Dynamical Automata

Whitney Tabor

Department of Psychology
Uris Hall
Cornell University
Ithaca, NY 14853

`tabor@cs.cornell.edu`

July 20, 1998

Abstract

The recent work on automata whose variables and parameters are real numbers (e.g., Blum, Shub, and Smale, 1989; Koiran, 1993; Bournez and Cosnard, 1996; Siegelmann, 1996; Moore, 1996) has focused largely on questions about computational complexity and tractability. It is also revealing to examine the metric relations that such systems induce on automata via the natural metrics on their parameter spaces. This brings the theory of computational classification closer to theories of learning and statistical modeling which depend on measuring distances between models. With this in mind, I develop a generalized method of identifying pushdown automata in one class of real-valued automata. I show how the real-valued automata can be implemented in neural networks. I then explore the metric organization of these automata in a basic example, showing how it fleshes out the skeletal structure of the Chomsky Hierarchy and indicates new approaches to problems in language learning and language typology.

0. Introduction

Smolensky (1990) argues that connectionist (or "neural") networks offer an opportunity to overcome the brittleness of symbolic devices without foregoing their powerful computational capabilities. "Brittleness" refers to the fact that many symbolic devices are catastrophically sensitive to small distortions in their encoding—a bit or a semicolon out of place can bring an entire system to its knees. Such sensitivity is reminisicent of the trademark behavior of "chaotic" dynamical processes: small differences in initial conditions give rise to substantial differences in long-term behavior. It would be ironic, then, if the interpretation of neural devices as dynamical systems with potentially chaotic behaviors led to a realization of Smolensky's ideal. Intriguingly, this is the character of the result that I report on here. Fractal objects, which turn up as the traces of chaotic processes, turn out to be especially useful for instantiating powerful computing devices in systems of neurons which exhibit graceful modification under small distortions (cf. Pollack, 1991). It is as though by embracing the caprice of a chaotic process, a computational system can stay in its good graces and make effective use of its complexity (cf. Crutchfield and Young, 1990; Crutchfield, 1994).

0.1 The Chomsky Hierarchy

I work here with the now-standard notion of what it means for a system to be computationally powerful or complex. The Chomsky Hierarchy is an ordering of formal languages into increasingly more-inclusive classes. A formal language, $\mathcal{L}$, is taken to be a set of strings of symbols. A computer which can be made to output any string of symbols in $\mathcal{L}$ but no string of symbols in the complement of $\mathcal{L}$ is called a *generator* for $\mathcal{L}$. A computer which can determine, for any input string, whether or not it is a member of $\mathcal{L}$ is called a *recognizer* for $\mathcal{L}$.

The lowest level on the Chomsky Hierarchy is the set of *finite state languages*, so-called because computers which generate (or recognize) them need only be in a finite number of distinct states. (We can think of two states of a computation as being distinct if they give rise to different expectations about what will happen in the future—see Crutchfield, 1994.) Next up on the hierarchy are *context-free languages*, which can be generated by the branching tree-structures that linguists often employ in descriptions of natural languages. Each context-free language can be recognized by a device called a *pushdown automaton*, which consists of a finite-state controller coupled with a *stack*, or record of important features of the computation that have already taken place (Hopcroft and Ullman, 1979). The stack is a string of symbols, only the first of which is accessible to the controller. The controller is allowed to remove this symbol (a *pop* move), leave the stack unchanged, or add additional symbols to the stack (*push* moves). Since the stack is allowed to be of arbitrary length, a pushdown automaton can be in arbitarily many distinct states. Therefore, there is a big distinction between the finite-state and context-free classes in that the latter but not the former include infinite state languages. Non-finite-state context-free languages are distinguished from finite state languages by the possession of strings which, when generated by branching tree-structures, require *center-embedded descriptions* of arbitary depth. A *center-embedded description* is a branching tree-structure in which a branching node dominates nodes to the left and right of a node dominating another branching tree structure.

Above the context-free languages on the Chomsky Hierarchy are context-sensitive languages. These can be recognized by *linear bounded automata*. A linear bounded automaton has, in place of a stack, a *tape* with a finite number of slots in which the finite controller can write symbols or erase them; the number of slots is a linear function of the length of the input string; the controller can move forward and backward on the tape, one slot at a time. Linguists generally agree that the branching tree-structures associated with context-free languages are very useful for describing large portions of most natural languages, but it is also well-known that a few natural languages (e.g. Dutch) require a more powerful computer than a context-free grammar—in most such cases, linear bounded automata are sufficient (e.g., Shieber, 1985). The most-inclusive class in the Chomsky hierarchy is that of the recursively enumerable languages. These can be recognized by *Turing machines*, which differ from linear bounded automata in that the tape has a countable infinity of slots.

There are also many languages that cannot even be recognized by Turing Machines. Such languages are called *unrestricted languages* (e.g., Siegelmann, 1996).

0.2 Dynamical Automata

In this paper, I discuss a class of devices called "Dynamical Automata" which, in their most general form, can be configured as recognizers (or generators) of unrestricted languages. I focus here, however, on a type of parameterization under which they emulate context free grammars.

The term "Dynamical Automaton" is intentionally like the term "Dynamical Recognizer" which has been used in closely related contexts. The dynamical automata I describe here are similar to but not quite the same as the "dynamical recognizers" that Pollack (1991), Blair and Pollack (to appear), and Moore (to appear) examine. All of these dynamical computing devices have in common that they perform their computations in a real-valued space and involve iterative computations (a function is repeatedly applied to its own output). Pollack takes the tack of training his machines on reasonable-looking tasks and analyzing their behavior in order to get clues to the kinds of computation they are performing. Blair and Pollack combine this approach with the kind of bottom up computational analysis developed by Crutchfield and Young (1990), in which one tries to fit various finite-state devices to an unknown machine in order to detect infinite-state computation. Moore examines his dynamical recognizers from the standpoint of their computing power and produces an elaboration of the Chomsky hierarchy involving many new language classes. Moore 1996 demonstrates context free generative capacity for one class of his Dynamical Recognizers by a method that is similar to the method which I describe in Section 2 below. The work I describe here complements both the complexity-detection approach and the complexity-classsification approach. Although I do not study learning mechanisms, I show explicitly how to parameterize infinite-state dynamical automata so that they emulate particular context-free grammars. Moreover, I show how the dynamical automaton framework (and, implicitly, the dynamical recognizer framework) allows us to situate formal languages in metric spaces (spaces in which distances can be measured between points). Such language-spaces look especially useful for addressing the flexibility problem noted above.

0.3 Neural networks

The robustly flexible connectionist devices (neural networks) to which Smolensky 1990 refers are, in fact, a rather varied collection of formal models. In this section, I describe those features of them which I make use of here. They have in common that their invention was inspired by research in neurobiology. They typically involve finite collections of similar computing *units* which receive and send information to each other along channels called *connections*. The connections are associated with scalars called *weights*. The units have *activation values*, a_i which are computed as function, f, of a weighted sum of the activations on units they are connected to, where the weight associated with the activation of unit a_i by unit a_j is the weight, w_{ij} on the connection running from unit j to unit i (Equation (1)).

$$a_i = f(\sum_{j=1}^{N} a_j w_{ij}) \tag{1}$$

The *activation function*, f, is typically a threshold function like

$$f(x) = \left\{ \begin{matrix} 1 & x > \theta \\ 0 & x \leq \theta \end{matrix} \right\}, \quad \theta \in R \tag{2}$$

or one of its continuous analogs, for example, the *sigmoid*,

$$f(x) = \frac{1}{1 + e^{-x}} \tag{3}$$

Sometimes the activation function is linear ($f(x) = kx$ for $k \in R$). Usually, when people talk about a linear activation function they mean the case $k = 1$.

Networks with *second-order* connections are sometimes studied as well. In this case, the activation of one unit, k serves to specify the weight, w_{ijk} on a connection between two other units (j to i). In the work described here, I make use of restricted form of second-order connection: *gating*. If a *gating* unit is sufficiently activated, it allows (or alternatively, blocks) transmission of activation along a connection between two other units.

An early cause for pessimism about the usefulness of neural computing units with threshold activation functions was the observation (Minsky and Papert, 1988[1969]) that they could only compute linearly separable boolean functions. A linearly separable boolean function on a real valued vector space, R^n, has value 1 for all points on one side of some subspace of dimension $n - 1$ and 0 for all points on the other side (i.e., a linear subspace separates the 1s from the 0's) . Examples of linear separable functions on bit vectors are the operations AND, OR, and NOT.

The early pessimism about using threshold units in computers was replaced by optimism in the 1980s on account of the realization that there are effective learning algorithms for functions computed across multiple layers of semi-linear units (e.g. the *backpropagation algorithm*—Werbos 1974; Rumelhart et al., 1986). This made it possible to automate to some extent the problem of decomposing a function into a succession of linearly separable mappings from one layer to the next.

More recently, the optimism has been tempered again, by the realization that our understanding of the mechanisms of neural network learning is very low-level. Essentially, it consists of some elaborations on the method of *gradient descent learning*, in which a poor solution to a problem is turned into a good one by making incremental improvements in response to different constraints on the task. This method can go a long way, but it tends to falter in complex domains (including the more complex grammar classes discussed above). In order to handle these domains, it appears that we need to have some additional insight into how neural devices are capable of solving complex problems.

This realization, in combination with the fact that incremental learning devices tend to be especially sensitive to the statistics of their environment has led many researchers to turn to statistical learning theory (e.g., Vapnik, 1995). One result has been that for problems in which one knows that the data have been drawn from a particular class of statistical distributions, it is possible to design the network and learning procedure in such a way that the activations of the outputs are guaranteed to converge on the probabilities of the outputs given the inputs (e.g., Rumelhart, et al., 1995; Bishop, 1995).

Helpful as the realizations have been, they have only slightly improved our understanding of how neural mechanisms might handle complex tasks, in part because our understanding of the high-order statistical models needed for such domains is still quite limited. This might be a cause, after so many ups and downs, for terminal pessimism, but there has been surprisingly little attention paid directly to the problem of representation. If we can achieve a better understanding of how complex neural devices might represent solutions to complex tasks, it may well make it easier to design learning devices that can discover these solutions. Therefore, in this paper, I do not address the problem of learning at all, but focus on a representation problem that has been a persistent challenge: representing arbitrary constituent structures with a context free grammar.

The idea is to design a device that represents such grammars in a way that builds on the strengths of neural computing. For example, it will not do, as Pollack (1987) notes, to implement context free grammars by using a distinct unit for every symbol on the stack (a localist representation of the stack). This would require the number of units to grow without bound, and would draw on none of the analogizing capabilities of distributed representations. Nor will it be particularly helpful to design a neural recognizer that models precisely the class of context-free grammars, no more, no less. Such a faithful implementation would have all the weaknesses of the symbolic prototype. A neural context free grammar is not of much interest unless it tells us something new about the nature of computation.

0.4 Previous work

In fact, the problem of finding a neurally reasonable representation for constituent structures and/or context-free grammars has been taken up many times. Several prior projects contain helpful ideas which I make use of here. Most of the proposals lack appeal not because the models don't have interesting new properties, but because they are not explored thoroughly enough to reveal these properties.

One strand of research focuses directly on using neural networks to recognize or generate complex languages. Another strand is more representationally oriented in that it focuses on the encoding of constituent structures in neural devices. I'll review the language recognizers first.

0.4.1 Language recognition with neural networks

Pollack (1987) suggests the essence of the proposal I make here in his "Neuring Machine", a connectionist version of the Turing Machine. The device uses two units with infinite precision to keep track of the two ends of the Turing Machine's infinite tape. The digits of the real-valued activations of these units correspond to successive symbols on the tape. Thus the model uses geometric scaling (multiplication by a constant contraction factor) to pack an unbounded amount of information into a bounded activation value. I describe a similar machine in more detail in Section 1.1.

For the multiplication operations, Pollack uses gating units, which, as I noted above are a variety of second-order unit. Siegelmann and Sontag (1991) redo Pollack's feat using only first-order connections. However, neither they nor Pollack show how a Turing machine embedded in connectionist machinery does anything different from its symbolic counterpart.

Sun et al. (1990a, b) propose a neural network pushdown automaton with a stack that is "external" in the sense that it uses distinct memory blocks in a symbolic machine to store the contents of the stack. The interesting thing about this model is that it represents the stack contents as a real-valued vector, which allows it to learn stack representations from data by gradient descent. The results are promising in that the network successfully learned to recognize the language of balanced parentheses and a few similarly simple languages. The authors are able to analyze each trained network and map its states to those in an isomorphic symbolic machine. However, they do not report on the learning of more complex languages. Nor do they explore the general computational properties of their mechanism, and hence do not reveal much about how it differs from its symbolic counterparts.

Kremer (1996) analyzes a number of types of neural networks with respect to their computational power, referring, again, to the Chomsky Hierarchy, thus helpfully relating neural architectures to known symbolic devices in a very systematic and comprehensive way. The proofs of formal equivalence, however, rely on hooking together pieces of connectionist machinery to simulate pieces of symbolic devices, and thus fail to reveal useful differences.

0.4.2 Constituent structure representation with neural networks

Several projects have been concerned with studying constituent structures in connectionist devices.

Pollack (1990) describes RAAMs (Recursive Auto-Associative Memories), a method of representing binary branching trees in a fixed-width vector of activations. The central device is a three-layer auto-associator which maps two separate vector representations for sequences of constituents to themselves via a hidden layer of half their combined length. This compression mechanism can be used iteratively to compile a fixed width representation for an entire tree. Moreover, the hidden-to-output mapping in the device can be used to recursively expand a tree representation back into its constituent symbols. Pollack notes that RAAMs can be thought of as recognizing the languages of strings which they successfully compress and uncompress. He finds them recognizing certain finite languages and showing signs of generalizing to constituent combinations that are not in the training set but their generalization ability is quite weak. Despite their weak generalization ability, RAAMs might turn out to provide some useful new insights if the principles by which they compress representations could be elucidated. However, no analysis of these principles has yet been provided.

