Representation of Information in Neural Networks

A Dissertation

Presented to

The Faculty of the Graduate School of Arts and Sciences

Brandeis University

Department of Neuroscience

Prof. Jordan B. Pollack, Neuroscience, Advisor

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

by

Ofer Melnik

October, 2000

This dissertation, directed and approved by Ofer Melnik'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:

____

Prof. Jordan B. Pollack, Neuroscience, Chair

____

Prof. Martin Cohn, Computer Science

____

Prof. Blake LeBaron, GSIEF

____

Prof. Nathan Intrator, Brown University

©Copyright by

Ofer Melnik

2000

Acknowledgments

This work is the culmination of various experiences, vision, promising paths, dead ends, encouragement, discouragement, long obsessive nights, immobile deadlines, reruns, respites, family, friends, girl-friends, and a tolerance for microwaved food.

In this journey, many have contributed both directly and indirectly. Here are some of the prominent protagonists.

First, I would like to thank Jordan Pollack, my advisor, for his contributions to my work and graduate school experience. Coupled with his enthusiasm for the important interdisciplinary questions, he has always afforded me the latitude to pursue the research that was genuinely important to me, making my PhD a truly meaningful experience. I am also grateful to him for forming a positive working environment in the DEMO lab, by attracting and keeping talented friendly people and giving them a place to interact.

My final committee members, Martin Cohn, Blake LeBaron and Nathan Intrator for their support, feedback and help making the final stretch as pain free as possible.

My mom Hana and my sister Littal, you've been a great source of support, being there through the various trials and tribulations with optimism and a sense of humor. This has meant a great deal to me.

My dad David, for showing me the way of science, encouraging me to think independently, ask questions and believe in my abilities.

Members of the Neuroscience faculty, in particular, Eve Marder, Larry Abbott, and Sacha Nelson, for being admirable role-models and for putting together such a great program that allows students to grow and realize their interests and potential in a (mostly) worry free and positive environment.

The DEMO lab members, past and present, who are an inspiration to researchers everywhere, original, independent, self-motivated, leaders who follow their own internal research voice: Alan Blair, Hugues Juille, Pablo Funes, Sevan Ficici, Richard Watson, Elizabeth Sklar, Greg Hornby, Paul Darwen, Miguel Schneider-Fonten, Simon Levy, Hod Lipson, Anthony Bucci, Stefano Battiston, Shivakumar Viswanathan, and Edwin de Jong.

The members of my year in the neuroscience program that survived at least the first year, Frances Chance, Ken Leslie, Brian Craft, Dave Rogers, and Bogdan Enoiu, for the best Christmas show ever in the history of the biology department, and for being brothers (and sister) in arms. It was a great group to be a part of.

Tim Martin, my closest friend of these years, who has endured my fluctuations with patience, poise and humor. I can only hope that I gave you half of what you gave me.

My friends Shai Machnes, Rami Shaft and Nadav Gur, who proved that good friendships withstand both distance and time.

Peter Freedman and the members of the dojo, whose consistent positive energy and faith are a perpetual source of inspiration.

And all my other friends and acquaintances at Brandeis and Boston, thank you.

ABSTRACT

Representation of Information in Neural Networks

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

by Ofer Melnik

Artificial neural networks are a computational paradigm inspired by neurobiology. As with any computational paradigm, its strengths are a direct result of how it represents and processes information. Despite being widely used and researched, many questions remain about how artificial neural networks represent information.

Feed-forward networks have seen wide application, but as complex, interdependent, non-linear models the question of assessing the exact computation performed by one has never been fully addressed. This thesis fills that void with a new algorithm that can extract an exact alternative representation of a multi-layer perceptron's function. Using this exact representation we explore scope questions, such as when and where do networks form artifacts, and what can we tell about network generalization from its representation. The exact nature of the algorithm also lends itself to answer theoretical questions about representation extraction in general. For example, what is the relationship between factors such as input dimensionality, number of hidden units, number of hidden layers, how the network output is interpreted to the potential complexity of the network's function.

Building on understanding gained from the first algorithm, a complementary method is developed that while not being exact allows the computationally efficient analysis of different types of very high-dimensional models. This non-specificity to model type and ability to contend with high-dimensionality is a unique feature due to the method's direct focus on the parts of a model's computation that reflect generalization.

The addition of recurrent connections to feed-forward networks transforms them from functions to dynamical systems, making their interpretation significantly more difficult. In fact, recurrent neural networks can not have a "correct" interpretation, as what part of their operation constitutes computation is biased by the observer. Thus the same exact network is capable of performing completely different computations under different interpretations. In this thesis two such interpretations of representation are explored for a four neuron highly-recurrent network. Despite its miniscule size we demonstrate that it can be used, on the one hand, to store and learn highly complex fractal images, or on the other hand, to represent an infinite context-free grammar.

Combining these elements, this thesis advances our understanding of how neural networks compute, both feed-forward and recurrent. It provides a coherent perspective on how to understand and analyze the function of feed-forward networks, and develops new perspectives on computation in recurrent networks.

Contents

Acknowledgments iv

Abstract v

List of Figures viii

1 Introduction 1
- 1.1 Neural Networks . . . 1
  - 1.1.1 Feed-Forward Networks . . . 3
  - 1.1.2 Recurrent Networks . . . 4
- 1.2 Decision Intersection Boundary Algorithm . . . 6
- 1.3 Decision Region Connectivity Analysis . . . 7
- 1.4 Neural Fractal Memory . . . 10
- 1.5 RAAMs can encode infinite context-free grammars . . . 11
- 1.6 Contributions . . . 12

2 Decision Intersection Boundary Algorithm 15
- 2.1 Introduction . . . 16
- 2.2 Network Analysis Methods . . . 19
- 2.3 Decision Regions . . . 21
- 2.4 The Decision Intersection Boundary Algorithm . . . 23
  - 2.4.1 Generative Recursion . . . 25
  - 2.4.2 Boundary Test . . . 26
  - 2.4.3 Traversing the Line . . . 29
- 2.5 Examples . . . 31
  - 2.5.1 Sphere Networks . . . 31
  - 2.5.2 Ball Throwing Network . . . 35
  - 2.5.3 Predicting the S&P 500 Movement . . . 36
  - 2.5.4 Vowel Recognition . . . 37
- 2.6 Discussion . . . 39
  - 2.6.1 Algorithm and Network Complexity . . . 39
  - 2.6.2 Extensions to the Algorithm . . . 42
  - 2.6.3 Rule Extraction and Network Validation . . . 43
  - 2.6.4 Sigmoidal Activation Functions . . . 44
  - 2.6.5 Generalization and Learning (Proximity and Face Sharing) . . . 45
- 2.7 Conclusion . . . 48

3 Decision Region Connectivity Analysis **54**
3.1 Introduction 55
3.2 Low Level Analysis 59
3.3 Analyzing a three-dimensional neural network 61
3.4 Higher Level Graph Analysis 64
3.5 Comparing two classifiers: A high-dimensional example 68
3.6 Learning from lower dimensions: A high-dimensional example 72
3.6.1 PCA on Ellipsoids 72
3.6.2 The task and classifier 74
3.6.3 PCA Interpretation 75
3.6.4 The Analysis 77
3.6.5 The ellipsoids as a model 79
3.7 Analysis Method Details and Discussion 81
3.7.1 Low-Level Analysis 81
3.7.2 Higher-Level Analysis 82
3.7.3 Convex Region Analysis 84
3.8 Extending and Generalizing the Method 85
3.9 Conclusion 86

4 Neural Fractal Memory **87**
4.1 Introduction 88
4.2 Architecture 90
4.3 Learning 91
4.4 Evaluation and Discussion 94

5 RAAM for Infinite Context-Free Languages **99**
5.1 Introduction to RAAM 100
5.2 Problems in RAAM 102
5.3 New RAAM Formulation 104
5.4 Hill-Climbing an $a^n b^n$ decoder 105
5.5 Competence model and proof 107
5.6 Discussion 113

6 Discussion **114**

List of Figures

2.1 A 3-layer perceptron neural network. 22
2.2 The generative portion of the DIBA algorithm recursively projects the hidden layer hyperplanes down till they are one-dimensional, in order to generate all the intersections of the hyperplanes. Here we see an illustration of two stages of projection from three dimensions to two and from two to one. 24
2.3 The boundary testing portion of the DIBA algorithm evaluates individual vertices and line segments, that were generated by the recursion, in order to test whether they form corners or edges of decision regions. Here we see the edges and corners of two decision regions the algorithm would recognize in an arrangement of hidden unit hyperplanes. 24
2.4 The decision region formed by the division of a two dimensional space by a hyperplane line. The line could be said to be partitioning the space. Or, in a complementary, functional fashion it could be said that the form of the boundary in the space is in the shape of a line. 27
2.5 Some possible decision regions formed at the intersection of two lines. Corners are only formed if both lines are boundaries. 28
2.6 The partitioning of a 2 and 3 dimensional space around the intersection of hyperplanes. 28
2.7 Unlike two dimensional spaces, a corner in a three dimensional space can be in the middle of multiple line segments. 29
2.8 The bi-directionality of hyperplane intersections on a line suggests a two pass algorithm to calculate the partial sums of the output unit weights. 31
2.9 A decision boundary of a network which classifies points inside and outside of a circle. 32
2.10 A zoom out of the network illustrated in figure 2.9 illustrates the existence of artifactual decision regions. 33
2.11 A decision boundary of a network which classifies points inside and outside of a sphere. 33
2.12 The artifacts around the sphere decision boundary. 34
2.13 Projections of the four dimensional hyper-sphere polytope across $x_4 = 0$ and $x_4 = 1$. 35
2.14 The network is supposed to predict whether the ball will hit the target given throwing angle β, initial velocity v and target distance x. 35
2.15 The decision regions of the S&P 500 prediction network. 37
2.16 The decision region of the ball throwing neural network (left) contrasted with the decision region of the actual analytical solution (right). Two different perspectives are shown. 38

2.17 The decision regions of the two input vowel recognition network with the training data in the background. The X's are positive test cases. 40
2.18 A hand constructed neural network which demonstrates the potential number of decision regions a network can have. 41
2.19 A sigmoidal activation function and its piecewise linear counterpart. 46
2.20 The intersection of two piecewise linear border hyperplanes forms an intermediate border face. 46
2.21 The decision regions of back-propagation networks trained on the decision regions of figure 2.18b. 48

3.1 Examples of some of the variations possible in decision region structure. . . 56
3.2 a) The connectivity graph is generated by sampling between the sample points. In this case we see how sampling between points A and B detects a boundary, but points A and C share a neighborhood. b) A connectivity graph for two decision regions, one convex and one concave. 60
3.3 The classification task of the ball throwing network is to predict whether a ball thrown at a certain velocity and angle will hit a target at a given distance. 62
3.4 The decision region of the ball throwing network contrasted with the analytic decision region. 62
3.5 a) The points extracted from the labeling in the connectivity graph superimposed in their correct position within the decision region. b) The connectivity graph of the decision region in figure 3.4 with respect to 78 internal points. The vertices are labeled by association to four different clique like clusters. . 63
3.6 The eigenvalues of the PCA analysis for each of the groups contrasted with the eigenvalues of all the points taken together. The decomposition into groups allows us to realize that the points in the decision region form a two-dimensional embedding in the space. This would not have been discernible by just performing a PCA on all the points together. 64
3.7 a) A concave decision region housing nine points in different convex subcomponents. b) The connectivity graph for the points in the decision region. c) The group graph associated with the labeling in the connectivity graph. . . 65
3.8 An example of two different concave decision regions and their respective group graph. 67
3.9 The connectivity graph of a 64-input neural network trained to classify the numerals 3 and 4 as one class and the other numerals as another class. The graph clearly illustrates that the network constructed a decision region with two separate subclasses. 69
3.10 For the two labeled sets in the connectivity graph in figure 3.9 a PCA analysis was done to estimate the extent of the convex subregions, and two extreme values are shown. As seen, group A contains threes, and group B contains fours. 69
3.11 The connectivity graph of a K-nearest neighbor classifier set to classify the numerals 3 and 4 as one class and the other numerals as another class. The labeling of the graph suggests a weaker concavity than the neural network's decision region. 71
3.12 A PCA analysis was done to estimate the extent of the convex subregions of the KNN connectivity graph. Two extreme values are shown for each region analyzed. The subregion containing groups A and C is suspicious. 71

3.13 The group graph of the six input SVM model classifying the van class from the vehicle dataset. 75
3.14 This figure shows the PCA analysis of each of the ellipsoids used to approximate the convex regions of the SVM classifier. The hi-lighted values are discussed in the text. 76
3.15 There is no absolute sampling frequency which will guarantee the detection of all decision boundaries. In this figure we see that the closer we come to the corner the higher the sampling rate must be to detect the boundary. . . . 82
3.16 The point at which the arrow is pointing has a unique connectivity signature which would preclude including it in any group. However, it is part of the convex region defined by the top 3 points. 84

4.1 Example fractal attractors that can be generated with a four neuron network. 89
4.2 The neural network architecture consists of four neurons, each pair acting as a transform. 90
4.3 This figure shows some example run results, on 16 by 16 attractors. The first for examples are subjective successes, while the last two leave something to be desired. 97
4.4 This is a histogram of changes in the Hausdorff distance after running the gradient descent algorithm. It was compiled across all 4000 trials. Each of the different graphs represent trials at different initial error levels. 97
4.5 Each graph represents a histogram of either the initial or final error across each of the three error measures. These were conducted for an initial error level of 4.0. 98

5.1 An example RAAM decoder that is a 4 neuron network, parameterized by 12 weights. Each application of the decoder converts an (X, Y) coordinate into two new coordinates. 100
5.2 The dynamics of a RAAM decoder as it decodes the tree (1 (1 2)) and its daughter tree (1 2). The left transform is shown as a dashed line, and the right transform as a straight line. 102
5.3 The equivalence classes of two solutions to $a^n b^n$ found by hill-climbing. . . . 107
5.4 An arrow diagram of a tree starting at (0.7,0.3). 108
5.5 A typical tree generated by a zigzag RAAM. 109
5.6 The line that divides the space of points that go to the upper right and lower left. 110
5.7 A terminal point arrow diagram for $\varepsilon = 1/64$ and $c = 5.703$. 111
5.8 A one-dimensional map diagram of the zigzag line for two values of c. 112

Chapter 1

Introduction

1.1 Neural Networks

The focus of this thesis is on questions of how neural-networks represent information. Even though neural networks are an example of a Turing equivalent computational system [59](or more [78]) or a universal approximator [27; 34], 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.

Neural networks mean different things to different people: Neuro-scientists see them as a model to explain biological data [1], cognitive scientists see them as formalism to capture cognitive processing [80], some computer scientists see them as a step towards machine intelligence [68], statisticians see them as interesting non-linear models [8], engineers use them in applications requiring non-linear processing of continuous data [31], and data-mining analysts use them to find regularities in massive databases [92]. The approach adopted in this research is to treat neural networks as interesting computational models, asking, can we understand what these models do? And, what kinds of computational information are

they well suited to represent and process?

Neural networks, the substrate for all high-level biological processing, are naturally an interesting computational paradigm for study. But, what has provided an impetus for their study these past two decades is the realization that even small simulated neural networks [30; 48] are adept at tasks like perceptual processing, pattern matching and associative memory [31], tasks that have traditionally been difficult for classical programming paradigms.

What is it about how neural networks compute that makes them well suited for these tasks? What is special in the way that they represent and process information? Traditionally, these have not been easy questions to answer, as the same intrinsic computational properties that make neural networks powerful also make them difficult to analyze. A neural network is effectively a large network of highly-interconnected non-linear processing units. It is by having multi-dimensional, tightly interdependent processing units that neural networks can extract relationships from high-dimensional continuous data. However, these very properties are also what make them difficult to analyze:

- The high interdependence between processing units implies a lack of decomposability. That is, a network can not typically be broken down into modular subparts to facilitate analysis. Rather, the network needs to be considered as a whole for each computation.
- The difficulty in "assigning credit" [55] to particular parameters due to the non-linearity of the computation, again forces a multi-dimensional examination of a large number of interdependent parameters concurrently.

Although the analysis of recurrent networks also needs to contend with contextualization in the form of iterative dynamics, the previous two issues are the ones that have plagued the analysis of all neural networks, recurrent and feed-forward. Thus hampered, the process of understanding the computational mechanisms of neural computation has taken multiple avenues of research. To start, let us focus on feed-forward networks, the most successful of neural network models.

1.1.1 Feed-Forward Networks

Purely theoretical research into feed-forward networks has tried to address the computational power of networks in general, for example by demonstrating their universal approximation properties [34] or by constraining the class of functions that finite networks can compute using measures from learning theory such as the VC dimension [90]. While theoretical constraints are interesting, they typically serve as bounds and do not directly address how actual networks compute.

Various concepts have been proposed to describe what kinds of computation networks perform. Some of these are vague, others more concrete. The idea of *receptive fields* [48] was borrowed from neurobiology [41], it asserts that individual neurons are tuned to detect distributions of patterns in their inputs. Thus, the hierarchy of neurons in a feed-forward network relates to multiple levels of pattern detection. This hierarchical processing concept is a common one that is generally referred to as "*hidden layer representations*" [21]. It postulates that layers of neurons act to find alternative representations of their inputs, where successive layers are believed to incrementally refine these representations to higher-levels [32]. An alternative explanation for the function of the hidden or feature space is found in the theory behind *Radial Basis Function* networks [13] and to some degree *Support Vector Machines* [90]. There, Cover's Theorem [19] is exercised to state that the function of the first layer of neurons (the hidden layer) is to project the otherwise lower-dimensional inputs to a much higher dimensional space where the next layer of neurons can more easily separate the different input patterns (linearly separate). This notion of separability is also present in the understanding that neural networks form decision regions in the input space [35]. *Hyperplane analysis* [73] describes the function of neurons as separating their inputs across a hyperplane in their input space, and suggests that this is done incrementally across layers.

While these approaches may offer some perspectives on what computation is in these networks, they do not help much in understanding specific networks. Initial attempts to analyze existing networks were *weight visualization* approaches, such as *Hinton Diagrams* [67], and *activation visualization* approaches, such as *Hierarchical Clustering* [30]. To date,

the only general approaches that have been able to shed light on the computation performed by non-toy-sized-networks are the *Rule-Extraction* approaches [2]. Rule-extraction tries to find an alternative representation of network function, typically as symbolic rules. That is, it tries to capture the function that a network performs in an alternative model, one which is hopefully simpler to understand. Although there have been many approaches to rule-extraction or representation-extraction [2; 20; 72; 86], the field has lacked grounding. It has focused more on the nature of the rules it extracts than on proving the relationship between the rules and the networks it analyzes. While it has had successes at generating rule sets with similar performance scores to counterpart neural networks, proving that the rule set completely and efficiently captures the computation performed by a network has not been done except for networks with very specific constraints on their topology [11; 3]. Thus, despite the plethora of neural network research, the following two core questions have not been adequately addressed in the literature:

1. How do you satisfactorily analyze existing neural networks?
2. What are the aspects of a network's computation that are salient for understanding its function?

The motivation behind the algorithms for network analysis developed in chapter 2 and 3 is to address these holes in our understanding of neural networks. As discussed in the following sections, the Decision Intersection Boundary Algorithm (DIBA) of chapter 2 fills the gap of an algorithm that can generate an alternative representation of network function which is exact and concise, allowing us to analyze a network's function completely. While the Decision Region Connectivity Analysis (DRCA) method of chapter 3 is a method that focuses on the salient aspects of a network's computation, using understanding about network computation gathered from the DIBA algorithm.

1.1.2 Recurrent Networks

Even though there are holes in our understanding of feed-forward networks, recurrent neural networks remain a greater mystery. This is not due to an oversight of the research com-

munity, rather they are inherently much more complex. Adding recurrent connections to a neural network, transforms it from a function to a dynamical system, and links recurrent networks with concepts like chaos, strange attractors, and fractals [82], placing them in a mathematical domain that is just now being developed.

Mathematics aside, from a computational perspective recurrent networks have inherently tougher issues with representation than do feed-forward networks. Whereas in a feed-forward network there is a clear delineation of the inputs, outputs and the process or mapping between the two, in a recurrent network it is not obvious what aspect of the network's dynamics is actually relevant to computation. Do we label the fixed-points of the network's dynamics as associative memories as in Hopfield type networks [33]? Or, do we consider the the dynamics as transitions in a state machine [28]? Or are the transitions iterations in a large data structure [60]? There are obviously many possible ways to interpret the computation performed. And, in fact, the observer's bias endows a meaning of its own to the computation [45]. From the perspective of neuroscience, this is an inherent obstacle, since it is not obvious where and in what form in the brain's recurrent network information is stored and processed. From an artificial neural networks perspective there is no "right representation", as different approaches may have different applications. Therefore, explorations into the different ways that recurrent networks may represent information is a natural direction of study into their underlying capabilities.

The material of chapters 4 and 5 considers the same small recurrent neural network from two different perspectives of what is being represented by the network. In chapter 4 the network is interpreted as storing "visual memories" in its underlying fractal attractor. The question addressed is whether this is a learnable representation, whether a learning algorithm can be developed to allow the networks to learn new attractors. Chapter 5 views the same network as a Recursive Auto Associative Memory [60], a representation that stores recursive data structures (grammar like) within the recursion of a network's dynamics. Here we are interested in the capacity of the network, what kind of structures can it store in its dynamics. Specifically, we prove that certain infinite context-free grammars can be encoded within its dynamics.

Following are introductions to each of the main contributions of this thesis, in the same order that they will be presented.

1.2 Decision Intersection Boundary Algorithm

In chapter 2 we present the Decision Intersection Boundary Algorithm (DIBA), a new algorithm to extract exact representations from feed-forward threshold networks. As previously discussed, the question of understanding a neural network's function dates back to at least Minsky and Papert's seminal Perceptron [55] where the question was posed as, how do we attribute a network's function to its individual neurons (credit assignment problem). In this work, an algorithm is developed to extract an exact concise alternative representation of network function in the form of polytopic decision regions. To explain, consider that the vector of inputs to a network defines an input space, a space of all possible inputs, then what this representation captures is: for what locations in the input space does the network make a specific "decision" or give a specific output. Due to specific properties of the networks the geometrical shape of these decision regions are polytopes, the high-dimensional extension of polygons. Thus what the DIBA algorithm does is to find the borders of these polytopes which are enough to completely describe them.

This is the first example of an algorithm able to extract exact and concise representations from continuous input, arbitrary topology, threshold feed-forward networks. Recently others have contributed algorithms that also use polytopic decision regions as an alternative representation [51], however their algorithms are not exact and concise, limiting their applicability to the questions of theory and scope that can be answered by the DIBA algorithm, as described next.

While Golea [29] has demonstrated that representation extraction from discrete input networks is NP-hard, little work has been done to explore the computational complexity of representation extraction in general. The fact that the DIBA algorithm is exact and concise allows us to answer theoretical questions about the complexity of network representations in general. Using the computational complexity of the algorithm as a measure for the

potential complexity of a network's representation, we examine how this measure is affected by variations in the input layers size, hidden layers size, number of hidden layers and output interpretation. This analysis leads to a better understanding of how these factors affect the function a network can potentially represent, and consequentially the computational cost of understanding what function a network is representing.

Unlike other representations which only approximately cover a subset of the network's function, as an exact algorithm, the other main contribution of the work is that it allows an objective analysis of the function of different networks. A complete alternative representation allows us to truly analyze what a network is or is not doing without fear that we are missing aspects of its computation or sugar-coating others. For example, by examining their extracted representation, we can identify decision region properties that networks exhibit when they generalize well as opposed to when they do not generalize. By isolating these specific properties we not only gain a better understanding of how neural networks compute, but these properties form the foundation of another representation extraction algorithm, described in chapter 3.

Another thing that can only be done with an exact representation is that we can examine something that is hardly addressed in the neural network literature– noise in neural networks. Under what conditions does noise appear, and what form does it assume in the network's decision regions.

1.3 Decision Region Connectivity Analysis

The "Curse of Dimensionality" [6] is a term used to describe different problems in modeling [8]. In the case of model analysis it strikes on two fronts: 1) The potential complexity of a model (e.g., the number of decision regions, as shown in the second chapter) can be exponential in the number of its inputs. 2) Our understanding of irregular high-dimensional structures (e.g., decision regions) is severely constrained by our own perceptual handicaps, mainly that we can not effectively "see" beyond three dimensions.

While the first problem has rarely been dealt with directly in network analysis, it has

usually been implicitly addressed by constraining the analysis. This is done either by performing directed sampling [20] in a subregion of the input space and/or constraining the model type [58] to one with limited complexity.

The difficulty in interpreting high-dimensional information appears in many domains. General approaches to high-dimensional data visualization range from Draftsman's Displays [53] which produce plots of all pairs of variables simultaneously, through Glyph scatter plots [24] or Chernoff faces [46] which use adjustable symbols to represent higher-dimensional data, to Andrew's curves [53] that use functional representations to differentiate between high dimensional data points. Another approach with a long history is to assume that the data can be reduced to a lower dimensional representation. The most common example of this is Principal Component Analysis [40]. Rule-Extraction is, in effect, also another approach, since it attempts to describe an alternative representation of a high-dimensional model using symbolic rules, typically if-then-else rules describing the domains of the input space. All of these approaches, being relatively general methods, can assist in taking an incremental step in understanding higher-dimensional information. However, as general methods their utility is limited, since models with more than just a few extra dimensions, or larger amounts of data, overload these methods.

The Decision Region Connectivity Analysis (DRCA) method described in chapter 3 is a method to analyze decision region type classifiers that directly addresses both of these issues of high-dimensionality, model complexity and visualization. In part, it is motivated by the analysis in chapter 2 of the basic properties of the decision regions of neural networks. It is thus motivated in the sense that the analysis served to illustrate what aspects of a neural network's decision region structure were salient to understanding its generalization strategy. Therefore, the DRCA method is as much a filtering tool as it is an analysis method, zeroing in on the important parts of a model's decision region structure while ignoring artifacts.

Considering that the "learning" or fitting performed by a classifier on training data results in the enclosure of the training points within decision regions, the DRCA method confronts the first dimensionality issue of model complexity by only looking at the "relevant" structure of a classifier. That is, it focuses on the decision regions only as they pertain to how they

enclose the training points, avoiding extraneous complexity or artifacts. Since, if a model generalizes it must have located patterns or redundancies in the data, or restated, that the relevant complexity of the model must be lower than that of the data it was trained on. Then, by focusing only on the "relevant" complexity, the DRCA method directly deals with the first dimensionality issue by extracting a representation that is typically much simpler than the model or the data it was trained on.

The second dimensionality issue of visualization is also surmounted by focusing on the relationships imposed on the training data by their enclosure in decision regions. As shown in chapter 2, the essential aspects of a classifier's function are the shape of its decision regions with respect to its training points. For example, variations in decision region structure of interest are: the number of decision regions, whether they are convex or concave, and if they are concave can they be decomposed into convex subcomponents. The DRCA uses a new representation, a graph representation (edges and vertices), that can clearly convey these essential relationships between the training points. In this representation, there is a direct correspondence between properties of the graph, such as unconnected sub-graphs, cliques and connectivity density, to geometrical properties of the underlying decision regions, such as unconnected decision regions, convexity and concavity. The main advantage of the graph representation over its decision region counterpart is that it is a dimension-less mathematical structure. That is the key to how the high-dimensional visualization issue is negated, since the graph, which can be displayed two-dimensionally, allows us to examine the essential aspects of classifier function, independently of its input dimension, allowing the analysis and visualization of very high-dimensional classifiers.

As shown in the examples in the chapter, the main contribution of the DRCA method is that it allows the computationally efficient analysis of decision region based classifiers independently of model type or input dimensionality. Thus, it allows the comparison of completely different kinds of models at the core level of how they use decision regions to generalize from the training data.

1.4 Neural Fractal Memory

Unlike feed-forward networks which are clearly defined as a mapping between inputs and outputs, what computation is performed by a recurrent neural network is as much driven by the observers interpretation of its function [45] as it is driven by a networks underlying dynamics. Chapters 4 and 5 both study the same highly-recurrent neural network. The difference between them is in the observer's interpretation of what computation is taking place in the network.

In chapter 4 the network is interpreted to be storing a memory within the fractal attractor of its dynamics. The relationship between neural networks and fractals is a straight forward one– as highly-recurrent neural networks are really examples of iterated function systems, the mathematical form of fractals [4]. While other researchers have recognized this relationship before [60; 83; 56; 87; 84], none have yet devised an adaptive learning mechanism that can exploit this highly rich representation. Instead, this mathematical insight into recurrent dynamics has remained a primarily didactic device about the dynamics of computation in recurrent neural networks, with a few interesting hard coded applications [87; 84].

Without a way to find network weights for specific attractors, the notion of memories being stored in the attractor remains unworkable. Thus the motivation behind the work in chapter 4 was to see if this representation can actually be made into a viable model, a model with adaptive learning, in the spirit of other neural-network models. What is developed in chapter 4 is a learning algorithm for networks that store memories in their fractal attractor. Like the ubiquitous back-propagation [66], this is a gradient descent method, where an error function is developed that, using its gradient, allows for incremental weight adjustments on the network. This seems like a non-trivial problem as the process of generating the attractor does not lead to a differentiable function, because it is iterative. However, as shown in the chapter, exploiting the natural self-similarity of fractal attractors in the form of the *collage theorem* [4] and concepts of mutual overlap from the *Hausdorff distance* [4], leads to a tenable error function. This error function actually has a significant success rate across a bank of various attractors and different noise levels, with just simple gradient descent.

