An Empirical Investigation of Biologically Plausible Hebbian Systems

A Thesis
Presented for the
Master of Science Degree
The University of Tennessee, Knoxville

George Shen

May 1996

Acknowledgments

I would like to thank Dr. Bruce MacLennan for the enthusiasm that he brings to the subject of neural networks and for helping me find a topic that combines the biological and computer sciences. Without his guidance and feedback during this project, completion would have been an impossibility. Sincere appreciation is also extended to Dr. David Straight and Dr. Jens Gregor for their insights, comments, and critiques; they kept me on my toes and made this project more successful than it would have been.

My deepest gratitude goes to Hanh Nguyen who has encouraged me when I needed encouragement, who has given me the time and space to finish my work, and who has been with me, never faltering in her support, through all the highs and lows of life. This paper is for my better half.

I would also like to thank my parents for their encouragement and strong backing, and for believing in my abilities. And of course, Jenny, my fellow compatriot who has understood me and helped me without expecting anything back, who knew exactly what I was going through, and gave me reason to hope and believe: I cherish your sharp wit, your sincere advice, and your unwavering support. I owe my sanity to you.

Abstract

This paper presents an investigation into three algorithms that have pattern matching and learning capabilities. There are many neural network models in use today that are computationally viable for numerous tasks. These models tend to be inspired from neuroscience; this is evident from some of the terminology used such as threshold levels. However, many of these models incorporate complex algorithms for which there is no biological basis; in addition, these models tend to be multi-pass systems and are too slow for biological implementation. Backpropagation is one such example. The algorithms discussed in this paper share the common property of being good candidates for neural implementation; they are one-pass systems incorporating variations of a learning rule that is known to exist in real neural networks. We will verify the maximal performance of these three systems. Of more importance, however, are their capabilities under biologically constrained conditions; that is, what is the system performance (1) if learning begins with non-optimal parameters, (2) if the system is only capable of low-precision mathematics, and (3) if the learning results are corrupted. These biologically motivated issues will be the focus of the investigation.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

For as long as there has been recorded history, humans have been problem solvers. Whether it was to figure out how to lure prey into the trap or how to build flying vehicles, our capacity to handle odd and interesting situations in productive ways is one of our more important traits. This ability is centered around the central nervous system and more specifically, the brain; it is a well known fact that removal of the brain renders a person incapable of solving even the simplest problems.

Since the task of a computer is to solve problems, does this imply that the brain is the first computer model and we are the prototypical robots? Intuitively, yes. But more precisely, both manufactured computers and the brain perform computations. So then what is computation? There has been much work and progress on the theory of computation from Turing(1937), von Neumann(1951), and others. The definition that we will use is the following:

> Computation is the physical realization of an abstract process for the purpose of manipulating abstract objects[18].

The realization is a homomorphism[1] from the physical process to the abstract process. Figure 1.1 is a schematic diagram of a homomorphism, $H : P \rightarrow A$, that maps the physical properties of P into the abstract properties of A[18]. Examples of abstract objects that are manipulated include numbers, sets, and even formulas.

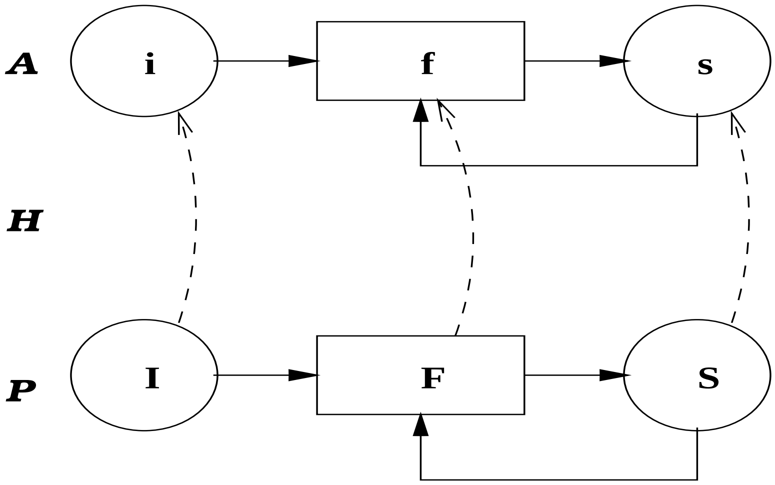

Figure 1.1: Physical Realization of an Abstract Process

From the definition, it is easy to see that a modern von Neumann based computer performs computations. Similarly, the brain also performs computations. For example, addition is an abstract process that manipulates abstract objects. Numbers are the input, i, and the transition function, f, determines the new state, s, in which the numbers have been added. The physical process in a computer involves the movement of electrons, I, through the circuitry of the CPU, F, until a state, S, that holds the sum total is reached. In the brain, it is the movement of ions, I, across synapses and electrical pulses through the dendritic nets, F, that lead to a new state, S, when we know the answer.[2]

[1]Homomorphism is defined to be a mapping of a system that preserves the basic operations and the basic relations.

[2]It is easy to get into a discussion of how ion movement has anything to do with what we know as the process of addition. In other words, how does low-level ionic movements across cell membranes lead to high level concepts such as addition? We do not know exactly how, but we do know the effect. This will be discussed briefly in the next chapter to support the use of biologically plausible algorithms.

1.1 Computers as Pure Problem Solvers

One way to view a computer is from a task-oriented perspective. Given a problem (i.e. some input), a computer cranks out the solution (i.e. the output). Humans have always invented tools to make life easier, and the computer as a tool is no different. For diehard members of this school of thought, a computer is a machine with no innate intelligence similar to its precursors such as the abacus and the slide rule. Speed and accuracy of computation is most important. It does not matter if the computer computes in the same way as the brain, as long as the task is successfully and efficiently accomplished.

1.2 Computers as Artificial Intelligence

Another way to look at the computer is to view it as a way to duplicate or mimic the way humans think. The computer can still find solutions to problems, but it no longer is just a number crunching machine. Turing officially posed the question "Can machines think?" back in 1950. In addition, he proposed a black box experiment to test if a machine is actually thinking. Thus the field of Artificial Intelligence (AI) was born [3].

There is a lot of debate as to whether we can ever develop computers into autonomous thinking machines and if it is a good idea to attempt this; for now, we will set aside the ethical questions raised by the potential for sentient machines since, if it is at all possible, then that time has not yet come. In addition, there is also the question of whether such a computer is intelligent or simply appears intelligent, and does it matter? The definition of AI and intelligence is not yet fixed in the Computer Science

[3] It is debatable when AI actually began. Some think it was actually 1956 with the Dartmouth conference. And of course man has always contemplated the possibility of a thinking machine even in ancient times. The *Illiad* describes Hephaestus, god of fire, forging workers with intelligence and speech.

community nor in the community at-large.

1.3 AI and Connectionism

AI research has been dominated by such topics as heuristic search and reasoning. Traditional AI takes a symbolic approach; it is very good at manipulating discrete structures such as graphs, mathematical and logical formulas. Through this approach, we have smart chess playing machines, expert systems, and theorem provers.

Systems based on traditional AI accomplish their tasks accurately and efficiently; they tend to give definite, reliable, and reproducible results. But they are also rigid, usually not able to adapt to changing environments or new, unforeseen inputs. Processes such as perception, recognition, and association do not fare well with the discrete symbolic approach[16].

More recently, a connectionist approach has emerged as an alternative to the traditional symbolic approach. Also known as neural networks, the connectionist architecture is composed of simple processors, known as nodes, that are massively interconnected. This is in contrast with the basic von Neumann machines which have sophisticated CPUs and execute commands sequentially.

Connectionist systems tend to be flexible, robust, adaptive, and responsive[14] because they are based on the notion of the continuous rather than the discrete. Errors usually have a small effect on the system and the system responds gradually rather than in steps. In cases where time is of the essence, a partially finished computation in a connectionist system may be viable whereas in a traditional system, computation must be complete before there are any usable results.

1.4 Traditional Neural Network Models

The design of artificial neural networks (ANN's) is loosely based on biological neural systems. Numerous biological details have been removed from most of the popular models and many features not found in nature have been added to simplify and render the nets more effective. As a result, the similarity of ANN's to their biological counterparts is only superficial.

There are various types of ANN's. One ANN that has found many applications and has become the standard in the neural network community is Backpropagation (BP). A few of the difficult and diverse problems BP has successfully solved include image compression and reconstruction, time-series prediction, and function approximation. Some of the varied applications of BP include medical diagnosis, recognition of hand-written ZIP codes, and autonomous vehicle navigation. Backpropagation is typically composed of several layers of nodes with each layer being fully connected to its adjacent layers. Each connection has a weight, w_{ij}, which is the connection strength between nodes i and j. The basic algorithm for BP is intuitive and appealing: If the network gives the wrong answer, then the error is propagated back through the net and the weights are corrected via the gradient descent delta rule to lessen the error upon the next iteration.

Traditional neural networks that are in use tend to be very powerful and are capable of solving a wide range of problems. They have all the characteristics of a connectionist system. They are able to extrapolate from the given test patterns what the result from a new pattern should be. Node failures are not catastrophic and do not result in a complete shut-down of the system. ANN's tend to be very fault-tolerant; the network recovers and compensates for the failure and continues to function, though perhaps at a reduced level. This propensity to smoothly handle failure is known as graceful degradation.

Each type of network has its own unique problems. One of the drawbacks common to most neural networks is the convergence time. It is well-known, for example, that Backpropagation is very slow [4, 12]; training sessions can require many iterations of the dataset even for relatively simple problems. BP also has the additional disadvantage of spurious solutions; instead of finding the correct answer, the system may actually converge onto a less than optimal response, commonly referred to as a local minimum.

One of the approaches to creating intelligent systems[4] is to model the brain as closely as possible but the standard neural networks in use today are only inspired by biological neural networks and not faithful models of biologic phenomena. For example, current neural net models tend to view the change in state of a neuron as a discrete process, but there is evidence that continuous electrochemical waves have an important effect on the neurons [12]. In addition, there has not been any evidence to date that error differencing and back-error propagation[5] occur in natural systems[4]. In fact, what we do know about the brain is that it uses what is called Hebb's rule, which from a computational point of view is considered to be too weak for many applications. The task then is to determine models that effectively incorporate Hebb's rule if our ultimate goal is to create a system that functions in a fashion similar to the brain[6]. The next chapter discusses in more detail Hebbian learning.

One final point is that these networks tend to be more complex due to their nonlinear nature. Observations of real neuronal networks have shown that much of the processing in the brain is highly nonlinear due to the inherent biophysics of the membrane. Some have even argued that current artificial models are too simplistic and only elucidate limited aspects of biological systems[22]; the implication is that to be faithful to its biological roots, ANN's should incorporate as many biological phenomena as

[4]Here, we define a system to be intelligent if it can perform tasks that humans currently do better.

[5]Backpropagation incorporates gradient descent into its algorithm but it is *not* gradient descent. This distinction will be important later on.

[6]This does not rule out other models as being potential intelligent systems. We do not argue that BP will never yield a truly intelligent system like the brain; perhaps some future variant of BP will lead to consciousness. What we do argue here is that the brain is the most intelligent computer in existence and it would be wrong to ignore models that are more faithfully based on it.

possible. Since biological neural processing is essentially nonlinear, it can be argued that to model the brain, it would be more accurate to keep the nonlinearity in the ANN's.

However, it has been shown that neural processes in dendritic nets can be modelled with a linear system[12, 19] and in many instances, sigmoid nonlinearities are not appropriate. In addition, universal nonlinear behavior can be simulated by purely linear systems[13]. Therefore, the nonlinear nature of ANN's is not necessary to model biological systems and only adds to the complexity of the network. It may seem incongruous that here we argue to be less faithful to the biophysics of neurology while in the previous paragraph we said Hebbian learning was more appropriate because it was more realistic. There is no contradiction, however, if we keep in mind that linear systems can still be biologically faithful. Linearity will be examined in greater depth in the next chapter.

The goal of this paper is to discuss and implement three biologically plausible learning algorithms and, from the results of experimentation, determine the positive aspects and drawbacks of each.

Chapter 2

The Learning Algorithms

Why should these algorithms be investigated? That will be addressed in this chapter. The algorithms will then be described in more detail.

2.1 The Emergence of the Discrete From the Continuous

There is new impetus for us to investigate algorithms that can possibly be found in a biological neural system. In the brain, "discrete and symbolic processes emerge from underlying continuous subsymbolic processes" [18]. By their very nature, most basic cognitive processes such as perception and association are continuous[12] and it is from this subsymbolic substrate that the symbolic emerges.

Figure 2.1 represents the emergence of each layer from the layer below. The very low-level neural processes include synaptic and dendritic activities like ion flux, transmitter release, and current propagation; this gives rise to low-level cognition such as perception and association. Finally, the high-level processes such as language processing and theorem proving derive from the low-level continuous processes. The figure also gives an analogy regarding the emergence of a color system from low-level

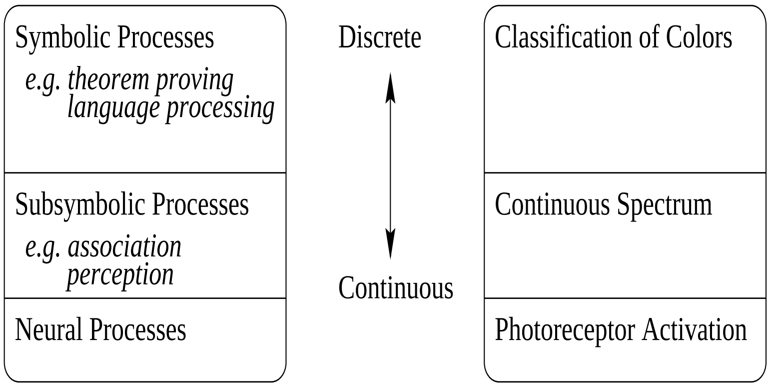

Figure 2.1: Emergence of the Discrete

functions. When presented with a spectrum, we perceive the colors to change from one to another smoothly so that the transition is continuous. But from this spectrum, we are able to group the colors into discrete categories such as "all colors that are orange and whitish".

One of the effects of the continuous giving rise to the symbolic is flexibility in our processing. For example, a purely symbolic approach to listing the rules of a language is very brittle; it cannot handle semantic constructs that are not predefined by the rules. Therefore all the rules must be previously defined; this, if not impossible, is certainly not a feasible task. In contrast, the subsymbolic continuity that is part of human intelligence seems to give flexibility to the symbolic processes that emerge[12, 14]; we certainly handle language and its constructs much better than a computer. For example, humans handle word ambiguity by looking at the context in which the word is used; computers do not process context-sensitive languages very well. In fact, the emergence of the discrete has characteristics similar to that of a connectionist system: robustness, adaptiveness, responsiveness, and flexibility. This is one of the reasons why neural networks are studied. To take this a step further, it would be nice if the network was also biologically plausible.

In the following sections, we discuss continuous computational processes that are neurally plausible and from which approximately discrete structures can emerge. Although these algorithms are biologically motivated and are examples of how the brain may be operating, the following discussion and analysis of these systems will be from an empirical and computational point of view to determine how well they self-organize and learn given patterns.

2.2 Overview of the Algorithms

These algorithms self-organize based on Hebbian learning rules, which are also called correlational learning rules[18]. The Hebbian synaptic modification hypothesis is the following: change the biological synaptic efficaces in proportion to the correlation between the pre- and postsynaptic signals[5]. In other words, if two neurons are both highly active[1], then strengthen the connection between them[3]. On the other hand, the connection is weakened if there is little activity between the neurons[2]. Although Hebb's rule is thought of as a weak learning rule, there is more biological evidence for Hebbian learning than for other processes. It has been found that learning can modulate long-term synaptic effectiveness by altering the number, size, and vesicle complement of synaptic active zones and these changes can be detected at the level of identified synapses that are critically involved in learning[2] and that short and long-term changes in synaptic efficacy underlie certain instances of short and long-term memory[8]; an example would be the Hebbian synapses in the hippocampus[26], a region of the brain associated with memory. Despite its reputation, Hebbian learning rules have been used to perform such nontrivial tasks as curve fitting[25] and the production of emergent properties through successive layers of a net[9, 10, 11], much like the emergence of orientation columns in the primary visual cortex. Correlational

[1]We can define activity between two units X and Y in terms of the correlation coefficient, $\rho(X,Y) = cov(X,Y)/\sigma(X)\sigma(Y)$ where $cov(X,Y) = E(XY) - E(X)E(Y)$, σ is the non-zero variance for the units, and E is the mathematical expectation for the units. In all cases $-1 \leq \rho \leq 1$. If the two units are independent of each other, then $\rho = 0$ and they are said to be non-correlated.

[2]This weakening should not be confused with anti-Hebbian learning in which the strength of the connection is changed in proportion to the activity, but in the *negative* direction[21].

learning is also fast, which could account for the fast learning in humans[18]; contrast this with Backpropagation which may take many thousands of iterations to find a solution. This evidence suggests that correlational learning is what biological neural circuitry implements at the synaptic level, and not other more complex rules such as BP[3].

Another common feature of these algorithms is the use of basis functions. Basis functions coarse-code the input space; for example, if $\mathbf{f}(x)$ is the function we want to learn, then $\mathbf{f}(x)$ can be expressed as a linear combination of the basis functions. The first algorithm uses trigonometric basis functions since the input is some temporal continuous signal. The other two algorithms use what is known as radial basis functions (RBFs) to coarse-code spatial patterns. An RBF network approximates some desired function by superposition of bell-shaped basis functions. Some common RBFs include the Gaussian function, some logistic functions, and the hyperbolic secant function. The response of each node v_k in an RBF net is graded so that the node responds most strongly to some input k and less strongly as the input gets farther from k. That is, the receptive field profile, θ, of v_k is the RBF which grades the response of v_k so that

$$v_k(\phi) = \theta(k - \phi) \tag{2.1}$$

where ϕ is some arbitrary input. These nodes also have a radius of support, meaning that if the input is beyond some range r of k, the response of v_k is effectively zero. If $\|k - \phi\| > r$, then $\theta(k - \phi) = 0$. As will be seen, the degree of accuracy of an RBF network is based on the number of RBFs, their location (i.e. their 'center'), and width (i.e. the shape of the bell curve). In addition to function approximation, RBFs have been used in classification problems[20, 23] and even speech recognition. The use of RBFs is motivated by the locally tuned response that is found in biological neurons[5]. These neurons respond to some finite range of the input signal space; each

[3]To date, there is no evidence that the back-error propagation mechanism exists in biological neural systems.

neuron responds most strongly to some particular input. The cochlear stereocilia cells in the ear are an example. Underlying these cells is the basilar membrane; different parts of this membrane will resonate upon detection of the proper sound frequency. This oscillation stimulates the overlying stereocilia cell to release neurotransmitters; in effect, the response of the cell is tuned to a specific frequency.

These systems are all linear in nature. It has been previously thought that linear systems are not powerful enough from both a purely computational and a biological point of view. As mentioned in the previous chapter, biologic neural processing is essentially nonlinear. Also, it is generally considered that linear systems are not effective computationally and that multilayer linear nets are equivalent to single-layer linear nets [19]. However, linear systems have been used to effectively model processes like excitatory synapses and hyperpolarizing inhibitory synapses. It would not be prudent to discount the capability of linear systems. This, of course, does not imply that there is no more use for nonlinearity; for example, the depolarization threshold of a neuron is still best modelled nonlinearly [19]. Also, the function of the axon hillock which is to trigger an action potential may or may not be modelled linearly depending on the role of the neuron. What we argue here is that we should not indiscriminately include nonlinearity into a system.

Traditionally, computation is thought to occur at the neuronal level; the basic computational unit is the neuron. In this model, the neuron is divided into three distinct components: input, integration, and communication (See figure 2.2). The dendritic

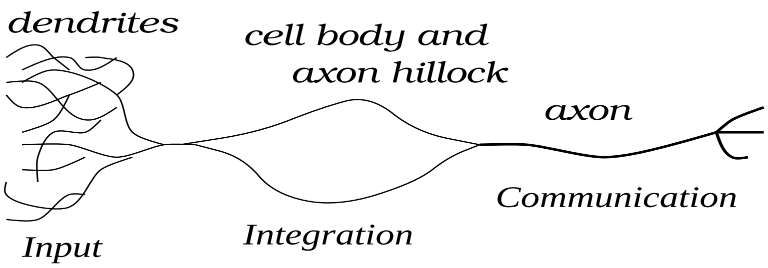

Figure 2.2: The Traditional Neuron Model

net gathers the signals, the axon hillock integrates them and generates an action potential, and the axon transmits the result. Each cell is only capable of limited computation so that complexity comes from networking these cells together. The result of this model is the development of current artificial neural nets where each node of the net approximates a neuron's function with nonlinear sigmoidal functions. However, there is increasing evidence that computation happens at the synapto-dendritic level so that the basic computational unit is the synapse[4]. Medical evidence supports the view that dendrites are involved in memory; studies of patients have found that as a person gets older, the dendritic net shrivels until only a stump remains. This occurs most markedly in patients with degenerative diseases that affect a person's cognitive capabilities such as Alzheimer[8]. In addition, it has been shown that dynamic linear systems are capable of significant information processing and that these systems can be used to explain the function of dendritic nets [15]. Much of the computation occurs in the dendritic net and the net can be modelled linearly. This alternative model of the neuron is significant because it allows the coarse-coding RBFs and filters to be implemented in the dendritic net. Figure 2.3 is a schematic of what this model looks like.

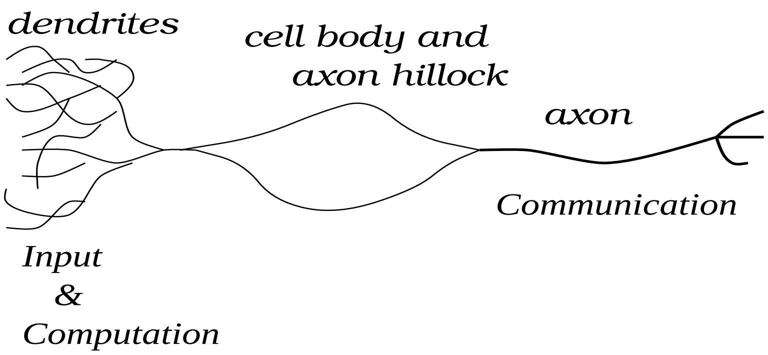

Figure 2.3: An Alternative Neuron Model

Finally, all three of these systems require only one pass of the raw input. This is

[4]This is not to say that *all* dendritic nets compute. Some do function as simple input gatherers, for example the dendrites in sensory neurons.

biologically significant because much of the data in our environment is fleeting and unreproducible; we must either reach a conclusion quickly or internalize the data for more leisurely processing.

All the algorithms to be discussed function similarly; inputs are coarse-coded and the result is modified by weights to yield some output. The difference lies in the manner that the weights are determined.

2.3 Learning with a Matched Filter System

2.3.1 Background

This first algorithm is based on the fact that for normalized patterns, the inner product $\langle \zeta, \phi \rangle$ is maximized when $\zeta = \phi$, where ζ is the temporal pattern learned by the system and ϕ is some temporal input pattern.

By definition, if a linear system[5] has an initial energy state of zero, then its response is

$$y(t) = x(t) \star h(t) = \sum_{m=0}^{t} x(m)h(t-m) \tag{2.2}$$

where $y(t)$ is the system response, $x(t)$ is the input signal, and $h(t)$ is the impulse response[6] of the system. So if we know the system's impulse response, then it is possible to determine the system response to any input pattern[24]; the system response is simply the convolution of the input sequence and the impulse response [7]. Let T be the effective signal duration; then the system response at $t = T$ is $\sum_{m=0}^{T} x(m)h(T-m)$

[5]This linear system is also known as a causal, or finite-length impulse response (FIR), system in signal processing terminology.

[6]Also known as *unit sample response*. An impulse, $\delta(t)$, is defined by $\delta(t) = 0$ for $t \neq 0$ and $\int_{-\epsilon}^{\epsilon} \delta(\lambda) d\lambda = 1$ for any real number $\epsilon > 0$. Note that $\delta(t)$ has unit area. $\delta(t)$ is not actually a function, but we can think of it as a pulse with a very short duration and very large amplitude centered at the origin.

[7]Convolutions and the role of impulse responses are discussed in greater detail in digital filter and signal processing texts, for example [1, 24].

which is the inner product between $x(t)$ and the reverse of $h(t)$. If the system's impulse response is $\zeta(T-t)$, then the system response at time T given input ϕ is the inner product between $\phi(t)$ and the reverse of $\zeta(T-t)$, which is $\zeta(t)$; thus $\langle\zeta,\phi\rangle$ is the final value of reverse convolution:

$$\zeta(T-t)\star\phi(t)\mid_{t=T}\ . \tag{2.3}$$

Therefore when the impulse response of some linear system is the time-reverse of some pattern, $h(t)=\zeta(T-t)$, the system will act as a pattern matcher for ζ; to see this, observe that if $h(t)=\zeta(T-t)$, then given some input ϕ the system response at $t=T$ (by Eq. (2.3)) is $\langle\zeta,\phi\rangle$ and the more closely ϕ matches ζ, the greater the system response at $t=T$. This linear system is called a *matched filter* for ζ[18].