Smolensky, in various writings (Smolensky 1988; Smolensky 1990; Smolensky et al. 1992; Prince and Smolensky 1993), has eloquently articulated the motivations for studying the relationship between symbolic and subsymbolic (connectionist) devices. In Smolensky (1990) and Smolensky et al. (1992), he and his colleagues propose *tensors* (or *tensor products*) as a formal model of this relationship. The tensor product of two vectors, $\vec{u}$ and $\vec{v}$ is the matrix T defined by $T_{ij} = u_i v_j$ where x_i is the ith entry of vector $\vec{x}$. Smolensky shows how tensor products can be used to address the *variable binding* problem for neural networks: how can multiple assignments of values to variables be stored in a fixed-width vector? The idea is to interpret one vector in the tensor as a variable and the other as a value. If certain conditions are met, then a number of such variable-value tensors can be added together without loss of information. Smolensky et al. describe a method of capitalizing on the variable-binding capabilities of tensor products to encode tree structures: iterated tensors of variable-denoting vectors identify positions in a tree structure; these can be combined (again, by tensor product) with value-denoting vectors to identify symbols at nodes; sums of such "phrasal" and "terminal" node tensors encode entire trees. Smolensky et al. (1992) note that such tree-descriptions can be used to encode context free grammars by defining a "harmony" function on their constituent tensors in such a way that only those trees that are well-formed with respect to a particular grammar have a total harmony in excess of some fixed threshold value.

Smolensky (1990)'s analysis of the graceful degradation in performance of the basic variable-binding tensors under the superimposition of many variable-value tensors is an appealing confirmation of the assertion that we can gain some advantages by seeking connectionist solutions to symbolic problems. The grammar encoding mechanism, on the other hand, is unhappily cumbersome because it requires the size of the storage vector to grow exponentially with the depth of the tree. Also, the harmony-assignment method of defining well-formed trees does not seem interestingly different from a symbolic grammar. For example, the framework of harmony-based grammar seems well-suited to modeling subtle differences among the grammaticality judgments that people assign to natural language sentences, but the mechanism proposed by Smolensky et al. (1992) only does this in a long-known-to-be-inaccurate way: counting ill-formed nodes. On the other hand, the use of iterated functions to keep track of recursive structures very useful. Plate (1994; 1995), to be discussed, and I, in Section 2, both propose ways of using iterated functions to encode tree structures with zero growth in the size of the storage vector.

Plate (1994; 1995) introduces *holographic reduced representations* (HRRs) which use *circular convolution* to solve the variable binding problem for vector-space computers. Closely related to Smolensky's tensor product, the circular convolution, $\vec{z}$ of vectors, $\vec{u}$ and $\vec{v}$, all of length n, is given by

$$z_i = \sum_{i=0}^{n-1} v_k u_{i-k}$$

where the subscripts are computed modulo n. Circular convolution can be thought of as a way of adding together various groups of entries in a tensor product in a way that keeps the dimension of the product equal to the dimension of its input vectors. As a result, superposition memories using circular convolution saturate more quickly than tensor products, and vectors of very high dimension (thousands of units) must be used to encode relatively few patterns. Nevertheless, the method supports the compression of tree structures into fixed-width vectors which are somewhat more analyzable than

Pollack's RAAMs. Plate works with vectors of random bits, which are likely to be nearly orthogonal to one another when their dimension is high, and provides an analysis of the saturation properties, also showing appealingly graceful degradation. One analysis shows that, in the processing of tree structures, one can interpret a partially completed backgrounded constituent as noise added to the salient representation of the constituent currently being processed. In this sense, HRRs also pick up on Pollack (1987)'s scaling technique for backgrounding information, which I make use of too. Plate does not explore the possibility of designing the representation vectors deterministically (rather than by using a random variable). This approach might allow one to construct more capacious HRR memories and gain more insight into their novel representational properties.

Elman (1991) studies a *simple recurrent network* (SRN) trained on the task of predicting words generated by a natural-language-like context free grammar. Although he does not analyze the computational power of the resulting machine, he provides examples of loop-shaped state space trajectories corresponding to constituents. This is a useful idea which I make use of below: since a constituent in a context free grammar can be thought of as a process which makes a relatively minor adjustment in the processor's state, it is natural to implement constituents in a metric space model using cycles or near-cycles (i.e., loops or near-loops). Wiles and Elman (1995) and Rodriguez (1995) study related networks which I discuss in more detail in Section 1.1.

Pollack (1991) defines a *Dynamical Recognizer* as a device,

$$M = (Z, \Sigma, \Omega, G)$$

where $Z \subset R^k$ is a vector space of states, Σ is a finite input alphabet, $\sigma_1, \sigma_2, \ldots, \sigma_n$, Ω is a set of transformations, $\omega_{\sigma_i} : R^k \rightarrow R^k$ corresonding to the symbols in Σ, and $G : Z \rightarrow \{0, 1\}$ is a "decision" function. The recognizer always starts in a particular state, called $z_k(0)$. It processes a string of symbols $\sigma_{s_1}\sigma_{s_2}\sigma_{s_3}\ldots\sigma_{s_k}$ one symbol at a time, performing transformation ω_i when symbol σ_i is processed. If G is 1 when the last symbol has been presented, the machine accepts the string, otherwise it rejects it. Pollack shows how a particular, trainable neural network with recurrent connections can be interpreted as a dynamical recognizer and he studies its behavior when it is trained on small samples of sentences from finite state languages that Tomita (1982) invented for testing learning machines. He finds that although it is trained on a finite number of strings, the network appears to be converging on an infinite-state device (modulo the limited precision of its implementation). Pollack argues that the limiting machine is an infinite state device by providing evidence that its distinct states form a *fractal*—that is, a set in which similar structures occur at arbitarily small scales.

The dynamical automata I discuss below differ from Pollack's dynamical recognizers only in that they permit the choice of function associated with a given symbol to be nonunique and contingent upon the current position in the state space. Instead of training dynamical devices and studying their behaviors, I design them explicitly to have particular behaviors. Although this leaves learning out of the picture and removes some of the appealing mystery associated with simulative investigations, it provides insight into the principles around which infinite-state computations may be organized. Thus it is a useful complement to Pollack's work.

0.5 Overview

There is a recurring theme in these research projects: iterative computations with functions that scale their input allow one to encode infinite state devices in a bounded, finite-dimensional representation space. In the remainder of the paper, I formalize this idea and study its implications. The iterative scaling is associated with fractals, which I use in the design of example dynamical automata in Section 1. In Section 2, I discuss a subset of dynamical automata which correspond to the class of context free languages. Section 3 shows how these dynamical context free grammars provide a new, more natural way of representing context-free grammars in neural devices. Section 4 shows how the dynamical automata framework reveals important relationships between computational devices which are invisible from the symbolic perspective. Section 5 discusses the usefulness of these findings.

1. Examples of dynamical automata.

In this section, I introduce fractals and dynamical automata informally. Section 2 provides a corresponding formal treatment.

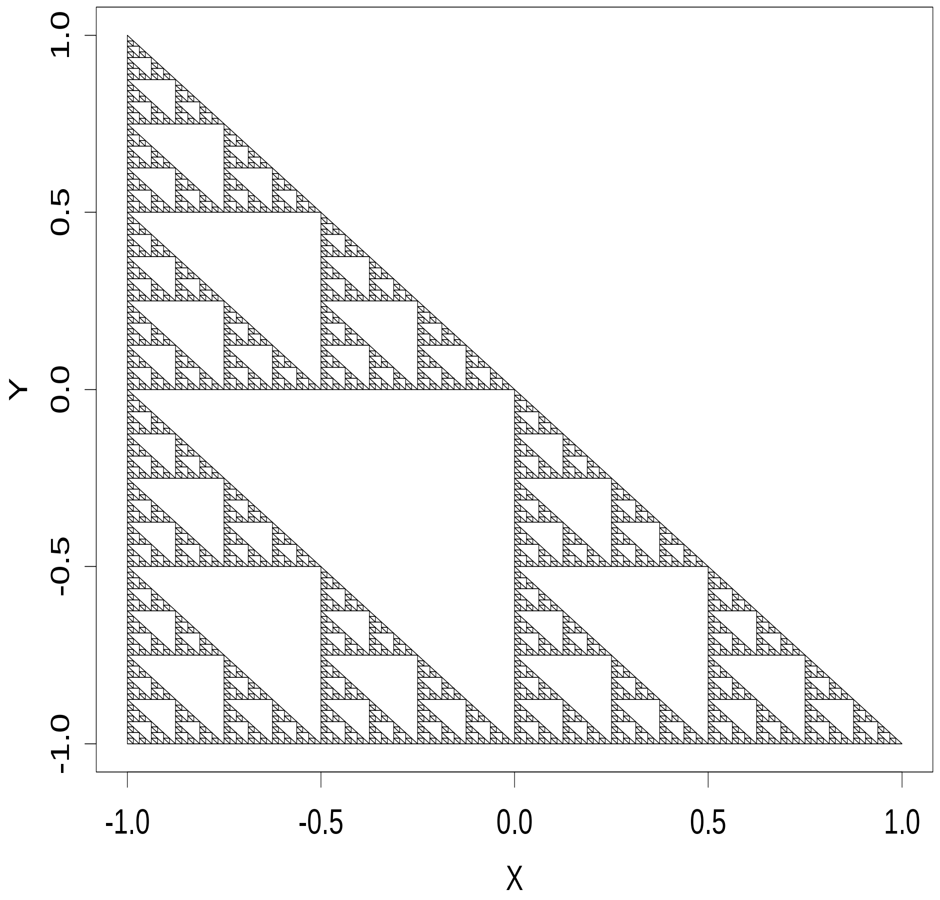
A fractal is a set of points which is self-similar at arbitrarily small scales. The classic example is the Cantor set. Consider the following infinite series of sets. The first set in the series is the interval [0, 1]. The next is the result of removing the middle third of this interval, namely, the set [0, 1/3] $\cup$ [2/3, 1]. The next is the result of removing the middle thirds of each of the contiguous intervals in the previous set. This process is repeated indefinitely. The set which is the limit of this process is called the Cantor Set.

The Cantor set is "self-similar at arbitrarily small scales" in the following sense. We associate the points in the set with their coordinates on the real number line. The function $f(x) = \frac{1}{3^n}x + x_0$ maps the original set in a 1-1 fashion onto a segment of the original set for various values of x_0 and all $n \in N$.[1] Thus, the Cantor Set contains arbitrarily small copies of itself.

It is worth noting that, under the definition just given, many other less exotic sets are also fractals. For example, the line segment [0, 1] is a fractal; the real number line is a fractal; the geometric series, $\{\frac{1}{r^n} : n \in N\}$ is a fractal. The bounded fractals are more useful for forming realistic implementations, so I'll focus on them here.

[1] N denotes the non-negative integers: 0, 1, 2, 3, etc.

Figure 1: The Sierpinski triangle.



Fractals can also exist in multiple dimensions. The Sierpinski Triangle (Figure (1)) is a simple generalization of the Cantor set. One can think of the Sierpinski triangle as the limit of the process of successively removing the "middle quarter" of a triangle to produce three new triangles.

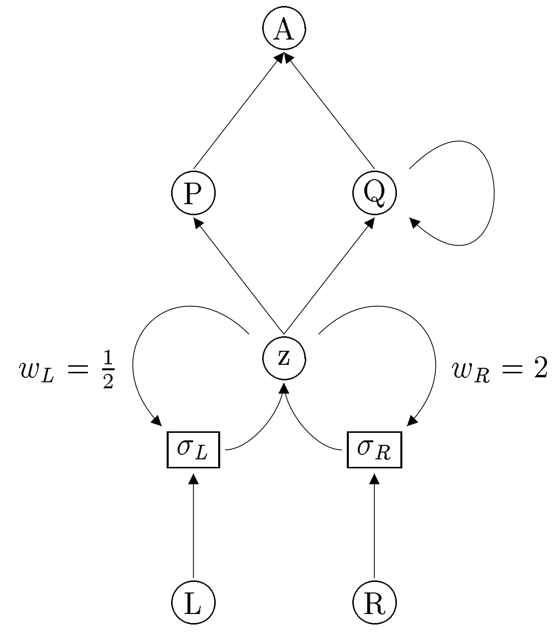
The next two subsections investigate some ways of using the recursive structure of fractals to keep track of computational processes.

1.1 A simple case: parenthesis balancing

Pollack (1991) noted that a very simple artificial neural device could recognize the language of balanced parentheses—the language in which left parentheses always precede corresponding right parentheses.[2] He describes a machine along the lines of that shown

[2]Moore (to appear) refers to this language as the *Dyck Language*.

Figure 2: A neural network for parenthesis balancing.



in Figure 2.

Initially, the activation of unit z is 1. If a left parenthesis is presented, the network activates unit L which has the effect of allowing transmission of activation along the connection labeled $w_L = 1/2$. Similarly, if a right parenthesis is presented, the network activates unit R which allows transmission of activation along the connection labeled $w_R = 2$. With each presentation of a symbol, z updates according to the rule $z(t+1) = f(\sum_i w_i a_i) = f(w_L \cdot z(t) + w_R \cdot z(t)) =$ either $f(w_L \cdot z(t))$ or $f(w_R \cdot z(t))$. The activation function f(x) is equal to x for $x \in [0, 2]$ and equal to 2 for $x > 2$. Unit P is a threshold unit which becomes active if $z > 0.75$. Unit Q is a self-reinforcing threshold unit which is initially inactive but becomes active and stays active if z ever exceeds 1.5. Unit A is a threshold unit which computes $P\ AND\ \neg Q$. Note that unit A becomes activated at the end of any string in which right parentheses follow and match left parentheses.

During the processing of grammatical strings, the activations of the z unit lie on the geometric series fractal, $\{\frac{1}{2^n} : n \in N\}$. In essence, this unit is simply a counter which keeps track of how many right parentheses are required to complete the string at any point. Although one could also use the set of non-negative integers, N, to perform the same function, the use of the bounded fractal permits the neural device to work with units of bounded activation.

This simple example thus provides an indication of how fractal objects are useful in forming neural recognizers for infinite-state languages.

