Cleanup Memory
in
Biologically Plausible Neural Networks

by

Raymon Singh

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Systems Design Engineering

Waterloo, Ontario, Canada, 2005

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

During the past decade, a new class of knowledge representation has emerged known as structured distributed representation (SDR). A number of schemes for encoding and manipulating such representations have been developed; e.g. Pollack's Recursive Auto-Associative Memory (RAAM), Kanerva's Binary Spatter Code (BSC), Gayler's MAP encoding, and Plate's Holographically Reduced Representations (HRR). All such schemes encode structural information throughout the elements of high dimensional vectors, and are manipulated with rudimentary algebraic operations.

Most SDRs are very compact; components and compositions of components are all represented as fixed-width vectors. However, such compact compositions are unavoidably noisy. As a result, resolving constituent components requires a cleanup memory. In its simplest form, cleanup is performed with a list of vectors that are sequentially compared using a similarity metric. The closest match is deemed the cleaned codevector.

While SDR schemes were originally designed to perform cognitive tasks, none of them have been demonstrated in a neurobiologically plausible substrate. Potentially, mathematically proven properties of these systems may not be neurally realistic. Using Eliasmith and Anderson's 2003 Neural Engineering Framework, I construct various spiking neural networks to simulate a general cleanup memory that is suitable for many schemes.

Importantly, previous work has not taken advantage of parallelization or the high-dimensional properties of neural networks. Nor have they considered the ef-

fect of noise within these systems. As well, additional improvements to the cleanup operation may be possible by more efficiently structuring the memory itself. In this thesis I address these lacuna, provide an analysis of systems accuracy, capacity, scalability, and robustness to noise, and explore ways to improve the search efficiency.

Acknowledgments

First and foremost, I am deeply indebted to Chris Eliasmith. It is no understatement to say that this thesis—and my graduate career—would be non-existent without Chris. The two years, since he encouraged me to join his lab, have been challenging, thought-provoking, and all too brief. He has not only given me expert guidance in computational neurobiology, but has also been willing to support all my other varied interests. I feel truly privileged to have worked with him.

Special thanks go to my past and present labmates, John Conklin, Phil Smith, Bryan Tripp, Mark Johnson, and Chris Parisien. I thank them for the stimulating discussions, the not-so-stimulating discussions, and the laughter.

I would also like to thank Ross Gayler and Pentti Kanerva for offering their valuable time to discuss this fascinating topic with me.

Lastly, I would like to thank my friends and family, even though they have no idea what I'm talking about. They have simultaneously kept me grounded and aloft, focused and alive, and supported me in so many ways that cannot be adequately expressed.

Dedication

This is dedicated to my grandparents, Balwantia and Dhanpaul Ramdyhal.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In everyday life we perform an astonishing variety of sensory recognition tasks. To survive, we must be able to distinguish friend from foe, fair from foul, sweet from sour, and so on. Recognition occurs at many levels of complexity; we can distinguish black from white, discern letters on a page, detect objects in a scene, and understand the difference between happiness and sadness.

Some algorithmic approaches to recognition involve serially searching through large databases of knowledge, picking up the pertinent, and ignoring the irrelevant. The computational resources required of these approaches are enormous—despite operating in an ideal setting and using fast, solid state devices. In contrast, people perform these tasks instantly and effortlessly, often with only partial and noisy information. Moreover, our 'hardware' is composed of slow, fragile neurons operating in a noisy and uncertain environment.

To achieve this feat, the brain employs massive parallelism and redundancy.

However, this is only part of the solution; one also needs a representation scheme that is conducive to parallelization and is robust to noise. Traditional symbolic schemes are representationally inadequate. Symbolic frameworks define operations that manipulate simple representative units. This approach lacks neural plausibility since it seems unlikely that a single neuron is responsible for representing a concept. Rather, the representation of a concept is more likely to be a pattern of activation distributed over a number of units. These *distributed representations* (DRs) have been codified in many ways. Typically, they are high dimensional vectors that encode information throughout the elements. Since the early 90s, researchers have been exploring how such representations can be manipulated with rudimentary algebraic operations to form *structured distributed representations* (SDRs). SDRs are ideally suited for recognition tasks, since they provide a highly parallel and efficient method for comparing both large and small scale structure.

