Infinite RAAM: Initial Explorations into a Fractal Basis for Cognition

A Dissertation

Presented to

The Faculty of the Graduate School of Arts and Sciences

Brandeis University

Department of Computer Science

Jordan B. Pollack, Advisor

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

by

Simon D. Levy

May, 2002

This dissertation, directed and approved by Simon D. Levy's committee, has been accepted and approved by the Graduate Faculty of Brandeis University in partial fulfillment of the requirements for the degree of:

DOCTOR OF PHILOSOPHY

Dean of Arts and Sciences

Dissertation Committee:

Jordan B. Pollack, Dept of Computer Science, Chair.

Harry Mairson, Computer Science

James Pustejovsky, Computer Science

George Berg, State University of New York at Albany

Acknowledgments

First and foremost, I would like to thank the members of my thesis committee: George Berg, Harry Mairson, Jordan Pollack, and James Pustejovsky, whose willingness to wade through the sometimes abstruse material herein made it possible for me to reach this milestone. Throughout my years at Brandeis, Jordan has been a constant source of inspiration and challenge. Life as his Ph.D. student was by turns exhilarating and frustrating, but never boring, and I will always be deeply grateful. Harry and James have provided mentorship, wisdom, and an example to follow. George agreed to serve on my committee with very little notice; his detailed comments and insights are greatly appreciated.

Not the least of Jordan's many talents is his uncanny ability to find outstanding students to do the cutting-edge work for which his DEMO Lab is now justifiably famous. With Anthony Bucci, Paul Darwen, Edwin de Jong, Sevan Ficici, Pablo Funes, Greg Hornby, Kristian Kime, Hod Lipson, Ofer Melnik, John Rieffel, Betsy Sklar, Shiva Viswanathan, and Richard Watson I have shared years of friendship and scholarly camaraderie, which will remain as one of the high points of my life. I would especially like to thank Ofer, whose mathematical analyses enabled much of the work in this thesis, and Anthony, whose insight (and Java wizardry) have contributed greatly to advancing the RAAM project.

One benefit of an unusually long graduate education has been the chance to meet and befriend some truly extraordinary people. With friends like Tom Bell, Margaret Dunn, Steve Jilcott, Zach Mason, Max More, Phil Mucci, Bill Stubbs, and Steve Witham, I have grown as a scholar and more importantly as a human being, and without their support and companionship would undoubtably have failed to reach the point at which I now find myself.

Phil Rubin, Elliot Saltzman, Mark Tiede, and the Haskins Interesting Stuff Group provided friendship and sparked my interest in connectionism and dynamical systems. Sensei Joe Arvidson and the members of the Newburyport Black Dragon Dojo enabled me to learn karate while pursuing my Ph.D., imparting lifelong skills and confidence for which I can never adequately repay them. Ken Lambert and the CS faculty, administration, and students at Washington and Lee University have given me an incredible opportunity for teaching and scholarship, fulfilling the dream of a lifetime.

Finally, and most profoundly, I have no words to express my gratitude for the love and support of my parents and my sister Sharon, through what must have seemed to them an interminable postgraduate career. They, above all, have made me who I am, and are therefore the most to be thanked for whatever contribution I make with this work, and all work to come.

ABSTRACT

Infinite RAAM: Initial Explorations into a Fractal Basis for Cognition

A dissertation presented to the Faculty of
the Graduate School of Arts and Sciences of
Brandeis University, Waltham, Massachusetts

by Simon D. Levy

This thesis attempts to provide an answer to the question "What is the mathematical basis of cognitive representations?" The answer we present is a novel connectionist framework called Infinite RAAM. We show how this framework satisfies the cognitive requirements of systematicity, compositionality, and scalable representational capacity, while also exhibiting "natural" properties like learnability, generalization, and inductive bias.

We begin with the requirements for cognitive representations, including traditional criticisms about the inability of standard connectionist models to satisfy these requirements. We then review some recent connectionist approaches and their limitations, and describe Pollack's Recursive Auto-Associative Memory (RAAM) a recurrent neural network model of hierarchical symbolic structure that addresses some of these limitations successfully. Scalability problems with RAAM lead us to an exploration of the network's behavior as an iterated function system (IFS), focussed on the concept of the attractor, which describes the behavior of the IFS in the limit. We show how exploiting the attractor overcomes the practical limitations of RAAM, leading to a model with provably infinite representational capacity, which we call Infinite RAAM, or IRAAM.

The subsequent chapters detail how the model can represent trees over a potentially infinite lexicon of terminals; the "bias" that the model exhibits in the relationship between these terminals and the tree structures; and the direct mechanism that the model provides for the unification algorithm. We then present experiments on IRAAM learning, followed by some applications of IRAAM to logic programming and language representation. We conclude with a discussion of the implications and current limitations of the model, with prospects for overcoming these limitations in future work.

The contributions of this work are twofold: First, Infinite RAAM shows how connectionist models can exhibit infinite competence for interesting cognitive domains like language. Second, our attractor-based learning algorithm provides a way of learning structured cognitive representations, with robust decoding and generalization. Both results come from allowing the dynamics of the network to devise emergent representations during learning.

An appendix provides Matlab code for the experiments described in the thesis.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 An adequate substrate for symbolic computation

Melnik [64] provides the following key insight about neural networks and computational paradigms in general:

> Even though neural networks are an example of a Turing equivalent computational system [68]..., in the real world, this theoretical equivalence is of only minor significance, as different computational paradigms may be suited for completely different tasks. What fundamentally distinguishes what a computational paradigm is suited for, is how it represents information and how it manipulates it. Thus, analyzing and characterizing the types of information that neural networks can encode efficiently is at the core of understanding neural computation.

Hence, any approach to a computational problem, such as language, must concern itself with efficient representation of the computational structures best suited to that problem, and efficient implementation of operations on those structures.

For symbolic computation, the representations and operations of interest include:

1. ***Atoms,*** or irreducible symbols. These correspond to dictionary items in a natural language, letters of the alphabet in a formal language, predicates and atomic arguments

in predicate calculus, tokens in a programming language, etc.

2. ***Structures*** built recursively from those atoms and from other structures. These correspond to phrases and sentences in a natural language, strings (or trees) in a formal language, literals in predicate calculus, programs in a programming language, etc.

3. ***Rules*** that relate structures to each other. Examples include the syntactic transformations of transformational generative grammar [13], like passive (*John loves Mary* $\Rightarrow$ *Mary is loved by John*), and quantified rules in predicate calculus ($\forall\ x\ human(x) \rightarrow\ mortal(x)$).

Atoms and structures support the requirements that Fodor [33] lays out for cognitive representations in his "language of thought". *Systematicity* dictates that the representation of an object in a language must be the same in all contexts which refer that object. Thus the representation of the person John must be identical in *John loves Mary, Mary loves John,* etc. *Compositionality* requires that representations be built up from other representations. The sentence *John loves Mary* must in some meaningful sense contain the representations for John, Mary, and his feelings for her.

Compositionality leads to the requirement that representations be *recursive*; that is, representations must be able contain other representations. Since any limit on the size of such representations (e.g., "no more than twelve embeddings") are necessarily arbitrary, argues Fodor, the fact that real human thoughts and linguistic utterances are generally not deeply nested must come from system-external constraints such as limits on short-term memory.[1] As Fodor points out, the exact structure of this language is of little concern to the main argument that the language must be systematic, compositional, and recursive.

[1] or, in a digital computer, the size of the push-down stack. Chomsky [12]has made similar arguments for the infinite generative potential of natural language.

1.2 Connectionism and its discontents

As a radical alternative to this "symbols-and-rules" approach, the connectionist field has offered a broad variety of cognitive models having in common the strategy of using (roughly) neuron-like units and the connections between them as a way of modeling systems like reasoning and language. The motivations for using these neural networks[2] range from the very practical (it is difficult to model many complicated systems using even a large number of symbols and rules) to the deeply philosophical (this is more like how the brain works, so it is closer to the truth).

Though it has roots going far back into the Western philosophical tradition [19], connectionism's most complete formulation to date came in the mid 1980's, with the publication of a collection of papers summarizing the ideas and work of the Parallel Distributed Processing (PDP) Research Group at USCD, Carnegie Mellon, and other institutions. [88]. One paper in particular involved a network that the authors claimed was able to "learn" the past tense of regular and irregular English verbs in a way similar to the manner in which children acquire those verbs, including an initial pure memorization phase, an intermediate phase involving overgeneralization of regularity, and a final, stable phase approximating the behavior of the linguistically competent adult. [87]. This result suggested the possibility of abandoning linguistic rules in favor of a "sub-symbolic" network approach, in conjunction with a learning mechanism such as back-propagation [86].

This work was criticized from a methodological standpoint as mischaracterizing what the network was actually doing, the conclusion being that traditional linguistic rules must still play a central role in modeling the acquisition and use of language[69]. Furthermore, the entire connectionist endeavor represented by the PDP group was called into question as being merely associationist, and therefore not adequate to the tasks of syntactic and semantic representation involved in human cognition as a whole. [36]

[2]We use the terms *neural* and *connectionist* interchangeably in referring to these networks.

The problem boils down to two facts about neural networks that at first glance seem fundamentally incompatible with important properties of language. First, connectionist representations (at least, those of the sort promoted by the PDP group) are *distributed.* The representation of a given entity or concept is not localized in some particular piece of memory, but is instead spread over the entire network, or a large part of it, somewhat similarly to the way that the identity of a number is distributed over its binary representation. This non-locality severely constrains the traditional computer-science use of pointers to refer to particular locations in memory, making the implementation of data structures a seeming impossibility.

Second, neural networks encode information in *fixed-width* representations. This property has led to the rather embarrassing situation of networks that can produce or accept, for example, sentences of only a certain length, with the number of units required growing polynomially in the number of words. Although people seem to be constrained in the size of the sentences they can utter and understand, such human limitations do not appear to be an all-or-nothing affair; instead, ability seems to fall off gradually with the size of the informational load. Given the connectionist argument for such "graceful degradation" as a crucial, appealing property of neural network models [62], the fixed-width limitation seems especially problematic.

The second problem has been addressed by a host of connectionists who have over the past fifteen years investigated network models which support generativity through recurrent (feedback) connections [29][50][113]. As in finite-state automata [44], such connections enable a neural network to travel through an unbounded number of states, and hence to recognize and generate strings of arbitrary length. Not surprisingly, then, one focus of recurrent-net research has been on the formal properties of the languages (sets of strings) that these networks can handle [83]. Nevertheless, despite analysis of how a network's dynamics contribute to its generativity, it is often uncertain whether the dynamics can support generation of well-formed strings beyond a certain length. That is, it is unknown whether the network has a true "competence" for the language of which it has learned a few exemplars, or is merely capable of generating a finite, and hence regular, subset of the language.

The first problem, that of representing structured information, has also received attention in the connectionist community, most notably in the guise of the SHRUTI project of Shastri [95][96], who is interested in answering the same fundamental question addressed by this thesis:

> How can a system of slow neuron-like elements represent a large body of knowledge and perform a wide range of inferences with such speed?

SHRUTI attempts to answer this question with a network architecture that essentially encodes the traditional role/filler models of classical AI and cognitive science [92][32]. SHRUTI combines this kind of model with the usual neural-net weighted connections and continuous-valued activation functions, resulting in a system in which the truth-value of propositions (e.g, "John gave Mary a book") is a matter of degree, rather than an absolute true/false distinction. This kind of "soft computing" is a standard feature of connectionist models; the unique contribution of SHRUTI is to represent the bindings of fillers to their roles (GIVER = *John*) via synchronous temporal firing of the nodes representing the roles and fillers, which the authors cite as consistent with evidence from the neurophysiological literature[27].

Although SHRUTI does successfully address the issues of systematicity and compositionality mentioned earlier, fundamental features of the model cast doubt on its ability to satisfy the requirements of a truly connectionist solution to the language-of-thought problem. First, representations in SHRUTI are extremely *local*: each role is represented by a single node, as is each filler, and each relation (e.g. GIVE) is represented as a "focal cluster" of such nodes. Although the authors argue that there is no inherent reason why a whole set of nodes could not be dedicated to to single item, this sort of representation is not truly distributed in any meaningful sense. The actual mechanism by which SHRUTI performs inferences is potentially highly parallel, via breadth-first traversal of an "inferential dependency graph"; however, this kind of parallelism is also available to traditional localist graph representations as well. Considered as a Parallel Distributed Representation, SHRUTI thus seems parallel but not distributed.

Second, SHRUTI's inference mechanism is limited in its depth of recursion. The authors present this feature as a psychologically motivated and therefore desirable feature of the model, putting them clearly in the ranks of cognitive scientists who do not see the language of thought as potentially infinite. The literature contains well-reasoned arguments on both sides of this issue[3], and contributing to the discussion is not an aim of this thesis. Nevertheless, it seems that a convincing response to connectionism's critics would involve a model that was at least in principle capable of infinite representations.

1.3 Taking the anti-connectionists seriously

The story so far[4] leaves us looking for a connectionist model that is both truly compositional, *i.e.*, capable in principle of representing infinite recursive structure, as well as distributed, that is, not based on a "grandmother cells" scheme in which a single unit or local cluster of units represents a single entity or idea. Three such models found in the connectionist literature are the BoltzCONS model of Touretzky, the Tensor Product model of Smolensky, and the Holographic Reduced Representations model of Plate. A fourth model, Infinite RAAM, is the subject of this thesis.

[3]With respect to natural language, Rounds et al. [84] argue for "finite approximations to an infinite language," whereas Savitch [91], on the basis of Occam's Razor, upholds an "essentially infinite" view . From a programming languages perspective, the later approach certainly makes more sense: although the storage of any physical machine is necessarily finite, nearly all modern programming languages are built using some variant of context-free grammars and make liberal use of "essentially infinite" recursive data structures, vastly simplifying the work of the programmer.

[4]This brief review is not meant to be a complete encapsulation of the history of connectionism, of which several excellent summaries already exist, such as [72]. Rather, my aim here is to highlight some milestones in the issue of representing compositional (linguistic) structure in a connectionist framework.

1.3.1 BoltzCONS

Touretzky's BoltzCONS [109] takes its inspiration from LISP[105], the *lingua franca* of AI. The basic data type of LISP is the linked list, which is build up recursively from a single item, called the CAR or head of the list, and a CDR or tail, which is itself a (possibly null) list. The operation that attaches the CAR to the CDR is called CONS, which together with the Boltzmann Machine learning algorithm [43] used to train the model, gives BoltzCONS its name . To this CAR/CDR representation, BoltzCONS adds a unique *tag* symbol,which functions like a traditional pointer, except that it accesses memory by associative retrieval, rather than being an address into a sequential memory. All representations in BoltzCONS consist of such <TAG,CAR,CDR> tuples. Having the CAR and/or CDR to be another tag allows this representational scheme to encode arbitrary list and tree structures. All such structured "knowledge" is distributed among a 2000-unit "tuple memory", using a coarse-coded representation [42], in which several representations can be present in the same vector, under certain assumptions of linear independence. Hence, BoltzCONS seems to satisfy the connectionist requirement of being distributed, as well as the criteria of systematicity and compositionality laid out earlier.

1.3.2 Tensor Products

Smolensky's Tensor Product model[101] is a general connectionist framework for modeling role/filler relations. In this framework, the bindings between a set of roles and their fillers are represented as the tensor (outer) product of a filler vector and a role vector. Unlike SHRUTI, however, the Tensor Product model is not necessarily localist. As Smolensky explains, Tensor Product representations are as distributed as the representations of the roles and fillers. Under the assumption of linear independence of the role vectors, the binding operation can be inverted by an "unbinding vector," allowing the original representations of the roles and their fillers to be recovered. Conjunction of bindings is represented by vector addition, which supports "graceful saturation" of the number of bindings that the system can represent. Smolensky describes how this scheme can be used to implement the stack operations PUSH and POP, as well as the aforementioned LISP operations CAR and CDR.

1.3.3 Holographic Reduced Representations

Like Smolensky's Tensor Products model, Tony Plate's Holographic Reduced Representations, or HRRs [70], use fixed-dimensional real-valued vectors to encode role/filler relations. For a given N-dimensional role vector A and N-dimensional filler vector B, the binding C of B to A is computed as the *circular convolution* $C{=}A{\otimes}B$, defined as $c_j = \sum_{k=0}^{n-1} a_k b_{j-k}$. Extraction of the role and filler vectors from a given role/filler binding is done using the mathematical inverse of this operation, called *circular correlation*: $D = A \# C$, defined as $d_j = \sum_{k=0}^{n-1} a_k c_{j+k}$. For typical applications, the vectors are quite large, e.g., $N = 512$, making the model ideal for a parallel architecture, but rather slow on a typical serial computer[5]

HRRs have several features that make them appealing as distributed representations of cognitive processes. Most notably, unlike tensor products, HRR's use fixed-size representations, so the size of the representation does not grow as more information is encoded. Further, an HRR can be computed by a recurrent network [71], adding to their biological plausibility. These features have led a number of researchers to adopt HRRs as a cognitive models of "lower-level" processes like vision [80], olfaction [100], and audition [58], as well as behaviors traditionally seen as beyond the capabilities of connectionism, such as structural analogy [28] .

1.3.4 Summary

BoltzCONS, Tensor Products, and HRRs all satisfy the requirements of compositionality and systematicity, using distributed representations. Hence these models provide a principled

[5]The authors of [28]report "run times... on the order of days" for their HRR application to analogical reasoning.

answer to the claims of Fodor and others about the inadequacy of connectionism.

Still, all three models suffer from limitations that call into question their ability to scale up to larger compositional problems. As Pollack [73] notes, coarse-coded representations can only simultaneously represent a comparatively small number of elements without sacrificing accuracy, but they require a comparatively large set of units to represent even those items. Although HRR's do not suffer from the "potentially prohibitive growth" that Smolensky observes in Tensor Products [101], the reduced nature of HRRs requires a "clean-up" operation (dot product) in order to recognize the extracted vectors obtained by the correlation operators. From a more theoretical standpoint, BoltzCONS, with its direct encoding of traditional symbolic operations CAR, CDR, PUSH, and POP, seems to fall into the category of "mere implementation" to which connectionism is relegated by its strongest critics [36].

This thesis presents an attempt to address both sets of questions, the practical and theoretical,by means of an architecture called Infinite RAAM. Representing a new fusion between recurrent neural networks and fractal geometry, Infinite RAAM allows us to understand the behavior of these networks as dynamical systems, and to overcome the current limitations in connectionist modeling of compositional structure. To see how Infinite RAAM accomplishes these goals, however, we need to understand the original version of RAAM, which is the subject of the next chapter.

Chapter 2

RAAM

2.1 The model

Recursive Auto-Associate Memory, or RAAM [73] is a method for encoding tree structures in fixed-width, real-valued vectors.[1] A RAAM network (or more simply, a RAAM) is identical in structure to a simple feed-forward neural network with one hidden layer [85]. A RAAM is trained as an auto-associator [1] [21], with the input→hidden weights treated as an encoder (compressor), and the hidden→output weights treated as a decoder (reconstructor). The goal of training is to obtain a set of encoder and decoder weights that will yield an output pattern identical to the input pattern. Figure 2.1 shows the layout of a typical autoassociator network.

The basic insight of RAAM is that the hidden-layer activations of an autoassociator network can be used to encode combinations of distinct items presented on the input layer, and can therefore be fed back into the input layer to encode structure recursively. Given a D-dimensional vector representation of of an atom (such as `fred`, `wilma`, or `loves`), a RAAM encodes the combination of K such atoms by (1) concatenating the reprentations of the

[1]When RAAM is implemented on a digital computer the vectors contain floating-point numbers, but nothing in the RAAM or IRAAM models depends on the distinction.

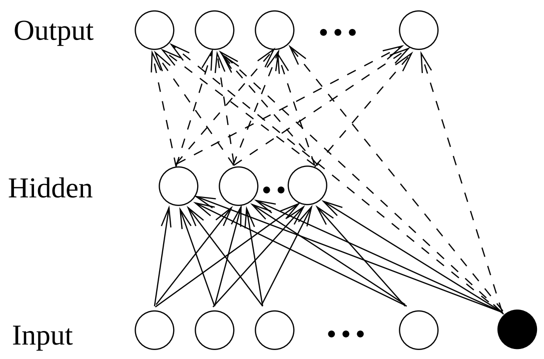
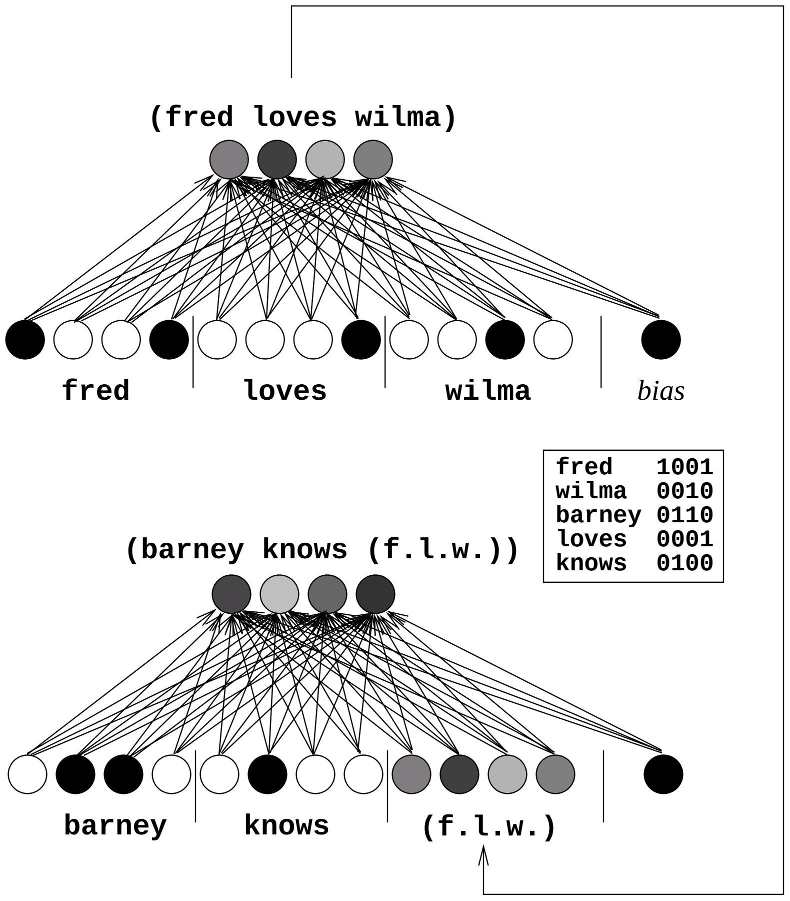


Figure 2.1: An auto-associator network. Solid lines represent encoder weights, dashed lines decoder weights. The filled unit at lower right is a bias input, whose value is always unity.

atoms into a KD-element vector, (2) placing this vector on the input layer, (3) multiplying the vector by the KD^2 input-to-hidden weights, (4) adding in a vector of D hidden-layer biases, and (5) passing the resulting KD-element vector through the logistic-sigmoid squashing function $f(x) = 1/(1 + e^{-x})$. The hidden vector now contains a D-element representation of the depth-one tree having the atoms at its leaves; e.g., `(fred loves wilma)`. This tree representation can now be fed back into the encoder and treated just like the representation of an atom. Given, for example, an intensional verb like `knows`, we can represent the sentence `(barney knows (fred loves wilma))` by concatenating the D-element vector representations of the atoms `barney` and `knows` with the D-element vector representation of the sentence `(fred loves wilma)`, and passing the resulting $3D$-element vector through the encoder as described above. Figure 2.2 illustrates this encoding process, for $D = 4$.

The RAAM decoder inverts the effect of the encoder, "breaking out" encoded representations into their K constituent parts. This process is illustrated in Figure 2.3, which continues the example from the preceding figure.

Continuing the example from these figures, here is the procedure for training a RAAM to represent the tree `(barney knows (fred loves wilma))`:



fred	1001
wilma	0010
barney	0110
loves	0001
knows	0100

Figure 2.2: A RAAM encoding the sentences `(fred loves wilma)`and `(barney believes (fred loves wilma))`. Vertical lines between inputs are for clarity and do not correspond to anything in the input vector. The key shows the hypothetical binary encodings of various atoms. Note that the representations encoding the trees are real-valued in the range (0,1), as indicated by the grayscale shading of the hidden units.

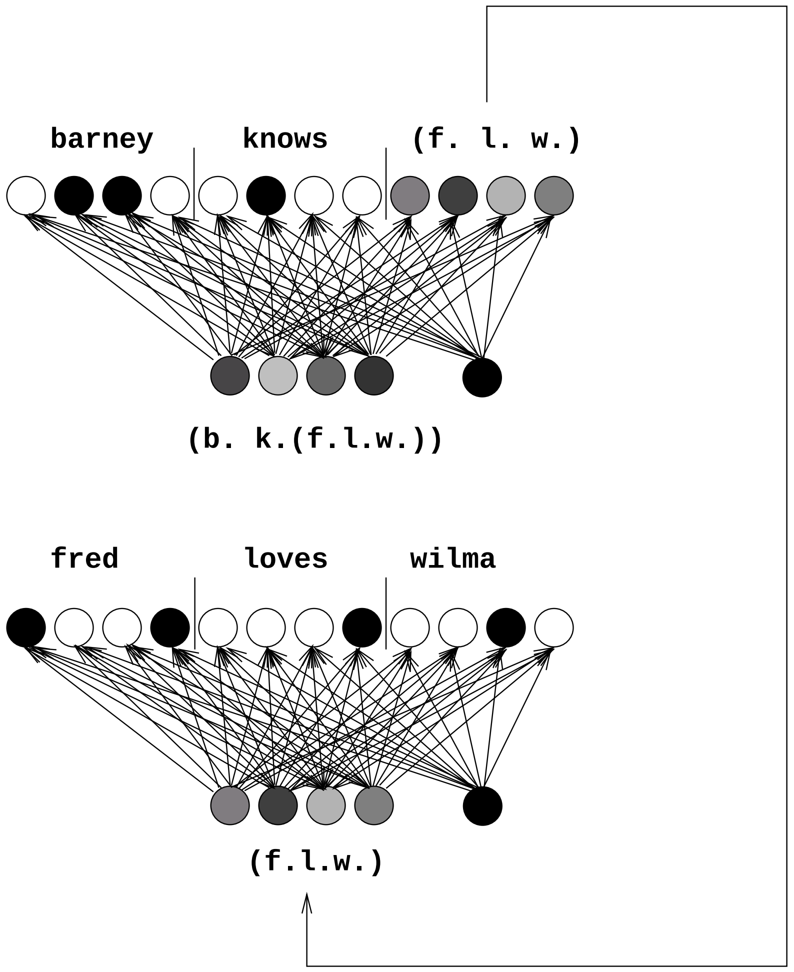


Figure 2.3: A RAAM decoding the representations from Figure 2.2. Note the absence of a bias input on the hidden layer during the decoding phase; hidden-unit values are set directly by feedback from the output units during decoding.

1. Concatenate the representations for `fred`, `loves`, and `wilma,` and place the resulting vector on the input layer.

2. Compute the activation at the hidden and output layers.

3. Compare the activation at the output layer with the encoding on the input layer. This difference is the error for the first pattern.

4. Take the hidden-layer activation of `(fred loves wilma)` from step (3), concatenate it after the representations for `barney` and `knows,` and place the resulting vector on the input layer.

5. Compute the activation at the hidden and output layers.

6. Compare the activation at the output layer with the encoding on the input layer. This difference is the error for the second pattern.