2.3.2 The System

The next task is to find a system that can self-organize into a matched filter that has an impulse response $h(t)=\zeta(T-t)$. To accomplish this, we use decomposition, a common technique to analyze some signal by breaking it down into other signals that are more basic [14]. Expand $\zeta(T-t)$ in a generalized Fourier series over the interval $(0,T)$. For our purposes, we will use a set of trigonometric basis functions ϱ_k. What this does is to decompose the signal into a sum of sinusoidal components; we are breaking up the pattern $\zeta(T-t)$ into its component basis functions:

$$\zeta(T-t)=\sum_{k=0}^{\infty}c_k\varrho_k(t). \tag{2.4}$$

Because the input signals are limited in their bandwidth, only a finite number of trigonometric basis functions are needed. In this system, the following trigonometric basis is used:

$$\varrho_0(t)=1,\quad \varrho_{2k}(t)=\cos(v_kt),\quad \varrho_{2k-1}(t)=\sin(v_kt)$$

where $v_k = 2\pi k/T, (k = 1, 2, ...)$.

What the system needs to do is find the Fourier coefficients c_k upon presentation of ζ. By definition, if we have some function f that is square integrable, then the k^{th} Fourier coefficient $c_k(f)$ of f with respect to ϱ_n is $c_k = c_k(f) = \langle f, \varrho_k \rangle$. $\zeta(T - t)$ is certainly square integrable since it is bounded in both time and amplitude, so

$$c_k = \int_0^T \zeta(T - t)\varrho_k(t)\, dt. \tag{2.5}$$

Then observe that the convolution of two functions f and g is defined by $f \star g \equiv \int f(x - t)g(t)\, dt$. Therefore Eq. (2.5) becomes

$$c_k = \zeta(t) \star \varrho_k(t) \mid_{t=T} . \tag{2.6}$$

Eq. (2.6) states that if we have a linear filter with impulse response $\varrho_k(t)$ and we input the pattern $\zeta(t)$, then at time $t = T$ the output of the system is the Fourier coefficient c_k [17, 18]. The correlation here is between the input signal and the impulse response of the linear filters.

The implementation of this algorithm is based on Eq. (2.6). That is, to find c_k, the convolution between ζ and the basis function ϱ_k is computed. Intuitively, this convolution is computed by summing a set of responses which are the impulse responses starting at each time step modified by the input signal at that time step. In other words, the first response starts at time t and is the impulse response modified by the signal at time t, the second response starts at time $t + 1$ and is the impulse response modified by the signal at time $t + 1$, and so forth. This algorithm can be made more efficient by observing that we are only interested in finding the system response at time $t = T$ when learning. We can thus use Eq. (2.2) and ignore the other system responses at other t values.

The system is composed of a set of parallel linear filters. The linear filters each have impulse response equal to one of the trigonometric basis functions. Each filter also has a weight attached to it that modifies its output. A copy of the input signal is passed to each filter and each filter produces an output which is modified by its associated weight. The final responses from each filter are summed into a final output for the system.

Initially, the weights are zero-valued or random. When an input pattern is to be learned, a learning flag is set. The pattern is passed through the filters and the current weights are used. The flag tells the system that once the pattern has been processed (at time $t = T$), the graded outputs become the new weights of each filter. Note that the weights are the coefficients c_k in Eq. (2.6).

Once learning is done, the weights are not changed (unless a new signal is to be learned). Observe that subsequent impulses to the system still yield the impulse response ϱ_k from each filter but now they are modified by the weight c_k; summation of this modified response gives the reverse of the pattern ζ at time $T - t$,

$$\zeta(T - t) = \sum_{k=0}^{n} c_k \varrho_k(t). \tag{2.7}$$

This means that the impulse response of the entire *system* is $\zeta(T - t)$ (although the impulse response of each *individual* filter is still a trig basis function). The system is now a matched filter for ζ and as stated in section 2.3.1, the closer some subsequent input $\phi(t)$ matches the signal ζ, the higher the final value from the system. Figure 2.4 is a diagram representing the linear system. There are n linear filters and n coefficients. The graded output is the output of the filter modified by the coefficients.

The algorithm can be summarized as: When the system receives a pattern that is to be learned, the system uses Eq. (2.6) to determine the coefficients. Once the coefficients are set, the pattern is learned. Patterns that are subsequently input to the system do not cause the coefficients to change so that the system is fixed once learning is done.

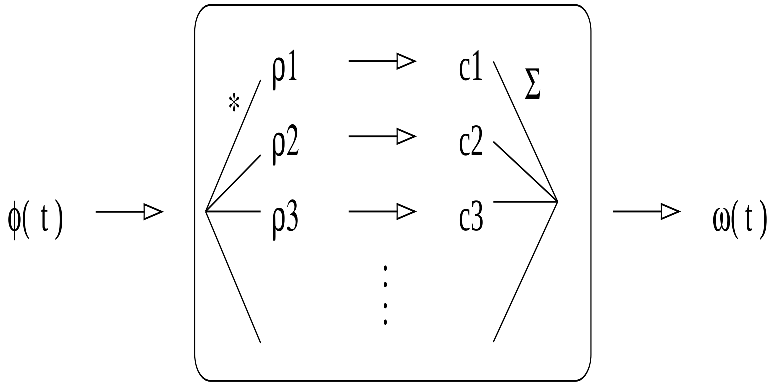

Figure 2.4: Schematic of Matched Filter System

2.3.3 Some Issues

This is an unusual system because it can both recognize and reproduce the pattern. In recognizing the pattern, the system output at $t = T$ is what matters. This is very similar to the thresholds found in biological systems in the sense that if the new pattern does not match the learned pattern, then the system output will be below some threshold and we know there is no match. But if the new pattern causes the output to be above some matching threshold, then we know there is a match. This is analogous to stimulating a neuron beyond some threshold so that an action potential is produced.

To reproduce the pattern, on the other hand, requires a unit impulse as input. The biological equivalent is stimulating a neuron with an action potential. Granted, the system response is the reverse of the pattern, but it is still recognizable and it is not difficult to reverse the pattern. It is important to note that the impulse must be a perfect δ function in order for the pattern to be precisely reproduced. A biological system will tend to generate an imperfect bell-shaped impulse which will blur the pattern.

This system works best when the patterns are normalized; we are not concerned whether one signal has twice the amplitude of another signal. Rather, what is important is the overall shape of the patterns, if the patterns are proportionally equivalent. As an algorithmic issue, this poses no problems since it is no trouble to normalize a signal before having it processed by the system. In the case of the impulse response generating the unnormalized learned pattern, we can intuitively see that the pattern generated is correct and is even easier to recognize when we normalize the pattern. For this system to be biologically plausible, there must be some evidence of biological systems performing some form of normalizing. Indeed, there is such evidence. Studies have shown that in humans, the colors that we perceive can all be mapped into a four-dimensional hypersphere and that different spectral distributions can map to the same color point [6, 7, 18]. This implies that light with different properties of brightness and color can map to the same color point and are therefore indistinguishable to us.

Algorithmically, the system learns when we tell it to learn and not otherwise so that we are in control of what pattern is learned and when the coefficients need to be determined. In a biological system, this learning signal can be generated by global reinforcement from a third source, perhaps some electrochemical signal from another neuron. There is another way to tell the system to learn. First, suppose that the system is implemented in a dendritic net[8]. So we need to send a learning signal to the dendrite. One method is through an antidromic pulse from the postsynaptic neuron [17] which is a biological form of reinforcement learning.

To see how an antidromic pulse can be used to learn, consider the effects of a signal coming into the dendritic system. The signal is processed and some output is given to the neuron. If the net is tuned to the signal, then an action potential is generated at the axon hillock. This causes the spread of an electric wave, the antidromic pulse, back into the dendritic net. By changing the membrane characteristics of the dendritic net, this pulse could change the synaptic efficacies to enhance learning of the signal.

[8] As discussed previously, computation most likely occurs in the dendritic net via linear processes.

One possible biological implementation is to have a bank of n neurons where the dendritic net of each acts as a linear filter, f, with impulse response equal to one of the basis functions. The signal, ϕ, is passed to each of the n filters and the response from each filter is modified by the long term efficacy coefficient[9] of neuron c_k. The modified outputs from each filter are summed by neuron S which produces some response signal, w. If the input signal was close to what the system is currently tuned to, S will produce a spike at the end of the response signal. An antidromic pulse travels back to each of the c_k neurons which causes them to modify their long term efficacies, thereby tuning the system more closely with the most recent signal. Figure 2.5 is a graphical depiction of this system. The arrow denotes the antidromic pulse travelling back into the dendritic net.

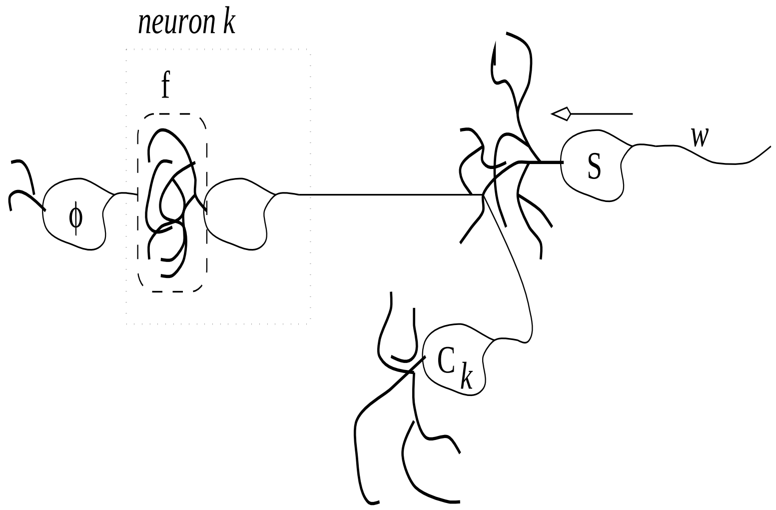

Figure 2.5: Possible Neural Implementation of Matched Filter System

Lastly, this model is an example of unsupervised learning. The pattern is input to the system and the system is told to learn it; we do not tell the system what its output should be.

[9] There are various ways that the learned coefficients can be represented in a neuron, for example in the form of cellular ion concentration or the degree of membrane permeability. We do not know, however, the method neurons actually use to encode the coefficients.

2.4 Learning with an RBF-based Gradient Descent System

2.4.1 Background

Suppose we are given a set of inputs, I, and also the set of possible corresponding outputs, O. The task is to learn some arbitrary mapping $S : I \rightarrow O$. The technique employed here is roughly analogous to the previous algorithm: the mapping S is expressed in terms of some linear combination of functions ξ_k, where $\xi_k : I \rightarrow O, k = 1, 2, ...$ is a complete[10] set of functions. S, then, is decomposed into a set of basis functions,

$$S = \sum_{k=0}^{n} c_k \xi_k. \tag{2.8}$$

For this algorithm, radial basis functions which coarse-code the input set will be used as the complete set of functions ξ_k. The correlation, therefore, is between the coarse-coded inputs and the output patterns. Since we are given ξ_k, the next task is to determine the coefficients c_k.

Suppose we have a set of input-output pairs to learn, $\phi^j \mapsto \zeta^j$. Both ϕ^j and ζ^j can be vectors; that is, the mapping is from one vector to another. For our purposes, however, we will treat ϕ^j and ζ^j as *scalars* [11]. Let $\delta^j = \zeta^j - S(\phi^j)$ so that δ^j is the difference between the output pattern ζ^j and the actual result when ϕ^j is input to S; δ^j shows the accuracy of the results of $S(\phi^j)$. If we let δ^j_h be element h of δ^j, then the summed squared error for δ^j is $\sum_h (\delta^j_h)^2$, which is $\|\delta^j\|^2$. Therefore, the error as a function of the coefficients for the least squares solution is defined to be:

$$E(\mathbf{c}) = \sum_j \|\delta^j\|^2. \tag{2.9}$$

[10]A set of functions ξ_k is complete if and only if any S can be approximated arbitrarily closely by a finite linear combination of ξ_k.

[11]Unless otherwise stated, the following derivation holds regardless of whether ϕ^j and ζ^j are scalar or vector values.

To find where the error is minimized, we take the derivative of Eq. (2.9) and set it to 0:

$$dE/dc_k = \sum_j \langle \delta^j, v_k^j \rangle = 0 \tag{2.10}$$

where $v_k^j = \xi_k(\phi^j)$. Substituting $\delta^j = \zeta^j - S(\phi^j)$ back into Eq. (2.10) and then separating the summation yields:

$$\sum_j \langle \zeta^j, v_k^j \rangle = \sum_j \langle S(\phi^j), v_k^j \rangle. \tag{2.11}$$

Store the output patterns ζ^j into a pattern vector z so that $\zeta^j = z_j$. Similarly, store v_k^j into a representation matrix U so that $v_k^j = U_{kj}$ [12]. The left-hand side of Eq. (2.11) then becomes:

$$a_k = \sum_j \langle z_j, U_{kj} \rangle. \tag{2.12}$$

If we expand Eq. (2.12), we see that

$$a_k = \langle z_1, U_{k1} \rangle + \langle z_2, U_{k2} \rangle + \langle z_3, U_{k3} \rangle + ... \tag{2.13}$$

As stated above, though z_j and U_{kj} can be vector values, we will treat them as scalars, so Eq. (2.13) reduces to

$$a_k = z_1 U_{k1} + z_2 U_{k2} + z_3 U_{k3} + ... = \langle \mathbf{z}, U_k \rangle \tag{2.14}$$

which is $\mathbf{a} = U\mathbf{z}$ [13]. Note that the length of vector $\mathbf{a}$ is the same as the number of coefficients.

[12] This is mainly for notational purposes which, as we shall see, will simplify the following mathematical manipulations.

[13] Since we will be treating the patterns as scalars, we can use $\mathbf{a} = \mathrm{U}\mathbf{z}$ in our implementation. In general, though, $\mathbf{a} = \mathrm{U}\mathbf{z}$ is just notational convenience for Eq. (2.12). This caveat also applies regarding the right-hand side of Eq. (2.11).

To expand the right-hand side of Eq. (2.11), observe that $S(\phi^j) = \sum_i c_i \xi_i(\phi^j)$ and so

$$\begin{aligned}
\sum_j \langle S(\phi^j), v_k^j \rangle &= \sum_j \langle \sum_i c_i \xi_i(\phi^j), v_k^j \rangle \\
&= \sum_{ji} \langle c_i \xi_i(\phi^j), v_k^j \rangle. \qquad (2.15)
\end{aligned}$$

Substituting v_i^j for $\xi_i(\phi^j)$ and taking out the coefficient, the right-hand side becomes

$$\sum_{ji} c_i \langle v_i^j, v_k^j \rangle. \qquad (2.16)$$

Splitting the double summation, Eq. (2.16) becomes

$$\sum_i c_i \sum_j \langle U_{ij}, U_{kj} \rangle. \qquad (2.17)$$

If we expand the inner summation of Eq. (2.17), we see that it is equal to

$$\langle U_{i1}, U_{k1} \rangle + \langle U_{i2}, U_{k2} \rangle + \langle U_{i3}, U_{k3} \rangle + ... \qquad (2.18)$$

Since we are treating the elements of U as scalar, not vector values, Eq. (2.18) reduces to

$$U_{i1}U_{k1} + U_{i2}U_{k2} + U_{i3}U_{k3} + ... = UU^T.$$

The inner summation is therefore an $N \times N$ matrix, where $i, k \,\epsilon\, \{1, ..., N\}$; we will refer to it as B and so Eq. (2.17) becomes

$$\begin{aligned}
\sum_i c_i (UU^T)_{ik} &= \sum_i c_i B_{ik} \\
&= \mathbf{c}B.
\end{aligned}$$

Since B is symmetric, $\mathbf{c}B = B\mathbf{c}$. We can then rewrite Eq. (2.11) as $\mathbf{a}= B\mathbf{c}$, therefore:

$$\mathbf{c} = B^{-1}\mathbf{a}. \qquad (2.19)$$

2.4.2 The System

From Eq. (2.19), we know how to find the coefficients. One way is to solve for $\mathbf{c}$ directly by finding the inverse of B. What we will do here instead is to solve for $\mathbf{c}$ through gradient descent; as we do this, we will see how gradient descent is related to expanding B^{-1} into its Neumann series.

Observe from Eq. (2.10) that $dE/dc_k = \sum_j \langle \delta^j, v_k^j \rangle$, which can be rewritten as

$$dE/d\mathbf{c} = U\boldsymbol{\delta} \tag{2.20}$$

since we are concerned with scalar patterns. For gradient descent, let $\dot{\mathbf{c}} = -\eta dE/d\mathbf{c}$ where η is the learning rate[14]. As stated previously,

$$\boldsymbol{\delta} = \boldsymbol{\zeta} - S(\boldsymbol{\phi}) \tag{2.21}$$

where $\boldsymbol{\zeta} = \mathbf{z}$ and from Eq. (2.8), $S(\boldsymbol{\phi}) = \sum_k c_k \xi(\ \boldsymbol{\phi}) = \sum_k c_k v_k^j = \sum_k c_k U_{kj} = \mathbf{c}U$. Using Eqs. (2.20) and (2.21), we see that:

$$\begin{aligned} \dot{\mathbf{c}} &= -\eta U(\mathbf{z} - \mathbf{c}U) \\ &= -\eta(U\mathbf{z} - U\mathbf{c}U) \\ &= -\eta(U\mathbf{z} - UU^T\mathbf{c}) \\ &= -\eta(\mathbf{a} - B\mathbf{c}). \end{aligned} \tag{2.22}$$

Let us take Eq. (2.22) one step further. Suppose we remove $-\eta$, in effect setting the learning rate to -1. Eq. (2.22) tells us that the change in coefficients between each successive iteration depends on the current coefficients. Therefore

$$\mathbf{c}_{k+1} - \mathbf{c}_k = \mathbf{a} - B\mathbf{c}_k. \tag{2.23}$$

[14] Also known as the *step size*.

From here, we will solve for $\mathbf{c}_{k+1}$[15]:

$$\begin{aligned}\mathbf{c}_{k+1} &= \mathbf{a} + \mathbf{c}_k - B\mathbf{c}_k \\ &= \mathbf{a} + \mathbf{I}\mathbf{c}_k - B\mathbf{c}_k \\ &= \mathbf{a} + (\mathbf{I} - B)\mathbf{c}_k. \end{aligned} \tag{2.24}$$

Eq. (2.24) is one way to iteratively determine the coefficients through gradient descent. There is a strong relationship between solving for $\mathbf{c}$ through gradient descent and solving for $\mathbf{c}$ by expanding B^{-1} into its Neumann series. Let us set $\mathbf{c}_0 = \mathbf{a}$ and calculate $\mathbf{c}_{k+1}$ using Eq. (2.24):

$$\begin{aligned}\mathbf{c}_0 &= \mathbf{a} \\ \mathbf{c}_1 &= \mathbf{a} + (\mathbf{I} - B)\mathbf{a} \\ \mathbf{c}_2 &= \mathbf{a} + (\mathbf{I} - B)\mathbf{a} + (\mathbf{I} - B)^2\mathbf{a} \\ &\dots \end{aligned}$$

Therefore,

$$\begin{aligned}\mathbf{c} &= \mathbf{a} + (\mathbf{I} - B)\mathbf{a} + (\mathbf{I} - B)^2\mathbf{a} + \dots \\ &= \left[\mathbf{I} + (\mathbf{I} - B) + (\mathbf{I} - B)^2 + \dots\right]\mathbf{a}. \end{aligned} \tag{2.25}$$

Of course, the terms within the brackets [...] of Eq. (2.25) is the Neumann expansion of B^{-1}. In effect, what gradient descent is doing (when we set $\mathbf{c}_0 = \mathbf{a}$) is calculating successive approximations to $\mathbf{c}$ using the Neumann expansion of B^{-1}. That is, gradient descent is computing:

$$\mathbf{c}_k = \sum_{j=0}^{k}(\mathbf{I} - B)^j\mathbf{a}. \tag{2.26}$$

As stated previously, this form of gradient descent will be used to compute $\mathbf{c}$, specifically Eq. (2.24). Given some inputs and the corresponding outputs, the first task

[15] $\mathbf{c}_{k+1}$ does not refer to *element* $k+1$ in the coefficient array; it refers to the *array* of coefficients at iteration $k+1$.

is to populate the representation matrix U_{kj} using some set of radial basis functions. With U_{kj}, we can then compute $\mathbf{a}$ and B. Let $\mathbf{c}_0 = \mathbf{a}$. With this information, use Eq. (2.24) to iteratively calculate c until $\dot{\mathbf{c}}$ falls below some acceptable level.

Figure 2.6 is a schematic diagram of this system. Note that it is similar but not the same as figure 2.4; the two systems are fundamentally different since the learning algorithms are different. Here, the input and output are spatial patterns as opposed to temporal patterns in figure 2.4. In addition, the nodes here are coarse-coding RBFs rather than linear filters performing convolutions. Observe that in figure 2.6, there is only one set of coefficients and the output vector elements are treated as a whole. In our implementation, we treat each element of the output separately; in this case, there would be a set of coefficients for each output element and so the system would be a fully interconnected layered network.

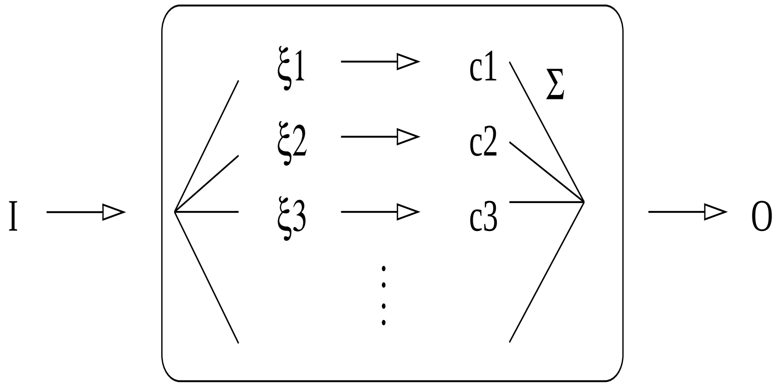

Figure 2.6: Schematic of Gradient Descent System

2.4.3 Some Issues

Gradient descent is a procedure that follows the contour of the error surface, always moving downhill in the steepest direction; this is akin to a skier skiing downhill, all the time looking for the steepest path from his current location. When a point is reached where the error can no longer be reduced, the procedure halts and we have found a minimum. The best possible solution is found in a global minimum, the lowest point on the error surface; there may be several equally good global minima, which is true for the algorithm described here. In most systems, there are also local minima which are not the lowest points on the error surface. When using gradient descent, there may be a chance for the system to find a local minima solution; this, of course, is usually undesirable and in fact is one of the drawbacks of Backpropagation. But because the error here is quadratic in the coefficients, there are no local minima and therefore gradient descent cannot get trapped; the system will compute the best possible coefficients. This can be seen intuitively by observing that gradient descent, in this case, is equivalent to finding the coefficients by expanding B^{-1}. Gradient descent will find the optimal coefficients just as if we solved for $\mathbf{c}$ directly using Eq. (2.19).

One parameter in the gradient descent procedure is the learning rate, η, which determines the speed the skier goes down the hill. Continuous time gradient descent will always find the minimum, but the discrete time gradient descent methodology used here can overshoot the minimum if η is too large. Simulated annealing is one method that overcomes this problem[16]. Like other gradient descent based algorithms, η must be chosen so that the minimum is found with the fewest number of iterations. Here, we choose to set η to -1.00 so that we are calculating successive approximations to $\mathbf{c}$ using the Neumann expansion of B^{-1}; the results are quite good and suit our purposes for analyzing the effects of the system parameters. Keep in mind, however, that lowering the learning rate does yield better results.

The condition placed on the representation functions is that they must be complete. Because they need not be orthogonal, there may be more than one linear combination of ξ_k that will yield the function S; in other words, there may be more than one equally

[16] With simulated annealing, the step size is gradually reduced. Simulated annealing is commonly used to escape from local minima but can be applied here as well.

good solution. To see why this is so, recall the orthogonality property:

$$\int \xi_m(\phi^j)\xi_n(\phi^j)d\phi^j = 0 \tag{2.27}$$

whenever $n \neq m$ [17]. One way to find c_k in Eq. (2.8) is to multiply both sides by $\xi_k(\phi^j)$ and then integrate both sides. If the orthogonality property holds, then all but the k^{th} term on the right-hand side vanishes and we can solve directly for c_k; that is, each c_k is unique. But since we do not require orthogonality, there may be more than one term on the right-hand side so that the coefficients may be underdetermined.