The act of recognition, using fixed-width representations, requires a cleanup memory, since these vectors are inherently noisy. A cleanup memory, in basic terms, removes the noise from a noisy vector, in order to recover basic components. Cleanup memories thus also facilitate pattern recognition, pattern completion and object detection.

1.1 Motivation

SDRs represent the patterns of activation of the brain and are capable of symbolic computation. As such, they offer a surprisingly short bridge between low-level processing and high-level cognition. Yet, no attempts have been made at modeling

SDR operations with realistic neurons. Plate (2000) shows that randomly connected sigma-pi neurons can perform the rather complicated operations necessary for his HRR scheme. However sigma-pi neurons are not very realistic, and Plate makes no attempt at performing a cleanup memory with neurons. His work nevertheless, shows that operations that are symbolically complex need not be so in a more neurally plausible substrate.

On the other hand, some operations may turn out to be neurally unrealistic. Relatively simple symbolic tasks may prove too difficult or unsustainable in the noisy environment of the brain. For example, SDRs have impressive accuracy and storage capacity that increases with higher dimensions. Will these properties hold when implemented with plausible neurons? Three neurobiologically plausible cleanup memories are presented in this paper to examine such issues.

In designing a cleanup circuit, we require a device that stores information in a manner that can be efficiently searched and recalled. As such, cleanups can also be used to quickly determine if an item is (or is not) in memory, thus providing a fast method of comparing and branching signals. Thus in practical terms, the cleanup networks we seek here are highly parallel, general-purpose search engines, that can serve as realistic neural devices for pattern recognition and machine intelligence.

1.2 Thesis Overview

The next chapter provides background to the cleanup problem. It describes the particular systems that require a cleanup memory, the mathematical formulation

of the problem, a solution, and performance measurements of that solution.

Chapter 3 provides more background by introducing the *Neural Engineering Framework* (NEF) that is used to create neurbiologically plausible networks. In particular, the importance of neural representation, computation, and dynamics is discussed. The discusion highlights features of the NEF that are pertinent for the proposed cleanup circuits.

Chapter 4 combines the cleanup problem of chapter 2 with the NEF described in chapter 3 to construct three plausible cleanup memories. We start with a traditional butterfly network implemented with plausible neurons. Then a novel function cleanup is introduced that uses suitably chosen weights to 'calculate' the cleanup function. Finally, we present the sparse cleanup which improves upon the function network by producing sparse codes. An analysis of these networks and their relative merits are discussed. Finally, the issue of learning is raised, and two preliminary learning algorithms are proposed.

The last chapter concludes with suggestions for future work.

Chapter 2

Background

To understand cognitive processes, one must understand what types of representations are likely to occur in the brain. In this section, we present a scheme that is representationally adequate for cognitive manipulation. This scheme, as most, requires a cleanup memory, which is discussed here in mathematical detail. We also introduce some performance measures to gauge the quality of the cleanup and to verify proposed properties.

2.1 Representation

One of the fundamental debates borne out of connectionist research is the issue of representation (Fodor 1987; Fodor and McLaughlin 1990; Smolensky 1990; Gelder 1990; Chalmers 1990). How is information represented in a neural substrate so as to facilitate human-like information processing? Here, we give a brief summary

of two general types of representation, symbolic and distributed, and discuss their suitability for cognitive processing.

2.1.1 Symbolic Representation

Researchers have had a long and successful history with symbol manipulating systems such as mathematics, formal languages, and traditional computation. Hence, it is no surprise that initial attempts at human-like knowledge representation were symbolic. In symbolic schemes, each piece of information is captured by a single unit with an explicit format. In computer science terms, the representations are like memory pointers; they provide access to the data they represent but bear no resemblance to them.

Knowledge is derived from the structures these symbols form, and the operations that are performed on them. Taking this to its theoretical extremes, Newell and Simon combine symbol systems with universal computation (e.g. a Turing Machine) and produce the *Physical Symbol System Hypothesis* (Newell and Simon 1976, p. 116):

> A physical symbol system has the necessary and sufficient means for general intelligent action.

By "necessary" Newell and Simon predict that "any system that exhibits general intelligence will prove upon analysis to be a physical symbol system" (Newell and Simon 1976, p. 116).