7. If the the average of the errors from steps (3) and (6) falls below a pre-determined error threshold, stop; otherwise, go to (1).

Of course, a real training set would consist of more than just one tree, but this small example gives a sense of how the learning algorithm works.

Using a 48-16-48 RAAM (48 inputs, 48 outputs, and 16 hidden units), Pollack was able to encode and decode a set of 14 such ternary propositional structures, including non-trivially nested ones representing complex facts like *The short man who thought he saw John, saw Pat.* In addition, he trained a 20-10-20 RAAM (branching factor $K = 2$)to represent seven natural-language syntactic structures like ((Det. (Adj. Noun)) (Verb (Prep. (Det. Noun))), as *The tall man ran around the room.* A third result was the ability of a 30-25-30 RAAM to represent letter sequences with repeated sub-patterns (*banana*, *barbarian*). This result directly addressed the criticism by Pinker and Prince [69]of the three-letter "Wickelphone"encoding used in the Rumelhart and McClelland past-tense model [87], in which such repetitions cannot be represented.

As with any learning model, however, the crucial test is generalization, or how the model behaves in response to novel input. For RAAM, this test is performed by taking trees from outside the training set, passing them through the encoder, and passing the resultant encoding out through the decoder. If the decoder correctly reconstructs the new trees, the network is considered to have generalized successfully on the novel input represented by their combination.

Using this generalization test, Pollack found that the 20-10-20 natural-language RAAM correctly encoded and decoded 31 syntactic trees, beyond the original seven in the training set. Of these 31 novel trees, 19 were grammatical, suggesting some degree of generativity. Similar results were obtained for the propositional and letter-sequence sets.

There is a subtlety in the generalization process that bears mentioning. Unlike training, generalization required the use of a *terminal test* on the decoder's output, to decide when to halt the feedback of outputs to the hidden layer. The reason is that, unlike the encoder, the decoder doesn't "know" the composition of the tree that it is supposed to be decoding. Given the encoding of terminals (atoms) as binary strings, a practical solution to this problem is to adopt the neural-net convention of treating values below a certain threshold (say, 0.2) as a zero, and values above a certain threshold (0.8) as a one. This is the convention that Pollack adopted for the primary RAAM terminal test. For each of the KD elements of the output vector, if all the values in that element were below 0.2 or above 0.8, the element was treated as a terminal; otherwise, it was treated as a non-terminal and fed back to the hidden layer for further decoding.

2.2 RAAM's early successes

RAAM answered the challenge of showing how neural networks could represent compositional structures in a systematic way, and led to much fruitful discussion by philosophers [111][45], as well as several novel applications. In general, these applications worked by give some special status one part of the input/output layer. The Sequential RAAM or SRAAM model of Kwasny and Kalman [56] treated the first several bits of these layers as a symbol, and the remaining bits as a stack, thereby turning RAAM into an architecture for learning lists. Sperduti's LRAAM [102] extended SRAAM by treating the initial symbol bits as graph labels and the remaining bits as graph pointers, thereby implementing general labeled graphs. Chrisman [16] modified SRAAM into a "dual-ported" RAAM that he used for natural language translation between Spanish and English. The model built up similar distributed representations for both languages on its hidden layer, and successfully decoded these representations to the corresponding sentences in each language.

2.3 Problems with RAAM

Nevertheless, the model seemed applicable to only a small set of trees [11][16], and attempts to extend it to larger sets met with mixed results [8]. The question remained whether RAAM could scale up to industrial-sized applications of the sort handled by classical symbolic AI.

The source of the problem, and the answer to the question, lies in the behavior of the decoder. As described in the preceding section, the RAAM decoder works in conjunction with a logical "terminal test," which determines whether or not a given representation requires further decoding. The default terminal test used in the primary set of experiments merely asked whether all elements in the code were above 0.8 or below 0.2. This "analog-to-binary" conversion was standard in back-propagation research of the late 1980's, but it led to several basic logical problems that prevented the scaling-up of RAAM:

1. The ***Infinite Loop*** problem in which there are representations that "break" the reconstructor by never terminating.

2. The ***Precision versus Capacity*** problem in which using tighter tolerances slow down convergence of training, while looser tolerances allow quicker conversion but lead to reduced representational capacity.

3. The ***Terminating Non-Terminal*** problem involving a "fusion" between a discovered non-terminal and a given terminal, such that the decoding of an encoded tree terminates early, resulting in an error.

Other researchers noticed problems in the default terminal test and came up with alternatives, such as simply testing membership in a list, or simultaneously training a "terminal-test network" to classify representations as terminal or non-terminal [16][106] . Nevertheless, scaling-up remained a problem, leading to the question of whether there was a "natural"behavior of a RAAM decoder that was not being taken advantage of. The answer to this question is the subject of the next chapter.

Chapter 3

IFS RAAM

[W]e take Smolensky to be claiming that there is some property D, such that if a dynamical system has D its behavior is systematic.... The least that has to happen if we are to have a substantive connectionist account of systematicity is: first, it must be made clear what property D is, and second it must be shown that D is a property that connectionist systems can have by law.

– J. Fodor and B.P. McLaughlin,

"Connectionism and the Problem of Systematicity" [35]

3.1 Iterated Function Systems

Consider again the RAAM decoder in Figure 2.3. The behavior of this decoder can be described by the set of equations:

$$T_k \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = f\left(\begin{bmatrix} w_{k,1,1} & w_{k,1,2} & w_{k,1,3} & w_{k,1,4} \\ w_{k,2,1} & w_{k,2,2} & w_{k,2,3} & w_{k,2,4} \\ w_{k,3,1} & w_{k,3,2} & w_{k,3,3} & w_{k,3,4} \\ w_{k,4,1} & w_{k,4,2} & w_{k,4,3} & w_{k,4,4} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} w_{k,1} \\ w_{k,2} \\ w_{k,3} \\ w_{k,4} \end{bmatrix}\right) \tag{3.1}$$

where $T_k, k \in \{1,2,3\}$ is the set of *transforms* that decodes the left, middle, and right part of the tree encoded on the hidden layer. The square-matrix elements $w_{k,i,j}$ and vectors $w_{k,i}, i \in \{1,2,3,4\}$ are the set of *weights* and *biases* for that transform; and$f$is the logistic-sigmoid squashing function $f(x) = 1/(1+e^{-x})$ .This set of equations bears a striking similarity to an *Iterated Function System* (IFS) , for example, the equations that generate the well-known Sierpinski Gasket:[1]

$$T_1 \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{3.2}$$

$$T_2 \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 0.5 \end{bmatrix} \tag{3.3}$$

$$T_3 \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} \tag{3.4}$$

Iterated Function Systems get their name from the fact that, starting with some initial conditions for the x_i , each of these transforms is applied iteratively to its own output or to the output of one of the other transforms, the choice of which transform to apply being made either deterministically or by non-deterministic probabilities associated with each transform. Starting on the unit square, all three transforms of the Sierpinski Gasket IFS scale the x and y coordinates of the square by one-half. The first transform T_1 leaves this contracted copy of the square in its original position at the origin (lower-left quadrant); the second transform T_2 translates (shifts) this copy to the top-left corner of the unit square, and the third transform T_3 translates the copy to the top-right corner. The second application of the transforms does the same thing to the three unit-square copies.

[1]The Sierpinski Gasket is usually shown with an equilateral triangle as its initial conditions; the version here, modified from the equations in [4], uses the unit square instead, to highlight the similarity with fractal RAAM.

The limit of this process as the number of iterations N approaches infinity is called the *attractor* of the IFS. Viewed as a set, the attractor is the fixed point of the IFS transforms, because the transforms take all points in the attractor to themselves or to other points in the attractor. An IFS has an attractor if and only if all of its transforms are *contractive* [4], meaning that for any two points x and y, and a distance metric d, there is a constant s, $0 \leq s < 1$, such that $d(T_k(x), T_k(y)) \leq sd(x, y)$. For a linear IFS like the Sierpinski Gasket, it is straightforward to determine whether a transform is contractive, by computing the eignvalues of the square matrix part of the transform. If the largest magnitude of the eigenvalues is less than one, the transform is contractive.

Figure 3.1 shows a representation of the Sierpinski Gasket attractor. The image is necessarily an approximation, because the actual attractor is a discontinuous *dust* of distinct points that exists only in the limit as the number of iterations N approaches infinity. The image in the figure, and the other attractor images in this thesis, were generated from a pixelized representation of the unit square, using the following algorithm [2]:

[2] In displaying algorithms we have adopted the conventions of [20]

```
ATTRACTOR($T, S$)

1 $A_{new}$ ←UNIT-SQUARE($S \times S$)
2 $A_{old} \leftarrow \emptyset$
3 while $A_{old} \neq A_{new}$
4     $A_{new} \leftarrow \emptyset$
5     for $k \leftarrow 1$ to $K$
6         do $A_{new} \leftarrow A_{new} \cup T_k(A_{old})$
7     $A_{old} \leftarrow A_{new}$
8 return $A_{new}$
```

Starting with an $S \times S$-pixel representation of the unit square, this algorithm applies the two transforms to unit square, takes the union of the transformed images, and iterates until no new points are generated. The resulting shape is called a *fractal*, a name credited to Mandelbrot, who applied such shapes to the study of patterns in nature, such as clouds and coastlines [60]. Unlike traditional geometrical objects like circles and cubes, a fractal typically does not a have a dimension that is a whole number; instead, as its name suggests, its dimension is a fraction (more accurately, a non-integer real). As described in [93], for an IFS this *Hausdorff dimension* can be computed as the ratio of the logarithm of the number of transforms to the logarithm of their scaling factor; the Sierpinski Gasket therefore has a Hausdorff dimension of $log(3)/log(2) \approx 1.58$.

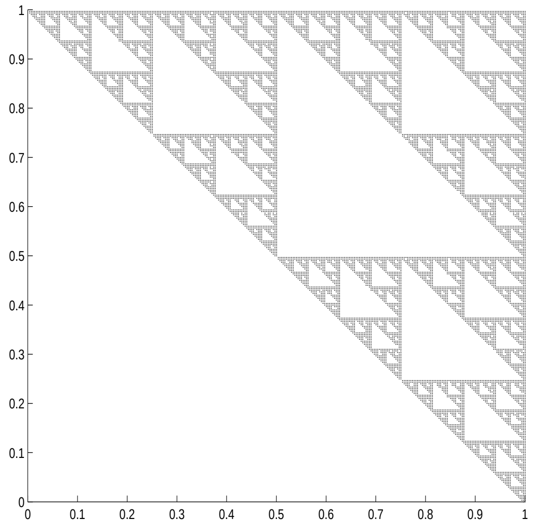


Figure 3.1: The Sierpinski Gasket attractor

3.2 RAAM as an IFS

Consider the following RAAM decoder, with $K = D = 2$:

$$T_1 \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = f\left(\begin{bmatrix} -1.272 & -4.914 \\ -1.569 & -1.515 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 2.964 \\ 4.272 \end{bmatrix} \right) \tag{3.5}$$

$$T_2 \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = f\left(\begin{bmatrix} -4.399 & 5.813 \\ -2.018 & -0.791 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 3.019 \\ -3.372 \end{bmatrix} \right) \tag{3.6}$$

Each transform of the decoder has the form $T(x) = f(g(x))$, where g is a linear function identical in form to an IFS transform, and f is the the non-linear logistic-sigmoid activation "squashing" function. The range of f is the closed interval (0,1). Therefore, even if the linear component of the transform is non-contractive, the point x is always "brought back" into the unit square by the squashing function. This property makes RAAM decoder transforms *pseudo-contractive*: although a transform may not necessarily move two points closer together, it will shrink the successive areas of the images of the unit square on successive

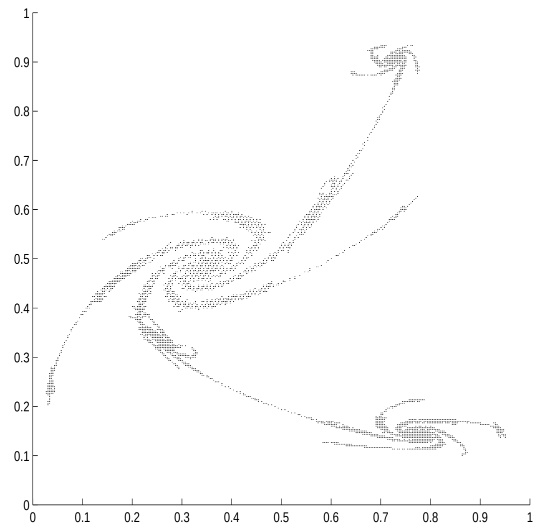


Figure 3.2: The Galaxy attractor

iterations. [3]

In fact, the non-linearity of the decoder yields a surprisingly rich variety of attractors, as compared with contractive linear IFS's. Figure 3.2 shows one such attractor, for the decoder in (3.5) and (3.6). This decoder was evolved to look like a galaxy, using a "blind watchmaker" paradigm[25].

3.3 The attractor as a terminal test

We are now in a position to answer the question that ended the last chapter. Treating the RAAM decoder as an IFS, the terminal test for a given point (vector) corresponds to asking the question "is this point on the attractor?" If the answer is yes, we treat the point as a terminal (primitive) and stop decoding; if not, we continue:

[3]This is easiest to see for a one-dimensional (D=1) RAAM. The equation for each transform is of the form $T(x) = f(ax + b)$. Let $a = 5, b = -2$, and consider what happens to the two points $x_1 = 0.4$ and $x_2 = 0.6$. The Euclidean distance between these points is 0.2. Their transforms are $f(x_1) = 0.5, f(x_2) = 0.7311$; the Euclidean distance between these two transformed points is $0.2311 > 0.2$. Nevertheless, the images of the unit line-segment decrease monotonically in length under this transform.

```
TREE-AT-POINT($x, y, A, S$)

1 $i \leftarrow \lfloor S \cdot x \rfloor$;  $j \leftarrow \lfloor S \cdot y \rfloor$
2 if $A_{i,j} = 1$
3   then return TERMINAL-TREE
4   else left-tree $\leftarrow$ TREE-AT-POINT($T_1(x, y), A, S$)
5        right-tree $\leftarrow$ TREE-AT-POINT($T_2(x, y), A, S$)
6        return NON-TERMINAL-TREE (left-tree, right-tree)
```

Given a random initial condition, any sequence of left/right decodings will ultimately put the output on the attractor; therefore, any initial condition not on the attractor will have a structured transient to the attractor.

The encoder under this formulation is simply the mathematical inverse of the decoder: given a decoder transform T_i having weight-matrix $\boldsymbol{A}$ and bias vector $\boldsymbol{b}$, the encoder for that transform is expressed as $T_i^{-1}(x) = [f^{-1}(x) - b] \times A^{-1}$, where A^{-1} is the matrix inverse of $\boldsymbol{A}$, and $f^{-1}(x) = -ln(1/x - 1)$. This inverse transform is expansive, so it will map a single pixel to a set of one or more contiguous pixels.[4]Given two D-dimensional pixels x and y encoding respectively the terminals a and b, applying the left inverse transform $T_1^{-1}$to x and the right inverse transform $T_2^{-1}$to y gives us a set of pixels representing the sets of points that contract to x and y on the "forward" transforms T_1 and T_2. Intersecting these sets (by taking the set of pixels common to both), and subtracting out any attractor points in the intersection[5], gives us the pixels that decode to the tree $(a\, b)$.

[4]The inverse transform is still a function, because it is in reality mapping a single *region* in $\Re^D$to another region in that space. Procedurally, we can take the pixel boundaries, pass them through the inverse map, and re-pixelize the resulting region, or we can maintain a lookup table mapping each point to the set of its inverses.

[5]The attractor is the fixed point of the IFS transforms, so applying an inverse transform to an attractor point can produce a point that is also on the attractor. Such a point is by definition not a non-terminal, so it must be removed from the inverse set.

X	(((X X) X)((X (X X)) X))
(X X)	((X X)(X ((X (X X)) X)))
((X X) X)	(((X X) X)((X (X X)) X))
(X (X X))	((X (X (X X)))((X (X X)) X))
((X X)(X X))	(((X X) X)((X (X X))(X X)))
((X (X X))(X X))	(((X X)(X X))((X (X X)) X))
(X ((X (X X)) X))	(((X X)(X X))((X (X X))(X X)))
(X (X ((X (X X)) X)))	(((X (X (X X)))(X X))((X (X X)) X))
((X (X (X X)))(X X))	(((X X)(X ((X (X X)) X)))(X ((X (X X)) X)))
(((X X) X)(X (X X)))	((X (X X))(((X X)(X ((X (X X)) X)))(X ((X (X X)) X))))
((X X)(X ((X (X X)) X)))	((((X X)(X ((X (X X)) X)))(X ((X (X X)) X)))(X ((X (X X)) X)))

Table 3.1: Trees decoded by the Galaxy system at 10x10 pixels. X indicates a terminal.

Hence we can fully understand the problems described at the end of Chapter 2:

1. By not using the attractor as the terminal test, decoding a random initial condition could lead to an infinite loop of decodings which are on the attractor yet never satisfy the "logical" terminal test.

2. Because the training allowed non-terminals to float around, some non-terminal could float into regions defined as terminals, leading to the early termination problem.

3. Finally, because the attractor has a fractal nature, it could not be easily modeled by a simple thresholding terminal test.

This new terminal test solves a fundamental problem in RAAM, resulting in decoders whose capacity seems extraordinarily high, as compared with the apparent limitations or original RAAM. For example, even at the comparatively low resolution of 10x10 pixels, the Galaxy RAAM was able to decode the 22 trees shown in Table 3.1.

Because each point (pixel) decodes to a unique tree, the new terminal test also gives us a spatial mapping between regions in the unit square and the trees that they decode. In this interpretation, each possible tree, instead of being described by a single point, is now an *equivalence class* of initial points sharing the same tree-shaped trajectories to the

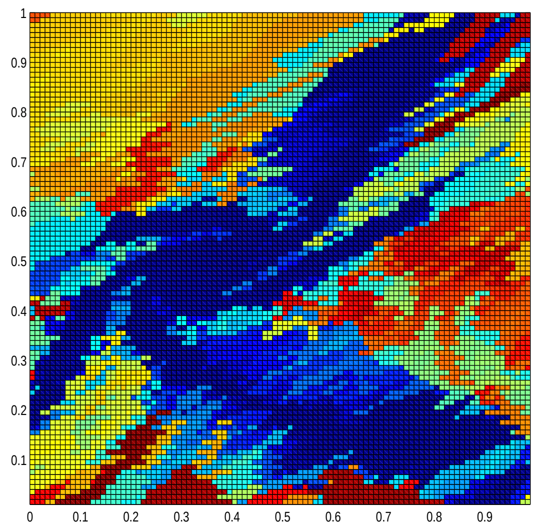


Figure 3.3: Map of tree locations for Galaxy, $S = 100$. Pixels of a given color decode to the same tree, pixels of different colors to different trees.

fractal attractor. Figure 3.3 shows the set of tree equivalence classes, for the Galaxy decoder sampled at $S = 100$.

3.4 Scaling behavior

Generative grammars contain recursive rules whose application to their own output produces structures (derivation trees) of increasing size. The ability of such systems to produce arbitrary numbers of such structures, of arbitrary size, accounts for much of their popularity as models of human language and related cognitive phenomena. Any candidate model of language, connectionist or otherwise, must likewise offer a mechanism by which novel structures can be decoded by a fixed-size model.

IFS RAAM offers two such mechanisms: iteration and resolution. The ATTRACTOR algorithm above is parametrized by a resolution argument S, and simulates an "infinite" number of IFS iterations via a **while** loop, which terminates when no new points are decoded. This loop could however be replaced by a **for** loop, which would terminate after some fixed number of iterations specified by an additional input parameter N. This parameter limits the depth of the trees that the IFS can decode; in fact, that number is exactly equal to N. [6] With a high enough resolution S, increasing values of N should give us a good idea of how the actual number of decoded trees scales up with increased iterations.

As an informal exploration of these two scaling parameters, we examined the scaling behavior of four different IFS RAAM decoders: (1) the Galaxy decoder; (2) a decoder with random weights in the same range as the Galaxy; (3) a decoder hill-climbed to maximize the proportion of left-branching trees[7]; and (4) a decoder hill-climbed to maximize the total number of trees. Figure 3.4 depicts the results of this experiment as a log/log plot of number of trees against pixel resolution, using the "infinite iterations" method. The figure shows that the random-weight and left-branching systems scale up rather modestly with increased pixel resolution. This result suggests that mediocre scaling is the baseline, or "un-biased" behavior of the IFS RAAM decoder. Because the left-branching network was trained to decode only one tree at a given depth – an infinitesimal fraction of the total number of possible trees at that depth – this network likewise fails to exhibit dramatic scaling behavior.

More interesting, however, is the behavior of the Galaxy decoder and the "Max" decoder trained to yield a larger number of trees. Both of these decoders show dramatic scaling with increased pixel resolution. Unsurprisingly, the Max decoder exhibits higher capacity than the Galaxy, across various resolutions. Intriguingly, however, the Galaxy is not far behind, and keeps pace with the Max decoder as the resolution increases, despite the fact that the criterion for producing the Galaxy network was aesthetic, making no explicit reference to tree capacity. This result points to the possibility of a deep correspondence between the visual notion of "interesting" images and the linguistic notion of generativity, related perhaps

[6] Proof by induction on N. Basis: $N = 0$, in which case whole unit square is attractor, so max tree depth is 0. Inductive step: For $N \geq 0$, assume max tree depth is N. Iterating once more means that some points on the attractor are now off the attractor but go to the attractor on one more iteration. These points were the leaves of the depth-N trees, which are now depth $N+1$. In practice, the number of trees can sometimes decrease slightly because of aliasing caused by finite pixel resolution.

[7] i.e., trees of the form (X (X X)), (X (X (X X))), (X (X (X (X X)))), etc.

to the degree of self-similarity exhibited by the IFS attractor.

Figure 3.5 shows the number of decoded trees as a function of the number of IFS iterations, at a fixed resolution of 256x256 pixels. The scaling properties here are similar to those of the previous figure, with a clear separation between the high-capacity Max and Galaxy decoders and the low-capacity left-branching and random decoders. The important and obvious difference is the leveling-off of the random network, as compared with the other three networks, after five iterations. Essentially, a random set of weights has no "inductive bias" to produce trees according to some (grammar-like) pattern, so at a fixed pixel resolution it rapidly reaches its maximum tree capacity.

It is also interesting to ask whether a particular decoder has a constant "scaling coefficient" that describes how the number of trees changes with increased resolution or iterations. Figures 3.6 and 3.7 show the scaling coefficients for the four decoders, defined as the square root of the number of trees decoded at a given resolution[8] (Figure 3.6) or the number of trees decoded at a given number of iterations (Figure 3.7). For pixel resolution, the results seem fairly clear. After an initial startup transient at low resolutions, the random and left-branching decoders pattern together, with a coefficient of around 0.05. This low value for the random decoder makes sense, as there is no reason to assume that an arbitrary set of decoder weights will exhibit interesting scaling properties. As for the left-branching decoder, an "ideal" network for left-branching trees would decode only one tree at a given number of iterations, regardless of pixel resolution; therefore, any decent approximation to that ideal is not expected to show interesting scaling properties with resolution. The Galaxy is next, with a coefficient of around 0.5, and the Max decoder scales up the fastest, with a coefficient of 0.9. The pattern is similar for scaling over iterations (Figure 3.7), although the tendency there is less clear than for scaling by resolution. The downward trend of the curves over iterations can probably be explained as an artifact of the fixed pixel resolution at which the

[8]The square root is taken in order to account for the fact that the number of possible trees goes up as the square of the pixel resolution, for two-dimensional networks like these.

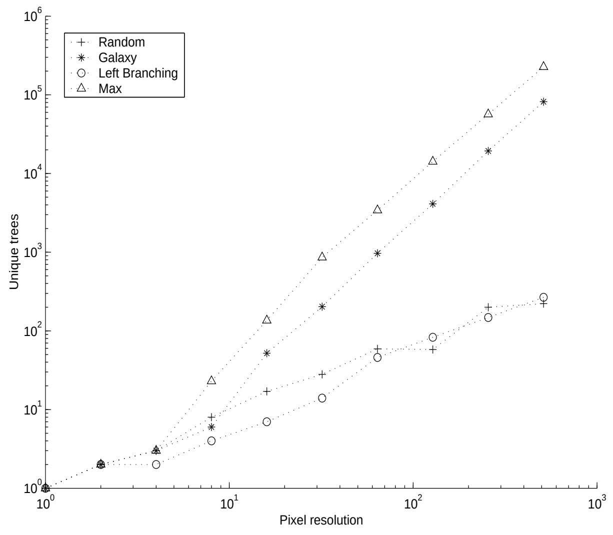


Figure 3.4: Number of trees as a function of pixel resolution, for four different IFS RAAM decoders

number of trees was computed. In the limit as the pixel resolution approaches infinity, the number of trees decoded per iteration might be expected to level off to a single value.

3.5 Discovering the competence of IFS RAAM

As Figures 3.4 - 3.7 demonstrate, treating the RAAM decoder as an IFS produces a neural network model capable of representing very large numbers of fixed-arity tree structures (and therefore very long strings at the tree frontiers), without the addition of any nodes to the network. IFS RAAM thus addresses the previously discussed inability of traditional connectionist language models to handle larger structures without prohibitive growth in the network size. We might therefore tentatively suggest the IFS attractor principle as a potential candidate for Fodor's "Property D" – a property which allows a dynamical system to have systematic behavior, and which connectionist systems can have by law.

The ability of IFS RAAM to represent such large number of trees leads one to wonder whether there is some grammar-like constraint on the set of trees that a given RAAM can

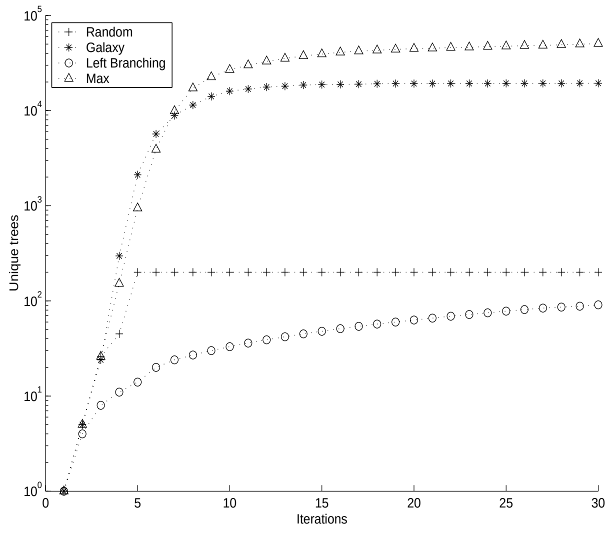


Figure 3.5: Number of trees as a function of IFS iterations

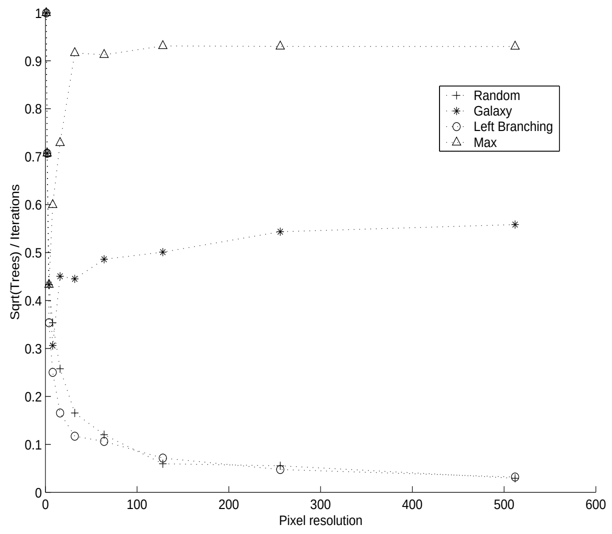


Figure 3.6: Scaling ratio as a function of pixel resolution

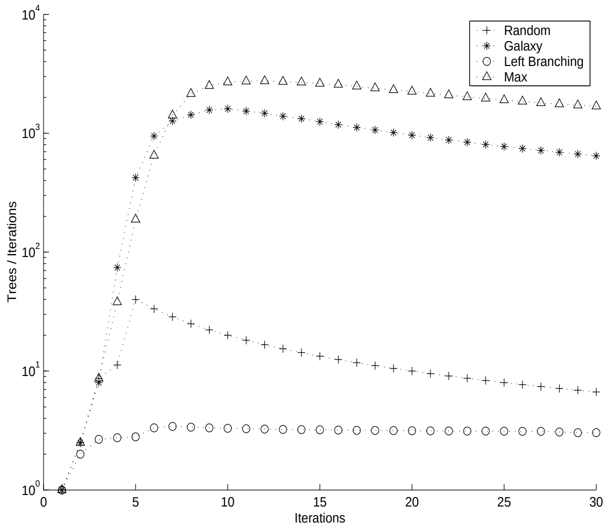


Figure 3.7: Scaling ratio as a function of IFS iterations

decode. Does a RAAM, like a generative grammar, have some "competence" for representing an infinite set of recursively enumerable structures, the actual "performance" of the RAAM being constrained by the pixel resolution at which the attractor and trees are represented?[9] The answer to this question is the subject of the next chapter.