Wiles and Elman (1995) study a backpropagation network that is trained on the closely-related language, $l^n r^n$. The model is presented with a sequence from $(l^n r^n)^*$ where n is randomly chosen from $\{1,\ldots, 11\}$ at each iteration. The task of the model is to predict successor symbols at each point. Note that the model will be effectively recognizing the language, $l^n r^n$, if it initially predicts just l, then predicts both l and r until an r occurs, then predicts just r until an appropriate number of rs have occurred, and returns, at this point, to predicting just l again. After many training episodes with different initial weight settings, Wiles and Elman found one network which generalized the pattern up to $n = 18$ (i.e. it performed as though it were recognizing $l^n r^n$ for $n \in \{1, \ldots, 18\}$).

Rodriguez et al. (to appear) noted that networks like Wiles and Elman's can be viewed as nonlinear dynamical systems. They analyzed the corresponding linear systems which closely approximated the behavior of the nonlinear systems and found that the computation of $l^n r^n$ was organized around a saddle point: when the network was receiving a string of l's, it was iterating the map associated with the stable manifold of the saddle point—in effect it was computing successive values of $x(t) = t_0 e^{-kt}$ for some positive k and $t = 0, 1, 2, 3, \ldots$; when it was receiving the corresponding string of r's it was iterating the map associated with the unstable manifold (same situation except $k < 0$ and the points are spread out along a different axis). With equally spaced values of t, the exponential equation $x(t) = t_0 e^{-kt}$ generates points on a geometric series fractal. Thus again, a parenthesis balancer is using geometric series fractals for its computation, this time along two different dimensions (the distinction between dimensions is a handy way of distinguishing the l and r states).

These two examples have shown how a particular type of fractal is useful for modeling parenthesis-balancing languages. This is helpful, but it is a very simple case. In the next section, I show how the same principles can be extended to a more complex case.

1.2: A more complex fractal grammar.

The grammar shown in Table 1 is a context free grammar.

This grammar generates strings in the standard manner: the start symbol, "S" is replaced with the string of symbols "A B C D" (in accord with rule 1a) or by no symbol (in accord with Rule 1b); if the former, then each of the symbols "A", "B", "C", and "D" is replaced with a string of symbols according to an appropriate rule. The process halts when the string contains no symbols that are recursively defined (no capital letter symbols, in this case). Rule 1b, where ϵ denotes the empty string, is included as a convenience—it gives the grammar the option of generating the empty string. Examples of strings generated by Grammar 1 are:

Table 1: Grammar 1.

Rule 1a.	S	$\rightarrow$	A B C D
Rule 1b.	S	$\rightarrow$	ϵ
Rule 2a.	A	$\rightarrow$	a
Rule 2b.	A	$\rightarrow$	a S
Rule 3a.	B	$\rightarrow$	b
Rule 3b.	B	$\rightarrow$	b S
Rule 4a.	C	$\rightarrow$	c
Rule 4b.	C	$\rightarrow$	c S
Rule 5a.	D	$\rightarrow$	d
Rule 5b.	D	$\rightarrow$	d S

(1) a b c d

(2) a a b c d b c d

(3) a b c a a b c d b c d d

(4) a b c d a b c d a b c d

Note that this grammar generates center-embedded structures (egs. (2) and (3)) to arbitrary depth. Thus its language is among those context free languages which cannot be generated by finite state machines.

A pushdown automaton for this grammar's language would need to keep track of each "abcd" string that has been started but not completed. For this purpose it could store a symbol corresponding to the last letter of any partially completed string on a pushdown stack. For example, if we store the symbol "A" whenever an embedding occurred under "a", "B" for an embedding under "b" and "C" for an embedding under "c", the stack states will be members of $\{A, B, C\}^*$.[3]

It turns out that we can use the Sierpinski Triangle to keep track of the stack states for this grammar. Consider the labeled triangle in Figure 3. Note that all the labels are at the midpoints of hypotenuses of subtriangles. The labeling scheme is organized so that each member of $\{A, B, C\}^*$ is the label of some midpoint.

[3]No stack symbol for "d" is needed since "d" completes the sequence "abcd".

Figure 3: An indexing scheme for selected points on the Sierpinski triangle. The points are the analogues of stack states in a pushdown automaton. The label on each point lists the stack with the top element first.

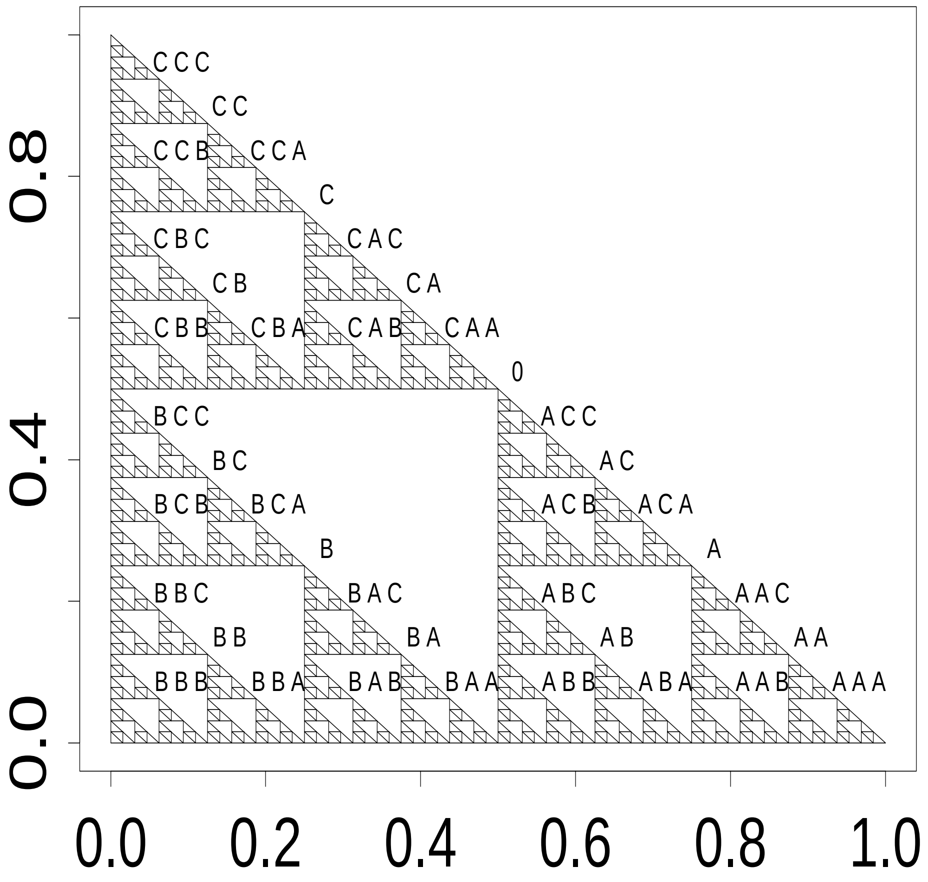


Table 2: State transitions for the Sierpinski version of Grammar 1.

Input	State change
a	$\vec{z} \leftarrow \frac{1}{2}\vec{z} + \binom{1/2}{0}$
b	$\vec{z} \leftarrow \vec{z} - \binom{1/2}{0}$
c	$\vec{z} \leftarrow \vec{z} + \binom{0}{1/2}$
d	$\vec{z} \leftarrow 2\left(\vec{z} - \binom{0}{1/2}\right)$

We define a computer that recognizes the language of Grammar 1 as follows. The essence of the computer is a pair of coordinates corresponding to a position on the Sierpinski triangle. Let $\vec{z}$ denote this pair. For convenience, we let the initial state of the system be the midpoint of the largest hypotenuse, i.e., $\vec{z}_0 = (1/2, 1/2)$. The state of the computer is updated as shown in Table 2.

One can intepret Table 2 intuitively in the following way. The system performs context-free embeddings by scaling its current state and switching origins. Initially, the origin is (0, 0) and the current state is (1/2, 1/2). When an "a" is presented, the system scales its current state down by a factor of two and moves the origin to (1/2, 0). When a "b" is presented, the system assumes the current origin is (1/2, 0) and moves it back to (0, 0). When a "c" is presented, the system assumes the origin is (0, 0) and moves it to (0, 1/2). When a "d" is presented, the system moves the origin back to (0, 0) and doubles the scale. Under these rules, the label on the current system state at each point (as per Figure 3) corresponds to the stack state of the pushdown automaton referred to earlier. This makes it possible to use the current position to predict which symbols are possible at each point during processing. Table 3 specifies the possibilities. If we specify that the Sierpinski computer must start at the point $(1/2, 1/2)$, make state transitions according to the rules in Table 3 as symbols are read from an input string, and return to $(1/2, 1/2)$ when the last symbol is read, then the computer functions as a recognizer for the language of Grammar 1.

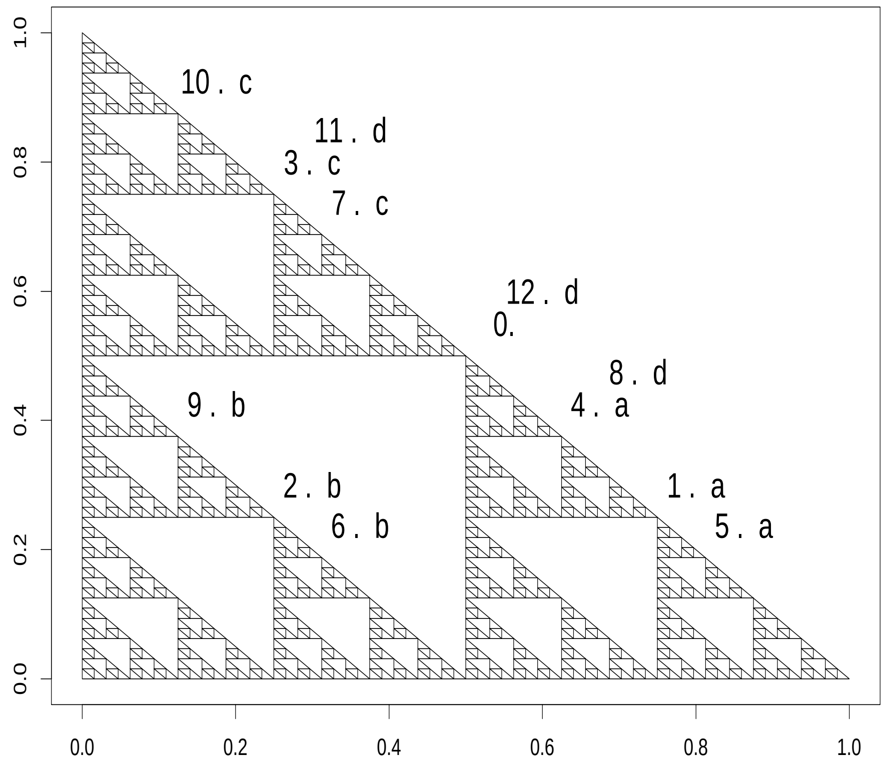
For illustration, the trajectory corresponding to string (3) above is shown in Figure 4 (**1. a** is the position after the first symbol, an *a*, has been processed; **2. b** is the position after the second symbol, a *b* has been processed, etc.)

This section has given an intuitive feel for how context free languages can be represented by dynamical automata. The next section formalizes the observations and provides a convenient way of designing stacks using fractals.

Table 3: Transition conditions for the Sierpinski version of Grammar 1.

State	Possible inputs
$z_1 > 1/2$ and $z_2 < 1/2$	a, b
$z_1 < 1/2$ and $z_2 < 1/2$	a, c
$z_1 < 1/2$ and $z_2 > 1/2$	a, d
$z_1 = 1/2$ and $z_2 = 1/2$	a

Figure 4: The trajectory on the Sierpinski triangle corresponding to the string, "a b c a a b c d b c d d".



2. General formulation.[4]

Distance measures and complete metric spaces provide the environment in which dynamical automata reside. I start by introducing these basic concepts.

Def. A *distance measure* on a set X is a function $d : X \times X \to R$ which satisfies:

(i) $d(x, x) = 0$ for all $x \in X$

(ii) $d(x, y) = d(y, x)$ for all $x, y \in X$

(iii) $d(x, z) \leq d(x, y) + d(y, z)$ for all $x, y, z \in X$

Def. A *metric space*, (X, d) is a set X together with a corresponding distance measure, d.

Def. A sequence $\{x_n\}_{n=1}^{\infty}$ of points in a metric space (X, d) is called a *Cauchy sequence* if for any number $\epsilon > 0$ there exists an integer $N > 0$ such that if m and n are integers greater than N then

$$d(x_n, x_m) < \epsilon$$

Def. A sequence $\{x_n\}_{n=1}^{\infty}$ of points in a metric space (X, d) is said to *converge* to a point $x \in X$ if, for any number $\epsilon > 0$ there is an integer N such that for all $n > N$,

$$d(x_n, x) < \epsilon$$

Def. A metric space $M = (X, d)$ is said to be *complete* if every Cauchy sequence in M converges to a point in X.

Another fundamental component of a dynamical automaton is the partition:

Def. A *finite partition* of a space X is a set of sets, $m_1, m_2, \ldots, m_K$ where $K \in \{1, 2, 3, \ldots\}$ such that

(i) $m_i \subseteq X$ for each $i \in \{1, \ldots, K\}$

(ii) $m_i \cap m_j = \oslash$ for $i \neq j$ and $i, j, \in \{1, \ldots, K\}$

(iii) $\bigcup_{i=1}^{K} m_i = X$

We are now in a position to define the dynamical automaton.

Def. A *dynamical automaton* is a device, M, with the following structure:

$$M = (X, F, P, \Sigma, IM, \mathcal{O}, FR) \tag{4}$$

[4]A summary of this discussion is provided in Tabor (submitted-b). A similar discussion is provided in Tabor (submitted-a).

(1) The *space*, X, is a complete metric space.[5]

(2) The *function list*, $\mathcal{F}$, consists of a finite number of functions, $w_1, \ldots w_N$ where $w_i : X \rightarrow X$ for each $i \in \{1, \ldots, N\}$.

(3) The *partition*, P, is a finite partition of X and consists of compartments, $m_1, \ldots m_K$.

(4) The *input list* is a finite set of symbols drawn from an alphabet, Σ.