1.5 RAAMs can encode infinite context-free grammars

People have been interested in the application of neural networks to structured information, specifically towards natural language processing, since their inception [54; 80]. Pollack [60] formulated an original approach for recurrent networks to store structured information, the RAAM model. In this model the recurrence of the network coincides with the relationships between elements in the data structure. That is, each activation of the recurrent connections in the network implies a transition in the data structure that it is representing. While this model answered the question of how neural networks could potentially perform symbolic computation, and generated much interest [10; 57], its development plateaued before clearly defining its strengths and capacities.

A central question for connectionist symbolic processing has been where in the Chomsky language hierarchy [17] do these models fit. Are they regular, as Casey [16] has demonstrated for noisy dynamical systems, or are they context-free or context-sensitive? The transition between regular and context-free has been of particular interest as it demonstrates a certain infinite memory capacity. Rodrigez, Wiles and Elman [64] have explored how the dynamics of a predictor network allow it to recognize a context-free language, $a^n b^n$. Moore [56] showed how to construct Dynamical Recognizers [61] that can recognize arbitrary context-free languages using the Cantor sets [82].

In this work we look at a RAAM decoder model whose network is identical to the one used in chapter 4. Using understanding of the underlying dynamics of this network, Pollack has reformulated the original RAAM model in a way that more naturally reflects these dynamics. By so doing the major “bugs” that caused inconsistencies in RAAM function are alleviated. My contribution is to prove that this new RAAM formulation, with its now more rigorous foundation, is powerful enough to generate infinite context-free grammars of the $a^n b^n$ variety. Thus, it demonstrates the computational power laden in this kind of interpretation of the representation of recurrent neural networks.

1.6 Contributions

As previously described, this thesis addresses fundamental questions of representation in both feed-forward and recurrent neural networks. Following is an itemized listing of the individual contributions of this thesis:

1. Developed the Decision Intersection Boundary Algorithm (DIBA), an algorithm that can extract exact and concise representations from threshold feed-forward networks, in the form of polytopic decision regions. This is the first example of an algorithm capable of extracting completely exact and concise representations.

2. Used the DIBA algorithm to explore the effects of artifacts on neural network decision regions. Showed that artifacts are common and their effects are exacerbated in higher-dimensional models. This is an issue that otherwise has hardly been directly addressed in the neural network literature.

3. Used the DIBA algorithm to examine networks with both good and bad generalization. Noted, for both cases, what properties in a network's decision region structure are indicative of its generalization strategy, allowing us to abstract away the aspects of network computation that are directly relevant to good generalization.

4. Generalized the DIBA algorithm to multiple hidden layers and arbitrary output interpretation schemes. In so doing demonstrated that the underlying potential complexity of the network is independent of those two factors.

5. Demonstrated by construction how a neural-network can generate an exponential number of complex decision regions in its input space. As such, showing the inherent computational complexity involved in exact and complete representation extraction from these types of models.

6. Decomposed the generalization mechanisms of these types of networks into *proximity* and *face-sharing*. Suggested by example that typical local weight modification learning approaches such as back-propagation are more likely to exploit proximity generalization than face-sharing.

7. Using the concepts of how networks use decision regions to generalize from their training data, seen using the DIBA algorithm, the aspects of decision region structure important a understanding a model were outlined, motivating the development of Decision Region Connectivity Analysis (DRCA).

8. The low-level analysis part of the DRCA method was developed, allowing the structure of a model's decision regions, as it pertains to the training data, to be extracted and represented as a mathematical graph. This a novel representation whose main strengths are its model and dimensionality independence. Thus, it allows completely different types of very high-dimensional models to be easily analyzed at the core level of their decision regions.

9. The high-level analysis part of the DRCA method was developed, which serves to extract the core decision region structure information form the low-level graphs. Thus, it allows even very dense graphs, representing many training points, to be efficiently analyzed and understood.

10. Showed how the DRCA method can be used to analyze and compare completely different types of very high-dimensional models, using a character recognition example.

11. Explained and demonstrated how using ellipsoid based PCA, important non-visual information about a model's classification strategy can be extracted. This shows the general applicability of the DRCA method.

12. Showed how the connectivity graph of an analyzed classifier can be used as a blue-print to construct an alternative classifier, typically simpler, with similar functionality.

13. Showed how this same graph blue-print can be used to take structure that exists in a lower input dimensional model and transfer it to create a higher-dimensional model with the same structure. This suggests an avenue of thought on how to potentially overcome some of the effects that the "The Curse of Dimensionality" has on model construction algorithms.

14. Devised a continuous error function in the weight space of a four neuron highly recurrent neural network, that allows the comparison of a desired fractal attractor with the network encoded attractor. Even though the fractal dynamics in these networks were previously recognized, no method existed to do learning for them.

15. Approximated the derivative of the error function to serve as a basis for any online gradient descent type learning algorithm for these networks.

16. Using simple learning constant decayed gradient descent this learning approach was tested in 4000 trials of different desired attractors and noise conditions. Using visual inspection and three test measures, the learning showed significant success, both visually and statistically, demonstrating the validity of the approach.

17. Using the analysis of a hill-climbed RAAM model that did a subset of the $a^n b^n$ language, a parameterized competence model was developed that exclusively generated words from the $a^n b^n$ language, for any attractor discretization level. This proves that such a model exists for any resolution of the space.

18. Using a one-dimensional map analysis of the trajectories on such a competence model it was proven that for certain settings of the competence model parameters a RAAM is capable of generating the complete infinite $a^n b^n$ language. This is the first known proof that a standard RAAM model can express a basic context-free language.

To summarize, through the development of new algorithms for network analysis, critical examination of real networks, refinement of generalization principles, innovative approaches to contend with "The Curse of Dimensionality", development of learning under fractal dynamics, proof of the capacity of RAAM, this thesis advances our understanding of both feed-forward and recurrent neural networks. It provides a coherent perspective on how to understand and analyze the function of decision region based models, while pushing new directions in recurrent network computation.

Chapter 2

Theory and scope of exact representation extraction from Feed-Forward Networks

In this chapter an algorithm to extract representations from feed-forward perceptron networks (threshold) is outlined. The representation is based on polytopic decision regions in the input space– and is exact, not an approximation. Using this exact representation we explore scope questions, such as when and where do networks form artifacts, or what can we tell about network generalization from its representation. The exact nature of the algorithm also lends itself to theoretical questions about representation extraction in general, such as what is the relationship between factors such as input dimensionality, number of hidden units, number of hidden layers, how the network output is interpreted to the potential complexity of the network's function. The material in this chapter have appeared in conference form at the International Joint Conference on Neural Networks 2000 (IJCNN, sponsored by the IEEE and INNS). A journal version of this material has been submitted to *Cognitive Systems Research.*

2.1 Introduction

There are important reasons to analyze trained neural networks: For networks deployed in real-world applications, which must be predictable, we want to use network analysis as a verification tool to gauge network performance under all conditions. In *Data Mining* applications, where networks are trained in the hope that they may successfully generalize, and in so doing capture some underlying properties of the data, we want to use network analysis to extract what the network has learned about the data. Or conversely, when a network fails to generalize, we want to use network analysis to find the causes of its failure.

However, despite being prolific models, feed-forward neural networks have been ubiquitously criticized for being difficult to analyze. There are multiple reasons for this difficulty: First, since neural networks consist of a large quantity of interconnected processing units that continuously pass information between them, there is a high degree of interdependence in the model. This implies a lack of locality, where small perturbations in one location can affect the complete network. As such, it is not possible to modularize a network's functionality directly with respect to its architecture, rather it has to be analyzed as a whole. Second, the input dimensionality of the network implicitly limits its comprehensibility. Most people's intuition resides in two or three dimensions. Higher dimensionality impedes our ability to comprehend complex relationships between variables. Third, the processing units are non-linear. As such, they are not easily attacked with standard mathematical tools.

Different approaches have been proposed to analyze and extract representations from neural networks (covered in the next section). But other than Golea's [29] proof of the NP-hardness of discrete network analysis, little work has been done to address questions of theory and scope for representation extraction in general. Some of these general questions relate directly to the previously discussed reasons for network analysis, including: How does the computational cost of verifying a real-world neural network scale? What are the common properties of networks that generalize well and those that do not? The focus of this chapter is to address these and other questions. Specifically, on the theory front we are interested in the following questions:

1. How does the potential complexity of a neural network change as a function of the number of its input dimensions and its hidden units? This relationship acts as a lower bound on the computational complexity of any representation extraction algorithm.

2. What are the computational costs of calculating the exact error between an approximate representation and the actual function that a neural network is computing? These costs act as a bound on the difficulty of verifying the correctness of an extracted representation.

3. How does the number of hidden layers, and the interpretation of the network outputs affect what a network can do? This correspondence is related to how the quality of a representation might change across different network models.

The questions of scope address what aspects of a network's function can be inferred from an extracted representation. Some questions we address are:

1. How can we gauge the potential generalization properties of a network by examining its extracted representation? In general, what properties might a network with good generalization exhibit that can be detected in its representation?

2. Where can we expect the network to exhibit unpredictable behavior or artifacts? What is the form of these artifacts that can be detected in an extracted representation?

The tool that we use to address these fundamental questions of theory and scope is a new algorithm able to extract exact and concise representations from feed-forward neural networks. By being an exact algorithm its properties reflect directly on all other representation extraction algorithms. Since any properties it exhibits or illustrates are inherently related to properties of the underlying networks it analyzes, and as such delimit the performance of representation extraction in general. Such an algorithm will help address the theory questions in the following ways:

1. The computational complexity of the representation extraction algorithm is related to the potential complexity of the network.

2. The computational cost of performing measurements on the exact representation is related to the cost of estimating how well other representations capture a networks function.

3. How the computational complexity of the algorithm changes when hidden layers are added, or how this computational complexity changes when the network output is interpreted differently, are both directly related to how these variations affect the network's potential complexity.

The questions of scope require that the representation will allow us to understand what features of a network's function are conducive or detrimental to good performance and generalization. Only by having an exact representation can we be sure that we are examining all the relevant aspects of a network's function, and not missing information. The scope questions are approached by example– by examining the representations of different networks. Specifically, these questions are addressed in the following ways:

1. By having a trustworthy representation, and some intuition about what generalization entails, we can compare and contrast the representations of networks with good and poor generalization to elucidate common properties of both.

2. By scrutinizing the representations of networks we can determine the form that deviations from desired function or artifacts take on. Ideally we would like to be able to predict what network situations are more likely to introduce artifacts.

This chapter is structured into three main parts: a description of the algorithm, an examination of the scope questions through example, and a discussion of the theoretical results. We start by presenting the goals of the algorithm and contrast those with other network analysis methods, leading to an examination of the principles of network computation on which the algorithm are based. The algorithm is then described in its basic form for single hidden-layer, single output threshold multi-layer perceptrons.

The example sections begin with examples that serve to introduce the representation while addressing the kind of artifacts present in multi-dimensional networks. We then give

examples that contrast representations of networks that do generalize well and those that do not.

The theoretical discussion first addresses the complexity of the basic algorithm and the relationship to potential network complexity. This complexity discussion is continued by describing how the algorithm can be extended to multi-layer multi-output networks and its computational implications, and also by examining complexity from the perspective of representation validation. We then discuss the differences between threshold and sigmoidal networks, where we examine what effect this change of activation function has on their representational ability. We conclude the discussion by tackling the issue of whether learning algorithms can successfully exploit all the potential complexity available in these networks.

2.2 Network Analysis Methods

To address the questions asked above we seek a way to extract from a network's parameters an alternative representation of its function, a direct representation, one without interdependence between parameters. The representation should be exact, matching the network's function fully, but concise, not introducing redundancy. Only if all three properties are met can the algorithm truly reflect the full underlying computation of these networks and their complexity. Thus, in examining the different approaches to neural network analysis proposed in the literature we need to focus on whether they exhibit these properties of being exact, concise and direct. There are different ways to categorize approaches to network analysis, of these we choose to examine them by the form of their results or representation.

The *rule extraction* approaches [2; 20; 72; 86] extract a set of symbolic "rules" to describe network behavior. These algorithms can be divided into two broad categories by their treatment of the network, *decompositional* and *pedagogical*. The decompositional approaches try to find satisfiability expressions for each of the network units with respect to its inputs. Depending on the algorithm, this is achieved by first applying discretization, large scale pruning or placing structural limitations on the networks and then performing an exhaustive search on the inputs of all the units. To their credit, by performing an exhaustive search

on all the network's constituent units, the decompositional approaches offer a complete alternative representation to the network's function. Nonetheless, the rules extracted are not exact, as in most cases the network's function can only be approximated by rules. This representation is also not independent as the rules represent individual network units and as such maintain their distributed representation dependence.

The pedagogical and hybrid approaches to rule extraction do not decompose a network unit by unit, rather they construct rules by sampling the network's response to different parts of the input space, using varying degrees of network introspection to refine their regimes. Sampling makes these algorithms computationally feasible, and depending on the algorithm at times independent, but the algorithms fail to examine the full space of possibilities of network behaviors, so their rules are a greater approximation than the decompositional approaches.

Weight-state clustering [30], *contribution analysis* [70; 76], and *sensitivity analysis* [36; 37], which do use the network's parameters directly in their analysis, unlike rule extraction do not generate an alternative representation, rather they try to ascertain the regularity in the effect that different inputs have on a network's hidden and output units. That being the case, these methods do not explain global properties of the network, but are limited to specific inputs and like the pedagogical approaches, do not meet our criteria of being exact.

Hyperplane analysis [62; 73] is a technique by which the underlying hyperplanes of neural network's units are visualized. As such it is global and uses the network's parameters in its analysis. However, it does not remove interdependence, and does not scale since it is based on understanding the network interdependencies by direct visualization.

Network inversion [49; 51] is a technique where locations in the input are sought which generate a specific output. Maire's recent work is promising in that it approaches the problem by back-propagating polyhedra through the network and as such does generate an exact direct representation. However its main shortcoming is its lack of conciseness, each stage of inversion can generate an exponential number of sub-polyhedra.

Ideally we would like to construct an algorithm that extracts exact, direct and concise representations from any arbitrary neural network. This is not feasible, however. The main

impediment is that the functional form of the alternative representation is dependent on the type of activation function used. As such, for different activation functions we would need to use different types of representations, breaking our notion of a unified algorithm. Nevertheless, different monotonic activation functions do not fundamentally impact the questions we posed previously– the basic methods of network generalization remain the same. Therefore, the approach taken in this chapter is to construct an algorithm for a specific activation function (threshold) to address the questions as they pertain to this activation function, and then discuss what effects varying the activation function would have. The concept behind the algorithm is to find the limitations or constraints on how the network output can change under differing input conditions. Stated another way: How does the network architecture govern its decision regions?

2.3 Decision Regions

The output of a network, interpreted as a classifier, partitions the space of its inputs into separate decision regions. For each possible network output value, there exists a corresponding region in the input space such that all points in that region give the same network output. Hence the name *decision region*, since a region reflects the network's output or decision.

The decision regions encompass the full function that a network computes. They describe the complete mapping between the input and the output of the network. Unlike the original network, which does this mapping through the interdependent computation of many units, the decision regions map the input and output directly.

Decision regions can have different shapes and other geometric properties. These properties are directly related to the network architecture used. The Decision Intersection Boundary Algorithm (DIBA) is designed to extract decision regions from multi-layer perceptron networks, a feed-forward network with threshold activation functions. The decision regions for this type of network are polyhedra, the n-dimensional extension of polygons. The algorithm is based on a few principal observations about how multi-layer perceptrons compute.

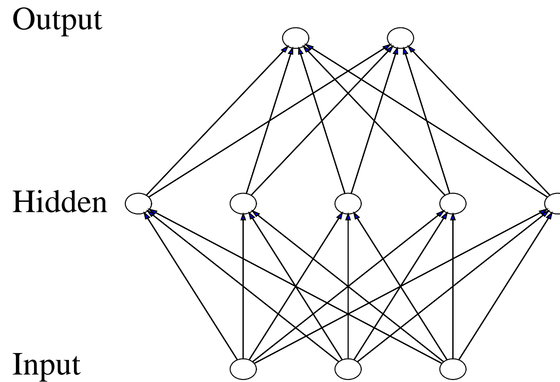


Figure 2.1: A 3-layer perceptron neural network.

Consider a three layer perceptron neural network, with full connectivity between each of the layers as in figure 2.1, where the output of each of the units is 1 if $\sum_i weight_i input_i > 0$ and 0 otherwise. The first observation is the independence of the output units. There is no information flow or connectivity between any of the output units. This is due to the strictly feed-forward nature of the network. Effectively, each output unit is computing its own value irrespective of the other output units. As such, for now, we can treat each output unit separately in our analysis of the network, generating separate decision regions for each output unit. Since this is a perceptron network with threshold activation functions, an output unit can have either a value of 0 or 1. Therefore we can pick to build decision regions corresponding to either an output value of 0 or 1.

An output unit value is dependent on the values at the hidden layer, which are dependent on the values at the input layer. Since the hidden layer units' output can only take on a value of 0 or 1 as well, their effect on the output unit's weighted sum is discrete. In fact, the output unit's weighted sum is just a partial sum of its weights. The full value of each weight is either included in the sum or completely left out.

Any location in input space where an output unit switches values is a decision boundary. Such a switch corresponds to a transition in the output unit's partial sum, either going above threshold when it was previously below or vice-versa. Since the partial sum is a function of which hidden units are active, this threshold transition must be coordinated with a state

change in one or more hidden units. This means that a decision boundary must correspond to a region in the input space where at least one hidden unit undergoes a state change.

Each hidden unit divides the input space into two regions, a region where its output is 0 and a region where its output is 1, in effect generating its own decision regions. In this context, each location in input space has associated with it a *hidden state*, a binary number corresponding to the output values of all the hidden units when given that input location. The hidden state corresponds directly to the output unit's partial sum and hence the output unit's value. This leads to the important observation that the only locations in input space where this hidden state can change are across the hidden units' own decision regions, or across the intersection of multiple decision regions. This implies that the basic building blocks of output unit decision regions are the decision boundaries between the intersections of hidden unit decision regions. As such the output unit decision regions are composed of parts of the hidden units' division of the input space.

In the multi-layer perceptron the hidden units divide the input space using hyperplanes. Therefore, the output unit decision regions are composed of high-dimensional faces generated by the intersection of hyperplanes, making them polyhedra. If we want to explicitly describe these faces, we need to specify their individual boundaries. As the intersection of hyperplanes, these boundaries are just lower dimensional faces. Assuming the space is bounded, we can repeat this process recursively, describing each face using its lower dimensional faces, until we reach zero dimensional faces or points. Consequentially, the output unit decision regions can be described by the vertices which delineate them.

2.4 The Decision Intersection Boundary Algorithm

The basic Decision Intersection Boundary Algorithm is designed to extract the polytopic decision regions of a single output of a three layer perceptron network. Later sections explain the simple modifications necessary to extend it to multiple hidden layers and multiple outputs and discuss the difference between threshold and sigmoid activation functions. The algorithm's inputs are the weights of the hidden layer and output unit, and boundary con-

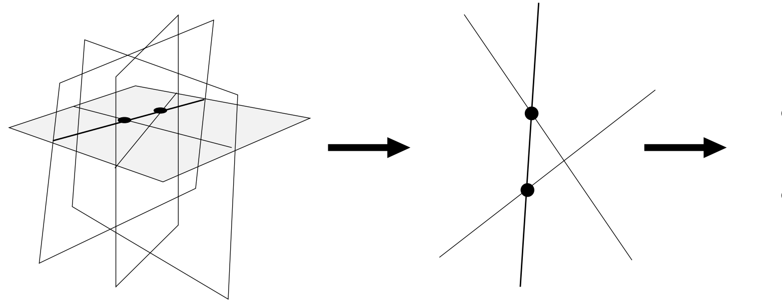


Figure 2.2: The generative portion of the DIBA algorithm recursively projects the hidden layer hyperplanes down till they are one-dimensional, in order to generate all the intersections of the hyperplanes. Here we see an illustration of two stages of projection from three dimensions to two and from two to one.

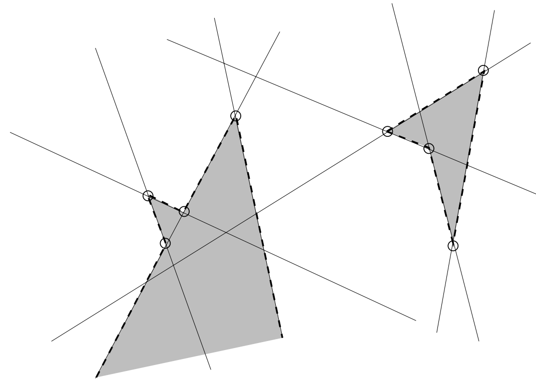


Figure 2.3: The boundary testing portion of the DIBA algorithm evaluates individual vertices and line segments, that were generated by the recursion, in order to test whether they form corners or edges of decision regions. Here we see the edges and corners of two decision regions the algorithm would recognize in an arrangement of hidden unit hyperplanes.

ditions on the input space. Using boundary conditions guarantees that the decision regions are compact, and can be described by vertices. Its output consists of the vertex pairs, or line segment edges, which describe the decision regions in input space.

As described in the previous section, decision region boundaries are at the intersections of the hidden unit hyperplanes. Thus the algorithm consists of two parts, a part which generates all the possible vertices and connecting line segments by finding the hyperplane intersections, and a part which evaluates whether these basic elements form the boundaries of decision regions. In figures 2.2 and 2.3 we see an illustration of these two parts of the algorithm.

2.4.1 Generative Recursion

The generative part of the algorithm is recursive. For each hidden unit hyperplane of dimension n, the algorithm projects onto it all the other hyperplanes. These projections are hyperplanes of dimension *d-1*. This procedure is performed recursively until one dimensional hyperplanes, or lines are reached– lines are the basic unit of boundary evaluation in this algorithm.

function extract(*hidden, out, borders*): Return Representation
Input Hidden Units' weights, Output Unit's weights, Borders
Output Representation containing vertices and line segments

```
if hyperplane_dimension(hidden) > 1
    for base_hyperplane in hidden
        for hyperplane in hidden
            if hyperplane ≠ base_hyperplane
                new_hyperplane :=
                    projection of hyperplane onto base_hyperplane
                if hyperplane is parallel to base_hyperplane then
                    store state of hyperplane with respect to base_hyperplane
                else
                    add new_hyperplane to new_hidden
                endif
            endif
        endfor
        call extract(new_hidden, out)
    endfor

textbfelse execute the traversing-the-line routine (appendix A)
```

2.4.2 Boundary Test

Along the line, the locations of output unit value changes are at the intersections with the remaining one-dimensional hidden unit hyperplanes. At each such location the algorithm needs to test whether the intersection, a vertex, forms a corner boundary– and if the intervening line segment forms a line boundary.

In order to understand the boundary test we need to examine the concept of a boundary in a d-dimensional input space. We start our examination by looking at one hidden unit. If a single hyperplane acts as an output unit border, it implies that at least for a portion of the hyperplane, in a neighborhood on one side of the hyperplane there is one output unit value and in the corresponding neighborhood on the other side of the hyperplane there is a different output unit value. For example, (see figure 2.4) if we take our input space to be two dimensional and use a line as our hyperplane, the line acts as a one-dimensional boundary if the output unit value on one side of the line is 1 and the output unit value is 0 on the other side of the line. One can think of the hyperplane as forming the boundary. Or inversely, one can think of the boundary taking on the form of a hyperplane in that particular region in space. That is, the location in input space where we have an output value transition can be described by a hyperplane. This is the functional view of the boundary. To functionally describe a boundary at that part of the space we need a hyperplane. Taking this view, a boundary is composed of various geometric entities which demarcate output value transitions in input space– and the corner test becomes: do we need a lower dimensional geometric entity to describe the boundary at the intersection of multiple hyperplanes?

Take two non-parallel hyperplanes which act as boundaries, around each of these hyperplanes the boundary takes the form of a d-1-dimensional manifold. If at their intersection both hyperplanes still act as boundaries, then the functional geometric form of the location with this output value transition across both hyperplanes is the d-2-dimensional hyperplane formed by the intersection of the two hyperplanes. One can say that the corner formed at the intersection is the description of the location of a complex boundary across multiple

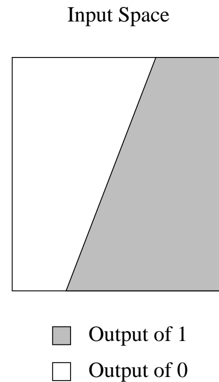


Figure 2.4: The decision region formed by the division of a two dimensional space by a hyperplane line. The line could be said to be partitioning the space. Or, in a complementary, functional fashion it could be said that the form of the boundary in the space is in the shape of a line.

hyperplanes.

To illustrate this, let us examine under what circumstances an intersection of lines (2-dimensional hyperplanes) forms a complex boundary. Two lines intersect at a vertex, this vertex could be a corner, a zero-dimensional boundary, or not. In figure 2.5 we see different boundary configurations at this vertex. Two of the configurations form corners, while the others do not. The two that do form corners have one element in common, both of the hyperplanes that make up the corner are still boundaries in their own right. That is, in the vicinity of the corner both hyperplanes have a part of them that engenders one output unit value on one side of the hyperplane, and another on the other side. As stated before, this corner marks a location in input space where a complex transition takes place, a transition across multiple hyperplanes. This corner test can be naturally generalized to higher dimensions. **In general, what we seek in an intersection that forms a boundary is that all hyperplanes making up the intersection have at least one face that is a boundary in its vicinity.**

How do we practically check for this intersection boundary condition? In a d-dimensional input space, an intersection of $n \leq d$ hyperplanes partitions the space into 2^n regions (see figure 2.6). If we consider our hidden units as these hyperplanes, then the space is partitioned into the 2^n possible hidden states. Each possible hidden state with respect to these hidden

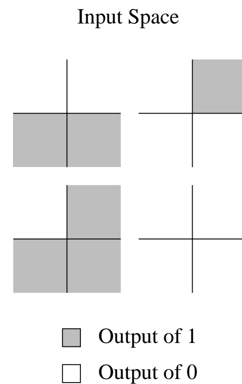


Figure 2.5: Some possible decision regions formed at the intersection of two lines. Corners are only formed if both lines are boundaries.

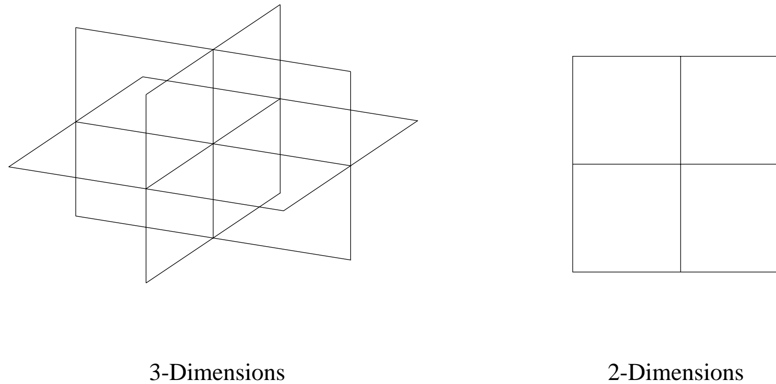


Figure 2.6: The partitioning of a 2 and 3 dimensional space around the intersection of hyperplanes.

units is represented in our input space around the intersection. A hidden unit hyperplane has a boundary face within the input space partitioning of the intersection, if within these 2^n hidden states there exist two hidden states that differ only by the bit corresponding to the hidden unit hyperplane being tested, such that one hidden state induces one value at the output unit and the other state induces another value. This basically says that at least in one location of the partitioning, if we cross the hyperplane we will get two different output unit values. Thus algorithmically the corner test is to go through all possible hidden states at the intersection and check that each hidden unit acts as a boundary.

In the 3-layer single output network case, where the output unit just computes a partial

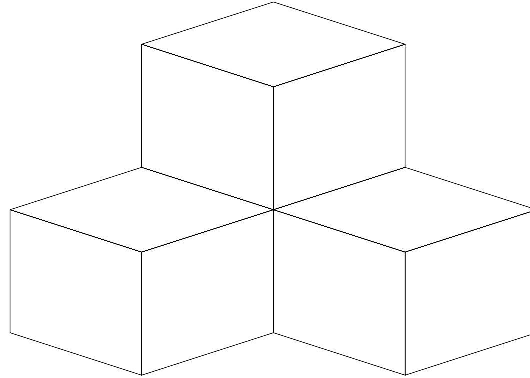


Figure 2.7: Unlike two dimensional spaces, a corner in a three dimensional space can be in the middle of multiple line segments.

sum of its weights, the test simplifies to finding whether the hyperplane corresponding to the smallest absolute valued weight has a boundary in the intersection. Of course, this is a necessary condition, but it is also sufficient, since if the hyperplane of the smallest absolute valued weight has a boundary then all other hyperplanes making up the intersection must also have boundaries. This is shown mathematically in appendix B.

The boundary test for line segments and corners is the same, since both lines and vertices are just intersections of hyperplanes. Lines are the intersection of *d-1* hyperplanes and vertices are the intersection of d hyperplanes.