[9]For the competence/performance distinction, see [14].

Chapter 4

Infinite RAAM

4.1 Representing formal languages

Although there has been some research into the properties of tree-generating systems [38], the bulk of knowledge in formal language theory comes from investigations of the set of strings that a given device (grammar, finite automaton) can generate or recognize[44]. Such investigations typically use strings over alphabets with more than one symbol, because single-symbol alphabets do not allow very fine-grained distinctions in computational complexity[1] Hence, investigating the formal string-generating properties of IFS RAAM requires a principled method for labeling the terminals

Fortunately, using the attractor as a terminal test also allows a natural formulation of assigning labels to terminals. Barnsley [4] notes that each point on an IFS attractor is associated with an address which is simply the sequence of indices of the transforms used to arrive on that point from other points on the attractor. The address is essentially an infinite sequence of digits. Therefore to achieve a labeling for a specific alphabet we need only consider a sufficient number of significant digits from this address; in the present study, we need just one. All pixels reachable from the attractor on the left transform are arbitrarily

[1]A context free grammar over a one-symbol alphabet is regular. See [90] for proof.

labeled a, and all pixels reachable on the right transform are labeled b. Figure 4.1 illustrates this process for the the Galaxy decoder, showing sample derivations for the trees $(a\ b)$ and $(a\ (a\ b))$. The addressing algorithm is given below:

```
ADDRESS-POINTS$(T, A, S, K, N)$

1  $B \leftarrow$ ZERO-MATRIX $S \times S$
2  **for** $i \leftarrow 1$ to $K^N$
3    **do** $j \leftarrow i - 1$
4       **for** $k \leftarrow 1$ **to** $N$
5         **do** $P_k \leftarrow$ MOD $(j, K) + 1$
6            $j \leftarrow \lfloor j/K \rfloor$
7       $C \leftarrow$TRANSFORM$(T, A, P, K)$
8       $C \leftarrow C \cdot K^{i-1}$
9      $B \leftarrow B + C$
10 **return** $B$
```

In this algorithm, T is the set of IFS transforms, A is the attractor, S is pixel resolution, K is the number of transforms, and N is the number of digits to use for computing the addresses. The TRANSFORM function called in line 7 accepts a sequence P of transforms (the "path") and returns an image computed by applying the successive transforms in P to the initial image A :

```
TRANSFORM($T, A, P, K$)

1 for $i \leftarrow 1$ to $K$

2     $A \leftarrow T_{P_i}(A)$

3 return $A$
```

C contains the image computed by TRANSFORM. C is an $S \times S$ matrix containing zeros and ones, so multiplying it by successive powers of K (line 8) produces an image containing zeros and integer addresses (1,2,4,8,...). In line 9, C is added into the (initially all zero) SxS matrix B, performing the desired accumulation of "impure" (overlapped) addresses. The algorithm returns an SxS matrix A containing the original attractor, but with positive integer values instead of ones for the "on" pixels. These values can then be used as indices into symbol table or dictionary, as they are in the two-symbol $a^n b^n$ hill-climbing described below.

4.2 Resolution as induction

Under the IFS formulation of RAAM, the set of trees generated and represented by a specific decoder is a function of the weights, but is also governed by how the initial condition space is sampled, and by the resolution of the attractor construction. The lower-resolution attractors contain all the points of their higher-dimensional counterparts (they cover them); therefore, as a coarser terminal set, they terminate trajectories earlier and so act to "prefix" the trees of the higher-resolution attractors, as illustrated in Figure 4.2.[2] Furthermore, a finite sampling means that each tree equivalence class takes up some finite percentage of the space. Hence, despite the well-established difficulty of inducing grammars in the general case [2][37][39],

[2]This behavior is reminiscent an L-System [53], a type of rewriting system which contains rules for replacing one symbol with a list of symbols drawn from the same alphabet. As formal grammars are a more common approach to the analysis of recurrent networks than are L-Systems, the former approach is taken here.

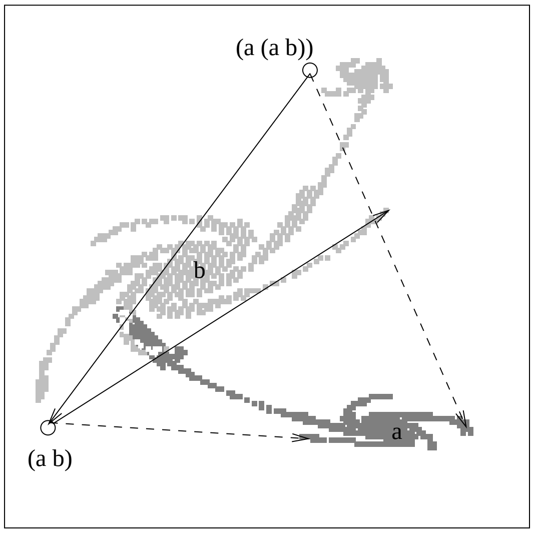


Figure 4.1: The Galaxy attractor, showing derivation of the tree $(a\ (a\ b))$ and its daughter tree $(a\ b)$. Attractor points with addresss a, reachable from the attractor on the left transform, are colored dark gray; points with address b, reachable on the right transform, are light gray. The left transients to the attractor are shown as dashed lines, and the right transients as solid lines.

an IFS RAAM that has "learned" a small set of exemplars from an infinite language and can generate longer exemplars of that language at an increased resolution, may be seen as an *inherently* probabilistic grammar for that language. This situation can be contrasted with traditional stochastic grammars, in which a stochastic component (transition probabilities) is added to a existing deterministic formalism (context-free grammar), as in [47]. We explore the issue of induction in subsequent chapters.

4.3 Hill-climbing an $a^n b^n$ decoder

So far, we have seen that the IFS interpretation of RAAM supports extremely large numbers of distinct tree structures. What is interesting however about real languages is not merely that they consist of large numbers of structures (or strings), but rather the fact that these structures conform to some set or properties, as expressed in a grammar or other generative

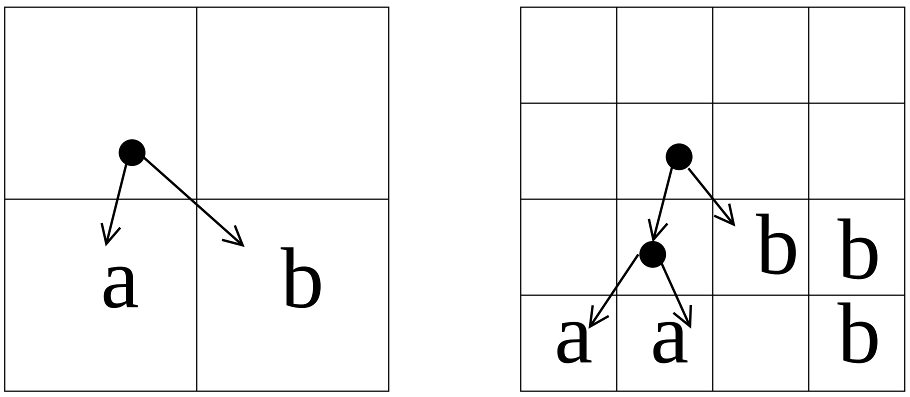


Figure 4.2: Lower-resolution attractor prefixes trees of higher-resolution attractor, in a hypothetical example. Same starting point decodes to tree `(a b)` at lower resolution (left), and to tree `((a a) b)` at higher resolution (right).

mechanism. A logical next step would therefore be to ask whether an IFS RAAM can represent (and learn) some (perhaps infinite) language of trees or strings conforming to a grammatical rule or rules. This leads to the question of which languages to use as targets.

A classic result in formal language theory is Chomsky's demonstration that a finite-state automaton is incapable of generating (recognizing) a certain set of natural language constructions, including nested *if... then* clauses and subject-verb agreement [12]. Because of the necessity of matching each *if* with its corresponding *then* (or each subject with its verb), such constructions are equivalent in complexity to the language $L = \{a^n b^n\}$, that is, the set of strings consisting of a sequence of a's followed by an equal-length sequence of b's. Hence, a finite-state automaton, which cannot count beyond a fixed number, cannot generate or recognize such strings. Therefore, this language has been used by others as a minimally non-trivial criterion for adequacy of connectionist language models [83], and is a good candidate for exploring the capacity of IFS RAAM.

Hill-climbing was used to arrive at a set of RAAM decoder weights for the language $L = \{a^n b^n\}$. Two ways to represent the targets for hill-climbing would be either a set of strings in $L : \{ab, aabb, aaabbb, ...\}$, or a set of parenthesized expressions representing binary-branching trees having those strings at their frontiers: $\{(a\ b), ((a\ (a\ b))\ b), ((a\ ((a\ (a\ b))\ b))\ b), ...\}$. RAAM is a method for representing structure, and not just strings of symbols. Therefore, the latter, tree-based representation was chosen. Specifically, trees generated by a simple context free grammar for L were used, under the assumption that this choice would drastically restrict the set of possible solutions to be explored and allow the hill-climbing RAAM to build upon existing structure as it navigated the space of decoder weights.

For the hill-climbing, the initial random weights came from a Gaussian distribution with zero mean and a standard deviation of 5.0. Starting with 12 random decoder weights, the space of weights was explored by adding random noise with zero mean and 5.0 standard deviation to each weight and using the resulting weights to generate trees on a 64×64 fractal RAAM. That is, the attractor was generated at that resolution and the initial starting point space was also sampled at that resolution. The terminals of these trees were addressed with an a or a b, using the scheme described in the section above. The training set consisted of 10 trees representing strings from $\{a^n b^n\} \cup \{a^{n+1} b^n\}$ with $n = \{1, 2, 3, 4, 5\}$ which had subpart relationships (*e.g.*, the tree for $aabb$ is a subpart of the $aaabbb$ tree) for the learning set[3]. Training was halted when the fraction of trees learned failed to increase after 30 random mutations. Note that successful representation of a single large tree, *e.g.*, $((a\ (a((a\ ((a\ (a\ b))\ b))\ b)\ b))\ b)$, would entail representation of all its smaller, well-formed sub-parts (and hence substrings). Hence, this single large tree could have been used as the sole member of the training set. Nevertheless, such a large structure would provide no "useful bias"[48] for hill-climbing, making the task extremely difficult. For this reason, we chose to present all the intermediary trees as well.

About a third of the trials were able to mutate successfully into patterns that "covered" the training set, yielding all ten tree structures, as well as trees of the form $a^n b^{n+1}$, plus additional, ill-formed trees. Although hill-climbing generated many different weight set

[3]We included the strings in $\{a^{n+1} b^n\}$ because they helped provide a gradient between successive strings in the target language.

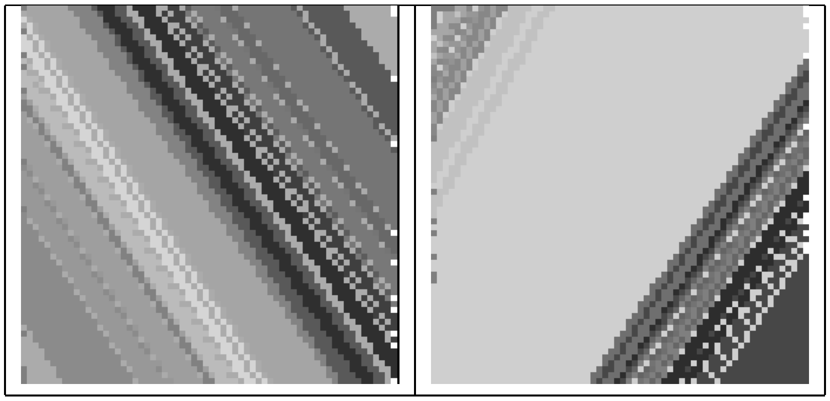


Figure 4.3: The equivalence classes of two solutions to $L = a^n b^n$found by hill-climbing

solutions to cover the training data, Figure 4.3 shows that all the solutions had a dramatic "striping" pattern of tree equivalence classes, in which members of a single class were located in bands across the unit square. (Recall that any point not on the attractor represents a tree.) So, for example, the wide gray band occupying most of the top of the image at right represents the equivalence class for the tree $(a\ b)$. Furthermore (and less noticeable in the figure), the attractor for these hill-climbed weights was located on or toward the edge of the unit square. In the figure, the b attractor points are the white squares on the right side of the image at left.

Beyond the 64×64 resolution for training the RAAM did not generalize deeply. However, the dramatic consistency in the solution patterns led us to wonder whether there was an underlying formal solution toward which our $a^n b^n$ hill-climbing RAAM was striving. As we discuss in the next section, the answer to this question turned out to be positive.

4.4 Competence model

The RAAM evolved by this hill-climbing experiment is indicative of a class of RAAM competence models which generate the languages $L_1 = a^n b^{n+1}$and $L_2 = a^{n+1} b^n$. This claim can be justified by demonstrating how an analysis of the specific RAAM dynamics garners

	w_{xx}	w_{xy}	w_x	w_{yx}	w_{yy}	w_y
Left	-32.7	0.442	*-5.037*	9.838	-6.989	*-5.349*
Right	28.138	-7.534	*29.634*	10.273	-6.866	*2.258*

Table 4.1: Transform weights for the first attractor in Figure 4.3. Biases are in italics.

the principles to design a parameterized class of $L_1$and L_2 RAAMs, some of which, in the infinite case, generate the whole languages.

Table 4.1 shows the transform weights for the attractor on the left side of Figure 4.3. Looking at these weights, it is not easy to discern an immediate pattern. A plot of the IFS dynamics, as in Figure 4.4, makes the effect of the transforms much more apparent. This figure shows what the left and right transforms do to various points in the unit square. For the $a^n b^n$ system, the dynamics reveal an attractor in the lower-left of the unit square (0, 0), and one in the upper-right (1,1), corresponding to the left and right transforms, respectively. All points move toward the left side on the left transform, and points already on the left side move toward the bottom. Conversely, all points move right on the right transform, and points already on the right move toward the top. [4]

Returning to the transform weights in Table 4.1, it is now easier to see what is going on. Looking first at the left transform, it is apparent that the overall large negative weight (-32.7, 0.442) and negative bias (-5.037) on the X coordinate mean that any point in the interval (0,1) will map to a value close to zero under the logistic sigmoid function. Similarly the overall very large positive weight (28.138, -7.534) and bias (29.634) on the X coordinate for the right transform guarantee that any point in that interval will map to a value close to one. Focusing on the Y coordinate, we see that the corresponding weights for both the left (9.838,-6.989) and right (10.273,-6.866) transforms are very similar to each other, but the biases differ in sign: -5.349 for the left transform, versus 2.258 for the right. This difference in the biases means that the left transform tends to push all points downward toward zero,

[4]This pattern differs dramatically from the dynamics of the Galaxy, in which the left transform has an attractor at roughly the point (0.75, 0.45), and the right transform as an attractor at roughly (0.5, 0.35).

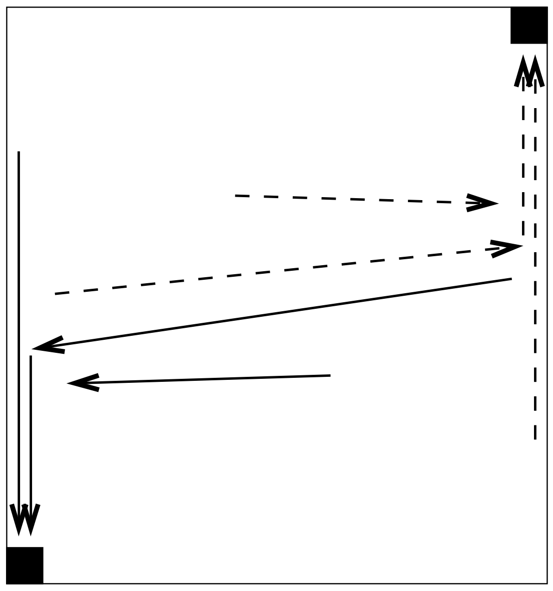


Figure 4.4: System dynamics for the $a^n b^n$ decoder. Left transforms are shown by solid arrows, right transforms by dotted arrows.

and the right transform pushes them up toward one.

More precisely, one can imagine a diagonal line between the upper-left and lower-right corners of the unit square. All points to the left of (below) this line go to the lower-left corner on the right transform, and all points the the right of (above) this line to the upper-right corner on the right transform. This behavior gives rise to the diagonal bands of equivalence classes in the first image of Figure 4.3.

With these dynamics in mind, an explanation of the functioning of the $a^n b^n$ decoder is close at hand. The left transform of a given point (x, y) will take the point to the left side of the unit square. From there, the left transform will take this $(0, y)$ point to the a attractor at $(0, 0)$. The right transform of the $(0, y)$ point will take this point to the right side of the unit square. The right transform of this $(1, y)$ point will take this point to the b attractor at (1,1), and the left transform will take it back to the left side, starting another iteration of the process. Put more simply, the dynamics specify an algorithm that says "go the the left, drop off an a , then go the right and drop off a b, the return to the left and drop off

x	w_{xx}	w_{xy}	w_x	w_{yx}	w_{yy}	w_y
Left	0	0	$-\ln\frac{1-\epsilon}{\epsilon}$	c	$c(1-2\epsilon)$	$c(1-\epsilon)+w_x$
Right	0	0	$\ln\frac{\epsilon}{1-\epsilon}$	c	$c(1-2\epsilon)$	$c(1-\epsilon)-w_x$

Table 4.2: Idealization of the first set of decoder weights in Table 4.1

another a, then ...."

This account of the $a^n b^n$ decoder leads to the question of whether there is a "pure" set of weights which guarantee the correct dynamics at any resolution. Table 4.2 displays these weights, as a function of the pixel size ϵ and a scaling factor c that parametrizes the diagonal line between the upper-left and lower-right corners. A recent paper of ours [65] explains the derivation of these weights, and gives a formal proof that they induce the entire language $L = a^n b^n$ under any choice of ϵ, for $c > 0$.

The existence of this set of pure weights allows us to consider the RAAM decoder as not merely an *approximation* to the context-free language $a^n b^n$, but rather as a full-blown *competence* model for that language. Our network is just as capable of generating all strings of this language as is a context-free grammar for the language, such as $S \rightarrow aSb|ab$. For this reason, we have chosen the name "Infinite RAAM" or "IRAAM" to describe this model and work arising from it, which forms the topic of this thesis.

These results may also be understood in the context of related work in the encoding of grammars by dynamical systems. Crutchfield and Young [22] found that at a "critical" parameter value, the quantized outputs of the logistic map formed an indexed context-free language; such a language is part of the same qualitative class of "slightly context-sensitive" languages.[5] A more general result has been obtained by Tabor [107], who discusses a general class of gated linear recurrent networks he calls Dynamical Automata. Tabor provides a proof that these automata can recognize context-free languages using rational weights, and context-sensitive languages using irrational weights. The non-linearity of the RAAM activation function makes this sort of analysis more difficult; however, our results with the

[5]We discuss the behavior of RAAM on one such language, in the next chapter.

$a^n b^n$ network, and the networks discussed in the next chapter, are consistent with Tabor's analysis.

4.5 Hill-climbing an ambiguous decoder

Another major feature of natural language noted in Chomsky's seminal 1956 paper [12] is syntactic ambiguity; *i.e.*, the ability of a single NL sentence or phrase to correspond to two or more representational structures[6]. Chomsky's example *They are flying planes*, corresponding to the parses [$They_N$ [[are_{AUX} $flying_V$] $planes_N$]] and [$They_N$ [are_V [$flying_{ADJ}$ $planes_N$]]], can be derived from the following grammar, in which *flying* is ambiguously a verb or an adjective:[7]

$S \rightarrow NP\ VP$

VP → V NP | IP NP

NP → (ADJ) N

IP → AUX V-ing

AUX → *are*

V → *are* | *fly*

ADJ → *flying*

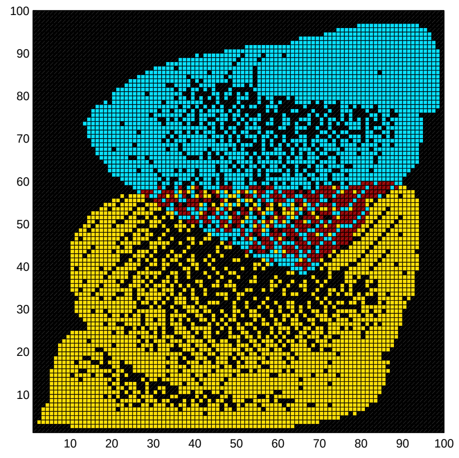
N → *they* | *planes*

Traditional connectionist models, which are deterministic maps from input to output vectors, are in principle unable to deal with this sort of indeterminism. In IRAAM, however, such multiple mappings between strings and structure arise naturally, as a result of the overlap between the images produced by the two (or more) IFS transforms.

As an illustration of this principle, consider the attractor shown in Figure 4.5. This

[6]This sort of ambiguity contrasts with purely *semantic* ambiguity, as in *Our spring was dry*, where it is the two different meanings of *spring*, and not two different phrasal structures, that give rise to the dual meanings.

[7]This grammar overgenerates, but serves as useful example.



11112	((1 1) ((1 1) 2))
11112	((1 1) (1 (1 2)))
11112	(1 ((1 1) (1 2)))
1112	((1 1) (1 2))
1112	(1 (1 (1 2)))
1112	(1 ((1 1) 2))
111212	(((1 1) (1 2)) (1 2))
111212	((1 1) ((1 2) (1 2)))

Figure 4.5: Attractor and sample trees for an IRAAM hill-climbed for maximal syntactic ambiguity. Note the high degree of overlap between the two attractor regions.

figure shows the attractor and eight sample trees for an IRAAM that was hill-climbed to have the maximum ratio between the number of trees and the number of unique strings at the frontiers of those trees: *i.e.*, for the maximum amount of ambiguity[8] The large amount of overlap in the attractor corresponds to a high degree of ambiguity, which is evident even for trees containing only the "pure" addresses *1* and *2*. We can contrast this system with the completely non-overlapping corner terminals obtained in the $a^n b^n$ network, whose "ambiguity ratio" is 1:1; i.e., each string corresponds to a single tree[9]

[8] .

[9]The strings decoded by the network hill-climbed for ambiguity are only a "language" in the trivial sense of being a set of strings. A more realistic approximation of natural language would exhibit both non-regular complexity and syntactic ambiguity at the same time.

Chapter 5

Bias

The previous chapter described an experiment and proof of the ability of an IFS RAAM to decode the language $a^n b^n$ for arbitrary values of n . As discussed in the chapter, the interest of that formal language lies in its structural similarity to a number of phenomena in natural languages such as English. For example, subjects agree with their verbs across arbitrary long distances, including nestings of other subject-verb pairs. This phenomenon is illustrated in Figure 5.1, which shows an example of the the isomorphism between subject-verb agreement and the $a^n b^n$ tree structures produced by the RAAM in the previous chapter[1].

Chomsky [12]used this $a^n b^n$example to argue that a finite state machine (FSM), lacking

[1]Again, although even such relatively shallow nestings as this may not be common in natural languages, their exclusion may be attributed to some principle external to the generative system itself, like short-term memory limitations, pixel resolution, finite stack depth, etc.

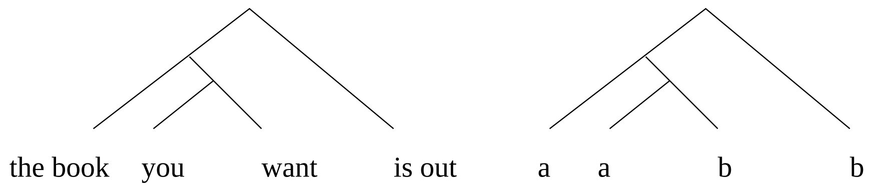


Figure 5.1: Structural relation of subject-verb agreement and $a^n b^n$ trees encoded by an IFS RAAM. Each subject (a) is matched with a verb phrase (b).

memory, is insufficient as a formal model of natural language (NL), and proposed a pushdown automaton (PDA) as the minimally adequate device to capture such phenomena. This observation gave rise to the well-known Chomsky Hierarchy of formal languages and their associate generating and recognizing automata, which led in turn to the question of exactly where in the hierarchy natural language belonged – essentially, whether NL was context-free (CF) or not.

This question was debated vigorously for several years [41][76][77][97], with uncertain outcome. Perhaps the most useful result to emerge from the debate was the realization that the Chomsky hierarchy may not be terribly relevant to natural language. Evidence for this belief came from from psycholinguistic experiments showing that human beings had less difficulty understanding non-CF constructions like crossed serial dependencies than they did understanding the corresponding nested CF constructions [3], as well as linguistic field work reporting the widespread use of non-CF phenomena like reduplication as a productive lexical phenomenon [23].

In light of such evidence, those still interested in the question proposed alternative formalisms in which these observations could be accommodated. Manaster-Ramer [59] suggested using a queue, rather than a stack, as the memory mechanism for a PDA. More formally, Joshi [52]proposed a grammar whose primitives were trees rather than symbols, and showed how the automaton corresponding to this grammar required more steps to parse a nested CF construction than a crossed serial construction, in correlation with the psycholinguistic evidence [51] .

In the present chapter, we wish to present a similar argument using IRAAM, rather than a grammar or PDA, as a formal NL model. Specifically, we present an experiment showing how our IFS RAAM terminal test, in conjunction with the fractal addressing scheme described in the previous chapter, conspires to give the IRAAM model a "bias" toward learning certain configurations more easily than others. In some cases, this bias turns out to agree more closely to the aforementioned NL phenomena than with the complexity measure offered by the Chomsky hierarchy.