(5) The *input mapping* is a three-place relation, $IM : P \times \Sigma \times \mathcal{F} \rightarrow \{0, 1\}$ which specifies for each compartment, m, and each input symbol, s, what function(s) can be performed if symbol s is presented when the system state is in compartment m. If $IM(m, s, f) = 0$ for all $f \in AR$ then symbol s cannot be presented when the system is in compartment m.

(6) The machine always starts at the *start state*, $\mathcal{O} \in X$. If, as successive symbols from an input string are presented, it makes transitions consistent with the Input Mapping, and arrives in the *final region*, FR, then the input string is accepted.

Dynamical automata in general have super-Turing computing capacity (Moore, to appear). Here I focus on a particular way of constraining them to let them emulate context free grammars/pushdown automata. The notion of an *iterated function system*, defined by Barnsley (1988), is useful in this regard. Barnsley employs the notion of a contraction mapping:

Def. A function $f : X \rightarrow X$ is called a *contraction mapping* on metric space X if there exists $0 \leq k < 1$ such that $d(f(x), f(y)) \leq kd(x, y)$ for all $x, y \in X$.

He then defines an iterated function system as, in effect, a set of contraction mappings which share a space:

Def. (Barnsley, 1988) An *Iterated Function System* (IFS) consists of a complete metric space (X, d) together with a finite set of contraction mappings, $w_n : X \rightarrow X$. The notation for such an IFS is $\{X; w_1, \ldots, w_N\}$.

Here, I find it convenient to generalize the notion by removing the requirement that the functions be contraction mappings. Hence:

Def. A *Generalized Iterated Function System* (GIFS) consists of a complete metric space (X, d) together with a finite set of functions, $w_i : X \rightarrow X$, $i \in \{1, \ldots, N\}$.

The idea is to use the functions of an iterated function system to move around in a bounded space in the manner illustrated in Section 1. For the purpose of emulating a context free grammar, it is essential to be able to keep track of arbitrary stack states. This implies that the system needs to be capable of following an arbitrary number of

[5]In this paper, I do not use completeness in the analysis. Nevertheless, it is a convenient property to have for other anlayses so I build it into the framework here.

branching paths. To this end, I define the notion of a "cascade point". First I give some preliminary definitions.

Def. Let $S = \{X; w_1 \ldots w_N\}$ be a GIFS. Consider a string, $\sigma = \sigma_1\sigma_2 \ldots \sigma_K$ with $\sigma_i \in \{1, \ldots, N\}$ for $i \in \{1, \ldots, K\}$. Let w_σ denote the composition $w_{\sigma_1} \circ w_{\sigma_2} \circ \ldots \circ w_{\sigma_K}$. For $x_0 \in X$, the set $C = \{x \in X : x = w_\sigma(x_0)$ for some $\sigma \in \{1, 2, \ldots N\}^*\}$ is called the *orbit* of x_0 under S.

Thus each point on the orbit of a point x_0 can be reached by starting at x_0 and applying some sequence of operations from the GIFS. The orbit itself is the set of all such points. The notion of orbit is perhaps more naturally applied to sets than to individual points (Barnsley , 1988) but there is no need to examine set orbits here so I have simply defined the case in which the starting set has only one point in it.

Def. Let $S = \{X; w_1 \ldots w_N\}$ be a GIFS. Let $\sigma = \sigma_1\sigma_2 \ldots \sigma_K$ be a string in $\{1, \ldots, N\}^*$. Consider the point $x = w_\sigma(x_0) \in X$. The string σ is called an x_0-*address* of the point x under S.

Def. Let $S = \{X; w_1 \ldots w_N\}$ be a GIFS. Let x_0 be a point in X. If every point in the orbit of x_0 has a unique x_0-address, then x_0 is called a *cascade point.* In this case, the orbit, C, of x_0 is called the *cascade* of x_0. If $x \in C$ and $x \neq x_0$ then the first symbol of the x_0-address of x is denoted $\text{top}_C(x)$. If $x = x_0$, we set $\text{top}_C(x) = \epsilon$.

Cascades form the essence of the mechanism to be proposed for simulating push-down automata with dynamical automata. Note that when it is associated with the functions of a GIFS, a cascade can be thought of as a binary branching tree of arbitrary depth. The mechanism to be described uses each point on a cascade as a representation for the stack contents. Cascades are a particular species of the fractals discussed in Section 1. Since I do not need to use fractals more generally, I do not define them formally here. (See Barnsely, 1988 for a formal treatment.)

It is useful to be able to identify cascade points of GIFSs. The following definition and theorem help with this.

Def. Let X be a metric space with GIFS $S = \{X; w_1 \ldots w_N\}$. A set $O \subset X$ is called a *pooling set* of S if it satisfies the following:

(i) $w_i(O) \cap w_j(O) = \oslash$ for $i, j \in \{1, \ldots n\}$ and $i \neq j$.

(ii) $\bigcup_{i=1}^{n} w_i(O) \subset O$

The set of points in O that are not in $\bigcup_{i=1}^{n} w_i(O)$ is called the *crest* of O.

Thm 1. Let $S = \{X; w_1 \ldots w_N\}$ be a GIFS where $w_1, \ldots, w_N$ are one-to-one functions. Suppose $O \subset S$ is a pooling set of S and x_0 is in the crest of O. Then x_0 is a cascade point of S.

Pf: This theorem claims, in effect, that if two identical dynamical automata follow a

GIFS into a cascade and their paths diverge at some point, then they will never rejoin. I prove the theorem by assuming that rejoins are possible and deriving a contradiction.

Suppose, contrary to fact, that there exists $\sigma = \sigma_1\sigma_2 \dots \sigma_J$ and $\rho = \rho_1\rho_2 \dots \rho_K \in \{1, \dots, n\}^*$ where $w_\sigma(x_0) = w_\rho(x_0)$ but $\sigma \neq \rho$. Let M be equal to the minimum of J and K. Then there are two possibilities to consider. Either the two paths diverge before they come to an end or one comes to an end and the other keeps going. That is, either (i) there exists $h \in \{1, \dots, M\}$ such that $\sigma_i = \rho_i$ for $i \in \{1, \dots, h-1\}$ and $\sigma_h \neq \rho_h$ or (ii) $\sigma_i = \rho_i$ for $i \in \{1, \dots M\}$ and $K \neq J$.

Under case (i), the fact that $w_\sigma(x_0) = w_\rho(x_0)$ and the fact that $w_1, \dots, w_N$ are one-to-one imply that $w_{\sigma_{-h}}(x_0) = w_{\rho_{-h}}(x_0)$ where $\sigma_{-h} = \sigma_h \dots \sigma_J$ and $\rho_{-h} = \rho_h \dots \rho_K$. But $w_{\sigma_{-(h+1)}}(x_0) \in O$ and $w_{\rho_{-(h+1)}}(x_0) \in O$ by condition (ii) of the definition of pooling set. Therefore $w_{\sigma_{-h}}(x_0) \neq w_{\rho_{-h}}(x_0)$ by condition (i) of the definition of pooling set, a contradiction.

Without loss of generality, we can assume that $J > K$ in case (ii). In this case, the fact that $w_\sigma(x_0) = w_\rho(x_0)$ and the fact that $w_1, \dots, w_N$ are one-to-one imply that $w_{\sigma_{-(M+1)}}(x_0) = x_0$. Condition (ii) of the definition of pooling set thus implies that x_0 is in the complement of the crest of O. But this contradicts the assumption that x_0 is in the crest of O.

Since the assumption led to a contradiction in both cases, it must be the case that $\sigma = \rho$. $\square$

Now we are in a position to define the analogs of the operations of a pushdown automaton.

Def. Let $S = \{X; w_1 \dots w_N\}$ be a GIFS with cascade point x_0 and corresponding cascade C. Then $w_i : C \to C$ is called a *push function* on C.

Def. Let $S = \{X; w_1 \dots w_N\}$ be a GIFS with cascade point x_0 and corresponding cascade C. Let $Y = \{x \in C : top_C(x) = i \text{ for } i \in \{1, \dots, N\}\}$. Suppose w_i is invertible on Y. Then the function $f : Y \to C$ such that $f(x) = w_i^{-1}(x)$ is called a *pop function* on C.

Def. The push, pop, and pop $\circ$ push functions (composition of one pop with one push) functions C are called *stack functions* on C.

Pursuing the analogy with a pushdown automaton, the following two definitions make it possible to keep track of changes in the control state.

Def. Let $S = \{X; w_1 \dots w_n\}$ be a GIFS. Let C_1 and C_2 be disjoint cascades under S with cascade points x_{10} and x_{20} respectively. Then the function $f : C_1 \to C_2$ such that for all $x \in C_1$ the x_{10}-address of x is equal to the x_{20}-address of $f(x)$ is called a *switch function.* Note that f is one-to-one and onto, and hence invertible for all $x' \in C_2$.

Def. If $C_1, \dots, C_K$ are cascades on GIFS S satisfying

$$C_i \cap C_j = \oslash \text{ for } i \neq j.$$

(i.e., they are *disjoint*) and $x \in C_i$ for $i \in \{1, \ldots, K\}$ then i is called the *index* of x with respect to the set $\{C_1, \ldots, C_K\}$

The idea, then, is to piece together stack functions and switch functions to design a machine that performs the operations of a pushdown automaton:

Def. Let M be a dynamical automaton on metric space X. We say M is a *pushdown dynamical automaton* (PDDA) if there exists a GIFS, $S = \{X; w_1, \ldots, w_N\}$ with cascade points $x_{10}, x_{20}, \ldots, x_{K0} \in X$, $K \in \{1, 2, 3, \ldots\}$ and corresponding cascades, $C_1, C_2, \ldots, C_K$ such that

(i) $C_1, C_2, \ldots, C_K$ are disjoint.

(ii) For $x \in \bigcup_{i=1}^{N} C_i$, the partition compartment of x is determined by the conjunction of the index, i, of x and $\text{top}_{C_i}(x)$.

(iii) Let m be a compartment of the partition of M. Each function $f : m \rightarrow X$ in the input mapping is either a stack function, a switch function, or a composition of the two when restricted to points on one of the cascades.

(iv) The start state, O, and final region, FR, of M are contained in $\bigcup_{i=1}^{K} x_{i0}$.

The next step is to show that PDDA's behave like pushdown automata and thus recognize context free languages. Here I follow the notation of Hopcroft and Ullman (1979) for pushdown automata.

Def. (Hopcroft and Ullman) A *pushdown automaton* (PDA) is a machine, $M = (Q, \Sigma, , , \delta, q_0, Z_0, F)$ where

(1) Q is a finite set of *states.*
(2) Σ is a finite alphabet called the *input alphabet.*
(3) , is a finite alphabet called the *stack alphabet.*
(4) $q_0 \in Q$ is the *initial state.*
(5) $Z_0 \in$, is the *start symbol.*
(6) $F \subset Q$ is the set of *final states.*
(7) δ is a mapping from $Q \times (\Sigma \cup \{\epsilon\}) \times$, to finite subsets of $Q \times$, *.

We assume that a pushdown automaton, M, is associated with a string of stack symbols called the *stack.* The first symbol in the stack string is called the *top of the stack.* Initially, the stack consists of just the symbol Z_0 and thus Z_0 is the top of the stack.

We think of M as processing a string of symbols $\sigma \in \Sigma^*$ one symbol at a time, from left to right. It is convenient to define the *instantaneous description* of pushdown automaton M at each point in time as the triple, (q, w, γ), where q is the current state, w is the right substring of σ that has not yet been processed, and γ is the current state of the stack. For $Z \in$, and $\alpha, \beta \in$, *, we say that $(q, a\sigma, Z\alpha)$ *can go to* $(p, w, \beta\alpha)$ (or $(q, a\sigma, Z\alpha) \vdash (p, w, \beta\alpha)$) if $\delta(q, a, Z)$ contains (p, β). If, by a series of "go to" moves, M can get from $(q, \sigma\rho, \gamma)$ to (p, ρ, κ) upon processing the symbols of σ, then we say that $(q, \sigma\rho, \gamma)$ *leads to* (p, ρ, κ). If (q_0, σ, Z_0) leads to (p, ϵ, γ) for some $p \in F$ and $\gamma \in$, *, then we say that M *accepts* σ *by final state.* If (q_0, σ, Z_0) leads to (p, ϵ, ϵ) for some $p \in Q$ then we say that M *accepts* σ *by empty stack.*

For convenience, I refer to the sequence of symbols on the stack at the current time as the *stack state* of the PDA. I refer to the current member of Q as the *control state.*

PDDAs are very similar in form to PDAs. The distinct cascade indices of a PDDA correspond to the distinct control states of a PDA. The distinct top values of each cascade in a PDDA correspond to the distinct top-of-stack symbols of a PDA. Computation in a PDDA is a matter of switching cascades and/or adding or removing symbols from the addresses of points. Computation in a PDA is a matter of switching control states and/or pushing or popping the stack. The rest of the formal development in this section is devoted to being precise about the details of this analogy. At the end of the section some examples are provided.

Def. The set of finite strings accepted by a pushdown automaton M is called the *language recognized by* M.

It turns out that the set of languages recognized by pushdown automata by final state is the same as the set of languages recognized by pushdown automata by empty stack (Hopcroft and Ullman, p. 114). Therefore, if one is interested only in demonstrating language recognition equivalence, it is sufficient to show equivalence under recognition by final state, or recognition by empty stack.

It is well known (Hopcroft and Ullman, pp. 115, 116) that the set of languages recognized by pushdown automata is precisely the set of context free languages.

It will be useful to consider some simple variants on Hopcroft and Ullman's pushdown automata.

Def. The *single in/out pushdown automata* are those which satisfy the condition that if $\delta(q, a, Z)$ contains (p, γ), then γ is either ϵ (a pop move), Z (no change in the stack), or YZ for some stack symbol Y (a push move).

It is easy to convert a general pushdown automaton to a single in/out pushdown automaton by adding extra states which push symbols onto the stack in the absence of input wherever multiple successive push moves are allowed. Moreover, the single in/out pushdown automata are a subset of general pushdown automata. Thus the set of languages recognized by single in/out PDA's is precisely the set of context-free languages.

