* Cambridge, MA:MIT Press.

Minsky, M. L. & Papert, S. A. (1988). *Perceptrons.* Expanded Edition. Cambridge, MA:MIT Press.

Mitchell, M., Crutchfield, J. P. & Hraber, P. T. (1993). Dynamics, Computation, and the Edge of Chaos: A Re-Examination. In G. Cowan, D. Pines & D. Melzner, (Eds.), *Integrative Themes*, Vol. Santa Fe Institute Studies in the Sciences of Complexity, Proceedings Vol. 19. Reading, MA:Addison-Wesley.

Moody, J. (1991). Note on generalization, weight decay, and architecture selection for nonlinear learning systems. In B. H. Juang, S. Y. Kung, and C. A. Kamm, (Eds.) *Neural Networks for Signal Processing*. Piscataway, NJ:IEEE Press.

Newel, A. (1990). *Unified Theories of Cognition*. Cambridge, MA: Harvard Press.

Newell, A. & Simon, H. A. (1962). GPS-A program that simulates human thought. In E. A. Feigenbaum & J. Feldman, (Eds.), *Computers and Thought*. New York, NY:McGraw-Hill.

Newell, A. & Simon, H. A. (1976). Computer science as empirical inquiry: symbols and search. *Communications of the Association for Computing Machinery, 19*, 113-126.

Oja, E. (1982). A simplified neuron model as a principal component analyzer. *Journal of Mathematical Biology, 15*, 267-273.

Penrose, A. (1991). *The Emperor's New Mind: Concerning computers, minds, and the laws of physics.* Oxford University Press.

Pinker, S. & Prince, A. (1988). On language and connectionism: analysis of a parallel distributed processing model of language inquisition.. *Cognition, 28*, 73-193.

Pirsig, R. J. (1988). *Lila.* New York:Bantam Books.

Pollack, J. B. (1987a). Cascaded back propagation on dynamic connectionist networks. In *The Proceedings of the Fourth Annual Cognitive Science Conference*, 391-404. Seattle, WA.

Pollack, J. B. (1987b). *On Connectionist Models of Natural Language Processing.* Ph. D. thesis. University of Illinois at Urbana-Champaign.

Pollack, J. B. (1990). Recursive autoassociative memories. *Artificial Intelligence, 46*, 77-105.

Pollack, J. B. (1991). The induction of dynamical recognizers. *Machine Learning, 7*, 227-252.

Pollack, J. B. (1993). Book review: Allen Newell, *Unified Theories of Cognition*. *Artificial Intelligence, 59*, 355-369.

Putnam, H. (1988). *Representation and Reality* . Cambridge, MA:MIT Press.

Reber, A. S. (1967). Implicit learning of artificial grammars. *Journal of Verbal Learning and Verbal Behavior, 5*, 855-863.

Reichenbach, H. (1947). *Elements of Symbolic Logic.* Macmillan.

Rice, A. (1985). *The Vampire Lestat.* New York:Knopf.

Rochester, N., Holland, J. H., Haibt, L. H. & Duda, W. L. (1956). Tests on a cell assembly theory of the action of the brain, using a large digital computer. *IRE Transactions on Information Theory, IT-2*, 80-93.

Rosenblatt, F. (1962). *Principles of Neurodynamics.* New York:Spartan.

Rumelhart, D. E., Hinton, G. E. & McClelland, J. L. (1986). A General Framework for Parallel Distributed Processing. In D. E. Rumelhart, J. L. McClelland & the PDP Research Group, (Eds.), *Parallel Distributed Processing: Experiments in the Microstructure of Cognition*, Vol. 1: Foundations, 45-76. Cambridge, MA:MIT Press.

Rumelhart, D. E., Hinton, G. E. & Williams, R. J. (1986a). Learning representations by back-propagating errors. *Nature, 323*, 533-536.

Rumelhart, D. E., Hinton, G. E. & Williams, R. J. (1986b). Learning Internal Representations Through Error Propagation. In D. E. Rumelhart, J. L. McClelland & the PDP Research Group, (Eds.), *Parallel Distributed Processing: Experiments in the Microstructure of Cognition*, Vol. 1: Foundations, 110-146. Cambridge, MA:MIT Press.

Rumelhart, D. E. & McClelland, J. L. (1986). On Learning the Past Tenses of English Verbs. In J. L. McClelland, D. E. Rumelhart & the PDP Research Group, (Eds.), *Parallel Distributed Processing: Experiments in the Microstructure of Cognition*, Vol. 2: Psychological and Biological Models, 216-271. Cambridge, MA:MIT Press.

Rumelhart, D. E., McClelland, J. L. & the PDP Research Group, (Eds.) (1986). *Parallel Distributed Processing: Experiments in the Microstructure of Cognition.* Cambridge, MA:MIT Press.

Rumelhart, D. E., Smolensky, P., McClelland, J. L. & Hinton, G. E. (1986). Schemata and Sequential Thought Processes in PDP Models. In J. L. McClelland, D. E. Rumelhart & the PDP Research Group, (Eds.), *Parallel Distributed Processing: Experiments in the Microstructure of Cognition*, Vol. 2: Psychological and Biological Models, 7-57. Cambridge, MA:MIT Press.

Rumelhart, D. E. & Zipser, D. (1986). Feature Discovery by Competitive Learning. In D. E. Rumelhart, J. L. McClelland & the PDP Research Group, (Eds.), *Parallel Distributed Processing: Experiments in the Microstructure of Cognition*, Vol. 1: Foundations, 151-193. Cambridge, MA:MIT Press.

Schaffer, J. D., Whitley, D. & Eshelman, L. J. (1992). Combinations of genetic algorithms and neural networks: a survey of the state of the art. In *Proceedings of COGANN-92 International Workshop on Combinations of Genetic Algorithms and Neural Networks*.