While there is little doubt that humans can manipulate symbols, there are reasons to believe that symbolic representation is psychologically unrealistic. First, symbols are brittle. Minor damage to a symbol leads to a complete loss of the concept. Second, symbolic representation is strongly propositional. While performing well on language tasks such as formal logic and mathematics, they are very awkward when dealing with non-language tasks requiring manipulation of images, sounds, and smells. Third, symbols are not probabilistic. Symbols, in themselves, cannot pick up the regularities in the environment. Fourth, symbolic representation is not conducive to parallel computation. Most symbolic frameworks operate serially, using simple rules of cause and effect, rather than in parallel with recurrent, non-linear dynamics.

Since symbol systems (e.g. a computer) and people are arguably instantiations of a Universal Turing Machine (Eliasmith 2002), they are, in a sense, computationally equivalent. However, claims of universality and equivalence are only true under the assumption of infinite time and infinite resources (LeCun and Denker 1992). Real physical computing devices are a product of compromise. The architecture of different machines will produce very different computational efficiencies (Eliasmith 2002). For example, to circumvent the problems of noise, computer engineers have developed devices that virtually eliminate noise in order to reliably process pristine symbols. Nature, on the other hand, has designed something different; something that operates asynchronously in and noisy environment; something in which each component is highly unreliable; something that was designed to survive—not perform boolean logic.

2.1.2 Distributed Representation

Connectionists, in contrast to symbolicists, have taken a wholly different view of neural computation. Taking a cue from the architecture of the brain, they attempt to model cognitive functions through artificial neural networks (ANN). These networks are graph-like structures where the nodes represent simplified neurons and the edges are excitatory or inhibitory connections. Computational power is not derived from moving symbols around, but rather from simple neural activation and connection weights.

Corresponding with this new architecture, connectionists found a new type of representation lying in the so-called hidden layers of their networks. While the input and output of these networks were locally encoded (i.e. symbolic) vectors, the activation patterns in the hidden layers formed something different. This layer seemed to encode a mixture of input and output. Notably, the representation was not symbolic; one could not point to a specific unit that represented a single item. Instead the information was encoded throughout all the units. Unlike memory pointers, these representations often resemble (statistically) the things they encode. They could represent individual items, complex data structures, transformations, or combinations thereof. These representations are known variously as sub-symbolic, holistic or distributed representation.

To understand the nature of these encodings, the analysis of distributed representation (DR) and the methods for making structured distributed representations (SDR) has developed over the years. Their study has become the study of high-dimensional representational spaces and transformations applied to them. It reveals

a new kind of computing that is very different from that of traditional symbolic machines. The encodings are stochastic patterns that typically take the form of large random vectors (codevectors). Gunnar Sjödin (1997) explains further:

> Stochastic computing derives its power from the unusual mathematical (geometric) properties of high-dimensional spaces and from the law of large numbers. Algorithms that are imprecise or unreliable in low dimensions converge to precise and reliable algorithms in high dimensions, resulting in computational behaviour that is not possible with deterministic methods.

Distributed representations do not suffer from the psychological shortcomings of symbolic representation. First, they are not brittle but degrade gracefully; damage to the representation leads to a graded loss of information—not a complete loss. Second, distributed representations are not heavily language-centric and are suitable in many diverse domains. Third, distributed representations are inherently statistical in nature and can easily accumulate properties of the environment. Fourth, these encodings are conducive to parallel computation. In particular, they can execute traditional serial searches in one parallel step.

Yet, despite these non-symbolic properties, distributed schemes can nevertheless perform high-level symbolic processing. Indeed, they must if they are to underwrite human cognition. To illustrate the symbolic nature of SDR, we must first delve into the mathematical foundations of these representations.

2.1.3 Mathematics of Structured Distributed Representation

A number of schemes for representing and manipulating structured distributed representations have been developed, e.g. Recursive Auto-Associative Memory (RAAM) (Pollack 1990), Binary Spatter Code (BSC) (Kanerva 1996, 1997, 1998), MAP encoding (Gayler 1998), and Holographically Reduced Representation (HRR) (Plate 1994, 1995). For or a summary of the most modern schemes see Plate (1997).

The minimal operations underlying SDR are superposition, binding, and unbinding.

Definition 1 *Superposition is a merging operation that takes two vectors and creates a third which is similar to the original two. Using + as the superposition operator and the metric, d, to measure similarity, we require*

$$C = A + B \qquad \textit{such that } d(A, C) \approx 0 \textit{ and } d(B, C) \approx 0 \tag{2.1}$$

where A and B are random vectors.