Figure 2.7 shows a possible biological implementation of this system for one coarse-coding neuron, k; note that there are n of these subsystems. To make this more understandable, we will again assume that the input-output pairs are scalar values and not vectors. Input ϕ is sent to the dendritic net of neuron k which computes the radial basis function ξ_k. The output of the net, v_k, is used in two ways. When there is no learning, v_k is sent to another neuron S where the effect of v_k is modified by the long term efficacy coefficient of neuron c_k; neuron S sums up all the modified results to produce output w. When the system is learning, w is compared to the target, ζ. Synapses combine the difference, δ, with v_k which is then used to modify the long term efficacy. Note that this biological implementation utilizes the dendritic net to perform the main computation. Other neurons are involved in simple subtraction, summation, and multiplication, all of which synapses are capable of performing.

In our program, we simply tell the system when to learn. There are several plausible ways for a biological system to learn, one of which is to send a signal from a third source to the δ neuron; we can thus turn on or off the gradient descent process. Another possibility is the presence or absence of a corresponding output pattern ζ; in this scenario, gradient descent operates (and the system learns) only when a pattern from the ζ neuron is present to stimulate the δ neuron. This is consistent with the system since in order to learn, we must have the output pattern.

[17] To be more correct, Eq. (2.27) should be $\int \xi_m(\phi^j)\overline{\xi_n(\phi^j)}d\phi^j = 0$ but since we are dealing only with real functions and $f(x) = \overline{f(x)}$ for any real function $f(x)$, we choose to simplify 2.27.

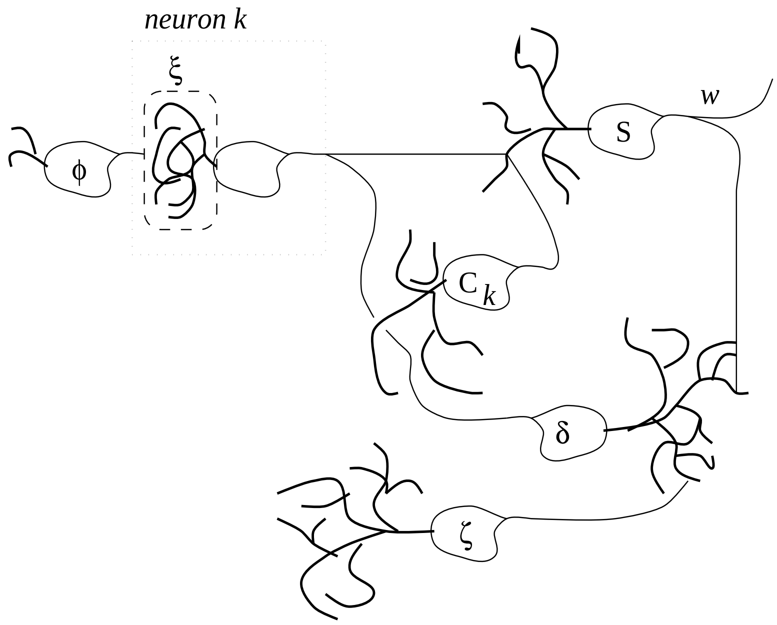

Figure 2.7: Possible Neural Implementation of Gradient Descent System

Throughout the discussion about this algorithm, we have alluded to the mappings as being from scalar to scalar value; doing this simplifies the derivation and allows us to see more clearly the relationship between gradient descent and the Neumann series. However, this algorithm is perfectly capable of handling vector to vector mappings as well. To do this, we simply use the more general definitions of $\mathbf{a}$ and B when computing $\mathbf{c}$. This algorithm can be viewed as a generalization of the one in [20] which describes a vector to scalar mapping system. If we are only interested in scalar outputs (e.g. if $\phi \in \Re$), then $\mathbf{a} = B\mathbf{c}$ holds and we can simplify it to

$$\mathbf{z} = U^T \mathbf{c}. \tag{2.28}$$

$\mathbf{c}$ can then be determined from Eq. (2.28) via matrix inversion or gradient descent. From Eq. (2.28), we can see more clearly that all inputs are fed into the system before learning takes place; in effect, all the patterns are learned simultaneously.

Lastly, this model is an example of supervised learning. Both the inputs and outputs must be supplied in order for the system to learn the proper coefficient values.

2.5 Learning with an RBF-based Layered Network System

2.5.1 Background

This last algorithm is most similar in form to traditional neural networks in the sense that this system consists of a layered network, each layer interconnected to its adjacent layers. However, a correlational learning rule and radial basis functions are used, setting this apart from most other layered networks.

Suppose we are given a set of inputs, I, and also the set of possible corresponding outputs, O. The task is to learn some arbitrary mapping $S : I \rightarrow O$. Similar to the previous systems, this algorithm must find a set of weights[18] that correlate the inputs with the outputs. Unlike the first algorithm, but similar to the second, the correlation here is between the outputs and the coarse-coded inputs.

2.5.2 The System

A set of nodes, v_k, uses radial basis functions to coarse-code the input, ϕ. The output of this system is a linear combination of the coarse-coded input $v_k(\phi)$; for each output node w_i, $w_i = \sum_k C_{ki} v_k(\phi)$ where C_{ki} is the connection weight between v_k and w_i. In other words, $\mathbf{w}= C\mathbf{v}(\phi)$. To find the interconnection matrix C, we correlate the coarse-coded input with the target response ζ so that $\dot{C}_{ki} = \zeta_i^j v_k(\phi^j)$ for the j^{th} pattern. The change in the interconnection matrix is thus $\dot{C} = \zeta^j \wedge \mathbf{v}(\phi^j)$.[19] Therefore,

$$C = \sum_j \zeta^j \wedge \mathbf{v}(\phi^j). \tag{2.29}$$

[18]The weights here serve the same function as the coefficients in the previous systems. The term 'weight' is used in traditional neural net literature and that is what we choose to follow since this system *looks* traditional, but the two terms are generally interchangeable.

[19]$\wedge$ is defined to be the outer product operator between two vectors; the result of this operation is a matrix. Here, $\dot{C}$ is a $k \times i$ matrix.

The entire system is composed of a three-layered network with each layer fully interconnected to its adjacent layers. The number of nodes in the input and output layers is equal to the size of the input and output, respectively. Initially, the weights can be random values or zeroed out. The input is sent to each of the intermediate layer nodes for coarse-coding. When learning, C is updated according to Eq. (2.29). Once all the input-output pairs have been processed, C is set and the system will have learned the patterns. Figure 2.8 shows the interconnections in this network. Note

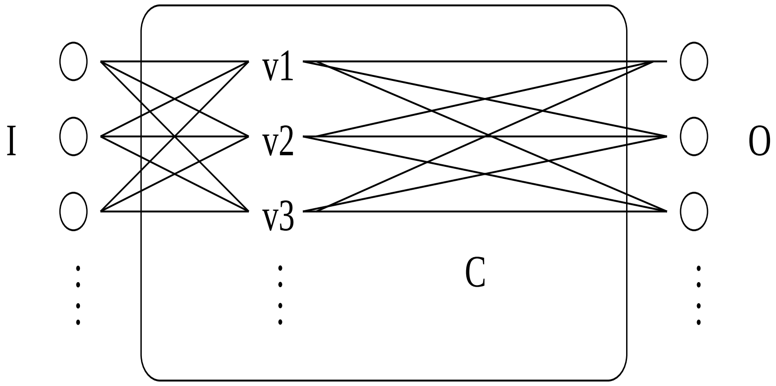

Figure 2.8: Schematic of Layered Correlational System

that this is virtually identical to figure 2.6. If the output elements in figure 2.6 are treated separately, then these two systems would be the same except for the learning algorithms.

2.5.3 Some Issues

Figure 2.9 shows a plausible biological implementation for one coarse-coding neuron, k; only the connections and neurons associated with k are shown, but keep in mind that there are n of these subsystems. Input pattern ϕ is sent to neuron k for coarse-coding; $v_k(\phi)$ is then distributed to each output neuron S_i after being modified by the long-term efficacy coefficient C_{ki}. Finally, S_i sums the results from each of the n coarse-coding neurons to produce the final value. When the system is in learning mode, ζ is present and ζ_i is distributed to each of the C_{ki} neurons, modifying its long-term efficacy and tuning it to the input-output pair. Note that the complexity comes from the treatment of the output elements as separate items.

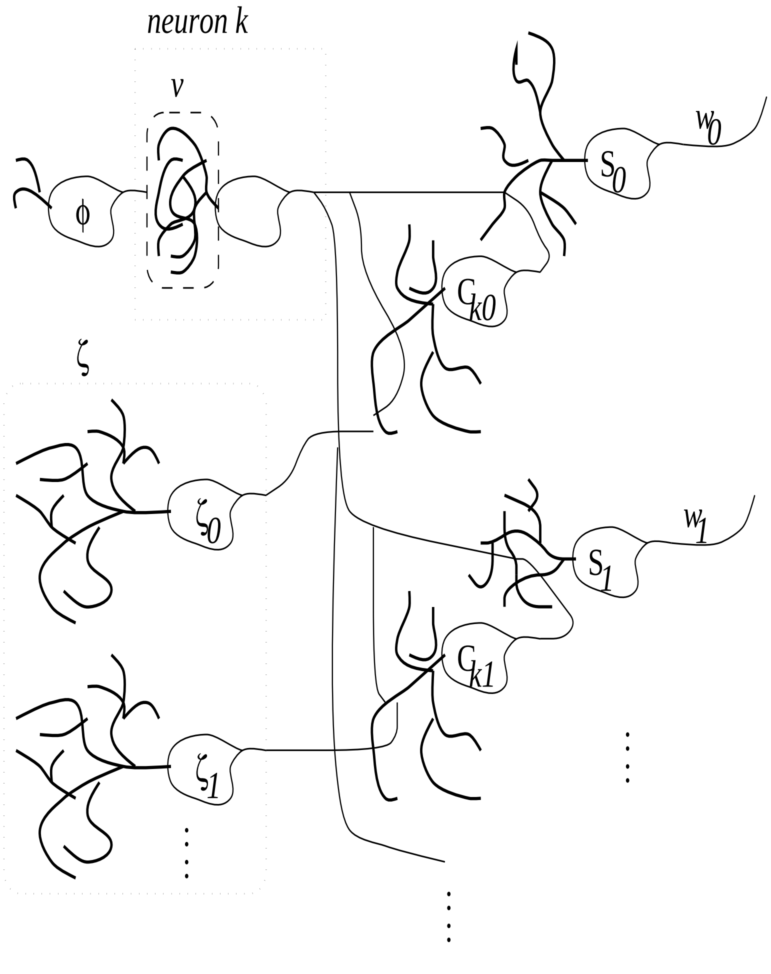

Figure 2.9: Possible Neural Implementation of Layered Correlational System

As with the system described in section 2.4.3, the simplest method to tell this system to learn is through the presence of the desired output pattern ζ since, like other supervised learning rules, ζ must be supplied when learning. Thus, when ζ is available, C_{ki} is stimulated into tuning itself, otherwise it is not motivated to change.

Chapter 3

Experimentation and Results

Much of the current research into neural networks has been concerned with determining the maximal performance of a system; part of this involves figuring out ways to tinker with system parameters in the hopes of fine-tuning the system. This is important in determining the computational potential of the system. One of the drawbacks, however, is that sometimes the ideal system is not biologically plausible. For example, the coefficients in all three systems are initialized to some value before learning; this normally produces the best solution. But what if they were not perfectly initialized, as would be the case in a biological system? How well can the system perform under these conditions?

The experiments on these three algorithms take into account both issues. That is, we will determine the system performance under ideal conditions. Of more importance to us, though, is taking into account biological realities that tell us system conditions may approach the ideal but never are; just as the passage of time or the consumption of alcohol, or even the normal frailties of our cellular system affects our performance, we must determine the effects on performance when initial conditions are not ideal, when the actual learning process is not ideal, and when the learned coefficients are corrupted.

3.1 Overview of Testing Procedure

All three systems were implemented in the C language under the Unix operating system; the programs should be readily portable to any platform. Several versions of each system were written, each specifically intended to investigate certain aspects of the system.

3.2 Learning with a Matched Filter System

3.2.1 Setup and Procedures

This system was tested with thirteen different inputs ranging from complex high bandwidth signals to simpler sinusoids and lines. These signals were chosen to represent a variety of forms; some look roughly similar while others are unique. Table 3.1 lists the functions used to generate the signals. Throughout this section, the signals will be identified by their function name, for example $a(x)$. Each signal's effective duration, T, was arbitrarily set at 500 time units. Figures 3.1 and 3.2 graphically

Table 3.1: Functions Used to Generate Signals

$a(x) = \sin(3\pi x) + \sin(9\pi x) + \sin((2\pi x) + 1)$
$b(x) = \sin(1.2\pi x) + \sin((9\pi x) + 2.5) + \cos((4\pi x) - 2)$
$c(x) = \sin(2\pi x) + \sin((4\pi x) + 0.5) + \sin(6\pi x)$
$d(x) = \sin(3.5\pi x)$
$e(x) = \sin(2\pi x) + \cos((2\pi x) - 0.5) + \cos(-\pi x)$
$f(x) = -\cos((-\pi x) + 2$
$g(x) = 0.25x + 1$
$h(x) = -3x + 2$

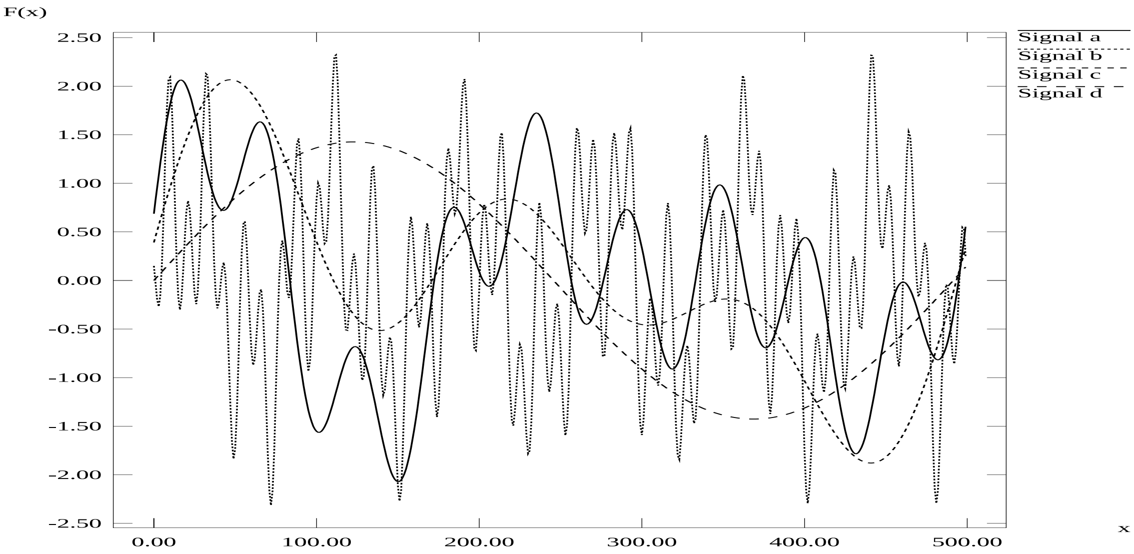

Figure 3.1: Signals a - d in the Matched Filter System

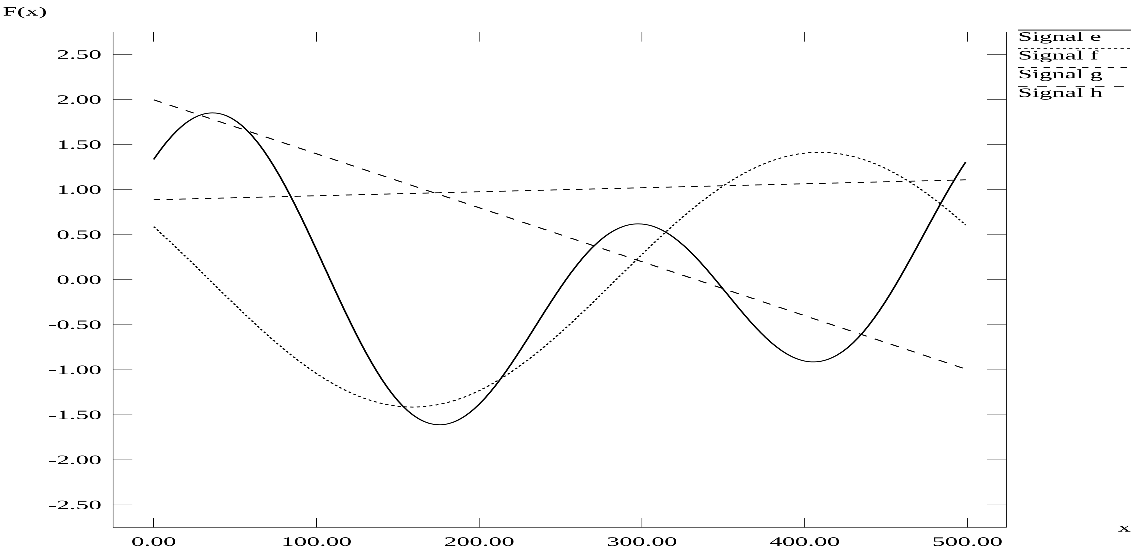

Figure 3.2: Signals e - h in the Matched Filter System

depict the periodic signals.

To determine how well the system learned the signal, the root-mean-squares error (RMS) was calculated using the normalized version of the actual signal and the normalized, reversed, impulse response:

$$RMS = \sqrt{\frac{\sum_t (a_t - r_t)^2}{T}}$$

where a_t is the actual output at time t and r_t is the impulse response at time t[1]. In addition, we also tested how well the system recognizes the signal if it was input once more after being learned; recall that the system recognizes an input signal by producing a high response at time T. We expect this final response to be highest when the input signal matches the learned signal[2].

3.2.2 Results

There are several parameters that contribute to the system's initial condition: the number of basis functions, their frequencies (v_k in Eq. (2.3.2)), the signals themselves, and the initial coefficients; we will attempt to determine their effect on signal learning. Later, we will investigate the effect of low-precision mathematics on the actual learning process. Finally, we will determine what effect there is on the system when the coefficients are corrupted after the learning process.

First, we need to find what initial conditions lead to the best possible results. We determined that the greater the number of functions, the better the results. In addition, to achieve excellent results, the c_k coefficients must be initialized to 1.0 and there must be an accurate orthogonal basis; this conclusion is consistent with the

[1]It does not matter how we look at the error; for example, the summed-squared error, $\sum_t (a_t - r_t)^2$, is also appropriate. Throughout this paper, we use the L_2 (euclidean) metric as it is most commonly found in neural network literature.

[2]Observe that we can not say exactly how high the output will be since the system determines the response, but we can compare the responses to other signals the system did not learn; this is one of the characteristics of an unsupervised learning system.

matched filter's mathematical derivation. The results from an impulse input signal[3] after learning can be seen in the next three figures while the final system responses from various input signals after learning are shown in table 3.3; these figures and tables will be discussed in more detail later in this section.

If this was a biological system, however, chances are that the initial conditions prior to learning would not be ideal. To various degrees, natural systems are always imperfect[4]. We can not expect a real neural network to properly initialize all system parameters before proceeding to learn some signal, and so we need to look at how system performance is affected under such conditions; that is, can the system perform adequately if the parameters are not set to their mathematically ideal values before learning? Figure 3.3 is a plot of RMS error as a function of the number of basis functions with c_k and v_k set to their ideal starting values. Figures 3.4 and 3.5 are the results from an average of eighty test runs that plot the RMS error as a function of the percentage deviation from the ideal parameter values[5] with 140 basis functions. Observe that in general, choosing random v_k and initial c_k values yield worse results. Also note that the greater the number of basis functions, the better the results. These generalities are consistent with our expectations. It is important to note, however, that there are exceptions, for example when more randomness leads to better results; this phenomenon will be discussed later.

From figure 3.3, it can be seen that in general the number of basis functions required varies with signal complexity. A signal with high bandwidth needs more functions to capture its features than a signal with low bandwidth. For example, observe that for signal $b(x)$, the RMS error is 0.61 when $n = 70$ resulting in some missing spikes (see figure 3.6). But when $n = 139$, the RMS goes down to 0.02, a very good match for

[3]Here, an impulse signal is a signal $i(t)$ where $i(0) = 1$ and $i(t) = 0$ for $0 < t < T$.

[4]Nature's goal is not perfection, but rather survival. It is a truism that short-term survival depends on the ability to fit flawlessly into the current environment, but a natural system that is too perfect will not be able to adapt to new situations easily leading to extinction; evolution builds imperfect machines.

[5]Here, percent deviation means that each coefficient is randomly generated to within some percentage of the ideal value. The percentage determines the range of values that each coefficient may take.