Schank, R. C. & Abelson, R. P. (1973). *Scripts, Plans, Goals, and Understanding*. Hillsdale, NJ:Lawrence Erlbaum.

Schroeder, M. (1991). *Fractals, Chaos, Power Laws* . New York:W. H. Freeman and Company.

Searle, J. (1990). Is the brain a digital computer? . In *Proceedings and Addresses of the American Philosophical Association*, Vol. 64, 21-37. .

Searle, J. (1992). *The Rediscovery of the Mind.* Cambridge, MA:MIT Press.

Sejnowski, T. J. & Rosenberg, C. R. (1987). Parallel networks that learn to pronounce english text. *Complex Systems, 1*, 145-168.

Sejnowski, T. J. & Rosenberg, C. R. (1989). Nettalk: a parallel network that learns to read aloud. *Complex Systems, 1*, 145-168.

Selman, B. (1985). Rule-Based Processing in a Connectionist System for Natural Language Understanding. CSRI-168:University of Toronto, Computer Systems Research Institute, Toronto, Canada.

Servan-Schreiber, D., Cleeremans, A. & McClelland, J. L. (1988). Encoding sequential structure in simple recurrent networks. Technical Report.

Sharkovski, A. N. (1964). Coexistence of cycles of a continous map of a line into itself. *Ukrainian Mathematical Zetischrift, 16*, 61-71.

Shastri, L. (1988). A connectionist approach to knowledge representation and limited inference. *Cognitive Science, 12*, 331-392.

Siegelmann, H. T. & Sontag, E. D. (1991). On the computational power of neural networks. Report.

Simon, H. A. (1969). *The Sciences of the Artificial.* Cambridge, MA:MIT Press.

Singer, J., Wang, Y. & Bau, H. H. (1991). Controling a chaotic system. *Physical Review Letters, 66*, 1123-1125.

Smolensky, P. (1986). Information Processing in Dynamical Systems: Foundations of Harmony Theory. In D. E. Rumelhart, J. L. McClelland & the PDP Research Group, (Eds.), *Parallel Distributed Processing: Experiments in the Microstructure of Cognition*, Vol. 1: Foundations, 194-281. Cambridge, MA:MIT Press.

Smolensky, P. (1987). A method for connectionist varible binding. Technical Report UU-CS-356-87: Computer Science Dept, Univ. of Colorado, Boulder, Colorado.

Smolensky, P. (1988) On the proper treatment of connectionism. *Behavioral and Brain Science*, *11*, 1-74.

Soffer, B. H., Dunning, G. J., Owechko, Y. & Marom, E. (1986). Associative holographic memory with feedback using phase-conjugate mirrors. *Optics Letters, 11*, 118-120.

Sompolinsky, H., Crisanti, A. & Sommers, H. J. (1988). Chaos in random neural networks. *Physical Review Letters, 61*, 259-262.

Stark, J. (1991). Iterated function systems as neural networks. *Neural Networks, 4*, 679-690.

Stucki, D. J. & Pollack, J. B. (1992). Fractal (reconstructive analogue) memory. In *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*, 118-123. Hillsdale, NJ:Earlbaum.

Sun, R. (1989). A discrete neural Network model for conceptual representation and reasoning. In *Proceedings of the Eleventh Annual Conference of the Cognitive Science Society*, 916-923. Hillsdale, NJ:Earlbaum.

Takens, F. (1981). *Lecture Notes in Mathematics 898.* Berlin:Springer-Verlag.

Tesauro, G. J. & Sejnowski, T. J. (1989). A parallel network that learns to play backgammon. *Artificial Intelligence, 39*, 357-390.

Texas Instruments, (1985). *The TTL Data Book: Standard TTL, Schottky, Low-power Schottky ciruits*.

Touretzky, D. S. (1986). BoltzCONS: reconciling connectionism with the recursive nature of stacks and trees. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society,* 522-530.

Towell, G. G. & Shavlik, J. W. (1993). The extraction of refined rules from knowledge-based neural networks. *Machine Learning, 13*, 71-101.

Turing, A. M. (1936). On computable numbers, with an application to the entscheidungsproblem. In *Proceedings of the London Mathematical Society*, Vol. Serial 2-42, 230-265. .

Van Gelder, T. (1990). Compositionality: a connectionist variation on a classical theme. *Cognitive Science, 14*, 335-384.

Verschure, P. F. M. J. (1991). Chaos-based learning. *Complex Systems, 5*, 359-370.

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. Lippman, (Eds.), *Advances in Neural Information Processing Systems 4*, 309-316. San Mateo, CA:Morgan Kaufman.

Weigend, A. S. (1994). On overfitting and the effective number of hidden units. In M. Mozer, P. Smolensky, D. Touretzky, J. Elman & A. Weigand, (Eds.), *Proceedings of the 1993 Connectionist Models Summer School*, 335-342. Hillsdale, NJ:Earlbaum.

Williams, R. J. & Zipser, D. (1989). A learning algorithm for continually running fully recurrent neural networks. *Neural Computation, 1*, 270-280.

Wolfram, S. (1984). Universality and complexity in cellular automata. *Physica, 10D*, 1-35.

Yuille, A. L., Kammen, D. M. & Cohen, D. S. (1989). Quadrature and the development of orientation selective cortical cells by hebb rules. *Biological Cybernetics, 61*, 183-194.

Zak, M. (1988). Terminal attractors for addressable memory in neural networks. *Physics Letters A, 133*, 18-22.

Zeng, Z., Goodman, R. M. & Smyth, P. (1993). Learning finite state machines with self-clustering recurrent networks. *Neural Computation, 5*, 976-990.