Definition 2 *Binding is a mixing operation that takes two vectors and produces another which is not close to the original two. Marking $*$ as the binding operator, we write*

$$C = A * B \qquad \textit{where } d(A, C) \textit{ and } d(B, C) \textit{ are large.} \tag{2.2}$$

Definition 3 *Unbinding is the inverse of binding. We must be able to extract any bound vector; typically, using the other bound vector as a cue. For example, using*

 # to represent unbinding we need

$$A * B\#A \approx B \qquad \textit{or equivalently } d(A * B\#A, B) \approx 0. \tag{2.3}$$

Some schemes use an inverse (or pseudo-inverse) operation to facilitate unbinding (e.g. $A * B * A^{-1} \approx B$). Likewise, many other algebraic properties can be useful. The examples in this paper assume associativity and commutativity under binding and superposition, distributivity under binding and unbinding, and the existence of an identity and a zero. The encoding schemes of Holographically Reduced Representation (HRR) and Binary Spatter Code (BSC) are summarized in table 2.1.

It is clear from the superposition requirement that no scheme can be exact; i.e. we cannot drop the approximately equal signs in (2.1). Even an optimal solution will generate noise. This is mostly a nuisance, but in some cases, distributed representations can actually exploit this noise, for example, in probing operations (Plate 1995; Kanerva 1997).

The probing task involves finding marked items in a list. Traditionally, this is executed serially by iteratively checking each item in the list for the particular mark. In contrast, using SDRs, we can perform this search in one step by casting the problem into a distributed form.

Example 1 *Probing Task*

Suppose the list X contains the items $\{A, B, C\}$, each of which can be marked by the properties $\{p_1, p_2, p_3\}$. First, we represent each item and property as HRR vectors.

Operation	**HRR**	**BSC**
Representation	$A = (a_0, a_1, \ldots, a_{n-1})$ $a_i \in N(0, 1/n)$	$A = (a_0, a_1, \ldots, a_{n-1})$ $a_i \in \{-1, 1\}$
Superposition	Vector addition $C = A + B$	Majority rules addition $C = sign(A + B + \eta)$
Binding	$C = A * B$ $c_i = \sum_{j=0}^{n-1} a_j b_{i-j}$	$C = A \otimes B$ $c_i = a_i b_i$
Unbinding	$C = A \# B$ $c_i = \sum_{j=0}^{n-1} a_j b_{i+j}$	$C = A \otimes B$ $c_i = a_i b_i$
Similarity	$d(A, B) = 1 - (A \cdot B)/\|A\| \, \|B\|$	$d(A, B) = 1 - (A \cdot B)/\|A\| \, \|B\|$
Inverse	$A^{-1} = (a_0, a_{n-1}, a_{n-2} \ldots, a_1)$	$A^{-1} = A$

Table 2.1: Distributed representation schemes. HRR uses real-valued vectors and employs the holographic operations of circular convolution and correlation to perform binding and unbinding respectively. Note: all subscripts in HRR are taken modulo-n. BSC vectors, on the other hand, work with binary elements, in this case ± 1, and employs the XOR operator to perform both binding and unbinding. Majority rules vector addition is an element-wise vote where ties are broken randomly with a small noise vector, η.

Then we encode the entire list as a vector. For example, a particular list might be encoded as

$$X = A * p_3 + B * p_2 + C * p_1 \tag{2.4}$$

Now, to find which item has the property p_2 in X, we simply unbind with p_2:

$$\begin{aligned} X\#p_2 &= (A * p_3 + B * p_2 + C * p_1)\#p_2 \\ &= A * p_3\#p_2 + B * p_2\#p_2 + C * p_1\#p_2 \\ &= \eta_1 + (B + \eta_2) + \eta_3 \\ &\approx B \end{aligned} \tag{2.5}$$

where each η_i is a nondescript codevector; i.e. noise. If the η_i are uncorrelated, they will sum to another HRR vector and B can be recovered.

The final step in (2.5) follows from the superposition property (2.1). In practice, however, we would like to recover B exactly. This example is endemic of a more general problem: noise. Most operations will result in noisy vectors. Furthermore, external noise will be present in realistic systems. In fact, the initial encodings of input vectors may be noisy or incomplete. These vectors must be resolved before further processing can continue. For these reasons, we must employ a cleanup memory.