2.4.3 Traversing the Line

When do we perform the boundary test? In two-dimensional input spaces, a corner always either starts or ends a line segment; a corner can not be in the middle of two line segments. In contrast, in higher dimensional spaces, corners can be in the middle of line segments. Figure 2.7 demonstrates how a three-dimensional corner can be in the middle of multiple line segments. However, even in high dimensional spaces a line segment can only start and terminate at the corners which delineate it. Therefore in traversing a line, we need to check corners at all hidden unit intersections, and check for line segments only between corners which start and end them.

For the intersection boundary test there are two kinds of hidden unit hyperplanes: hy-

perplanes that make up the intersection and hyperplanes that do not, but whose state *does* affect the output unit value. Before performing the boundary test on a line segment or vertex, we need to assess the contribution of these additional hyperplanes to the output unit partial sum in the intersection vicinity. It is possible to simply sample the hidden state of these hidden units in the vicinity of the intersection. But on the line there is a more economical solution. On the line we are traversing, the projection of these additional hyperplanes is a point. These hyperplanes each divide the line into a half where their hidden state value is 1 and a half where their hidden state value is 0. This gives a directionality to each hyperplane. That is, if we traverse the line from one end to the other, some of the hyperplanes will be in our *direction*, meaning that as we pass them their hidden value state will become 1, and the others will be in the other direction, their hidden value state changing to 0 as we pass them. See figure 2.8. We can use this property to incrementally quantify the contribution of these units to the hidden state/partial sum.

The bi-directionality suggests a two pass algorithm. Initially, we arbitrarily label the two ends of our line, *left* and *right*. The forward pass starts with the leftmost hyperplane and scans right, hyperplane by hyperplane. We are interested in finding the partial sum contribution for each line segment between hyperplanes. So, as each hyperplane is encountered it is checked for right directionality, and its weight is tallied to a running sum of the weights. This sum is assigned to the current line segment, accounting for the contribution of all the right directed hyperplanes on this line segment. The backwards pass is identical, except it starts from the rightmost hyperplane and scans left, adjusting its running sum with left directed hyperplanes, and adding its sum to the values already generated in the forward pass. Thus, completion of both passes calculates the partial sum contribution of all left and right directed hyperplanes for all the line segments on the line. The actual boundary tests can be conducted during the backwards pass, since the full contribution to the partial sum has been tabulated at that stage. Appendix A contains a pseudo code description of the "Traversing the Line" portion of the algorithm.

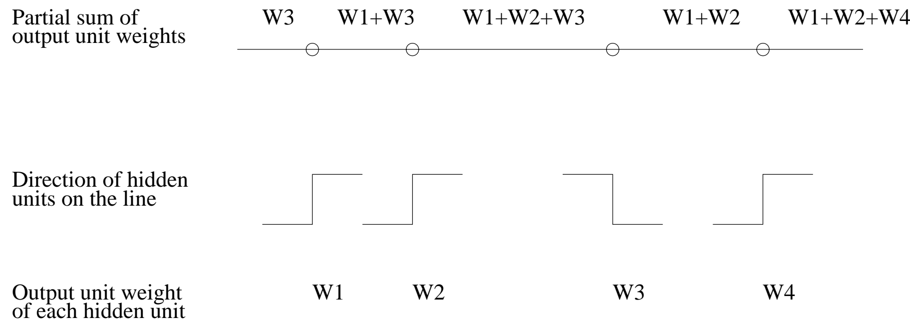


Figure 2.8: The bi-directionality of hyperplane intersections on a line suggests a two pass algorithm to calculate the partial sums of the output unit weights.

2.5 Examples

The following are examples of applying the Decision Intersection Boundary Algorithm on trained three layer perceptron networks. The training consisted of a mixture of momentum back-propagation and weight mutation hill-climbing[1]. However the units were always treated as threshold for the purposes of the DIBA algorithm. The examples are used to demonstrate the kind of decision region descriptions that can be extracted using the DIBA algorithm and how they can be interpreted. In parallel, by examining the representations of multiple networks we address the questions of scope: What is the form of network artifacts? And what can we learn about how networks generalize from their representation?

2.5.1 Sphere Networks

Two-Dimensions: The Circle

Using back-propagation an 80 hidden unit sigmoidal neural network was trained to classify 300 points inside and outside of a circle of radius 1 around the origin. Treating the activation functions as threshold, the DIBA algorithm was then applied to the weights of the network. In figure 2.9 we see the decision region extracted at the origin, and the points corresponding to the training sample. The decision region allows us to directly view what the network does– exactly where it succeeds and where it fails. The representation is concise, since all

[1]We do not go into the specifics of these generic training methods, as the focus of the paper is on the trained networks, not how they were trained. For further reading, see [8] on these and other training methods.

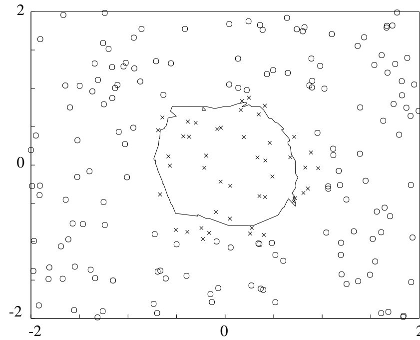


Figure 2.9: A decision boundary of a network which classifies points inside and outside of a circle.

the vertices used are necessary to completely describe the decision region. With this exact representation we can make out the nuances of the network's function, for example, note the small protrusion on the bottom right part of the decision region that covers two extremal points.

The DIBA algorithm also allows us to examine other aspects of this network. By zooming out from the area in the immediate vicinity of the origin we can see the network's performance, or generalization ability, in areas of the input space that it was not explicitly trained for. In figure 2.10 we recognize large artifactual decision regions at a distance of at least 50 from the decision region at the origin. Thus, in this area of the input space where there was no actual training data, a few decision regions formed as artifacts of the network weights that were learned.

Three-Dimensions: The Sphere

A 100 hidden unit network was trained to differentiate between points inside and outside of a sphere centered at the origin. In figure 2.11 we see the rather successful decision region encapsulating the network's internal representation of the sphere from different angles. Figure 2.12 illustrates the same phenomena of additional artifact decision regions we saw in the circle for the sphere– the miniature sphere appears amid a backdrop of a large cliff face like decision region. Note that the artifacts are more complex in the three-dimensional case. As the higher dimensionality of the input space implies the existence of exponentially

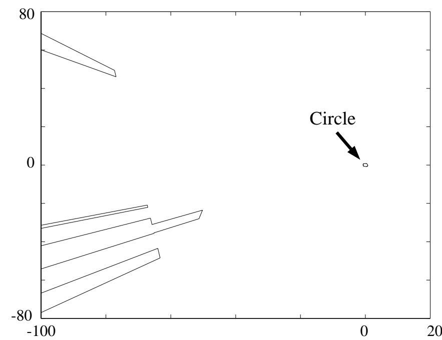


Figure 2.10: A zoom out of the network illustrated in figure 2.9 illustrates the existence of artifactual decision regions.

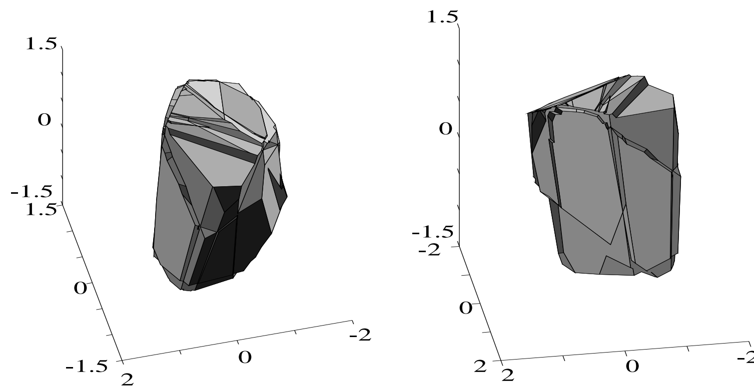


Figure 2.11: A decision boundary of a network which classifies points inside and outside of a sphere.

more hyperplane intersections (vertices) than in the two-dimensional case and with that potentially more artifactual decision boundary corners.

Four-Dimensions: The Hyper-Sphere

Another 100 hidden unit network was trained to recognize points inside and outside of a 4-dimensional hyper-sphere centered around the origin. Due to human limitations it is difficult for most people to visualize objects in more than three dimensions (even three can be a challenge at times.) One way to gather information from our high dimensional polytopic decision regions is to describe them in terms of *rules*. That is, we bound each polytope inside of a hyper-rectangle, and examine the rectangles' coordinates. At this first approximation

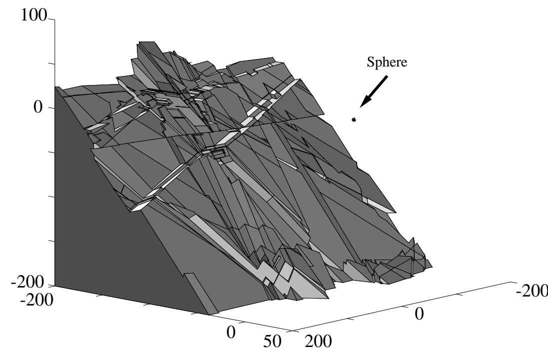


Figure 2.12: The artifacts around the sphere decision boundary.

we can elucidate how many decision regions there are, their location in input space and a coarse approximation of their volume. The rectangles can later be incrementally refined to enclose parts of polytopes, thereby giving higher resolution rules, refining our perception of their structure and volume. In this case the hyper-rectangle which covers the polytopes representing the hyper-sphere has the following minimum and maximum coordinates:

Min:	(-4.45	-5.79	-3.90	-7.93)
Max:	(6.47	6.01	7.39	6.07)

Another way to examine the high dimensional space is to examine projections on to lower dimensional spaces. In figure 2.13 we see projections of the four dimensional polytope with the fourth component set to zero and one respectively. Needless to say the four dimensional hyper-sphere looks less and less like a sphere. As before, increasing the number of input dimensions increases the potential for more artifactual boundaries. But this is also coupled with the curse of dimensionality [8], which says that as we increase the input dimension while keeping the number of training samples constant, our problem (the decision region we are trying to learn) becomes exponentially less specified. Thus, artifacts appear not only at a distance from out training samples, but in higher-dimensional spaces, might appear on our actual decision regions in the form of unwanted complexity.

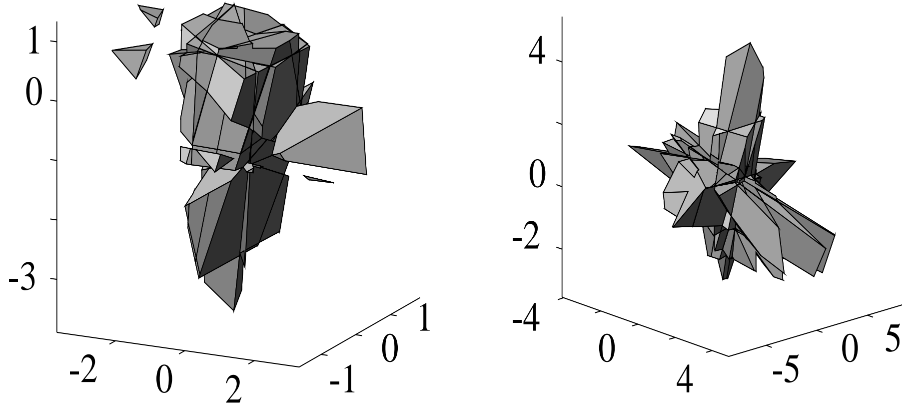


Figure 2.13: Projections of the four dimensional hyper-sphere polytope across $x_4 = 0$ and $x_4 = 1$.

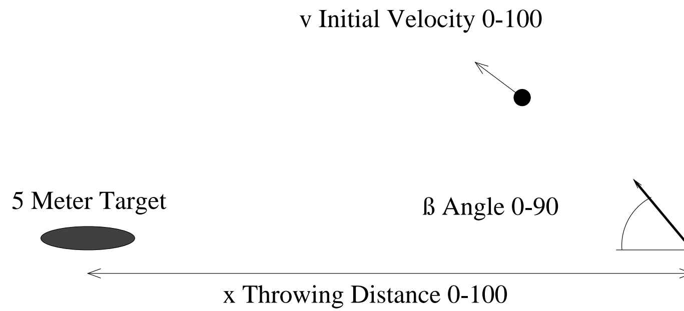


Figure 2.14: The network is supposed to predict whether the ball will hit the target given throwing angle β, initial velocity v and target distance x.

2.5.2 Ball Throwing Network

A 15 hidden unit network was trained to predict whether a thrown ball will hit a target. As input, it received the throwing angle, initial velocity and the target distance (see figure 2.14.) After training it achieved an 87% success rate on the training data.

This is a simple non-linear system which when solved analytically gives us the following relationship:

$$\left| x - \frac{v^2 sin 2\beta}{2g} \right| < 2.5$$

Figure 2.16 contrasts the decision region generated from a 15 hidden unit neural network with the analytical solution. It is apparent that the network managed to encapsulate the gist of the model, its decision region approximates the analytically derived decision region fairly well. The decision region possesses properties which suggest good network generalization, independently of knowing the actual solution. The decision region clearly defines a specific sub-manifold of the space. It consists of 4 main sub-decision regions, that are each highly convex but distinct from each other, suggesting that those locations in input space were clearly individually defined in the training set. In addition, the region of input space not in the decision region (the "misses the target" area) is also clearly defined and separated, consisting of a few large convex regions.

2.5.3 Predicting the S&P 500 Movement

A neural network with 40 hidden units was trained to predict the average direction of movement (up or down) for the following month's Standard and Poor's 500 Index. It was trained on monthly macro economic data from 1953 to 1994 from Standard & Poor's DRI BASIC Economics Database. The network's inputs were the percentage change in the S&P 500 from the past month, the differential between the 10 year and 3 month Treasury Bill interest rates, and the corporate bond differential for AAA and BAA rates, indicators used in non-linear economic forecasting [15]. After training, the network achieved a better than 80% success rate on the in sample data. Figure 2.15 shows the network's decision regions. from it we can surmise that the network would probably not generalize very well to out of sample data. We can see that rather than learning some underlying regularity, the network tended to highly partition the space to try and enclose all the sporadic data points.

We can use the rule method outlined before to quantify some of these partitioning effects. In the region of input space where the training data resides the network has 28 different decision regions. Of these all but five have a bounding rectangle with a volume of less than one. One decision region has a bounding rectangle which encompasses the whole input space. We can refine the rule for this large decision region by slicing the decision region using another hyperplane and examining the bounding rectangles for the resultant sub-decision

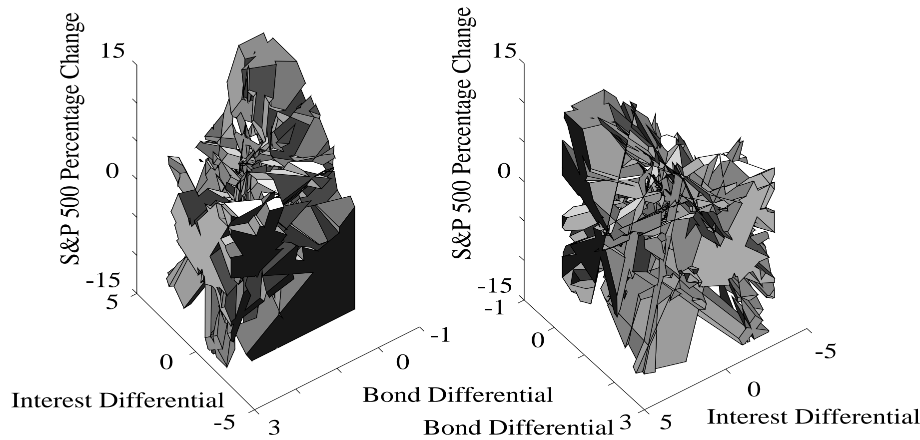


Figure 2.15: The decision regions of the S&P 500 prediction network.

regions. If we simultaneously slice this polytope using three hyperplanes, each bisecting the input space across a different dimension, then if the polytope were completely convex, we would expect, at most, to get eight sub-polytopes. However for this decision region, this refinement procedure generated 23 separate sub polytopes, implying that the polytope has concavity and probably has some form of irregular branching structure needed to partition the space. A simple analogy would be to contrast an orange with a comb. Both are solid objects. No matter how we slice the orange we will always be left with two pieces. However if we slice the comb across its teeth, it will decompose into many pieces.

2.5.4 Vowel Recognition

Two neural networks were trained on the vowel data available at the CMU neural network benchmark repository [22]. This data consists of 10 log area parameters of the reflection coefficients of 11 different steady state vowel sounds. Our interest in this example was to gauge the effect of using different dimensioned input spaces. The reflection coefficients are particularly suited for this test [52] because of their mathematical properties: they are orthogonal, the coefficients do not change when larger sets are generated, and using their log area parameters confers a greater spectral sensitivity. Both networks were trained to recognize the vowel sound in the word *had* within the background of the other 10 vowel

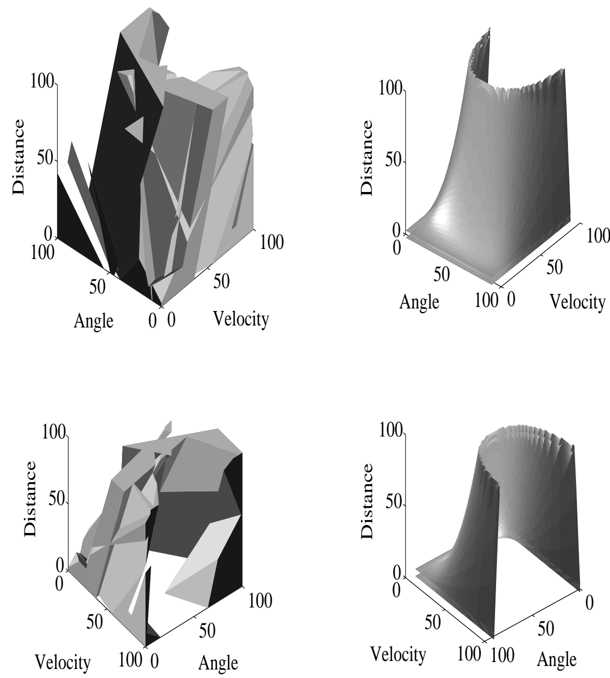


Figure 2.16: The decision region of the ball throwing neural network (left) contrasted with the decision region of the actual analytical solution (right). Two different perspectives are shown.

sounds.

The first network received as input the first two coefficients. After training, it achieved a better than 86% success rate on the training data. Its decision regions and the training data are shown in figure 2.17. In the input region of $[-3.2, -2.3] \times [0.7, 1.2]$ we see a relatively high degree of partitioning, in that many decision regions are being used in order to secure a perfect classification, implying problematic generalization in that region.

The second network received the first four coefficients as inputs, and enjoyed a success rate of over 91% on the training data. It also achieved perfect classification within the input region described. However, it appears to have done so with less partitioning of the space. We can see this using the rule refinement procedure described in the previous example. If we extract the decision regions in this part of input space we get only one rectangle which spans it completely. Conducting the same kind of concavity test we previously used, that is slicing the space using four hyperplanes, each bisecting an input dimension of the region, we get only 10 sub polytopes suggesting a small degree of concavity (less than the 2^4 for a perfectly convex shape). In addition these sub-decision regions are mostly delimited in the third and fourth dimensions with the first two dimensions left to span the whole area of the sub-space. Therefore it appears that the network makes use of these added dimensions to form a more regular decision region in that difficult region of the input space. This must be qualified, since by the curse of dimensionality the increase in the dimensionality made the problem less specified, and as such it became easier to enclose the data points within one, more convex decision region. But the fact that the decision region makes exclusive use of the added dimensions to discriminate, significantly strengthens the claim of potentially better generalization.

2.6 Discussion

2.6.1 Algorithm and Network Complexity

The Decision Intersection Boundary Algorithm's complexity stems from its transversal of the hyperplane arrangement in the first layer of hidden units. As such, that part of its complexity is equivalent to similar algorithms, such as arrangement construction [23], which

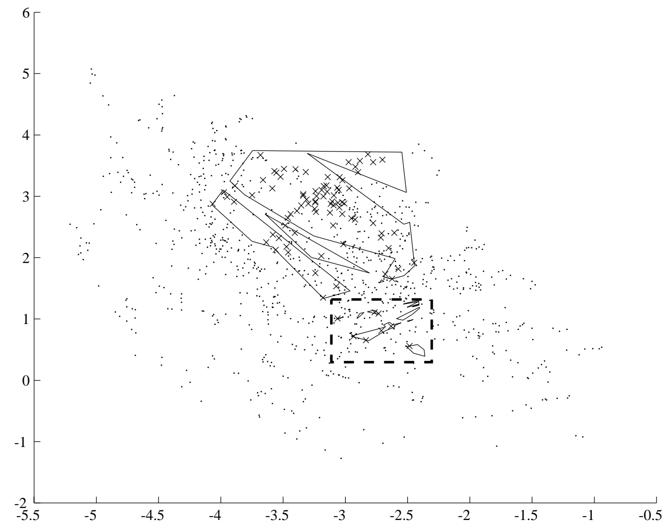


Figure 2.17: The decision regions of the two input vowel recognition network with the training data in the background. The X's are positive test cases.

are $O(n^d)$, where n is the number of hyperplanes and d is the input dimension. Another aspect of complexity stems from the corner test, which is $O(2^d)$. Even the partial sum case is of similar complexity, since the question of asking whether the lowest magnitude weight acts as a border is equivalent to the *knapsack* problem [71] which is NP-complete.

Do we really need to examine every vertex though? Perhaps the network can only use a small number of decision regions, or it is limited with respect to the complexity of the decision regions. In this section we prove that this is not the case by hand construct a network with $\Omega\left(\left(\frac{n}{d}\right)^d\right)$ different decision regions, where each decision region has 2^d vertices, concluding that networks are capable of having exponential complexity.

We begin with a one dimensional construction of a three layer, single output perceptron network. Start with k hidden units with the same directionality, and assign to them alternating $+1$ and -1 weights. In figure 2.18a we see such a construction with $k = 4$ hyperplanes. If we set the output unit threshold to 0.5, we see that the hyperplanes partition the input space into three decision regions, a decision region for each line segment with a weight sum of zero.

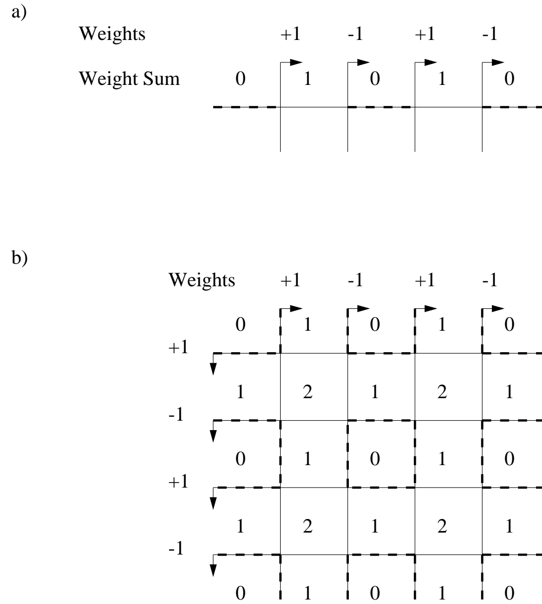


Figure 2.18: A hand constructed neural network which demonstrates the potential number of decision regions a network can have.

Let us extend this construction to a two dimensional input space. First we extend the one-dimensional hyperplanes, points, to two-dimensional hyperplanes (lines). We do this by making them parallel in the added dimension. Next we add an additional k hyperplanes that are orthogonal to the original hyperplanes, and again assign to them alternating -1 and $+1$ weights. Figure 2.18b illustrates this. We see that the additional hyperplanes continue each zero term in a checkerboard pattern in the added dimension, but above zero terms remain above zero in the added dimension. Thus, by adding a dimension, each lower dimensional decision region multiplies to become either $(k+1)/2$ or $k/2+1$ different decision regions, depending on whether k is odd or even.

This construction can be extended to any number of dimensions, so by induction we can see that this construction has $\Omega\left(\left(\frac{n}{d}\right)^d\right)$ decision regions. Since each decision region is a hypercube in d dimensions, it has 2^d vertices. Another point to note is that each hyperplane contributes a face in $\Omega\left(\left(\frac{n}{d}\right)^{d-1}\right)$ decision regions. This example uses parallel lines, the complexity is obviously higher with non-parallel lines in which all hyperplanes intersect with each other.

The result of this construction extends Golea's proof [29] of the NP-hardness of discrete network analysis to the continuous input case, by showing that it is necessary for the algorithm to be exponential as the complexity of the network may also be exponential in its decision regions structure. This complexity result addresses our first theoretical question as it applies not only to this algorithm, but to any algorithm which on some level tries to describe a neural network by enumerating its functions, listing its decision regions (e.g., rule extraction). Any algorithm that deals with network function on the level of decision regions will need space exponential in the input dimension to fully describe an arbitrary network.

2.6.2 Extensions to the Algorithm

The formulation of the DIBA algorithm given previously was for a three layer, single output network. Now we address on of the theoretical questions posed earlier: How does the complexity of the network and algorithm change when we allow more hidden layers and multiple outputs?

Additional Hidden Layers

Fundamentally, the addition of more layers to a perceptron neural network does not change the underlying possible locations and shapes of the decision regions. Consider adding an additional layer to our previous three-layer network. We already know that the units in the first hidden layer divided the input space into half-spaces, the units in the second hidden layer (previously the output units) form decision regions composed of the intersections of the divisions in the first layer, so what does the next layer do? Like the units in the second layer, the output values in the third hidden layer can only change across a boundary in the previous layer (another partial sum). And since the values in the second layer can only change across the boundaries of the first hidden layer's hyperplanes, then the boundaries of the third hidden layer are still only composed of the intersections of the first layer's hyperplanes. This argument holds for any additional layers. So in essence the first layer fundamentally defines what possible decision regions the network can express.

In terms of the algorithm itself, the modification is straight forward. The potential locations of the polytope vertices are still the same, we just need to adjust the corner test.

Rather than doing the partial sum test for the smallest absolute valued weight, we perform the general corner test. We check for a transition across all the intersection hyperplanes at a vertex, but with respect to an output unit at a higher level. Specifically, we form all the possible hidden states at the vertex, feed them to the rest of the network, and see if they qualify as a corner boundary with respect to the output unit. That is, each hyperplane acts as a boundary at least once in the vicinity of the vertex. Thus, the algorithm complexity does not change with the introduction of multiple hidden layers.

Multiple Output Units

Multiple output units can be handled in much the same way as additional hidden layers, using the generalized corner test. Instead of checking for an output transition with respect to one output unit (either on or off), we can check for an output transition with respect to a combination of output units. For example we might be interested in extracting the decision regions described by having exactly six output units on. Then in order to check if a vertex forms part of such a decision region we check the value of all the output units with respect to the possible hidden states at the vertex. If there is a transition across all hyperplanes forming the intersection, that is, in the vicinity of the intersection across each hyperplane there is a location where the number of on output units is six on one side and different from six on the other side, then it is a corner of the decision region. This can be applied to any output interpretation regiment that is applied to the output units. As such, the algorithm complexity stays the same for different output interpretations.

2.6.3 Rule Extraction and Network Validation

Polytopic decision regions offer us a direct representation of the underlying neural network. However, high dimensional polytopes are not immediately fathomable to most people either. The interdependence in the polytopic case is limited to understanding the linear relationships governed by the vertices which delineate the faces.

As proposed earlier, one way to generate more comprehensible descriptions is to generate independent rules in the form of minimum bounding hyper-rectangles (MBR). That is, for each polytope we find the minimum hyper-rectangle which completely encloses it. This is a

trivial operation, and it gives us a zeroth order approximation for the location and size of the network's decision regions.

We can improve this rule approximation to an arbitrary degree by dividing the polytopic decision regions into sub-polytopes. In the previous section we saw one approach to this, which was to systematically bisect the polytopes across its dimensions. This incrementally refines each rule into sub-rules each time it is applied.

It is feasible to imagine a more intelligent method to divide the polytopes. For example, we could only divide those polytopes for which the MBR is a bad approximation. It would seem that our geometric analogy would also give us the tools to exactly calculate the efficacy of the approximation, since all we would need to do is compare the volume of the hyper-rectangle with the volume of polytope it was enclosing to get an exact error measure. However, volume computation of n-dimensional polytopes is $\#P$-hard in the exact case (worse than NP-hard) and of high complexity for approximate cases [42]. This is a general property of the comparison of different models with perceptron neural networks, not just symbolic approximations– to get an exact error measure we have to compute the volume of the polytopic decision regions. This addresses another of the theoretical questions: The cost of validating the accuracy of alternative representations of network function is computationally hard.

2.6.4 Sigmoidal Activation Functions

It is not possible to model the underlying decision regions of a sigmoidal neural network using only vertices and lines, since the sigmoid is a smooth transition. However we can get a good idea of the implications of this form of non-linearity by modeling the units with piecewise linear units (see figure 2.19.) Unlike the threshold activation function unit hyperplanes, where an output transition is completely localized, the transitions in piecewise or sigmoidal activation function units are gradual and take place over the *width* of the hyperplane. In figure 2.20 we see the form of a typical intersection between two border piecewise linear hyperplanes. Notice how each hyperplane resides within the width of their linear region. The actual location of the boundary falls somewhere within that region, wherever the output

unit exactly passes its decision threshold. When multiple linear regions overlap they form a composite linear region. As the figure illustrates, the boundary in this region has the effect of smoothing out the edge or corner by defining a sub-face between the faces of the intersecting hyperplanes. A sigmoidal unit would generate a smoother transition.