Def. The *ground zero* pushdown automata are those which start with an empty stack and accept a string when the stack is empty and the control state, q, is among the set of final states, F.

Given the equivalence of standard PDA's and single in/out PDAs, it is easy to see that ground zero PDA's are also equivalent. A single in/out PDA which accepts string σ must eventually arrive at a point where Z_0 is on the stack and it pops Z_0. Instead of letting the machine pop Z_0 at this point, let it switch to a new state q' where $\{q'\} = F$. This machine recognizes the same language as the single in/out PDA but Z_0 is superfluous. Therefore, with some minor adjustments in the notation associated with the transition function, we can let the machine start and end with an empty stack. To make the reverse translation, we add another state which is only entered when the ground-zero automaton is in F and its stack is empty, and make use of the equivalence between acceptance by final state and acceptance by empty stack.

With these technical adjustments out of the way, it is straightforward to show that pushdown dynamical automata (PDDA's) generate precisely the context free languages.

Thm 2. The set of languages recognized by pushdown dynamical automata (PDDA's) is precisely the set of context free languages.

Pf: I proceed by showing equivalence between pushdown dynamical automata and ground zero pushdown automata.

Part (i). (Every pushdown dynamical automaton recognizes the language of some ground zero pushdown automaton).

Consider the pushdown dynamical automaton

$$M_d = (X, F, P, \Sigma_d, IM, x_{10}, FR)$$

Let the associated GIFS be $S = \{X; w_1, \ldots, w_N\}$ with cascade points $x_{10}, x_{20}, \ldots, x_{K0} \in X$ for $K \in \{1, 2, 3, \ldots\}$ and corresponding cascades, $C_1, C_2, \ldots, C_K$. Define a corresponding ground zero pushdown automaton

$$M_a = (Q, \Sigma_a, \Gamma, \delta, q, Z_0, F)$$

as follows. Because M_a is a ground zero PDA, its stack is initially empty ($Z_0 = \epsilon$). Let $\Sigma_a = \Sigma_d$. For each cascade, C_i, if M_d is on C_i, let the control state of M_a be q_i. Thus define Q as $\{q_i : i \in \{1, \ldots, K\}\}$. Let Γ be $\{1, \ldots, N\}$. Let δ be defined as follows. Consider x a point in $\bigcup_{i=1}^{K} C_i$. For each possible index value $i \in \{1, \ldots, K\}$, and each possible value $Z = \text{top}_{C_i}(x) \in \{\epsilon, 1, \ldots N\}$, compute the partition compartment to which x belongs (this is possible under the definition of a PDDA). Let this partition compartment be compartment j. Examine the input mapping, IM, for rows containing j. For each such row, (j, s, f) for $s \in \Sigma$ and $f \in \mathcal{F}$, let δ be defined as follows:

(i) If $f = w_h$ is a push function, then let (q_i, hZ) be a member of $\delta(q_i, s, Z)$.

(ii) If $f = w_h^{-1}$ is a pop function, then let (q_i, ϵ) be a member of $\delta(q_i, s, Z)$.

(iii) If f is a switch function which switches from cascade i to cascade l, then let (q_l, Z) be a member of $\delta(q_i, s, Z)$.

(iv) Handle the composite functions analogously.

(v) If x_{i0} is in the final region of M_d, then let $q_i \in F$.

Note that δ is well-defined in every case because $C_1, \ldots, C_K$ are disjoint cascades and M_d performs its computations on their union.

Let q_1 be the initial state of M_a (i.e., let the initial state bear the index of the start state of M_d).

For $x = x_{10}$ it can be truly asserted that if the index of x is i, the state of M_a is q_i, and if $\text{top}_{C_i}(x)$ is Z, then the top of the stack of M_a is Z. Moreover, this situation is preserved under δ. Thus, if the state x, moves by legal transitions from x_{10} to x_{j0} under M_d during the processing of string σ, then (q_0, σ, ϵ) leads to $(q_j, \epsilon, \epsilon)$ under M_a and vice versa. Thus M_a and M_d recognize the same language.

Part (ii). (The language of each ground zero pushdown automaton is recognized by some pushdown dynamical automaton).

This part is the reverse of the previous part, with a particular fractal specified.

Consider the ground zero pushdown automaton,

$$M_a = (Q, \Sigma_a, , , \delta, q_0, F) \tag{5}$$

We need to define a corresponding pushdown dynamical automaton,

$$M_d = (X, F, P, \Sigma_d, IM, x_{10}, FR) \tag{6}$$

Suppose $| \, , \, | = N$, N a positive integer. Let $\vec{e}_n \in R^N$ be the vector with a 1 on dimension n and 0's on all other dimensions. Suppose $| \, Q \, | = K$, K a positive integer. Define the GIFS,

$$S = \{R^{N+1}; w_n, \quad n \in \{1, \ldots, N\}\}$$

where

$$w_n(\vec{x}) = \begin{pmatrix} \frac{1}{2}\vec{x}_{[N]} \\ x_{N+1} \end{pmatrix} + \begin{pmatrix} \frac{1}{2}\vec{e}_n \\ 0 \end{pmatrix}$$

where $\vec{x}_{[N]} = \begin{pmatrix} x_1 \\ \ldots \\ x_N \end{pmatrix}$

Consider the set $\mathcal{C}_0 = \{\vec{x}_{k0} : \vec{x}_{k0} = \binom{\vec{1/2}}{k/K}, \vec{1/2} = \frac{1}{2}\sum_{i=1}^N \vec{e}_i$ and $k \in \{1,\ldots,K\}\}$. Then the members of $\mathcal{C}_0$ are cascade points of S. To see this, consider $\vec{x}_{k0}$ and let O be the open unit hypercube in the positive quadrant of R^{N+1} with a corner at the origin. Note that for all $\vec{x} \in O$, the nth coordinate of $w_n(\vec{x})$ is in the interval $(1/2, 1)$ and the mth coordinate of $w_n(\vec{x})$ is in $(0, 1/2)$ for $m \neq n$, $m, n \leq N$. Therefore, each $w_n(O) \subset O$, and $w_n(O) \cap w_m(O) = \oslash$. Moreover, $\vec{x}_{k0} \notin w_n(O)$ for each n. Thus O is a pooling set of S and, by Theorem 1, x_{k0} is a cascade point. Moreover, the cascades $C_1, \ldots, C_K$ corresponding to $x_{10}, \ldots, x_{K0}$ are disjoint.

Let the N stack states of M_a be labeled $1, \ldots, N$. Assume, without loss of generality, that $Z_0 = 1$. Let the K "finite states" of M_a be labeled $q_1, \ldots, q_K$. Assume, without loss of generality, that $q_0 = q_1$.

Now I define the parts of M_d.

Let X be $[0,1]^N$.

Let $\Sigma_d = \Sigma_a$.

Let the partition P be partially specified by $\{X_n \times \frac{i}{K}$ for $n \in \{0,\ldots,N\}$ and $i \in \{1,\ldots,K\} : X_n = (0,\frac{1}{2})^N + \frac{1}{2}\vec{e}_n$ for $n \in \{1,\ldots,N\}$ and $X_0 = \vec{1/2} \in R^N\}$. Let the index of compartment $X_n \times \frac{i}{K}$ be $M_{(i,n)}$ for $i \in \{1,\ldots,K\}$ and $n \in \{0,1,\ldots,N\}$. Note that since x_{10} is the start state of M_d, M_d is initially in compartment $M_{(1,0)}$.

Build the input mapping, IM as follows.

(i) If (q_i, hn) is a member of $\delta(q_i, s, n)$ then let $(M_{(i,n)}, s, w_h)$ be a member of IM. (push function)

(ii) If (q_i, ϵ) is a member of $\delta(q_i, s, n)$ then let $(M_{(i,n)}, s, w_k^{-1})$ be a member of IM. (pop function)

(iii) If (q_i, n) is a member of $\delta(q_j, s, n)$ then let $(M_{(i,n)}, s, \binom{\vec{x}}{x_{N+1} + \frac{i-j}{K}})$ be a member of IM. (switch function)

(iv) Handle the composite cases analogously.

Note that s could be ϵ (no symbol) and n could be ϵ (empty stack) in each of these cases.

Compile $\mathcal{F}$ as the collection of functions that occur in the third column of IM.

For each, $i \in \{1,\ldots,K\}$, if $q_i \in F$, then let x_{i0} be a member of FR.

At the beginning of processing, when the state is q_1 and top of the stack of M_a is 1, the current state $\vec{x}$ of M_d (namely, $\vec{x}_{10}$) has index 1 and $\text{top}_{C_1}(\vec{x}) = \epsilon$. Thus it can be said at this point that when the state of M_a is q_i and the top of the stack is Z, then the current state x of M_d has index i and $\text{top}_{C_i}(\vec{x}) = Z$ and vice versa. Moreover, the definition of the input mapping implies that this situation never changes during the course of processing a grammatical string. Thus if (q_0, σ, ϵ) leads, under M_a to $(q_j, \epsilon, \epsilon)$ for $q_j \in F$ then the state $\vec{x}$, moves by legal transitions under M_a from $\vec{x}_{10}$ to $\vec{x}_{j0} \in FR$ during the processing of string σ and vice versa. Thus M_a and M_d recognize the same language. □

Examples.

The remainder of this section gives some examples which show how the results above make it easy to design pushdown dynamical automata for various context free languages.

Example 2.1. A simple dynamical automaton for balancing parentheses is given by

$$M = ([0,1], \{\frac{1}{2}x, 2x\}, P, \{l, r\}, IM, 1, \{1\}) \tag{7}$$

where the partition, P, is given by

Index	Compartment
1	$\{1\}$
2	$[0,1)$

and the input mapping, IM is given by

Compartment Index	Symbol	Function
1	l	$x \rightarrow \frac{1}{2}x$
2	l	$x \rightarrow \frac{1}{2}x$
2	r	$x \rightarrow 2x$

We can use Theorem 2 to show that this dynamical automaton generates a context free language. First, we need to choose an appropriate iterated function system. Consider the GIFS, $S = \{R; w_1 = \frac{1}{2}x\}$. Note that 1 is a cascade point of S. This is evident from the fact that w_1 is the only function in S and that $0 < w_1(x) < x$ for all $x > 0$.

Now we will show that M is a PDDA under S. Let C be the cascade of the point, 1. $\{C\}$ satisfies condition (i) of the definition of a PDDA trivially since there is only one cascade. Also, for $x \in C$, if $\text{top}_C(x) = 0$ then the partition is 1 and if $\text{top}_C(x) = 1$ then the partition is 2 so $\text{top}_C(x)$ alone determines the partition (condition (ii)). There are three functions in the input mapping. The first, $f : \{1\} \rightarrow R = \frac{1}{2}x$ is a push function when restricted to points on C. The second, $f : [0,1) \rightarrow R = \frac{1}{2}x$ is similarly a push function. The third, $f : [0,1) \to R = 2x$ is similarly a pop function. Thus condition (iii) is satisfied. Moreover, M is initially at the cascade point, 1, its final region is $\{1\}$, so condition (iv) is satisfied. This shows that M recognizes a context-free language.

Example 2.2. A closely-related PDDA recognizes $l^n r^n$ for $n \in \{1, 2, 3, \ldots\}$. Let

$$M = (\begin{pmatrix}[0,1]\\ [0,1]\end{pmatrix}, \{\begin{pmatrix}\frac{1}{2}x_1\\ x_2\end{pmatrix}, \begin{pmatrix}2x_1\\ x_2+1\end{pmatrix}, \begin{pmatrix}2x_1\\ x_2\end{pmatrix}\}, P, \{l, r\}, IM, \begin{pmatrix}1\\ 0\end{pmatrix}, \begin{pmatrix}1\\ 1\end{pmatrix}) \tag{8}$$

where $\binom{[a,b]}{[c,d]}$ denotes the set of points $\binom{x_1}{x_2}$ such that x_1 is in the interval $[a, b]$ and x_2 is in the interval $[c, d]$. Relevant parts of P are given by

Index	Compartment
1	$\binom{1}{0}$
2	$\binom{(0,1)}{0}$
3	$\binom{(0,1)}{1}$
4	$\binom{1}{1}$

and IM is given by

Compartment Index	Symbol	Function
1	l	$\vec{x} \to \binom{\frac{1}{2}x_1}{x_2}$
2	l	$\vec{x} \to \binom{\frac{1}{2}x_1}{x_2}$
2	r	$\vec{x} \to \binom{2x_1}{x_2+1}$
3	r	$\vec{x} \to \binom{2x_1}{x_2}$

Note that input mapping IM permits no moves out of the final state so concatenations of sentences from $l^n r^n$ are properly disallowed.

In this case to establish context free language status, we consider the GIFS, $S = \{R^2; w_1 = \binom{\frac{1}{2}x_1}{x_2}\}$. Note that $x_{10} = \binom{1}{0}$ and $x_{20} = \binom{1}{1}$ are both cascade points and they have disjoint cascades (condition (i) of the definition of a PDDA). Let these cascades be called C_1 and C_2, respectively. Note that the partition compartment is 1 if the index of x is 1 and $\text{top}_{C_1}(x) = \epsilon$. It is 2 if the index of x is 1 and $\text{top}_{C_1}(x) = 1$. It is 3 if the index of x is 2 and $\text{top}_{C_2}(x) = 1$. It is 4 if the index of x is 2 and $\text{top}_{C_2}(x) = \epsilon$. Thus the compartment is predictable from the conjunction of the index and the value of top in every case (condition (ii)). The function $\binom{x_1/2}{x_2}$ restricted to C_1 is a push function. The function $\binom{2x_1}{x_2}$ restricted to C_2 is a pop function. The function $\binom{2x_1}{x_2+1}$ is equal to $\binom{2x_1}{x_2} \circ \binom{x_1}{x_2+1}$, i.e., it is the composition of a switch function from C_1 to C_2 and a pop function on C_2 (condition (iii)). Since, moreover, the start and end points are the cascade points of C_1 and C_2 respectively (condition (iv)), M satisfies the definition of a PDDA. Thus, M recognizes a context free language.