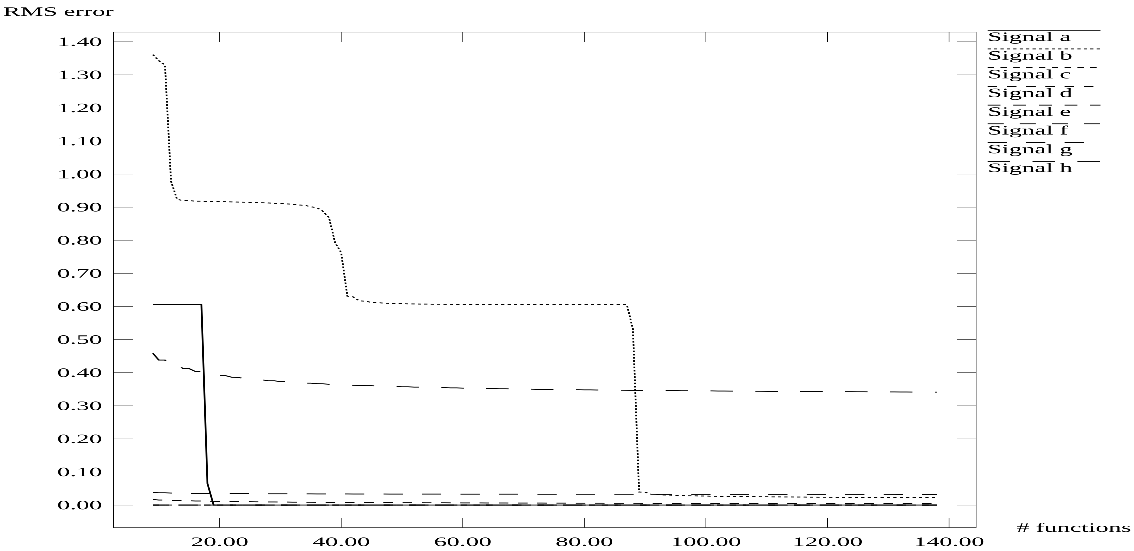

Figure 3.3: RMS error vs. #Basis Functions

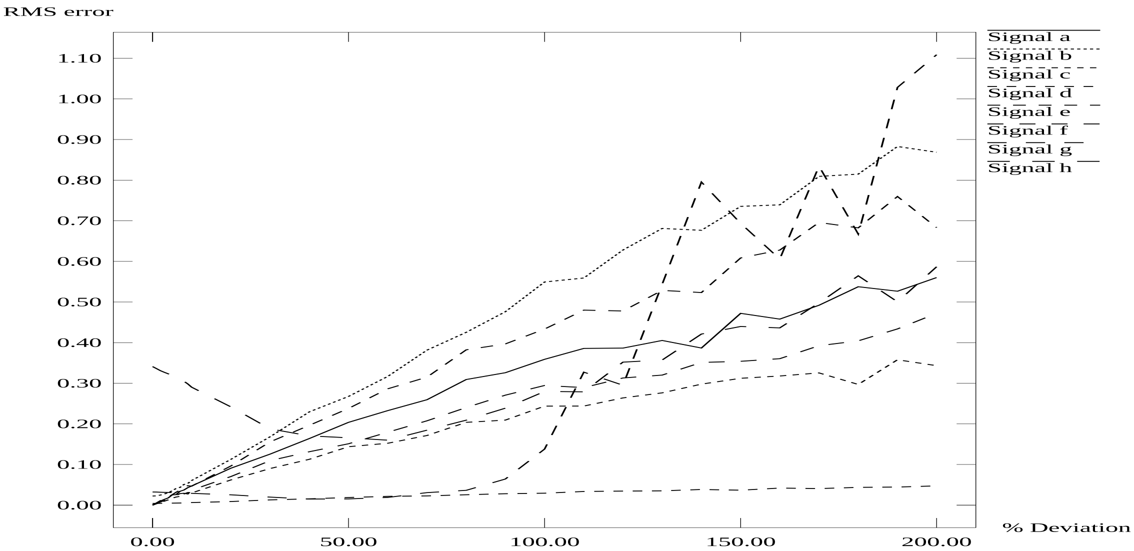

Figure 3.4: Avg. RMS error vs. % Deviation from Ideal c_k

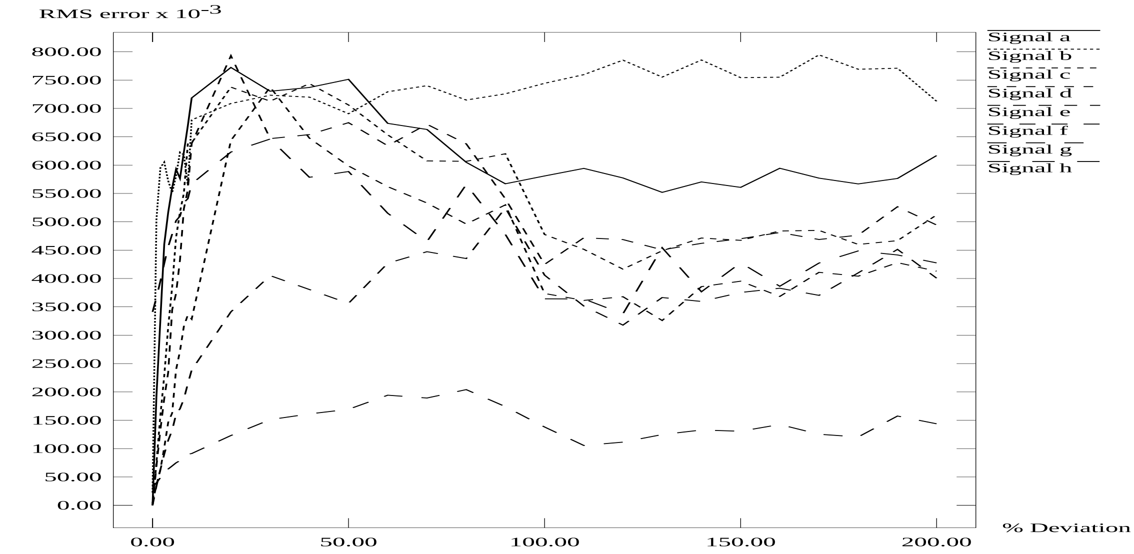

Figure 3.5: Avg. RMS error vs. % Deviation from Ideal v_k

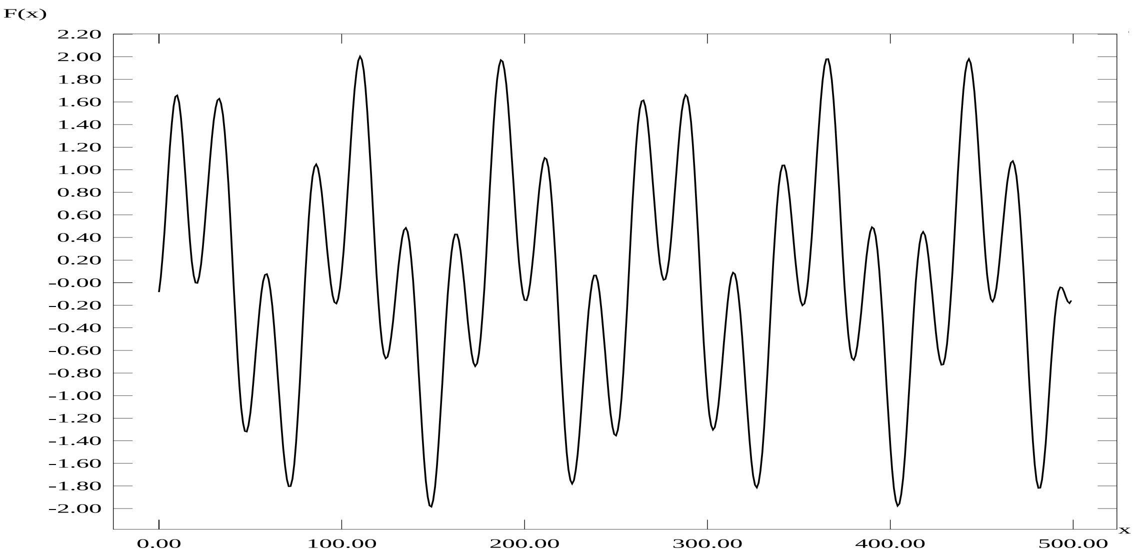

Figure 3.6: Learned $b(x)$ with $n = 70$

$b(x)$. We expect that in a biological system, there are a great many of these filters since there will be more than enough neurons to implement them; the number of these filters is not a constraining factor to this system's performance.

Another noticeable feature is the inability of the system to achieve a low RMS value for some lines. For $h(x)$, the best RMS value reached is 0.33 even with 900 basis functions. It seems as if the system is trying to make the line continuously periodic leading to a distortion of the ends as figure 3.7 shows; the slope suffers as a result. Note, however, that the fit to the line is rather good, so a high RMS value does not necessarily imply a poor fit for nonperiodic signals. $h(x)$ is an extreme example of

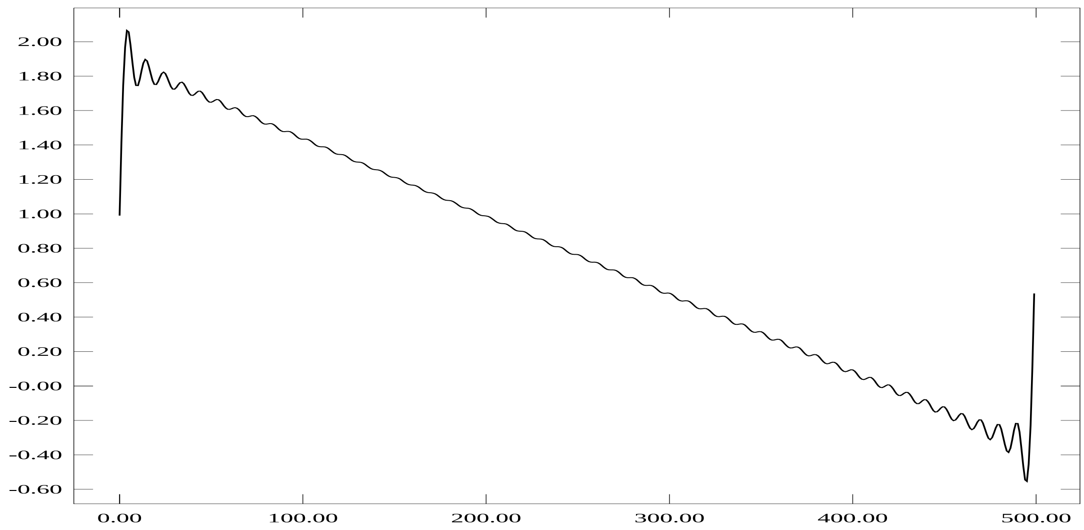

Figure 3.7: Learned $h(x)$ with $n = 139$

what happens when the signal is nonperiodic. The implication is that for this system, nonperiodic signals are more difficult to learn than others and the data supports this. Figure 3.8 shows five signals that are nonperiodic and figures 3.9, 3.10, 3.11 show the average RMS values from a set of eighty test runs per signal.

Observe that the system can not perfectly learn nonperiodic signals under ideal parameter conditions; the extent of the problem hinges on whether the system perceives

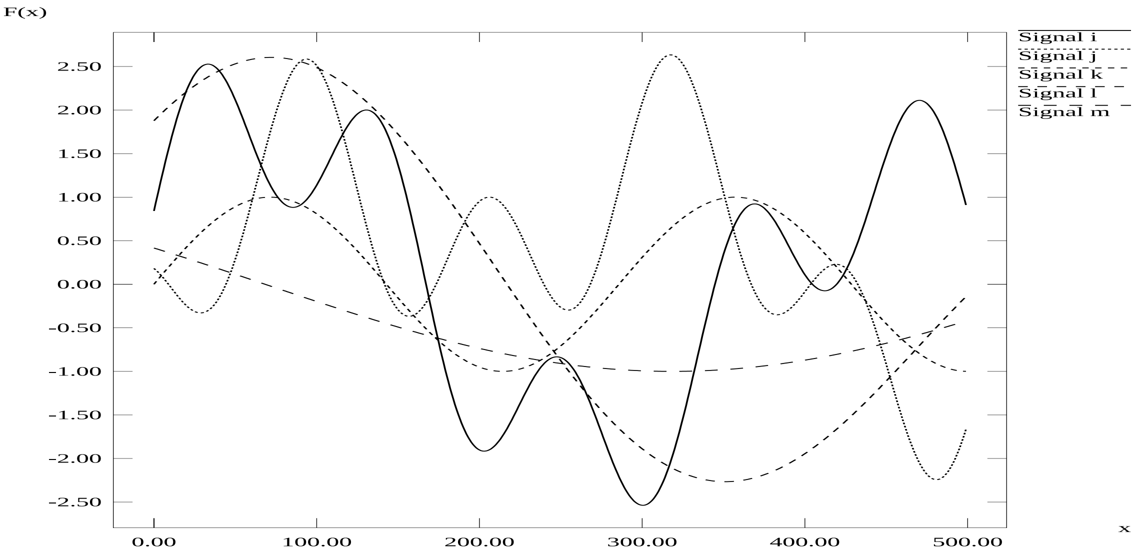

Figure 3.8: Signals i - m in the Matched Filter System

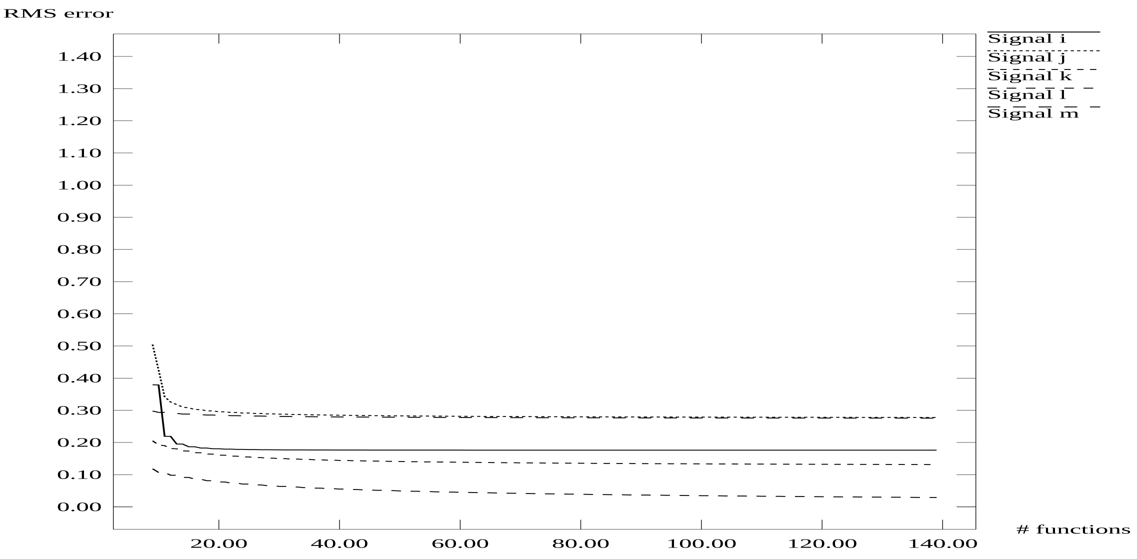

Figure 3.9: Avg. RMS error vs. #Basis Functions (nonperiodic)

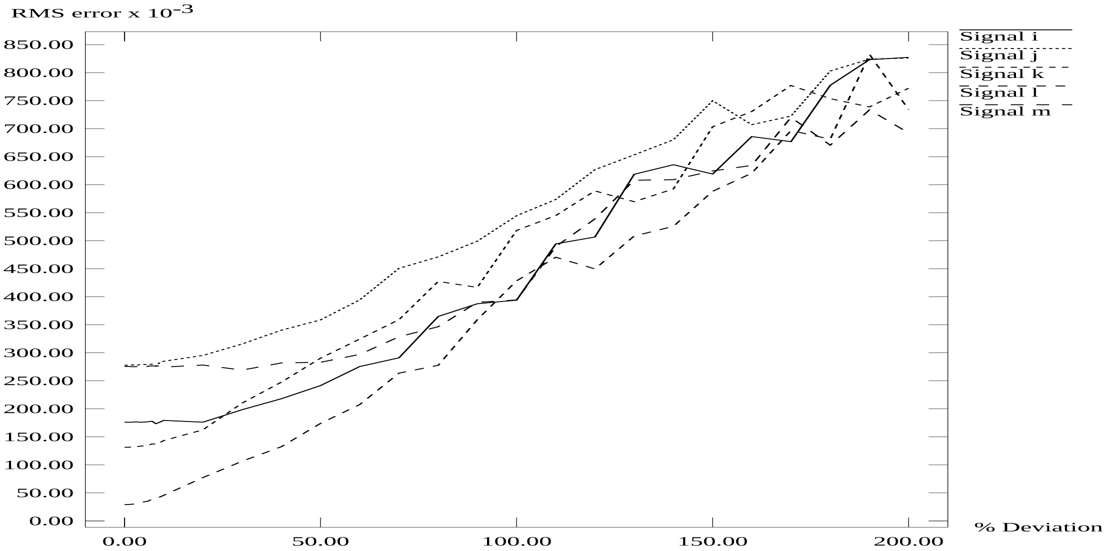

Figure 3.10: Avg. RMS error vs. % Deviation from Ideal c_k (nonperiodic)

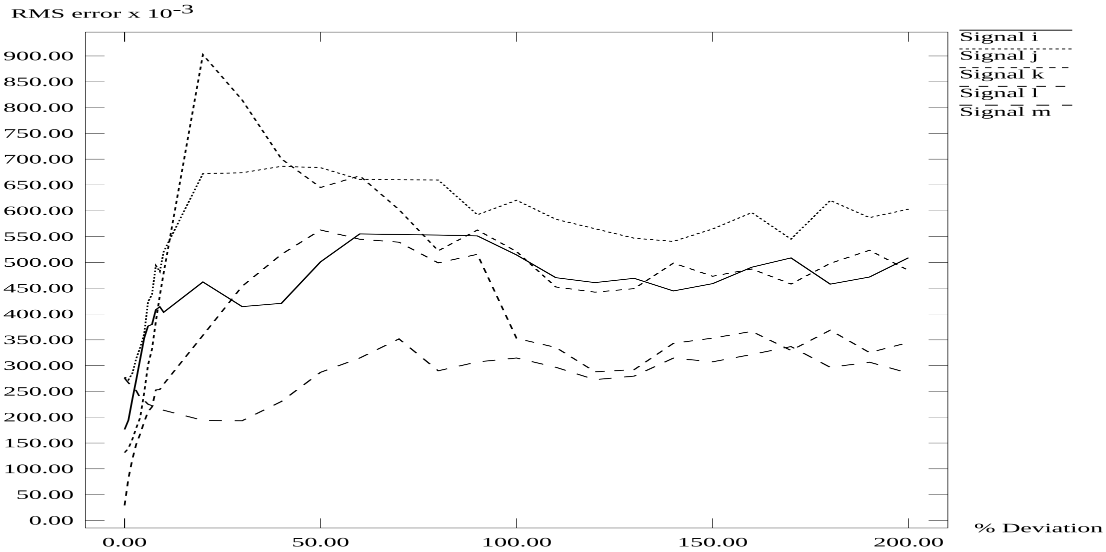

Figure 3.11: Avg. RMS error vs. % Deviation from Ideal v_k (nonperiodic)

the signal to have a period that is a multiple of the time duration. More importantly, however, is that when the initial parameters deviate from the ideal, the dynamics and the results are no different between periodic and nonperiodic signals; the treatment of all signals is indistinguishable under biologically constrained conditions where variance from the ideal parameters is expected.

From figure 3.4, we can see that the error is linearly proportional to the percent deviation from ideal c_k except for the linear signals, in which case there is a dip in the error before it starts to rise; in figure 3.10, we can see that for nonperiodic signals, up to the 50% point, the error rises more slowly or remains flat. The dip in error for linear signals $g(x)$ and $h(x)$ seems to be the extreme of this feature. As the error rises, the learned signal becomes an increasingly distorted version of the original. For example, figure 3.12 shows $b(x)$ learned by the system given some initial c_k set within 90% of the ideal. Note that the learned signal is essentially $b(x)$ with its spikes distorted.

Using a set of basis functions that are not completely orthogonal shows that the system is much less tolerant of v_k deviating from the ideal than deviations in c_k; from figure 3.5, observe that even with a deviation of 10%, the RMS error for $b(x)$ is approximately 0.68. These errors are quite high and a quick visual inspection of the system's impulse response shows that sometimes its resemblance to the original signal is superficial, indicating the importance of orthogonality to this system. Figure 3.13 is the impulse response after the system attempted to learn $b(x)$. Note that the similarity between the impulse response and $b(x)$ seems slight but this is acceptable because, as we shall see later, the system response is good[6]. From figures 3.5 and 3.11, notice that for each signal, there seems to be a 'hump' in the error, with the highest point at about 50%; beyond 100%, the error stabilizes. Where the error stabilizes depends on each signal's complexity which seems to be defined by two attributes:

[6]Acceptability is very subjective. For this system, an acceptable set of parameters will be defined as those which cause the final system response to be higher for the learned signal than for other signals. We will discuss system response later in this section.

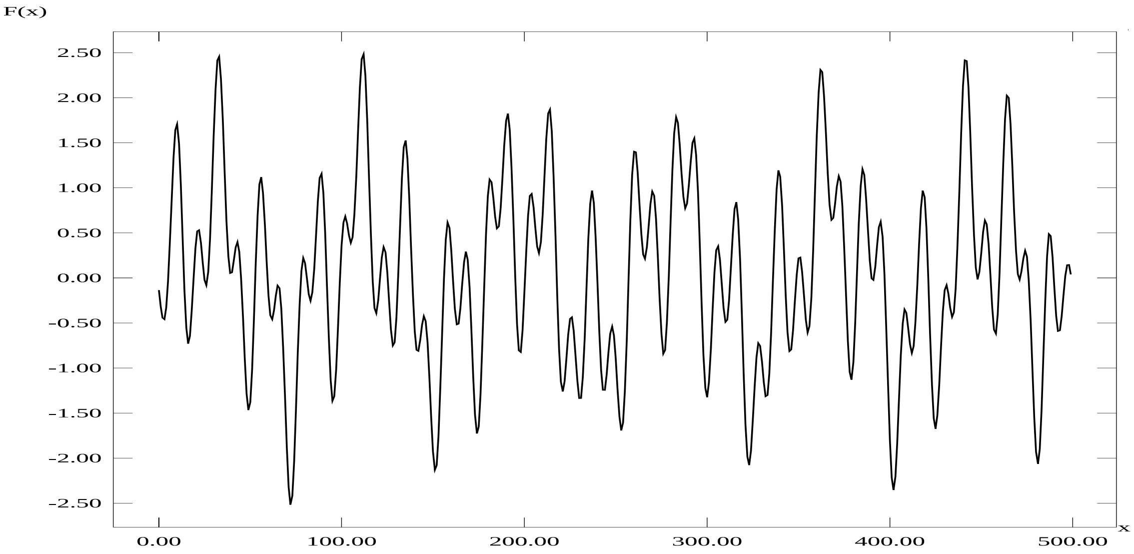

Figure 3.12: Learned $b(x)$ with Deviation from c_k of 90%

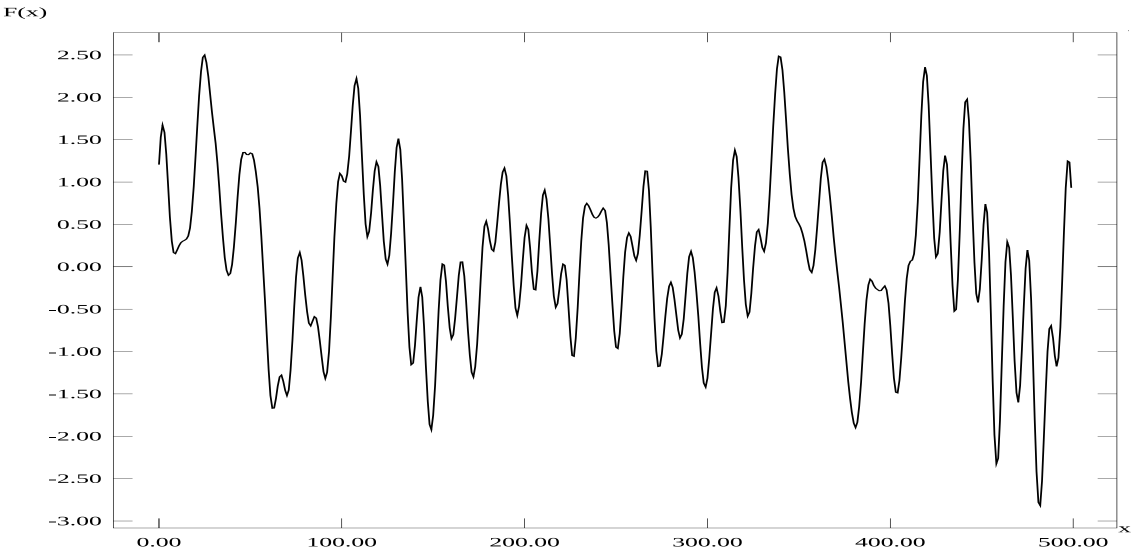

Figure 3.13: Learned $b(x)$ with Deviation from v_k of 10%

the bandwidth and amplitude. A complicated signal such as $b(x)$ has an error that stabilizes at 0.80. On the other hand, simple sinusoids like $d(x)$ and $f(x)$ stabilize at lower error values. The simplest signal in the group is $g(x)$, virtually a flat line; the error, accordingly, is the lowest. This can also be observed for the noncontinuous signals in figure 3.11. These results suggest that if v_k is to be varied, then it is better to generate them in a completely random fashion; small random deviations from the ideal yields the worst performance, as evidenced by the 'hump'. One possible explanation for this unusual result is that small deviations actually cause the basis functions to be nonorthogonal, while introducing randomness at least gives the basis a chance to be orthogonal. This was checked by computing the inner product between pairs of basis functions after they were normalized; the closer the inner product approaches zero, the more orthogonal the basis. Figure 3.14 is a plot of the average inner product between basis function pairs and the percent deviation from ideal v_k. Note that with 0% v_k deviation, the inner product computed for all pairs is 0.0. Also, the 'hump' is visible, showing that the basis is most non-orthogonal at the 50% mark.

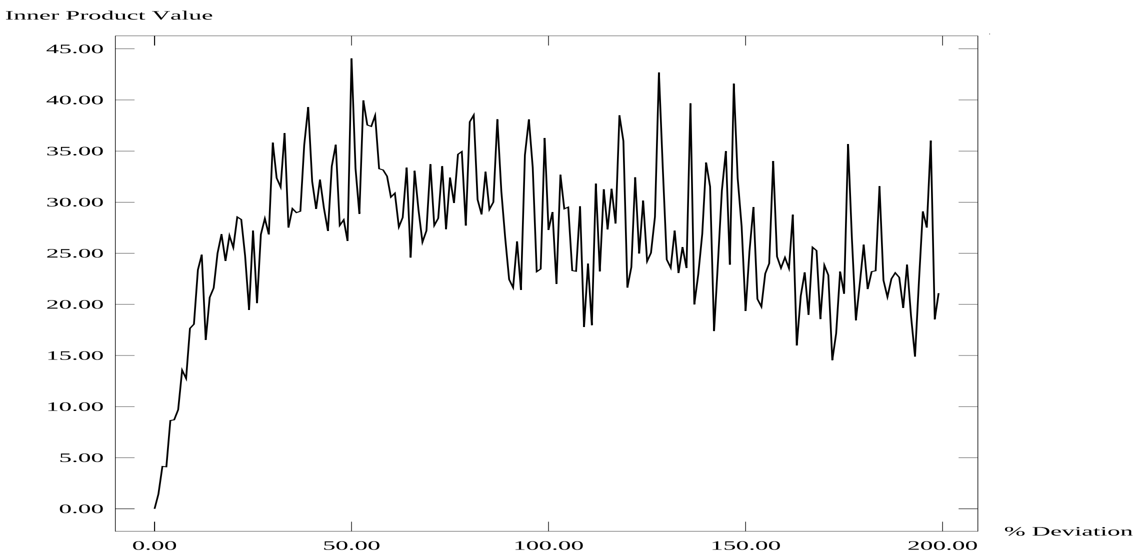

Figure 3.14: Avg. Inner Product between Basis Pairs vs. $\%v_k$ deviation

One of the more interesting observations is that randomness can actually produce better results than what we considered to be ideal conditions. For each signal, we kept track of which combination of parameters gave the lowest RMS values; the results for one of the test runs are shown in table 3.2. Notice that for the two linear

Table 3.2: Best RMS error Values

Signal	# basis functions	v_k % Deviation	c_k % Deviation	RMS error
a(x)	19	0	0	0.000000
b(x)	138	0	0	0.022422
c(x)	9	0	0	0.000000
d(x)	138	0	0	0.004250
e(x)	9	0	0	0.000000
f(x)	9	0	0	0.000000
g(x)	112	0	50	0.008276
h(x)	43	75	50	0.098395

signals, the best match comes from randomized parameters; the biggest improvement is for $h(x)$ which dramatically decreased its RMS error from 0.35 to 0.10. This occurs infrequently but the fact that it did happen at all was surprising. In addition, observe from figure 3.11 that for $m(x)$, the error actually went *down* before going back up. Randomness in the system's initial condition can sometimes produce better matches than if ideal parameters were used; this is most evident for the nonperiodic signals.