The intersection of multiple piecewise linear hyperplanes generates a potentially exponential number of different linear regions in its vicinity. Any of these regions could potentially house a border face. The test for whether a border passes through such a region is trivial (check the smallest and largest corners.) However, finding the exact face of the border is also a hard problem [42].

The addition of more hidden layers still does not substantially change the possible underlying decision regions. The output of any higher level unit is dependent on the values of the lower level units. This means that a transition in the value of a higher level unit must accompany a transition in a lower level unit. Therefore, the decision regions can only be within the width of the first layer's hyperplanes, where all basic transitions take place.

To summarize, the effect of using sigmoidal activation functions on network computation is to change the shape of the boundary at the hyperplane intersections, specifically it has a smoothing effect. The fundamental locations of the decision region boundaries is still governed the hyperplanes of the first hidden layer. However, we can get additional complexity within the widths of the intersections. The computational cost of extracting this additional complexity by approximating it with piece-wise linear activation functions is potentially high.

2.6.5 Generalization and Learning (Proximity and Face Sharing)

Generalization is the ability of the network to correctly classify points in the input space that it was not explicitly trained for. In a semi-parametric model like a neural network, generalization is the ability to describe the correct output for groups of points without explicitly accounting for each point individually. Thus, the model must employ some underlying mechanisms to classify a large number of points from a smaller set of parameters.

In our feed-forward neural network model we can characterize the possible forms of

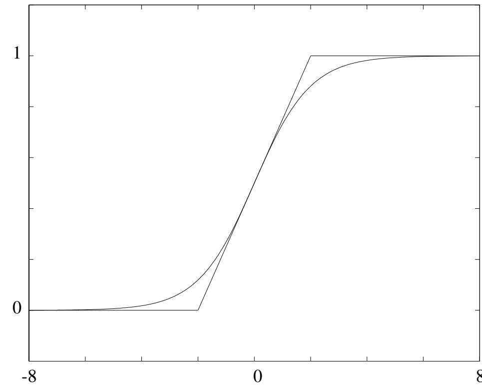


Figure 2.19: A sigmoidal activation function and its piecewise linear counterpart.

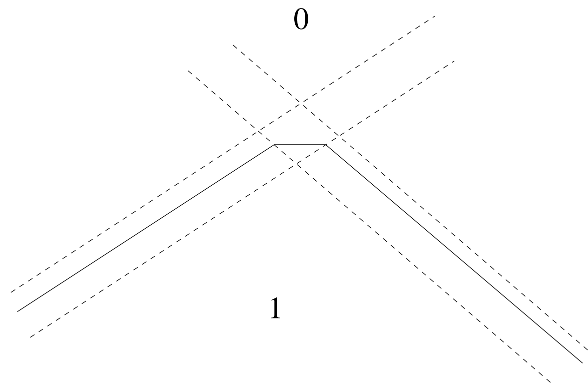


Figure 2.20: The intersection of two piecewise linear border hyperplanes forms an intermediate border face.

generalization into two mechanisms. The first is by *proximity*: nearby points in the same decision region are classified the same way. The second is by *face sharing*: the same hyperplane is used as a face in either multiple decision regions or multiple times in a decision region. An analogy for this type of generalization would be Manhattan streets, where each street forms the boundary of many different blocks.

Changing the network architecture does not radically affect the kind of generalization possible. Instead of hyperplanes another shape could be used or the non-linear activation functions could be modified in a locally continuous fashion. Fundamentally we would still have just two kinds of generalization, proximity and face sharing, in these types of feed-forward network architectures.

Given these two mechanisms, how well do learning algorithms exploit them? Proximity mandates the ability to enclose regions in space with similar outputs. It is intuitive that learning algorithms which pull borders (Hyperplanes in this case) towards similar output points and push borders away from different output points, should be geared to do some form of proximity generalization by forming decision regions around similar points. However, face sharing generalization is more combinatorial in nature, and might not be as amenable to continuous deformation of parameters as found in many algorithms.

To illustrate this point a group of 8 hidden unit neural networks were trained on the decision regions illustrated in figure 2.18b, a problem requiring face sharing to solve. One thousand data points were taken as a training sample. Two hundred different networks were used. Each was initialized with random weights in the range of -2 to 2, and trained for 300 online epochs of back-propagation with momentum. Of the 200 networks, none managed to learn the desired 9 decision regions. The network with the best performance generated only six decision regions (figure 2.21a). In examining its output unit's weights, we saw that only one weight changed sign, the weight initially closest to zero, indicating a predisposition to this configuration in the initial conditions with respect to the learning algorithm. Or stated more directly, the learning algorithm did not cause sufficient combinatorial modifications to facilitate face sharing. Figures 2.21b and c show the decision regions for some of the other, more successful networks.

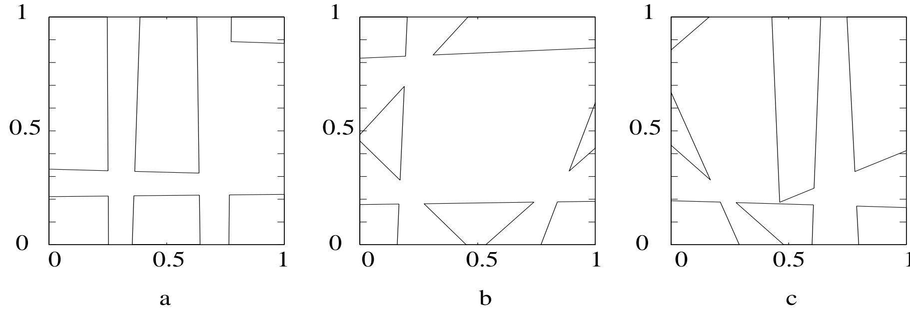


Figure 2.21: The decision regions of back-propagation networks trained on the decision regions of figure 2.18b.

This issue brings up some potential research questions. How do direct network construction algorithms contrast or coexist with training algorithms with respect to the decision regions they generate? Can we employ geometric regularization in learning algorithms to generate decision regions with specific properties?

2.7 Conclusion

We started the chapter by asking what factors affect the complexity of neural network representation extraction in general. We further asked, what can be learned about a network's success from its representation, especially in terms of generalization and artifacts? The tool we developed for the task is the Decision Intersection Boundary Algorithm. An algorithm that can be used to extract exact, concise and direct representations of the decision regions of threshold multi-layered perceptron networks.

The scope questions were addressed by analyzing multiple example networks to see where they introduce noise, when they generalize, and what forms their computation can take on. Using the examples of sphere networks at different dimensionality we explored the relationship between the input dimension, the location of the training data and the appearance of artifacts, clearly demonstrating that artifacts are not only not uncommon but become quickly prevalent in higher dimensional input spaces. We then explored generalization for three different networks by examining the properties of their decision regions, looking at: convexity, concavity, the quantity of decision regions, their location, and their orientation.

In general, how the decision regions partition the input space was seen as an indicator of generalization. In addition, for higher-dimensional spaces where the decision regions can not be directly visualized we explained how using hyper-rectangles and slicing the decision regions can be analyzed for these different properties at any desired resolution.

We started addressing the theory questions by analyzing the complexity of the algorithm, proving that even though the algorithm's complexity is exponential, it is unavoidable, since networks are capable of generating an exponential number of decision regions, where each decision region's complexity is also exponential. By explaining how the algorithm can be extended to additional hidden layers, and multiple output units, we showed that these modifications do not fundamentally impact the kind of processing the network is capable of, or their complexity, as the arrangement of hyperplanes in the first layer of hidden units fundamentally defines what possible decision regions the network can express.

Next, we discussed the computational cost of finding the exact error between a network and any approximate representation, showing the cost to be exponential due to the complexity of volume calculations in high-dimensional spaces. Then, we explored the ramifications of switching the activation function to a sigmoidal one, concluding that even though it may induce additional complexity to the network's decision regions, this complexity is highly localized to the widths of the boundaries of the first layer of hidden units, otherwise their decision regions behave the same as for threshold units.

We concluded by asking how well can learning algorithms use the two forms of generalization, proximity and face sharing. Is it possible for these networks to exploit all their potential complexity? Even though back-prop did not show promising results for face sharing, this remains an open question and a direction for further exploration of future learning and network construction algorithms.

Beyond our questions, the DIBA algorithm can also be used as an applied tool. As an exact, direct and concise representation extraction algorithm, the DIBA algorithm can easily be used to analyze reasonably sized threshold networks,[2] where knowing the exact representation is essential to validate them for real-world deployment. Or, using the concepts

[2] A five dimensional, 100 hidden unit network takes less than 10 seconds to analyze on a Pentium II.

and methods presented in this chapter, it can be used to shed light on how any such network generalizes. DIBA can also be used to study learning, in the simple case to visualize decision regions changing during learning, or as we have done in other experiments, to explore how well learning algorithms form specific decision regions. For examples, please look at http://www.demo.cs.brandeis.edu/pr/DIBA.

Appendix A: Pseudo Code of Traversing the Line

In this appendix a pseudo code description of the Traversing the Line portion of the Decision Intersection Boundary Algorithm (DIBA) is outlined. It is the termination step of the recursion described in the Generative Recursion portion of the algorithm.
It calculates the contribution of the different hyperplanes to each potential line segment, and then finds the corners and actual line segments.

else { *hidden* hyperplane dimension ≤ 1 }
    sort the *hidden* by their position on the line

Forward pass.

    running_sum := 0
    clear array *segments*
    current_segment := 1
    for *hyperplane* = leftmost hyperplane to rightmost hyperplane
        if *hyperplane* not a border **then**
            if *hyperplane* is forward directed **then**
                Add weight *out(hyperplane)* to *running_sum*
            endif
            segment(current_segment) := *running_sum*
            current_segment++
        endif

endfor

Backward pass.

```
running_sum := 0
in_line := false
current segment--
for hyperplane = rightmost hyperplane to leftmost hyperplane
    if beginning of border delimited region and check_line(current_segment) then
        in_line := true
        Add the node to the representation
    elseif hyperplane not a border then
        current_segment--
        if hyperplane is backward directed then
            Add weight out(hyperplane) to running_sum
        endif
        segment(current_segment) := segment(current_segment) + running_sum
        if in the border delimited region then
            new_in_line := check_line(current_segment)
            if in_line and new_in_line then
                if check_corner(current_segment, out(hyperplane)) then
                    Add the vertex to the representation
                    Connect current vertex to last corner
                endif
            elseif in_line and not new_in_line then
                Add the vertex to the representation
                Connect current vertex to last corner
            elseif not in_line and new_in_line then
                Add the vertex to the representation
```

endif

endif (in border region)

endif (not border)

endfor

if *in_line* **then**

Add the vertex to the representation

Connect current vertex to last corner

endif

endif

Appendix B: Proof of partial sum corner test

For a 3-layer network with one output unit (partial sum case) it is sufficient to test whether the hidden unit with smallest absolute valued weight has a boundary in the intersection. If it does, then all the other hyperplanes which make up the intersection also have boundaries.

Let $W = \{w_1 \dots w_n\}$ be the set of partial weights corresponding to the hyperplanes making up the intersection. Define $S \subseteq W$, as a hidden state. Let T be the threshold, such that if $\sum_{w \in S} w \geq T$ then the output value is 1, otherwise the output value is 0.

Assume that there exists a w_m, such that for all $w \in W$, $|w_m| \leq |w|$, and there exists a hidden state, S, which does not contain w_m, such that if $w_m > 0$ then $\sum_{w \in S} w < T$ and $w_m + \sum_{w \in S} w \geq T$, or if $w_m < 0$ then $\sum_{w \in S} w \geq T$ and $w_m + \sum_{w \in S} w < T$.

If $\alpha \in W$, and α is not w_m, we need to demonstrate that there exists a hidden state such that α acts as a boundary.

- If $w_m > 0$, $\alpha > 0$ and $\alpha \notin S$, then $\sum_{w \in S} w < T$ and $\sum_{w \in S} w + \alpha \geq T$.
- If $w_m > 0$, $\alpha > 0$ and $w \in S$, then $\sum_{w \in S} w + w_m \geq T$ and $\sum_{w \in S} w + w_m - \alpha < T$.
- If $w_m > 0$, $\alpha < 0$ and $\alpha \notin S$, then $\sum_{w \in S} w + w_m \geq T$ and $\sum_{w \in S} w + w_m + \alpha < T$.
- If $w_m > 0$, $\alpha < 0$ and $\alpha \in S$, then $\sum_{w \in S} w < T$ and $\sum_{w \in S} w - \alpha \geq T$.

- If $w_m < 0$, $\alpha > 0$ and $\alpha \notin S$, then $\sum_{w \in S} w + w_m < T$ and $\sum_{w \in S} w + w_m + \alpha \geq T$.
- If $w_m < 0$, $\alpha > 0$ and $\alpha \in S$, then $\sum_{w \in S} w \geq T$ and $\sum_{w \in S} w - \alpha < T$.
- If $w_m < 0$, $\alpha < 0$ and $\alpha \notin S$, then $\sum_{w \in S} w \geq T$ and $\sum_{w \in S} w + \alpha < T$.
- If $w_m < 0$, $\alpha < 0$ and $\alpha \in S$, then $\sum_{w \in S} w + w_m < T$ and $\sum_{w \in S} w + w_m - \alpha \geq T$.

Therefore all hyperplanes have a boundary in the vicinity of the intersection.

Chapter 3

Decision Region Connectivity Analysis: A method to analyze high-dimensional classifiers

The DIBA algorithm described in the previous chapter allowed us to exactly analyze the decision regions of multi-layer perceptron networks. By doing so, we came to realize that the classification strategy (and the potential to generalize) of a network is to a large degree guided by how it uses decision regions to partition its training data. Different partitioning strategies may be appropriate for different cases, but the fundamental choices given to a model are issues such as the number of decision regions and their convex/concave structure. Another aspect of neural computation explored by applying the DIBA algorithm were artifacts, model complexity that is unrelated to the training task. In exploring this issue we realized that it is quite insidious, as higher-dimensional networks are prone to exponentially more artifacts. These two issues combined raised the question, is there a computationally efficient way to analyze a model (neural networks or otherwise) that gets out all its important decision region structure choices but filters out the irrelevant artifactual complexity?

The answer is yes, and in this chapter we present a method to extract decision-region structural information from any classification model that uses decision regions to generalize (e.g. neural nets, SVMs, etc) while filtering out artifacts that are not related to the training task. The method's complexity is independent of the dimensionality of the input data or model, making it computationally feasible for the analysis of even very high-dimensional

models, something which can not be said for any other method that I am aware of. The qualitative information extracted by the method can be directly used to analyze the classification strategies employed by a model, and also to compare strategies across different model types.

This material has been presented at the Montreal Workshop on Selecting and Combining Models, and a conference version of this material has appeared in the proceedings of the IJCNN 2000, where it received the best of session distinction. In addition, a journal paper has been submitted to a special issue of *Machine Learning*.

3.1 Introduction

It is typically difficult to understand what a high-dimensional classifier is doing. The most common form of analysis usually consists of examining raw performance scores. However, as simple one-dimensional measures, they do not lend much insight as to what a model's advantages and shortcomings may be. This problem is exacerbated when we want to compare across different methods that solve the same problem. For instance, across a bank of different neural networks, different graphical model's, or different SVMs etc . . . .

A model is usually trained or constructed by being given sample input/output pairings that demonstrate its desired function, from which the model is expected to generalize to the rest of the input space. Thus, the way that a model can form sets in the input space (with an infinite number of points) from a finite training sample is intrinsically tied in to how it can generalize. Many of the models used today for classification such as Feed-Forward Neural Networks, Support Vector Machines, Nearest Neighbor classifiers, Decision Trees and many Bayesian Networks generate classification sets that are mostly manifolds or manifolds with boundaries. Sets of this sort exhibit strong locality properties. That is, most of the points in the set have a neighborhood surrounding them such that all points in the neighborhood are also part of the set. Thornton [85] demonstrated that many of the datasets in the UCI machine learning repository [9] contain data points that exhibit neighborhood properties, and as such are amenable to generalization by manifold type classifiers.

Given this common generalization method of classifiers, what differentiates between dif-

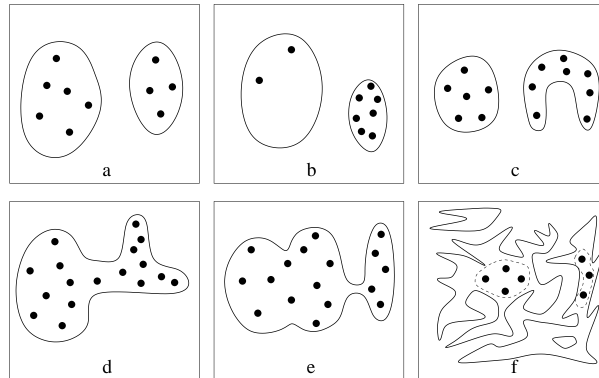


Figure 3.1: Examples of some of the variations possible in decision region structure.

ferent classifiers is how they individually partition the training points into decision regions. A classifier might only use separate convex decision regions to classify (e.g. a linear discriminant classifier). In which case, sample points are separated explicitly by completely segregating them from each other in separate decision regions. However, most interesting classifiers use more complex decision regions to organize the sample points. The points are organized into decision regions with concavity, thus creating a partitioning of the points without explicitly placing them in disconnected decision regions. In a sense this partitioning allows a finer grain of differentiation since points may be closely associated by being in a convex subcomponent of the decision region or be distantly associated through a "network" of other convex subcomponents. Figure 3.1 illustrates some of the questions that we may want to ask about the decision region structure of a classifier, and some of the interpretations of this information:

1. How many separate decision regions are used (figure 3.1a): On the one hand, too many decision regions might imply that the classifier had a hard time fitting the data to decision regions, and would lead to bad generalization. But some separate decision regions may be indicative that the data points themselves come from multiple subclasses.

2. How many and which sample points were used to decide the structure of each decision region (figure 3.1b): Generally, more points throughout a decision region raises our

confidence that it actually encapsulates the data. But sometimes having one big decision region may also imply that the data used to construct the classifier is too sparse.

3. The geometry and topology of each decision region (figure 3.1c): A decision region may be convex (the left one), such that if we take any group of points inside the decision region, the area between them is also inside the decision region. Such a decision region describes a kind of uniformity, an extended neighborhood in the input space where all points on the inside belong to one class. In contrast, a decision region might be concave (the right one), where locations between points from the decision regions may not belong to it, implying some substructure between the components of the decision region.

4. The geometry of convex subcomponents (figure 3.1d): A concave decision region can be decomposed into convex subcomponents, like the one in the figure which can be decomposed into three convex subcomponents. The purpose of this is twofold: First, the decomposition allows us a glimpse into the structure of the decision region, how it separates its different constituent portions and their interrelationships. Second, being convex, the subcomponents can be analyzed using common tools to understand their geometry. Thus, decomposition is an important step in the analysis of the larger concave decision region.

5. Connection strength of convex subcomponents (figure 3.1e): The degree that the convex subcomponents are attached together in a concave decision region is indicative of how separated they are. Concavity might be almost negligible between two strongly attached convex subcomponents (left pair), or the concavity might act almost like a complete wall between two very weakly connected subcomponents (right pair).

Our aim is to analyze the decision regions of classifiers. We would like a means to extract a classifier's decision regions and be able to decompose them individually. To do this in an exact manner would seem to be ideal. Unfortunately, in doing so we run into two related problems, the problem of dimensionality and the problem of complexity. Initially,

dimensionality appears to be our bane in the form of visualization difficulties. We can not directly visualize a decision region of more than three dimensions. However, assuming we can overcome that hurdle, dimensionality also influences the complexity of the models. For example, a neural network can have a number of decision regions that is exponential in the input dimension, where the complexity of the individual decision regions is also exponential with respect to the input dimension, as seen in the previous chapter.

In this context, it seems like an almost futile endeavor. However, there is an intrinsic discrepancy between the potential complexity of the model, the complexity of the data and the relevant complexity of a trained model. In figure 3.1f we see an example of this. The model could be representing some highly complex decision regions. However, the actual data points only reside in a simple part of the of decision regions. And with respect to these data points, the model is basically enclosing them in two pseudo-decision regions, one convex and one slightly concave. Not only is the additional complexity of the model artifactual, but if a model is successful at generalizing it must have found some underlying redundancy in the data, therefore in some respect its relevant complexity is even less than that of the data.

Our analysis method tries to extract this relevant complexity, by elucidating the properties of the decision regions in the vicinity of the data points. This is done not by directly examining the decision regions, but rather by examining the effects that the decision regions have on the relationships between the data points, and encapsulating this information in a mathematical graph, a form that allows us to make out the properties of the decision regions. This examination of the relationships between the points instead of the general decision regions is not only what allows us to extract only the relevant complexity, but also what makes the analysis method practically independent of the model type and dimensionality of the input space.

The rest of the chapter is organized as follows: We first introduce the core analysis method, a method that extracts the structure of decision regions by representing the relationships of the internal points using graphs. Following, we give a relatively simple example of its application to a neural network that classifies points in a three-dimensional space. The fact that it is three-dimensional allows us to visually compare the structure described by the

graph with the actual network decision regions. In the section after, we refine the graph analysis method, and explain the method by which we decompose the graph into the subgraphs which correspond to decision region subcomponents. We then continue with an example where we analyze two different types of classifiers, both applied to a high-dimensional letter recognition problem. Using the graph analysis method allows us to clearly show differences in the classification strategies of the two classifiers, and show where one of them will generalize incorrectly. The section after that contains a discussion about the analysis of convex subcomponents, leading to an in depth example of this on an SVM model applied to a dataset from the Statlog project[43]. We conclude with a detailed discussion of the method and some of its caveats and then discuss how the method can be extended, suggesting possible avenues of new research.

3.2 Low Level Analysis

The fundamental way that manifold type classifiers create decision regions is by enclosing points together in common neighborhoods, which is what our analysis method tries to detect. As input we are given two things, the classifier we wish to analyze and relevant labeled sample points, possibly the training data. It is important that the sample points embody the part of the input space that is of interest, otherwise we would be analyzing the classifier's artifacts and not the relevant regions. From now on, when we refer to decision regions we will mean only the relevant portions of the decision regions (Note that this relevant area can be extended almost arbitrarily if needed.)

Figure 3.2a graphically illustrates how the analysis method works. We take all pairs of points with the same classification label (in this case points A,B and C). Between each pair we extend a line segment in the input space. We then sample along this line using the classifier. In other words, we find a series of points in the input space along the line and apply the classifier to them. What we look for is a break in the connectivity, a change in the classification label in one or more of the points. Such a change implies that between the two points there is a decision region boundary, and the two points do not share a common neighborhood. Algorithm 1 explicitly describes this operation. Note that if we take a

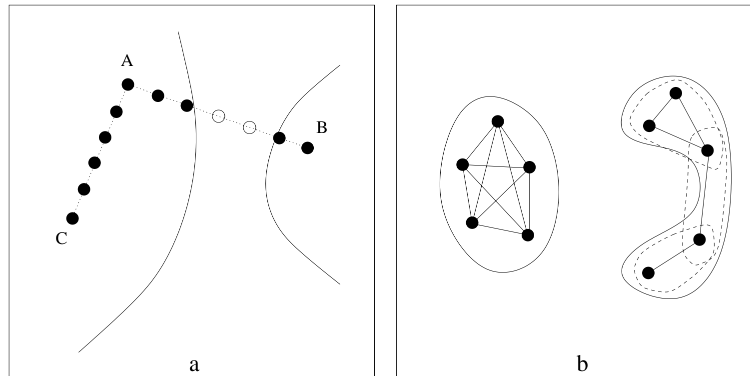


Figure 3.2: a) The connectivity graph is generated by sampling between the sample points. In this case we see how sampling between points A and B detects a boundary, but points A and C share a neighborhood. b) A connectivity graph for two decision regions, one convex and one concave.

constant number of samples on the line between each pair of points then this algorithm's complexity is $O(n^2)$, where n is the number of points.

With this connectivity information we construct a graph in the mathematical sense. In this graph each sample point is assigned a vertex, and the edges are the actual connectivity information. That is, if two points are connected in the actual input space with respect to the classifier then their vertices are connected in the graph.

This *connectivity graph* can tell us three basic pieces of information: What points reside in separate decision regions, if points are colocated in a convex decision region, or if points reside in a concave decision region. Moreover, in the latter case we can decompose this concave decision region and find what points reside in its different convex subcomponents.

Figure 3.2b illustrates how the graph relates these three pieces of information:[1]

1. If decision regions are disconnected then the graphs of the points they enclose are also disconnected. In the figure we see this with respect to two decision regions, whose internal points form two disconnected graphs in the connectivity graph.

2. When points are in a convex decision region then by definition they are fully connected

[1] In the figure the graph is superimposed over the actual two-dimensional decision regions. Thus the vertices of the graph correspond to the actual points in the input space. This is done only for visual convenience, essentially, we could have drawn these vertices anywhere.

Algorithm 1 The Connectivity Algorithm generates the Connectivity Graph by examining the connectivity between points in the decision regions.

```
X is the set of sample points.
G(V,E) is the undirected connectivity graph, where |V| = |X|.
c(x) is the classifier function, returns a 1 if x belongs to the class and other values otherwise.
v(x) is a function that returns the vertex v ∈ V corresponding to the sample point x.

Y ← X
for all x ∈ X do
  Y ← Y \ x
  for all y ∈ Y do
    delta ← (x − y) / (NUMSAMPLES + 1)
    set (v(x), v(y)) in E
    for i = 1 ... NUMSAMPLES do
      if c(x + i · delta) ≠ 1 then
        clear (v(x), v(y)) in E
        break
      end if
    end for
  end for
end for
```

and as such form a clique in the graph. We see this convexity property in the left decision region– it is convex and hence its graph is fully connected.

3. Cliques within the graphs of concave decision regions correspond to its convex subcomponents. The right decision region is not convex and so its graph is not fully connected. However cliques within its graph represent convex subregions of this concave decision region. In this example decision region there are three cliques, representing a decomposition into three convex subcomponents (shown with dashed lines.)

3.3 Analyzing a three-dimensional neural network

A 15 hidden-unit threshold neural network was trained to predict whether a thrown ball will hit a target (as shown in the previous chapter). As input, it received the throwing angle, initial velocity and the target distance (figure 3.3), Having only three inputs makes it possible to visualize its decision regions. After iterations of back-propagation and hill-climbing it achieved an 87% success rate on the training data. This system can be easily

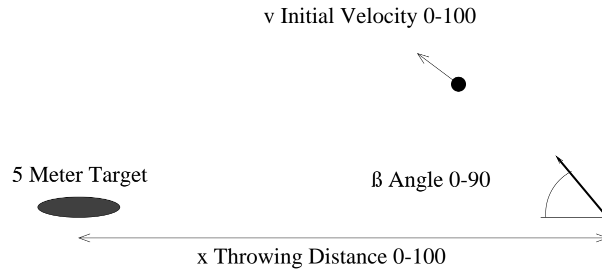


Figure 3.3: The classification task of the ball throwing network is to predict whether a ball thrown at a certain velocity and angle will hit a target at a given distance.

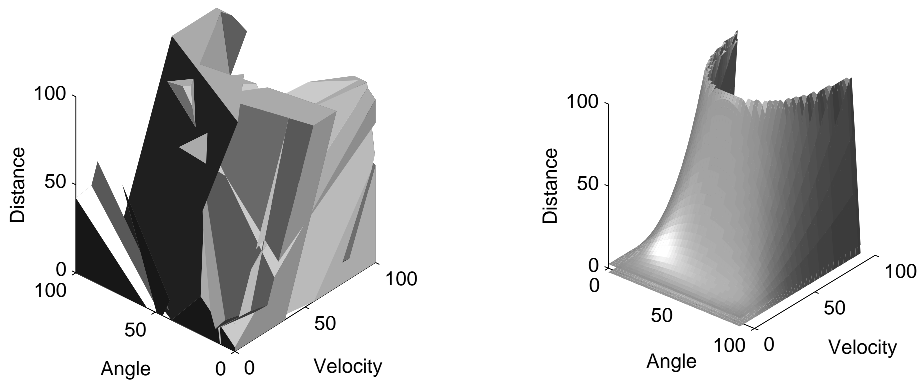


Figure 3.4: The decision region of the ball throwing network contrasted with the analytic decision region.

solved analytically, and the analytic decision region is shown in figure 3.4 contrasted with the neural network decision region which was extracted using the DIBA algorithm.

Using the first 78 of the positive training points, a connectivity graph was generated for the hit class, as seen in figure 3.5b. The graph was drawn using a spring-gravity type algorithm [14], where the edges are modeled as springs in a physical model. This drawing algorithm has the property of making highly interconnected vertices cluster together, allowing us to recognize cliques.