Example 2.3. The dynamical automaton described in Section 1.2 recognizes the language of Grammar 1. We can write the Dynamical Grammar of Section 1.2 in the notation of Section 2 as follows.

$$M = (triangle(\begin{pmatrix}1\\0\end{pmatrix}, \begin{pmatrix}0\\0\end{pmatrix}, \begin{pmatrix}0\\1\end{pmatrix}), \mathcal{F}, P, \{a, b, c\}, IM, \begin{pmatrix}1/2\\1/2\end{pmatrix}, \{\begin{pmatrix}1/2\\1/2\end{pmatrix}\}) \tag{9}$$

where $triangle(x, y, z)$ refers to the interior and boundary of the triangle with vertices at x, y and z. The function list, $\mathcal{F}$, is the list of functions given in Table 2. The partition P is

Index	Compartment
1	$\{\binom{\frac{1}{2}}{\frac{1}{2}}\}$
2	$triangle(\binom{1}{0}, \binom{\frac{1}{2}}{0}, \binom{\frac{1}{2}}{\frac{1}{2}}) - \binom{\frac{1}{2}}{\frac{1}{2}}$
3	$[0, \frac{1}{2}) \times [0, \frac{1}{2})$
4	$triangle(\binom{\frac{1}{2}}{\frac{1}{2}}, \binom{0}{\frac{1}{2}}, \binom{0}{1}) - \binom{\frac{1}{2}}{\frac{1}{2}}$

and IM is given by

Compartment Index	Symbol	Function
2	b	$\vec{z} \leftarrow \vec{z} - \binom{1/2}{0}$
3	c	$\vec{z} \leftarrow \vec{z} + \binom{0}{1/2}$
4	d	$\vec{z} \leftarrow 2\left(\vec{z} - \binom{0}{1/2}\right)$
$1 \cup 2 \cup 3 \cup 4$	a	$\vec{z} \leftarrow \frac{1}{2}\vec{z} + \binom{1/2}{0}$

In this case, it makes sense to examine the GIFS, $S = \{triangle(\binom{1}{0}, \binom{0}{0}, \binom{0}{1}); w_1 = \frac{1}{2}\vec{z}, w_2 = \frac{1}{2}\vec{z} + \binom{\frac{1}{2}}{0}, w_3 = \frac{1}{2}\vec{z} + \binom{0}{\frac{1}{2}}\}$. The point $\binom{\frac{1}{2}}{\frac{1}{2}}$ is a cascade point S. Theorem 1 guarantees this as follows. The functions, w_i, are all one-to-one. Consider, O, the interior of $triangle(\binom{1}{0}, \binom{0}{0}, \binom{0}{1})$. Note that $w_1(O)$ is the interior of $triangle(\binom{1}{0}, \binom{0}{0}, \binom{0}{\frac{1}{2}})$, $w_2(O)$ is the interior of $triangle(\binom{1}{0}, \binom{\frac{1}{2}}{0}, \binom{\frac{1}{2}}{\frac{1}{2}})$, and $w_2(O)$ is the interior of $triangle(\binom{\frac{1}{2}}{\frac{1}{2}}, \binom{0}{\frac{1}{2}}, \binom{0}{1})$. Thus, O is a pooling set of S. Moreover, the start state of M, $\binom{1/2}{1/2}$, is in the crest of this set and hence is a cascade point of S. It is easy to check that M satisfies the definition of a PDDA: there is only one cascade so disjointness is trivial (condition (i)); the partition compartment is predictable from $\mathrm{top}_C(x)$ in every case (condition (ii)); the functions are all stack functions in this case and happen to be uniquely associated with the symbols of the alphabet: a evokes a push function, d evokes a pop function, b and c evoke pop $\circ$ push's (condition (iii)); the automaton starts at the cascade point and ends at it (condition (iv)). Thus M models a context free language.

It is also not hard to use the proof of Theorem 2 to convert M into a pushdown automaton. Well-known techniques (e.g., Hopcroft and Ullman, 1979, pp. 116-119) can then be used to convert this pushdown automaton into Grammar 1.

In all of these examples, the functions of the underlying GIFS's are contraction mappings. One may well wonder if cascades only arise in standard iterated function systems (*a la* Barnsley, 1988) in which all the functions are contraction mappings. The following case is an interesting counterexample.

Example 2.4 Consider the GIFS, $S = \{[0,1]; w_1, w_2\}$ where the functions are given by

$$w_1 = \frac{1}{2}(1 + \sqrt{1-x}) \tag{10}$$

$$w_2 = \frac{1}{2}(1 - \sqrt{1-x}) \tag{11}$$

Since w_1 maps the interval $O = (0,1)$ onto $(1/2, 1)$ and w_2 maps this interval onto $(0, 1/2)$, O is a pooling set of S. Moreover, the point $1/2$ is (in) the crest of O. Thus S can be used to record stack states over a two-symbol alphabet. However, w_1 and w_2 are not contraction mappings. For example, $w_2(0.99) - w_2(0.98) > 0.01$.

This example is interesting in part because w_1 and w_2 are the two inverses of the much studied "logistic map", $f(x) = rx(1-x)$ when $r = 4$. The logistic map has attracted attention because it is a relatively simple (one-dimensional) function with chaotic trajectories (see Stogatz, 1994, for an introduction). Crutchfield and Young (1990, 1994) analyzed f as the generator for a string $\sigma = \sigma_1\sigma_2\sigma_3\ldots$ where $\sigma_i \in \{0,1\}$ for all i by considering $f^k(1/2)$ for $k = 1, 2, 3, \ldots$ and letting $\sigma_k = 1$ if $f^k > 1/2$ and 0 otherwise. They found that when $r = 3.57\ldots$ (the so-called "onset of chaos"), the set of initial 2^n-character substrings of σ for $n \in N$ constitutes a context free language. Here, I have taken the opposite tack: inverting a closely-related map introduces an indeterminism which allows us to distinguish histories and thus use the map to model arbitrary grammars (one instance of the inverted logistic suffices for stacks alphabets with only two symbols; multiple instances can be used to accomodate more stack symbols). Loosely, one can say that while Crutchfield and Young have analyzed a chaotic map to assess the specific character of its complexity, I have described a way of using the same map to perform a general class of computations of a similarly complex sort. Given Crutchfield and Young's results, an interesting question is whether there is a canonical grammar associated with each value of r for the inverse logistic map device just described. It may be, for example, that for some languages, the corresponding inverse logistic automaton is more tolerant of imprecision in the identification of the final region than for others. I leave this as a question for future research.

3. Implementation in a neural network

Dynamical automata can be implemented in neural networks by using a combination of signaling units and gating units. By a *signaling unit*, I mean the standard sort of unit which sends out a signal reflecting its activation state to other units it is connected to. By a *gating unit*, I mean a unit which serves to block or allow transmission of a signal along a connection between two other units (see discussion in Section 0.3 above). All units (signalling and gating) compute a weighted sum of their inputs and pass this through an *activation function*—either identity or a threshold (a sigmoid can be used in place of both of these with some distortion of the computation due to the nonlinearity).

The use of simple affine functions ($\vec{z} \leftarrow q\vec{z} + r$) to define the state changes in a dynamical automaton makes for a simple translation into a network with signalling and gating units. The coefficients q and r determine weights on connections. The connections corresponding to linear terms (e.g., q) are gated connections. The connections corresponding to constant terms (e.g., r) are standard connections.

I will illustrate a neural network implementation of the dynamical automaton which generates the same set of strings as Grammar 2 (see Table 4). Grammar 2 is identical to Grammar 1 (discussed in Section 1.2 above) except that it contains the extra rule, 4c: C $\rightarrow$ a. This rule introduces some structural ambiguity in the sense that there are initial substrings for Grammar 2 (e.g. 'a b a') which can be generated by trees involving different rules. Although dynamical automata handle structural ambiguity in an analagous way to the way pushdown automata do (that is, by guessing a parse as soon as the ambiguity is encountered), and thus do not offer obvious new insights, I am illustrating an ambiguous case here in order to make it clear what the treatment of structural ambiguity looks like in this framework.

A neural implementation of Grammar 2 is shown in Figure 5. Table 5 specifies the weight values and unit types. The network processes strings by representing successive symbols as localist bit vectors on its input layer (the I units) and predicting possible successor symbols on its output layer (the O units) (Elman 1990). The units z_1 and z_2 form the core of the network. Their values at each point in time specify the coordinates of the current position on the Sierpinski triangle. They have multiple connections going

Table 4: Grammar 2.

Rule 1a.	S	$\rightarrow$	A B C D
Rule 1b.	S	$\rightarrow$	ϵ
Rule 2a.	A	$\rightarrow$	a S
Rule 2b.	A	$\rightarrow$	a
Rule 3a.	B	$\rightarrow$	b S
Rule 3b.	B	$\rightarrow$	b
Rule 4a.	C	$\rightarrow$	c S
Rule 4b.	C	$\rightarrow$	c
Rule 4c.	C	$\rightarrow$	a
Rule 5a.	D	$\rightarrow$	d S
Rule 5b.	D	$\rightarrow$	d

out of and into them. For example, z_1 has three self-connecting loops, one through gate σ_{11}, one through gate σ_{12}, and one through gate σ_{13}. These gates are enabling threshold gates in the sense that if a gate is not activated, then no signal is transmitted along the corresponding connection. The units $I_a, I_b, I_{a,c}$, and I_d are input units. I_b, $I_{a,c}$, and I_d have activation 1 when the input symbol is b, c, or d, respectively. When the input symbol is an a, then one or the other of I_a and $I_{a,c}$ takes on value 1 with equal probability (a stochastic neuron not shown here can implement this feature). Each input unit interacts with the z units in two ways: it opens one gate on the self-recurrent connections; and it transmits a weighted signal directly to each z unit. The z units take on activations equal to the weighted sum of their inputs. Although this means that in principle, their activations could be unbounded, when the net is processing grammatical strings, all of their computations take place in the bounded region $(0, 1)$—this is why it works reasonably well to use a quasi-linear (e.g. sigmoidal) activation function that is almost linear in this region. The p units are threshold units which serve to translate the z activations into binary values. The output units, O_b through O_d are threshold units which respond to the p units. Unit O_a happens to need to be on all the time in this grammar so it has no inputs and its threshold is below 0. When an output unit is on, the letter or letters corresponding to it are interpreted as possible next words. A string is deemed grammatical if, at each step of processing, the activated input unit is one of the predicted outputs from the previous step and if the network arrives at the initial state ($\vec{z} = (1/2, 1/2)$) when the last word is presented.

As indicated above, this network handles ambiguity in the same way that a push-

Figure 5: Network 1. A neural implementation of Grammar 1. Square nodes denote gating units and circular nodes denote signalling units.

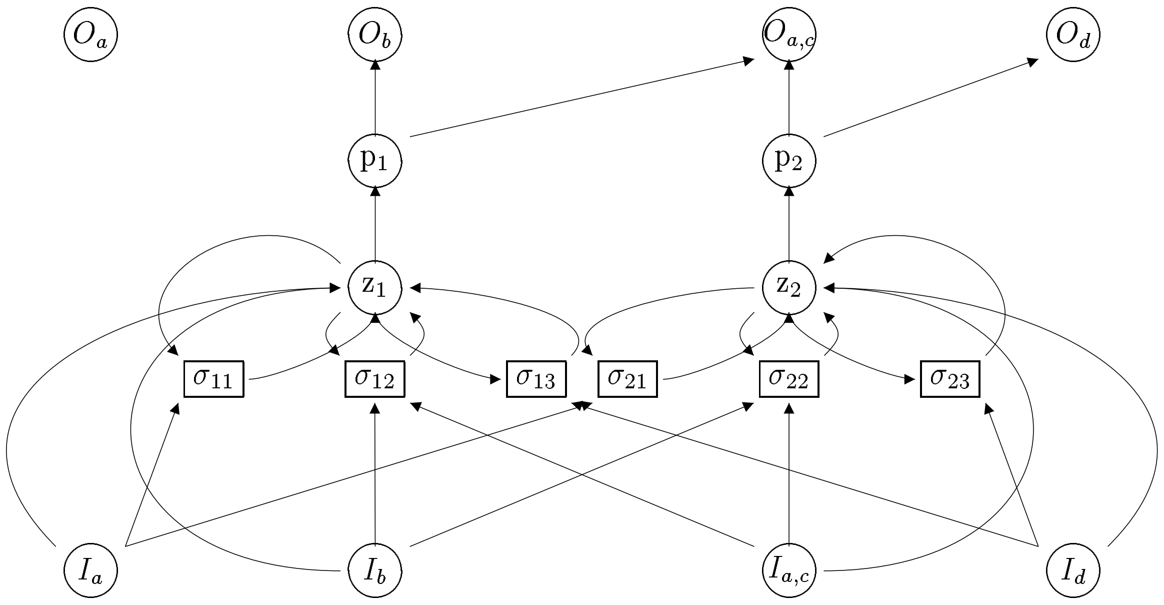


Table 5: Weights and unit types for the neural implementation of Grammar 1.

Unit	Type	Input	Weight
σ_{11}	Threshold = 1/2	I_a	1
σ_{12}	Threshold = 1/2	$I_b, I_{a,c}$	1
σ_{13}	Threshold = 1/2	I_d	1
σ_{21}	Threshold = 1/2	I_a	1
σ_{22}	Threshold = 1/2	$I_b, I_{a,c}$	1
σ_{23}	Threshold = 1/2	I_d	1
z_1	Linear	I_a	1/2
		I_b	-1/2
z_2	Linear	$I_{a,c}$	1/2
		I_d	-1
z_1	Linear	z_1 via σ_{11}	1/2
z_1	Linear	z_1 via σ_{12}	1
z_1	Linear	z_1 via σ_{13}	2
z_2	Linear	z_2 via σ_{21}	1/2
z_2	Linear	z_2 via σ_{22}	1
z_2	Linear	z_2 via σ_{23}	2
p_1	Threshold = 1/2	z_1	1
p_2	Threshold = 1/2	z_2	1
O_a	Threshold = -1	—	—
O_b	Threshold = 1/2	p_1	1
$O_{a,c}$	Threshold = $-1/2$	p_1	-1
		p_2	-1
O_d	Threshold = 1/2	p_2	1

down automaton does: by guessing. In the example at hand, since "a" is an ambiguous symbol, and the guesses are evenly distributed, the network has an equal chance of guessing wrong and right each time it encounters an "a". Thus, we consider the language generated by the network as the set of strings that it can deem grammatical, even though in a given instance, it may judge any legal string "ungrammatical". Because context often determines the proper interpretation of an ambiguous symbol (e.g. an "a" occuring initially can only be generated by rule 2b), it is possible to use additional neural machinery to constrain the choice to the contextually appropriate one. In this case, a connection from output unit $O_{a,c}$ to the stochastic neuron mentioned above can ensure that a random choice is made between I_a and $I_{a,c}$ on input "a" when $O_{a,c}$ is on and that only I_a is activated on input "a" when only O_a is on.