So far, we have been concerned with how well the system can reproduce a learned signal. Now we will discuss the effect of the initial parameter conditions on the system's ability to recognize a signal; along with this is the issue of generalization. Here, generalization is taken to mean the system's ability to produce a final response proportional to how closely an input signal matches the one learned by the system[7].

Recall that an input signal will generate a high system response at $t = T$ if the input

[7] In other types of systems, generalization may have a slightly different meaning. In a supervised learning system, for example, it may mean that given some new input, the system will produce the one correct response.

signal matches the learned signal. How high is relative to other input signals that do not match since the system is unsupervised. For example, figure 3.15 shows the system response when each of the eight signals is passed through again after the system learns $b(x)$ with ideal parameters. Notice that the highest value at $t = T$ comes from $b(x)$,

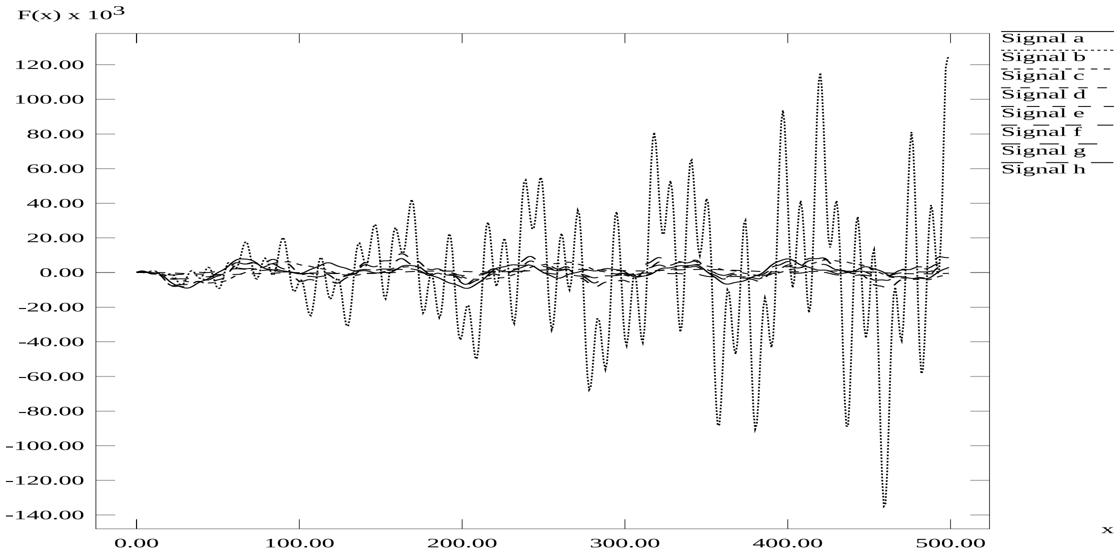

Figure 3.15: System Response for Each Signal after Learning $b(x)$

much higher than the responses from any other signal. Table 3.3 shows the system responses at $t = T$ after learning all eight signals under ideal parameter conditions. The values from input signals processed by a system matched for it are italicized. Note that given any two different signals, the system response is the same whether one signal is learned and the other input, or vice versa. This phenomenon is a result of the convolution performed by the system. For example, if the system matches $a(x)$ and then signal $b(x)$ is given, the system performs the identical convolution as if $b(x)$ was the learned signal and $a(x)$ was the input; the convolution operation is symmetrical, just like the inner product operation. This symmetry holds only under ideal conditions and therefore is biologically insignificant.

In addition, periodic signals generate the same system response when it is input to a

Table 3.3: System Response at $t = T$ (Ideal Parameters)

Input Signal	Learned Signal			
	$a(x)$	$b(x)$	$c(x)$	$d(x)$
a(x)	*125000.00*	2423.47	78232.61	-2216.46
b(x)	2423.47	*125040.39*	-863.51	-603.99
c(x)	78232.61	-863.51	*125000.00*	69532.52
d(x)	-2216.46	-603.99	69532.52	*124998.10*
e(x)	55778.12	2702.44	62352.55	1234.70
f(x)	0.00	1943.31	-65622.89	-110538.62
g(x)	-2593.19	4608.95	-6395.72	-5772.24
h(x)	35019.66	8618.29	86370.88	82083.69
Input Signal	**Learned Signal**			
	$e(x)$	$f(x)$	$g(x)$	$h(x)$
a(x)	55778.12	0.00	-2593.19	35019.66
b(x)	2702.44	1943.31	4608.95	8618.29
c(x)	62352.55	-65622.89	-6395.72	86370.88
d(x)	1234.70	-110538.62	-5772.24	82083.69
e(x)	*125000.00*	26144.00	-2357.03	31830.41
f(x)	26144.00	*125000.00*	5658.31	-76412.47
g(x)	-2357.03	5658.31	*249483.29*	118439.08
h(x)	31830.41	-76412.47	118439.08	*155766.41*

system matched to it, about 125000. This is rather unusual but is simply a result of the characteristics of the signals tested; each of the signals have similar amplitude and oscillate around $y = 0$. In other words, each signal has the same vertical displacement. To check this, each signal was displaced upwards. The resulting system response for each was higher; for example, modifying $a(x)$ by vertically displacing it by $+4$, gives the final system response of 239286. Another way we verified this was to directly calculate the inner product; if this technique yields the same results as that from the system, then this unusual system response is not due to the system. Both checks confirm that the relative level of system response is a characteristic of the signal.

From inspecting table 3.3, we can see that this system is capable of generalizing; the closer the input signal is to the learned signal, the higher the final system response. Of course, being close is up to the system and what may look close to us may not be to the system. Nevertheless, there is a rough correlation between what we and what the system considers similar. For example, consider signals $a(x)$, $c(x)$, and $h(x)$. $c(x)$ looks like a simple version of $a(x)$ and a complex version of $h(x)$, but $a(x)$ and $h(x)$ are dissimilar. The data supports this observation; after learning $c(x)$ the final system responses for inputs $a(x)$ and $h(x)$ are approximately 80000. When the system learns $a(x)$, the final system response for input $h(x)$ is only ~ 35000 but for input $c(x)$ the response is ~ 80000; the result is the same for the reverse situation.

So the final system response under ideal parameter conditions are as we expected; there is a marked gap between the response when the matched signal is input versus the response from other signals. Under random parameter conditions, however, we expect worse results and the data reflects this. Tables 3.4 and 3.5 show the final system response from two trial runs; c_k and v_k are varied within 10% of their ideal values. Observe that the system still performs well most of the time as evidenced by the relatively high final response for an input to a system matched for it compared to the response for an input to a system not matched for it. But there are instances when the system turned out to match another signal better than the one it should

Table 3.4: System Response at $t = T$ (10% c_k Deviation)

Input Signal	Learned Signal			
	$a(x)$	$b(x)$	$c(x)$	$d(x)$
a(x)	*124491.91*	2522.71	75222.70	-2070.22
b(x)	2506.00	*124753.37*	-729.23	-555.20
c(x)	77529.62	-744.19	*116834.81*	63200.81
d(x)	-2158.10	-527.36	62673.31	*113763.12*
e(x)	56046.82	2940.45	59353.03	1343.78
f(x)	0.00	2001.13	-59287.44	-100309.71
g(x)	-2523.99	4829.97	-5871.45	-5217.77
h(x)	34085.21	8136.90	79290.82	74643.78

Input Signal	Learned Signal			
	$e(x)$	$f(x)$	$g(x)$	$h(x)$
a(x)	54979.32	0.00	-2435.01	33061.23
b(x)	2911.96	2074.69	4893.78	8309.65
c(x)	58956.35	-64295.36	-5910.58	82518.29
d(x)	1443.67	-107976.14	-5200.71	79497.42
e(x)	*120377.01*	28571.87	-2291.48	29803.63
f(x)	26329.57	*124919.50*	5145.33	-73966.33
g(x)	-2135.12	5542.00	*261350.46*	122525.97
h(x)	28833.71	-74841.75	124981.33	*153579.00*

Table 3.5: System Response at $t = T$ (10% v_k Deviation)

Input Signal	Learned Signal			
	$a(x)$	$b(x)$	$c(x)$	$d(x)$
a(x)	*196633.89*	11627.63	142366.89	18220.86
b(x)	741.70	*179409.79*	572.52	1669.05
c(x)	96959.37	-13351.60	*210553.70*	120423.27
d(x)	-18996.67	-10593.57	87097.53	*171660.55*
e(x)	62541.03	9674.38	114335.82	46406.34
f(x)	17138.04	13938.82	-63512.68	-132229.32
g(x)	11638.04	8691.14	-27800.52	-48331.26
h(x)	53166.96	23038.02	118556.16	113829.26
Input Signal	Learned Signal			
	$e(x)$	$f(x)$	$g(x)$	$h(x)$
a(x)	50508.07	-23385.60	-25880.38	46241.19
b(x)	6776.22	-5340.19	10564.61	22008.55
c(x)	65927.54	-126321.91	-37025.57	102580.40
d(x)	33810.67	-183469.61	-27287.18	110610.99
e(x)	*62823.83*	58.28	-32851.19	26719.25
f(x)	-9439.55	*173885.49*	12149.98	-97431.61
g(x)	705.31	21057.92	*740684.74*	358244.64
h(x)	50597.59	-141244.95	333062.94	*289094.67*

have learned. Most of the time, the bad match is between two signals that are similar (for example, $h(x)$, $c(x)$, and $d(x)$) even under ideal parameter conditions. Note that in many instances, the highest final response is still very small, especially for random v_k. Though this implies that the system did not match any of the signals very well, nevertheless it frequently recognized the correct signal.

To find out the extent of the system's recognition capability when the parameters are not properly initialized, we attempted to determine how many times the final system response for a signal it did not learn was higher than the response for the signal it did learn. In general, using a basis that is not exactly orthogonal gives worse results than starting with random coefficients, again showing the importance of orthogonality. Tables 3.6 and 3.7 show the percentage of times the system matches correctly when varying c_k and v_k, respectively.

Observe that some signals are easier to recognize than others. In addition, note that the effect on signal recognition is the same as on signal reproduction; these results are strongly correlated with how well the system learned the signal and hence how well it reproduces a signal. That is, as the c_k bound increased, the system found it progressively harder to reproduce the signal *and* recognize it; the RMS error and the number of bad matches both increased as the percent c_k deviation increased. Similarly, as the v_k bound increased, the RMS error and the number of bad matches remained stable. The percentage of times the system recognized the correct signal is very high; if we allow for one bad match, then the success rate approaches 100%. For random v_k, this was unexpected given the rather high RMS error. This suggests that as long as the signals are sufficiently distinct from each other, the system has no problems recognizing the correct one even with random initial conditions. We can conclude that though the system may err at times, total failure can usually be avoided and reasonable results produced. This inability to match perfectly and at all times makes this an inexact system under non-ideal conditions, but the error rate is low enough to be acceptable.

Table 3.6: System Response Matching Ability with Non-Ideal c_k

	Learned Signal (10% c_k Deviation)							
	$a(x)$	$b(x)$	$c(x)$	$d(x)$	$e(x)$	$f(x)$	$g(x)$	$h(x)$
No Bad Matches	100%	100%	100%	100%	100%	100%	100%	100%
Up to One Bad Match	100%	100%	100%	100%	100%	100%	100%	100%

	Learned Signal (50% c_k Deviation)							
	$a(x)$	$b(x)$	$c(x)$	$d(x)$	$e(x)$	$f(x)$	$g(x)$	$h(x)$
No Bad Matches	100%	100%	100%	100%	100%	100%	100%	56%
Up to One Bad Match	100%	100%	100%	100%	100%	100%	100%	100%

	Learned Signal (100% c_k Deviation)							
	$a(x)$	$b(x)$	$c(x)$	$d(x)$	$e(x)$	$f(x)$	$g(x)$	$h(x)$
No Bad Matches	100%	100%	86%	100%	100%	100%	96%	56%
Up to One Bad Match	100%	100%	96%	100%	100%	100%	100%	100%

	Learned Signal (200% c_k Deviation)							
	$a(x)$	$b(x)$	$c(x)$	$d(x)$	$e(x)$	$f(x)$	$g(x)$	$h(x)$
No Bad Matches	82%	100%	62%	78%	94%	76%	80%	32%
Up to One Bad Match	90%	100%	68%	80%	100%	82%	82%	86%

Table 3.7: System Response Matching Ability with Non-Ideal v_k

	Learned Signal (10% v_k Deviation)							
	$a(x)$	$b(x)$	$c(x)$	$d(x)$	$e(x)$	$f(x)$	$g(x)$	$h(x)$
No Bad Matches	98%	100%	98%	100%	96%	100%	100%	94%
Up to One Bad Match	100%	100%	100%	100%	100%	100%	100%	100%

	Learned Signal (50% v_k Deviation)							
	$a(x)$	$b(x)$	$c(x)$	$d(x)$	$e(x)$	$f(x)$	$g(x)$	$h(x)$
No Bad Matches	98%	100%	86%	100%	90%	86%	100%	74%
Up to One Bad Match	100%	100%	98%	100%	100%	98%	100%	100%

	Learned Signal (100% v_k Deviation)							
	$a(x)$	$b(x)$	$c(x)$	$d(x)$	$e(x)$	$f(x)$	$g(x)$	$h(x)$
No Bad Matches	90%	100%	88%	100%	98%	100%	100%	72%
Up to One Bad Match	100%	100%	98%	100%	100%	100%	100%	100%

	Learned Signal (200% v_k Deviation)							
	$a(x)$	$b(x)$	$c(x)$	$d(x)$	$e(x)$	$f(x)$	$g(x)$	$h(x)$
No Bad Matches	92%	100%	90%	98%	98%	94%	100%	74%
Up to One Bad Match	100%	100%	96%	100%	100%	100%	100%	100%

From these results, we can see that starting with biologically constrained parameters still produce good system responses. In general, of course, the results are less than ideal; what is surprising is that in many instances, they are quite acceptable and occasionally may even be better than when initial conditions are non-ideal.

So far, we have discussed the initial system conditions and their effect on performance. Another biological concern is the degree of precision of the calculations; biological neural net processing is not capable of the high-precision computation that is a cornerstone of modern computers, so we need to determine if this system can perform under similar conditions. We limited the digits of precision throughout the convolution process; that is, the results from the basis functions are rounded as well as the results from each filter. Table 3.8 shows the RMS values from this test. Note that even with truncation to one decimal place at every step of the convolution, the

Table 3.8: RMS error When Convolution Calculations are Rounded

	Precision			
Signal	1	.1	.01	.001
a(x)	0.807643	0.076636	0.007426	0.000739
b(x)	1.186301	0.074519	0.025613	0.022566
c(x)	0.814973	0.098581	0.009525	0.000945
d(x)	0.451993	0.058753	0.006493	0.004188
e(x)	0.596428	0.054143	0.005911	0.000547
f(x)	0.950609	0.120609	0.011730	0.001164
g(x)	0.091312	0.076304	0.042999	0.031229
h(x)	1.009763	0.553874	0.341844	0.340924

results do not degrade much from the high-precision results of previous tests. Using integers in the calculation gives a noticeably higher error; the data suggests that this jump in error is due to signal complexity. Although $b(x)$ and $g(x)$ have similar error rates under ideal conditions, note that with integer precision, $b(x)$'s error is 1.19 while $g(x)$'s error remains at a low 0.09. There is some evidence that biological systems are capable only of single digit precision before the processing time becomes prohibitive.

If this is the case, then this system's performance does not degrade much under those conditions.

The final biological concern is memory retention after learning; that is, how severe is the damage to the system after a few beers? We need to determine the effect if the coefficients are corrupted after learning. From figure 3.16, we can see that the dynamics are the same as in figure 3.4 which set the initial c_k values to within various bounds of ideal; the error steadily climbs as the bounds get larger. Observe also that randomly corrupting the final c_k values can actually lead to better results, for example $h(x)$.

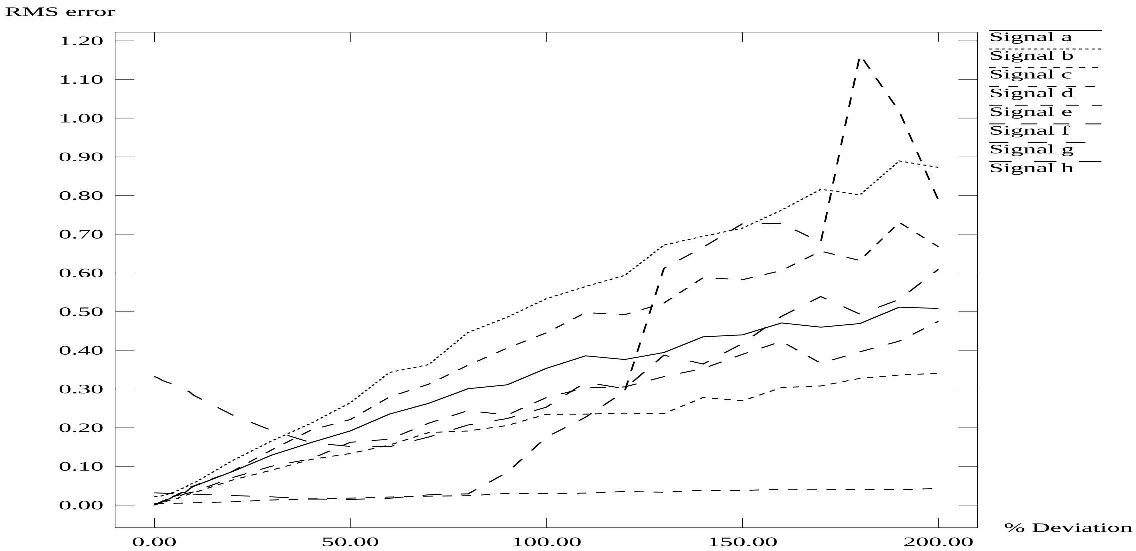

Figure 3.16: Avg. RMS error vs. % Deviation from Final c_k

In this section we discussed a matched filter's performance under various conditions and tried to shed light on some interesting behavior. We conclude that the system is an excellent signal matcher with ideal parameters and still performs acceptably under biologically constrained conditions[8].

[8]This is *not* an endorsement of alcoholic beverages; if you drink, drink responsibly and do not drive.

3.3 Learning with an RBF-based Gradient Descent System

3.3.1 Setup and Procedures

For this system, we decided to see if it could learn a nonlinear mapping, specifically a mapping that normalizes a 2-element vector. There are other interesting nonlinear mappings; we picked this because normalization is a non-trivial problem that is useful both mathematically and biologically. Without loss of generality, we will assume that all the input datapoints are in the first quadrant so that the vector elements are all positive real numbers less than 4.0. We are able to limit the space size and generalize because of each RBF's localized nature. Figure 3.17 shows some sample input datapoints and figure 3.18 shows the datapoints normalized. Since the input

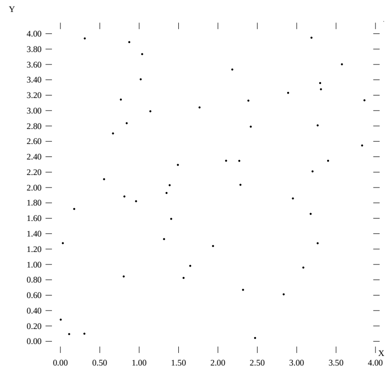

Figure 3.17: Sample input datapoints

could be from anywhere within the space, the RBFs were initially spread as evenly as possible in an orderly fashion to cover as much of the space as possible. In addition, an appropriate set of training data must be chosen; there must be enough datapoints appropriately distributed to capture the details of the function. For this reason, the training data were randomly but evenly distributed. We chose this distribution because we were not only interested in how well the system learns the training set, but also how well the system generalizes after the training set is learned.

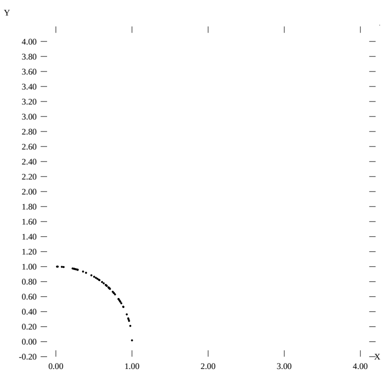

Figure 3.18: Input datapoints normalized

The system performance is determined by how well it normalizes a datapoint. We will measure this in terms of the average distance between the actual and expected normalized output; a distance of 0.0 means the actual and expected results matched perfectly.

For most of the testing, we used the hyperbolic secant function as the RBF; it is bell-shaped and is nearly identical to the Gaussian curve, the most prevalent RBF type in the neural network community. However, we also tested the performance with other RBF shapes. For each kind of curve, there is a parameter that determines its sharpness; note that using the same value for this parameter in different curve forms does not imply that the two curves have the same width since the curves are not related. To roughly compare the results from using different kinds of RBF shapes, the following tables do not list the radius parameter; instead, the area underneath the RBF is shown to give an idea of the RBF size[9].

[9] In the following result tables, the RBF Size refers to the area underneath the RBF. The smaller the size, the more narrow the RBF.

3.3.2 Results

There are several parameters that contribute to the system's initial condition: the RBF's radius, the RBF and data density, the shape of the RBFs, and the initial coefficients; we will attempt to determine their effect on learning. Later, we will investigate the effect of low-precision mathematics on the actual learning process. Finally, we will determine what effect there is on the system when the coefficients are corrupted after the learning process.

This system can learn the given training data very well as long as there is an adequate number of RBFs, each with an appropriate radius. The best results come from placing an RBF on each datapoint so that there is a one-to-one correspondence between the set of RBFs and the set of datapoints; in this case, the radii must be small enough so that there is only slight overlap between each RBF. Observe from table 3.9 that the radius of each RBF must decrease as the RBF density increases; too much overlap between RBFs give very poor results (when the error is $\gg 100$) and in fact the coefficient calculations never converge to some value. Note, however, that because in this example there is exactly one RBF directly over exactly one datapoint, there is a radius threshold such that any radius smaller than the threshold will achieve excellent results.