In the graph we can discern four different clusters that practically form cliques. Assigning a label to the vertices based on which cluster they belong to (if they belong to any cluster), we can plot the position of the actual points in the decision region corresponding to the labeled vertices (figure 3.5a, compare with figure 3.4). In this figure we can literally see that the points that make up each of these clusters correspond to four different "slabs" or

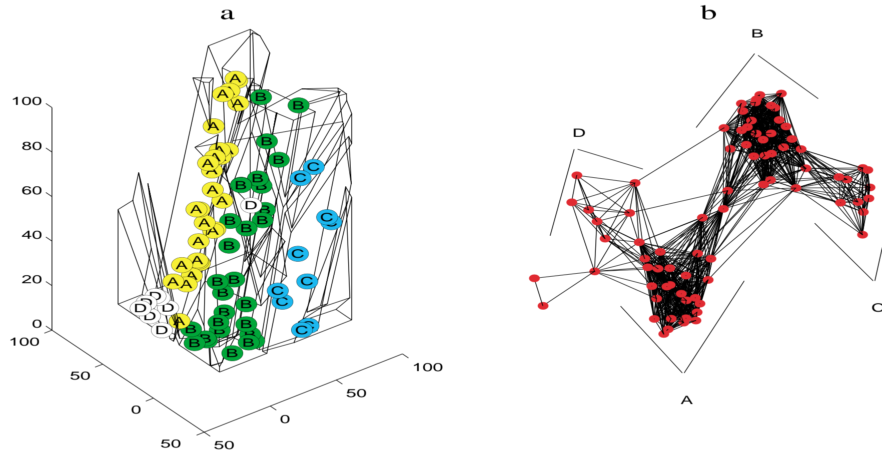


Figure 3.5: a) The points extracted from the labeling in the connectivity graph superimposed in their correct position within the decision region. b) The connectivity graph of the decision region in figure 3.4 with respect to 78 internal points. The vertices are labeled by association to four different clique like clusters.

conspicuous convex subregions that make up the actual neural network concave decision region. Also, notice how the connections between the clusters in the graph correspond to the relationships between the subregions. That is, how the slab containing the C points touches the slab containing the B points which in turn touches the A slab which touches the D slab, all properties evident in the connectivity graph.

Since we have separated the points into convex subregions we can also analyze their geometric properties. For example, by performing principal component analysis [8] (PCA) on each of these clusters of points, we can discern their dimensionality and also their orientation. Figure 3.6 shows the three eigenvalues for each of the clusters as well as for all the points combined. The eigenvalues of the clusters all have a practically negligible third eigenvalue. This indicates that they all form part of a decision region which takes up little volume in the input space, rather it is almost a two-dimensional embedding in a three-dimensional space. In contrast, this is not a property we could have discerned by just performing a PCA of all the points, since the eigenvalues of the PCA of all the points have a sizeable magnitude in all three dimensions (as seen in the figure).

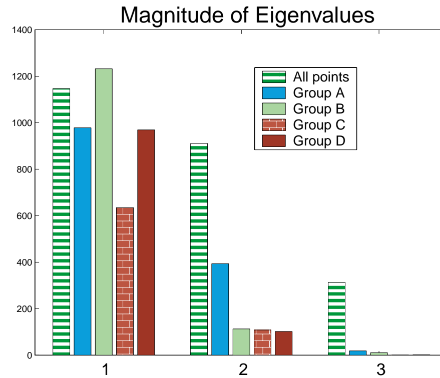


Figure 3.6: The eigenvalues of the PCA analysis for each of the groups contrasted with the eigenvalues of all the points taken together. The decomposition into groups allows us to realize that the points in the decision region form a two-dimensional embedding in the space. This would not have been discernible by just performing a PCA on all the points together.

3.4 Higher Level Graph Analysis

The connectivity graph contains the topological information about how the sample points are partitioned into decision regions. With relatively small graphs it may be possible to visually discern the different components of the graph. But with larger or more complex graphs we need an automated method to decompose the graph into convex subcomponents. This higher-level analysis method extracts two pieces of information from the connectivity graph: First, which points are colocated in the same convex subregions. Second, how do the convex subregions combine to form the original decision region. The basic assumption of the method is that multiple points inhabit the convex subcomponents of the decision region. This is a fair assumption if we expect the classifier to have generalization properties, since a manifold type classifier can only generalize by recognizing neighborhoods of points to enclose together.

In the first stage of the method we seek to group sample points with similar properties, to find points in similar locations with respect to the decision region. This is done as follows: Consider the connectivity matrix associated with the graph, where each row enumerates the edges of a vertex in the form of a binary vector. The basic grouping operation is simply: If the hamming distance between two such vectors is sufficiently small then we group their

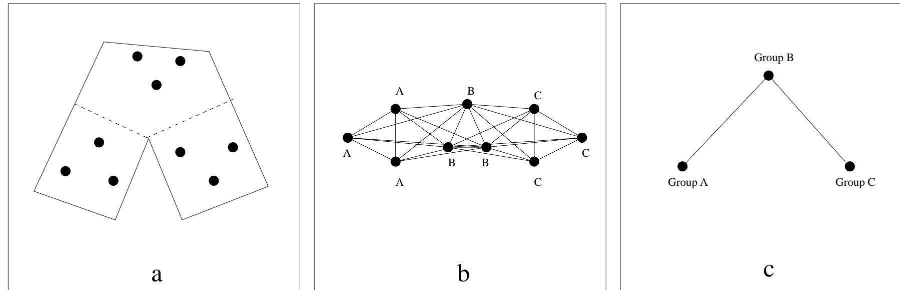


Figure 3.7: a) A concave decision region housing nine points in different convex subcomponents. b) The connectivity graph for the points in the decision region. c) The group graph associated with the labeling in the connectivity graph.

respective vertices together. From the perspective of the graph, this means that we group together vertices that are mostly connected to the same vertices and disconnected from the same vertices. The logic of this grouping mechanism comes by considering how the vertices in the graph relate to the actual decision regions. We know that if a group of points are in the same convex subregion then they should all be connected with each other, but by being in the same part of the decision region they should also all be connected to the same vertices outside their immediate clique. Therefore, they should all have a similar *connectivity signature* pattern, and that is what we look for in forming groups.

For example, consider the concave decision region in figure 3.7a and its respective connectivity graph in figure 3.7b. Notice how the points in the bottom left part of the decision region are all connected to each other by belonging to the same convex subcomponent. In addition notice that they are also all connected to the points in the top portion of the decision region, but not to any of the points in the bottom right portion of the decision region. Thus, all these points exhibit a distinctive connectivity signature which is distinct from the other convex subcomponents of this decision region. In contrast, for example, the points in the top portion of the decision region are connected to all the other points, where as the points in the bottom right portion are connected to all but the points in the bottom left region. Thus, if we label the vertices based on similar connectivity, we end up with three groups of points as illustrated. Each group represents a convex subregion, as each group's vertices forms a clique. However, these groups are differentiated with respect to

their position in the decision region, which is divined by their intergroup connectivity.

Algorithm 2 shows the grouping procedure. If we discount the computational cost of calculating the Hamming distance (which on some platforms is almost an atomic operation) then the complexity of this algorithm ranges from $O(n^2)$ in the worst case to $O(n)$ depending on the density of the groups.

Algorithm 2 The Grouping Algorithm groups together vertices with a similar *connectivity signature.*

G(V,E) is the connectivity graph.
α is the Hamming similarity.
β is the minimum group size.

$M \leftarrow V$
while $|M| > 0$ **do**
  arbitrarily pick $i \in M$
  assign new group label to i
  $M \leftarrow M \setminus i$
  for all unlabeled $j \in V$ **do**
    if HammingDistance$(i, j) < \alpha$ **then**
      assign group label of i to j
      $M \leftarrow M \setminus j$
    end if
  end for
  if group size $< \beta$ **then**
    remove label for all group members
    $M \leftarrow M \cup \{group \setminus i\}$
  end if
end while

function HammingDistance (vertex i, vertex j)
$distance \leftarrow 0$
for all $k \in V$ **do**
  if $((i,k) \in E \wedge (j,k) \notin E) \vee ((i,k) \notin E \wedge (j,k) \in E)$ **then**
    $distance \leftarrow distance + 1$
  end if
end for
return *distance*

In the second stage of the analysis we wish to simplify the original connectivity graph, in order to gauge the relationships between the convex subregions. This is done as follows: For each group, take all its vertices and merge them, transforming the group into one labeled vertex with the same intergroup connectivity as the group originally had. Doing so, we are

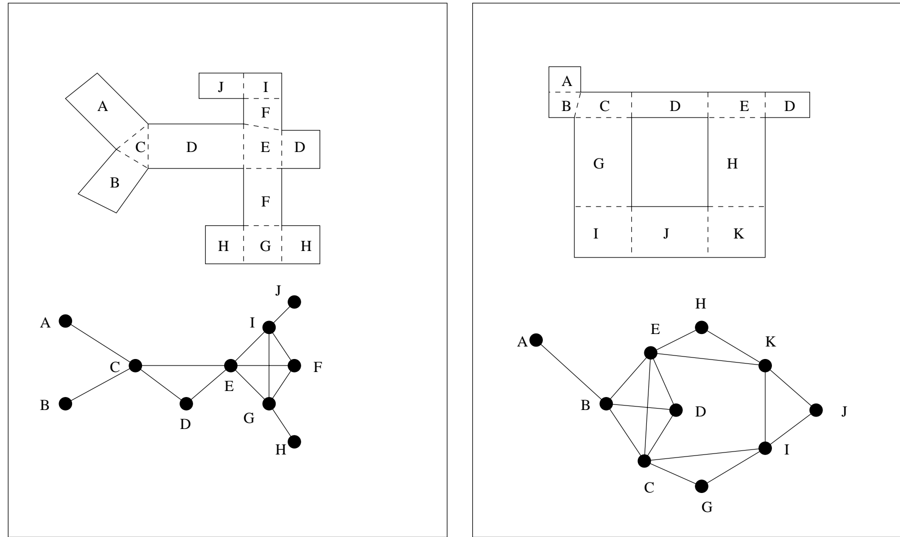


Figure 3.8: An example of two different concave decision regions and their respective group graph.

left with a much smaller and sparser *group graph* that relates the relationships between the groups– their intergroup connectivity. In figure 3.7c we see this operation performed on the connectivity graph in figure 3.7b. This new graph shows us that with respect to the sample points, the original decision region partitioned the space into three groups such that one of the groups (group B) is connected to both of the other groups, but that the other two groups are not directly connected. Notice how the graph is similar in structure to the actual decision region.

Figure 3.8 illustrate the relationship between the group graph and two different concave decision regions. Cliques in the group graph represent groups that may be combined to form larger convex subcomponents, possibly for a geometrical or statistical analysis of the points in the subregion. In the trivial case, every edge in the graph is a clique and represents a potential combined convex region. Another group graph characteristic is loops of cliques. Loops of cliques in the group graph represent the existence of holes in the decision region, as the convex subregions that go around them must be connected together. The fundamental characteristic of interest in the group graph is the branching structure. It relates the actual partitioning between convex subregions, which subregions are directly connected, which are

distantly connected and their connection paths.

The next example applies the higher-level analysis method to compare between two different types of high dimensional classifiers on the same task. By understanding the different ways that these two classifiers partition the training data we can learn how they generalize differently.

3.5 Comparing two classifiers: A high-dimensional example

The UCI repository [9] contains a dataset contributed by Alpaydin and Kaynak of handwritten digits. The original dataset contains 32 by 32 pixelated images, normalized for scale and position. There is also a preprocessed version of the dataset, where the 32 by 32 images are shrunk to 8 by 8 by counting the number of pixels in each 4 by 4 of the original. This training set contains 3823 samples from 30 people.

Using the preprocessed dataset the following classification task was constructed. The data corresponding to the numerals 3 and 4 were assigned to one class, while the remaining numerals were assigned to a second class. Thus the task consisted of classifying a 64-dimensional input into two classes.

Two classifiers were used, a sigmoidal feed-forward neural network with one hidden layer of 7 units and a K-nearest neighbor classifier with K set to 9 [8]. The network was trained using conjugate gradient [8] until it reached perfect classification on the test data.

In order to make the connectivity graph more presentable, only the first 63 cases of the 3-4 class were used to draw it. In the additional levels of analysis 300 exemplars were used.

Figure 3.9 shows the connectivity graph for the neural network. Since the graph is connected it consists of one decision region. However, it is apparent that this graph is illustrating a concave decision region because the graph consists of two highly connected regions with only very sparse connectivity between them. The points were labeled using the labeling method described above at a 90% Hamming similarity, which labeled the points as expected into two classes corresponding to these (practically) clique subregions. In the

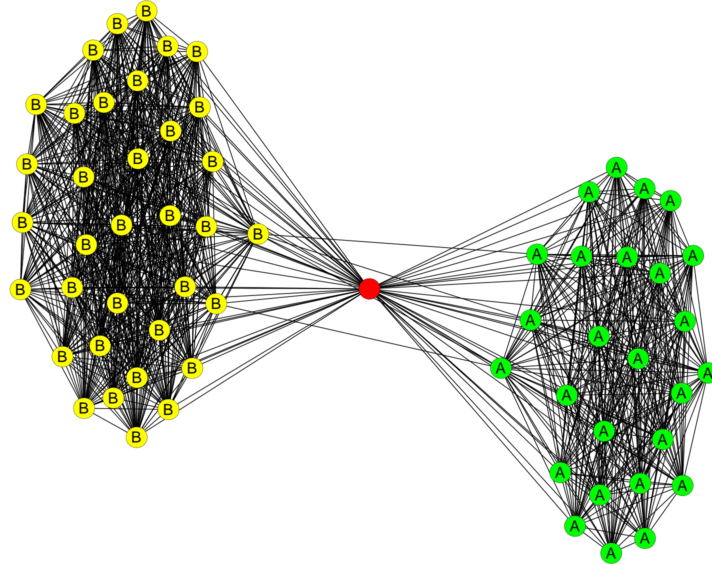


Figure 3.9: The connectivity graph of a 64-input neural network trained to classify the numerals 3 and 4 as one class and the other numerals as another class. The graph clearly illustrates that the network constructed a decision region with two separate subclasses.

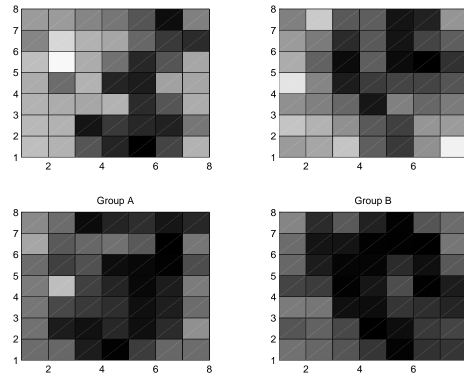


Figure 3.10: For the two labeled sets in the connectivity graph in figure 3.9 a PCA analysis was done to estimate the extent of the convex subregions, and two extreme values are shown. As seen, group A contains threes, and group B contains fours.

connectivity graph the clusters are almost completely disconnected, therefore we do not need to draw a group graph, since a group graph is only constructed when the groups make up parts of larger convex subregions.

When we examine the actual numeral associated with the labeled points we realize that the points associated with the first label all correspond to the threes, and all the points with the second label correspond to the fours. What this means is that the network discovered that the 3-4 class really consists of two subclasses and divided its decision region to clearly separate between them. Suppose that we did not know that the class was decomposable and wanted to know the composition of the subregions that the neural network generated. As before, since the subregions are convex we can analyze them using PCA. In figure 3.10, for each group, we took the mean of the points, in the top half we added the first 20 eigenvectors of the PCA normalized by their standard deviation, and in the bottom half we subtracted the same values. This gives a coarse approximation of the decision region's scope, the part of the input space that in encapsulated by the region. As can be seen, the left images correspond to threes and the right to fours, so we can literally see that the two subregions correspond to two logically separate subclasses, without looking up the original classes of the points.

Figure 3.11 shows the connectivity graph for the K-Nearest Neighbor classifier. Again, it is a connected graph and hence has one decision region. This graph doesn't lend itself to a simple visual analysis, since it is more dense. However, when we apply the labeling method at 80% Hamming similarity we get three labeled classes as illustrated. The group graph analysis of the three labeled sets shows that the vertices in group C are connected to both groups A and B, but that there are very few connections between groups A and B directly. Therefore the group graph is of the form we saw in the example in figure 3.7. In figure 3.12 we present the results of applying PCA analysis (as before) on the three different groups as well as on the two cliques of the group graph. The figure shows that group A corresponds to threes, and groups B and C correspond to fours. Since B+C and A+C form cliques in the group graph, they form larger convex regions. The PCA analysis of the composition of B and C corresponds (as expected) to fours, but the images of the composition of groups A

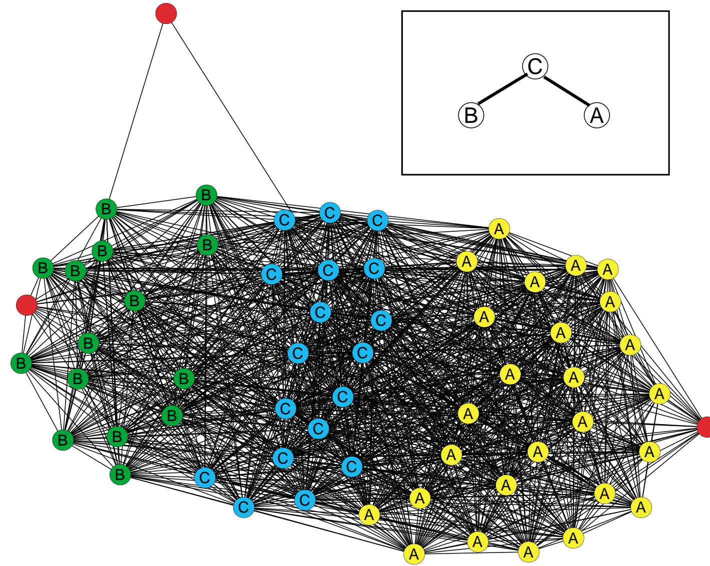


Figure 3.11: The connectivity graph of a K-nearest neighbor classifier set to classify the numerals 3 and 4 as one class and the other numerals as another class. The labeling of the graph suggests a weaker concavity than the neural network's decision region.

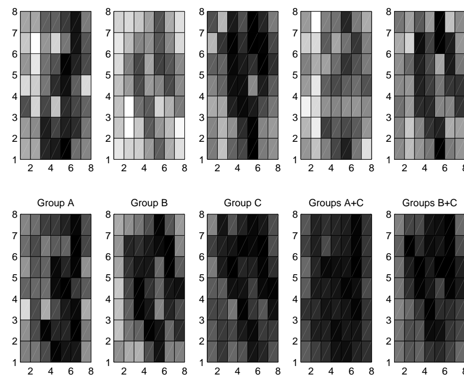


Figure 3.12: A PCA analysis was done to estimate the extent of the convex subregions of the KNN connectivity graph. Two extreme values are shown for each region analyzed. The subregion containing groups A and C is suspicious.

and C are not interpretable. This lack of interpretability goes with what we know about how the data is structured, a convex subregion, such as this one, consisting of both threes and fours would have to contain spurious data, data which is neither a three or four, and thus lead to a malformed classification set. When we examine the actual numerals associated with the labeled points, we see that the A-labeled points do correspond to threes, and the B and C labeled points do correspond to fours.

Both of the classifiers realized that the points making up the 3-4 class are not homogeneous. This is demonstrated by the fact that both classifiers used a concave decision region to house the points of the class. However the discrepancy between them lies in how clearly they realized what the two subclasses are. The neural network made a very clean distinction, clearly dividing the space between the threes and fours. Whereas the K-Nearest neighbor classifier divided some of the threes completely from some of the fours (groups A and B). However, it did not differentiate between the threes in group A and the fours in group C, hence we would expect potential misclassification in that region of the input space.

3.6 Learning from lower dimensions: A high-dimensional example

3.6.1 PCA on Ellipsoids

Due to specific properties of the previous two examples (low dimensionality and visual interpretation of the input space) we could perform a visual interpretation for the PCA results of the convex subcomponents. For many applications a visual interpretation is not possible. However, PCA results can still be numerically interpreted to gain an understanding of the strategies employed in different parts of the input space by a classifier. The next example is one which requires such a numerical interpretation, as its variables are not readily visually interpreted. Using PCA we will see how the classifier treats the variables differently across the subcomponents of its decision region. But before we proceed we need to reexamine how we perform PCA on the points of the convex subcomponents.

In the previous two examples we performed a PCA on the points that make up cliques in the standard way, using the covariance matrix. PCA is typically performed on the

covariance or correlation matrix of the data points, making the assumption that the points were sampled from an underlying distribution. The assumption of a distribution attributes a significance to the density or measure of the points. Thus, if the sample contained more points in a particular part of the space the impact on the covariance matrix would be greater than points coming from a sparser part of the space.

In our application there is no real meaning to the density of the points. Rather than representing a sample from a distribution, the points define the shape and extent of a convex part of a decision region. We know that all the points are inside the convex region and that all the space between the points is also inside the convex region. Therefore, looking at the density of the sample points would be deceiving since the complete interior of the convex hull of these points is uniformly inside the decision region. Instead, our interest lies with the geometric properties of the convex hull defined by the points.

It is difficult to study the convex hull directly. As such, we aspire to approximate the hull with a more regular shape, specifically an ellipsoid. Using an ellipsoid to approximate data is a common approach both in statistics [65] and in other domains [69]. This is in part due to the relationship proved by Lowner and John [50] between the minimum volume enclosing ellipsoid of the hull and the maximum volume enclosed ellipsoid. They proved that these two ellipsoids are concentric and the same except for a constant shrink factor. Thus implying that the *minimum volume ellipsoid* (MVE) would make a reasonable approximation to a convex region by somehow capturing and bounding its geometry.

The advantage of using an ellipsoid to approximate the convex region is that it allows us to continue using PCA to understand the shape of the region. Instead of performing the PCA on the covariance matrix we apply it to the scatter matrix of the ellipsoid, using the geometric interpretation of PCA [40]. Given an ellipsoid described by the equation:

$$(x - \mu)'\Sigma^{-1}(x - \mu) = c$$

Where μ is the center of the ellipsoid, Σ^{-1} is the scatter matrix and c is a constant equal to the dimension of the space, then the principal components of Σ correspond to the directions of the principal axes of the ellipsoid, and the eigenvalues can be used to calculate the half-

lengths or radii of the axes.

In the next example we demonstrate how to use this method of performing PCA on the MVE's scatter matrix. There, we calculate the MVE using the algorithm proposed by Titterington [88], a relatively fast iterative algorithm. Note that this is an area of active research and there are other algorithms to compute the MVE [75].

In this example we will also show another advantage of using ellipsoids. By enclosing points in ellipsoids in the analysis we indirectly build an alternative classifier with the same group structure as the one we are studying. On the one hand, this ellipsoid classifier allows us to verify our analysis. But, as we will see, it also allows us to transfer decision region structure across different input dimensions.

3.6.2 The task and classifier

The vehicle database [77] used in the Statlog project [43] describes the silhouettes of four different types of vehicles: an Opel, a Saab, a van and a truck. Each entry in the database contains 18 different geometrical and statistical measures of the respective silhouette. There are 846 entries in the database, which after random shuffling, were divided into a 550 entry training set and 296 entry test set. As stated before, the previous two examples had a visual component to their PCA/convex analysis. In this dataset we do not have that luxury, which is one reason it was chosen, to demonstrate how the analysis can be done on raw numerical data.

Using the SVM-Light package [39] a degree 3 polynomial kernel SVM (61 support vectors) was trained to recognize the van vehicle using only the first 6 of the 18 measures (normalized to the range [-1,1]). The reason we used a smaller number of inputs was primarily to simplify the analysis, having fewer variables implies less interdependencies between them. Using the first six inputs was an arbitrary choice. Any subset which allowed the classifier to achieve sufficient accuracy on the task could have been used. This particular classifier achieved 98.1% on the training data and 93.9% on the test set.

Using the training data a connectivity analysis was conducted rendering the group graph in figure 3.13. As can be seen, the classifier consists of one concave decision region with 4

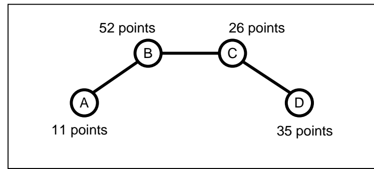


Figure 3.13: The group graph of the six input SVM model classifying the van class from the vehicle dataset.

connected convex subregions, A, B, C and D. The connectivity between these regions means that A and B form a larger convex subregion, so do B and C, and C and D. So, for the sake of analysis, we can decompose the decision region into 3 overlapping convex subregions.

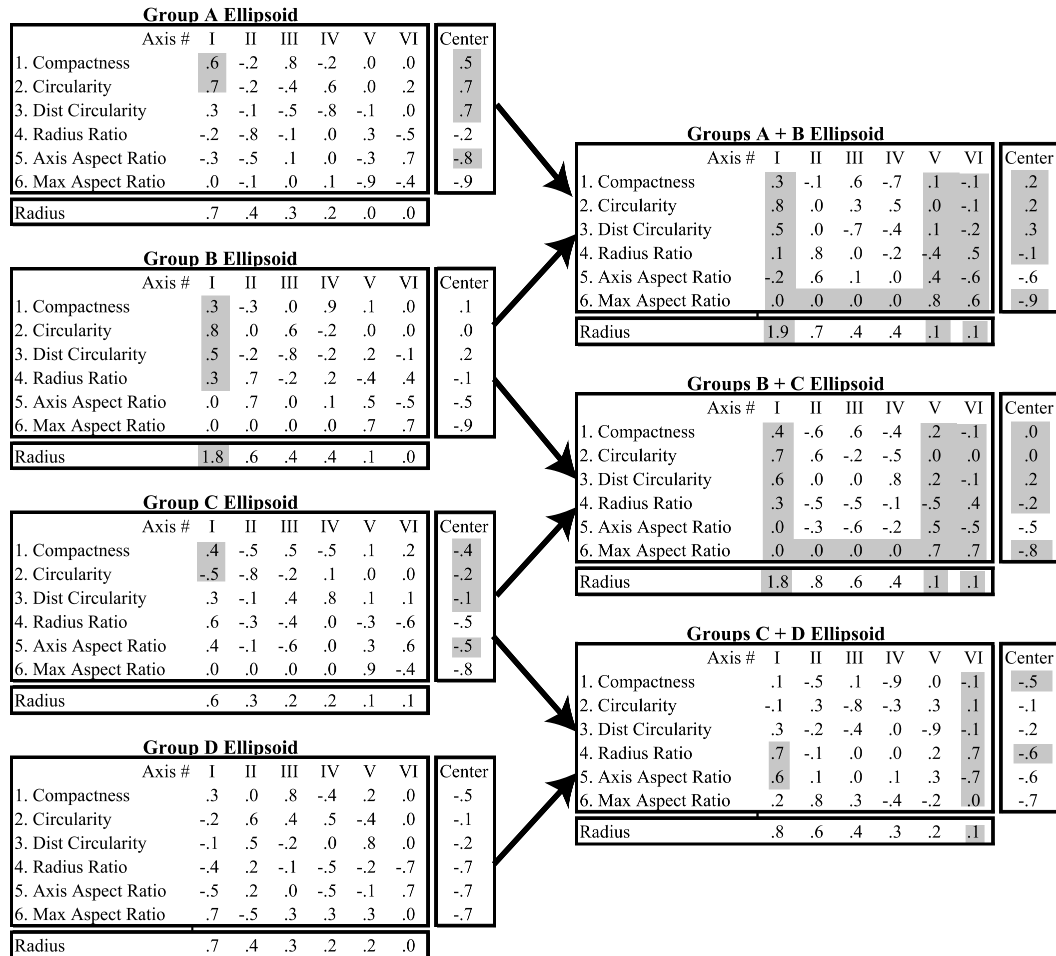
Figure 3.14 contains the results of the ellipsoid PCA method applied to each of the convex subregions. Note that the values in the tables are rounded. This is common practice as PCA is robust to rounding with respect to interpretability [40]. Next, we will interpret these tables in order to understand what geometric properties define membership in the van class, for each of these different convex subregions of the input space.

3.6.3 PCA Interpretation

Interpretation of PCA data is an art unto its own [40]. To generalize for the sake of simplifying the process, the principal components are roughly divided into two sets: those with a relatively high eigenvalue or radius, and those with a small eigenvalue or radius.

Those components with a large radius describe correlations in the data. That is, in a principal component vector with a large radius, the variables with a large magnitude are variables that move together: Either they move in the same direction together (positive correlation) if they have the same sign, or they move in opposite directions together (negative correlation) if they have opposite signs.

On the other hand, components with a small radius act as constraints, describing relationships between the variables in the component vector which must be maintained in order to stay within the decision region. Constraints are probably more important to understanding a classifier's decision regions than are correlations, as the constraints describe the sharp boundaries between belonging or not belonging to the class.