2.2 Cleanup memory

A cleanup memory is a structure which, when given a noisy vector, will return the closest match to an item in a collection of stored vectors. It is crucial to have an effective cleanup function, for without one, repeated operations will quickly degenerate into noise. Surprisingly, a simple device like this can also perform powerful, high-level symbolic operations.

Example 2 *Simple Cleanup*
Suppose the cleanup memory contains all the vectors in example 1; i.e. $\{A, B, C, p_1, p_2, p_3, X\}$ *are the clean vectors. If the cleanup is presented with a noisy vector,* $A + \eta$ *where* η *is a noise term, then the cleanup should output* A*. Similarly, when presented with* $X \# p_2$ *the result should be* B*.*

Functionally, cleanups are like classification devices used in the field of pattern recognition. However, they differ in the types of objects they operate over. Classification typically operates with low dimensional, localist feature vectors and output a symbolic token that represents the vectors class. On the other hand, the input and output of a cleanup is always a high dimensional, distributed vector which, as discussed earlier, is more appropriate for neural processing.

Cleanup memories have a number of uses other than noise reduction. They can also execute pattern recognition, pattern completion, and object detection. A few toy examples will illustrate this point.

Example 3 *Pattern Completion*

Figure 2.1: Sitting bear. How can this image be represented?

Consider the scene portrayed in Figure 2.1. In HRR notation, the bear could be represented literally as the sum of its parts:

$$\textbf{\textit{bear}} = \textbf{\textit{eyes}} + \textbf{\textit{nose}} + \textbf{\textit{paws}} + \textbf{\textit{tail}} + \textbf{\textit{teeth}} + ...$$

where each of the terms is a vector. Other vectors can be similarly composed of their constituent parts (e.g. ***flowers*** *and* ***grass****). Now, suppose* {***bear***, ***flowers***, ***grass***} *are stored in a cleanup memory and upstream perceptual processes have constructed the vector,*

$$\textbf{\textit{thing}} = \textbf{\textit{paws}} + \textbf{\textit{nose}} + \textbf{\textit{eyes}}.$$

Because ***bear*** *and* ***thing*** *are composed of similar vectors, they are likely to point in a more similar direction than* ***flower*** *and* ***grass****. (In SDRs, this likelihood increases with vector size.) Thus when* ***thing*** *is presented for cleanup,* ***bear*** *is likely to be*

returned. Notably, ***bear*** *contains items that were missing in* ***thing****. Hence by returning* ***bear****, the cleanup has effectively inferred, based on partial information, that* ***thing*** *has a* ***tail****,* ***teeth*** *and other items associated with* ***bear****. In this very simple manner, distributed representations and cleanup memories can accomplish pattern recognition and completion.*

Example 4 *Object Detection*

Now consider the HRR encoding of the entire scene:

$$\boldsymbol{scene = flower + bear + grass + ...}$$

where the terms are codevectors generated by processes like the one discussed above. The task now is to construct a device to detect dangerous elements in the scene. A cleanup memory that contained ***bear*** *and other harmful elements (not* ***flower*** *or* ***grass****) can quickly perform this task. It would recognize* ***bear*** *as a danger and consider the other vectors,* ***flower*** *and* ***grass****, as noise. Similarly, since these vectors can represent anything, one can imagine other structures geared towards detecting particular objects (friend, foe, food, etc...).*

These examples exhibit some of the power of sub-symbolic computation. Cleanup memories form an important and integral part of these systems and can be used to either accumulate or ignore information.

It is clearly advantageous for an individual to have a fast danger detector like the one described above, but is it possible that the brain could perform such operations? Previous work has shown that superposition, binding, and unbinding can be implemented with realistic neurons (Eliasmith 2004; Conklin and Eliasmith forthcoming), but what about the cleanup mechanism? Can a collection of neurons be

configured to perform the cleanup operation? This is the central question of this thesis. We will attempt to construct accurate and efficient cleanup systems using a network of spiking neurons. This is done not only to give the systems biological merit, but also to explore issues, that have not been tackled by related work, such as accuracy and capacity performance under external noise.