This section has illustrated a method of implementing dynamical automata in neural networks. The next section considers the implications of dynamical automaton structure for the study of relationships among languages of different "complexities" in the Chomsky Hierarchy sense.

4. Non context-free languages.

When a dynamical automaton is configured as a pushdown dynamical automaton (PDDA), its functions exhibit precise symmetries in the following sense: it is essential that pop operations have the effect of undoing push operations exactly. One might well wonder what happens if one adopts a more physically realistic perspective and allows the push and pop operations to be only approximate mirrors of one another.

The result can be loss of context-freeness. But the loss is not catastrophic in this case. Instead, the neighboring languages in parameter space take on what one might aptly call "mild context sensitivity".[6]

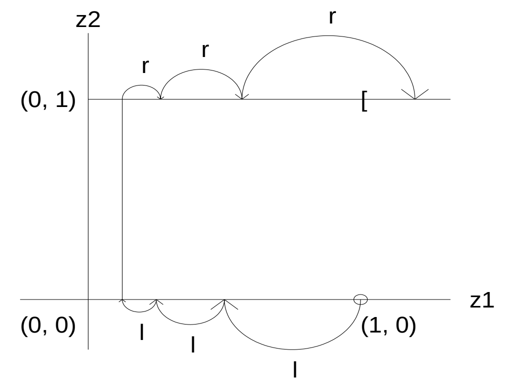
I discuss a simple case to make this point. Consider the following parameterized extension of the dynamical automaton for the language $l^n r^n$ which was discussed in Example 2.2:

$$M = (\begin{pmatrix} [0,\infty) \\ [0,1] \end{pmatrix}, \{\begin{pmatrix} m_L z_1 \\ z_2 \end{pmatrix}, \begin{pmatrix} m_R z_1 \\ z_2 + 1 \end{pmatrix}, \begin{pmatrix} m_R z_1 \\ z_2 \end{pmatrix}\}, P, \{l, r\}, IM, \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \{\begin{pmatrix} z_1 \geq 1 \\ z_2 = 1 \end{pmatrix}\}) \quad (12)$$

where m_L and $m_R > 0$, and the relevant part of P is now

[6]I thank Jordan Pollack for drawing my attention to this "precarious" quality of context-free dynamical recognizers, and thus motivating the investigation described in this section.

Figure 6: $\mathcal{M}(1/2, 17/8)$ accepting l^3r^3.



Index	Compartment
1	$\binom{1}{0}$
2	$\binom{(0,1)}{0}$
3	$\binom{(0,1)}{1}$

and IM is given by

Compartment Index	Symbol	Function
1	l	$\vec{z} \rightarrow \binom{m_L z_1}{z_2}$
2	l	$\vec{z} \rightarrow \binom{m_L z_1}{z_2}$
2	r	$\vec{z} \rightarrow \binom{m_R z_1}{z_2+1}$
3	r	$\vec{z} \rightarrow \binom{m_R z_1}{z_2}$

The scalars, m_L ("Leftward move") and m_R ("Rightward move") are parameters which can be adjusted to change the language the DA recognizes. Figure 6 illustrates the operation of this dynamical automaton. When $0 < m_L = m_R^{-1} < 1$, $\mathcal{M}$ recognizes the language $l^n r^n, n \in \{1, 2, 3, \ldots\}$.

We can analyze this automaton as follows. It recognizes strings of the form $l^n r^k$, $n \in \{1, 2, 3, \ldots\}$ where k is the smallest integer satisfying

$$m_L^n m_R^k \geq 1$$

Since we are only considering cases where $m_L > 0$,

Table 6: A context free grammar for generating $l^n r^{2n}$.

S	$\rightarrow$	l r r
S	$\rightarrow$	l S r r

$$m_R^k \geq m_L^{-n}$$

$$k \geq log_{m_R}\ m_L^{-n}$$

$$k = [[-n\ log_{m_R}\ m_L]]$$

where $[[x]]$ denotes the smallest integer greater than or equal to x.

If m_L is a negative integer power of m_R, then the language of M can be described with a particularly simple context-free grammar. Example 4.1 illustrates.

Example 4.1. Let $m_L = \frac{1}{4}$ and $m_R = 2$. Thus $k = [[-n\ log_2\ \frac{1}{4}]] = 2n$. The language is thus $l^n r^{2n}$. This language is generated by a context free grammar with only two rules (Table 6).

If m_L is a non-whole-number rational power of m_R (or vice versa), then the resulting language is still context free, but its grammar is more complicated. Example 4.2 illustrates a case like this.

Example 4.2. Let, $m_L = \frac{1}{4}$ and $m_R = 4^{\frac{5}{6}} \approx 3.174802$. Thus $k = [[1.2n]]$. The language recognized by this particular parameterization of M is thus $\mathcal{A} = l^n r^{[[1.2n]]}$. This language is substantially more complicated than the language of the previous example. It requires seven rules in context free grammar format (Table 7). The number of rules grows with the length of the cycle of the coefficient of n.[7]

If m_L is an irrational power of m_R or vice-versa, then M generates a non context-free language. We can show this using the Pumping Lemma for Context Free Languages (Hopcroft and Ullman, 1979, p. 125).

Proposition. The language $\mathcal{A} = l^n r^{[[qn]]}$ for q irrational is not a context free language.

Pf: I proceed by showing that if $\mathcal{A} = l^n r^{[[qn]]}$ is a context free language, then q is rational.

The Pumping Lemma for Context Free Languages says that if $\mathcal{A}$ is a context free language, then there is a non-negative integer n such that any string $\sigma \in \mathcal{A}$ whose length is greater than n can be written $\sigma = uvwxy$ in such a way that

[7]By the *cycle* of a real number q, I mean the smallest positive integer, p such that pq is an integer.

Table 7: A context free grammar for generating $l^n r^{[[1.2n]]}$.

S	$\rightarrow$	l S$_s$ r^2
S	$\rightarrow$	l^2 S$_s$ r^3
S	$\rightarrow$	l^3 S$_s$ r^4
S	$\rightarrow$	l^4 S$_s$ r^5
S	$\rightarrow$	l^5 S$_s$ r^6
S$_s$	$\rightarrow$	ϵ
S$_s$	$\rightarrow$	l^5 S$_s$ r^6

(i) $\mid vx \mid \geq 1$

(ii) $\mid vwx \mid \leq n$

(iii) for all $i \geq 0$, uv^iwx^iy is in $\mathcal{A}$.

Suppose $\mathcal{A}$ satisfies the Pumping Lemma for $n > n_0$. Consider a string $uvwxy$ which can be pumped in accord with condition (iii). Clearly, v must consist of a positive number of l's and x must consist of a positive number of r's. Let $\mid v \mid = c_l$ and $\mid x \mid = c_r$. Without loss of generality, we can assume that v is rightmost in the string of initial l's so $\mid w \mid = 0$. Let $\mid u \mid = d_l$ and $\mid y \mid = d_r$. Then, by the definition of $\mathcal{A}$ we can write

$$c_r i + d_r = [[q(c_l i + d_l)]] \tag{13}$$

For each $i \in N$, let δ_i be the fractional part of $q(c_l i + d_l)$. Then, by (13), we can write

$$q = \frac{c_r(i+1) + d_r + \delta_{i+1} - (c_r i + d_r + \delta_i)}{c_l(i+1) + d_l - (c_l i + d_l)} = \frac{c_r + \delta_{i+1} - \delta_i}{c_l} \tag{14}$$

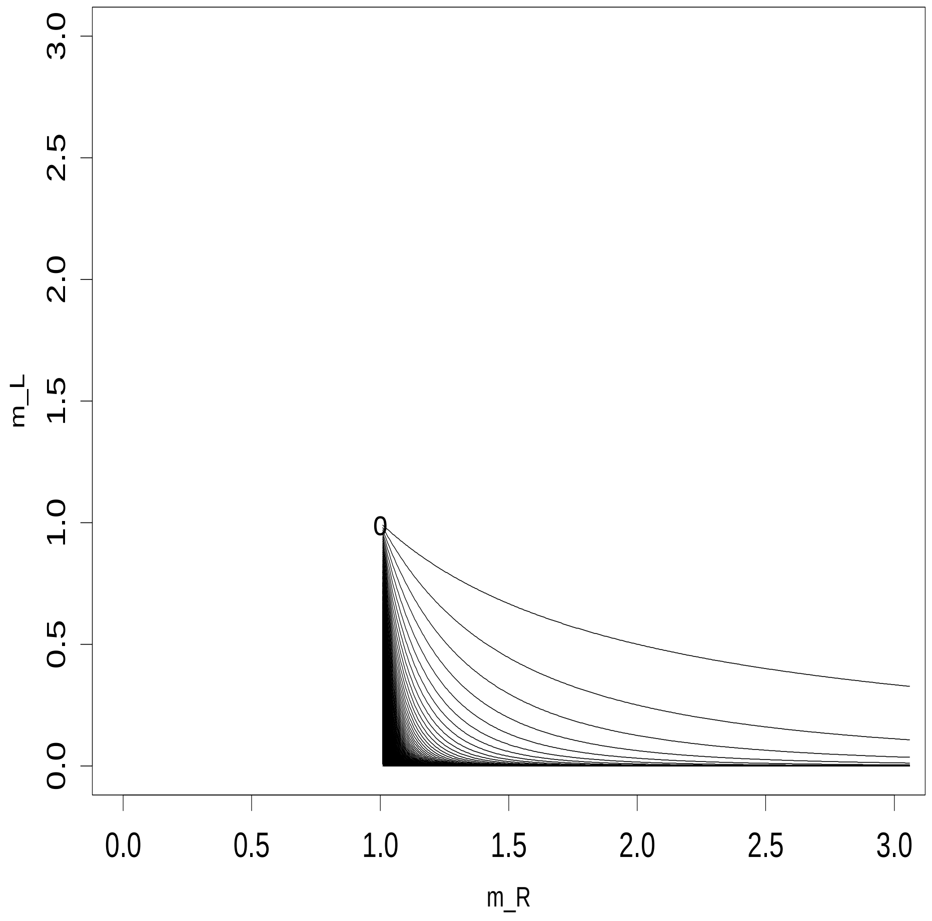
But unless $\delta_{i+1} = \delta_i$ for all i, equation (13) is false for sufficiently large i. Therefore,

$$q = \frac{c_r}{c_l} \tag{15}$$

Since c_r and c_l are integers, q is rational. □

These examples show that even one of the simplest parameterized dynamical automata can emulate computing devices with a significant range of complexities. The framework also suggests an interesting new way of examining the relationships between formal languages: we can look at their locations in the parameter space of the dynamical automaton. Figure 7 shows how the simplest (two-rule) context-free languages among $a^n b^{[[-n \; log_{m_R} \; m_L]]}$ are distributed in the first quadrant of $< m_R \times m_L >$. By adopting a

Figure 7: The bands in the space $m_L \times m_R$ where the simplest (two-rule) context free languages reside. (Example 4.2).



natural metric on this space (e.g. Euclidean distance), we can talk about relationships between languages in terms of distance. On this view, each two-rule language is surrounded by more complex languages (both in the rule-counting sense and the Chomsky hierarchy sense). Although the grammars of the languages near each two-rule language are substantially different from its grammar in possessing large numbers of rules or requiring context-sensitive rules, their distributional properties are rather similar. For example, the language $l^n r^{[[1.01n]]}$ differs from $l^n r^n$ only in very long strings. If we assume that unbiased probabilities are associated with those rows in the Input Mapping of the dynamical automaton which specify transitions out of the same compartment, then the strings on which these two languages differ are quite rare. In Section 5, I speculate how this property may be useful in designing learning algorithms for dynamical automata.

5. Conclusions.

5.1 Review

I have examined a particular type of computing device, the *dynamical automaton*, whose variables and parameters take on real values. In contrast to other work on real-valued automata which focuses on complexity and tractability issues, I have emphasized the interpretation of real-valued spaces as metric spaces and examined consequences for our understanding of how computing devices are related to one another. First, I identified a class of computing devices called pushdown dynamical automata whose computation is organized around fractals, and showed that this class corresponded to the class of context free grammars. I illustrated a simple method of implementing dynamical automata in neural networks. Then I examined a simple dynamical automaton with real-valued parameters and showed how this automaton behaved like various pushdown automata under some parameter settings and like more powerful automata under other settings. Moreover, the different automata were organized in the metric space of the computer's parameters in such a way that nearby automata in the parameter space generated similar sets of strings (in a probabilistic sense). The context free grammars with the fewest rules occupied disconnected regions of the parameter space. In between these were grammars with more rules, including many non-context-free grammars.

5.1 Comparison with Moore's method

Moore (to appear), studying closely related dynamical recognizers, shows that some dynamical recognizer with piecewise linear transition and decision functions can recognize any context free language. His result is quite similar to the second half of my Theorem 2 but the two approaches both have strengths and weaknesses so it is useful to compare them.

Recall that a dynamical recognizer is like a dynamical automaton except that in a dynamical recognizer the choice of function applied when a given symbol appears is not contingent on the current state and acceptance or rejection is only evaluated when the last symbol of a string has been processed.