Group A Ellipsoid

Axis #	I	II	III	IV	V	VI	Center
1. Compactness	.6	-.2	.8	-.2	.0	.0	.5
2. Circularity	.7	-.2	-.4	.6	.0	.2	.7
3. Dist Circularity	.3	-.1	-.5	-.8	-.1	.0	.7
4. Radius Ratio	-.2	-.8	-.1	.0	.3	-.5	-.2
5. Axis Aspect Ratio	-.3	-.5	.1	.0	-.3	.7	-.8
6. Max Aspect Ratio	.0	-.1	.0	.1	-.9	-.4	-.9
Radius	.7	.4	.3	.2	.0	.0	

Group B Ellipsoid

Axis #	I	II	III	IV	V	VI	Center
1. Compactness	.3	-.3	.0	.9	.1	.0	.1
2. Circularity	.8	.0	.6	-.2	.0	.0	.0
3. Dist Circularity	.5	-.2	-.8	-.2	.2	-.1	.2
4. Radius Ratio	.3	.7	-.2	.2	-.4	.4	-.1
5. Axis Aspect Ratio	.0	.7	.0	.1	.5	-.5	-.5
6. Max Aspect Ratio	.0	.0	.0	.0	.7	.7	-.9
Radius	1.8	.6	.4	.4	.1	.0	

Group C Ellipsoid

Axis #	I	II	III	IV	V	VI	Center
1. Compactness	.4	-.5	.5	-.5	.1	.2	-.4
2. Circularity	-.5	-.8	-.2	.1	.0	.0	-.2
3. Dist Circularity	.3	-.1	.4	.8	.1	.1	-.1
4. Radius Ratio	.6	-.3	-.4	.0	-.3	-.6	-.5
5. Axis Aspect Ratio	.4	-.1	-.6	.0	.3	.6	-.5
6. Max Aspect Ratio	.0	.0	.0	.0	.9	-.4	-.8
Radius	.6	.3	.2	.2	.1	.1	

Group D Ellipsoid

Axis #	I	II	III	IV	V	VI	Center
1. Compactness	.3	.0	.8	-.4	.2	.0	-.5
2. Circularity	-.2	.6	.4	.5	-.4	.0	-.1
3. Dist Circularity	-.1	.5	-.2	.0	.8	.0	-.2
4. Radius Ratio	-.4	.2	-.1	-.5	-.2	-.7	-.7
5. Axis Aspect Ratio	-.5	.2	.0	-.5	-.1	.7	-.7
6. Max Aspect Ratio	.7	-.5	.3	.3	.3	.0	-.7
Radius	.7	.4	.3	.2	.2	.0	

Groups A + B Ellipsoid

Axis #	I	II	III	IV	V	VI	Center
1. Compactness	.3	-.1	.6	-.7	.1	-.1	.2
2. Circularity	.8	.0	.3	.5	.0	-.1	.2
3. Dist Circularity	.5	.0	-.7	-.4	.1	-.2	.3
4. Radius Ratio	.1	.8	.0	-.2	-.4	.5	-.1
5. Axis Aspect Ratio	-.2	.6	.1	.0	.4	-.6	-.6
6. Max Aspect Ratio	.0	.0	.0	.0	.8	.6	-.9
Radius	1.9	.7	.4	.4	.1	.1	

Groups B + C Ellipsoid

Axis #	I	II	III	IV	V	VI	Center
1. Compactness	.4	-.6	.6	-.4	.2	-.1	.0
2. Circularity	.7	.6	-.2	-.5	.0	.0	.0
3. Dist Circularity	.6	.0	.0	.8	.2	-.1	.2
4. Radius Ratio	.3	-.5	-.5	-.1	-.5	.4	-.2
5. Axis Aspect Ratio	.0	-.3	-.6	-.2	.5	-.5	-.5
6. Max Aspect Ratio	.0	.0	.0	.0	.7	.7	-.8
Radius	1.8	.8	.6	.4	.1	.1	

Groups C + D Ellipsoid

Axis #	I	II	III	IV	V	VI	Center
1. Compactness	.1	-.5	.1	-.9	.0	-.1	-.5
2. Circularity	-.1	.3	-.8	-.3	.3	.1	-.1
3. Dist Circularity	.3	-.2	-.4	.0	-.9	-.1	-.2
4. Radius Ratio	.7	-.1	.0	.0	.2	.7	-.6
5. Axis Aspect Ratio	.6	.1	.0	.1	.3	-.7	-.6
6. Max Aspect Ratio	.2	.8	.3	-.4	-.2	.0	-.7
Radius	.8	.6	.4	.3	.2	.1	

Figure 3.14: This figure shows the PCA analysis of each of the ellipsoids used to approximate the convex regions of the SVM classifier. The hi-lighted values are discussed in the text.

How does a small radius principal component express a constraint? Since the component represents a small axis of the underlying ellipsoid then it implies that moving from the center in the direction of the axis vector will quickly take us out of the ellipsoid. Hence the component acts as the direction of the border off the decision region. A complementary algebraic perspective is to consider that any point $\overline{x}$ in the ellipsoid can be expressed as $\overline{z}$ using the coordinate system given by the PCA vectors from the center of the ellipsoid. The component z_i of the coordinate corresponding to the i-th principal component $\overline{p_i}$ is calculated by the inner product $\overline{x} \cdot \overline{p_i}$, due to the orthonormality of the principal components. To be inside the ellipsoid z_i has to be smaller than the radius of $\overline{p_i}$. But since the radius of $\overline{p_i}$ is very small or almost negligible we get the approximate equation $\overline{x} \cdot \overline{p_i} = 0$, a direct linear constraint on the values of $\overline{x}$.

3.6.4 The Analysis

We would like to analyze the larger convex subregions, AB, BC and CD. In our analysis we are interested in answering two questions: 1) What part of the input space does the ellipsoid address? 2) What unique identifier of belonging to the class, or constraints on the data are imposed by the ellipsoid?

First consider the group consisting of C and D (figure 3.14). Starting with what part of the input space is covered by the ellipsoid, we contrast the center of this ellipsoid with the other two ellipsoids to find that it differs mostly from the other two by having a low Compactness (COM) and a lower Radius Ratio (RR). Looking at the COM values in the principal components factored by their radii, we find that this low COM value is true for the full ellipsoid as it is limited in this ellipsoid to stay below -0.2. We next try to find the properties of the points in the ellipsoid. Examining the PCA, what is particularly interesting is the last PC, one which expresses a constraint. In this PC all but the 4th and 5th elements, RR and Primary Axis Aspect Ratio (PAAR), have an almost negligible value. Since the value of these two significant variables is almost equal but of opposite sign, then we arrive at the approximate constraint: $RR - PAAR = 0 \Rightarrow RR = PAAR$, a direct strong limitation on class membership. The first PC, the one with the largest radius, also demonstrates that this

constraint is the source of greatest variance for the set. In the first PC, the RR and PAAR elements have the most significant magnitude of any other variable. Their magnitude is also very similar between them, implying that they are correlated and move together across the full stretch of the large radius. Thus the group CD limits membership in the van class to points with a relatively low compactness, which are constrained to have an almost equal RR and PAAR.

The groups AB and BC share many common properties. First, their centers are very similar, hovering near zero for the first 4 variables and having a low Maximum Length Aspect Ratio (MLAR). In fact this low MLAR is a constraint for both of these groups. Looking, for example, at the group BC (the same is true for AB), whose last two PCs act as constraints, we can see that other than the MLAR the variables in the two constraints are mirrors of each other, their magnitudes are close but their signs are reversed. Thus, adding the two constraints together, the other variables cancel out and we conclude that the MLAR is constrained to be zero (relative to the center of the ellipsoid). Note that this is also borne out in the larger radius PCs, where the magnitude of the MLAR is negligible. Another point of similarity between these two groups is their first PC. This PC has a very large radius, allowing for the Circularity (CIR) and Distance Circularity (DC) to move together through almost their full value range, between -1 and 1. There is also some correlation with COM across this major axis of the ellipsoid. This major axis is obviously a contribution from the points in the B group which also has a very similar first PC.

Given these 3 large similarities between AB and BC, what are the differences that put them into two separate convex regions? While these differences are hidden in the respective PCs of these groups, it is easier to examine their two unconnected subcomponents, A and C. The centers of A and C convey that these two groups are from largely different regions of the input space. Where A is located to enclose points with a high COM, CIR, DC and relatively low PAAR, C encloses points with a low COM, relatively low CIR and DC, and a relatively higher PAAR. Other than that important difference, the constraints of these two groups are similar– both constraining the MLAR to zero, with a few small variations. The fact that their kernel is similar (the subspace defined by their constraints) implies

that these ellipsoids have a similar orientation in the input space. Nevertheless, we can still discern an important difference between them by looking at their first PC. Where as group A's PC shows a positive correlation between COM and CIR, group C's PC shows a negative correlation. Thus, in the direction of maximal change, these groups show an opposing relationship between these variables (as well as others).

In summary, using the PCA analysis on the scatter matrix of the ellipsoids we saw that the CD ellipsoid primarily addresses data points with a low COM and varying MLAR, by constraining membership in the class to points having similar RR and PAAR values. We then saw that the AB and BC ellipsoids both allow their CIR, DC and to some degree their COM to vary together a great deal under the constraint that their MLAR stays constant. Their main differences stem from the contributions of the A and C groups which tackle points in the opposite extremes of COM, CIR and DC by imposing variations on the relationships of these variables. Thus, we conclude from the analysis that the classifier's construction of a concave decision region facilitates imposing a different classification strategy on the different parts of the input space.

3.6.5 The ellipsoids as a model

In our previous analysis we constructed ellipsoids to enclose the points belonging to each of the composite groups in order to analyze them using PCA. In doing so we have indirectly constructed an alternative classifier, the model which consists of these ellipsoids. That is, we can take an unclassified point in the input space and check whether it is on the inside of any of the ellipsoids, if so it is classified as belonging to the van class.

We now compare this model with the original SVM model. Where the SVM model achieved 93.9% on the test set, the ellipsoid model achieved 91.2%. Of the 296 entries in the test set the output of the two models agreed on 254 entries, 85.8%. The SVMs average positive output response was 5.64 and -10.8 for negative outputs. For the points where the models disagreed the SVMs average positive output was 2.84 and the average negative output was -2.59. The fact that these responses are closer to zero implies that these points of contention between the models are points which are close to the SVM's boundary. We

would not expect the ellipsoids to be an exact match to the SVM model; differences in the underlying forms of the decision boundaries, and limited information about the exact nature of the SVM decision boundary would preclude that. However, as an approximation the ellipsoid model gives a reasonable match to the SVM, capturing a large part of the essence of its classification strategy.

Another SVM was trained using the same kernel on the full 18 input classification problem (56 support vectors). It achieved 97.6% accuracy on the test set. The connectivity analysis of this classifier showed that its decision strategy with respect to the training set consists of one large convex region. Thus, in the process of adding input variables, some of the concave structure present in the lower dimensional model was removed. There could be different reasons for this, but it is a fair assumption that the "Curse of Dimensionality" [8], the fact that as we increase the input dimensions the problem becomes exponentially less specified, is involved. This allows for a structurally simpler model (one convex decision region as opposed to a concave one composed of 4 parts) to fit the data, as the added dimensionality loosens the restrictions on the shape of the decision regions.

Fitting a minimum volume ellipsoid to the data gave a classifier with 87.84% accuracy on the test data. However, this model does not take into account any margin information (where to put the boundary between the van and other classes.) We took a naive approach to this, just expanding the model by a factor. As such, the ellipsoid would remain with the same center and relative axes proportions, but we would expand or shrink it appropriately. Rather than using an absolute factor we calculated the factor that it would take to bring the ellipsoid to just touch the nearest member of the other class outside the ellipsoid, and normalized the factors to that value. Thus, a factor of 1 corresponds to just touching the first member of the other class. This expansion and contraction approach is related to the Restricted Coulomb Energy algorithm [63]. Using the test set for validation the ellipsoid was expanded by a factor of 2.4, giving an accuracy of 96.96% on the test set, misclassifying only two examples more than the SVM.

Even though the 18 input SVM model did not display the same structure as its lower dimensional counterpart, that structure can still be applied to the 18 input problem. Con-

sider the connectivity graph as a way to organize the data points. In essence it defines which points go together in convex decision regions. Thus, we can build the same ellipsoid model used in the 6 input case, in terms of which points to place in which ellipsoid, but use the full 18 dimensions of the input for the MVE construction. Doing so renders a model with 80.4% accuracy on the test set. Using a factor of 1 to adjust the margins of all three ellipsoids gives a model with 97.97% accuracy, and validating with respect to the test set (factor=1.15) gives a model with 98.99% accuracy on the test set. That is an improvement from 2.4% error for the SVM to 1% error. Even though it is hard to draw strong conclusions from this result it does speak to an important issue in decision region based classifiers, over-generalization. The model with one convex decision region can perform well on the test data. However, ultimately the task of the classifier is to define the class set, what it means to be a van. By using one convex region to enclose all the points, the model allowed the class to be, at the very least, any point between the vans represented in the training data. Thus, rather than finding what makes a van a van, it found what makes a van not an Opel, Saab or truck. In a world with more than four types of cars the model would most probably misclassify other inputs which fell into its decision region. In the lower dimensional case, the training data was denser (relative to the dimensionality of the input space) and forced greater constraints on the shape of the decision regions. By using this additional structure in the higher dimensional case we may be getting closer at what it means to be a van.

3.7 Analysis Method Details and Discussion

3.7.1 Low-Level Analysis

In the low-level analysis stage the connectivity graph is constructed by sampling between the sample points. There is only one parameter that may be varied at this stage, the sampling rate on the line. Unlike DSP applications where assumptions can be made about the data source, allowing use of the Nyquist frequency to sample, in our particular case there is no one correct frequency, as illustrated in figure 3.15. In the figure we see that as our two sample points approach the decision boundary at the corner, we will need a continuously increasing sampling rate to detect the output transition. The consequence of losing that transition is

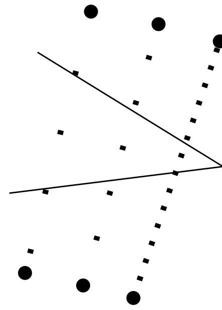


Figure 3.15: There is no absolute sampling frequency which will guarantee the detection of all decision boundaries. In this figure we see that the closer we come to the corner the higher the sampling rate must be to detect the boundary.

that the two sample points may seem connected when there is in fact a region between them with a different output. Thus, we can never lose the fact that two points are connected, only that there may be a hole between them. Increasing the sampling frequency increases our sensitivity to smaller and smaller holes. From an applied perspective, ultimately it comes to the question of how small a transition is important to detect. Typically, the heuristic used is to try a few sampling frequencies until an increase in the sampling rate does not significantly change the connectivity graph.

Another decision to be made is what points to use. In this chapter we have used the points of the training set. However, there is no reason to be limited to only those points. One can also use the points of the test set, validation set, and unlabeled data. In addition it is possible to generate additional points to explore the classifier's response in under represented parts of the input space. For example, if we were interested in the shape that the decision region has between two classes (at the margin), we might generate new points by sampling between the two classes to find points near the boundary to be used for the connectivity analysis.

3.7.2 Higher-Level Analysis

At the higher-level analysis stage we want to construct a group graph out of the connectivity graph. The grouping approach at this stage is to compare the connectivity vector of vertices using the Hamming distance. The first stage is to calculate a rough grouping using algorithm

2.

Having a rough group division in hand we construct a matrix G describing the intergroup connectivity. Let i,j be the labels of two groups, let S_i be the set of vertices with label i, and define $n(l, S_i)$ to be the number of connections that vertex l has with group S_i. Then the matrix G is constructed as (where the equalities are due to the symmetry of the connectivity matrix):

$$G_{ji} = G_{ij} = \frac{\sum_{l \in S_i} n(l, S_j)}{|S_i|\,|S_j|} = \frac{\sum_{l \in S_j} n(l, S_i)}{|S_i|\,|S_j|}$$

What G describes is how connected any two groups are. Each entry is a value between 0 and 1, where 1 implies that the two groups are 100% connected. Typically most values are not 0 or 1, but somewhere in between. Usually we apply a cutoff, for example groups with over 85% connectivity are considered connected, and groups with less than 15% connectivity are considered disconnected.

After the first round of grouping we may end up with groups with ambiguous connectivity. That is, their intergroup connectivity value is somewhere between the cutoff values, not allowing us to explicitly state whether they are connected or not. The reason for this is that during the grouping procedure we purposely allow a degree of robustness in the form of the α parameter. The function of this robustness is to allow vertices with similar but not identical connectivity signatures to be grouped together, and as such, to gloss over small noise or irregularities in the decision region structure. However, the side effect of this robustness is that it may allow groups to form which consist of points that, at a coarse level, have a similar connectivity signature, but with respect to a few groups may have different connectivity. Fortunately, the solution to this is quite simple, a refined grouping. After identifying two groups with ambiguous connectivity, we perform a second round of grouping on the members of one of the groups. However, when we check the Hamming distance we do so only with respect to the members of the other group. What this does is to split the original group up with respect to how the vertices are connected to the ambiguous second group. Typically, we may end up splitting the group up into two groups based on whether or not they are connected to the ambiguous group. The procedure may be repeated until

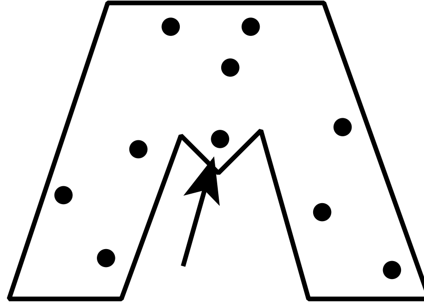


Figure 3.16: The point at which the arrow is pointing has a unique connectivity signature which would preclude including it in any group. However, it is part of the convex region defined by the top 3 points.

we are left with an interpretable group connectivity matrix G, allowing us to draw a group graph of the relationships between the groups.

3.7.3 Convex Region Analysis

In the convex analysis stage, we wish to locate which points go together in convex regions, and to apply a convex analysis methodology to those points. During the labeling of the various groups, some points that are part of convex regions may end up not being labeled. We call these *corner points*. As figure 3.16 illustrates these corner points are points that, due to their position in the decision region, may not have a connectivity pattern that is shared by other points. Thus, before we start the convex analysis, we go through each unlabeled point and check which groups it is fully connected to. Then, when we do the actual convex analysis we include those unlabeled points that are also fully connected to the group clique being analyzed.

In this chapter we have used PCA on the covariance matrix of the data and PCA on the scatter matrix of the MVE. There are other methodologies that may be applied to analyze convex regions, for example convex regions may be approximated by axis-parallel boxes [7; 93], allowing a "rule like" representation of the data. Unfortunately, a detailed discussion of these and other methods are beyond the scope of this manuscript.

3.8 Extending and Generalizing the Method

The Decision Region Connectivity Analysis method (DRCA) as described here is applied to decision regions in the input space. At times we may want to interpret the classifier as first applying a coordinate transformation to the data (feature extraction, hidden unit space, kernel space, etc) and then enclosing the transformed data in decision regions. In such a case we can apply the DRCA method in this transformed space instead of the input space. For example, for a neural network we may apply the method to the data points only as they are represented in the first layer of hidden units, thus sampling along lines in the hidden-unit space instead of the input space.

It is also interesting to note that the only location in the basic algorithm that we make an assumption about the metric of the input space is in how we define what it means to sample "on the line". Thus, another possible modification to the algorithm would be to adjust this definition to reflect prior knowledge about how either the input space is organized or how the classifier interprets it. However, modifications of this sort reflect changes in the neighborhood definitions, and as such would preclude the typical analysis of convex regions using PCA.

In general though, how do we extend this method to other types of classifiers? It may seem that we want to mathematically abstract the cornerstones of the method, introducing abstract notions of convexity and concavity. Singer [79], describes this procedure as "One selects a certain property that usual convex sets in R^n have, but many other objects in possibly other settings also have, and one uses that property to define a 'generalized' sort of 'convexity'". Even though this sounds vague it hints at the real question, why are we interested in convexity to begin with?

When a model is given training data, it is given a sampling of a class set, and asked to deduce the actual complete set. So if a model has generalized from training data then the set it has constructed approximates the actual underlying category set. Thus the model builds a larger (probably infinite) set from a finite sample. Whether the model will generalize successfully depends on whether these larger sets are the right form of extrapolation from

the sample. Our goal being to understand the model's class set, we want to describe these infinite sets that are derived from the sample. In a decision region type classifier, points are enclosed in contiguous volume filling areas of the input space. By examining where points are fully enclosed in convex portions of decision regions, we can infer how the model generalizes. These convex portions describe the unique, contiguous, volume filling parts of the input space where all the points belong to the class. Thus, convex regions are the essential mechanism by which these models construct infinite sets from finite samples. In order to adapt this method to a different type of classifier we would need to understand how it constructs its infinite class sets from samples and find a method to recognize and analyze these sets.

3.9 Conclusion

Many classifiers operate by constructing complex decision regions in the input space. These decision regions can be few or many, convex or concave, have large or small volumes, etc. By focusing on the sample points enclosed in these regions we have demonstrated a method with low computational complexity, DRCA, to extract these properties which is independent of the classifier type or the dimensionality of the input space. It thus allows us not only to analyze individual high-dimensional classifiers, but to compare completely different classifier models on the same problems. We demonstrated this method on a number of examples: Analyzing a 3-dimensional neural network, allowing a comparison of the method with its actual decision region; Comparing a neural network and KNN classifier on a handwritten digit classification problem, and demonstrating fundamental differences in their generalization strategy; Analyzing a high-dimensional SVM model, and demonstrating how it partitioned its decision region to apply different classification strategies to different parts of the input space.

This method offers a significant opportunity in helping to unite a field with many models and approaches by giving an analysis tool which addresses their greatest common denominator, their method of generalization, thus allowing the qualitative comparison of present and future high-dimensional classifiers.

Chapter 4

A Gradient Descent Method for a Neural Fractal Memory

Unlike the direct-mapping computation exhibited by feed-forward and other decision region type models in the previous two chapters, in recurrent neural networks there is no obvious interpretation of computation. As such, there have been numerous interpretations of the function that dynamics serve in recurrent network computation. The Hopfield network uses the fixed points of the network dynamics to represent memory elements. Networks studied by Pollack [61], Giles [28], and Casey [16] use the current activation of the network as a state in a state machine while using the dynamics of the network as the transition map. Some try to model existing dynamical systems with recurrent neural networks [89]. RAAMs [60] (as seen in the next chapter) use the network dynamics to describe complex data structures such as trees and lists.

In this work we exploit the fact that it has been demonstrated that higher order recurrent neural networks exhibit an underlying fractal attractor as an artifact of their dynamics [83]. These fractal attractors offer a very efficient mechanism to encode visual memories in a neural substrate, since even a simple twelve weight network can encode a very large set of different images.

The main problem in this memory model, which so far has remained unaddressed, is how to train the networks to learn these different attractors. Following other neural training methods, this chapter we propose a Gradient Descent method to learn these attractors.

The method is based on an error function which examines the effects of the current network transform on the desired fractal attractor. It is tested across a bank of different target fractal attractors and at different noise levels. The results show positive performance across three error measures. This material has appeared at the IJCNN 98 conference, where it received the **Best Student Paper Award**.

4.1 Introduction

We employ a novel interpretation of the computation performed by the network dynamics [83], in which the network is treated as an *Iterated Function System* that is coding for its underlying fractal attractor [4]. The fractal attractors used in this work are two dimensional, hence the network is in effect coding for fractal images, and may be acting as a form of visual memory.

Iterated Function Systems (IFS's) are a set of simple functions. Each function receives as input a coordinate from a space and returns a new coordinate which is usually a simple transformation of the input coordinate. When these functions are applied iteratively to points in a space, they converge on a set of points, called the IFS's attractor. This attractor is a fractal, a set with similar structure at different resolutions.

The connection between IFS's and recurrent neural networks comes from thinking of a network's neurons as the functions or transforms of an IFS. As such, these neurons receive (X,Y) coordinates as input and return new ones in a recurrent manner.

It has been suggested that coding fractals by Iterated Function Systems may be an effective mechanism for compressing images [5]. As such this interpretation of network dynamics may form the basis of a highly efficient method for storing visual information and other related memories.

A small sample of some of the fractals which a simple network of only four neurons can encode is shown in Figure 4.1. It is conceivable that this rich and interesting set of fractals may be used to encode real-world visual images or at least some of their properties. To make this interpretation of recurrent neural networks as storing a fractal attractor applicable, it is necessary to demonstrate a mechanism by which these attractors can be learned or

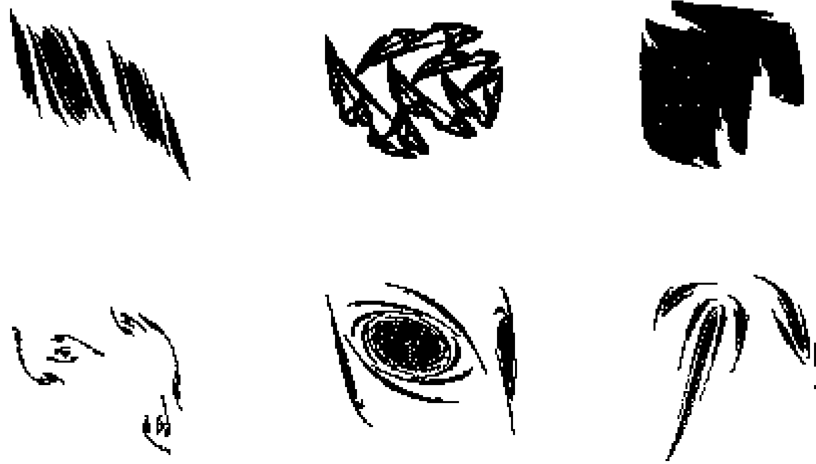


Figure 4.1: Example fractal attractors that can be generated with a four neuron network.

approximated by the network. This issue is related to the *Inverse Fractal Problem*, an area of active research, which asks how, given an image, do you find the Iterated Function System which can generate the image.

There have been different approaches towards solving the inverse fractal problem; the main motivation for this research has been image compression using fractals. Many of the approaches use generalizations of IFS's such as LIFS's which use local-similarity as well as self-similarity. The *method of moments* uses invariant measures of moments to match a function system to a target image [91]. *Genetic Algorithms* have also been used to address this problem [74]. The current generation of successful fractal image compression algorithms succeed by severely limiting the space of transforms to be used [38; 25]. At present there is still no general algorithm for solving the inverse fractal problem. It seems to be an elusive problem related to the classic problem of object recognition under transformation.

In the vein of other learning algorithms for neural networks, such as the ubiquitous *back-prop* [66] and many which have come since, we developed a training method for our network which relies on an energy or error function that we seek to minimize by means of a gradient descent on its energy landscape. In effect, minimization of this error function will lead to the network learning the desired attractor. In the rest of this chapter we describe

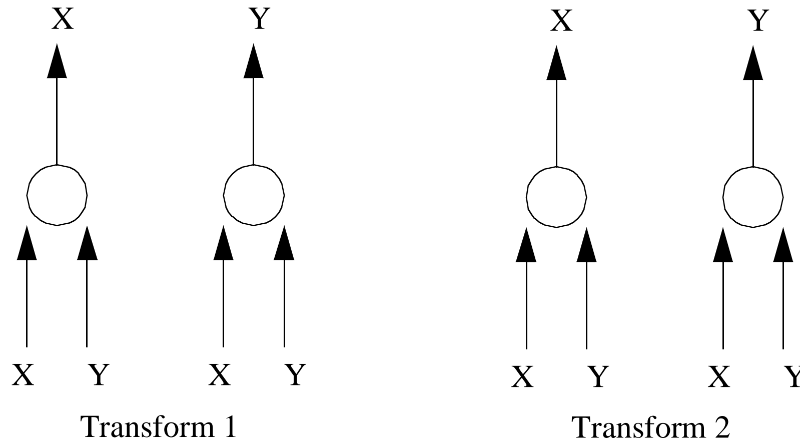


Figure 4.2: The neural network architecture consists of four neurons, each pair acting as a transform.

the network architecture employed, explain the ideas behind the choice of the error function and evaluate its efficacy.

4.2 Architecture

A neural network based IFS can have an arbitrary number of transforms. We have constrained ourselves to two transforms in this implementation, because this is the minimal amount needed to have a rich set of underlying attractors.

Since each transform is a mapping from one X,Y coordinate to another, it must be composed of two scalar functions– one function for each output component, either X or Y. Therefore the recurrent network we employed in our study consists of four neurons, operating as a two transform IFS, where all neurons receive an X,Y coordinate as input and return either the X or Y component of a transform. This is seen in figure 4.2. The function each neuron computes is the standard sigmoid of the weighted sum of the inputs, with a bias term:

$$Out = \frac{1}{1 + e^{-w_3 X + w_2 Y + w_1}}$$

As such each neuron has three modifiable parameters giving a total of twelve parameters

for the whole network.

Since the attractor is an intrinsic property of its IFS, different methods can be used to generate it. In our implementation in order to generate the fractal attractor at a user specified resolution, we initially located a single point on the attractor, by applying one of the transforms on a random point for a number of steps, until it converged. Next, the network was iteratively run on this point, generating new points on the attractor. The process was repeated for all new points, until no new points were found, and the set of points on the attractor was assumed to be complete.

4.3 Learning

The problem we are trying to solve is: given a fractal attractor, find a set of weights for the network which will approximate the attractor. Our approach to this problem consists of finding an error function which will be minimized when the *network coded attractor* is equal to the *desired attractor*.

A common metric or error function used to compare fractal attractors is the Hausdorff distance [4]. The distance is calculated by finding the farthest point on each set relative to the other set and returning the maximum of these two distances. By calculating from both sets in a symmetrical manner, the Hausdorff distance gives a measure of mutual overlap. In other words it will equal zero only when each set is contained within the other, or when they are both equal. The Hausdorff distance between two point sets, A and B, is defined as:

$$H(A,B) = \max\left(h,(A,B),h(B,A)\right)$$

where

$$\begin{aligned} h(A,B) &= \max\left\{\Delta(a,B) \mid a \in A\right\} \\ \Delta(a,B) &= \min\left\{\|b-a\| \mid b \in B\right\} \end{aligned}$$

The Hausdorff distance does not lend itself to a gradient descent approach to minimization because it is not differentiable. In the fractal case, this is due to two reasons. First, the

fractal attractor is generated by an iterative process, which is inherently not differentiable. Second, the Hausdorff distance effectively uses only one point from each set. This point is not constant and its selection may lead to discontinuities.