5.1 Hypotheses

The experiment was designed to test two hypotheses. The first was that the relative position of symbols is more of a determinant of IRAAM learnability than is Chomsky hierarchy complexity. Based on earlier informal observations of IRAAM behavior, it was hypothesized that strings requiring that a b symbol terminate a left branch or an a symbol terminate a right branch, would be more difficult to learn than strings in which the a was on the left and the b on the right. Second, it was hypothesized that increasing the dimensionality of the representation (*i.e.*, the number of input units) would increase the IRAAM's ability to learn languages of higher Chomsky-hierarchy complexity.

To understand the motivation behind the first hypothesis, consider again the IRAAM terminal-labeling scheme described in the previous chapter. First, the attractor is computed at a fixed sample size; then attractor points (terminals) reachable on the left transform are labeled using one symbol (a), and attractor points reachable on the right transform are labeled with another (b). This scheme might interact with the tree-representing capability of IRAAM in one of three ways: (1) No interaction, in which case some other factor such as Chomsky-hierarchy complexity, and not relative symbol order, should determine learnability: for example, the CF languages $a^n b^n$ and $b^n a^n$ should be equally easy to learn. (2) Complete interaction, resulting in a trivial system in which the label of a terminal is completely determined by which branch it terminates. (3) Some interaction, in which the addressing scheme creates a bias toward learning certain types of ordering, but does is not fully redundant with the tree configuration.

5.2 Experimental design

The experiment was a two-factor analysis of variance (ANOVA) in which the first factor was the training set and the second was the number (one, two, or three) of decoder input units[2], and the dependent variable was the percentage of each training set successfully learned by hill-climbing. As in the hill-climbing experiment reported in the previous chapter, the search space consisted of 4096 samples points chosen uniformly from the input space of three inputs units (16x16x16 samples), two input units (64x64 samples), or one input unit (4096 samples). This scheme allowed us to avoid the artifact of exponential sample-size increase as the number of dimensions went up. Also following the previous experiment, both the initial random weights and the random noise added to each weight came from a Gaussian distribution with zero mean and a standard deviation of 5.0, and training was automatically halted after 30 unsuccessful mutations.

As in the previous experiment, the training sets consisted of sets of 10 strings chosen from $\{a, b\}^*$. Each set was chosen as an exemplar of a particular class in the Chomsky hierarchy. The first set, $a^n b, 1 \leq n \leq 10$, represented regular languages, and the second set, $a^n b^n, 1 \leq n \leq 10$, was the familiar context-free example. The third and fourth sets $a^n b^m b^m a^n$, $a^n b^m a^n b^m$, $1 \leq m, n \leq 4$, were chosen as letter-equivalent sets [90] differing only in belonging to different Chomsky classes. The former is the CF palindrome language, and the latter is the non-CF language similar in form to reduplication in NL. A fifth set, $b^n a, 1 \leq n \leq 10$, served as a test of the trivial case in which trees with a b on the left and an a on the right are unlearnable. The sets are shown in Table 5.1. For each set, ten different training runs were made, each with a different random seed. Some runs produced forests of trees with such great depth and size that they exceeded the memory capacity of our computer [3]; such runs were re-tried with a new random seed until a total of ten completed

[2]Note that the number of network weights scales as the square of the number of input units. A one-input network has four weights, a two-input network has 12 weights, and a three-input network has 24 weights.

[3]Consider the memory requirements for representing 4096 binary trees, each with a maximum depth of

ab	ab	abba	abab	ba
aab	aabb	aabbaa	aabaab	bba
aaab	aaabbb	abbbba	abbabb	bbba
aaaab	aaaabbbb	aaabbaaa	aaabaaab	bbbba
aaaaab	aaaaabbbbb	aabbbbaa	aabbaabb	bbbbba
aaaaaab	aaaaaabbbbbb	abbbbbba	abbbabbb	bbbbbba
aaaaaaab	aaaaaaabbbbbbb	aaaabbaaaa	aaaabaaaab	bbbbbbba
aaaaaaaab	aaaaaaabbbbbbbb	aaabbbbaaa	aaabbaaabb	bbbbbbbba
aaaaaaaaab	aaaaaaaaabbbbbbbbb	aabbbbbbaa	aabbbaabbb	bbbbbbbbba
aaaaaaaaaab	aaaaaaaaaabbbbbbbbbb	abbbbbbbba	abbbbabbbb	bbbbbbbbbba
a^nb	a^nb^n	palindrome	reduplicated	b^na

Table 5.1: Training sets for the bias experiment

	Language	a^nb	a^nb^n	palin	redup	b^na
# Dimensions						
1		10 (0)	7 (3)	1 (1)	3 (1)	4 (4)
2		10 (0)	5 (3)	2 (1)	5 (1)	4 (3)
3		9 (1)	5 (1)	2 (1)	6 (1)	4 (2)

Table 5.2: Experimental results. Entries show the *mean* (*std. dev.*) for the number of strings learned, out of 10, from each language.

runs was obtained for each set.

5.3 Results

Results of this experiment are shown in Table 5.2, which reports the mean and standard deviation for the fraction of each set learned over ten trial runs.

The first thing to note about these results is that the non-zero success rates in the final column of the table allow us to rule out the "trivial" hypothesis in which languages like b^na, with leading b 's, are unrepresentable. The second thing to notice is that there is a main effect of language class on learnability: some languages, like a^nb , are learned almost perfectly, whereas others, like the palindrome language, appear much more difficult. This result is statistically significant, at $p<.001$, and is plotted in Figure 5.2, which shows percent

4096.

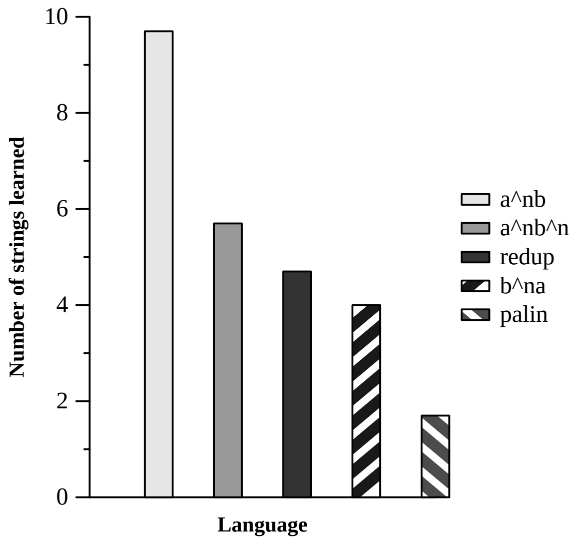


Figure 5.2: Mean number of exemplars learned for each language set, averaged across dimensions.

of targets learned for the languages, averaged over the three dimensonality sizes.

Finally, there was no main effect of dimensionality (number of inputs units) on the success rate. Though the languages $a^n b$, $a^n b^n$, and *palindrome* show a rough correlation between learning rate and number of dimensions, the remaining two languages actually show a *decrease* in the learning rate from two to three dimensions.

5.4 Learnability versus generalization

As with any learning model, ability to cover the training set is only half of the picture. A more interesting issue is how the model generalizes beyond the training set. With respect to the type of experiment described above, this issue breaks down into two questions: First, having "learned" a particular set of strings, does the model generalize the pattern to longer strings? Second, has the model actually learned to generate some particular pattern, or has it merely generated a variety of arbitrary strings large enough to cover the training set?

To explore these issues, we examined the strings generated by each trained network in

	Language	a^nb	a^nb^n	palin	redup	b^na
# Dimensions						
1		2122 (1545)	561 (468)	4 (6)	258 (322)	65 (85)
2		2284 (767)	1147 (1119)	24 (35)	292 (296)	229 (405)
3		2389 (809)	1272 (727)	28 (49)	146 (74)	67 (71)

Table 5.3: Mean (std. dev.) number of strings learned (out of 4096) fitting the pattern of the training set.

	Language	a^nb	a^nb^n	palin	redup	b^na
# Dimensions						
1		660 (1109)	78 (184)	0	0	0
2		213 (230)	1 (2)	0	76 (173)	9 (26)
3		14 (25)	0	0	1 (2)	1 (4)

Table 5.4: Mean (std. dev.) number of grammatical strings longer than the longest string in the training set.

the experiment, and computed two values from the generated strings: (1) the number of strings (out of the 4096 strings generated) that fit the pattern of the training language (2) the number of strings from (1) that were longer than the longest string in the training set. Tables 5.3 and 5.4 show these values.

These two tables reveal a number of things about the way that hill-climbed IRAAMs are generalizing the patterns in the training set. First, Table 5.3 suggests a finer-grained categorization of difficulty for the five languages than was observed from the coverage data in Table 5.2. The regular language a^nb is clearly the most generalized, with more than half of the encoded strings fitting the pattern. Next is the context-free language a^nb^n, followed by the reduplicated language, the regular language b^na, and finally the palindrome language, for which fewer than two percent of the encoded strings are palindromes of the form $a^nb^mb^ma^n$. These averages are plotted in Figure 5.3.

Table 5.4, which presents generalization as the number of grammatical encoded strings beyond the training set, shows the same ordering, with a^nb having a significant number of grammatical strings outside the training set (average 296), and the palindrome language

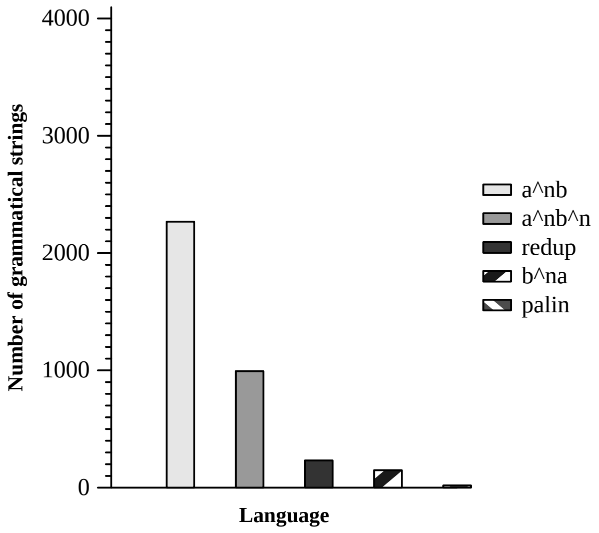


Figure 5.3: Mean number of grammatical encoded strings for each training set.

having none. This table is also interesting in revealing a oft-noted property of neural networks and other learning models: there seems to be an "ideal" number of dimensions (units) for generalizing each language, beyond which the data set is being over-fitted. For the languages a^nb and a^nb^n , generalization falls off after only one dimension; for the reduplicated language $a^nb^ma^nb^m$ and the regular language b^na, generalization appears to peak at two dimensions, though the comparatively small number of strings generalized for these language calls the statistical validity of this phenomenon into question.

5.5 Network dynamics

To explore these results, we plotted the tree equivalence classes generated by two-dimensional networks that had successfully learned all the strings in the training sets a^nb, a^nb^n , and b^na . All three of these plots, shown in Figure 5.4, reveal the striping pattern for the smaller a^nb^n training set from the previous chapter; however, it is instructive to compare the differences. For the the a^nb plot (leftmost in figure), both the a and b attractor blobs are found along one side of the unit square, and there is a single "grain" of striping, from top-right to bottom-

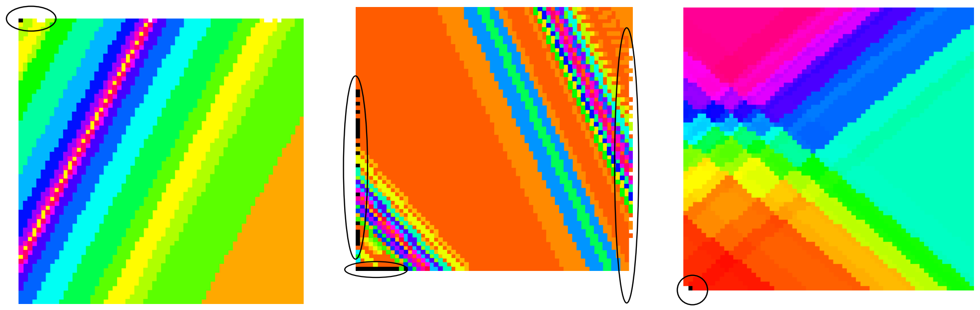

(a) $a^n b$ (b) $a^n b^n$ (c) $b^n a$

Figure 5.4: Tree equivalence classes for successful learners of the languages $a^n b$, $a^n b^n$, and $b^n a$. Attractor points are circled.

left. The $a^n b^n$ plot (middle of figure) shows a someone less regular but similar single-grained pattern, but as in the previous chapter, the attractor points are distributed on opposite sides of the unit square (left and right), supporting the back-and-forth dynamic observed in that chapter. Most interesting is the plot for $b^n a$ (rightmost in figure). Like the $a^n b$ plot, the plot for this regular language shows attractor points concentrated in one part of the space (lower left); however, unlike either of the other two plots, the plot for this language shows two entirely separate striping patterns, crossing at roughly the middle of the unit square.

To understand these patterns, it is helpful to look at the dynamics of the three networks. Figure 5.5 shows the dynamics of the left and right transforms for the the networks, using a “quiver” plot in which the distance traveled by a point is proportional to the arrow’s length. For the $a^n b$ network, there are two simple “ravine” basins of attraction, both pointing to the top of the unit square. The $a^n b^n$ network has two distinct basins of this sort, each pointing to an opposite corner (left or right) at the bottom of the square. The $b^n a$ network, however, reveals a quite different pattern. The dynamics of the left transform are the familiar ravine terminating in the lower left part of the space; however, the dynamics of the right transform show a more complicated saddle-point pattern. Points in the lower-left part of the space are attracted to the lower left corner, but points in the upper-right part of the space will first move toward the lower-right, and then travel along the bottom to the lower left. This bifurcated dynamics can be seen as the network's attempt to reconcile the "natural" shuttling pattern between the left and right sides of the space, with the placement of a terminal in the "wrong" part of the space: essentially, the $b^n a$ network is trying to be the $a^n b^n$ network, but with both terminals jammed into a single side.

5.6 Induction

As described in an earlier chapter, the component of IRAAM that corresponds to recursion in a grammar, or the inductive step in an inductive proof, is the pixel resolution at which the D-dimensional attractor and tree space $[0,1)^D$ are sampled. Just as a fractal image like the Mandelbrot set will reveal smaller copies of itself as the pixel resolution is increased ("zooming in"), increasing the sampling resolution of the IRAAM should reveal more detailed patterns, in the form of longer well-formed strings of the language.

To test this hypothesis, we performed one more analysis of the weights generated by hill-climbing. This time, we increased the number of sample points to 15,625 (i.e., 125x125 for the two-dimensional network, 25x25x25 for the three-dimensional), and generated trees at this resolution using the networks trained at 4096 sample points. For a given set of weights, the question was how many new grammatical strings the network would generate at the higher resolution, as compared with the number that same network generated at the lower resolution[4]. These values are given in Table 5.5.

The values in this table follow the general pattern observed so far: the $a^n b$ and $a^n b^n$ sets are induced best (as an inverse function of the number of dimensions), with the remaining three sets being induced poorly[5]. The results do, however, support the notion of resolution

[4]It would also be possible to ask how many grammatical strings outside the training set were generated at the higher resolution, but that approach does not as directly address the question of how the capabilities of a particular set of weights change as resolution is increased.

[5]The high mean for the two-dimensional network learning the reduplicated language is an artifact of a single grammatical string being over-represented in the induced string set.

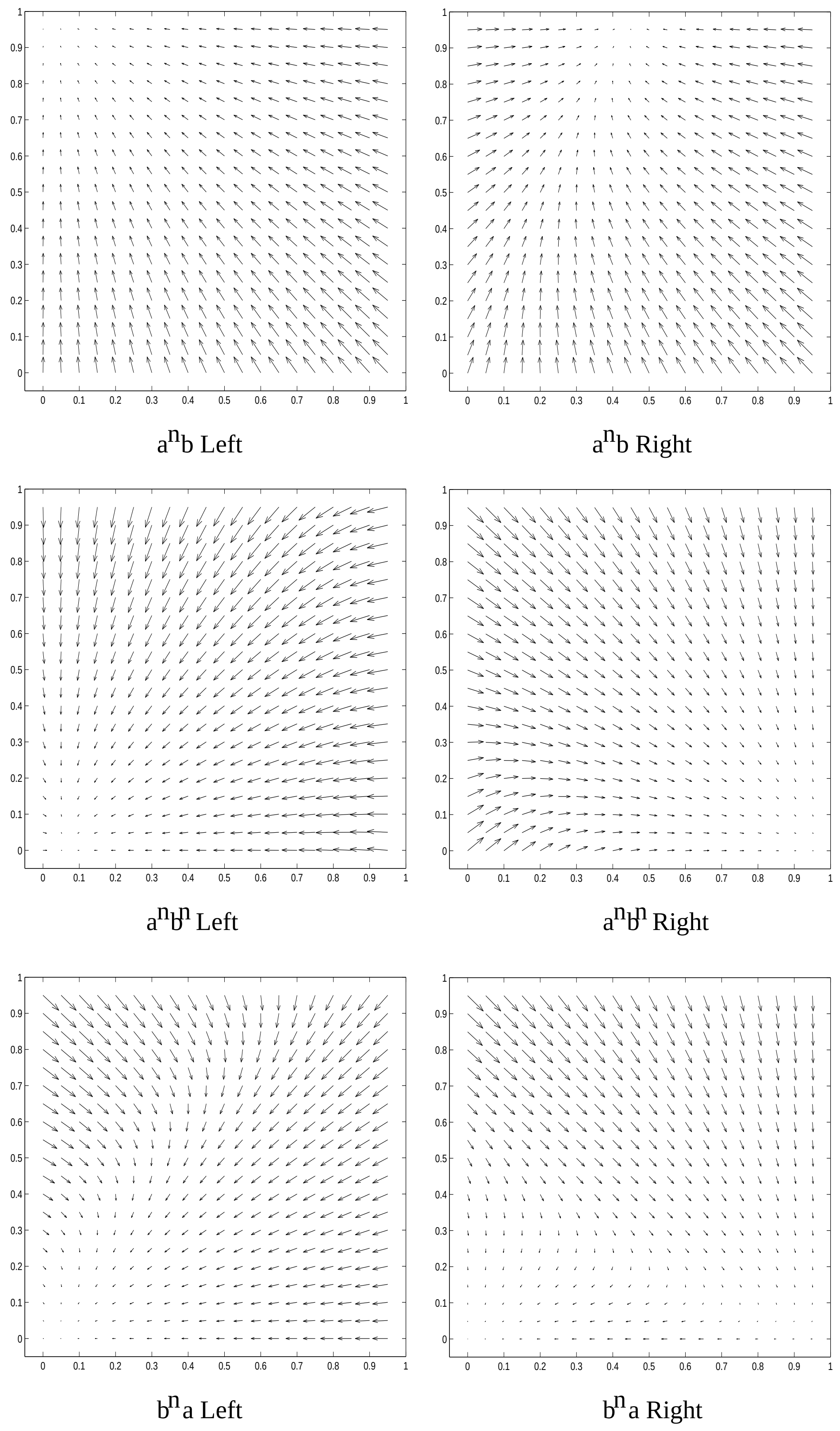


Figure 5.5: Dynamics for successful learners of the languages $a^n b$ (top), $a^n b^n$ (middle), and $b^n a$ (bottom). Left transforms are on left side of figure, right transforms on right side. Length of arrow is proportional to distance traveled by point at base of arrow, on one application of transform.

	Language	a^nb	a^nb^n	palin	redup	b^na
# Dimensions						
1		2129(3512)	32(38)	1(3)	1(1)	1(2)
2		390(513)	16(45)	18(54)	423(1245)	2(5)
3		2(4)	13(38)	0(1)	3(4)	0(1)

Table 5.5: Mean (std. dev.) number new grammatical strings induced at higher resolution .

as induction: if an IRAAM learns a language well at one resolution, it is likely to generate further, longer exemplars from that language at a higher resolution.

5.7 Cross-induction

Another interesting question to examine is the extent to which a decoder trained on a given language set can generalize to other language sets. To investigate this issue, we took the set of weights obtained for the two-dimensional networks and tested each set of weights on the complement of the language on which it was trained. Table 5.6 shows the results of this experiment.

The cross-induction results follow a similar pattern to the other experiments: decoders trained on a^nb, a^nb^n, and *redup* generalize the best across all languages, with *palin* and b^na tied for last place. Essentially, the a^+b^+ sub-pattern is being reused successfully for those first three languages, supporting generalization to strings containing that pattern, regardless of the language class to which the strings belong. A somewhat surprising result is that decoders trained on these three "easy" languages do better on *palin* than do the decoders trained on the *palin* language itself. A possible explanation for this result is the weakness of the hill-climbing method used to learn the weights. It may be that the palindrome training set used here provides such a poor gradient, that a network producing arbitrary sequences of a's followed by b's does better at covering this set than a network trained on the set itself.

	Language	a^nb	a^nb^n	palin	redup	b^na	Overall
Weights							
a^nb		10 (0)	5.4 (2.6)	3.5 (2.4)	5.9 (3.0)	2.6 (0.8)	5.5 (2.9)
a^nb^n		1.6 (0.8)	5 (3)	2.3 (1.1)	2.6 (1.2)	2.1 (0.9)	2.7 (1.3)
palin		0.1 (0.3)	0.3 (0.7)	2 (1)	0.8 (1.0)	2.2 (1.8)	1.1 (1.0)
redup		2.5 (2.3)	1.7 (1.5)	2.3 (1.3)	5 (1)	2.0 (1.3)	2.7 (1.3)
b^na		0.1 (0.3)	0.1 (0.3)	0.9 (1.2)	0.6 (0.8)	4 (3)	1.1 (1.6)

Table 5.6: Cross-induction results. Each row represents 10 sets of weights hill-climbed on a particular training set. Entries show mean (std. dev.) for number of strings (max=10) from other training sets successfully decoded by these weights. Success on identical sets is also reproduced from Table 5.2 for comparison.

5.8 Discussion

Though this chapter has by no means presented an exhaustive analysis of the formal linguistic capability of Infinite RAAM, some important tendencies of the model are now evident. These tendencies are best understood by comparing IRAAM with a traditional formal grammar. In a formal grammar, the labels of the terminal are entirely arbitrary; there is no meaningful difference, *e.g.*, between the languages a^nb^n and b^na^n . Complexity is determined by the memory apparatus required to parse the language (machine model), which corresponds to the relative number of symbols permitted in the left- and right-hand sides of grammar rules. Such a model fails to predict the scarcity in natural language of patterns such as palindromes, which have a lower complexity (context-free) than widely attested patterns like reduplication (context-sensitive). Cross-linguistic patterns, such as the universal tendency to put subjects before objects in declarative sentences [40], must be coded independently into the grammar of each language, or must be derived by stipulating such principles as part of the "Universal Grammar" that children have as part of their genetic heritage [15]. This situation necessarily follows from the hypothesis of the the autonomy of syntax [14], in which structure and meaning are handled by two completely separate model components.

Infinite RAAM, by contrast, makes no such distinction. Indeed, both the identity of a symbol and the structures in which it participates are determined by the same mechanism, the transform weights of a non-linear iterated function system. The difference comes from whether the transforms are applied to the IFS attractor, yielding terminal equivalence classes (roughly, semantics), or to the complement of the attractor, yielding tree-shaped transients to the attractor (roughly, syntax). The result is a *bias,* or statistical tendency, toward putting a given equivalence class ("part of speech") on the branch corresponding to the transform used to label that class. This tendency results in certain languages being more difficult to learn than others, despite their belonging to equivalent classes in the Chomsky Hierarchy; in fact, relative word order, as opposed to relative position in this hierarchy, is a better prediction of IRAAM learning success, as it is perhaps for natural language.

Two caveats are "in order" here. First, simple languages consisting of a 's and b 's are far from adequate as models of natural language. Generative grammar as a field has had nearly a half century in which to investigate how simple formal grammars can be scaled up to describe the tremendous richness and variety of natural language.; accounting for more than a tiny fraction of the phenomena that that field addresses is beyond the scope of the present thesis. Second, hill-climbing is by no means an optimal learning method, and the parameters chosen for our hill-climbing experiment (network size, sampling resolution, initial conditions, mutation rate, maximum attempts before failure) have by necessity restricted us to exploring an infinitesimal part of the search space of IFS weights.

Nevertheless, the results obtained in this experiment do suggest that the lion's share of the "grammar space" of IRAAM is occupied by systems having the ordering bias. If valid, this hypothesis puts IRAAM in a rather unique position among cognitive models, including traditional grammars and most connectionist alternatives to them. Unlike the former, IRAAM derives syntax, semantics, and their "semi-autonomy" from a single underlying principle. Unlike the latter, IRAAM embodies a strong tendency toward certain linguistic patterns, independent of training data or initial conditions. Together, these properties go a long way toward satisfying the criteria presented in the Introduction: understanding what a model is doing, and thereby gaining insight into its applicability.

5.9 Supporting a generative lexicon

A major trend in current linguistic theory is a shift toward the lexicon as the locus of syntactic knowledge, notably Lexical Functional Grammar [24], Head-Driven Phrase Structure Grammar [74], and the Generative Lexicon model of Pustejovsky [79]. In contrast to earlier theories that employed subcategorization and other constraints to "clean up" the over-generation of powerful phrase structure rules [14], these recent approaches put much or all of the combinatorial information about a word into that word's entry in the lexicon. A striking difference between the old and new approaches can be found in Pustejovsky's treatment of nominals, which instead of being simple atoms acted upon by verbs, have a rich compositional structure of their own, specifying the semantics (or *qualia*) of their interaction with verbs and with other nouns.

The relevance of this linguistic research to the present discussion lies in the unified account that IRAAM provides for syntactic composition of lexical items. The same set of decoder network weights that we use to label the attractor terminals, determines how they combine with other terminals to form tree structures. This mechanism applies equally to all terminals, regardless of their "phrasal category" (position in the tree)[6]. Therefore, in a very meaningful sense, every item in the IRAAM lexicon comes with a built-in specification of its own combinatorics, very much in the spirit of the lexico-centric approaches described above. Furthermore (and unlike associative models in which new a network connection must be added for every relation [96]), IRAAM embodies this knowledge in a fixed-size network.

Obviously, there is a great deal of work to be done before the IRAAM model can begin to support even the rudiments of the generative lexicon framework, and it is not at all clear how

[6]The following chapter explains how phrasal structure can be induced over lexical items in a "bottom-up" fashion.

IRAAM might represent powerful mechanisms like type coercion that give that framework so much of its appeal. Nevertheless, we have reason to consider IRAAM a promising first step in providing a fully connectionist account of lexical phenomena.

Chapter 6

Learning

So far, we have shown that IRAAM networks are capable of representing a non-trivial infinite formal language, and that the addressing mechanism by which the terminal symbols of this language are enumerated can be extended to provide an infinite lexicon. We also have an algorithm that will tell us whether, and to what extent, a given IRAAM network sampled at a given pixel resolution encodes a given fixed-arity tree. Finally, we have some intuition about the interaction of the lexical addressing scheme with the structures of the trees themselves. The obvious question to ask is whether these capabilities can be exploited to devise an algorithm which, given a lexicon and a set of trees over that lexicon, will arrive at a set of weights to encode those trees, for a particular address length and pixel resolution.