2.3 Previous work

Plate (2003) notes that many artificial neural systems employ a form of cleanup memory. BoltzCONS uses pull-out networks (Touretzky 1986, 1990), CRAM and DCPS use cleanup circuits (Touretzky and Hinton 1985, 1988; Dolan 1989), TODAM uses an R-system (Murdock 1982, 1983), BSC uses Sparse Distributed Memory (Kanerva 1988), and Plate himself uses a conventional list structure (Plate 2003). The implementation of these cleanup memories depend largely on the architecture they work with. In the next section, we explicitly define a very generic cleanup operation that operates over a set of codevectors with minimal constraints.

The functionality of cleanups is similar to associative memories (Willshaw et al. 1969; Kohonen 1977). An associative memory is a content-addressable structure that builds associations between input and output items. They come in three flavours: autoassociative, heteroassociative, and multiassociative. Autoassociative memories learn to associate the input with itself. They are usually recurrent networks that, when given a cue, drift to a nearby attractor point. These attractor points correspond to the clean vectors mentioned above. Popular network implementations include Hopfield nets (Hopfield 1982), and RAAMs (Pollack 1990). Like

cleanups, autoassociative memories are capable of pattern recognition and completion.

One can view autoassociative memories as a form of cleanup since the output is similar to the input. Heteroassociative memories, on the other hand, learn to output a pattern that is different from its input. This is useful in stimulus/response applications. In section 4.2, a network that is conducive to both forms of memory is presented.

Multiassociative memories (Kolen and Pollack 1991) are associative memories that take as input a composition of arbitrary items, but return a composition of items only in memory; i.e. it cleans all the items at once. Unlike the previous memories, these systems can return compositions that the system has never learned. Attractor networks, in particular, are incapable of multiassociation, since they can only return a single item. If one reads composition as a superposition of vectors, then these networks are equivalent to heteroassociative memories. A formulation of a cleanup memory that can output multiple items is described in the next section.

The material presented here is relatively novel in the sense that the majority of neural systems 1) consider cleanup as an external operation that is not necessarily a part of the system, and/or 2) are concerned with practical computation rather than with biological credibility. The reason for 1) is that there exists a plain solution to the problem that is readily implemented. We present a somewhat non-obvious approximation to the solution that is essential in a neurally plausible context and is particularly useful for multiassociation. The reason for 2) stems from the need for simplicity of construction and ease of analysis. In chapter 3 a neural framework

that alleviates many of these concerns is introduced. Both 1) and 2) dismiss the importance of the brain and lead to cognitively improbable implementations (e.g. list structures).

2.4 Mathematical definition

Before we present our solution, let us fully characterize the problem and discuss some standard nomenclature used throughout this thesis.

Definition 4 *Formally, we seek a cleanup operation, C, such that when given a cue vector, x, it returns the closest vector in the set, $\mathbb{S} = \{v_1, v_2, v_3, \ldots, v_m\}$. All vectors are of the same dimension, n, and all v_i are the same length; usually $\|v_i\| = 1$ or more loosely $E[\|v_i\|] = 1$. Thus, using the metric, d, we require,*

$$C(x) = \arg\min_{v_i \in \mathbb{S}} d(v_i, x) \tag{2.6}$$

We can store all vectors in an $m \times n$ matrix, $M = [v_1, v_2, v_3, \ldots, v_m]^T$, and define the measure, $d(\cdot)$, as the normalized dot product of the two vectors.[1] This choice allows us to efficiently calculate the m-dimensional similarity vector, $s = xM^T$, which is effectively the dot product of the cue vector with all the vectors in $\mathbb{S}$. The maximum value in s points to the clean vector.

[1]The dot product is not a true metric since, for example, it can be negative. Nevertheless, it suffices as a practical and efficient similarity measure.

Most of the cleanup memories presented here are subjected to external noise, and an exact solution is never expected. Thus it might be computationally advantageous, and hence more biologically plausible, to consider an approximation.

Definition 5 *A useful approximation to the operation* (2.6) *can be accomplished with fuzzy sets (Zadeh 1965). Rather than returning the single closest vector, it outputs a weighted superposition of the closest vectors. The weights are determined by a suitably defined membership function,* $\mu(\cdot)$*. The resulting cleaned output,*

$$C_\mu(x) = \sum_{v_i \in \mathbb{S}} \mu_{v_i}(x) v_i, \tag{2.7}$$

is a vector that contains both the set of objects in $\mathbb{S}$*, and the degree of membership of each.* C_μ *thus forms a concrete encoding of a fuzzy set.*

This type of cleanup is useful for operations that return multiple answers as in multiassociative systems. Also, C_μ can automatically return a near zero vector if no suitable match is found. The max cleanup, C, on the other hand, must be augmented with another mechanism that rejects vectors with low similarity.