Our error function borrows principles from the Hausdorff distance and the *collage theorem* [5]. The collage theorem provides the basis for most approaches to the fractal inverse problem or fractal image compression. It states that in order to find an IFS for a given fractal attractor, it is necessary to find a transformation which maps the attractor to itself. As such, our error function will be minimized when the desired attractor and *transformed desired attractor* mutually overlap. Since the network transformations are pseudo contractive due to the sigmoid non-linearity, it follows from the collage theorem that this is equivalent to the network coded attractor being equal to the desired attractor.

As stated previously, there are two issues which need to be addressed to make our error function differentiable: the issue of attractor generation being iterative and the issue of only using one point to calculate the error function. By comparing the desired attractor with the transformed desired attractor instead of the network coded attractor, as per the collage theorem, we can overcome the iterative process issue. This works since, instead of comparing two attractors generated by an iterative process, we will be comparing one attractor with the same attractor acted on by a function, one iteration of the IFS. The second differentiability issue is the number of points used in actually calculating the error function. Our approach is to sum over all points, both on the desired attractor and transformed desired attractor, rather than selecting only the furthest points.

For a given attractor A and a set of transforms T our error function is defined as follows.

$$E(T,A) = \sum_{i} \sum_{x,y \in A} \Delta\left(T_{ix}(x,y), T_{iy}(x,y), A\right) + \sum_{x,y \in A} \Delta\left(x,y,T(A)\right)$$

Where T_{ix} and T_{iy} represent the i-th transform for x and y respectively (as calculated by each neuron) and $T(A)$ is the transformed attractor. The definition for Δ here is:

$$\Delta(x,y,A) = \min\left\{ \|(x,y) - a\|^2 \ \mid a \in A \right\}$$

This error function is similar to the Hausdorff metric in being symmetrical, i.e., taking distances from the desired attractor to the transformed desired attractor as well as distances from the transformed desired attractor to the desired attractor. It is also similar in its use of the Δ function for measuring the distance between a point and a set. There are two advantages to this error function: first, by summing over all the points in the calculation, we get a better measure of the number of points of actual mutual overlap. Second, this error function is practically differentiable with respect to the weights of the transform, allowing its use in our gradient descent approach to minimization.

In examining the error function, it is apparent that it is composed of essentially continuous and differentiable functions. The only part which is not is the min function. In most applications the min function is not continuous, but in this particular case it is. This continuity stems from the continuity of Δ with respect to its parameters. For example, if we were to examine the continuity of Δ with respect to the x variable, we can imagine that for a while the min function picks a certain point on the attractor which gives the minimum distance, and at some value of x the min function switches to another point on the attractor. However, by geometric reasoning we know that while switching to that other point there is a certain intermediate x for which the distances to the first and second points are equal. Thus there is no jump in the value of the min function when switching points, therefore it is continuous.

A similar argument can be presented for y and the weights of the transform. Continuity does not mean differentiability for the min function. At the points where the min function could go to multiple points on the attractor there is no single derivative. Since this is a relatively rare event we have chosen to pick arbitrarily one of the points for the derivative calculation.

Due to the fact that in the first term of E the transform applies to x and y, and in the second term it applies to the set of the attractor, the derivatives must be handled differently for each term. Specifically, for each point in $T(A)$ we must remember the point in A from which it came and which transform was used.

We now calculate the gradient of the function E with respect to the weights of a T_x

transform. The calculation for the other transforms is identical and will not be repeated.

The gradient is given by:

$$\left(\frac{\partial E}{\partial w1}, \frac{\partial E}{\partial w2}, \frac{\partial E}{\partial w3}\right) =$$
$$\frac{\partial E}{\partial \Delta} \cdot \frac{\partial \Delta}{\partial T_x} \cdot \left(\frac{\partial T_x}{\partial w1}, \frac{\partial T_x}{\partial w2}, \frac{\partial T_x}{\partial w3}\right) +$$
$$\frac{\partial E}{\partial \Delta} \cdot \frac{\partial \Delta}{\partial T(A)} \cdot \left(\frac{\partial T(A)}{\partial w1}, \frac{\partial T(A)}{\partial w2}, \frac{\partial T(A)}{\partial w3}\right)$$

The derivative for the transform is the standard one used in back-prop as given by:

$$\left(\frac{\partial T_x}{\partial w1}, \frac{\partial T_x}{\partial w2}, \frac{\partial T_x}{\partial w3}\right) = x_a(1 - x_a)(x, y, 1)$$

Therefore the gradient can be written as

$$\left(\frac{\partial E}{\partial w1}, \frac{\partial E}{\partial w2}, \frac{\partial E}{\partial w3}\right) =$$
$$\sum_{x,y \in A} 2(T_x - x_{\Delta 1})T_x(1 - T_x)(x, y, 1) +$$
$$\sum_{x,y \in A} 2(x - x_{\Delta 2})x_{\Delta 2}(1 - x_{\Delta 2})(x^o_{\Delta 2}, y^o_{\Delta 2}, 1)$$

where T_x refers to $T_x(x,y)$; $x_{\Delta 1}$ is the x chosen by the min function of the Δ function in the first term of E ; $x_{\Delta 2}$ is the x chosen by the min function of the Δ function in the second term of E ; and $x^o_{\Delta 2}, y^o_{\Delta 2}$ are defined by $x_{\Delta 2} = T_x(x^o_{\Delta 2}, y^o_{\Delta 2})$.

Notice that for the second term $x_{\Delta 2}$ may not exist with respect to the mapping from the attractor given by a particular T_x, or be mapped to multiple times. By adjusting the sum respectively, both terms in E even out in magnitude.

4.4 Evaluation and Discussion

The motivation behind the evaluation of the error function was twofold. First, we wanted to demonstrate that fractal attractors are learnable using this error function. Second, we

wanted to assess the domain in which the error function was successful. This was done by applying the algorithm across a diverse set of fractal attractors, and by varying the initial noise conditions. Since it is well known that it is difficult to objectively gauge the difference between images, we chose to use three different metrics to evaluate the results as well as our own eyes.

The error function was tested on a set of 100 fractal attractors. The fractals used were randomly selected from a set of fractals previously generated using a hill-climbing algorithm to locate non-point attractors. For each fractal, the set of weights used to generate the attractor with a certain amount of noise were used as initial conditions for the gradient descent algorithm. The attractors were represented at 16 by 16 pixels. The weight values ranged between -5 to 5. The uniform noise introduced had a variance of 0.25, 0.5, 1.0 and 4.0 across different trials. The gradient descent was weighted using a linear decay function, and consisted of 20 iterations. For each noise level, 10 trials were conducted on each attractor, thus 4000 trials were conducted.

The performance of the algorithm was gauged using the Hausdorff distance, Hamming distance, and similar to the error function described in this chapter, the sum of the point distances between two attractors. These distances to the desired attractor were computed on the initial conditions and on the final conditions after the gradient descent run. On average, 79 out of the 100 attractors tested showed an improvement across all three measures.

In figure 4.3 we see subjective confirmation of the success of the algorithm. In the example runs A, B, C and D, we see that the final network coded attractor is very similar to the desired attractor, in spite of having a different initial network coded attractor. Sample runs E and F show how the algorithm may fail, since we see that the final network coded attractor differs more from the desired attractor than the initial network coded attractor.

It seems that the algorithm performs more impressively with higher initial error. This is seen clearly in figure 4.4, where a histogram of changes in the Hausdorff distance across trials is displayed. We see that for the 4.0 and 1.0 error cases, across most trials, there is a strong negative change in the Hausdorff distance, meaning an improvement. However for the 0.5 and 0.25 error cases, the changes do not seem as significant and unilateral. One obvious

explanation for this effect might be that a larger initial error would bring a greater perceived improvement. However this still does not account for the skew in these graphs, which may be due to the large initial jumps in the gradient descent algorithm and the non-linear nature of the error landscape.

The graphs in figure 4.5 suggest that in many cases the algorithm is driving towards the global minimum, as opposed to local minima. This is seen most clearly in the error function graph, where most of the final network coded attractors have an error near zero, meaning they are close to the desired attractor. Across the other two measures this migration towards zero is present but less obvious. We would expect it be most conspicuous in the error function which is closest to the measure used in the actual gradient descent.

In physically examining the results across the different attractors, it appears that the algorithm performs better on non-overlapping transforms and spatially distinct attractors. This makes sense because there is less ambiguity with respect to the transformation in relation to the self similarity of the attractor.

This algorithm represents the first step in harnessing fractal attractors of recurrent neural networks for computation. There are still many avenues of exploration with this error function. For example, manipulating the error constant decay in ways appropriate to the degree of error, running simulations on different categories of attractors, using ensemble techniques, as well as modifying the error function itself.

These network fractal attractors can be used in different applications. Obviously they can be used to store images. But they can also represent operating ranges and domains. For example an attractor may represent the range of freedom of a joint or the field of vision from a vantage point. The main advantage of using fractal attractors is their inherently very compact coding of visual information. For this reason we believe that this approach warrants further research in the context of neural storage of visual information.

	Desired Attractor	Initial	Final
A			
B			
C			
D			
E			
F			

Figure 4.3: This figure shows some example run results, on 16 by 16 attractors. The first for examples are subjective successes, while the last two leave something to be desired.

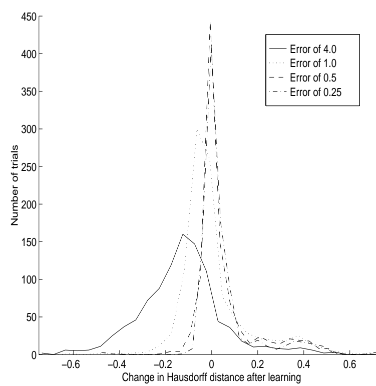


Figure 4.4: This is a histogram of changes in the Hausdorff distance after running the gradient descent algorithm. It was compiled across all 4000 trials. Each of the different graphs represent trials at different initial error levels.

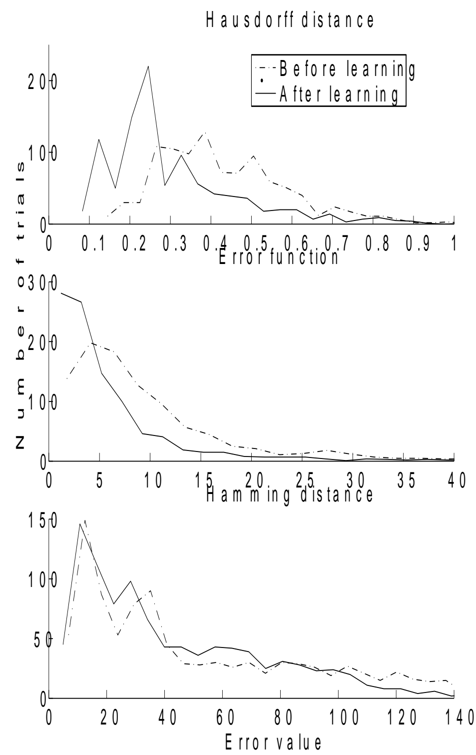


Figure 4.5: Each graph represents a histogram of either the initial or final error across each of the three error measures. These were conducted for an initial error level of 4.0.

Chapter 5

RAAM for Infinite Context-Free Languages

The recurrent neural network in the previous chapter was interpreted as encoding an image in its fractal attractor. In order to extract a network's memory it was iterated repeatedly to generate the points that made up its attractor. The exact order in which these attractor points were generated (the dynamics) was not important to that representation, only what the points were. In this chapter we look at another interpretation of the same network in which both the attractor and the dynamics play a role. In fact this is a new dynamical systems perspective on a representation interpretation of recurrent networks that has been around for over a decade [60], RAAM.

RAAM is a bridge between connectionist and symbolic systems, in its ability to represent variable sized trees in recurrent networks. In the past, due to limitations in our understanding of how dynamics influenced computation in this model, its development plateaued. By examining RAAM from a dynamical systems perspective we overcome most of the problems that previously plagued it. In fact, using a dynamical systems analysis we now prove that not only is RAAM capable of generating parts of a context free language, $a^n b^n$, but is capable of expressing the whole language. Thus demonstrating the inherent capacity of this paradigm.

This material was published in the proceedings of the IJCNN 2000 conference, and formed the basis of a paper in COGSCI 2000.

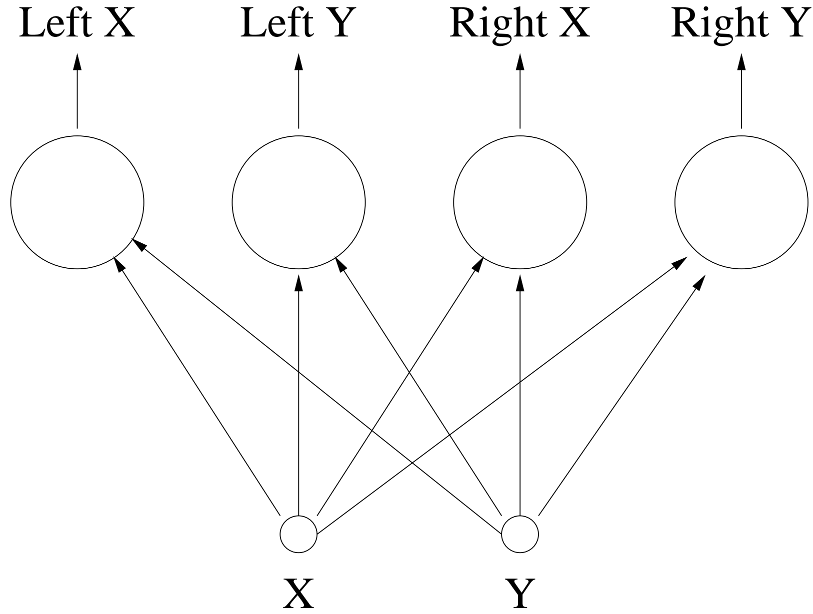


Figure 5.1: An example RAAM decoder that is a 4 neuron network, parameterized by 12 weights. Each application of the decoder converts an (X, Y) coordinate into two new coordinates.

5.1 Introduction to RAAM

A Recursive Auto-Associative Memory or RAAM [60] decoder is a highly recurrent neural network that maps an input into a data-structure, in this case into a binary tree. Thus, from this perspective, each location in the input space encodes for a specific tree.

An example RAAM decoder used in this chapter is shown in figure 5.1, consisting of four neurons that each receive the same (X, Y) input. The output portion of the network is divided into the right and left pairs of neurons. This network is described by the following equations:

$$\begin{aligned}
Left_X &= \frac{1}{1+e^{-(w_{LXX}x+w_{LXY}y+w_{LX})}} \\
Left_Y &= \frac{1}{1+e^{-(w_{LYX}x+w_{LYY}y+w_{LY})}} \\
Right_X &= \frac{1}{1+e^{-(w_{RXX}x+w_{RXY}y+w_{RX})}} \\
Right_Y &= \frac{1}{1+e^{-(w_{RYX}x+w_{RYY}y+w_{RY})}}
\end{aligned}$$

Before using the decoder, the part of the input space that consists of just single node trees is prespecified. This is called the *terminal set* , as it acts to terminate the decoding process, as will be described. If we wish, we can also assign labels to these different leaves, to give them additional meaning. Note the action of deciding whether a point belongs to the terminal set is called the *terminal test.*

The decoding starts with a location in the input space, for this network an (X,Y), two-dimensional coordinate. This initial coordinate represents the root of the tree. What the decoding step does is to find the sub-trees on the left and right branch of this initial root node.

The decoder can be decomposed into two transformations, one for the left branch and one for the right branch. Applying one of these transforms to a coordinate in the space generates a new coordinate. Thus, when both transforms are applied to the initial root coordinate they generate two new coordinates that correspond to the roots of the sub-trees on the left and right branches of the initial coordinate.

It is fairly evident how the process can be made recursive. Each of the these new coordinates is recursively rerun through the network, generating twice as many new coordinates, in order to generate the next level of the tree. This recursion continues, diverging into new branches with each iteration, unless these new coordinates are part of the terminal set. When a branch lands on a point in the terminal set it indicates that we have reached a leaf of the tree, and the recursion is stopped.

Figure 5.2 illustrates this process for a small tree. In the figure the terminal set is shown as the darkened shape. It consists of two parts in two shades of gray, where each part is associated with a different leaf label. The circle in the top portion of the screen indicates the initial root coordinate. Arrows emanating from this circle point to the new coordinates generated by applying the two network transform to this initial coordinate. Note how one of the transforms immediately leads to the terminal set, causing the left branch of this tree to terminate on a leaf with the symbol "1". The other branch continues to a non-terminal part of the space, and thus causes the reapplication of the transforms to this new coordinate in order to generate its branches, as evidenced by the two arrows emanating from this

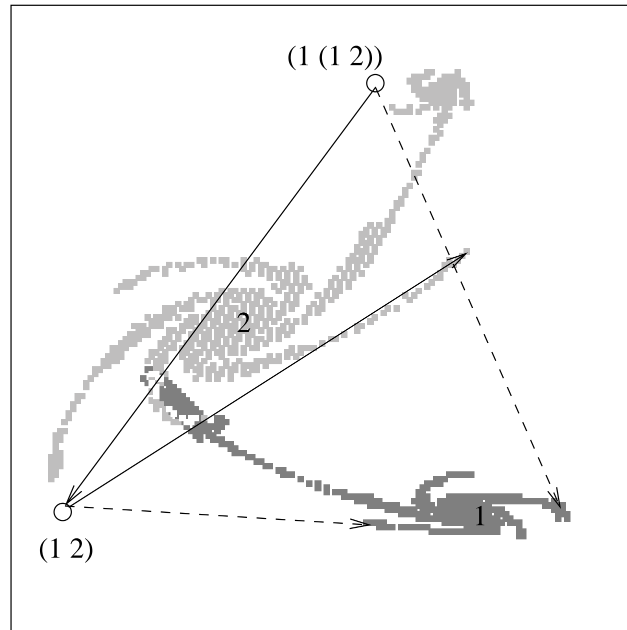


Figure 5.2: The dynamics of a RAAM decoder as it decodes the tree (1 (1 2)) and its daughter tree (1 2). The left transform is shown as a dashed line, and the right transform as a straight line.

coordinate. As both of these new arrows land on parts of the terminal set this signifies the end of the recursion process, and the complete decoding of the tree.

5.2 Problems in RAAM

Although RAAM has found wide use in demonstration and feasibility studies, and in philosophical discussions of what could be done with networks using symbolic representations, our understanding of what RAAM could do and HOW it works has been incomplete. Depending on some factors like the trees themselves and the dimensionality of the network, learning parameters, etc., the system might or might not converge, and even when it converged it might not be reliable. Several studies of sequential RAAM demonstrated that the network could find variable-valued codings which were understandable [10]. Douglas Moreland discovered that under certain conditions sequential RAAM had a very high counting capacity [57] similar to subsequent work by Rodriguez et al [64].

The decoder works in conjunction with a terminal test, which answers whether or not a given representation requires further decoding. The default terminal test merely asks if all elements in a given code are boolean, e.g. above 0.8 or below 0.2. This analog-to-binary conversion was a standard interface in back-propagation research of the late 1980's to calculate binary functions from real valued neurons. However, although it enabled the initial discovery of RAAM training, it led to several basic logical problems which prevented the scaling up of RAAM:

1. The "Infinite Loop" problem is that there are representations which "break" the decoder by never terminating. In other words, some trees appear "infinitely large" simply because their components never pass the terminal test. This behavior breaks computer program implementations or requires depth checking.

2. The "Precision vs. Capacity" problem is that tighter tolerances lead to more decoding errors instead of a greater set of reliable representations.

3. The "Terminating Non-Terminal" problem arises when there is a "fusion" between a non-terminal and a terminal, such that the decoding of an encoded tree terminates abruptly.

Various people have noticed problems in the default terminal test of RAAM and have come up with alternatives, such as simply testing membership in a list, or simultaneously training a "terminal test network" which classifies representations as terminal or nonterminal. [18]. In addition, there are attempts at increasing capacity through modularization and serialization [47; 81].

In the rest of this chapter we present a new formulation of RAAM decoders based on an analysis of the iterated dynamics of decoding, that resolves all these problems completely. This formulation leads to a new "natural terminal test", a natural labeling of terminals, and an inherent higher storage capacity. We then continue the dynamical systems analysis to prove that based on a prototype $a^n b^n$ RAAM decoder generated by hill-climbing, we can generate a competence class of parameterized decoders that not only exclusively generate $a^n b^n$ sequences but in some cases are capable of generating the full infinite language.

5.3 New RAAM Formulation

In the operation of the decoder the output from each pair of neurons is recursively reapplied to the network. Using the RAAM interpretation, each such recursion implies a branching of a node of the binary tree represented by the decoder and initial starting point. However, this same network recurrence can also be evaluated in the context of dynamical systems. This network is a form of *iterated function system* (IFS) [4], consisting of two pseudo-contractive transforms which are iteratively applied to points in a two dimensional space.

In the past we have examined the applicability of the IFS analogy to other interpretations of neural dynamics, as in the previous chapter and in [61; 44]. But unlike these previous cases, in the context of RAAMs the main interesting property of contractive IFSes lies in the trajectories of points in the space– when we take a point and recursively apply the transforms to it (applying both or randomly choosing between them) where will the point eventually end up? For contractive IFSes the space is divided into two sets of points: The first set consists of points located on the underlying attractor (fractal attractor) of the IFS. Points on the attractor never leave it. That is, repeated applications of the transforms to points on the attractor only moves them within the attractor– effectively coercing them to take on orbits within its confines. The second set is the inverse of the first, points that are not on the attractor. The trajectories of points in this second set are characterized by a gravitation towards the attractor. Finite, multiple iterations of the transforms have the effect of bringing the points in this second set arbitrarily close to the attractor.

As noted before, problems 1 and 3 arise from a problematic terminal test. To solve them, there needs to be a clear separation between terminals and non-terminals, such that for any non-terminal starting point a terminal test must eventually "catch" a trajectory delineated by the decoder's dynamics. Since some trajectories never leave the attractor and all others eventually hit the attractor, a terminal test must contain points on the attractor. There is no guarantee that a trajectory on the attractor will hit all parts of the attractor, therefore the only terminal test that guarantees the termination of all trajectories of the RAAM (IFS) is a test that includes all the points of the attractor itself.

By taking the terminal test of the decoder network to be "on the attractor", not only are problems of infinite loops and early termination corrected, but it is now possible to have extremely large sets of trees represented in small fixed-dimensional neural codes. The attractor, being a fractal, can be generated at arbitrary resolution (see the previous chapter or [4] for how the attractor is generated). 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 fractal attractor. For this formulation, the set of trees generated and represented by a specific RAAM are a function of the weights, but are also governed by how the initial condition space is sampled, and by the resolution of the attractor construction. Note that 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 therefore act to "prefix" the trees of the higher dimensional attractors.

Two last pieces complete the new formulation. First, the encoder network, rather than being trained, is constructed directly as the mathematical inverse of the decoder. The terminal set of each leaf of a tree is run through the inverse left or right transforms, and then the resultant sets are intersected and any terminals subtracted. This process is continued from the bottom up until there is an empty set, or we find the set of initial conditions which encode the desired tree.

Second, using the attractor as a terminal test also allows a natural formulation of assigning labels to terminals. Barnsley [4] noted that each point on the attractor is associated with an address which is simply the sequence of indices of the transforms used to arrive on the attractor 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.

5.4 Hill-Climbing an $a^n b^n$ decoder

We used hill-climbing to arrive at a set of RAAM decoder weights for the simple non-regular context-free language $a^n b^n$; that is, the set of strings consisting of a sequence of a's followed

by an equal-length sequence of b's. Two ways to represent the targets for hill-climbing would be either a set of strings in a^nb^n: ab, $aabb$, $aaabbb$, ... , or a set of parenthesized expressions representing binary-branching trees having those strings at their frontiers: (ab), $((a(ab))b)$, $((a((a(ab))b))b)$, ... RAAM is a method for representing structure, and not just strings of symbols. Therefore, we chose the latter, tree-based representation. Specifically, we used trees generated by a simple context free grammar which generates a^nb^n. We were guided by the assumption that this choice would drastically restrict the set of possible solutions to be explored and allow our hill-climbing RAAM to build upon existing structure as it navigated the space of decoder weights.

For the hill-climbing, 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. Starting with 12 random decoder weights, we explored the space of weights by adding random noise to each weight and using the resulting weights to generate trees on a 64-by-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. We used 10 trees representing strings from a^nb^n and $a^{n+1}b^n$ with n = 1, 2, 3, 4, and 5 which had subpart relationships (e.g., the tree for a^2b^2 is a subpart of the a^3b^3 tree) for our learning set.

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^nb^{n+1}, plus additional, ill-formed trees. Though we were able to generate many different weight set solutions to cover the training data, figure 5.3 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 (ab). 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 below, the b attractor points are the white squares on the right side of the image at left.

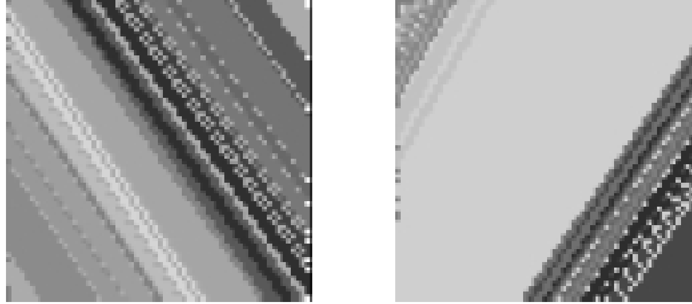


Figure 5.3: The equivalence classes of two solutions to $a^n b^n$ found by hill-climbing.

Beyond the 64-by-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.

5.5 Competence model and proof

We claim that the RAAM evolved by our hill-climbing experiment is indicative of a class of RAAM competence models which generate $a^n b^{n+1}$ and $a^{n+1} b^n$ languages. We justify our claim by demonstrating how an analysis of the specific RAAM dynamics garners the principles to design a parameterized class of$a^n b^{n+1}$ and $a^{n+1} b^n$ RAAMs, some of which, in the infinite case generate the whole languages.

The dynamics of our RAAM can be examined by an arrow diagram. Starting from an initial point, we apply both transformations, and plot arrows to designate the new points thus generated. This process is continued until all new points are on the terminal set.

In figure 5.4 we see a typical example of an arrow diagram from our evolved RAAM. The initial point is marked by a circle. The terminal points on the left side are the 0 terminals and the terminals on the right side are the 1 terminals. The solid lines correspond to the right transform, and the dashed lines correspond to the left transform. By examining the dynamics we can discern a few specific properties:

1. The top right and bottom left corners are both terminals. From any initial starting point, at least one of the transforms goes to a corner terminal.

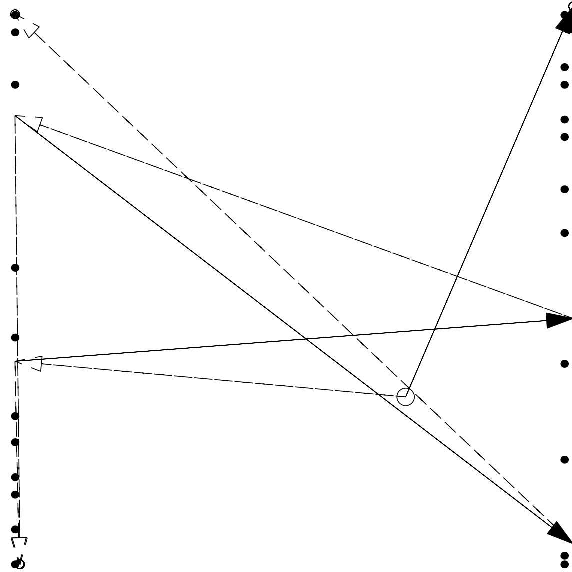


Figure 5.4: An arrow diagram of a tree starting at (0.7,0.3).

2. The left transform always takes us to the left side. The right transform always takes us to the right side.

3. On the left side, the left transform takes us to the bottom left corner terminal. On the right side the right transform takes us to the upper right corner terminal.

The effect of these properties is to generate a very specific variety of trees. By property 1, we see that from any initial point our tree will immediately have one of its branches terminate. By property 2 we see that the continuing branch will hit one of the sides. But by property 3 we see that this branch will also lose one of its sub-branches, only continuing the tree across one branch.

We can characterize this behavior as follows: The tree dynamics consist of a zigzag line which goes between the left side and right side. At each side one of the transforms goes to a terminal, while the other continues the zigzag. This continues until the zigzag hits a terminal on one of the sides.

The effect of these zigzag dynamics is that at each successive tree level we get an alternating 0 or 1. In figure 5.5 we see what such a tree might look like.

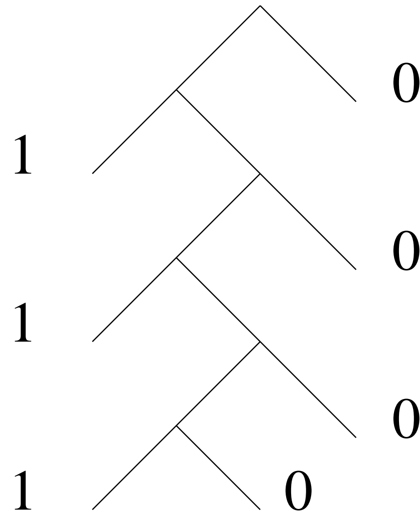


Figure 5.5: A typical tree generated by a zigzag RAAM.