In attempting to devise such an algorithm, we are confronted immediately with the choice of whether to use only the decoder by itself, or try to exploit original RAAM's success in learning, via the encoder/decoder autoassociative model. As described earlier, IRAAM essentially does away with the encoder, by using the decoder inverses to determine whether a particular tree is encoded. Nevertheless, without the original RAAM encoder it is not clear how to use back-propagation to train the decoder by itself.[1]

[1] Melnik [66] reports some success using gradient descent to train a RAAM decoder on various attractor shapes, but has not been able to generalize the algorithm to learning trees [63]. We have had only limited

The present chapter reports four experiments using back-propagation to train a small RAAM network on a data set from Pollack's original RAAM paper [73]. The point of the experiments is to compare various terminal representations, including some derived from the RAAM's decoder network, to understand how the attractor principle can be exploited to improve learning. The data set comes from a simple context-free grammar for a subset of English, using five parts of speech:

S $\rightarrow$ NP VP | NP V

NP $\rightarrow$ D AP | D N | NP PP

PP $\rightarrow$ P NP

VP $\rightarrow$ V NP | V PP

AP $\rightarrow$ A AP | A N

From this grammar we extracted the following derivations for use as a training set, following [73]:

(D (A (A (A N))))

((D N) (P (D N)))

(V (D N))

(P (D (A N)))

((D N) V)

((D N) (V (D (A N))))

((D (A N)) (V (P (D N))))

We conducted the following four experiments on various subsets of this training set.

success in extending our $a^n b^n$ hill-climbing method beyond formal languages.

	Corners	Middle
RMS error	0.119105	0.078502
Max error	0.543962	0.277351

Table 6.1: Mean errors for corner and middle terminals. Differences are significant for max error ($p < .01$), not for RMS error.

6.1 Extreme versus general terminals

Noelle et al. [67] provide a rather exhaustive study of a fully recurrent (Hopfield) network, demonstrating the usefulness of "extreme" attractor targets from the perspective of learning capacity, accuracy, and learning speed. They show that bit vectors – *i.e.*, targets with points in the extrema (corners) of the unit hypercube – yield better performance than general real-valued vectors in all three respects [2].

To explore this issue in a RAAM context, we chose the subset of the training set containing the terminals D, A, N, and V, so we could put one terminal in each corner. We compared this configuration with one in which these terminals were located at a Euclidean distance of 0.3536 from the corners; that is, at a distance of 0.25 from each axis. Starting with 10 different initial random 4x2x4 networks, we trained each network for 10,000 epochs, a number determined empirically by observing how many epochs it took for the error to stop decreasing significantly.

6.1.1 Results

Table 6.1 reports the mean RMS and maximum errors (maximum over any pattern) for the corner and "middle" terminal sets. As the table shows, both errors were higher for the corner terminals than for the middle terminals, although the results are only statistically significant ($p < .01$) for the max error means. Representative attractors are plotted in figure 6.1

[2]Their analysis of this results rests on the fact that a sigmoidal activation function has a maximum of two stable fixed points

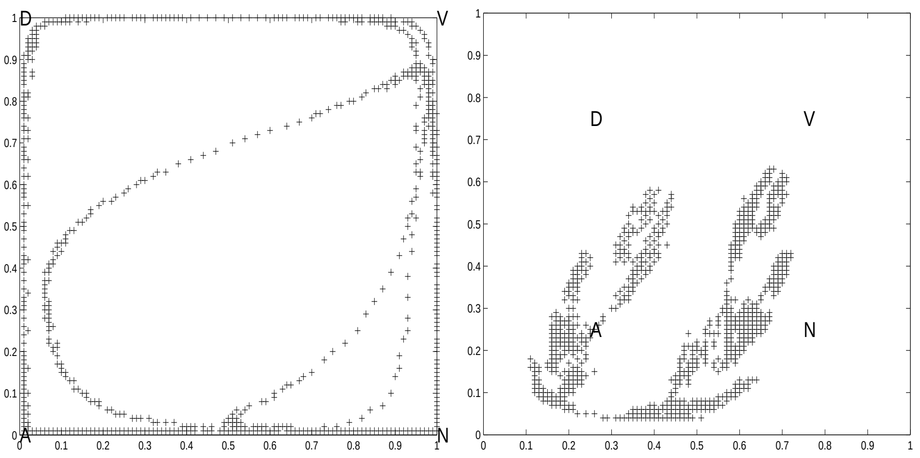


Figure 6.1: Representative attractors for corner and middle terminals

6.1.2 Discussion

Our results disagree with those of Noelle et al., in that we found no significant benefit to putting the terminals in the corners of the space, as far as the error was concerned. In fact, our errors were lower for the terminals placed toward the middle of the space.

The most obvious explanation for this result is that RAAM, unlike the recurrent network used by Noelle et al., uses general real-valued hidden-layer vectors, and not just the terminal vectors, as targets, in order to represent non-terminal trees. Hence, RAAM is solving an additional set of constraints, which may be considered to favor non-extreme values overall.

6.2 Moving versus fixed terminals

A focus of this thesis has been the advantages yielded by treating the RAAM decoder as a dynamical system in which the terminals lie on the attractor. Hence, we might hypothesize that learning would be facilitated by moving the terminals onto the attractor after each training epoch, or some fixed number of training epochs, making the terminals into "moving targets" like the non-terminals. This experiment tested that hypothesis.

	Fixed	Moving
RMS error	0.132947	0.016862
Max error	0.428662	0.060375

Table 6.2: Mean errors for fixed and moving terminals. Differences are significant at $p < .001$.

Using the entire seven-phrase training set, we started with a random "dictionary" of terminals dispersed throughout the unit square. As in the previous experiment, we started with 10 different random 4x2x4 RAAM networks. For each such network, we compared the errors obtained by leaving the terminals in a fixed position against the errors obtained by moving each terminal onto the attractor point closest to it (measured as minimal Euclidean distance) after every 100 training epochs. Conflicts (more than one terminal mapping to the same attractor point) were resolved by putting terminals on the next-to-closest point.

6.2.1 Results

Table 6.2 reports the mean RMS and maximum errors for the fixed and moving terminal sets. As the table shows, both errors were much higher for fixed terminals than for the moving terminals. This is not the whole story, however: As Figure 6.2 shows, the very low errors for the moving terminals were gained at the expense of a very small attractor onto which all the terminals "collapsed" during training. Apparently, having all the terminals on the same point or small set of adjacent points supported a "solution" in which all the targets (including hidden-layer targets) were nearly identical. In other words, the error could become arbitrarily low by having all targets be nearly the same.

6.2.2 Discussion

For a random set of RAAM decoder weights, the (pseudo-) contractivity of the decoder transforms results in an attractor that is generally a small, uninteresting set of points. A larger, more space-filling attractor can be obtained by hill-climbing, a Blind Watchmaker

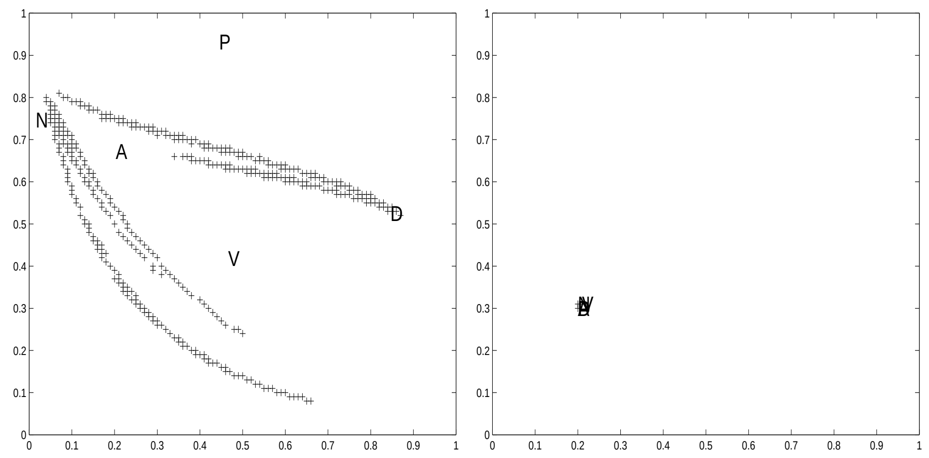


Figure 6.2: Representative attractors for fixed and moving terminals. Left side (fixed) shows the configuration of the initial conditions for the moving terminals, which all end up in a region near the point (0.2, 0.3) in the right-hand figure.

algorithm, or back-propagation to a fixed, dispersed terminal set, but unless there is some such pressure on the system, contraction into a small set of points appears to be the default behavior of the decoder.

Even with a dispersed target set, though, a two-dimensional network appears incapable of learning a non-trivial test set like the one we used in this experiment. This result is not surprising, given that Pollack [73] used ten-dimensional representation for this problem.

The point of these experiments was not, however, to replicate Pollack's results on a much smaller network, but rather to use the visualization afforded by a two-dimensional network to discover useful properties of the learning problem based on the attractor principle. The left-hand side of Figure 6.2 suggests that the learning algorithm is attempting to "cover" the terminal set with the decoder's attractor, and failing. Interpreting this two-dimensional image as the projection of the attractor of a hypothetical, higher-dimensional network, we can imagine a solution in which the terminals are much closer to each other along some dimensions than others – i.e., a solution incorporating the features of both the moving- and fixed-target approaches to arrive at a low error without collapsing all the terminals into a single, infinitesimal ball. The next experiment describes how we might develop such a solution.

6.3 A hybrid approach

The dilemma raised by the previous experiment – low error with inadequately distributed terminals versus distributed terminals with high error – relates to problems encountered in other fields that use gradient-descent, and search methods in general. The weights discovered by back-propagation on distributed terminals represent a local optimum in the weight space. Simulated annealing [57] is a well-established approach to gradient-descent problems that attempts to avoid such local optima by the addition of noise. The amount of noise decreases with time, following an "annealing schedule" analogous to the cooling process for annealing of metals and other materials. More recently, research in genetic programming has considered innovative ways of dealing with the loss of diversity in a population [49], which can likewise lead to local optima in the search space.

In this experiment, we borrowed ideas from both of these approaches. Specifically, we kept as a goal the maintenance of spatial diversity in our terminal set, while also using a schedule of parameter changes to avoid getting stuck in a local optimum. Starting with the terminal set from the previous experiment, we trained a 4x2x4 RAAM on the entire seven-tree training set, and moved the terminals onto the attractor only after the error leveled off. This movement was done "by eye," based on observation of the two-dimensional attractor image, with a view toward keeping the terminals spaced out along the attractor.

6.3.1 Results

The mean RMS error for this experiment, computed over 10 random initial networks, was 0.08948. The maximum error was 0.303282. The useful comparison to make is with the

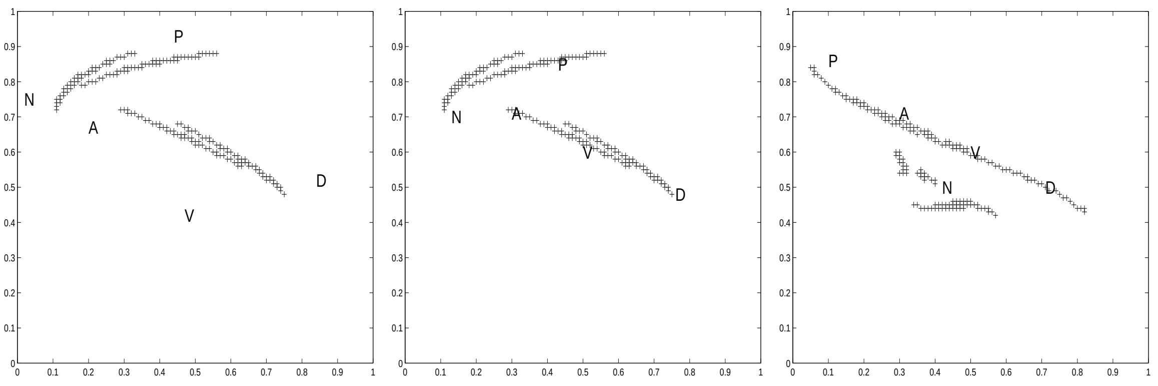


Figure 6.3: Moving terminals onto the attractor by eye. (a) Attractor and terminals after 1000 back-prop epochs. (b) Terminals moved onto attractor points. (c) Attractor and terminals after 10,000 epochs

corresponding means for the fixed-terminal networks from the previous experiment. For both RMS and max error, the means for the current experiment were significantly lower ($p < .001$).

6.3.2 Discussion

The results of this experiment strongly support the usefulness of the attractor-as-terminal concept first discussed in Chapter 3. Moving the terminals onto the attractor on an “annealing schedule” supports diverse terminals as well as significantly lower errors, as compared with permanently fixed terminals of the sort used by Pollack in his original RAAM experiments.

However, the errors reported in the present experiment are still much too high to support successful decoding of the trees in the training set. Furthermore, the approach taken here requires human supervision not just for monitoring the error decrease, but also for moving the terminals onto the attractor. The following section describes how these problems may be overcome.

6.4 Scaling up with a "pseudo-attractor"

The most common way to deal with intolerably high error rates in a neural network is to add more hidden units. Unfortunately, for our IRAAM network, this means increasing the dimensionality the attractor to be computed. The ATTRACTOR algorithm of Chapter 3, which samples the entire unit square at $S \times S$ pixels, rapidly becomes impractical as we move to the unit cube, the unit hypercube, etc.: for D dimensions, the number of such pixels increases exponentially as S^D. This leaves us in the position of either abandoning the attractor principle when using higher-dimensional networks, or devising a way of approximating the attractor that will scale up more readily.

One possible way to overcome this "curse of dimensionality" [5] would be to approximate the attractor by decoding each tree to its terminals, and using this decoded set of terminal points as the attractor. In this approach, each terminal will have a point (vector) for every place that it appears in a tree in the training set. The mean of each such set could then be used as the attractor point to which the terminal should be moved, once the error fails to drop significantly during training[3]. Given that the "true" attractor is by definition the set of points to which all non-terminal tree vectors eventually decode, this approach can be seen as an attempt to focus on the subset of trees that we are interested in learning.

To explore this possibility, we trained networks of two, four, and eight dimensions on our seven-phrase English-grammar data set[4]. Starting with randomly distributed terminals, we trained each network for 10,000 epochs, moved the terminals onto the pseudo-attractor, and trained for another 10,000 epochs, repeating until the error bottomed out.

[3]Using the mean is equivalent to putting an epsilon ball around the set of points and computing the centroid of the ball

[4]Given that Pollack [73] was able to learn this training set with a ten-dimensional network using fixed terminals, it made little sense to go beyond that number of dimensions in the present experiment.

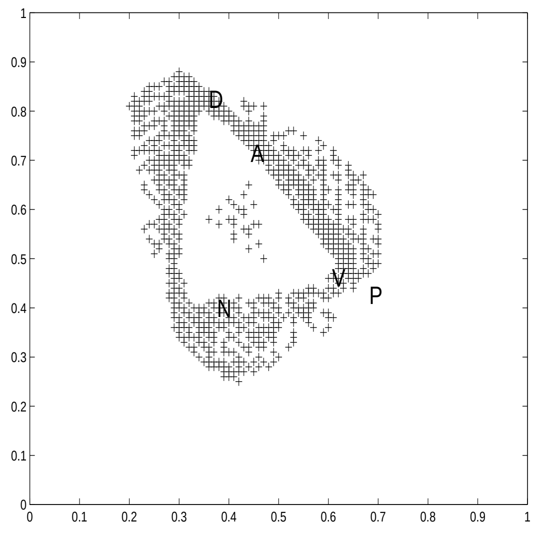


Figure 6.4: Terminals on "pseudo-attractor," with actual attractor in background.

6.4.1 Results

The main result of this experiment was that by trying different random initial dictionaries and networks, we were able to obtain an eight-dimensional solution to the problem of learning and decoding the seven syntactic trees. This compares favorably with the 10-dimensional solution reported by Pollack [73] for the same problem using a fixed terminal dictionary.

We found two noteworthy phenomena in this experiment. First, as shown for the two-dimensional example of Figure 6.4, the pseudo-attractor points tended to fall on or near the "true" attractor of the RAAM decoder. Second, as suggested by this figure, success in decoding depended largely on there being enough distance between terminals to avoid confusion between one terminal and another. In general, the more successful networks had a larger average Euclidean distance between terminals.

6.4.2 Discussion

This experiment further validates the attractor-as-terminal approach, by allowing us to use smaller networks than in the fixed-terminal examples. It also points toward an even more radical version of RAAM, in which not just the non-terminal representations, but also the terminal representations, are “moving targets,” albeit at a much slower rate. Finally the possibility (or even desirability) of learning with real-valued terminal vectors instead of one-in-N codes or other “extreme” representations, opens the door to interaction between RAAM and variety of of recent semantic models that use such vector representations[5].

In conclusion, the results in this chapter are a successful validation of the IFS interpretation of RAAM. Using the IFS attractor provides significantly lower error rates in learning, when compared to the use of (arbitrary) bit string terminal representations. This result held in even the one “failed” experiment, in which automatic movement of terminals onto the attractor resulted in a lack of distinction among terminals (a result that we had not anticipated). That result suggests, however, that a separate principle is needed to initialize and/or maintain the terminal vectors during learning. Our final chapter concludes with a proposal for how that might be accomplished.

[5]We provide a few examples of such models in our final chapter.

Chapter 7

Application: A Neurally Plausible Language of Thought

This chapter presents examples of our application of the IRAAM model to selected problems in cognitive science. Our goal here is not to convince the reader of the usefulness or desirability of the model as a practical technology for real-world AI problems. Instead, we aim to illustrate how IRAAM provides a principled, general connectionist basis for cognitive representations, and to do this by applying the model to problems that have traditionally been considered the exclusive domain of symbolic rule-based approaches.

7.1 Background: Unification

Unification, a pattern-matching algorithm popularized by Robinson [82] as a basis for automated theorem-proving, is at the core of logical programming languages like Prolog, as well as a number of recent models of natural language [98]. The basic unification algorithm can be found in many introductory AI textbooks (e.g., [81] p. 152), and can be summarized recursively as follows:

(1) A variable can be unified with a literal.

(2) Two literals can be unified if their initial predicate symbols are the same and their arguments can be unified.

If, for example, we have a Prolog database containing the assertion `male(fred)`, meaning "Fred is male", and we perform the query `male(Who)`, asking "Who is male?" the unification algorithm will first attempt to unify `male(fred)` with `male(Who)`, and will succeed in matching on the predicate symbol `male`, by rule (2). The algorithm will then recur, attempting to unify the variable `Who` with the atomic literal `fred`, and will succeed by rule (1) and terminate, with the result that `Who` will be bound to `fred`, answering the query.

Looked at another way, unification amounts to answering the question "does this tree appear in the current database?"[1] or "can this tree be generated as a derivation in the current grammar?" From the perspective of an IRAAM, these questions can be expressed as follows: For a particular set T of network transforms, a pixel resolution S, a number of address digits N, and a given tree $\mathcal{T}$ over the N-digit addresses, does the tuple $<T,S,N>$ encode $\mathcal{T}$?

7.2 Unification as intersection of inverses

To understand how an IRAAM provides an efficient model of the unification algorithm, recall the IRAAM encoder definition from Chapter 3: the encoder is simply the mathematical inverse of the decoder. For example, to determine the code (equivalence class of vectors) for a tree $(A\ (B\ \ C))$, we take the set of attractor points whose address is A, compute their left inverse[2], take the attractor points whose address is B, compute their right inverse, in-

[1]Prolog terms have a natural interpretation as trees, with the functor at the root and the arguments in the branches (c.f. [9], p.33)

[2]Because the inverse transforms are expansive, we cannot simply compute the bijective inverse transfer function on this set of points, which would give us only a subset of the actual inverses. Instead, for a given point, we compute the set of points that go to that point on the given transform, and store this set as the

tersect these two inverses, and remove from this intersection any points that are also on the attractor. This gives the points for the tree $(B\ C)$. Taking the right inverse of these points, intersecting this inverse with the left inverse of the attractor points for A, and removing points also on the attractor, gives us the points for the target tree $(A\ (B\ C))$. If this set of points is empty, the answer to our query is negative; otherwise, it is affirmative. In general, for a given tree $\mathcal{T}$, IRAAM $<T,S,N>$, and its addressed attractor A, the following algorithm returns the points in the (possibly empty) "unification set" of $\mathcal{T}$:

```
UNIFY($\mathcal{T}, T, A, S, K$)
1  **if** IS-TERMINAL($\mathcal{T}$)
2     **then** $V \leftarrow$ TERMINAL-VALUE($\mathcal{T}$)
3           $< i, j > \leftarrow$ POINTS-W ITH-VALUE($A, V$)
4           **return** $A_{<i,j>}$
5     **else** $B \leftarrow$ ONES-MATRIX($S \times S$)
6           **for** $k \leftarrow 1$ **to** $K$
7              **do** $\mathcal{U} \leftarrow$ BRANCH($\mathcal{T}, k$)
8                 $C \leftarrow$ UNIFY($\mathcal{U}, T, A, S, K$)
9                 $C \leftarrow T_k^{-1}(C)$
10                $B \leftarrow B \wedge C$
11          **return** $B - A$
```

point's inverse. This need only be done once for each transform, after which the inverses are stored in a lookup table.

The function POINTS-WITH-VALUE(A,V) in line 3 returns all and only those tuples $<i,\ j>$ such that $A_{i,j} = V$. This is how the points corresponding to a terminal tree (the base-case of the recursion) are computed. For non-terminal trees, line 5 initializes the points to the entire unit square, and reduces this set by recurring on the branches of the tree (lines 7,8) and intersecting with the inverses of the points corresponding to the branches (lines 9,10). Line 11 removes the attractor points from the non-terminal points thus computed.

A couple of aspects of this unification algorithm merit comment. First, far from being a connectionist "implementation" of an existing classical algorithm [36], IRAAM unification is an entirely novel approach to modeling this sort of "rule-governed" phenomenon *without explicit rules,* or the intermediary representation of a Turing machine or other traditional architecture [99]. Furthermore, the basic operations of our algorithm (inversion and intersection) are inherently parallelizable, a feature that has been cited as one of the main appeals of connectionist models [62].

Second, the algorithm returns not just a yes-or-no truth value, but a set of equivalence-class regions representing partitions of the unit square. By computing the area of these partitions, it is possible to represent truth values as intermediate between zero and one. These "grayscale" values lend themselves naturally to an interpretation as degree of belief or grammaticality [54], corresponding more closely to our intuitions about the non-discrete nature of such phenomena.

7.3 Application to a Language of Thought

As discussed in earlier chapters, Fodor's "Language of Thought" view has been the dominant paradigm in cognitive science over the past quarter century, reflected in the artificial intelligence community by the popularity of logic-programming languages such as Prolog [18]. Though not committed to any particular formalism, Fodor has argued for a rules-and-representations approach in the spirit of Prolog, using discrete symbolic representations that are manipulated by structure-sensitive rules. As noted in Chapter 2, he sees this view as fundamentally at odds with connectionism, based on the latter's putative inability to build systematic, compositional representations and operate on those representations in a meaningful way [34]. Given the ability of our IRAAM model to perform rule-like operations (such as unification) without explicit rules, we consider IRAAM to be a viable candidate for addressing Fodor's objections to connectionism.

7.3.1 A simple example

To see how the IRAAM unification model addresses Fodor's objections, we consider a simple example of a Prolog database containing the following assertions and rule: [3]

```
gender(fred, male).
married(fred, wilma).
(resents(barney, (married(fred, wilma)).
husband(Y, Z) :- male(Y), married(Y, Z).
jealous(X, Y) :- resents(X, married(Y, Z)).
```

Ignoring the *husband* and *jealous* rules for the time being, we can represent the facts in this database using ternary trees, as follows:

```
(gender fred male)
(married fred wilma)
(resents barney (married fred wilma))
```

Starting with a random 2D dictionary, we trained a 6x3x6 RAAM network to represent

[3]Prolog uses rules in Horn clause form, with implicit universal quantification, upper-case letters for variables, and lower-case letters for constants. Hence, the rule shown here corresponds to the predicate calculus formula $\forall x, y$ gender(x,male) $\wedge$ married(x, y) $\rightarrow$ husband(x, y).

these three trees, using the moving-terminals-by-eye technique described in the previous chapter. We ran 5000 training epochs, moved the terminals onto the attractor, and ran 5000 more epochs, to an RMS error of approximately .003. This yielded the attractor shown in the left side of Figure 7.1. As a representation of the terminal "dictionary," an epsilon ball with a radius of 0.05 was put around each point, resulting in the map shown in the right side of the figure.

With the decoder weights and attractor dictionary thus obtained, we can now use the IRAAM unification algorithm to query the database, as we would in Prolog. Consider, for example, the simple interaction:

```
| ?- gender(fred, male).
yes
| ?-
```

Prolog answers this query in the affirmative by searching through the database and finding an assertion that unifies with the query. In IRAAM, the query corresponds to finding the unification set of the tree `(gender fred male)` – in other words, finding a region of space corresponding to the set

$$T_1^{-1}(\mathcal{G}(gender)) \;\wedge\; T_2^{-1}(\mathcal{G}(fred)) \;\wedge\; T_3^{-1}(\mathcal{G}(male))$$

where $\mathcal{G}$ is the "symbol grounding" function that returns the set of points corresponding to the named terminal symbol [4]. As shown in Figure 7.2, this set is non-empty, so the answer to the query is affirmative.

In addition to returning a yes or no answer, Prolog's unification algorithm will return all possible bindings of a variable if the answer is affirmative:

```
| ?- married(fred, Who).
```

[4]This function was called POINTS-WITH-VALUE in the UNIFY algorithm.

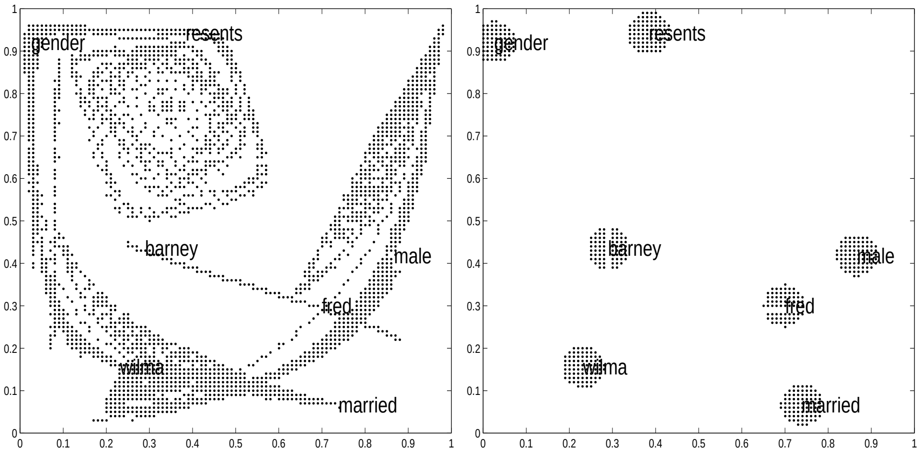


Figure 7.1: Attractor and epsilon-ball terminals for the Fred/Wilma example.