In a biological system, however, it is improbable that an RBF can be placed directly over each pattern since we do not know beforehand what the training set will be. To see what would happen in this scenario, the RBFs as well as the datapoints were evenly but randomly distributed through the space; there is still the same number of RBFs as there are datapoints, but now there is no one-to-one correspondence. From the results shown in table 3.10, we can see that system performance does not reach the perfect accuracy as seen in table 3.9; for example, for a datapoint density of 25, the best average distance between actual and expected results is 0.098243 and occurs when the RBF size is 0.209440. More importantly, the dynamics under random

Table 3.9: Avg. Distance; Exactly One RBF on Each Datapoint (GD)

RBF Size	Data and RBF Density per Unit Square				
	1	9	25	49	81
3.141593	0.000000	$\gg$100.0	$\gg$100.0	$\gg$100.0	$\gg$100.0
1.047198	0.000000	$\gg$100.0	$\gg$100.0	$\gg$100.0	$\gg$100.0
0.628319	0.000000	$\gg$100.0	$\gg$100.0	$\gg$100.0	$\gg$100.0
0.448799	0.000000	$\gg$100.0	$\gg$100.0	$\gg$100.0	$\gg$100.0
0.349066	0.000000	0.010754	$\gg$100.0	$\gg$100.0	$\gg$100.0
0.285599	0.000000	0.000000	$\gg$100.0	$\gg$100.0	$\gg$100.0
0.241661	0.000000	0.000000	$\gg$100.0	$\gg$100.0	$\gg$100.0
0.209440	0.000000	0.000000	7.153000	$\gg$100.0	$\gg$100.0
0.184800	0.000000	0.000000	0.000138	$\gg$100.0	$\gg$100.0
0.165347	0.000000	0.000000	0.000000	$\gg$100.0	$\gg$100.0
0.149600	0.000000	0.000000	0.000000	65.421710	$\gg$100.0
0.136591	0.000000	0.000000	0.000000	0.021437	$\gg$100.0
0.125664	0.000000	0.000000	0.000000	0.000011	$\gg$100.0
0.116355	0.000000	0.000000	0.000000	0.000000	$\gg$100.0
0.108331	0.000000	0.000000	0.000000	0.000000	0.313142
0.101342	0.000000	0.000000	0.000000	0.000000	0.052719
0.095200	0.000000	0.000000	0.000000	0.000000	0.000104

Table 3.10: Avg. Distance; Same Number of RBFs and Datapoints (GD)

RBF	Data and RBF Density per Unit Square				
Size	1	9	25	49	81
3.141593	0.070480	$\gg$100.0	$\gg$100.0	$\gg$100.0	$\gg$100.0
1.047198	0.241367	$\gg$100.0	$\gg$100.0	$\gg$100.0	$\gg$100.0
0.628319	0.340768	$\gg$100.0	$\gg$100.0	$\gg$100.0	$\gg$100.0
0.448799	0.378757	$\gg$100.0	$\gg$100.0	$\gg$100.0	$\gg$100.0
0.349066	0.687453	0.146429	$\gg$100.0	$\gg$100.0	$\gg$100.0
0.285599	0.749690	0.202512	$\gg$100.0	$\gg$100.0	$\gg$100.0
0.241661	0.876280	0.344866	$\gg$100.0	$\gg$100.0	$\gg$100.0
0.209440	0.989245	0.434700	0.098243	$\gg$100.0	$\gg$100.0
0.184800	0.994824	0.425459	0.109328	$\gg$100.0	$\gg$100.0
0.165347	1.000000	0.637221	0.109339	$\gg$100.0	$\gg$100.0
0.149600	0.983868	0.706930	0.265595	$\gg$100.0	$\gg$100.0
0.136591	0.997935	0.837380	0.290038	0.141889	$\gg$100.0
0.125664	0.992508	0.721585	0.249611	0.218504	$\gg$100.0
0.116355	0.940261	0.868048	0.495425	0.253674	$\gg$100.0
0.108331	0.999279	0.980590	0.493613	0.208730	$\gg$100.0
0.101342	0.991274	0.738343	0.490880	0.253347	0.109569
0.089760	0.893763	0.827858	0.614821	0.333838	0.103164
0.080554	0.993861	0.859811	0.848114	0.434206	0.157783
0.073060	0.999992	0.999683	0.744219	0.513717	0.276485
0.066842	1.000000	0.998505	0.728510	0.530695	0.325589
0.061600	0.997382	0.997091	0.791986	0.617565	0.339708
0.057120	0.905471	0.998033	0.786705	0.782748	0.447868
0.053247	0.996364	0.999879	0.828110	0.778513	0.480252
0.049867	1.000000	0.999212	0.847870	0.804497	0.546471
0.046889	1.000000	0.997156	0.839786	0.750308	0.640759
0.044248	1.000000	0.998605	0.866062	0.808401	0.655806
0.041888	1.000000	0.999934	0.811691	0.854564	0.646670
0.039767	0.992019	0.990866	0.929010	0.810724	0.686172
0.037851	1.000000	0.999225	0.966123	0.749307	0.784478
0.036110	0.999276	0.999999	0.957261	0.873823	0.770686
0.034523	1.000000	0.942150	0.994536	0.849957	0.778583

conditions is different from that observed in table 3.9. As with the results from the one-to-one pairing of RBFs and datapoints, there is a certain RBF radius for which the system can not perform at all but beyond that point, observe that as the radius decreases, the error goes back up to a plateau of 1.0 rather than converging on 0.0; the best results do not come from using the smallest radius. In fact, it comes from using the largest radius possible that yields converging coefficients. To see why this is so, observe that as the radius shrinks, each RBF encompasses less space and therefore more datapoints fall outside the area covered by the set of RBFs; when this happens, the output for each datapoint not under an RBF falls to $(0,0)$ and since $(0,0)$ is 1 unit away from the unit circle, the error rises to 1.0.

Figure 3.19 shows the results after normalizing 784 randomly distributed datapoints using hyperbolic secant RBFs with an area of approximately 0.125; 90 iterations were performed to achieve an average distance from the norm of 0.102286. Figure 3.20 shows the results after normalizing 400 datapoints with an RBF area of 0.210; 120 iterations were performed to achieve an average distance of 0.063492. Note that most of the points cluster closely around the unit circle but a few points are well within the unit circle; this is because these stray points were originally outside the effective radius of the RBFs.

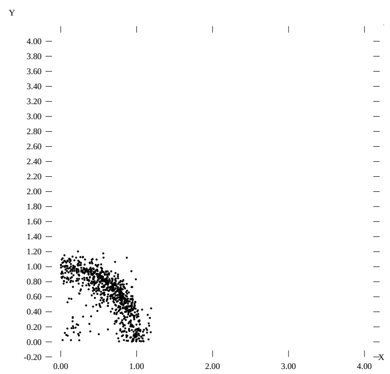

Figure 3.19: Normalization of 784 points with RBFs of size 0.125

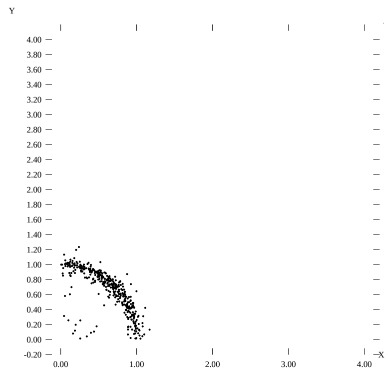

Figure 3.20: Normalization of 400 points with RBFs of size 0.210

Since we can not expect that the number of RBFs is the same as the number of datapoints in a biological system, we also tried scenarios in which the RBF and datapoint densities are dissimilar. For this test, we held the RBF density constant and varied the size of the training set. Table 3.11 shows the results when the training set size is one quarter that of the number of RBFs while table 3.12 shows the results when the training set size is four times that of the number of RBFs. Observe that as the number of datapoints increases relative to the number of RBFs, the results degrade. For example, from table 3.11, we can see that for an RBF density of 81, the best result is 0.033141 and is achieved with an RBF size of 0.136591. Compare this with the result of 0.211457 achieved with an RBF size of 0.080554 in table 3.12. To explain this, observe that the bigger the training set, the more narrow the radius must be for convergence to occur, but a smaller radius also means that more datapoints fall outside the scope of the RBFs which brings the error closer to the plateau of 1.0. We can conclude from this that the best results occur when there are many more RBFs than datapoints; in this situation, the radius must also decrease as the number of datapoints increase, but the results are very good, almost comparable to those from table 3.9. Because of the very large number of neurons available in our brain, the requirement that there be a great number of RBFs does not render this kind of system biologically implausible.

Table 3.11: Avg. Distance; Sparse Datapoints (GD)

RBF Size	RBF Density per Unit Square (#datapoints = 1/4 #RBFs)				
	1	9	25	49	81
3.141593	0.006020	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
1.047198	0.280617	40.965201	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.628319	0.598732	0.000031	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.448799	0.633896	0.005208	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.349066	0.749021	0.036210	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.285599	0.999406	0.032851	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.241661	0.992879	0.051721	0.020298	$\gg$ 100.0	$\gg$ 100.0
0.209440	0.999851	0.107820	0.046324	$\gg$ 100.0	$\gg$ 100.0
0.184800	1.000000	0.497698	0.045904	0.092935	$\gg$ 100.0
0.165347	0.999567	0.685061	0.103924	0.010599	$\gg$ 100.0
0.149600	0.999988	0.816787	0.370730	0.129551	11.209376
0.136591	0.999289	0.788369	0.400450	0.114321	0.033141
0.125664	0.993290	0.958561	0.398394	0.165230	0.060594
0.116355	1.000000	0.977151	0.400005	0.075890	0.080611
0.108331	0.964956	0.861130	0.459132	0.210667	0.063088
0.101342	1.000000	0.848938	0.506530	0.319335	0.040276
0.089760	1.000000	0.999531	0.811467	0.358986	0.213030
0.080554	1.000000	0.999888	0.553055	0.454255	0.209745
0.073060	1.000000	0.870709	0.753376	0.487350	0.272021
0.066842	0.927065	1.000000	0.983827	0.478598	0.304904
0.061600	1.000000	1.000000	0.980569	0.595129	0.477916
0.057120	0.999929	0.735975	0.828426	0.695648	0.553735
0.053247	0.999993	0.999903	0.739146	0.781372	0.573516
0.049867	1.000000	1.000000	0.858958	0.619213	0.720302
0.046889	1.000000	1.000000	0.980409	0.802810	0.643918
0.044248	1.000000	1.000000	0.996733	0.829500	0.646087
0.041888	0.932818	0.972700	0.997930	0.687068	0.751895
0.039767	0.993789	1.000000	0.992228	0.847344	0.763616
0.037851	1.000000	1.000000	0.839456	0.860446	0.730184
0.036110	1.000000	0.999923	0.999998	0.910916	0.647340
0.034523	1.000000	0.998754	0.999970	0.819708	0.681592

Table 3.12: Avg. Distance; Dense Datapoints (GD)

RBF Size	RBF Density per Unit Square (#datapoints = 4× #RBFs)				
	1	9	25	49	81
3.141593	0.192273	≫ 100.0	≫ 100.0	≫ 100.0	≫ 100.0
1.047198	0.273803	≫ 100.0	≫ 100.0	≫ 100.0	≫ 100.0
0.628319	0.682277	≫ 100.0	≫ 100.0	≫ 100.0	≫ 100.0
0.448799	0.759867	≫ 100.0	≫ 100.0	≫ 100.0	≫ 100.0
0.349066	0.789509	84.847605	≫ 100.0	≫ 100.0	≫ 100.0
0.285599	0.839990	0.359352	≫ 100.0	≫ 100.0	≫ 100.0
0.241661	0.899958	0.250348	≫ 100.0	≫ 100.0	≫ 100.0
0.209440	0.902887	0.541458	≫ 100.0	≫ 100.0	≫ 100.0
0.184800	0.949800	0.495078	≫ 100.0	≫ 100.0	≫ 100.0
0.165347	0.989904	0.758578	≫ 100.0	≫ 100.0	≫ 100.0
0.149600	1.000000	0.645944	≫ 100.0	≫ 100.0	≫ 100.0
0.136591	0.948903	0.674502	0.224384	≫ 100.0	≫ 100.0
0.125664	1.000000	0.755011	0.340858	≫ 100.0	≫ 100.0
0.116355	1.000000	0.696787	0.418186	≫ 100.0	≫ 100.0
0.108331	0.993143	0.939997	0.431925	0.237182	≫ 100.0
0.101342	0.999998	0.713432	0.541250	0.237199	≫ 100.0
0.089760	1.000000	0.916187	0.524881	0.258986	≫ 100.0
0.080554	0.999675	0.867840	0.637373	0.373278	0.211457
0.073060	1.000000	0.931711	0.718571	0.460973	0.230125
0.066842	0.992763	0.774020	0.785492	0.522954	0.2843171
0.061600	0.998390	0.954066	0.929148	0.551456	0.319915
0.057120	1.000000	0.997790	0.785241	0.604047	0.426802
0.053247	1.000000	0.997862	0.804855	0.661795	0.508049
0.049867	0.999671	0.958025	0.849946	0.727049	0.523732
0.046889	1.000000	0.915274	0.877855	0.758197	0.630438
0.044248	0.992605	0.985373	0.870856	0.798199	0.644090
0.041888	1.000000	0.944697	0.920018	0.827639	0.655657
0.039767	0.917658	0.919793	0.887087	0.813776	0.695221
0.037851	1.000000	0.998225	0.909938	0.849675	0.741661
0.036110	1.000000	0.991834	0.897550	0.858947	0.733878
0.034523	1.000000	0.997986	0.924970	0.845208	0.792931

An RBF, of course, comes in many shapes. What we have been using so far is of the bell-shaped variety, a hyperbolic secant; another bell-shaped curve is the Gaussian. These curves gradually taper off to zero as the distance from the center increases. Another kind of continuous curve that can be used as an RBF is the dome-shaped variety which can be represented by sections of cosine or parabolic curves; these curves do not taper off to zero; after a certain distance from its center, the function returns exactly zero. Table 3.13 shows the results when the RBF is the parabolic function $f(x) = -rx^2 + 1.0$ where r is the radius parameter, x is the distance of the datapoint to the RBF center, and $f(x) = \max(f(x), 0.0)$. Although the dynamics from using a dome-shaped curve are the same compared to that from using a bell-shaped curve, there seems to be slightly worse results here. This suggests that the tapering effect of bell-shaped curves has a minor but positive effect on system performance. A more severe version of the dome-shaped curve is the non-continuous RBF shaped like a tophat; within a certain radius, the tophat produces 1.0, else the output is 0.0. Table 3.14 shows the results from using tophat RBFs. Note that although the dynamics are the same, the results are worse than those found in tables 3.10 and 3.13. This can be attributed to the non-continuity of a tophat; when the radius is too large, they overlap too much, but when the radius is small, they leave gaps around the tophats that are not at all covered. As stated previously, though, we can expect many more RBFs than there are datapoints. And just as when using continuous curves, the results become markedly better when this is also the case for the discontinuous tophat RBF; table 3.15 shows the results. This implies that this system learns the given training set well as long as there is a sufficiently dense coverage of the space by RBFs, regardless of the exact shape of the RBFs. This is good news biologically since it means the precise shape of the RBFs implemented by the neurons is not that relevant.

We also wanted to see how well this sytem can generalize; that is, after learning the training set, how well does the system normalize a new set of random datapoints? For our purposes, 700 random datapoints were generated, again within the space

Table 3.13: Avg. Distance; Dome-Shaped Parabolic Curve as RBF (GD)

RBF Size	Data and RBF Density per Unit Square				
	1	9	25	49	81
1.333333	0.210310	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.769800	0.520761	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.596285	0.489260	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.503953	0.847865	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.444444	1.000000	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.402015	0.893081	38.655709	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.369800	0.995763	0.352933	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.344265	0.999284	0.245853	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.305888	1.000000	0.405791	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.271819	1.000000	0.340854	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.256600	1.000000	0.388251	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.239474	0.908963	0.587670	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.225374	0.992768	0.692731	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.213504	0.999923	0.672812	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.203331	1.000000	0.732926	0.206516	$\gg$ 100.0	$\gg$ 100.0
0.194487	0.957807	0.780208	0.234838	$\gg$ 100.0	$\gg$ 100.0
0.186704	1.000000	0.710729	0.259876	$\gg$ 100.0	$\gg$ 100.0
0.179787	1.000000	0.681744	0.242865	$\gg$ 100.0	$\gg$ 100.0
0.173585	1.000000	0.744143	0.319474	$\gg$ 100.0	$\gg$ 100.0
0.167984	1.000000	0.608323	0.234542	$\gg$ 100.0	$\gg$ 100.0
0.162893	0.999926	0.690025	0.326621	$\gg$ 100.0	$\gg$ 100.0
0.158238	0.986938	0.755613	0.358634	$\gg$ 100.0	$\gg$ 100.0
0.153960	0.992674	0.757161	0.400718	0.232836	$\gg$ 100.0
0.150012	1.000000	0.682452	0.395587	0.201863	$\gg$ 100.0
0.146352	1.000000	0.809338	0.385795	0.286538	$\gg$ 100.0
0.142948	1.000000	0.762548	0.465068	0.342876	$\gg$ 100.0
0.139771	1.000000	0.742637	0.536501	0.372591	$\gg$ 100.0
0.136797	1.000000	0.878313	0.463885	0.448410	$\gg$ 100.0
0.134005	1.000000	0.857329	0.721972	0.330591	0.250386

Table 3.14: Avg. Distance; Non-Continuous Tophat as RBF (GD)

RBF	Data and RBF Density per Unit Square				
Size	1	9	25	49	81
1.00	0.573277	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.96	0.592100	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.92	0.610322	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.88	0.679210	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.84	0.758391	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.80	0.659745	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.76	0.653829	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.72	0.690923	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.68	0.682611	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.64	0.590127	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.60	0.792033	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.56	0.782381	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.52	0.806553	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.48	0.782891	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.44	0.853527	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.40	0.906430	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.36	1.000000	0.673818	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.32	0.982791	0.759021	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.28	0.999792	0.699807	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.24	0.872094	0.717721	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.20	0.910026	0.894458	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.16	1.000000	1.000000	0.723398	$\gg$ 100.0	$\gg$ 100.0
0.12	1.000000	0.912810	0.993067	$\gg$ 100.0	$\gg$ 100.0
0.08	1.000000	0.878296	0.897731	0.836735	$\gg$ 100.0
0.04	1.000000	1.000000	0.921863	0.959184	0.938272

Table 3.15: Avg. Distance; Non-Continuous Tophats; Sparse Datapoints (GD)

RBF Size	Data and RBF Density per Unit Square (#datapoints = 1/4 #RBFs)				
	1	9	25	49	81
1.00	0.259187	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.96	0.284733	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.92	0.409210	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.88	0.395921	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.84	0.201066	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.80	0.528050	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.76	0.683122	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.72	0.740019	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.68	0.802683	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.64	0.753987	0.382197	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.60	0.993173	0.309892	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.56	0.983208	0.578302	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.52	0.903237	0.509012	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.48	0.998657	0.749930	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.44	0.984572	0.793201	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.40	0.994997	0.897320	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.36	0.979721	0.861086	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.32	0.902838	0.947718	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.28	0.985776	0.924781	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.24	0.937815	0.873928	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.20	0.859812	0.967100	0.392713	$\gg$ 100.0	$\gg$ 100.0
0.16	0.909974	0.912974	0.609388	0.360293	$\gg$ 100.0
0.12	0.927876	0.998953	0.893201	0.658094	0.409009
0.08	0.947850	0.997216	0.850926	0.838337	0.782136
0.04	0.930156	0.967701	0.890021	0.855737	0.849021

discussed previously. From table 3.9, we can see that the results are worse than that from learning a training set of random datapoints; but because the increase in error is generally small, this system's generalizing ability is not much worse than its learning ability. Furthermore, the effect is the same; for some fixed data and RBF density, there is a threshold radius level such that if the RBF radius is bigger than the threshold, the coefficients will not converge. This means that the results from learning the training set is a good indicator of the overall system performance. Another factor that we checked was whether the training set makes a difference in how well the system generalizes. Generalizing ability was compared after learning two kinds of training sets: evenly spaced, non-random datapoints and random datapoints. Recall that the results and dynamics from these two sets (as seen in tables 3.9 and 3.10) differ. Because of this, we expected the generalizing ability to also differ, which is verified, but the difference, again, is small; generalizing after learning random datapoints leads to slightly worse results than after learning non-random datapoints. Tables 3.16 and 3.17 show the results of normalizing 700 new datapoints after learning a non-random and random training set, respectively.

We also determined whether the system can generalize outside the space. This is a question of extrapolation rather than the interpolation discussed so far. The results are very poor, which is expected since the RBFs are localized; the system has no idea what to do in the space that has no training datapoints. This may not be desirable when trying to approximate a function mathematically, but in biological terms, this system mimics what we do very well; given a space of knowledge, humans can often make educated guesses on questions still within that space, but questions beyond the scope of that space are not easy to answer. One such example would be the classification of objects into two groups: living and nonliving. We can perform this task well because we are aware of our space, the earth, in which carbon is the basis of life. But on a planet with silicon chemistry, for example, we would have trouble identifying living objects.

Table 3.16: Generalizing Ability after Learning Orderly Training Set (GD)

RBF Size	Data and RBF Density per Unit Square				
	1	9	25	49	81
3.141593	0.172085	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
1.047198	0.332677	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.628319	0.644026	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.448799	0.802462	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.349066	0.882684	0.206995	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.285599	0.917347	0.354097	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.241661	0.946113	0.482922	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.209440	0.961620	0.592234	6.840577	$\gg$ 100.0	$\gg$ 100.0
0.184800	0.969011	0.670094	0.209778	$\gg$ 100.0	$\gg$ 100.0
0.165347	0.972910	0.725290	0.364419	$\gg$ 100.0	$\gg$ 100.0
0.149600	0.976634	0.773776	0.444581	62.381113	$\gg$ 100.0
0.136591	0.978293	0.813395	0.514340	0.198455	$\gg$ 100.0
0.125664	0.988188	0.842553	0.577475	0.304843	$\gg$ 100.0
0.116355	0.987844	0.866596	0.618883	0.366707	$\gg$ 100.0
0.108331	0.987239	0.882227	0.665009	0.425548	0.166975
0.101342	0.991582	0.895009	0.715414	0.476277	0.274959
0.089760	0.989969	0.928129	0.774899	0.567878	0.374158
0.080554	0.991324	0.931841	0.812112	0.642896	0.465881
0.073060	0.995220	0.950822	0.840002	0.705193	0.526047
0.066842	0.995499	0.953167	0.860203	0.732823	0.595525
0.061600	0.994726	0.964367	0.883046	0.778391	0.651593
0.057120	0.994989	0.970330	0.910274	0.826326	0.698376
0.053247	0.998681	0.969909	0.916968	0.831132	0.735673
0.049867	0.996921	0.978982	0.920475	0.856326	0.771791
0.046889	0.997003	0.976900	0.938570	0.880481	0.797050
0.044248	0.999357	0.980842	0.943983	0.894840	0.820758
0.041888	0.999240	0.980595	0.947339	0.903732	0.826088
0.039767	0.998230	0.984898	0.959518	0.909633	0.847160
0.037851	0.997113	0.983540	0.961274	0.928250	0.869632
0.036110	0.999454	0.984118	0.951775	0.924767	0.882409
0.034523	0.999715	0.985945	0.965307	0.930425	0.879856
0.033069	0.999901	0.990241	0.971522	0.931398	0.903861
0.031733	0.998097	0.990831	0.975604	0.943207	0.899817

Table 3.17: Generalizing Ability after Learning Random Training Set (GD)

RBF Size	Data and RBF Density per Unit Square				
	1	9	25	49	81
3.141593	0.247238	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
1.047198	0.319501	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.628319	0.512602	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.448799	0.940111	0.252916	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.349066	0.962748	0.379411	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.285599	0.984730	0.417182	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.241661	0.968443	0.568699	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.209440	0.998245	0.549267	96.867926	$\gg$ 100.0	$\gg$ 100.0
0.184800	0.974068	0.769626	0.227427	$\gg$ 100.0	$\gg$ 100.0
0.165347	0.993693	0.724367	0.262470	$\gg$ 100.0	$\gg$ 100.0
0.149600	0.999884	0.719392	0.396981	$\gg$ 100.0	$\gg$ 100.0
0.136591	0.999693	0.810913	0.408789	0.249505	$\gg$ 100.0
0.125664	0.999969	0.914423	0.570443	0.255015	$\gg$ 100.0
0.116355	0.992998	0.915640	0.554589	0.317466	$\gg$ 100.0
0.108331	0.994277	0.957376	0.605159	0.401069	0.251291
0.101342	0.999981	0.923915	0.661384	0.412250	0.272915
0.089760	0.999271	0.978646	0.752900	0.468958	0.331840
0.080554	0.998499	0.987398	0.845540	0.646955	0.369919
0.073060	0.999872	0.988924	0.873886	0.632853	0.424494
0.066842	0.998997	0.969706	0.923500	0.736217	0.487221
0.061600	0.992891	0.992530	0.933056	0.803169	0.614758
0.057120	0.999987	0.995155	0.940586	0.828989	0.639132
0.053247	0.919954	0.989366	0.950693	0.834180	0.689319
0.049867	0.909219	0.991962	0.967960	0.879122	0.764110
0.046889	0.997981	0.936429	0.968082	0.914182	0.799764
0.044248	0.999273	0.995694	0.981558	0.905131	0.787154
0.041888	0.999921	0.996073	0.968929	0.927195	0.862526
0.039767	1.000000	0.999970	0.983098	0.961680	0.871783
0.037851	0.962100	0.984669	0.987169	0.958644	0.871093
0.036110	0.997385	0.999970	0.988567	0.968756	0.905199
0.034523	0.999891	0.959998	0.995174	0.963864	0.931790
0.033069	0.991348	0.996989	0.993386	0.971499	0.932542
0.031733	0.999992	0.999994	0.997399	0.976486	0.948100

Another factor with a big role in the initial system conditions is the setting of the coefficients; mathematically, as discussed in section 2.4.2, the initial $\mathbf{c}$ vector is set to $\mathbf{a}$, but biologically it is unrealistic to assume $\mathbf{c}$ can be so precisely initialized. Therefore we began by varying the initial coefficient values within a certain percentage of $\mathbf{a}$; the results are shown in table 3.18[10]. Observe that there is no increase in the average distance as the percent deviation goes up; this suggests that the coefficients do not have to be perfectly initialized to $\mathbf{a}$, the reason being that the gradient descent process will still find the best final set of coefficients regardless of their initial values. Gradient descent compensates for the imperfect coefficients; this suggests that without some form of gradient descent, as with the matched filter system or the layered network system that follows, the initial coefficient values will have more of an impact on performance.

Table 3.18: Results from Non-Ideal Initial Coefficients, % Deviation (GD)

Bound	0%	10%	20%	30%
Avg. Distance	0.111554	0.110027	0.099253	0.109673
Bound	40%	50%	100%	200%
Avg. Distance	0.130525	0.121296	0.084775	0.115836

Initial conditions are a major factor in determining the performance of the system and with a perfect system, that may be the only factor. However, the actual process in calculating the final coefficients are also important here since biological systems are incapable of high-precision calculations that is a hallmark of modern Von Neumann computers. To check the effect of low-precision mathematics, the results from the coefficient calculations were rounded to various degrees of precision. From table 3.19, observe that even with only one digit of precision, the results did not degrade substantially; only with integer mathematics are the results seriously affected. Since there is

[10]The training set used was composed of 81 random datapoints and 81 RBFs per unit square, randomly distributed within the space. The RBF size was set to 0.089760, which gave the best result in table 3.10.

evidence that our neurons may be capable of calculations of at best one decimal place [19], these results suggest that this system still performs very well under biological conditions.