Moore uses the following functions to construct a dynamical recognizer for any context free grammar:

Name	Function
push$_i$	$\alpha x + (1-\alpha)\frac{i}{m}$
pop$_i$	$\frac{1}{\alpha}\left(x - (1-\alpha)\frac{i}{m}\right) = push_i^{-1}(x)$

Here, $1 \leq i \leq m$ and $0 < \alpha \leq \frac{1}{3m+1}$. These $2m$ functions, which form a GIFS on R, can be used to simulate a pushdown stack with m symbols. The functions "push$_i$" and "pop$_i$" correspond to "push symbol i onto the stack" and "pop symbol i off of the stack" respectively. It can be shown that pushdown automata generate the same languages as the subset of them in which there are no control state changes (Hopcroft and Ullman, 1979). Thus all context free languages can be generated by a device which consists simply of a pushdown stack. Therefore, by composing push$_i$'s and pop$_i$'s appropriately, Moore's functions can be used to build a mechanism for generating any context-free language.

There are two technical details that need to be looked after. First, how can Moore's recognizer detect an illegal pop, i.e., a move of the form pop$_j\cdot$ push$_i$ where $i \neq j$? Second, what guarantees that each sequence of pushes and pops is uniquely associated with a point in R? Moore answers the first question but does not address the second. Conveniently, Theorem 1 can be applied to address the second.

Regarding the first question (illegal pops), Moore's mechanism is cleverly designed to allow easy detection of illegal pops: any move or partial move of the illegal form pop$_j\cdot$ push$_i$ makes $\mid x \mid \geq 2$ but all legal moves and partial moves keep $\mid x \mid \leq 1$. In order to allow his device to record the information about whether $\mid x \mid$ ever exceeded 2 so that it is available when the string as a whole has been processed, Moore introduces a new variable, y with initial value 0 and updates y according to

$$y \leftarrow f(x,y) = max(\mid x \mid, \mid y \mid)$$

Requiring, that $y \in [0,1]$ at the end of processing a string ensures that all moves have performed pushdown stack operations.[8] For comparison, note that a pushdown dynamical automaton ensures that pop moves always undo push moves by choosing a partition

[8]The fact that push$_i$, pop$_i$ and f are all at least piecewise linear in x and y forms the basis of Moore's claim that a dynamical recognizer with all piecewise linear operations can recognize any context free language.

in which the current compartment is always uniquely associated with the most recent push move and allowing only the corresponding pop move out of that compartment.

The second question, What guarantees that each sequence of pushes and pops is uniquely associated with a point in R?, can be addressed by choosing an appropriate starting value, using Theorem 1 above as a guide. In fact, not all starting values will work. Consider the case $m = 3$ and $\alpha = 1/10$. If the starting value is $x_0 = 1$, then it is easy to check that the stack states "ϵ", "3", "33", "333", etc. all generate the same dynamical recognizer state, namely $x = 1$. This means the device will fail to distinguish these stack states from each other. To avoid this problem, it suffices to note that [0, 1] is a pooling set for the GIFS of pop$_i$'s and push$_i$'s and the set $[0, 3/10) \cup (4/10, 6/10) \cup (7/10, 9/10)$ is its crest. Thus any x_0 in this set will give rise to unique locations for all stack states.

5.2 Avenues worth exploring

Several interesting questions are raised by the results described here.

Fleshing-out of Chomsky Hierarchy Relationships. The current proposal to organize languages in a metric space seems, at first glance, to differ substantially from standard complexity-based approaches. However, a close look at the example of Section 4 suggests that the relationship between the current results and the complexity-based results may be one of augmentation rather than revision. In particular, the overarching contrast between context free languages and non-context-free languages is preserved in this metric space example as a contrast between machines with a rational versus an irrational parameter. Two further questions are also worth exploring: (i) Is there a natural way, in the dynamical automaton framework, of defining precisely the recursively enumerable languages, or of defining a set of recursively enumerable languages which contains a set of context free languages as a proper subset? (ii) Do all dynamical automata map standard complexity classes onto independently motivated parameter classes?

Approximations of Infinite Machines. Just as the irrational numbers can be viewed as the limits of infinite series of rationals, so the non context-free devices in the model of Section 4 can be viewed as limits of infinite series of context-free devices. A similar idea has been explored by Crutchfield and Young (1990) and Crutchfield (1994). These authors analyze a particular *indexed context free grammar* as the limit of an infinite series of increasingly complex finite-state devices. They project from their results an approach to signal analysis in which one studies the growth in size of successive machine approximations at one level on the Chomsky Hierarchy in order to find out if a jump to a machine at a higher level is warranted. However, they only explore the case in which bigger and bigger finite state devices approximate a context free device. The current results may thus be useful in extending their method to transitions between higher levels.

Finding all CFLs. The cascade-based analysis makes it possible to identify certain dynamical automata in a parameterized dynamical automaton family which generate context free languages. However, it doesn't necessarily characterize all context free language generators in a particular family. A case in point is the generator described in Section 4: only the simplest two-rule grammars follow a single cascade throughout their computations. It would be desirable to generalize the analysis so that one could identify the entire set of context-free language generators in a given dynamical computing system.

Universality. I have been promoting parameterized dynamical automata as a better way of organizing formal languages than various systems stemming from the Chomsky hierarchy. There is one sense in which the Chomsky hierarchy is more appealing as an organizational tool: it is nearly machine-independent, and thus very universal. The dynamical automaton spaces I define depend on the particular parameterized automaton under study, and do not provide a way of organizing all languages that can be generated using a finite alphabet. Nevertheless, it is interesting to consider the possibility that there might be a privileged set of functions which serves as the basis for a general automaton space to which all dynamical automata can be related. Such a framework might be useful for studying processes which involve incremental search over a very wide range of devices—e.g., evolution, signal identification.

Neural Networks. Certain kinds of neural networks are real-valued computers. As computing mechanisms, neural networks are interesting because they are closely associated with general theories of learning, and because they are sensitive to statistical properties of their environment and thus can operate well even in a noisy world. Despite their appealing properties, however, it is hard to get neural networks to learn complex functions like those above the finite state level on the Chomsky hierarchy. Although we know how to program neural networks to implement Turing machines as well as other, less powerful symbolic devices (e.g., Siegelmann and Sonntag, 1991; Kremer, 1996), the implementations have not particularly taken advantage of the special strengths of neural representations. Instead they have largely been remakes of computers we already know how to build using standard machinery. The current work offers a new line on this problem. As shown in Section 3, dynamical automata are easily implemented in certain kind of recurrent neural networks. Because the implementation is based on the metric structure of the representation space and metric relationships are fundamental to neural network learning and statistical sensitivity, the proposed representation may be an especially useful one for improving the performance of learning networks on hard symbolic problems. I develop this point more in the next paragraph.

Learning. A natural approach to language learning is to think of it as a process of making small adjustments in a grammar in order to improve predictive accuracy. This approach requires us to define what constitutes a "small adjustment", i.e., to define similarity among grammars. Using standard symbolic formalisms, it is hard to choose among the myriad ways one might go about defining similarity among grammars, especially if the grammars are infinite-state devices. We can examine the number of rules that two grammars have in common. But then the rules that are not shared between the two grammars can have so many forms that it is hard to know how to take them into account. We can assign probabilities to rules and compare the implications for local symbol transition likelihoods. But it is hard to know how to compare between the case in which a rule belongs to a grammar but occurs with very low probability and the case in which the rule is simply absent from the grammar. The hypothesis spaces defined by parameterized dynamical automata (as in the example described in Section 4) look promising in this regard. The automata are organized in a continuum in such a way that nearby automata give rise to similar transition behaviors (in a probabilistic sense, as mentioned at the end of Section 4). Thus it may be possible to use dynamical automata as the basis for a gradient descent search without having to make an arbitrary decision about rule cost. Instead of trying to postulate an arbitrary balance between complexity of a grammar and coverage of the data, the learning algorithm can simply search the space for the best-fit automaton.

Language Typology. The metric organization of dynamical automata in their parameter spaces places quite different machines next to each other. The analysis in Section 4 shows that, for a simple parameterized Dynamical Automaton, the computational class of a specific parameterization depends upon the rationality of one parameter: rational values yield context free languages; irrational values yield non-context-free languages. Since rational and irrational numbers can be arbitrarily close to one another, the Chomsky Hierarchy status of this dynamical automaton fluctuates dramatically as this parameter value is adjusted. As a cognitive model, such an organization system works against the notion that Chomsky hierarchy complexity makes essential distinctions between languages, but it may lead to a more useful way of accounting for the appearance of a mixture of context-free and context-sensitive patterns among the world's languages (e.g., Shieber, 1985).

5.3 Comment on relevance

Grammars and metric spaces seem like strange bedfellows. But the switch in focus from classification (complexity hierarchies) to location (metric space organization) may make working with grammars more practical by allowing us to deal with approximations and to get a handle on learning. Moreover, if we focus on grammatical representation in real-valued machines we can understand the mechanisms of their computations better than if we focus on their computational power alone.

References

Barnsley, M. (1988). *Fractals Everywhere.* Academic Press, Boston.

Bishop, C. (1995). *Neural Networks for Pattern Recognition.* Clarendon Press, Oxford.

Blair, A. and Pollack, J. B. (to appear). Analysis of dynamical recognizers. To appear in *Neural Computation.*

Crutchfield, J. P. (1994). The calculi of emergence: Computation, dynamics, and induction. *Physica D*, 75:11–54. In the special issue on the Proceedings of the Oji International Seminar, *Complex Systems—from Complex Dynamics to Artificial Reality.*

Crutchfield, J. P. and Young, K. (1990). Computation at the onset of chaos. In *Complexity, Entropy, and the Physics of Information*, pages 223–70. Addison-Wesley, Redwood City, California.

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

Elman, J. L. (1991). Distributed representations, simple recurrent networks, and grammatical structure. *Machine Learning*, 7:195–225.

Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Menlo Park, California.

Kremer, S. C. (1996). A theory of grammatical induction in the connectionist paradigm. PhD Thesis, Department of Computing Science, Edmonton, Alberta.

Minsky, M. and Papert, J. (1988[1969]). *Perceptrons: an introduction to computational geometry.* MIT Press, Cambridge, MA.

Moore, C. (1996). Dynamical recognizers: Real-time language recognition by analog computers. TR No. 96-05-023, Santa Fe Institute.

Moore, C. (To appear). Dynamical recognizers: Real-time language recognition by analog computers. *Theoretical Computer Science.*

Plate, T. A. (1994). Distributed representations and nested compositional structure. Ph.D. Thesis, Computer Science, University of Toronto.

Plate, T. A. (1995). Holographic reduced representations. *IEEE Transactions on Neural Networks*, 6(3):623–641.

Pollack, J. B. (1987). On connectionist models of natural language processing. Ph.D. Thesis, Department of Computer Science, University of Illinois.

Pollack, J. B. (1990). Recursive distributed representations. *Artificial Intelligence*, 46:77–106. Special issue on Connectionist symbol processing edited by G. E. Hinton.

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

Prince, A. and Smolensky, P. (1993). *Optimality Theory: Constraint Interaction in Generative Grammar.* MIT Press, Cambridge, MA.

Rodriguez, P. (1995). Representing the structure of a simple context-free language in a recurrent neural network: A dynamical systems approach. On-line Newsletter of the Center for Research on Language, University of California, San Diego.

Rodriguez, P., Wiles, J., and Elman, J. (ta). How a recurrent neural network learns to count. *Connection Science.*

Rumelhart, D., Durbin, R., Golden, R., and Chauvin, Y. (1995). Backpropagation: The basic theory. In *Backpropagation: Theory, Architectures, and Applications.* Lawrence Erlbaum Associates.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning internal representations by error propagation. In Rumelhart, D. E., McClelland, J. L., and the PDP Research Group, editors, *Parallel Distributed Processing, Volume I*, pages 318–362. MIT Press.

Shieber, S. M. (1985). Evidence against the context-freeness of natural language. *Linguistics and Philosophy*, 8. Also in Savitch, W. J., et al. (eds.) *The Formal Complexity of Natural Language*, pp. 320–34.

Siegelmann, H. T. and Sontag, E. D. (1991). Turing computability with neural nets. *Applied Mathematics Letters*, 4(6):77–80.

Smolensky, P. (1988). On the proper treatment of connectionism. *B.B.S.*, 11(1):1–74.

Smolensky, P. (1990). Tensor product variable binding and the representation of symbolic structures in connectionist systems. *Artificial Intelligence*, 46. Special issue on Connectionist symbol processing edited by G. E. Hinton.

Smolensky, P., Legendre, G., and Miyata, Y. (1992). Principles for an intergrated connectionist/symbolic theory of higher cognition. Ms., Department of Computer Science, University of Colorado at Boulder.

Strogatz, S. (1994). *Nonlinear Dynamics and Chaos.* Addison-Wesley, Reading, MA.

Sun, G. Z., Chen, H. H., Giles, C. L., Lee, Y. C., and Chen, D. (1990a). Connectionist pushdown automata that learn context-free grammars. In Caudill, M., editor, *Proceedings of the International Joint Conference on Neural Networks*, pages 577–580. Lawrence Earlbaum, Hillsdale, NJ.

Sun, G. Z., Chen, H. H., Giles, C. L., Lee, Y. C., and Chen, D. (1990b). Neural networks with external memory stack that learn context-free grammars from examples. In *Proceedings of the 1990 Conference on Information Sciences and Systems*, volume 2, pages 649–53.

Tabor, W. (submitted-a). Metrical relations among analog computers. Draft version available at http://www.cs.cornell.edu/home/tabor/papers.html.

Tabor, W. (submitted-b). Context free grammar representation in neural networks. Draft version available at http://simon.cs.cornell.edu/home/tabor/papers.html.

Tomita, M. (1982). Dynamic construction of finite-state automata from examples uising hill-climbing. In *Proceedings of the Fourth Annual Cognitive Science Conference*, pages 105–108. Ann Arbor, Michigan.

Vapnik, V. (1995). *The Nature of Statistical Learning Theory.* Springer, New York.

Werbos, P. J. (1974). Beyond regression: New tools for prediction and anlysis in the behavioral sciences. Ph.D. Thesis, Harvard University, Cambridge, MA.

Wiles, J. and Elman, J. (1995). Landscapes in recurrent networks. In Moore, J. D. and Lehman, J. F., editors, *Proceedings of the 17th Annual Cognitive Science Conference*. Lawrence Erlbaum Associates.