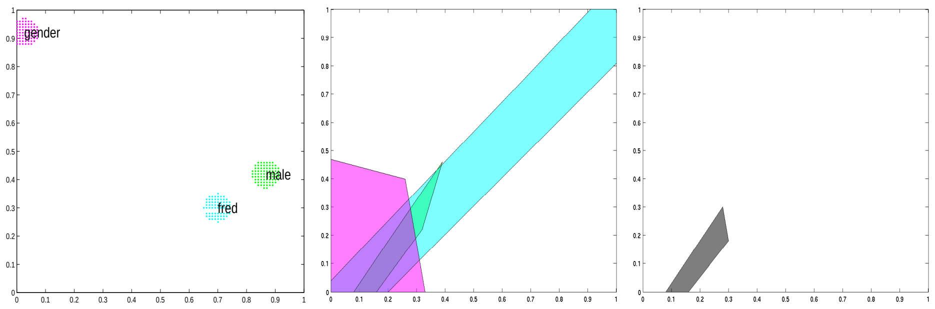


Figure 7.2: Terminals, inverses, and intersection for the query `gender(fred, male)`

```
Who = wilma

yes
```

The IRAAM version of variable binding consists of four steps:

1. computing the inverses of the constants in the query (`married`, `fred`)
2. computing the intersection of these inverses
3. taking the appropriate *forward* transform of that intersection
4. determining which (if any) terminal regions intersect with the region computed in (3)

Of course, the power of a logic-programming language like Prolog comes from its ability

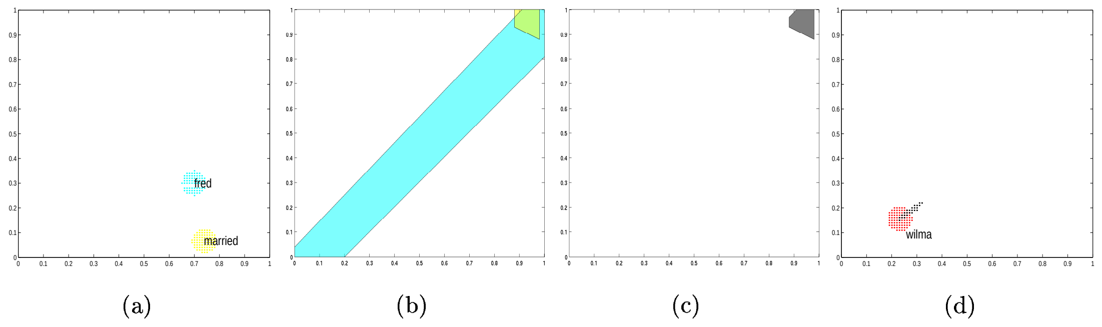


Figure 7.3: Computing the answer to the query `married(fred, Who)`. Intersection (c) of inverses (b) of `married` and `fred` (a) is put through third forward transform, which intersects the region for `wilma`, answering the query (d).

to do rule-based inference. For example, if we issue the query `husband(Who, wilma)`, Prolog will perform the following sequence of actions:

1. Unify the head of the rule `husband(X, Y) :- gender(X, male), married(X, Y)` with `husband(Who, wilma)`, resulting in the sub-goals `gender(X, male)` and `married(X, wilma)`.

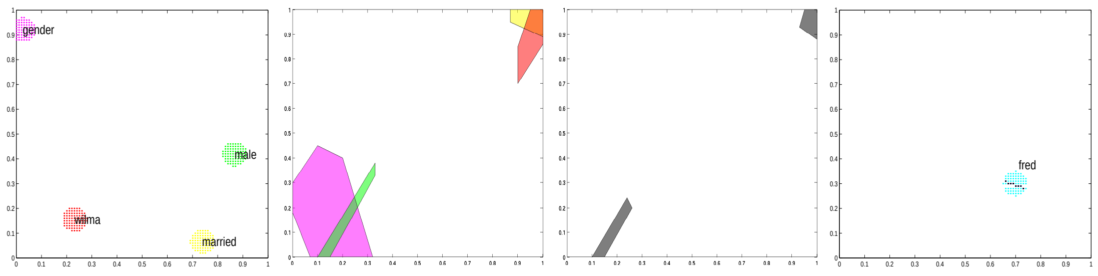
2. Unify the sub-goal `gender(X, male)` with the given `gender(fred, male)`, resulting in the binding `X = fred`

3. Unify the sub-goal `married(X, wilma)` with the given `married(fred, wilma)`, resulting in the binding `X = fred`

4. Verify the consistency of the bindings `X = fred`, `X = fred`, terminating with a return value of `yes`, and the binding `X = fred`.

As shown in Figure 7.4, the IRAAM version of sub-goal conjunction is intersection of the relevant spatial regions. Not surprisingly, disjunction would be implemented by union of the regions.

A distinctive feature of the IRAAM language-of-thought model is its a ability to answer queries involving embedded structure. Consider for example the query "Of whom is Fred



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

Figure 7.4: Computing the answer to the query `husband(Who, wilma)`. Intersections (c) of inverses (b) of `married,wilma` and `gender,male` (a) are put through second forward transform; these forward-transformed regions intersect in the region for `fred`, answering the query (d).

jealous?" Handling this query involves the relation between an element in a tree and an element in a tree nested inside that tree: (`resents` ***barney*** (`married` ***fred*** `wilma`)). In the IRAAM model, this query corresponds to the operations

$$T2(T3(T_1^{-1}(\mathcal{G}(resents)) \wedge T_2^{-1}(\mathcal{G}(barney))) \wedge T_1^{-1}(\mathcal{G}(married)))$$

That is, we take the set of things that Barney resents, then intersect this set with the set of propositions about who is married, and finally find the items in the second slot of the `married` predicate to obtain the set of things of which Barney is jealous – in this case, Fred. These operations are somewhat cumbersome to show as figures, but are nevertheless straightforward to implement in the model.

A few remaining points deserve comment. First, we observe that the failure to find a unification set corresponds to a negative answer to a query: as in Prolog, we assume a "closed world" in which anything not provably true is considered false. Such a failure is illustrated in Figure 7.5, for the query `married(barney, Who)`, in which the lack of an assertion about Barney's being married corresponds to a failure of `married` to unify with `barney` [5]. Second, unlike Prolog, our IRAAM language-of-thought model can just as easily

[5]Apologies to Flintstones fans who know that Barney is married to Betty.

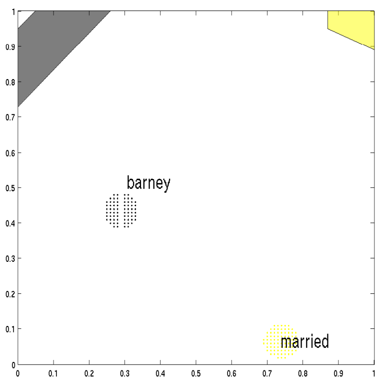


Figure 7.5: Query failure by absence of intersection.

perform queries on functors as on their arguments, allowing us to ask, for example, "What is true of Fred?"[6] Third, it should be clear that we are not interested in modeling the full (Turing equivalent) recursive functionality of Prolog. Our model is designed to capture the part of the language's functionality relevant to the concerns of Fodor and others, not to provide a general programming mechanism.

7.3.2 Discussion

In calling our IRAAM language-of-thought application "neurally plausible," we do not purport to offer a concrete model of the way that real brains or nervous systems actually perform reasoning and deduction. We aim, rather, to suggest a way in which real brains and nervous systems *might* do these sorts of things, using the sorts of representations and operations that we have presented in detail.

The boldest implication of our model – that mental representation is fractal – remains to our knowledge untested. Nevertheless, the essential features of IRAAM – recurrent connections, dynamical computation, and spatial clustering of related representations (predicates in one part of the space, arguments in the other) – have received considerable support from

[6] A recent implementation of Prolog [89] comes with a "Hilog" feature, supporting these sorts of second-order operations.

experimental neuroscience [7]

Independent of these issues, however, is the extent to which our model has dealt with the problems pointed out by Fodor. Summing up, we agree with Fodor [34] that (despite some connectionists' assertions to the contrary), "there still must be a language of thought." Like Fodor, we believe that this LOT consists of structured representations and operations sensitive to them. We disagree, however, that the LOT cannot be connectionist. We hope to have convinced the reader that it can, not just implementationally, but fundamentally.

[7]though the controversy associated with each of those three properties increases in the order that we have listed them. Churchland and Sejnowski [17] state with confidence that "Our brains are dynamical, not incidentally or in passing, but essentially, inevitably, and to their very core." But claims about localization of brain function and representations, ostensibly supported by studies using electrical potentials [78] and PET and functional MRI imaging techniques [75], have come under attack recently [110].

Chapter 8

Conclusions

I wonder: if enough of this sort of data were crunched through a computer, would we begin to be able to model language in terms of complex dynamical systems? Grammars then would not be "innate," but would emerge from chaos as spontaneously evolving "higher orders," in Prigogine's sense of "creative evolution." Grammars could be thought of as "Strange Attractors," ... patterns which are "real" but have "existence" only in terms of the sub-patterns they manifest. If meaning is elusive, perhaps it is because consciousness itself, and therefore language, is fractal.... I find this theory more satisfyingly anarchistic than either anti-linguistics or Chomskyanism. It suggests that language can overcome representation and mediation, not because it is innate, but because it is chaos.

– Hakim Bey, "Chaos Linguistics" [7]

8.1 Summary

We began this thesis by enumerating the necessary conditions of any system claiming to be a model of cognitive processes: the system must be able to (1) compose structured representations over a set of atomic primitives, (2) relate these representations in a systematic way, and (3) be capable (in principle at least) of infinite generalization over the representations. We presented Pollack's RAAM, an encoder/decoder model of hierarchical structure, as a candidate solution to (1) and (2), and described its shortcomings with respect to (3). The main contribution of this thesis was showing how a dynamical-systems understanding of the RAAM decoder – notably, the use of its attractor as the set of atomic primitives – provides a principled solution to (3). We also showed how this interpretation of RAAM, dubbed Infinite RAAM, allows us to reason geometrically about grammatical processes like unification, which have heretofore been considered as mainly or exclusively the domain of recursive symbol systems.

A second contribution of this work was to offer a principled alternative to Chomsky's competence/performance perspective on the knowledge of language [14]. Under that view, a given language community shares the same competence grammar; the empirical phenomena of actual language use (individual differences, slips of the tongue, misunderstandings) are attributed to a distinct performance component, which can vary from person to person. The alternative offered in this thesis is to view *both* competence *and* performance as manifestations of a single underlying (dynamical) system. Observed with enough of a "fuzz factor" (finite precision) this system behaves *as if* it were following a small set of discrete symbolic rules [94], but we cannot consider the system to be literally following these rules in the Chomskyan sense [1]. Further, just as there were several 12-weight solutions to the $a^n b^n$ problem, it is not only the performance, but also the competence, that is (potentially) represented in a different way for each language user. The correspondence we found in those results between simple, general grammar rules and linear dynamical behavior, suggests that the exceptionless, rule-based grammars proposed by Chomsky and others may correspond to an extreme or pathological dynamical regime. In other words, *real language may be fundamentally chaotic.*

[1]In this respect our work instantiates what Horgan and Tienson have called the "Representations without Rules" approach [46]

We do not offer this work, in its present form, as any kind of useful artifact for solving real problems in artificial intelligence. Our examples have been intentionally simple, using small two-dimensional networks for reasons both practical and pedagogical. Computing the attractor of these networks is relatively straightforward with existing software for two-dimensional sparse matrix computations, and the attractor and tree equivalence classes are easily visualized in two dimensions. Instead, we hope to have illustrated a principle by which a neural network may satisfy the traditional cognitive science objections to connectionism. Unfortunately, many earlier attempts at addressing this issue have suffered from overly simplistic assumptions that enabled them to be defeated as straw-man arguments [36][35]. Though we cannot claim to have resolved this fundamental debate, we hope to have nudged it in what we perceive as the right direction.

8.2 Future work

We can envision at least two directions for Infinite RAAM.

8.2.1 Higher-dimensional attractor networks

The error reduction gained from moving terminals onto the attractor, the apparently limited capacity of two-dimensional networks to learn non-trivial training sets using back-prop, and the inability of the pseudo-attractor method to generate attractor points beyond those in the terminal set, all point out the desirability of computing the full attractor in higher dimensions. As we noted in Chapter 8, the exponential complexity of computing a high-dimensional attractor forces us to abandon any method which samples the entire unit hypercube as an initial condition for the iterative attractor computation.

Note however that by definition the attractor is the “fixed set” of the iterated function system (IFS) represented by the RAAM decoder. This means that once we have determined that a point is “on” the attractor (at some fixed resolution), we are guaranteed that any further applications of the IFS transforms to that point will result in points that are also on the attractor. Applying the IFS transforms to these points will likewise yield (other) points on the attractor, until no new points are generated. Furthermore, we know that successive applications of the IFS transforms will always "take" us to the attractor after some finite (and typically small) number of random applications, no matter where in the hypercube we start. Therefore, we can approximate the attractor by starting at a random point, iterating a random sequence of IFS transforms some fixed number of times to "land" on the attractor, and then generating the transitive closure (orbits) of this point. The following algorithm accomplishes this, using 25 random iterations to land on the attractor, and computing the transitive closure via breadth-first search:

```
FAST-ATTRACTOR($T, S, K, D$)

1  $\mathbf{x} \leftarrow$ RANDOM - POINT($D$)
2  for $i \leftarrow 1$ to 25
3      do $k \leftarrow$ RANDOM-INDEX($1, K$)
4          $\mathbf{x} \leftarrow T_k(\mathbf{x})$
5  $A_{new} \leftarrow \mathbf{x}$
6  $A_{old} \leftarrow \emptyset$
7  while $A_{old} \neq A_{new}$
8      do $A_{old} \leftarrow A_{new}$
9          for $k \leftarrow 1$ to $K$
10             do $A_{new} \leftarrow A_{new} \cup T_k(A_{new})$
11 return $A_{new}$
```

Lines 1-4 of this algorithm find an arbitrary point on the attractor. Lines 7-10 use breadth-first search to take the transitive closure of this point until no new attractor points are generated. Though the algorithm can in principle "miss" some attractor points by landing on a small sub-orbit of the attractor, such degenerate cases can be detected, making FAST-ATTRACTOR as good an approximation to the attractor as the original ATTRACTOR algorithm of Chapter 3.

Unlike ATTRACTOR, however, FAST-ATTRACTOR does not compute the IFS transform of every pixel. This means that our UNIFY algorithm will not be able to exploit the one-to-many inverse mapping between a given point and the set of points that go to that point on a particular transform. Therefore, UNIFY must compute the one-to-one inverse of each point directly, using the formula

$$T_k^{-1}(x) = W^{-1}[f^{-1}(T_k(x)) - b]$$

where f is the logistic-sigmoid squashing function, whose inverse is

$$f^{-1}(z) = -\ln(1/z - 1)$$

The loss of the one-to-many mapping can be compensated for by a number of different strategies, including the use of epsilon balls around the attractor points, and replacing point-intersection with a minimal-distance metric.

Even with an efficient algorithm for computing high-dimensional attractors, we are still left with the problem of how to compute intersections of inverses for the UNIFY algorithm of Chapter 7. For two-dimensional networks, the intersection of two sets of points $\mathcal{U}$ and $\mathcal{V}$ can be computed simply as the set of pixels that $\mathcal{U}$ and $\mathcal{V}$ have in common. As the dimensionality of the network increases, however, it becomes easier for this method to "miss" points that are close enough in hyperspace to be considered identical at a given resolution.

We foresee two ways out of this problem. The first is a geometric approach, using existing software for computing intersections of high-dimensional convex polytopes [112]. The

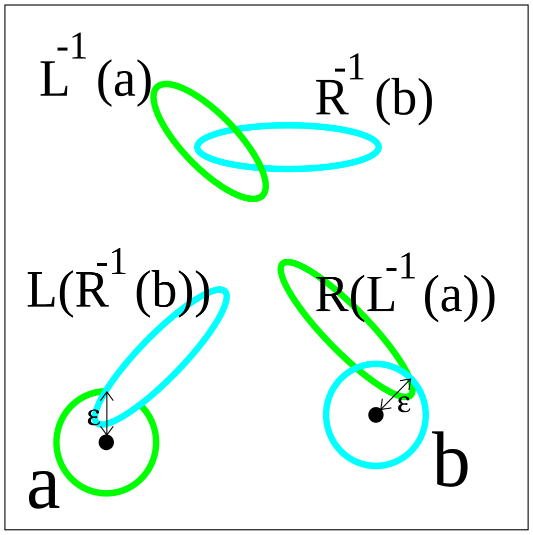


Figure 8.1: Hypothetical hyper-dimensional unification on the tree `(a b)` using a stack of operation history. Example is two-dimensional for ease of visualization; method is designed to work in any number of dimensions.

second is a symbolic approach, in which we maintain a stack containing the "history" of the operations (inverse transform, intersection, forward transform) performed on the terminal points, with the latter represented as epsilon balls or bounding hyperspheres. Figure 8.1 portrays this method.[2]

8.2.2 Exploiting vector representations

As Pollack [73] points out, RAAM's use of fixed-width vectors for representing structured information opens up a variety of opportunities for using these representations in conjunction with other vector-based cognitive approaches.

In the Latent Semantic Analysis model of word meaning [26], each word is represented as a vector in high-dimensional space, based on its co-occurrence with other words in context. Recently, Kintsch [55] has used a simple subject-predicate model to explore how LSA repre-

[2]Though such a method might be seen as undermining our geometric notion of unification by re-introducing stacks and symbols, it would, like back-propagation [17], be a way of obtaining a result, rather than an inherent property of the model.

sentations can be combined to form sentential meanings. In particular, the model provides a way of extracting the various senses of a particular predicate in the context of different subjects. We can envision using IRAAM to extend this model beyond simple subject-predicate relations (i.e., semantic trees of depth one), including relations like direct and indirect object and adjunct, all of which can influence the interpretation of a given predicate.

Current work by Farkas and Li also seeks to model word sense relations, but with a much lower-dimensional vector space. In their Growing Lexical Model, or GLM [31], these researchers develop a recurrent neural network architecture that, unlike LSA and related models, can acquire a lexicon that develops incrementally over time. Using co-occurrence relations from a large corpus, the model inserts a new network node for each word learned, laying out the nodes in a 2D topology. As shown in Figure 8.2, words with a similar sense or syntactic category tend to cluster close together in the space[3]. This layout is very reminiscent of our 2D IRAAM models, suggesting the possibility of using GLM representations as the atoms of a low-dimensional IRAAM model. One interesting avenue to pursue would be to show how an IRAAM network, having learned a set of trees over some subset of a GLM, could generalize to other well-formed trees, based on the proximity of related lexical items coincident with attractor points.

Finally, we envision IRAAM as a potentially important piece in the puzzle of providing a fully connectionist account of natural language parsing and understanding. A good deal of the controversy surrounding such endeavors has come from the interpretation of Simple Recurrent Networks (SRN's) as parsing models. Elman [30] showed that an SRN can be trained to predict the next word in a sequence, which (as illustrated by our logical query answering network in Chapter 9) is one feature of structured representations. Tabor and Tanenhaus [108] employ a similar architecture, and use principle components analysis of hidden unit vectors as a way of attributing grammatical categories to the states of the

[3]I thank Igor Farkas and Ping Li for providing this figure.

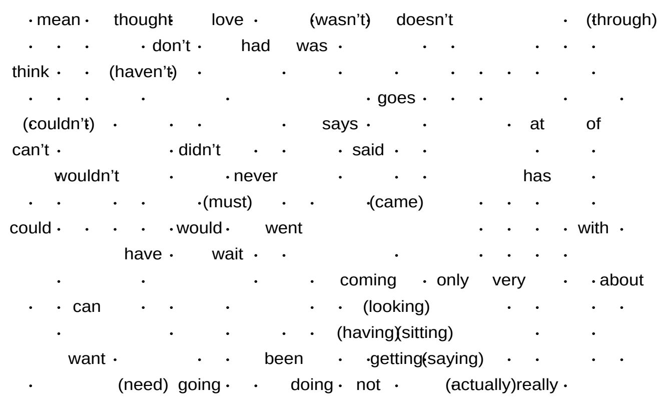


Figure 8.2: Snapshot of a Growing Lexical Model. Labels and dots denote existing nodes. Reproduced, with permission, from [31]

model.

In assessing this work, Steedman [103] argues for an interpretation of the SRN model as a part-of-speech (POS) tagger rather than a parser. Like a standard N-gram POS tagger [10], an SRN uses the context of preceding words to predict the category of the current word, without assigning the words to locations in a parse tree. For this reason, suggests Steedman, some other connectionist principle is needed to account for the structural semantic relations that are only weakly implicit in the states of the SRN models. He cites RAAM (and the Holographic Reduced Representations discussed in our Introduction) as a potential candidate, adding that

> The interesting point of a such a representation is that we might assume that during training conceptual structures are available prelinguistically, and result relatively directly from the structure of connections to the sensorium, short-term memory, and the like.

We agree strongly with Steedman's characterization of this issue, and see the integration of the SRN and IRAAM models as a potentially fruitful direction for future work [4].

[4]Mayberry [61] presents a variant of an SRN that uses a traditional RAAM as a stack for parsing nested relative clauses, but runs up against the problems associated with the traditional model. Berg [6] has trained

Specifically, we can envision a model in which intonational contours [5] and other physically characterizable aspects of linguistic input can be used to guide a SRN through different trajectories in a multi-dimensional IRAAM vector space. An integrated model of this sort could be a significant first step toward a "grand unification" of the symbolic and connectionist approaches.

a variant of RAAM called the XERIC network to parse sentences from large text corpora, and describes the problems associated with decoding words using bit-string representations

[5] as in Steedman's *ANNA married MANNY* example [104], where a "marked" intonation signals type raising

Appendix A

Matlab classes

This appendix contains object-oriented Matlab code to support the experiments described in the thesis. For each Matlab class, the user should create a directory named @*class*, and copy-and-paste the code for the class into appropriately named files. Private methods for a class should go into a directory named `private` under that class's @*class* directory.
For example:

```
% mkdir @raam
% mkdir @raam/private
```

The classes are:

@raam Original RAAM from Pollack 1990

@ifs Two-dimensional IFS RAAM decoder

@tree K-ary tree

@dict Dictionary to map from terminal strings to floating-point vectors

A.1 RAAM class

File @raam/raam.m

```
function r = raam(k, d, seed)
% RAAM Class constructor for RAAM network.
%
%  RAAM(K, D) creates and returns a K-transform, D-dimensional RAAM
%  object with random encoder and decoder weights.
%
%  RAAM(K, D, SEED) allows you to specifiy a seed for the initial
%  random weights, for reproducibility.

% empty constructor
if nargin == 0
  r.encw = [];
  r.decw = [];
  r.d = 0;
  r.k = 0;
  r = class(r, 'raam');

% copy constructor
elseif isa(k, 'raam')
  r = k;

% default constructor
else

  % total number of units in input (output) layer
  nunit = k * d;

  % create initial random weights with biases
  if nargin > 2, randn('seed', seed), end
  r.encw = randn(nunit+1, d);
  r.decw = randn(d+1, nunit);

  r.d = d;
  r.k = k;

  r = class(r, 'raam');

end
```

File `@raam/learn.m`

```
function r = learn(r, targs, dict, eta, mu, niter)
% RAAM/LEARN Learn trees using RAAM backprop algorithm.
%
%    R = LEARN(R, TREES, DICT, ETA, MU, NITER) uses RAAM backprop with
%    fixed dictionary to compute encoder end decoder weights from RAAM
%    object R.  TREES should be a cell array of tree structures.  DICT
%    is a hash table dictionary of learned terminal symbols and their
%    values.

% report error at after this many iterations
REPORT = 100;

% reset last-iteration weight-changes for momentum
dwenc1 = zeros(size(r.encw));
dwdec1 = zeros(size(r.decw));

% get all subtrees
subs = {};
for j = 1:size(targs, 2)
  nonterms = subtrees(targs{j});
  for k=1:size(nonterms,2)
    subs{end+1} = nonterms{k};
  end
end

% partition into terminal and non-terminal subtrees
nonterms = {}; terms = {};
for j = 1:size(subs, 2)
  t = subs{j};
  if is_term(t)
    terms{end+1} = show(t);
  else
    nonterms{end+1} = t;
  end
end

% bozo filter for wrong arity
for j = 1:length(nonterms)
  nonterm = nonterms{j};
  k = arity(nonterm);
  if k ~= r.k
```

```
    error(['Arity mismatch between tree (' num2str(k) ') and network (' ...
            num2str(r.k) ')'])
  end
end

% these are the patterns to be learned
npat = size(nonterms, 2);

% convert patterns to strings
for j = 1:size(nonterms, 2)
  nonterms{j} = show(nonterms{j});
end

% uniquify strings
nonterms = unique(nonterms);
terms = unique(terms);

% put indices into trees, for optimizing tree-vector compuation
for j = 1:size(targs, 2)
  targs{j} = index(targs{j}, nonterms);
end

% train for specified number of iterations
for i = 1:niter

  % reset weight changes, RMS error, tree codes table
  dwenc = zeros(size(r.encw));
  dwdec = zeros(size(r.decw));
  rmserr = zeros(1, r.k*r.d);
  maxerr = 0;

  % compute codes for all trees
  codes = zeros(npat, r.k*r.d);
  for j = 1:size(targs, 2)
    codes = encode_branches(r, targs{j}, dict, codes);
  end

  % set up null errors array for memoizing
  errors = cell(1, npat);

  % loop over trees, accumulating weight changes
  for j = 1:size(targs, 2)
```

```
    [dwenc, dwdec, rmserr, errors, maxerr] = ...
         learn_tree(r, targs{j}, dwenc, dwdec, rmserr, codes, errors, maxerr);
  end

  % report error first time and at fixed intervals
  if (i==1) | ~mod(i, REPORT)

    % report RMS error, max error
    rmserr = sqrt(sum(rmserr)/(length(rmserr)*npat));
    fprintf('%5d: %f %f\n', i, rmserr, maxerr)

  end

  % average error derivative over patterns
  dwenc = dwenc / npat;
  dwdec = dwdec / npat;

  % update weights
  [r.encw,dwenc1] = update_weights(r.encw, eta, mu, dwenc, dwenc1);
  [r.decw,dwdec1] = update_weights(r.decw, eta, mu, dwdec, dwdec1);

end
```

File @raam/encode.m

```
function e = encode(r, t, dict)
% RAAM/ENCODE retrieve hidden-layer encoding for tree using RAAM encoder.
%
%    E = ENCODE(R, T, DICT) returns hidden-layer encoding E of tree T
%    using RAAM R and dictionary DICT.

% get input pattern vector for tree branches
[codes,pat] = encode_branches(r, t, dict, []);

% forward-pass to hidden layer
e = forward(pat, r.encw);
```

File `@raam/decode.m`

```
function t = decode(r, code, dict, tol, ttest)
% RAAM/DECODE decode tree using RAAM decoder
%
%    T = DECODE(R, CODE, DICT, TOL, TTEST) returns tree T decoded by
%    RAAM R using dictionary DICT with tolerance TOL and terminal-test
%    function named TTEST.

% terminal test returns terminal on success, null on failure
s = feval(ttest, r, code, tol, dict);

% terminal: stop
if ~isempty(s)
  t = tree(char(s));

% non-terminal: recur
else
  actout = forward(code, r.decw);

  for k = 1:r.k
    trans = actout(r.d*(k-1)+1:r.d*k);
    branches{k} = decode(r, trans, dict, tol, ttest);
  end
  t = tree(branches);
end
```

File `@raam/private/update_weights.m`

```
function [w,dw1] = update_weights(w, eta, mu, dw, dw1)
% generic weight update with learning rate ETA and momentum MU

w = w + eta*dw + mu*dw1;
dw1 = dw;
```