Table 3.19: Results from Low-Precision Calculations (GD)

Precision	1	.1	.01	.001
Avg. Distance	1.586329	0.233124	0.134481	0.105137

Again, in a perfect system, once the coefficients are determined, they will not be changed. But here, we need to determine the effect on performance when the coefficients are corrupted after learning the training set. This is akin to having a beer after finals to facilitate a memory coredump. Note from table 3.20 that the results become progressively worse as the coefficients are increasingly corrupted; the final coefficients can be altered only slightly before the results are severely affected. But if we limit the amount of variance to 10%, then performance degradation is nominal.

Table 3.20: Results from Corrupting Final Coefficients, % Deviation (GD)

Bound	0%	10%	20%	30%
Avg. Distance	0.091554	0.122758	0.171681	0.221193
Bound	40%	50%	100%	200%
Avg. Distance	0.321989	0.332871	0.572311	0.947156

One final point, since this system uses gradient descent to determine the coefficients, is the number of iterations required. The results in this section come from using coefficients determined with 150 iterations. This is better than the average number of iterations required with Backpropagation, 300. If the coefficients are ideally set, then it is possible to perform as few as 20 iterations and still achieve the same results. As stated earlier, gradient descent compensates for non-ideal initial coefficients, but the price is the increase in number of iterations.

3.4 Learning with an RBF-based Layered Network System

3.4.1 Setup and Procedures

Since this system is constructed for supervised learning of spatial patterns like the previous system in section 3.3, we again decided to learn a mapping that normalizes a 2-element vector with the same setups and procedures.

Most of the results came from using the bell-shaped hyperbolic secant function as the RBF style, however other RBF styles were also investigated.

3.4.2 Results

This system's initial condition is determined by the following factors: the RBF's radius, the RBF and data density, the shape of the RBFs, and the initial coefficients. Other biologically important concerns include this system's performance when calculations are not accurate and when the coefficients are corrupted after the learning phase.

Given an adequate number of RBFs, each with an appropriate radius, this system can learn any given training set very well. Placing one RBF on each datapoint yields the best results when the radius is smaller than some threshold. This threshold is determined by the number of datapoints. From table 3.21, observe that the average distance gets smaller as the radius shrinks and unlike the results in table 3.9, there is no sudden decline in the error. This difference is due to the algorithm itself; the RBF-based gradient descent network will converge on a set of coefficients if there is one, otherwise repeated iterations of the gradient descent algorithm will actually cause the error to increase, hence the sharp drop in error. This algorithm, however, is simpler; the decline in error is not due to convergence. Instead, the decline here is due to the decrease in data redundancy. Note that each weight is calculated by a straightforward summation over all the input data: $C_{ki} = \sum_j \zeta_i^j v_k(\phi^j)$; multiple datapoints falling under the same RBF causes C_{ki} to increase. Then note that the output of each node i is a summation over all the RBFs: $w_i = \sum_k C_{ki} v_k(\phi)$; a point falling under more than one RBF will contribute multiple times to the output. Therfore, the decline in error is proportional to the decrease in RBF size.

Table 3.21: Avg. Distance; Exactly One RBF on Each Datapoint (LN)

RBF Size	Data and RBF Density per Unit Square				
	1	9	25	49	81
3.141593	5.290096	50.881003	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
1.047198	0.409258	15.068245	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.628319	0.047191	4.515605	37.691801	$\gg$ 100.0	$\gg$ 100.0
0.448799	0.006112	1.613998	14.206618	56.022942	$\gg$ 100.0
0.349066	0.000814	0.666396	6.237606	25.023267	69.310942
0.285599	0.000109	0.301376	3.077626	12.557649	35.082870
0.241661	0.000015	0.143587	1.656776	6.884973	19.403919
0.209440	0.000002	0.070432	0.950250	4.043690	11.492171
0.184800	0.000000	0.035125	0.570110	2.506821	7.186788
0.165347	0.000000	0.017689	0.352980	1.621371	4.697014
0.149600	0.000000	0.008962	0.223380	1.083925	3.182762
0.136591	0.000000	0.004558	0.143533	0.743421	2.221724
0.125664	0.000000	0.002324	0.093218	0.520039	1.589188
0.116355	0.000000	0.001187	0.061002	0.369333	1.159693
0.108331	0.000000	0.000607	0.040139	0.265371	0.860206
0.101342	0.000000	0.000311	0.026517	0.192389	0.646606
0.095200	0.000000	0.000159	0.017570	0.140448	0.491334

In a biological system, of course, we can not expect to know the input beforehand, therefore there is no way to determine where to exactly place each RBF; the best solution in this situation is to simply spread the RBFs through the space as evenly as possible. From table 3.22, observe that the dynamics under such conditions is radically different from those when an RBF is placed directly over a datapoint. For any given combination of datapoint and RBF density, there is a particular radius that produces the smallest average distance; the error rises to 1.0 as the radius shrinks

Table 3.22: Avg. Distance; Same Number of RBFs and Datapoints (LN)

RBF Size	Data and RBF Density per Unit Square				
	1	9	25	49	81
3.141593	6.368974	50.878679	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
1.047198	0.427894	15.623404	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.628319	0.761092	4.085713	45.739342	$\gg$ 100.0	$\gg$ 100.0
0.448799	0.816377	1.324585	12.293943	53.960860	$\gg$ 100.0
0.349066	0.899539	0.643931	7.434243	27.320921	81.494076
0.285599	0.929736	0.381419	2.606401	12.992955	38.095194
0.241661	0.957110	0.713688	1.310500	7.563757	22.244174
0.209440	0.948941	0.806297	0.704134	4.381490	11.101796
0.184800	0.964315	0.895599	0.395062	2.950385	7.618021
0.165347	0.989331	0.892571	0.479839	1.426841	4.884510
0.149600	0.999217	0.871737	0.611587	1.233429	3.128034
0.136591	0.999683	0.882959	0.716740	0.397551	1.850447
0.125664	0.999997	0.893559	0.839354	0.457356	1.387092
0.116355	0.985820	0.995828	0.815616	0.434321	0.806538
0.108331	0.994782	0.934262	0.804016	0.553426	0.708163
0.101342	0.947827	0.994079	0.841954	0.674115	0.382817
0.089760	0.994986	0.998014	0.938704	0.675929	0.478131
0.080554	1.000000	0.998211	0.908523	0.830791	0.555313
0.073060	1.000000	0.904861	0.878588	0.846973	0.675883
0.066842	1.000000	0.999337	0.974934	0.910520	0.742145
0.061600	0.993672	0.999865	0.989442	0.873019	0.805066
0.057120	1.000000	0.999976	0.914943	0.924736	0.842128
0.053247	0.999921	0.991570	0.981030	0.940763	0.868722
0.049867	1.000000	0.958053	0.914197	0.931658	0.889193
0.046889	1.000000	0.996367	0.956738	0.973082	0.870009
0.044248	1.000000	0.998685	0.979753	0.960707	0.899458
0.041888	1.000000	0.997298	0.975020	0.971879	0.941286
0.039767	0.999970	0.999983	0.965388	0.977418	0.921499
0.037851	1.000000	0.997167	0.959671	0.989632	0.952797
0.036110	1.000000	0.999684	0.998285	0.978607	0.971319
0.034523	1.000000	0.999894	0.999516	0.935055	0.966206

below that radius. Like the RBF-based gradient descent system, this is because as the RBFs shrink in size, more datapoints fall outside the effective radius of the RBFs, and the output for each of these points falls to $(0,0)$. Note that the smallest error achieved here is quite large relative to that in table 3.10; this system does not iterate to converge on a set of weights and so is not capable of fine-tuning the weights. We ran the same test again with a density of 225 and 1600 to see if the number of datapoints and RBFs affect the outcome, however the results were the same; the only difference was that a much smaller radius was required (smallest average distance was 0.382817 with the RBF size set to 0.101342).

To determine the effect when the amount of data in the training set is not dependent on the number of RBFs, we tested various training set sizes. Tables 3.23 and 3.24 show the results when the number of datapoints is one tenth and four times that of the RBFs, respectively. Observe that the average distance declines as the data gets sparser, implying the need for a great many RBFs. Also observe that the smallest average distance is about the same, a feature also seen in table 3.22. This suggests that the smallest average error is dependent on the ratio between the training set size and the RBF density; we should be able to predict how well a large system will perform by running simulations on smaller systems.

So far, we have used the hyperbolic secant function for the RBFs. Now we will determine whether the shape of the function has any effect on this system's performance. First, we will use the same parabolic RBF as for the previous system; the results in table 3.25 are slightly worse than in table 3.22 where the RBFs are bell-shaped. Nevertheless, the dynamics in both tables are identical. Table 3.26 shows the results when tophat RBFs are used in this system; in this case, the results are even worse though, again, the dynamics are the same. And just as with the other RBF shapes, these results can be improved if the system has many more RBFs than datapoints.

Another concern is this system's generalizing ability. To test this, 700 new random datapoints were generated and input into the system after learning was complete.

Table 3.23: Avg. Distance; Sparse Datapoints (LN)

RBF Size	RBF Density per Unit Square #datapoints = 1/10 #RBFs				
	1	9	25	49	81
3.141593	4.743232	$\gg$ 100.0	31.482728	$\gg$ 100.0	$\gg$ 100.0
1.047198	0.337311	10.667625	12.517336	45.266990	$\gg$ 100.0
0.628319	0.782807	1.818952	6.629492	20.141052	34.667826
0.448799	0.832261	0.299623	0.826027	6.840022	12.554110
0.349066	0.891236	0.444341	0.308449	2.179697	8.337990
0.285599	0.934431	0.644164	0.249026	1.476600	3.377912
0.241661	0.927397	0.775815	0.387128	0.685391	2.015777
0.209440	0.953836	0.769049	0.439243	0.312998	1.605834
0.184800	0.969993	0.854438	0.502909	0.268008	1.048570
0.165347	0.927690	0.914417	0.629099	0.359001	0.513604
0.149600	0.969964	0.826569	0.660163	0.396469	0.347646
0.136591	0.992192	0.906374	0.729591	0.478543	0.273068
0.125664	0.990089	0.952748	0.744799	0.552560	0.336216
0.116355	0.999971	0.974961	0.873213	0.628081	0.425326
0.108331	0.998266	0.977988	0.901867	0.666454	0.462263
0.101342	0.999038	0.980929	0.800276	0.700011	0.494809
0.089760	1.000000	0.989386	0.996564	0.784478	0.561283
0.080554	0.993902	0.955883	0.998180	0.886814	0.587751
0.073060	1.000000	0.966884	0.999892	0.857289	0.650898
0.066842	1.000000	0.996736	0.999701	0.770902	0.667543
0.061600	0.999982	0.956185	0.999935	0.897954	0.675779
0.057120	1.000000	0.999200	0.999998	0.943542	0.736252
0.053247	1.000000	0.997895	0.994712	0.984713	0.711049
0.049867	1.000000	0.998054	0.999949	0.958849	0.740455
0.046889	1.000000	0.999693	0.999956	0.959477	0.782511
0.044248	1.000000	0.997735	0.999996	0.978847	0.823867
0.041888	1.000000	0.996971	0.999688	0.979685	0.847547
0.039767	0.999267	0.995485	0.692878	0.780526	0.903281
0.037851	0.999989	0.999962	1.000000	0.970726	0.891383
0.036110	1.000000	0.992457	0.999728	0.998501	0.901428
0.034523	0.999913	0.999258	0.999990	0.999535	0.949971

Table 3.24: Avg. Distance; Dense Datapoints (LN)

RBF Size	RBF Density per Unit Square #datapoints = 4× #RBFs				
	1	9	25	49	81
3.141593	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
1.047198	3.503006	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.628319	0.565885	58.185265	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.448799	0.679757	14.528436	57.163646	$\gg$ 100.0	$\gg$ 100.0
0.349066	0.719768	5.149398	23.651731	$\gg$ 100.0	$\gg$ 100.0
0.285599	0.841699	1.899886	12.366269	55.806625	$\gg$ 100.0
0.241661	0.882852	0.680816	6.003423	28.235755	77.223114
0.209440	0.893052	0.523891	3.596154	16.546052	47.483956
0.184800	0.943626	0.608270	2.197020	9.573493	28.896807
0.165347	0.984044	0.725172	1.307762	6.261382	18.150356
0.149600	0.981361	0.767795	0.759107	4.368322	13.223675
0.136591	0.990913	0.840307	0.535480	3.304062	8.910988
0.125664	0.997284	0.877738	0.570167	2.260278	6.553438
0.116355	0.995929	0.888002	0.637209	1.106595	4.532194
0.108331	0.996865	0.900486	0.672783	0.806117	3.358849
0.101342	0.998158	0.930737	0.737428	0.644336	2.332594
0.089760	0.989066	0.941763	0.799109	0.576332	1.363533
0.080554	0.996653	0.972094	0.896420	0.659345	0.755944
0.073060	0.999829	0.975142	0.891704	0.724526	0.560381
0.066842	0.999558	0.978183	0.935827	0.772131	0.548874
0.061600	0.994981	0.975758	0.880535	0.807083	0.637371
0.057120	0.999660	0.987037	0.951763	0.861663	0.724270
0.053247	0.999298	0.989738	0.954151	0.881519	0.764265
0.049867	0.993685	0.988042	0.983256	0.906814	0.800471
0.046889	0.999733	0.990807	0.972232	0.911668	0.838055
0.044248	0.994429	0.986599	0.986603	0.955053	0.862807
0.041888	1.000000	0.990744	0.994430	0.941729	0.871262
0.039767	1.000000	0.991797	0.988943	0.928888	0.901900
0.037851	1.000000	0.995070	0.944352	0.953360	0.899100
0.036110	0.999997	0.995698	0.978064	0.952182	0.911702
0.034523	1.000000	0.996771	0.975679	0.978960	0.941405

Table 3.25: Avg. Distance; Dome-Shaped Parabolic Curve as RBF (LN)

RBF Size	Data and RBF Density per Unit Square				
	1	9	25	49	81
1.333333	0.815712	27.698965	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.769800	0.561600	9.843290	86.366317	$\gg$ 100.0	$\gg$ 100.0
0.596285	0.988170	4.177635	29.040062	$\gg$ 100.0	$\gg$ 100.0
0.503953	0.859253	1.838441	19.314525	67.848530	$\gg$ 100.0
0.444444	1.000000	1.220628	13.534756	46.540004	$\gg$ 100.0
0.402015	1.000000	1.178876	9.465708	27.798876	87.084313
0.369800	0.659905	0.532106	7.068043	27.139060	61.861433
0.344265	1.000000	0.432334	6.962917	17.773813	55.416132
0.323381	0.967630	0.504890	2.653493	18.281093	46.978054
0.305888	0.996142	0.590163	2.481672	14.876339	35.496712
0.290957	1.000000	0.672651	2.366086	11.139417	27.538434
0.278019	1.000000	0.702193	1.618637	8.229012	20.278875
0.239474	1.000000	0.725467	1.073878	4.754893	18.055768
0.225374	1.000000	0.821660	0.567048	3.423916	12.046598
0.203331	1.000000	0.810238	0.468830	2.202537	7.190466
0.186704	1.000000	0.826210	0.432642	1.626811	4.660955
0.167984	1.000000	0.834137	0.510453	1.035331	3.107422
0.158238	1.000000	0.837901	0.570990	1.036211	2.747619
0.153960	1.000000	0.844150	0.592609	0.652016	2.480122
0.142134	1.000000	0.853364	0.610201	0.571401	2.045945
0.139771	1.000000	0.862967	0.672235	0.576057	1.356214
0.136797	1.000000	0.851749	0.779868	0.543506	1.348711
0.131377	1.000000	0.843520	0.790542	0.429000	1.077623
0.126554	1.000000	0.866819	0.785503	0.516414	0.980116
0.124334	1.000000	0.901077	0.842896	0.581479	0.997719
0.114755	1.000000	0.905690	0.869030	0.656935	0.602827
0.111499	1.000000	0.929676	0.913172	0.725776	0.573122
0.108505	1.000000	0.996692	0.894840	0.719006	0.568649
0.099658	1.000000	0.951909	0.889349	0.715228	0.441933
0.098563	1.000000	0.964067	0.898300	0.720211	0.460860

Table 3.26: Avg. Distance; Non-Continuous Tophat as RBF (LN)

RBF	Data and RBF Density per Unit Square				
Size	1	9	25	49	81
1.00	1.736860	22.665302	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.96	0.854802	22.667286	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.92	0.730385	11.663990	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.88	0.719053	17.081457	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.84	0.682219	18.838457	82.964344	$\gg$ 100.0	$\gg$ 100.0
0.80	0.762904	8.813061	65.391568	$\gg$ 100.0	$\gg$ 100.0
0.76	0.789980	13.103902	82.408346	$\gg$ 100.0	$\gg$ 100.0
0.72	0.806735	11.838954	54.348222	$\gg$ 100.0	$\gg$ 100.0
0.68	0.859002	8.892142	55.279833	$\gg$ 100.0	$\gg$ 100.0
0.64	0.849833	6.855438	40.656637	$\gg$ 100.0	$\gg$ 100.0
0.60	0.858929	5.092243	34.848491	$\gg$ 100.0	$\gg$ 100.0
0.56	0.885766	4.958199	30.697116	89.814972	$\gg$ 100.0
0.52	0.857802	1.333069	25.289582	70.957777	$\gg$ 100.0
0.48	0.892673	1.107563	14.892878	55.675699	$\gg$ 100.0
0.44	0.908932	1.107725	10.847541	47.198221	$\gg$ 100.0
0.40	0.890487	1.173167	6.772998	22.222141	63.777970
0.36	0.899845	0.795342	7.995779	21.149061	55.431791
0.32	0.943889	0.739942	5.871549	13.203265	33.499429
0.28	0.967863	0.706664	3.065139	8.121805	23.367339
0.24	0.960911	0.751111	1.396599	5.141499	12.878041
0.20	0.997601	0.786667	0.689770	2.421772	7.195510
0.16	0.987993	0.824444	0.764882	0.679388	1.369245
0.12	0.979736	0.884444	0.770021	0.816315	1.197061
0.08	0.978367	0.942290	0.831008	0.959184	0.720105
0.04	0.990446	0.969332	0.947651	0.979592	0.901235

Observe that the results after learning a training set composed of evenly spaced, nonrandom points is better than after learning a set of completely random points. Of more significance biologically, however, is that the results after learning a set of random points is equivalent to learning the training set itself; this system generalizes as well as it learns. Note also that this suggests we can predict this system's generalizing ability from how well it learns the training set. Tables 3.27 and 3.28 show the results of this system's attempt at normalizing 700 random datapoints after learning a non-random and random training set, respectively.

Another aspect of generalization is the ability to extrapolate outside the space. Just like the system incorporating gradient descent, this one is a poor extrapolator, again probably due to the localized nature of RBFs.

One last factor contributing to this system's initial condition is the setting of the weights; for the above tests, the weights have been initialized to 0.0. Although we can expect a biological system to try to initialize the weights, we should expect to do so only within some bound of the ideal, therefore we varied the weights; the results are in table 3.29. Observe that the average distance worsens as the weights are increasingly randomized, but when the randomization is within a small bound from ideal, the degradation is negligible.

Noise is a normal part of biological processing and we must also account for it in this system. The effect of noise is to reduce the accuracy of the calculations, which is much like using low-precision arithmetic. To see what would happen to this system in such a situation, the results were rounded to various degrees of precision for every step in the weight calculation. Observe from table 3.30 that the results are noticeably worse only when the system performs integer arithmetic; even with only one digit of precision, there is no observable performance degradation.

One last issue is the effect on the performance if the weights are corrupted after the learning process. Note from table 3.31 that the results get worse as the weights are

Table 3.27: Generalizing Ability after Learning Orderly Training Set (LN)

RBF Size	Data and RBF Density per Unit Square				
	1	9	25	49	81
3.141593	42.872380	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
1.047198	0.709717	97.785730	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.628319	0.351360	14.098922	42.871998	$\gg$ 100.0	$\gg$ 100.0
0.448799	0.592913	3.439239	16.304801	63.026274	$\gg$ 100.0
0.349066	0.647373	0.919674	7.086442	28.126351	75.997607
0.285599	0.818207	0.155006	3.337351	14.079868	38.477595
0.241661	0.833775	0.347113	1.606539	7.564929	21.378473
0.209440	0.852173	0.524406	0.695683	4.315054	12.568841
0.184800	0.907261	0.635818	0.233506	2.489564	7.677659
0.165347	0.962634	0.705589	0.200235	1.436197	4.947618
0.149600	0.975647	0.770229	0.317805	0.808920	3.167531
0.136591	0.979364	0.803418	0.445002	0.389404	2.106955
0.125664	0.979000	0.850246	0.533132	0.166786	1.359789
0.116355	0.987712	0.865577	0.602697	0.195752	0.865898
0.108331	0.986422	0.873928	0.648659	0.285946	0.489636
0.101342	0.993263	0.907797	0.737054	0.459766	0.156696
0.089760	0.990192	0.917368	0.786835	0.586059	0.277442
0.080554	0.992857	0.931297	0.797273	0.620798	0.342833
0.073060	0.993861	0.949492	0.858051	0.690899	0.489389
0.066842	0.991835	0.950675	0.876227	0.752433	0.580789
0.061600	0.996437	0.958782	0.890107	0.787612	0.623061
0.057120	0.998188	0.968031	0.914289	0.812461	0.692384
0.053247	0.996925	0.969911	0.918990	0.826335	0.737347
0.049867	0.998264	0.977116	0.925877	0.855709	0.770830
0.046889	0.998615	0.975967	0.940607	0.876556	0.798499
0.044248	0.998352	0.980040	0.938192	0.879457	0.816260
0.041888	0.996639	0.982015	0.947803	0.907328	0.842868
0.039767	0.996758	0.980614	0.953933	0.908262	0.846604
0.037851	0.997616	0.987553	0.956039	0.917409	0.857886
0.036110	0.998509	0.984898	0.964734	0.930960	0.887058
0.034523	0.998827	0.990816	0.968612	0.930702	0.895446
0.033069	0.999239	0.982290	0.971463	0.940108	0.889224
0.031733	0.998421	0.990504	0.973538	0.949036	0.910869

Table 3.28: Generalizing Ability after Learning Random Training Set (LN)

RBF Size	Data and RBF Density per Unit Square				
	1	9	25	49	81
3.141593	34.779883	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
1.047198	0.400251	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0	$\gg$ 100.0
0.628319	0.848456	13.798153	44.516711	$\gg$ 100.0	$\gg$ 100.0
0.448799	0.960953	3.102647	18.639001	62.492040	$\gg$ 100.0
0.349066	0.994763	0.673431	6.569084	28.109089	75.483194
0.285599	0.992901	0.383777	3.154476	13.672610	34.155043
0.241661	0.996938	0.648713	1.124691	7.188618	20.903109
0.209440	0.998914	0.808538	0.605445	4.199871	12.829876
0.184800	0.999782	0.887489	0.363648	1.818538	7.593532
0.165347	0.999610	0.923094	0.475846	1.147254	4.425116
0.149600	0.998356	0.941093	0.611221	0.551163	2.793390
0.136591	0.999954	0.958390	0.804635	0.455294	1.502856
0.125664	0.999139	0.973878	0.852194	0.375760	1.251397
0.116355	0.999818	0.980637	0.853148	0.553217	0.500732
0.108331	0.999968	0.984766	0.889367	0.619076	0.439083
0.101342	0.999953	0.988608	0.944359	0.731747	0.367644
0.089760	0.999994	0.989281	0.929399	0.770978	0.413826
0.080554	0.999998	0.994574	0.977705	0.882462	0.730311
0.073060	0.999999	0.998254	0.973809	0.923813	0.760504
0.066842	0.999990	0.998231	0.990834	0.937357	0.836721
0.061600	1.000000	0.999374	0.990919	0.955466	0.882631
0.057120	0.999997	0.998690	0.990777	0.963008	0.923245
0.053247	1.000000	0.998613	0.995705	0.975742	0.934813
0.049867	1.000000	0.998733	0.997192	0.970685	0.942147
0.046889	1.000000	0.999813	0.995457	0.984685	0.958392
0.044248	1.000000	0.999628	0.994856	0.992024	0.970765
0.041888	0.999994	0.999623	0.999248	0.993329	0.969183
0.039767	1.000000	0.999940	0.998840	0.993911	0.985029
0.037851	1.000000	0.999880	0.998084	0.997931	0.984017
0.036110	1.000000	0.999599	0.998220	0.997869	0.980495
0.034523	1.000000	0.999951	0.999448	0.995407	0.992569
0.033069	0.999818	0.999713	0.998380	0.996978	0.989853
0.031733	1.000000	0.999931	0.997602	0.996707	0.994073

increasingly corrupted, but the increase in error is slow. This is typical for any neural network in which the weights are corrupted and is indicative of graceful degradation.