It is apparent that any set of transforms which obey properties 1,2 and 3 will generate trees of this sort. In order to demonstrate the existence of a competence model we need to show that such transforms exist for any resolution, and that they can generate arbitrarily sized trees.

Let $\varepsilon > 0$ be the width of a terminal point. Properties 1 and 2 mandate a parameter dependency on ε . This is due to the transform being composed of a sigmoid function, hence it can never quite reach 0 or 1, but can be made arbitrarily small, so no single transform can fulfill properties 1 and 2 for any ε.

By property 3, in the rectangular region defined by the coordinates $(0,0) \times (\varepsilon, 1)$ the left transform goes to the bottom left corner $(x, y \leq \varepsilon)$ and in the rectangular region $(1-\varepsilon, 0) \times (1, 1)$ the right transform goes to the upper right corner $(x, y \geq 1-\varepsilon)$. Therefore to fulfill property 1 as well we can divide the space into two regions, defined by the line that connects $(\varepsilon, 1)$ and $(1-\varepsilon, 0)$. On the upper half of the line the right transform will go to the upper right corner, and on the bottom half of the line the left transform will go to the bottom left corner. See figure 5.6.

The equations that define this RAAM were shown in the introduction. Since the X-transforms always take us to their respective sides, we can make them constant by setting

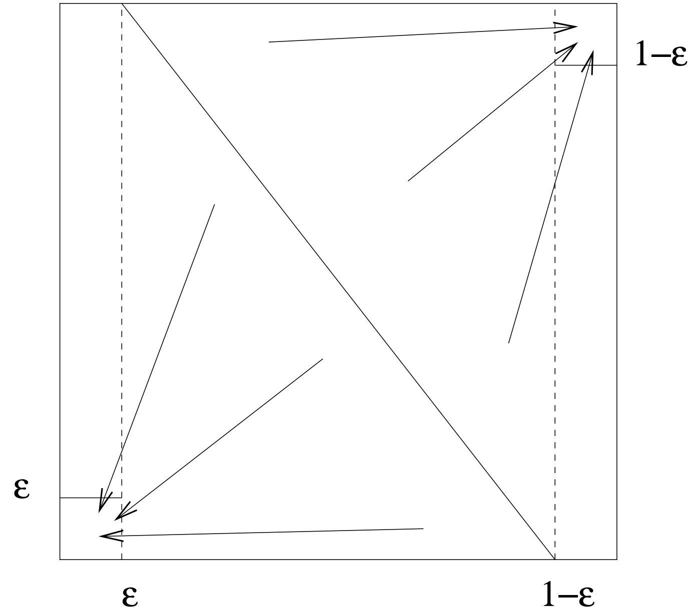


Figure 5.6: The line that divides the space of points that go to the upper right and lower left.

$w_{LXX} = w_{LXY} = w_{RXX} = w_{RXY} = 0$, $w_{LX} = -\log\frac{1-\varepsilon}{\varepsilon}$ and $w_{RX} = -\log\frac{\varepsilon}{1-\varepsilon}$. Thus the left x-transform will always take us to ε and the right x-transform will always take us to $1-\varepsilon$.

The line from $(\varepsilon, 1)$ to $(1-\varepsilon, 0)$ can be parameterized by $cx + c(1-2\varepsilon)y - c(1-\varepsilon) = 0$, where c is a constant. If we set $c > 0$, then we can plug these values directly into our transforms. We need to set $w_{LYX} = w_{RYX} = c$, $w_{LYY} = w_{RYY} = c(1-2\varepsilon)$ and to adjust the constants to $w_{LY} = -c(1-\varepsilon) - \log\frac{1-\varepsilon}{\varepsilon}$ and $w_{RY} = -c(1-\varepsilon)) - \log\frac{\varepsilon}{1-\varepsilon}$. These weights guarantee the right transform takes all points above the line to the upper right terminal and vice-versa for the left transform. These weight are dependent on ε, thus we have demonstrated the existence of weights which obey all three properties for any resolution.

To show the existence of arbitrarily sized trees we need to examine the terminal set locations and the specific dynamics of the zigzag line. In figure 5.7 you see an arrow diagram for the points on the terminal set for a particular setting of c and ε. This diagram shows what points on the attractor go to what other points on the attractor. We see a few interesting characteristics of the terminal set: Initially, we see that out of the two terminal corners we

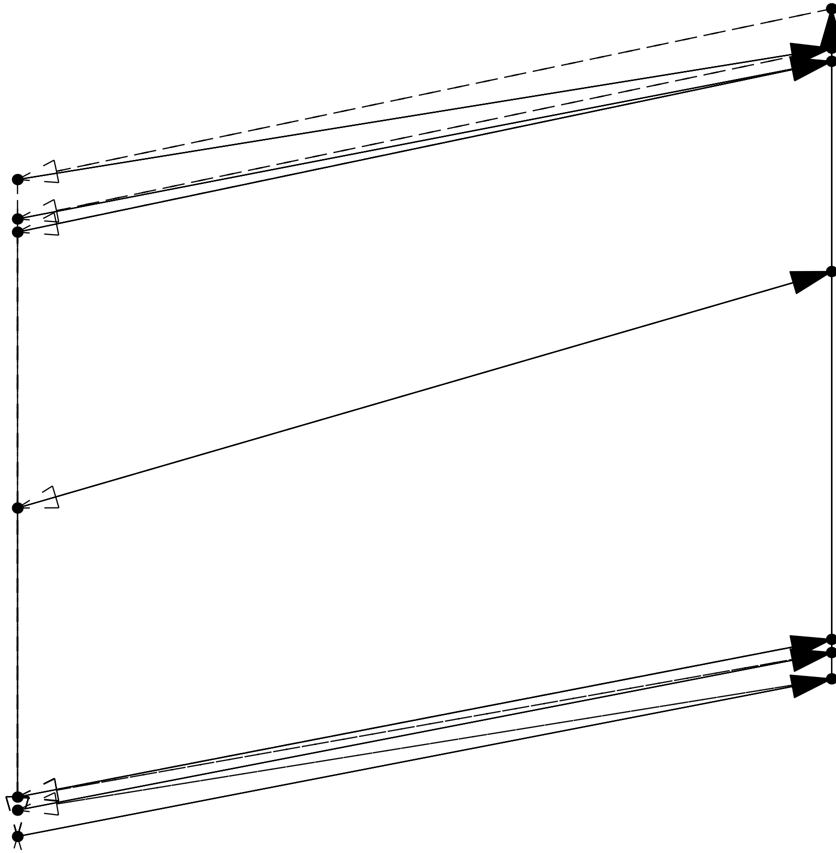


Figure 5.7: A terminal point arrow diagram for $\varepsilon = 1/64$ and $c = 5.703$.

get a zigzag line which seems to generate the terminal set points for the upper and lower part of the space. Both of these zigzags terminate before intersecting. Since these zigzags always go to the corners on their respective sides, they are in fact orbits of different lengths which include a corner. The only other terminals are a pair of points somewhere in between the termination of the two zigzags. These points just go back and forth between themselves.

We can analyze these different properties by treating the zigzag as a one-dimensional map. From every point on the left side, the application of the right and then left transform brings us to a new point on the left side. In figure 5.8 we see two one dimensional maps which both have an ε value of $1/256$ but two values of c, 6.3 and 7. By drawing the line of slope 1 through these maps we see that they have 3 fixed points. Two of these are stable,

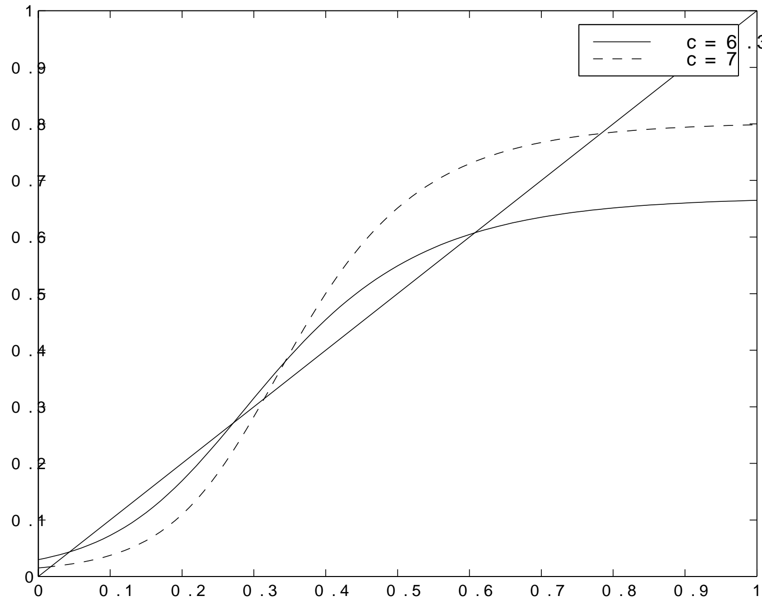


Figure 5.8: A one-dimensional map diagram of the zigzag line for two values of c.

attractive fixed points (this can be shown by linearizing about the fixed points). We can see that the attractor zigzags which emanate out of the corner terminals go towards these fixed points and terminate on them. The third fixed point in the center, is unstable, so zigzag trees above it will head towards the upper half of the space and zigzags below it will head towards the bottom half of the space.

As can be seen in 5.8 the location of this unstable fixed point is dependent on c. Since this dependence is continuous, and between $c = 6.3$ and 7 the location of the fixed point moved by more than $1/256$, then by the mean value theorem we know that there exists a c such that the location of the unstable fixed point is at an integer multiple of ε . Assume that we are discretizing by flooring to the nearest integer multiple of ε, then since the unstable fixed point is at an integer multiple of ε, we can pick an initial tree starting point on the left side, below it, which is arbitrarily close to it. The unstable fixed point represents an orbit which never reaches the other terminal points. If it were not a terminal it would correspond to an infinite tree. By coming from below it we are guaranteed not to hit it. But by coming arbitrarily close to it we can generate arbitrarily long zigzag trees, which hover in its vicinity

for an arbitrary number of iterations before heading towards the attractive fixed point below it. Thus we can create trees of any size, and this model can generate the whole language in an infinite resolution sense.

5.6 Discussion

By examining the underlying dynamics, we have presented a new approach to understanding RAAM. This approach allows us to describe the "natural terminal test" of a RAAM decoder as well as represent trees in accordance with the intrinsic dynamics. The most serious problems of the original RAAM are solved by this new reformulation, allowing a RAAM decoder to provably terminate for any input and potentially generate an infinite grammar.

Specifically, based on initial weights generated by a evolved decoder, we have generalized and proven that there exist a set of 12 weights for a RAAM decoder which not only exclusively generate words from the $a^{n(+1)}b^{n(+1)}$ language, but are capable of generating the whole language. In fact in the region of the unstable fixed point, the RAAM exhibits a monotonic behavior. As we approach the fixed point, the size of the generated trees increases divergently. The significance of this is not only in the context of RAAM but in the context of connectionist processing in general, since we have demonstrated how a monotonically increasing continuously varying input (initial starting point) can incrementally generate the members of a context free grammar, within the confines of a neural substrate. Thus we have shown how a smooth mapping could exist between the tonic varying outputs of a single neuron and the generativity of a context-free grammar.

With this reformulation of RAAM, we have a new and deeper connection between connectionist and symbolic representational and generative theories. In the future we expect to examine the fuller potential of this model, including the expressiveness of other grammars and modes of lexical assignment, as well as more informed learning methods.

Chapter 6

Discussion

In this thesis we have explored both feed-forward and recurrent neural networks from the perspective of understanding how they represent information. In this section we discuss some of the consequences of this work, while exploring questions and future work that it suggests.

The DIBA algorithm for the first time allowed us a complete glimpse into the computation performed by an MLP. Using DIBA, we explored questions about the computational complexity of representation extraction from MLPs in general. Concurrently, we examined example networks to understand the decision region structure of good and bad generalization, as well as the sources and form of noise in networks. Thus, the main goal of using the DIBA algorithm to further our understanding of computation and representation in feed-forward networks has been achieved. In so doing, issues about generalization and learning were brought to light that warrant further exploration. It was shown that a great deal of combinatoric complexity, in the form of face-sharing, exists as an intrinsic part of these models. Learning experiments suggested that most of this complexity is not exploited by typical learning algorithms like back propagation. Rather, this complexity typically manifests itself as undesired artifactual decision regions. Can this complexity be used? This question has both a conceptual and practical component to it. From a practical perspective, assuming that we have a particular decision region structure in mind, the question becomes, how can a learning/construction algorithm be made that can efficiently navigate through the large quantity of combinatoric options available to it. The conceptual question may be

more difficult, but probably more interesting. Face-sharing is a generalization mechanism, a way to express a pattern using the intersections of boundaries; the conceptual question is, when is this a useful generalization strategy? What kind of problem invariances could be captured effectively by face-sharing? And then, how would we detect that training data suggests the use of face-sharing for generalization?

DRCA is a new method to analyze decision region based classification models. The DRCA method focuses on the "relevant complexity" of a model as it pertains to the decision region's relationships that a model imposes on its training data. As such, the potentially exponential complexity of models and perceptual limitations in high-dimensional visualization–two large obstacles imposed by the curse of dimensionality on model analysis–are uniquely and directly addressed. In a field with an overwhelming number of models, with very few relevant ways to choose between them, this method is not only a real tool to look inside their workings, but it acts as a common denominator. Irrespective of a model's functional form, if it generalizes using decision regions its classification strategy can be compared with any other model that uses decision regions. There is a particularly urgent need for such comparability, as models, rather than becoming more comprehensible, are going in the other direction quickly. This is evidenced by a current trend in the state of the art, the migration towards combined models, or *ensemble techniques*. Techniques such as bagging [12] and boosting [26] build models which can be the combination of hundreds of other models. These combined models consist of thousands to millions of parameters, making them impervious to any decompositional analysis technique. However, as these models are almost linear compositions of decision region type models, there is no reason to think that these models do anything more than generalize using decision regions. Thus, DRCA naturally complements the current trends in model development, and allows extremely complex models to be analyzed with the same facility as a decision tree. This has applications to Data Mining, a field that because it mandates an explanatory ability for its models has been forced to adhere to simple models, until now.

In truth, the fact that most of these models have similar underlying functionality, coupled with the clear transparent understanding that this method offers, allows us to ask a

more interesting question aside from ranking models. We can now examine the benchmark datasets and in an unequivocal manner analyze the generalization strategies that seem to succeed on the data. The importance of this is that it would allow us to better understand the actual problems that are motivating some of the machine learning field. Namely, to get a better handle on what the difficult problems really are, the problems where significantly different and hopefully interesting approaches may be needed for their solution.

Another interesting avenue that is opened up by this work is the ellipsoid model construction method, a method that demonstrated how a new model may be constructed based on a connectivity graph generated in another context. The ellipsoid model uses the connectivity graph as a blueprint for how to enclose data points. Thus, it acts to abstract away the model construction problem, to the problem of building a graph, connecting the data points (not addressing the margin issue). This raises many questions for exploration, such as, can we hand construct connectivity graphs with specific properties directly from the data, or can they be constructed by synthesizing multiple models?

Neural fractal memory is an exciting perspective on how information can be encoded in recurrent networks. The work presented demonstrated that it is a viable model, a model that despite its roots in chaos is learnable by continuous parameter modification, in the form of a gradient descent error function. This is really a novel perspective on how recurrent neural networks can compute and learn, as such, rather than filling voids in existing approaches to recurrent neural computation, it opens the doors to research in this uncharted territory. As with any new computational paradigm there are two parallel courses of research. The first is to explore the theoretical boundaries of the paradigm. Possible theoretical questions might ask, how do the number of transforms, their activation function, and dimensionality, impact what the network can represent and learn. The second course of research is applied, to try and apply this approach to real problems which may benefit from this representation. For instance, it is well suited to a problem requiring a more “holistic” visual approach, such as object recognition. This application would probably be a hybrid, including a neural fractal memory component as a “holistic” memory/feature detector.

RAAM is an interesting model, in that it conceptually demonstrates how generative

grammatical information can be encoded in a recurrent network. The competence model proof showing that a RAAM can encode for an entire infinite context-free language was important as it accentuated underlying principles of how recurrent dynamics can be harnessed for grammatical generativity. This work is in line with the similar questions posed about grammar representation in other recurrent models [84; 64; 56]. Thus it is part of a research trend trying to understand how recurrent neural computation can be coupled with grammar and language processing. In that vein, a natural continuation to this work is to find RAAM models that suggestively code for other interesting languages (as we did for the hill-climbed RAAMs used in the analysis) and to analyze them to find other components of dynamical mechanisms that allow this grammatical coding. The loftier goal would be to not only enumerate these different mechanisms, but to find the underlying meta-principles that may allow us to generalize this topic and may lead to efficient learning in these models.

Bibliography

[1] L.F. Abbott. Decoding neuronal firing and modelling neural networks. *Quarterly Review of Biophysics*, 27:291–331, 1994.

[2] R. Andrews, R. Cable, J. Diederich, S. Geva, M. Golea, R. Hayward, C. Ho-Stuart, and A.B. Tickle. An evaluation and comparison of techniques for extracting and refining rules from artifical neural networks. *Knowledge-Based Systems Journal*, 8(6), 1995.

[3] R. Andrews and S. Geva. Rules and local function networks. In *Proceedings of the Rule Extraction from Trained Artificial Networks Workshop*, 1996.

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

[5] M.F. Barnsley and L.P. Hurd. *Fractal Image Compression.* AK Peters, Wellesley, 1992.

[6] R. Bellman. *Adaptive Control Processes.* Princeton, 1961.

[7] S. Bespamyatnikh and M. Segal. Covering a set of points by two axis-parrallel boxes. In *Proc. 9th. Canad. Conf. Comput. Geom.*, pages 33–38, 1997.

[8] C.M. Bishop. *Neural Networks for Pattern Recognition.* Oxford University Press, 1995.

[9] C.L. Blake and C.J. Merz. UCI repository of machine learning databases, 1998.

[10] D.S. Blank, L.A. Meeden, and J.B. Marshall. Exploring the symbolic/subsymbolic continuum: A case study of raam. Technical Report TR332, Computer Science Department, University of Indiana, 1991.

[11] G. Bologna. Rule extraction from a multi layer perceptron with staircase activation functions. In *IJCNN 2000*, 2000.

[12] L. Breiman. Bagging predictors. *Machine Learning*, 26(2):123–140, 1996.

[13] D.S. Broomhead and D. Lowe. Multivariable functional interpolation and adaptive networks. *Complex Systems*, 2:321–355, 1988.

[14] I. Bruss and A. Frick. Fast interactive 3-d graph visualization. In *Proceedings of Graph Drawing '95*, pages 99–110. Springer-Verlag, 1995.

[15] J.Y. Campbell, W. Andrew, and A.C. Mackinlay. *The Econometrics of Financial Markets*. Princeton University Press, 1997.

[16] M. Casey. The dynamics of discrete-time computation, with application to recurrent neural networks and finite state machine extraction. *Neural Computation*, 8(6), 1996.

[17] N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2:113–124, 1956.

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

[19] T.M. Cover. Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition. *IEEE Transactions on Electronic Computers*, EC-14:326–334, 1965.

[20] M.W. Craven and J.W. Shavlik. Extracting comprehensible concept representations from trained neural networks. In *IJCAI 95 Workshop on Comprehensibilty in Machine Learning*, 1995.

[21] D. DeMers and G. Cottrell. Non-linear dimensionality reduction. In *Advances in Neural Information Processing Systems*. Morgan Kaufmann, 1993.

[22] D.H. Deterding. *Speaker Normalisation for Automatic Speech Recognition*. PhD thesis, University of Cmabridge, 1989.

[23] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer-Verlag, 1987.

[24] B.S. Everitt. *Graphical Techniques for Multivariate Data.* Heinemann Educational Books, London, 1978.

[25] Y. Fisher. *Fractal Image Compression.* Springer-Verlag, New York, 1994.

[26] Y. Freund and R. Schapire. Experiments with a new boosting algorithm. In *Machine Learning: Proceedings of the Thirteenth International Conference*, pages 148–156, 1996.

[27] K. Funahashi. On the approximate realization of continuous mappings by neural networks. *Neural Networks*, 2:183–192, 1989.

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

[29] M. Golea. On the complexity of rule-extraction from neural networks and network-querying. In *Proceedings of the Rule Extraction from Trained Artificial Neural Networks Workshop, AISB'96*, 1996.

[30] R.P. Gorman and T.J. Sejnowski. Analysis of hidden units in a layered network trained to classify sonar targets. *Neural Networks*, 1:75–89, 1988.

[31] S. Haykin. *Neural Networks: A Comprehensive Foundation.* IEEE Press/Macmillan College Publishing, 1994.

[32] G.E. Hinton, P. Dayan, B.J Frey, and R.M. Neal. The wake-sleep algorithm for unsupervised neural networks. *Science*, 1995.

[33] J.J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences of the U.S.A.*, 81:2554–2558, 1982.

[34] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2:359–366, 1989.

[35] W.Y. Huang and R.P. Lippmann. Neural net and traditional classifiers. In D.Z. Anderson, editor, *Neural Information Processing Systems*, 1988.

[36] N. Intrator and O. Intrator. Interpreting neural-network models. In *Proceedings of the 10th Israeli Conference on AICV*, pages 257–264. Elsevier, 1993.

[37] O. Intrator and N. Intrator. Robust interpretation of neural-network models. In *Proceedings of the VI International Workshop on Artificial Intelligence and Statistics*, 1997.

[38] A. Jacquin. A novel fractal block-coding technique for digital images. In *IEEE ICASSP Proc. 4*, pages 2225–2228. IEEE, 1990.

[39] T. Joachims. Making large-scale svm learning practical. In B. Schelkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods - Support Vector Learning*. MIT Press, 1999.

[40] I.T. Jolliffe. *Principal Component Analysis*. Springer-Verlag, 1986.

[41] E.R. Kandel, J.H. Schwartz, and T.M. Jessel, editors. *Principles of Neural Science*. McGraw-Hill Professional Publishing, 2000.

[42] L. Khachiyan. Complexity of polytope volume computation. In *New Trends in Discrete and Computational Geometry*, chapter 4. Springer-Verlag, 1993.

[43] R.D. King, C. Feng, and A. Shutherland. Statlog: comparison of classification algorithms on large real-world problems. *Applied Artificial Intelligence*, 9(3):259–287, May/June 1995.

[44] J.F. Kolen. *Exploring the Computational Capabilities of Recurrent Neural Networks*. PhD thesis, Ohio State, 1994.

[45] J.F. Kolen and J.B. Pollack. The observer's paradox: Apparent computational complexity in physical systems. *The Journal of Experimental and Theoretical Artificial Intelligence*, summer, 1994.

[46] W.J. Krzanowski. *Principles of Multivariate Analysis: A User's Perspective*. Clarendon Press, Oxford, 1988.

[47] S.C. Kwasny, B.L. Kalman, and A. Abella. Decomposing input patterns to facilitate training. In *Proceedings of the World Congress on Neural Networks*, volume 3, pages 503–506, Portland, Oregon, 1993.

[48] Y. Lecun, B. Boser, J.S. Denker, D. Henderson, R.E. Howard, W. Hubbard, and L.D. Jackel. Handwritten digit recognition with a back-propagation network. In *Advances in Neural Information Processing Systems*, volume 2, pages 598–605. Morgan Kaufmann, 1990.

[49] A. Linden. Iterative inversion of neural networks and its applications. In *Handbook of Neural Computation*. Oxford University Press, 1997.

[50] L. Lovasz. *An Algorithmic Theory of Numbers, Graphs and Convexity*. Capital City Press, Vermont, 1986.

[51] F. Maire. Rule-extraction by backpropagation of polyhedra. *Neural Networks*, In Press.

[52] J. Makhoul. *Linear Prediction in Automatic Speech Recognition*, chapter 2. Academic Press Inc., New York, New York, 1975.

[53] B.F.J. Manly. *Multivariate Statistical Methods: A Primer*. Chapman and Hall, London, 1994.

[54] W.S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[55] M. Minsky and S. Papert. *Perceptrons*. MIT Press, 1969.

[56] C. Moore. Dynamical recognizers: Real-time language recognition by analog computers. *Theoretical Computer Science*, July 1998.

[57] D.D. Moreland. An investigation of input encodings for recursive auto associative neural networks. Master's thesis, Ohio-State, 1989.

[58] D.W. Opitz. and J.W. Shavlik. Dynamically adding symbolically meaningful nodes to knowledge-based neural networks. *Knowledge-Based Systems*, 8(6):301–311, 1995.

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

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

[61] J.B. Pollack. The induction of dynamical recognizers. *Machine Learning*, 7:227–252, 1991.

[62] L.Y. Pratt and A.N. Christensen. Relaxing the hyperplane assumption in the analysis and modification of back-propagation neural networks. In *Cybernetics and Systems 94*, pages 1711–1718. World Scientific, 1994.

[63] D.L. Reily, L.N. Cooper, and C Elbaum. A neural model for category learning. *Biological Cybernetics*, 45:35–41, 1982.

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

[65] P.J. Rousseeuw. *Handbook of Statistics*, volume 15, chapter Introduction to Positive-Breakdown Methods, pages 101–121. Elsevier, Amsterdam, 1997.

[66] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. *Learning Internal Representations by Error Propagation*, volume 1, chapter 8. MIT Press, Cambride, MA, 1986.

[67] D.E. Rumelhart and J.L. McClelland, editors. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1. MIT Press, 1986.

[68] S.J. Russel and P. Norviq. *Artificial Intelligence: A modern approach*. Prentice Hall, 1994.

[69] A. Sabharwal and L.C. Potter. Set estimation via ellipsoidal approximation. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, May 1995.

[70] D. Sanger. Contribution analysis: A technique for assigning responsibilities to hidden units in connectionist networks. *Connection Science*, 1:115–138, 1989.

[71] A. Schrijver. *Theory of Linear and Integer Programming*. Wiley-Interscience, 1987.

[72] R. Setino. Extracting rules from neural networks by pruning and hidden-unit splitting. *Neural Computation*, 9(1):205–225, 1997.

[73] N.E. Sharkey and A.J.C. Sharkey. Adaptive generalization. *Artificial Intelligence Review*, 7:313–328, 1993.

[74] R. Shonkwiler, F. Mendivil, and A. Deliu. Genetic algorithms for the 1-d fractal inverse problem. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, San Diego, 1991.

[75] N.Z. Shor and Berezovski. New algorithms for constructing optimal circumscribed and inscribed ellipsoids. *Optimization Methods and Software*, 1:283–299, 1992.

[76] T.R. Shultz, Y. Oshina-Takane, and Y. Takane. Analysis of unstandardized contributions in cross connected networks. In *Advances in Neural Information Processing Systems 7*. MIT Press, 1995.

[77] J.P. Siebert. Vehicle recognition using rule based methods. Technical report, Turing Institute, March 1987.

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

[79] I. Singer. *Abstract Convex Analysis*. Wiley-Interscience, 1997.

[80] P. Smolensky. On the proper treatment of connectionism. *Behavioral and Brain Sciences*, 11:1–74, 1988.

[81] A. Sperduti and A. Starita. An example of neural code: Neural trees implemented by lraam. In *International Conference on Neural Networks and Genetic Algorithms*, pages 33–39, Innsbruck, 1993.

[82] S.H. Strogatz. *Nonlinear Dynamics and Chaos*. Addison-Wesley, 1994.

[83] D.J. Stucki and J.B. Pollack. Fractal (reconstructive analogue) memory. In *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*, pages 118–123. Cognitive Science Society, 1992.

[84] W. Tabor. Fractal encoding of context free grammars in connectionist networks. *Expert Systems: The International Journal of Knowledge Engineering and Neural Networks*, in press.

[85] C. Thornton. Separability is a learner's best friend. In J.A. Bullinaria, D.W. Glasspool, and G. Houghton, editors, *Proceedings of the Fourth Neural Computation and Psychology Workshop: Connectionist Representations*, pages 40–47. Springer-Verlag, 1997.

[86] S. Thrun. Extracting provably correct rules from artificial neural networks. Technical report, University of Bonn, 1993.

[87] P. Tino and G. Dorffner. Predicting the future of discrete sequences from fractal representations of the past. *Machine Learning*, 2000. in print.

[88] D.M. Titterington. Estimation of correlation coefficients by ellipsoidal timming. *Appl. Statist.*, 27(3):227–234, 1978.

[89] F.S. Tsung and G.W. Cottrell. Phase-space learning for recurrent networks. Technical Report CS93-285, Dept. Computer Science and Engineering, University of California, San-Diego, 1993.

[90] V.N. Vapnik. *The Nature of Statistical Learning Theory.* Springer, 1995.

[91] E.R. Vrscay and C. J. Roehrig. *Iterated Function Systems and the Inverse Problem of Fractal Construction Using Moments*, pages 250–259. Springer-Verlag, New York, 1989.

[92] I.H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations.* Morgan Kaufmann, 1999.

[93] B. Zhu. Approximating the convex hull polyhedra with axis-parallel boxes. *International J. Comput. Geom. Appl.*, 7:253–267, 1997.