File `@raam/private/learn_tree.m`

```
function [dwenc, dwdec, rmserr, errors, maxerr] = ...
    learn_tree(r, t, dwenc, dwdec, rmserr, codes, errors, maxerr)
% recursively learn encoder, decoder weight changes based on tree, memoized

% only learn from non-terminal
if ~is_term(t)

  % recur on children
  for k = 1:r.k
    [dwenc, dwdec, rmserr, errors, maxerr] = ...
      learn_tree(r, branch(t,k), dwenc, dwdec, rmserr, codes, errors, maxerr);
  end

  % use memoized errors if available
  index = get_value(t);
  s = errors{index};
  if ~isempty(s)
    dwd = s.dwdec;
    dwe = s.dwenc;
    errout = s.errout;

  % otherwise, compute errors
  else

    % get input pattern vector for tree
    index = get_value(t);
    input = codes(index, :);

    % this is what it means to be an autoassociator
    target = input;

    % forward-pass to hidden layer
    acthid = forward(target, r.encw);

    % forward-pass to output layer
    actout = forward(acthid, r.decw);

    % error on output layer
    errout = target - actout;

    % delta on output layer, using Alan Blair's formula
    delout = (1 - target.*actout) .* (target - actout);

    % error on hidden layer
    errhid = delout * r.decw(1:end-1,:)';

    % delta on hidden layer via standard delta rule
    delhid = errhid .* acthid .* (1 - acthid);

    % compute error derivative on hidden->output weights
    dwd = weight_change(acthid, delout);

    % compute error derivative on input->hidden weights
    dwe = weight_change(input, delhid);

    % memoize errors
    errors{index} = struct('dwdec',dwd, 'dwenc',dwe, 'errout',errout);

  end

  % accumulate weight change on hidden->output weights
  dwdec = dwdec + dwd;

  % accumulate weight change on input->hidden weights
  dwenc = dwenc + dwe;

  % accumulate RMS error, max error
  rmserr = rmserr + errout.*errout;
  if max(abs(errout)) > maxerr
    maxerr = max(abs(errout));
  end

end
```

File `@raam/private/forward.m`

```
function y = forward(x, w)
% forward propagation: vector X with bias time weights W followed by squashing

y = [x 1] * w;
y = 1 ./ (1 + exp(-y));
```

File `@raam/private/encode_branches.m`

```
function [codes,pat] = encode_branches(r, t, dict, codes)
% get uncompressed vector code for tree branches

% if already memoized, just return code
if ~isempty(codes)
  index = get_value(t);
  pat = codes(index, :);
  if nnz(pat), return, end
end

% initialize
pat = [];

% recur on branches
for k = 1:r.k
  b = branch(t, k);

  % terminal gets pattern directly from dictionary
  if is_term(b)
    subpat = lookup(dict, show(b));

  % non-terminal gets pattern by recurring on branch
  else
    [codes,input] = encode_branches(r, b, dict, codes);
    subpat = forward(input, r.encw);
  end

  % combine branch patterns into single pattern
  pat(end+1:end+r.d) = subpat;

end

% memoize
if ~isempty(codes), codes(index,:) = pat; end
```

File `@raam/private/weight_change.m`

```
function dw = weight_change(a, d)
% weight change equals activation vector with bias, times delta vector

dw = [a 1]' * d;
```

A.2 IFS class

File `@ifs/ifs.m`

```
function r = ifs(w, s, f)
%IFS Class constructor for IFS RAAM decoder network.
%
%  IFS(W, S) creates and returns a IFS object from the
%  matrix W of IFS transform weights, using pixel resolution S.
%  Matrix W should have D+1 rows and K*D columns, where K is the
%  number of transforms and D is the number of dimensions (hidden
%  units) of the RAAM network
%
%  IFS(W, S, F) allows you to specify an output function F to
%  apply to the transforms.  The default output function is the
%  logistic sigmoid function f(x) = 1 / (1+exp(-x)).

% empty constructor
if nargin == 0
  r.w = [];
  r.k = 0;
  r.s = 0;
  r.cache = [];
  r.f = [];
  r = class(r, 'ifs');

% copy constructor
elseif isa(w, 'ifs')
  r = w;

% default constructor
else

  % default to logistic sigmoid output function
  if nargin > 3
    r.f  = f;
  else
    r.f  = 'sigmoid';
  end

  % extract transforms, their count
  r.w = w;
  r.k = size(w, 2) / 2;
  % store resolution
  r.s = s;

  % set initial condition, using square matrix for 2D
  a = sparse(ones(s));

  % normalize to (0,1)^2
  ind = find(a);
  [i j] = ind2sub([r.s r.s], ind);
  xy = [i-1 j-1] / r.s;

  % run forward transform
  xy = [xy ones(size(xy, 1), 1)]; % augment with bias "unit"
  txy = xy * r.w;   % multiply by weights
  txy = feval(r.f, txy);          % apply activation function

  % cache transforms of each point
  for k = 1:r.k

    % discretize transforms
    tij = fix(1 + r.s*txy(:, (k-1)*2+1:k*2));

    % cache discrete transform
    r.cache(k,:) = sub2ind([r.s r.s], tij(:,1), tij(:,2));

  end

  r = class(r, 'ifs');

end
```

File `@ifs/attractor.m`

```
function a = attractor(r, n)
%IFS/ATTRACTOR IFS attractor of RAAM network decoder.
%
%  ATTRACTOR(R) returns the IFS "infinite iterations" attractor of the
%  2D decoder of RAAM network R.  The attractor is an SxS sparse
%  matrix that can be displayed using SPY or PCOLOR
%
%  ATTRACTOR(R, N) restricts the number of iterations to N.
% set initial condition, using square matrix
a = sparse(ones(r.s));

% pump-primer for loop
nz = 0;

% use fixed iterations, or "infinity" if unspecified
if nargin < 2
  n = realmax;
end

% loop until fixed size or specified iterations
for i = 1:n
  if nnz(a) == nz, break, end
  nz = nnz(a);
  a = condense(r, a, 1, r.k);
end
```

File `@ifs/address.m`

```
function b = address(r, a, h)
% IFS/ADDRESS address IFS attractor of RAAM decoder.
%
%  B = ADDRESS(R, A, H) uses H digits of "history" to label attractor
%  A of decoder of IFS network R using Barnsley's IFS addressing scheme.
%  Returns labeled attractor B.

% reset original, using square matrix for 2D
b = sparse(zeros(r.s));

% build K^H copies of attractor
for i = 1 : power(r.k, h)

  % follow one path along the derivation tree to make an address
  j = i - 1;
  for k = 1 : h
    path(k) = mod(j, r.k) + 1;
    j = fix(j / r.k);
  end

  % condense the original attractor using the path
  c = a;
  for j = 1:length(path)
```

```
    c = condense(r, c, path(j), path(j));
  end

  % address the condensed copy (starts out as 1, so multiplication
  % creates address)
  c = c * power(r.k, i-1);

  % add the addressed copy into the original: overlapped points get
  % sum of each part
  b = b + c;

end

% set all attractor points to 1
a = spones(a);
```

File `@ifs/trees.m`

```
function [trees, a] = trees(r, a, n)
% IFS/TREES get trees from IFS attractor of RAAM network decoder.
%
%  T = TREES(R, A, N) returns an MxK matrix of codes representing the
%  M unique K-ary trees of depth N decoded by Ifs network R using
%  attractor A.  Tree for a particular code can be viewed using
%  SHOWTREE.
%
%  If N is unspecified, all decoded trees are returned.
%
%  [T,B] = TREES(R, A) also returns matrix B containing indices into
%  T, which for 2D decoders can be viewed with PCOLOR.
%

% get maximum value in attractor
nl = max(unique(full(a)));

% initialize trees array with zeros up through this value
trees = zeros(nl, r.k);

% default to all trees via "infinite" iterations
if nargin < 3
  n = Inf;
end
```

```
% loop to specified number of iterations
for i = 1:n

  % find empty points in space
  ind = find(a == 0);

  % if none, we're done
  if isempty(ind), break, end

  % get number of points this iteration
  [i, j] = ind2sub([r.s r.s], ind);
  m = length(i);

  % preinitialize array
  indices = zeros(r.k, m);

  % build indices into tree array by transforming current empty points
  for k = 1:r.k
    indices(k,:) = a(r.cache(k,ind));
  end

  % append new trees to current
  found = all(indices);
  newtrees = indices(:,found);
  newtrees1 = newtrees';
  newtrees2 = unique(newtrees1, 'rows');
  index = size(trees,1) + getpos(newtrees2, newtrees1);
  trees = [trees;newtrees2];

  % update map using new tree indices
  b = sparse(i(found), j(found), index, r.s, r.s);
  a = a + b;

end
```

File `@ifs/unify.m`

```
function b = unify(r, a, t)
% IFS/UNIFY get points for tree using inverse IFS RAAM decoder.
%
%  B = UNIFY(R, A, T) uses the inverse of IFS RAAM decoder network R
%  to generate (possibly empty) sparse matrix B of points
%  corresponding to tree object T.  A is R's attractor. The encoder
%  generates tree points via intersection of inverses.  A should
%  contain values corresponding to terminals in T.

% terminal is encoded by attractor
if is_term(t)

  % convert tree string to number
  addr = str2num(char(show(t)));

  % return points at that address; special handling for 2D
  [i,j] = find(a == addr);
  b = sparse(i, j, 1, r.s, r.s);

% non-terminal is encoded by intersection of inverses
else

  % bozo filter for arity mismatch
  if arity(t) ~= r.k
    error(['arity mismatch between tree ' num2str(arity(t)) ' and' ...
    ' RAAM ' num2str(r.k)])
  end

  % start with full space
  b = sparse(ones(r.s, r.s));

  % unify over successive inverses
  for k = 1:r.k
    c = unify(r, a, branch(t, k));
    c = invert(r, c, k);
    b = b & c;
  end

  % remove attractor
  b = b & ~a;
end
```

File `@ifs/private/getpos.m`

```
function pos = getpos(a, b)
% compute positions of rows of matrix A within matrix B.

for i=1:size(b,1)
  pos(i) = find(all((a ==(ones(size(a,1),1)*b(i,:)))'));
end
```

File `@ifs/private/condense.m`

```
function a = condense(r, a, kbeg, kend)
% run one iteration of the IFS roder "copy" machine on image A

ind = find(a);
a = sparse(zeros(r.s));
for k = kbeg:kend
  [i, j] = ind2sub([r.s r.s], r.cache(k, ind)');
  a = a | sparse(i, j, 1, r.s, r.s);
end
```

File `@ifs/private/invert.m`

```
function b = invert(r, a, k)
% compute Kth inverse of image A under IFS R

% start with all zeros
b = sparse(zeros(r.s, r.s));

% find where cached transform points equal points in A
map = find(ismember(r.cache(k,:), find(a)));

% set inverse matrix at those points to 1
b(map) = 1;
```

File `@ifs/private/sigmoid.m`

```
function fx = sigmoid(x)
% logistic sigmoid squashing function

% avoid overflow
EPSILON = 1e-6;

fx = min(1 ./ (1+exp(-x)), 1-EPSILON);
```

A.3 Tree class

File @tree/tree.m

```
function t = tree(s)
% TREE Tree class constructor.
%
%   TREE(s) creates a tree object from the string representation S. For example
%
%     >> t = tree('(a (b c d))')
%
%   creates a tree whose left branch is the terminal 'a', and whose
%   right branch is a tree whose terminals are 'a', 'b', and 'c'.
%
%   TREE({B1,B2,...,Bn}) creates an N-ary tree from branches B1,...,Bn:
%
%     >> t = tree({tree('a'), tree('(b c d)')})

% empty constructor
if nargin == 0
  t.value = [];
  t.branches = [];
  t = class(t, 'tree');

% copy constructor
elseif isa(s, 'tree')
  t = s;

% construct from subtrees
elseif iscell(s)
  t.value =[];
  for k = 1:length(s)
    t.branches{k} = struct(s{k});
  end
  t = class(t, 'tree');

% default string constructor
else
  t = tree_from_string(stripws(s));
  t = class(t, 'tree');
end
```

File `@tree/arity.m`

```
function r = arity(t)
% TREE/ARITY    ARITY(T) returns the arity (number of branches) of tree T.

if is_term(t)
  r = 0;
else
  r = size(t.branches, 2);
end
```

File `@tree/branch.m`

```
function b = branch(t, n)
% TREE/BRANCH    BRANCH(T, N) returns Nth branch of tree T.

% bozo filters
if is_term(t)
  error 'tree is terminal';
elseif n > arity(t)
  error 'branch number exceeds arity';

else
  b = t.branches{n};
  b = class(b, 'tree');
end
```

File `@tree/front.m`

```
function s = front(t)
% TREE/FRONT    FRONT(T) returns the string at the frontier of tree T.

% terminal gets converted to string (may be a number)
if is_term(t)
  s = num2str(t.value);

% recur on non-terminal
else
  s = '';
  for i = 1:arity(t)
    s = [s, front(branch(t, i))];
  end
end
```

File `@tree/index.m`

```
% TREE/INDEX   index a tree using a sorted cell array of subtree strings
function t = index(t, s)

if ~is_term(t)
  t.value = strmatch(show(t), s, 'exact');
  for k = 1:arity(t)
    t.branches{k} = struct(index(branch(t,k), s));
  end
end
```

File `@tree/is_term.m`

```
function yn = is_term(t)
% TREE/IS_TERM    IS_TERM(T) returns 0 if tree T branches, 1 otherwise.

  yn = isempty(t.branches);
```

File `@tree/show.m`

```
function s = show(t)
% TREE/SHOW    SHOW(T) returns the string representation tree T.

% terminal gets converted to string (may be a number)
if is_term(t)
  s = num2str(t.value);

% recur on non-terminal
else
  s = '(';
  for i = 1:arity(t)
    s = [s, show(branch(t, i))];
    if i < arity(t)
      s = [s, ' '];
    end
  end
  s = [s, ')'];
end
```

File `@tree/get_value.m`

```
function  v = get_value(t)
% TREE/GET_VALUE    GET_VALUE(T) returns object (string, vector) from tree.

v = t.value;
```

File `@tree/subtrees.m`

```
function subs = subtrees(t)
% TREE/SUBTREES Returns cell array of all unique subtrees of tree
% (including tree itself)

subs = {};
subs = subtrees_r(t, subs);
```

File `@tree/private/subtrees_r.m`

```
function subs = subtrees_r(t, subs)
% tail-recursive subtree computation

subs{end+1} = t;

% only care about non-terminals
if ~is_term(t)

  for k = 1:arity(t)
    subs = subtrees_r(branch(t, k), subs);
  end

end
```

File `@tree/private/tree_from_string.m`

```
function t = tree_from_string(s)
% recursively build tree structure from string representing tree

% terminal
if ~ismember('(', s)
  t.value = s;
  t.branches = {};

% non-terminal
else

  % initial conditions: empty stack, start after first paren, no subtrees
  t.value = [];
  stack = 0;
  last = 2;
  r = 1;
  i = 2;

  % loop over characters between parens
  while i < length(s)

    % current character
    c = s(i);

    % use stack counter to track sub-trees
    if c == '('
      stack = stack + 1;
    elseif c == ')'
      stack = stack - 1;
    end

    % on empty stack, build a branch by recurring on subtree
    if stack == 0

      while s(i) ~= '.' & s(i) ~= '(' & s(i) ~= ')', i = i + 1; end

      if s(last) == '('
        curr = i;
      else
        curr = i - 1;
      end
      ssub = s(last:curr);
      t.branches{r} = tree_from_string(ssub);
      r = r + 1;
      last = i + 1;

    end
    i = i + 1;
  end
end
```

File `@tree/private/stripws.m`

```
function t = stripws(s);
% strip whitespace (blanks and tabs) from string, replacing between-
% terminal whitespace with period: '(a (b c))' ==> '(a(b.c))'

interm = 0;
j = 1;

for i = 1:length(s)

  c = s(i);
  switch c

   case {'(', ')'}
    t(j) = c;
    j = j + 1;
    interm = 0;

   case {' ', '\t'}
    if interm
      t(j) = '.';
      j = j + 1;
    end
    interm = 0;

   otherwise
    t(j) = c;
    j = j + 1;
    interm = 1;
  end
end
```

A.4 Dictionary class

File `@dict/dict.m`

```
function d = dict
% DICT Class constructor for dictionary object

  d.table = struct('dummy', 0);
  d = class(d, 'dict');
```

File `@dict/enter.m`

```
function d = enter(d, k, v)
% DICT/ENTER   D = ENTER(D, K, V) put V into dictionary D, keyed by K

key = fixkey(char(k));
d.table = setfield(d.table, k, v);
```

File `@dict/lookup.m`

```
function v = lookup(d, k)
% V = LOOKUP(D, K) returns entry keyed by K in dictionary D

k = fixkey(char(k));
v = getfield(d.table, k);
```

File `@dict/show.m`

```
function show(d)
% SHOW(D) displays contents of dictionary D

names = fieldnames(d.table);

if length(names) == 1, 'empty', return, end

table = struct(names{2}, lookup(d, names{2}));
for i = 3:size(names,1)
  table = setfield(table, names{i}, lookup(d, names{i}));
end

table
```

File `@dict/getkeys.m`

```
function keys = getkeys(d)
% KEYS = GETKEYS(D) returns a cell array of keys in dictionary D

keys = fieldnames(d.table);
keys = keys(2:end);
```

File `@dict/private/fixkey.m`

```
function k = fixkey(k)
% support non-alphabetic characters in dictionary keys

for i = 1:length(k)

  c = k(i);

  if c == ' '
    c = '_';
  elseif c == '('
    c = 'o';
  elseif c == ')'
    c = 'c';
  end

  k(i) = c;

end
```

Appendix B

Blind Watchmaker code

You should save these three files, and all subsequent code, in the directory containing the Matlab classes.

File `bw.m`

```
% BW Blind Watchmaker program.
%
%   BW(W, PFUN) provides a Blind Watchmaker interface for evolving a
%   vector W of floating-point values.  PFUN is a string naming a
%   function PFUN(W), a user-defined display function that accepts the
%   floating-point values and creates the appropriate graphics.  The
%   program displays the graphics in a large window at left, and four
%   smaller windows at right representing the results of mutating the
%   vector with normally-distributed random noise, scaled by the
%   standard deviation of the vector.  The magnitude of the noise, as
%   well as the random seed, can be set via interface controls, and
%   there is a button to save the vector when you are done.
%
%   BW(W, PFUN, IFUN) allows you to specify a string IFUN naming an
%   initialization function IFUN(), so you optimize-out any
%   computation that needs to be done only at the beginning of the
%   process.  IFUN should return the data D resulting from this
%   computation, and PFUN should be PFUN(W, D).
%
%   If W is a string instead of a matrix, the program will load the
%   weights from a file named W.
%
%   Examples:
%
```

```
%     >> bw(rand(3,4), 'myplot')
%
%     >> bw(rand(2,5), 'myplot', 'myinit')
%
%     >> bw('myfile.mat', 'myplot')

function bw(w, pfun, ifun)

  % set globals for callbacks
  global MU_SLI SEED_EDT W IFUN PFUN INIT
  PFUN = pfun;

  % load or init global weights
  if (ischar(w)), w = getfield(load(w), 'w'); end
  W = w;

  % initial random seed and mutation rate
  S = 0;
  MU = .5;

  % reset normally-distributed random number generator state to seed value
  randn('state', S);

  % add controls
  MU_SLI = add_slider('Mutation Rate: %2.2f', 0, 1, MU, .05, 100);
  SEED_EDT = add_edit('Seed:', S, 250, 22, 20);
  add_button('Reset', 400);
  add_button('Load', 500);
  add_button('Save', 600);
  add_button('Close', 700);

  % get initialization data for optimization, if specified
  if nargin > 2
    IFUN = ifun;
    INIT = feval(IFUN);
  else
    INIT = [];
    IFUN = '';
  end

  % plot initial weights with variants
  bw_plot(w, MU, PFUN, INIT);
  % set figure name
  set(gcf, 'Name', strcat('Blind Watchmaker:\t', PFUN));

% add an editable text control to the current figure
function txt = add_edit(label, value, xoff, yoff, width)

  % text that labels the editable text
  uicontrol(gcf, 'Style', 'text', ...
    'String', label, ...
    'FontSize', 14, ...
    'Position', [xoff yoff 80 20]);

  txt = uicontrol(gcf, 'Style', 'edit', ...
                  'Position', [xoff+80 yoff width 20], ...
  'String', value, ...
  'FontSize', 14);

% add a slider to the current figure
function sli = add_slider(label, lo, hi, value, step, xoff)

  % text that labels the slider
  sli_lbl = uicontrol(gcf, 'Style', 'text', ...
      'String', sprintf(label, value), ...
      'Position', [xoff 40 130 20]);

  % text that labels the slider range
  range = sprintf('%d . . . . . . . . %d', lo, hi);
  uicontrol(gcf, ...
    'Style', 'text', ...
    'String', range, ...
    'Position', [xoff 25 130 20]);

  % slider that controls the value
  sli = uicontrol(gcf, ...
                  'Style','slider', ...
  'Callback', 'bw_callback(''Slider'')', ...
  'UserData', struct('ctl',sli_lbl, 'fmt',label), ...
          'Min', lo, 'Max', hi, 'Position', [xoff 5 130 25], ...
  'Value', value);
```

```
% add button to current figure
function add_button(label, xoff)

  uicontrol(gcf, ...
    'Style', 'push', ...
    'String', label, ...
    'Position', [xoff 15 80 35], ...
    'Callback', ['bw_callback(''' label ''')'])
```

File bw_callback.m

```
function bw_callback(what)
% callback for activity in Blind Watchmaker GUI

global FILE_EDT MU_SLI SEED_EDT IFUN PFUN N INIT W

switch what

 case 'Reset'
  mu   = get(MU_SLI, 'Value');
  seed = sscanf(get(SEED_EDT, 'String'), '%d');
  if size(INIT, 1) > 0
    bw(W, PFUN, IFUN)
  else
    bw(W, PFUN);
  end

 case 'Load'
  [filename, pathname] = uigetfile('*.mat', 'Load Genotype');
  fname = [pathname filename];
  if fname, load(fname); end
  mu = get(MU_SLI, 'Value');
  bw_plot(w, mu, PFUN, INIT);

 case 'Save'
  w = get(gcf, 'UserData');
  [filename, pathname] = uiputfile('*.mat', 'Save Genotype');
  fname = [pathname filename];
  if fname, save(fname, 'w'); end

 case 'Close'
  close
 case 'Slider'
  label = get(gco, 'UserData');
  label_ctl = getfield(label, 'ctl');
  label_fmt = getfield(label, 'fmt');
  value = get(gco, 'Value');
  if (ismember('%d', label_fmt))
    value = fix(value);
  end
  set(label_ctl, 'String', sprintf(label_fmt, value));

 % click on image
 otherwise
  mu = get(MU_SLI, 'Value');
  d = get(gcbo, 'UserData');
  w = getfield(d, 'w');
  bw_plot(w, mu, PFUN, INIT);

end
```

File bw_plot.m

```
function bw_plot(w, MU, FN, init)
% plot current phenotype, plus four mutations

  % set positions of individual image squares
  main_pos = [.05,.175, .4, .7];
  img1_pos = [.5, .175, .2, .35];
  img2_pos = [.5, .6,   .2, .35];
  img3_pos = [.75,.175,   .2, .35];
  img4_pos = [.75,.6,   .2, .35];

  % rescale figure
  pos = get(gcf, 'Position');
  pos(3) = 800; % width
  pos(4) = 500; % height
  set(gcf, 'Position', pos);

  % plot un-mutated in large square
  plot_system(w, main_pos, FN, init);
  % plot four mutations in little squares
  plot_mutation(w, img1_pos, MU, FN, init);
  plot_mutation(w, img2_pos, MU, FN, init);
  plot_mutation(w, img3_pos, MU, FN, init);
  plot_mutation(w, img4_pos, MU, FN, init);

  % store current weights
  set(gcf, 'UserData', w);

% plot one phenotype using mutation of specified weights
function w = plot_mutation(w, pos, MU, FN, init)

  w = w + MU*randn(size(w))*mean(std(w));
  plot_system(w, pos, FN, init);

% plot one phenotype using specified weights
function plot_system(w, pos, FN, init)

  h = subplot('position', pos);

  if size(init, 1) > 0
    feval(FN, w, init);
  else
    feval(FN, w);
  end

  set(h, 'ButtonDownFcn', 'bw_callback(''Click'')')
  set(h, 'UserData', struct('w',w));
```

Appendix C

Hill-climbing code

File `climb.m`

```
function w = climb(w, mu, maxit, evalfun, seed)
%CLIMB   Generic hill-climbing on real-valued weights
%
%  W = CLIMB(WINIT, MU, MAXIT, EVALFUN, [SEED]) uses evaluation
%  function EVALFUN to hill-climb on initial weights WINIT.  MAXIT
%  specifies the maxinum number of iterations to perform, MU the
%  learning rate.  SEED is an optional seed for the random number
%  generator, to support reproducible results.  EVALFUN should
%  return a number between 0 and 1.
%
%  Example:
%
%    >> w = climb(randn(3,4), .1, 1000, 'myeval');

% initial conditions
wnew = w;
w = wnew;
iter = 0;
best = feval(evalfun, wnew);
fprintf('%d: %f\n', 0, best);

% use random seed if provided
if nargin > 4, randn('seed', seed), end

% hill-climb to maximum specified failure iterations or to success
while iter < maxit
```

```
  % call evaluator function on mutated weights
  success = feval(evalfun, wnew);

  % quit on unidimensional 1
  if (length(success) == 1) & success(1) >=1, break, end

  % update weights on improvement
  if prod(success >= best) & find(success>best)
    best = success;
    fprintf('%iter: %f\n', 0, best);
    w = wnew;
    iter = 0;

  % mutate on failure by an amount proportional to mutation rate
  % and success rate
  else
    wnew = w + (1-success) * mu * randn(size(w));
    iter = iter + 1;
  end

end

if iter < maxit
  fprintf('succeeded\n');
else
  fprintf('failed\n');
end
```

Appendix D

Miscellaneous code for experiments

File `allstrings.m`

```
function strings = allstrings(map)
% ALLSTRINGS(MAP) returns strings made from trees encoded by MAP
% obtained from IFS.

% find row containing first non-terminal tree
where = find(map(:,1));
first = where(1);

% build strings from this row to last
for i = first:length(map)
  t = maketree(map, i);
  strings{i-first+1} = front(t);
end
```

File `ifsplot.m`

```
function ifsplot(w)
% IFSPLOT - plot function for 100x100 IFS RAAM with Blind Watchmaker

i = ifs(w, 100);
a = attractor(i);
spy(a)
```

File evaltre.m

```
function success = evaltre(w)
% evaluation function for hill-climbing to lots of trees

% tweakable params
RES = 64;

i = ifs(w, RES);
a = attractor(i);
a = address(i, a, 1);
t = trees(i, a);

success = size(t,1) / (RES*RES);
```

File evalcfg.m

```
function success = evalcfg(w)
% evaluation function for hill-climbing to a^nb^n

% tweakable params
RES = 64;
TARGS = {'12','112', '1122', '11122', '111222', '1111222', '11112222', ...
 '111112222', '1111122222'};

i = ifs(w, RES);
a = attractor(i);
a = address(i, a, 1);
t = trees(i, a);
s = allstrings(t);

total = 0;
for i = 1:length(TARGS)
  total = total + ismember(TARGS{i}, s);
end

success = total / length(TARGS);
```

File corner_dict.m

```
function d = corner_dict(words)
% CORNER_DICT(WORDS) returns a dictionary mapping from WORDS to
% orthonormal basis) vectors made from one-in-N-codes padded with
% zeros.

d = dict;
n = length(words);
for i = 1:n
  word = words{i};
  value = zeros(1, 2*n);
  value(i) = 1;
  d = enter(d, word, value);
end
```

File load_trees.m

```
function trees = load_trees(fname)
%LOAD_TREES load trees from ASCII file
%
%  LOAD_TREES(FILENAME) loads a set of trees from the ASCII file named
%  FILENAME, returning the trees as a cell array of tree objects.

fid = fopen(fname, 'r');

trees = {};

while 1
  line = fgets(fid);
  if line == -1, break, end
  if ~isempty(line)
    trees{end+1} = tree(line);
  end
end

fclose(fid);
```

File maketree.m

```
function t = maketree(map, i)
% MAKETREE(MAP, I) returns Ith tree encoded by MAP obtained from IFS.

row = map(i,:);

% on attractor; terminate
if ~nnz(row)
  t = tree(num2str(i));

% off attractor; recur
else
  for k = 1:size(map, 2)
    branches{k} = maketree(map, row(k));
  end
  t = tree(branches);
end
```

File ttold.m

```
function s = ttold(r, code, tol, dict)
% thresholding terminal test from (Pollack 1990)

% all above or below threshold
if prod(code < tol | code > (1-tol))

  s = 'X'; % assume bogus
  code(find(code<tol)) = 0;
  code(find(code>(1-tol))) = 1;
  keys = getkeys(dict);
  for i = 1:length(keys)
    key = keys(i);
    val = lookup(dict, key);
    if prod(val == code)
      s = key;
    end
  end
else
  s = '';
end
```

Appendix E

Sample experiments

If you have successfully copied or downloaded the code from the preceding appendices, you should be able to run the following experiments by invoking Matlab and entering the commands provided.