Table 3.29: Results from Non-Ideal Initial Coefficients, % Deviation (LN)

Bound	0%	10%	20%	30%
Avg. Distance	0.375921	0.363284	0.391597	0.412426
Bound	40%	50%	100%	200%
Avg. Distance	0.401419	0.405996	0.504766	0.715635

Table 3.30: Results from Low-Precision Calculations (LN)

Precision	1	.1	.01	.001
Avg. Distance	0.610894	0.402674	0.391100	0.384245

Table 3.31: Results from Corrupting Final Coefficients, % Deviation (LN)

Bound	0%	10%	20%	30%
Avg. Distance	0.387018	0.369650	0.403421	0.403199
Bound	40%	50%	100%	200%
Avg. Distance	0.386917	0.402433	0.446788	0.587371

Chapter 4

Conclusion

4.1 Review of System Features

4.1.1 Matched Filter System

As a rule, the more the initial parameters deviate from the ideal, the worse the results are both in terms of reproducing and recognizing a learned signal. There are two exceptions to this rule, which will be discussed below.

This system has a harder time learning nonperiodic signals as seen by the higher RMS error under ideal initial conditions; however, varying the initial c_k slightly actually yields a lower RMS error for these signals. For this system, nonideal initial c_k has its benefits.

Note also that this system has low tolerance for a nonorthogonal basis when its function is to reproduce a signal, as seen by the high RMS error. But observe that very random v_k leads to better results than v_k slightly off from the ideal. This suggests that completely randomized v_k can produce a better basis than attempting to produce an orthogonal basis and failing. Nevertheless, the RMS error is still high. If the system's function is to recognize a signal, however, then its ability is not seriously affected by nonorthogonality, as seen by the high percentage of good matches, *as long as the incoming signals are fairly distinct from each other.* Of course, this implies that the percentage of good matches will decline if the signals are similar; it will have a harder time distinguishing between two similar looking signals. With an orthogonal basis, a signal, $f(x)$, looking like the learned signal, $g(x)$, will cause the system to produce a high final response, but not as high as if the incoming signal was identical to $g(x)$; this may not happen if the basis is not accurately orthogonal. If the purpose is to accurately identify the learned signal, then this would be bad. But often in a biological system, identification is not an exact process and two signals need only be similar for both to illicit the same biological reaction.

It is a fact that a smaller set of basis functions can not handle complex signals very well; complex signals need more basis functions for good results. If we allow one or two neurons to implement one filter and hence, one basis function, then this should not be of any biological concern, given the great many neurons available.

Noise is always a consideration in biological systems leading to processing errors. As long as noise is kept to a minimum, the results should not degrade severely, and that is what the data suggests; even with only one digit of precision in the weight calculations, the RMS error does not rise significantly.

Just as noise can affect the initial conditions, it can also affect the weights once learned. The system dynamics after corrupting the learned weights is exactly the same as not initializing the weights properly; the rise in RMS error is slow but steady and for some signals, there is a dip in the error before it rises.

4.1.2 RBF-based Gradient Descent System

An important feature of this system is that the error is proportional to the ratio between the training set size and the number of RBFs. This strongly suggests that we can predict the results of huge systems by looking at smaller versions.

The effect of the gradient descent process is to make the results more pronounced; if the coefficients do not converge, then the error will be huge, but on the other hand, if there is a good set of coefficients, it will be found, hence the lowest possible error. This is the cause of the sharp drop in error as seen in the tables when the error for one radius parameter is $\gg 100$ and for the next size radius is less than 1.0. Another effect of the gradient descent process is the less stable nature of the results at the critical radius parameter where the big drop in error happens; depending on the actual distribution of the random datapoints, there may or may not be a good set of coefficients. However, this is compensated by the low error when there is a good set of coefficients. In addition, gradient descent in this system allows the initial coefficient values to vary without any effect on performance; gradient descent will find the nearest local minimum depending on the initial coefficients, but since this system has no local minima, the initial conditions do not matter. Contrast this with Backpropagation in which the initial coefficients do affect the results.

The exact shape of the RBFs do not have a major effect on performance as long as there is an adequate amount of RBFs. If the number of RBFs is limited, then the best choice would be the bell-shaped RBFs. Also note that the error decreases as the number of RBFs increases relative to the training set size. Both these factors suggest that there must be a great many RBFs coarse coding the input space for this system to work well. Since we can allow for one neuron to implement one RBF, the number of RBFs is not a major consideration biologically, given the great number of neurons available.

Because of the requirement for many RBFs (which is this system's equivalent of BP's hidden units), this system will be slower than Backpropagation if the implementation is not parallel. However, there are caveats to this. We expect this system to be implemented in parallel in a biological system with all the neurons processing simultaneously. In addition, for this test, a maximum of 150 iterations were required and for the test sizes generating the tables, only 50 iterations were performed. BP, on the other hand, needed only 7 hidden units to learn with equal results but required 300 iterations. In addition, there are optimization techniques[20] to reduce the number of RBFs by performing some pre-processing to determine where to ideally place the RBFs. Under ideal conditions, BP can not match this system's error. Under biologically constrained conditions, BP can perform as well as this system, but BP is not biologically feasible.

Even when taking into account noise during and after learning, the results only show a gradual rise in error which is negligible when the noise is small. Can we assume that the noise in a biological system is small but present? This assumption is reasonable; we know that it is present, so the question really is to what degree? This will be a matter for debate, but we do know that there is a certain operating range for all biological systems; this range accounts for the noise and its size varies depending on the system under consideration. Regardless of the actual range, it is important to note that the error rises gradually as the amount of noise increases, typical of biological systems.

This system generalizes with almost the same average error as when learning. But this is within the space. Datapoints outside the space do not fare well; this inability to extrapolate is most likely a feature of all RBF-based networks.

4.1.3 RBF-based Layered Network System

This system has the same dynamics as the RBF-based gradient descent system. Its performance is great under ideal conditions. Under biologically constrained conditions, there is a decline in the error as the RBF radius decreases and there is a specific radius, depending on the RBF density, that yields the best results. Using a radius smaller than this ideal gives progressively worse results. Also, the average error seems dependent on the ratio of training set size to RBF density.

This system does not incorporate gradient descent and the effect is noticeable. As the radius shrinks, the error drops gradually and then begins to rise again; there is no sudden drop in error as with the gradient descent based system. This is good since the error falls more consistently, but this is also bad because the average error is higher. As with the gradient descent system, however, we can decrease the error by increasing the number of RBFs relative to the training set size. Another drawback is that the starting weights do have an effect on this system, unlike the gradient descent system where the starting weights have no effect on the results.

The type of RBF used is not significant as long as there is a lot of RBFs to finely coarse-code the space; if this is not possible, then a bell-shaped RBF performs better than the other shapes.

This system is faster than gradient descent systems, but the price is a higher average error. To compensate for this, more RBFs are needed, which would slow the system down in a non-parallel implementation. This of course, is not a factor in a biological system where we expect processing to take place in parallel.

Noise also has an effect on this system, but again the effect is minimal. Note that for the same amount of noise, this system's performance degrades much less than for the gradient descent system. Extreme corruption has less effect on this system than for the gradient descent system.

This system's generalization ability is as good as its learning ability, suggesting that how well it learns is a good indicator of its overall performance. But like the gradient descent system, this one is also a very poor extrapolator.

4.2 Future Focus

This paper presents an investigation into three systems presumed to be biologically plausible. The performance characteristics of each was determined with the focus on verifying that they perform adequately under biologically constrained conditions; these conditions include the effect on performance when the initial conditions prior to learning are not ideal, when noise is a factor in the actual calculations, and when the learned weights are not stable; this encompasses all facets of the biological process that may affect performance. The conclusion we draw is that within limits that are biologically reasonable and sound, the performance degradation of these three systems is acceptable.

There are several areas of possible focus in the future: central, horizontal, and vertical. The central focus is a continued investigation of these algorithms, including the fine-tuning and extension of the systems. For example, the RBF algorithms should be tested further with various distributions of both RBFs and radii. The RBF algorithms could also be used on other problems besides normalization. Also, the matched filter system must always start learning with zero initial energy; an extension should be found that relaxes this requirement. Here, we used a trigonometric basis; other basis functions could be tested to see if they yield any improvements. In addition, parallel implementations would fully realize the speed of these systems.

The horizontal focus is to determine other algorithms that can be applied in a neural net framework. For example, the matched filter system is derived from signal processing concepts. We must not ignore theory and algorithms from other disciplines of Computer Science and Mathematics that may lead to alternative neural network models.

The vertical focus is to continue investigating these algorithms but at a lower level. The derivation of these systems has been analyzed previously. This paper gives empirical data to support the theory at a high level. We must also determine if these systems can be assembled out of low-level neuronal circuits; this can be accomplished by implementing these systems with detailed neural simulators that take into account neural cell physiology. Finally, a search must be made for these systems in the brain's neuronal structures.

Bibliography

Bibliography

[1] A. Antoniou. *Digital Filters*. McGraw-Hill, New York, NY, second edition, 1993.

[2] C. H. Bailey and M. Chen. Morphological Basis of Longterm Habituation and Sensitization in Aplysia. *Science*, 220(4592):91–93, 1983.

[3] P. S. Churchland and T. J. Sejnowski. *The Computational Brain*. MIT Press, Cambridge, MA, first edition, 1992.

[4] J. Dayhoff. *Neural Network Architectures*. Van Nostrand Reinhold, New York, NY, first edition, 1990.

[5] M. H. Hassoun. *Fundamentals of Artificial Neural Networks*. MIT Press, Cambridge, MA, first edition, 1995.

[6] C. A. Izmailov and E. N. Sokolov. Spherical Model of Color and Brightness Discrimination. *Psychological Science*, 2(4):249–259, 1991.

[7] C. A. Izmailov and E. N. Sokolov. A Semantic Space of Color Names. *Psychological Science*, 3(2):105–110, 1992.

[8] E. R. Kandel and J. H. Schwartz. *Principles of Neural Science*. Elsevier Science Publishing, New York, NY, second edition, 1988.

[9] R. Linsker. From Basic Network Principles to Neural Architecture: Emergence of Orientation-selective Cells. *Proceedings of the National Academy of Science*, 83(21):8390–8394, 1986.

[10] R. Linsker. From Basic Network Principles to Neural Architecture: Emergence of Orientation Columns. *Proceedings of the National Academy of Science*, 83(22):8779–8783, 1986.

[11] R. Linsker. From Basic Network Principles to Neural Architecture: Emergence of Spatial-opponent Cells. *Proceedings of the National Academy of Science*, 83(19):7508–7512, 1986.

[12] B. J. MacLennan. Research Issues in Flexible Computing. Technical Report CS–92–172, University of Tennessee, Knoxville, TN, September 1992.

[13] B. J. MacLennan. A Computationally Universal Field Computer That is Purely Linear. Technical Report CS–93–206, University of Tennessee, Knoxville, TN, September 1993.

[14] B. J. MacLennan. Characteristics of Connectionist Knowledge Representation. *Information Sciences*, 70(1):119–143, 1993.

[15] B. J. MacLennan. Field Computation in the Brain. In *Rethinking Neural Networks: Quantum Fields and Biological Data, edited by Karl Pribram*, pages 199–232, Hillsdale, NJ, 1993.

[16] B. J. MacLennan. Image and Symbol: Continuous Computation and the Emergence of the Discrete. Technical Report CS–93–199, University of Tennessee, Knoxville, TN, August 1993.

[17] B. J. MacLennan. Information Processing in the Dendritic Net. In *Rethinking Neural Networks: Quantum Fields and Biological Data, edited by Karl Pribram*, pages 161–197, Hillsdale, NJ, 1993.

[18] B. J. MacLennan. Continuous Computation and the Emergence of the Discrete. In *Origins: Brain and Self Organization, edited by Karl Pribram*, pages 121–151, Hillsdale, NJ, 1994.

[19] B. J. MacLennan. Neural Computation Without Sigmoids. Technical Report , University of Tennessee, Knoxville, TN, forthcoming.

[20] M. T. Musavi, W. Ahmed, K. H. Chan, K. B. Faris, and D. M. Hummels. On the Training of Radial Basis Function Classifiers. *Neural Networks*, 5(4):595–603, 1992.

[21] F. Palmieri, J. Zhu, and C. Chang. Anti-Hebbian Learning in Topologically Constrained Linear Networks: A Tutorial. *IEEE Transactions on Neural Networks*, 4(5):748–761, 1993.

[22] W. T. Rogers, S. C. Dembinski, E. B. Graves, K. M. Spyer, A. R. Moser, and J. S. Schwaber. Neural Computation in a Vertebrate Adaptive Reflex System. *International Joint Conference on Neural Networks*, 1:255–258, 1990.

[23] A. Roy, S. Govil, and R. Miranda. An Algorithm to Generate RBF-Like Nets for Classification Problems. *Neural Networks*, 8(2):179–201, 1995.

[24] R. D. Strum and D. E. Kirk. *First Principles of Discrete Systems and Digital Signal Processing.* Addison-Wesley, New York, NY, first edition, 1989.

[25] L. Xu, E. Oja, and C. Y. Suen. Modified Hebbian Learning for Curve and Surface Fitting. *Neural Networks*, 5(3):441–457, 1992.

[26] A. Zad, C. Koch, and T. H. Brown. Biophysical Model of a Hebbian Synapse. *International Joint Conference on Neural Networks*, 1:138–141, 1990.

Appendix

Appendix A

Relation Between RBF Density and Sharpness

From table 3.10, we can determine the RBF size needed to achieve the smallest distance for any RBF density up to 81. What we want to see is if there is some pattern that will allow us to predict the best RBF size for higher densities, perhaps 100 or even 1000. Indeed there is such a relation. Figure A.1 is a plot of RBF size versus RBF density for all three RBF types used in the RBF-gradient descent algorithm; here, we are comparing the case when the density of RBFs and datapoints are the same. Note that regardless of the RBF type, the ideal RBF size for any given RBF density is virtually the same. The ideal RBF size asymptotically approaches zero as the RBF density increases; it is easy to see that it should never reach zero since that is equivalent to having no RBFs coarse-coding the region. So how does the datapoint density affect the ideal RBF size? To see this, we look only at the hyperbolic secant RBF for ease of comparison. Figure A.2 plots RBF size versus RBF density for three cases: when the number of datapoints is 1/4, equal to, and $4\times$ the number of RBFs (sparse, equal, and dense respectively). Note that again, the ideal RBF size approaches zero as the RBF density increases.

An interesting result to note is that regardless of the type of RBF or the density of

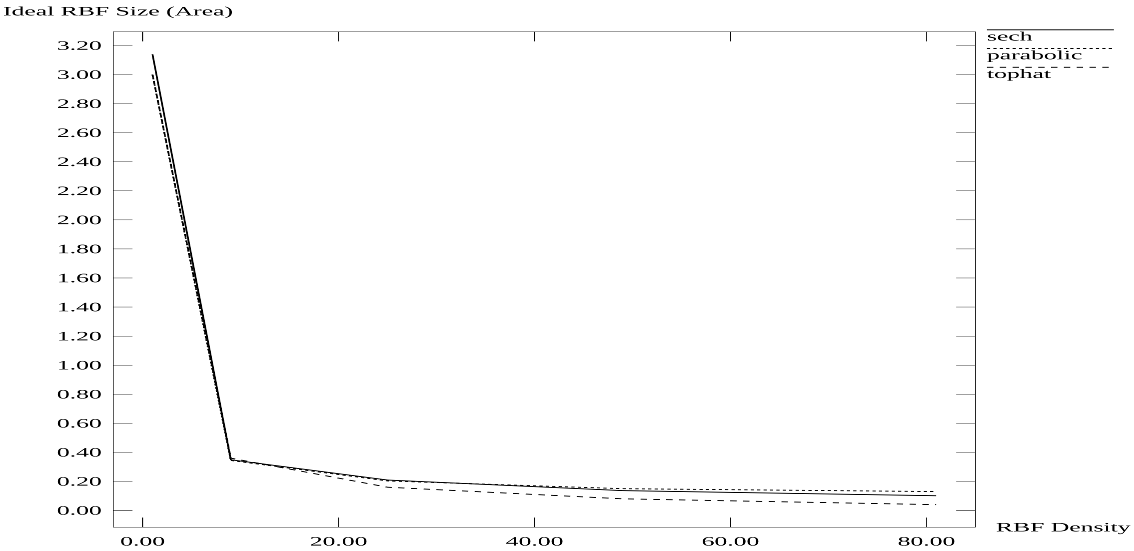

Figure A.1: Ideal RBF Size vs. RBF Density, varying RBF type (GD)

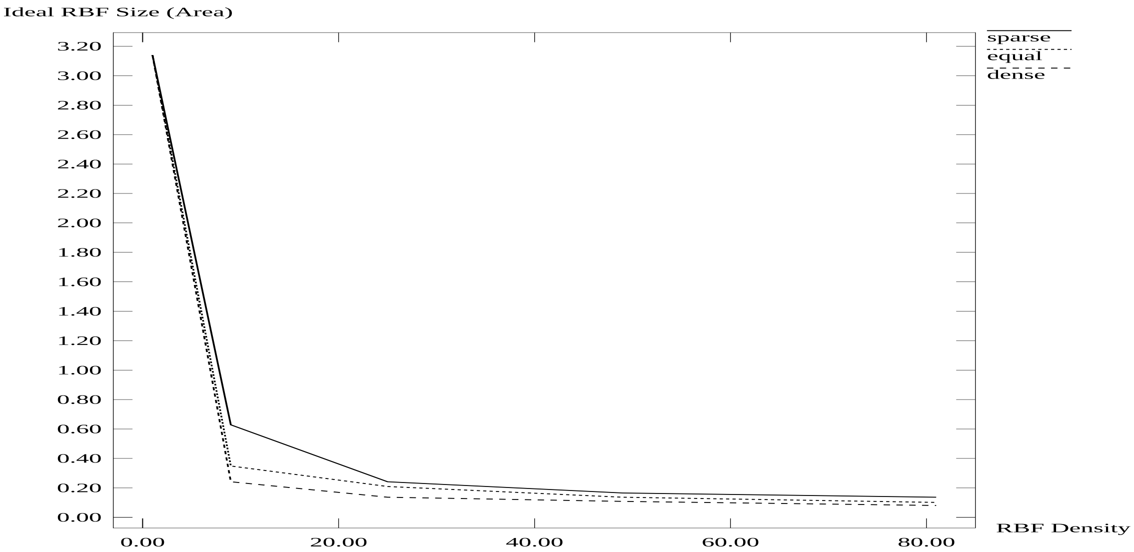

Figure A.2: Ideal RBF Size vs. RBF Density, varying data density (GD) the datapoints, the RBF size that produces the smallest error is nearly the same. However, this does not mean that the smallest error is the same regardless of the RBF type or the datapoint density; we saw in section 3.3 that the smallest error achievable is drastically different between *sech* and tophat RBFs. These results also hold true for the RBF-based layered network; figures A.3 and A.4 show the ideal RBF size approaching zero as the RBF density increases. The main difference here is that the ideal RBF size does not drop as sharply as with the RBF-based gradient descent network. In addition, the datapoint density has a more pronounced effect on this system as can be seen in figure A.4.

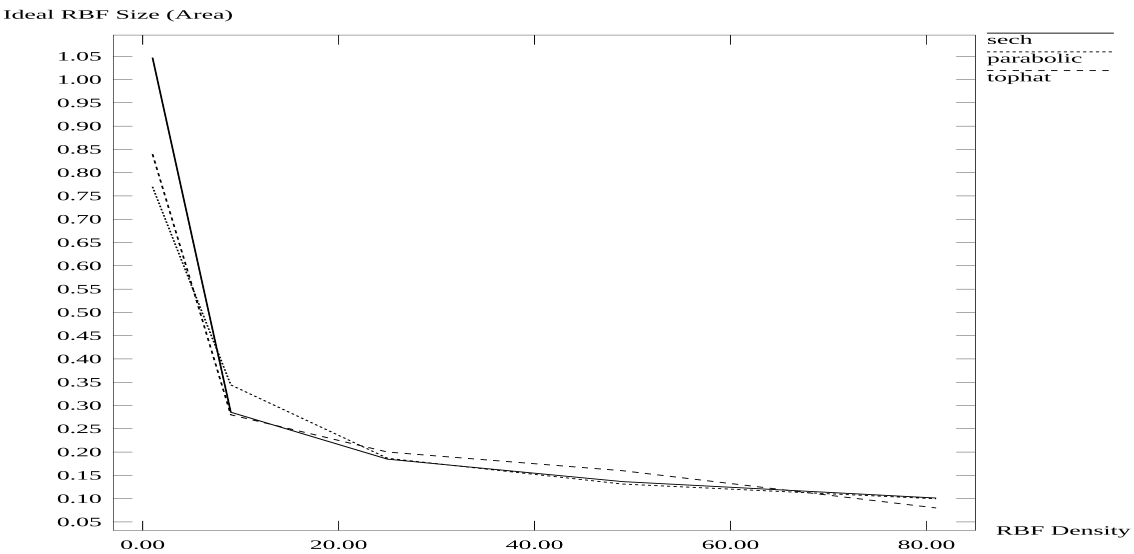

Figure A.3: Ideal RBF Size vs. RBF Density, varying RBF type (LN)

From these figures, we can predict the ideal RBF size for higher RBF densities by extending the curves. Ideally the curves should asymptotically approach zero, but since they become nearly linear at high RBF density, we can roughly approximate them linearly. Averaging the curve sections beyond RBF density of 60 from figures A.1 and A.2 yields the line $y = -0.000949x + 0.175$. Repeating this process for the curves in figures A.3 and A.4 yields the line $y = -0.001363 + 0.136$. It is important to note

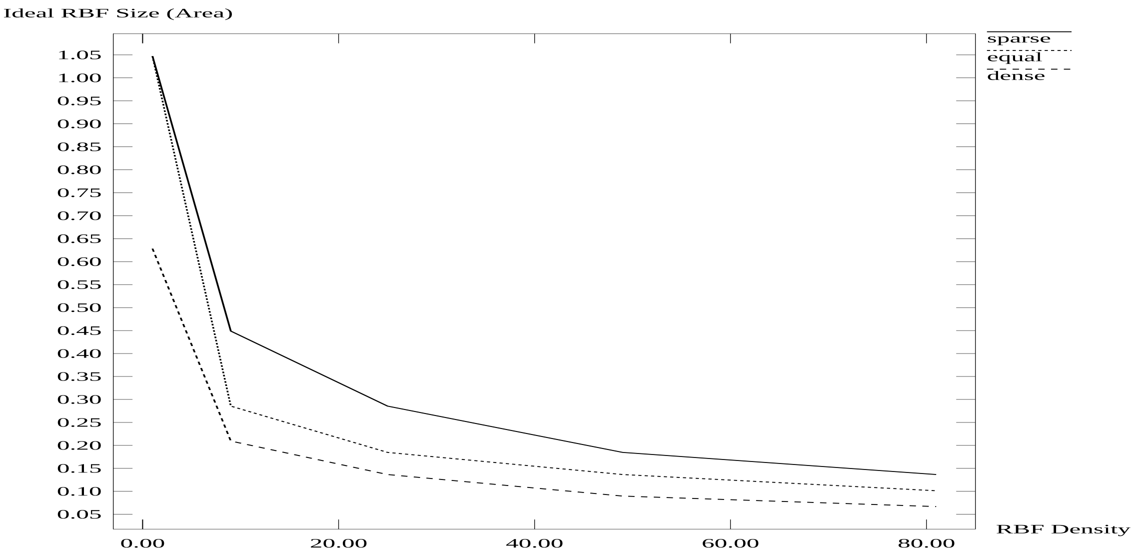

Figure A.4: Ideal RBF Size vs. RBF Density, varying data density (LN)

that these lines are rough approximations; curve-fitting should be performed if a more accurate extension is required. In addition, note that though the curves are almost identical, the precise shape is dependent on the type of RBF as well as the datapoint density. To accurately predict, the RBF type and datapoint density must be known in advance.

Vita

George Shen was born in Ping-Tung, Taiwan on March 5, 1967. He came to the United States when he was three and grew up in the suburbs east of Los Angeles. In 1984, he graduated from Arcadia High School and entered the University of California at San Diego. He managed to get out in five years with a Bachelor of Arts degree in General Biology and Mathematics/Computer Science. From 1989 through 1992, he was a member of the DB2 software development team at IBM's Santa Teresa Laboratory located in San Jose, California. Interestingly, two members of the team were from the Knoxville/Oak Ridge area. Intrigued by their stories of East Tennessee and Big Orange, Mr. Shen then headed for the University of Tennessee where he received his Masters in Computer Science.