E.1 Blind Watchmaker for IFS images

```
>> bw(randn(3,4), 'ifsplot')
```

E.2 Hillclimbing to $a^n b^n$

```
>> w = climb(4*randn(3,4), 4, 30, 'evalcfg');
>> i = ifs(w, 64);
>> a = attractor(i);
>> a = address(i, a, 1);
>> [t,b] = trees(i, a);
>> pcolor(b)
>> allstrings(t)
```

E.3 Hillclimbing to lots of strings

```
>> w = climb(randn(3,4), 1, 100, 'evaltre');
>> i = ifs(w, 64);
>> a = attractor(i);
>> t = tree( '(1 1)' ); % or whatever tree you like
>> spy(unify(i, a, t))
```

E.4 Learning a set of related sentences in old RAAM

Copy the following lines to a file called npvp.tre:

```
(D (A (A (A N))))
((D N) (P (D N)))
(V (D N))
(P (D (A N)))
((D N) V)
((D N) (V (D (A N))))
((D (A N)) (V (P (D N))))
```

Then do the following in Matlab:

```
>> r = raam(2,10);
>> t = load_trees('npvp.tre');
>> d = corner_dict({'D', 'A', 'N', 'P', 'V'});
>> r = learn(r, t, d, .1, .9, 5000);
>> e1 = encode(r, t{1}, d);
>> t1 = decode(r, e1, d, .2, 'ttold');
>> show(t1)
```

You can experiment with the learning rate, momentum, and number of iterations in the `learn` function, in order to decode various trees from the training set, as well as new trees. Five thousand iterations is probably the fewest required for successful decoding of the trees.

Appendix F

Java IFS class

This Java code implements the FAST-ATTRACTOR algorithm of Chapter 8.

```
import java.util.Vector;
import java.util.Random;
import java.util.Hashtable;

 /**
  * IFS is a class supporting fast multi-dimensional IFS attractor
  * computations using the FAST-ATTRACTOR algorithm described in Simon
  * D. Levy's Ph.D. thesis. This algorithm is not guaranteed to find
  * all the attractor points, but in practice it finds all or most of
  * them.
  *
  * @author      Simon Levy
  * @author      Anthony Bucci
  * @version     %I%, %G%
  * @since JDK1.2
  */
public class IFS {

    // transforms count
    int m_K;

    // dimensions count
    int m_D;

    // transforms -- weights + biases
    double[][][] m_fW;
    double[][] m_fB;
    // points on attractor
    Vector m_vPoints;

    // resolution
    int m_nRes;

    // supports random seed for reproducibility
    Random random;

    // base for computing hashcodes
    int m_base;

    // flag supporting independent point computation/return
    boolean m_computed;


    /**
     * Creates an IFS with arbitrary initial conditions, using
     * rectangular array of weights. (Simon's preference) Array
     * should have D+1 rows and K*D columns, where D is the number of
     * dimensions and K the number of transforms.
     *
     * @param weights linearized IFS weights
     * @param n resolution for attractor computation
     *
     */
    public IFS(double[][] weights, int n) {
        super();
        random = new Random();
        initialize(weights, n);
    }

    /**
     * Creates an IFS with arbitrary initial conditions, using one-dimensional
     * array of weights. (Anthony's preference)  Weights are linearized
     * by taking usual rectangular weights array and reading top to bottom,
     * left to right.
     *
     * @param weights linearized IFS weights
     * @param k number of IFS transforms
     * @param d number of dimensions
     * @param n resolution for attractor computation
```

```
 * @param seed random seed
 *
 */
public IFS(double[] weights, int k, int d, int n) {
    super();
    random = new Random();
    initialize(weights, k, d, n);
}

/**
 * Creates an IFS with initial conditions fixed via random seed,
 * using rectangular array of weights.  Seed supports
 * reproducibility of results.
 *
 * @param weights linearized IFS weights
 * @param n resolution for attractor computation
 * @param seed random seed
 *
 */
public IFS(double[][] weights, int n, long seed) {
    super();
    random = new Random(seed);
    initialize(weights, n);
}

/**
 * Creates an IFS with initial conditions fixed via random seed,
 * using a one-dimensional array of weights.
 *
 * @param weights linearized IFS weights
 * @param k number of IFS transforms
 * @param d number of dimensions
 * @param n resolution for attractor computation
 * @param seed random seed
 *
 */
public IFS(double[] weights, int k, int d, int n, long seed) {
    super();
    random = new Random(seed);
    initialize(weights, k, d, n);
}
```

```
    /**
     * Find attractor points.  This method is called automatically by
     * {@link #getPoints() getPoints}, but can be called independently to
     * support timing for optimization.
     *
     */
    public void findPoints() {

        // start in center of space
        Point p = new Point();
        for (int i=0; i<m_D; ++i) {
            p.coord[i] = m_nRes / 2;
        }

        // move to attractor
        for(int i = 0 ; i < 25; i++) {
            int k = Math.abs(random.nextInt()%m_K); // random transform
            p = t(k, p);
        }

        // find points by breadth-first search
        Vector vPoints = new Vector(); // point queue
        Hashtable seen = new Hashtable(); // HashSet not yet in Matlab
        Hashtable [] cache = new Hashtable [m_K];  // pt |-> transform(pt)
        for (int k=0; k<m_K; ++k) cache[k] = new Hashtable();
        vPoints.addElement(p); // root of BFS search tree

        while(vPoints.size() > 0) {

            // get point
            p = (Point) vPoints.elementAt(0);
            vPoints.removeElementAt(0); // treats vPoints like a queue

            // run through transforms
            for (int k=0; k<m_K; ++k) {

                Point tp =

                    // if point in cache, use cached value
                    cache[k].containsKey(p) ?

                    (Point)cache[k].get(p) :
```

```
                    // otherwise compute transform of point
                    t(k, p);

                // cache result
                cache[k].put(p, tp);

                // if this transform gives a new point, keep the new point
                if (!seen.containsKey(tp)) {
                    Integer dummy = new Integer(0);
                    seen.put(tp, dummy);
                    vPoints.addElement(tp);
                    m_vPoints.addElement(tp);
                }
            }
        }

        // flag for one-shot point computation
        m_computed = true;
    }

    /**
     * Returns attractor points as 2D array.  First invokes
     * {@link #findPoints() findPoints} if that method has not been invoked
     * already.
     *
     * @return MxD integer array of M D-dimensional attractor points
     */
    public int [][] getPoints() {

        // support one-shot point computation, return
        if (!m_computed) findPoints();

        int [][] points = new int [m_vPoints.size()][m_D];
        for (int i=0; i<m_vPoints.size(); ++i) {
            Point p = (Point)m_vPoints.elementAt(i);
            for (int j=0; j<m_D; ++j) {
                points[i][j] = p.coord[j];
            }
        }
        return points;
    }
```

```
// compute Kth transform of point P
private Point t(int k, Point p) {

    double [] pf = new double[m_D];
    Point p2 = new Point();

    // normalize to (0,1)^D
    for (int i=0; i<m_D; ++i) {
        pf[i] = p.coord[i] / (double)m_nRes;
    }

    // multiply by square weights matrix
    double [] pf2 = new double[m_D]; // initializes to zero
    for (int i=0; i<m_D; ++i) {
        for (int j=0; j<m_D; ++j) {
            pf2[i] += m_fW[k][j][i]* pf[j];
        }
    }

    // add bias vector
    for (int i=0; i<m_D; ++i) {
        pf2[i] += m_fB[k][i];
    }

    // apply sigmoid
    for (int i=0; i<m_D; ++i) {
        pf2[i] = sigmoid(pf2[i]);
    }

    // discretize to [1,N]^D
    for (int i=0; i<m_D; ++i) {
        p2.coord[i] = Math.min(m_nRes - 1, (int) (m_nRes * pf2[i]));
    }

    return p2;
}

// standard sigmoidal squashing function
private double sigmoid(double x) {
    return (double) (1 / (1 + Math.exp(-x)));
}
```

```
// generic initialization
private void initialize(int K, int D, int N) {

    // system params
    m_K = K;
    m_D = D;
    m_nRes = N;

    // storage for points, caches
    m_vPoints = new Vector();

    // use largest hashcode base that doesn't cause integer overflow
    m_base = N;
    while (true) {
        long maxcode = 0;
        for (int i=0; i<D; ++i) {
            maxcode += (long)D * m_base;
            if (maxcode > (long)Integer.MAX_VALUE) break;
        }
        if (maxcode < (long)Integer.MAX_VALUE) break;
        m_base /= D;
    }

    // flag no points computed yet
    m_computed = false;

    // square weights matrices, bias vectors
    m_fW = new double[K][D][D];
    m_fB = new double[K][D];
}
```

```
// initialze for rectangular weights
private void initialize(double [][] w, int n) {

    int cols = w[0].length;
    int D = w.length - 1; // extra row for biases
    int K = cols / D;
    initialize(K, D, n);

    // put weights into square matrices
    for (int i=0; i<D; ++i) {
        for (int j=0; j<cols; ++j) {
            m_fW[j/D][i][j%D] = w[i][j];
        }
    }

    // put weights into vector of biases
    for (int j=0; j<cols; ++j) {
        m_fB[j/D][j%D] = w[D][j];
    }
}

// initialize for Anthony-style weights
private void initialize(double [] w, int K, int D, int n) {

    initialize(K, D, n);

    // put weights into square matrices, vectors of biases
    for (int k=0; k<K; ++k) {
        int koff = k * D * (D+1);
        for (int i=0; i<D; ++i) {
            int ioff = koff+i*(D+1), joff = ioff;
            for (int j=0; j<D; ++j) {
                m_fW[k][j][i] = w[joff];
                joff++;
            }
            m_fB[k][i] = w[joff];
        }
    }
}
```

```
    // internal class for discrete points
    class Point {

        private int [] coord;

        public Point() {
            coord = new int[m_D];
        }

        // we cheat here because this class is never exported
        public boolean equals(Object obj) {
            return (hashCode() == obj.hashCode());
        }


        // hash function linearizes coordinates
        public int hashCode() {
            int key = 0, fac = 1;
            for (int i=0; i<m_D; ++i) {
                key += coord[i] * fac;
                fac *= m_base;
            }
            return key;
        }
    }
}
```

Bibliography

[1] D. H. Ackley, G.E. Hinton, and T.J Sejnowski. A learning algorithm for boltzmann machines. *Cognitive Science*, 9:147–169, 1985.

[2] D. Angluin and C.H. Smith. Inductive inference: Theory and methods. *ACM Computing Surveys*, 15(3):237–269, 1983.

[3] E. Bach, C. Brown, and W. Marslen-Wilson. Crossed and nested dependencies in german and dutch: A psycholinguistic study. *Language and Cognitive Processes*, 1(4):249–262, 1986.

[4] M. F. Barnsley. *Fractals everywhere.* Academic Press, New York, 1993.

[5] R. Bellman. *Adaptive Control Processes.* Princeton University Press, 1961.

[6] G. Berg. A connectionist parser with recursive sentence structure and lexical disambiguation. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 32–37. AAAI Press, 1992.

[7] H. Bey. *T. A. Z.: The Temporary Autonomous Zone.* Autonomedia, New York, 1991.

[8] A. Blair. Scaling up RAAMs. Technical Report CS-97-192, Computer Science Department, Brandeis University, 1997.

[9] I. Bratko. *Prolog Programming for Artificial Intelligence.* Addison Wesley, 1990.

[10] E. Brill. A simple rule-based part-of-speech tagger. *Machine Learning*, 24(3):173–202, 1992.

[11] D.J. Chalmers. Syntactic transformations on distributed represenations. *Connection Science*, 2:53–62, 1990.

[12] N. Chomsky. Three models for the description of language. *IRE Transactions on information theory*, 2:113–124, 1956.

[13] N. Chomsky. *Syntactic Structures.* Mouton, The Hague, 1957.

[14] N. Chomsky. *Aspects of the Theory of Syntax.* MIT Press, Cambridge, Mass., 1965.

[15] N. Chomsky. *Lectures on Government and Binding.* Foris, Dordrecht, 1981.

[16] L. Chrisman. Learning recursive distributed representations for holistic computation. *Connection Science*, 3(4):354–366, 1991.

[17] P.S. Churchland and T.J. Sejnowski. *The Computational Brain.* MIT Press, Cambridge, Mass., 1999.

[18] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer Verlag, Berlin, 1994.

[19] M. Collier. Connectionism and the empiricist program. *Hume Studies*, April/November 1999.

[20] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, Cambridge, Mass., 2001.

[21] G. Cottrell, P. Munro, and D. Zisper. Learning internal representations from grayscale images: An example of extensional programming. In *Proceedings of the Seventh Annual Conference of the Cognitive Science Society*, pages 461–473, Seattle, Washington, 1987.

[22] J.P Crutchfield and K. Young. Computation at the onset of chaos. *Santa Fe Institute Studies in the Sciences of Complexity*, 8:223–269, 1990.

[23] C. Culy. The complexity of the vocabulary of bambara. *Linguistics and Philosophy*, 8:345–351, 1985.

[24] M. Dalrymple. *Lexical Functional Grammar*. Academic Press, New York, 2001.

[25] R. Dawkins. *The Blind Watchmaker: Why the Evidence of Evolution Reveals a Universe Without Design*. W.W. Norton and Co., New York, 1986.

[26] S. Deerwester, S.T. Dumais, G.W. Furnas, T.K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41:391–407, 1990.

[27] R. Eckhorn, R. Bauer, W. Jordan, M. Brosch, W. Kruse, M. Munk, and H.J. Reitboeck. Coherent oscillations: A mechanism of feature linking in the visual cortex? *Biological Cybernetics*, 60:121–130, 1988.

[28] C. Eliasmith and P. Thagard. Integrating structure and meaning: a distributed model of analogical mapping. *Cognitive Science*, 25(2):245–286, 2001.

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

[30] J.L. Elman. Representation and structure in connectionist models. In G. Altmann, editor, *Cognitive Models of Speech Processing*, pages 345–382. MIT Press, Cambridge, Mass., 1990.

[31] I. Farkas and P. Li. Modeling the development of lexicon with a growing self-organizing map. In *Proceedings of the Fifth International Conference on Computational Intelligence*, 2002.

[32] C. Fillmore. The case for case. In E. Bach and R. Harms, editors, *Universals in Linguistic Theory*, pages 1–90. Holt, Rinehart, 1968.

[33] J.A. Fodor. *The Language of Thought*. Crowell, New York, 1975.

[34] J.A. Fodor. Why there still has to be a language of thought. In D. Partridge and Y. Wilks, editors, *The Foundations of Artificial Intelligence: A Sourcebook*, chapter 9, pages 289–305. Cambridge University Press, Cambridge, England, 1990.

[35] J.A. Fodor and B. McLaughlin. Connectionism and the problem of systematicity: Why Smolensky's solution doesn't work. *Cognition*, 35:183–204, 1990.

[36] J.A. Fodor and Z.W. Pylyshyn. Connectionism and cognitive architecture: A critical analysis. *Cognition*, 28:3–71, 1988.

[37] K.-S. Fu and T. L. Booth. Grammatical inference: Introduction and survey. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-5(4,5):95–111,409–423, January, July 1975.

[38] F. Gécseg and M. Steinby. Tree languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, pages 1–68. Springer Verlag, Berlin, 1997.

[39] E. M. Gold. Language identification in the limit. *Information and Control*, 10:447–474, 1967.

[40] J.H Greenberg. Some universals of grammar with particular reference to the order of meaningful elements. In J.H. Greenberg, editor, *Universals of Language*, pages 73–113. MIT Press, Cambridge, Mass., 1966.

[41] J. Higginbotham. English is not a context-free language. *Linguistic Inquiry*, 15(2):225–234, 1984.

[42] G.E. Hinton. Distributed representations. Technical Report CMU-CS-84-157, Computer Science Department, Carnegie Mellon University, 1984.

[43] G.E. Hinton and T.J. Sejnowski. Learning and relearning in boltzmanm machines. In D.E. Rumelhart and J.L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1. MIT, 1986.

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

[45] T. Horgan and J.Tienson. Structured representations in connectionist systems? In S. Davis, editor, *Connectionism: Theory and Practice*, volume 3 of *Vancouver Studies in Cognitive Science*. Oxford University Press, 1991.

[46] T. Horgan and J. Tienson. Representations without rules. *Philosophical Topics*, XVII(1):147–175, 1989.

[47] F. Jelinek, J.D. Lafferty, and R.L. Mercer. Basic methods of probabilistic context-free grammars. In P. Laface and R. De Mori, editors, *Speech Recognition and Understanding: Recent Advances, Trends, and Applications*, volume F75 of *NATO Advanced Sciences Institute Series*, pages 345–360, 1992.

[48] E.D. De Jong and J.B. Pollack. Utilizing bias to evolve recurrent neural networks. In *Proceedings of the International Joint Conference on Neural Networks*, pages 2667–2672, Washington, DC, 2001. IEEE Press.

[49] E.D. De Jong, R.A. Waston, and J.B. Pollack. Reducing bloat and promoting diversity using multi-objective methods. In *Proceedings of the Genetic and Evolutionary Computation Conference*. Morgan Kaufmann, 2001.

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

[51] A.K. Joshi. Processing crossed and nested dependencies: an automaton perspective on the psycholinguistic results. *Language and Cognitive Processes*, 1989.

[52] A.K. Joshi and Y. Schabes. Tree-adjoining grammars. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages and Automata*, chapter 3. Springer Verlag, Berlin, 1997.

[53] L. Kari, G. Rozenberg, and A. Salomaa. L systems. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, pages 253–327. Springer Verlag, Berlin, 1997.

[54] F. Keller. *Gradience in Grammar: Experimental and Computational Aspects of Degrees of Grammaticality*. PhD thesis, University of Edinburgh, 2000.

[55] W. Kintsch. Predication. *Cognitive Science*, 25(2):173–202, 2001.

[56] S.C. Kwasny and B.L. Kalman. Tail-recursive distributed representations and simple recurrent neural networks. *Connection Science*, 7(1):61–80, 1995.

[57] P. J. M. Van Laarhoven and E. H. L. Aarts. *Simulated Annealing*. Reidel, 1987.

[58] J. Lazzaro and C. Mead. A silicon model of auditory localization. *Neural Computation*, 1:47–57, 1989.

[59] A. Manaster-Ramer. Copying in natural languages, context-freeness, and queue grammars. In *Proceedings of the 24th meeting of the Association for Computational Linguistics*, pages 85–89, 1986.

[60] B. B. Mandelbrot. *The Fractal Geometry of Nature*. W.H. Freeman and Company, 1988.

[61] M. R. Mayberry and R. Mikkulainen. Sardsrn: A neural network shift-reduce parser. Technical Report AI98-275, University of Texas, Austin, TX, 1998.

[62] J.L. McClelland, D.E. Rumelhart, and G.E. Hinton. The appeal of parallel distributed processing. In D.E. Rumelhart and J.L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1. MIT Press, 1986.

[63] O. Melnik. personal communication.

[64] O. Melnik. *Representation of Information in Neural Networks.* PhD thesis, Brandeis University, 2000.

[65] O. Melnik, S. Levy, and J.B. Pollack. RAAM for an infinite context-free language. In *Proceedings of the International Joint Conference on Neural Networks*, Como, Italy, 2000. IEEE Press.

[66] O. Melnik and J.B. Pollack. A gradient descent method for a neural fractal memory. In *WCCI 98.* International Joint Conference on Neural Networks, IEEE, 1998.

[67] D. Noelle, G. Cottrell, and F. Wilms. Extreme attraction: On the discrete representation preference of attractor networks. In M.G. Shafto and P. Langley, editors, *Proceedings of the Nineteenth Annual Conference of the Cognitive Science Society*, 1997.

[68] C.H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[69] S. Pinker and A. Prince. On language and connectionism: Analysis of a parallel distributed processing model of language acquisition. *Cognition*, 28:73–193, 1988.

[70] T. Plate. Holographic reduced representations. Technical Report CRG-TR-91-1, Department of Computer Science, University of Toronto, 1991.

[71] T. Plate. Holographic recurrent networks. In *Advances in Neural Information Processing Systems*, volume 5. Morgan Kaufman, 1993.

[72] J.B. Pollack. Connectionism: Past, present, and future. *Artificial Intelligence Review*, 3:3–20, 1989.

[73] J.B. Pollack. Recursive distributed representations. *Artifical Intelligence*, 36:77–105, 1990.

[74] C. J. Pollard and I.A. Sag. *Head-Driven Phrase Structure Grammar*. University of Chicago Press, Chicago, 1994.

[75] M. I. Posner and M. E. Raichle. *Images of Mind*. Scientific American Library, New York, 1994.

[76] P.M. Postal and D.T. Langendoen. English and the class of context-free languages. *Computational Linguistics*, 10(3–4):177–181, 1984.

[77] G.K. Pullum. On two recent attempts to show that English is not a CFL. *Computational Linguistics*, 10(3–4):182–186, 1984.

[78] F. Pulvermüller, W. Lutzenberger, and H. Preissl. Nouns and verbs in the intact brain: evidence from event-related potentials and high-frequency cortical responses. *Cerebral Cortex*, 9:497–506, 1999.

[79] J. Pustejovsky. *The Generative Lexicon*. MIT Press, Cambridge, Mass., 1995.

[80] N. Qian and T. Sejnowski. Learning to solve random-dot stereograms of dense and transparent surfaces with recurrent backpropagation. In D. Touretsky, G. Hinton, and T. Sejnowski, editors, *Connectionist Models Summer School*. Morgan Kaufman, San Mateo, 1988.

[81] E. Rich and K. Knight. *Artificial Intelligence*. McGraw-Hill, New York, 1991.

[82] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[83] P. Rodriguez, J. Wiles, and J.L. Elman. A recurrent neural network that learns to count. *Connection Science*, 11:5–40, 1999.

[84] W.C. Rounds, A. Manaster-Raimer, and J. Friedman. Finding natural languages a home in formal language theory. In A. Manaster-Raimer, editor, *Mathematics of Language*, pages 349–359. John Benjamins, Amsterdam, 1987.

[85] D.E. Rumelhart, G.E. Hinton, and J.L. McClelland. A general framework for parallel distributed processing. In D.E. Rumelhart and J.L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1. MIT, 1986.

[86] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning internal representation by error propagation. In D.E. Rumelhart and J.L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1. MIT Press, 1986.

[87] D.E. Rumelhart and J.L. McClelland. On learning the past tenses of english verbs. In D.E. Rumelhart and J.L. McClelland, editors, *op. cit.*, volume 2. MIT, 1986.

[88] D.E. Rumelhart and J.L. McClelland, editors. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition.* MIT Press, 1986.

[89] K. Sagonas, T. Swift, D. S. Warren, J. Freire, P. Rao, B. Cui, and E. Johnson. *The XSB System Version 2.5 Volume 1: Programmer's Manual.* The XSB Research Group, SUNY Stony Brook.

[90] A. Salomaa. *Formal Languages.* Academic Press, New York, 1973.

[91] W.J. Savitch. Why it might pay to assume that languages are infinite. Technical Report Article 6-4, Center for Research in Language, USCD, 1992.

[92] R.C. Schank and R.P. Abelson. *Scripts, Plans, Goals, and Understanding.* Erlbaum, Hillsdale, NJ, 1977.

[93] M. Schroeder. *Fractals, Chaos, Power Laws : Minutes from an Infinite Paradise.* W.H. Freeman and Company, 1992.

[94] J. R. Searle. *The Rediscovery of the Mind.* MIT Press, 1992.

[95] L. Shastri and V. Ajjanagadde. From simple associations to systematic reasoning. *Behavioral and Brain Sciences*, 16(3):417–494, 1993.

[96] L. Shastri and C. Wendelken. Soft computing in shruti – a neurally plausible model of reflexive reasoning and information processing. In *Proceedings of the Third International Symposium on Soft Computing*, pages 741–747, Genoa, Italy, 1999.

[97] S.M. Shieber. Evidence against the context-freeness of natural language. *Linguistics and Philosophy*, 8:333–343, 1985.

[98] S.M. Shieber. *An Introduction to Unification Based Approaches to Grammar.* Number 4 in CSLI Lecture Notes. University of Chicago Press, 1986.

[99] H. Siegelmann. Computation beyond the turing limit. *Science*, 268:545–548, 1995.

[100] C.A. Skarda and W.J Freeman. How brains make chaos in order to make sense of the world. *Behavioral and Brain Sciences*, 10:161–195, 1987.

[101] P. Smolensky. Tensor product variable binding and the representation of symbolic structures in connectionist systems. *Artificial Intelligence*, 46:159–216, 1990.

[102] A. Sperduti. Labeling raam. Technical Report TR-93-029, International Computer Science Institute, 1993.

[103] M. Steedman. Connectionist sentence processing in perspective. *Cognitive Science*, 23(4):615–634, 1999.

[104] M. Steedman. *The Syntactic Process.* MIT Press, Cambridge, Mass., 2000.

[105] G.L. Steele. *Common LISP: The Language.* Digital Press, Bedford, Mass., 1990.

[106] A. Stolcke and D. Wu. Tree matching with recursive distributed representations. Technical Report TR-92-025, International Computer Science Institute, Berkeley, California, 1992.

[107] W. Tabor. Fractal encoding of context-free grammars in connectionist networks. *Expert Systems: The International Journal of Knowledge Engineering and Neural Networks,*, 17(1):41–56, 2000.

[108] W. Tabor and M. Tanenhaus. Dynamical models of sentence processing. *Cognition*, 23(4):491–515, 1998.

[109] D.S. Touretzky. Boltzcons: Dynamic symbol structures in a connectionist network. *Artificial Intelligence*, 46:5–46, 1990.

[110] W.R. Uttal. *The New Phrenology: The Limits of Localizing Cognitive Processes in the Brain.* MIT Press, Cambridge, Mass., 2001.

[111] T. Van Gelder. Compositionality: a connectionist variation on a classical theme. *Cognitive Science*, 14:355–384, 1990.

[112] S.M. Veres. *Reference of the Geometric Bounding Toolbox Version 7.0.* SysBrain, Southampton, England, May 2001.

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