Once again, using the dot product as a similarity measure, (2.7) can be written as

$$C_\mu(x) = \mu(xM^T)M = \mu(s)M \tag{2.8}$$

where, $\mu(s)$, returns the membership value of each element in s.

The effectiveness of these types of memory will depend on the distribution of the stored vectors and the membership function. In Matlab notation, defining

$\mu(s) = (s == \max(s))$ is equivalent to the optimal cleanup defined in (2.6). For HRR vectors that are uniformly distributed, the expected similarity distribution of a clean vector v and v plus $k-1$ random vectors is $N(1, k/n)$ (Plate 2003, p. 100) Therefore, a natural membership function is the Gaussian function,

$$\mu(s) = e^{-n(s-1)^2/2k} \tag{2.9}$$

Normally, $\mu(s) \in [0,1]^m$, but since useful information can be gained from vectors that point in opposite directions, one can extend μ to return values in the range $[-1,1]$ and define membership functions such that $\mu(s) = -\mu(-s)$.[2] Thus, a vector that is nearly opposite to a clean vector will be weighted negatively and produce a vector pointing in the same direction as the clean one. Therefore, to credit negative information in HRR cleanups we use

$$\mu(s) = e^{-n(s-1)^2/2k} - e^{-n(-s-1)^2/2k} \tag{2.10}$$

Other forms of cleanup memory have additional features that may be useful in some applications. For instance, some return the closeness measure (i.e. strength or valence) to the cue (Plate 2003). If no match is found, some systems return the cue rather than a zero vector. Here, we concentrate primarily on performing a generic cleanup operation.

Table 2.2 gives a summary of variables used throughout this thesis.

[2]Of course, such an extension will have deleterious affects on the formalisms of fuzzy logic, but it is useful with respect to (2.7)

Symbol	Description
n	Vector size
m	# of vectors in memory
$\mathbb{S}$	Set of all clean vectors
v_i	The ith vector in $\mathbb{S}$
v	A generic vector in $\mathbb{S}$
M	Memory matrix
x	Cue vector
s	Similarity vector
η	Random noise vector
$C(x)$	Cleanup function
$C_\mu(x)$	Weighted cleanup
$d(u, v)$	Similarity measure: Normalized dot product
$\mu(\cdot)$	Cleanup membership function
k	Number of superposed vectors

Table 2.2: Cleanup symbols and operations.

2.5 Direct Solution

The above characterization of the problem admits a solution that can be directly encoded in Matlab. We present here the results of this solution and motivate the tools used for assessing its effectiveness. These results will serve as a benchmark for other cleanup memories.

For consistency, the following tests will work with HRR vectors; that is, an n-dimensional vector where each element is independently, identically, and normally distributed with zero mean and a variance of $1/n$ (see Table 2.1). Additionally, all the vectors in M are normalized.

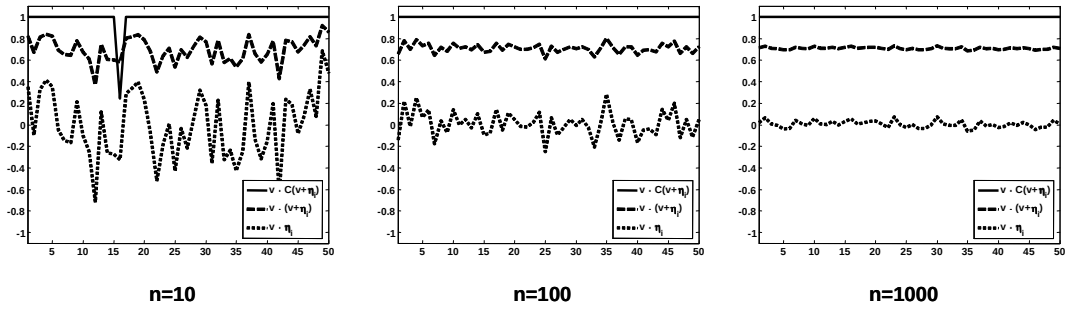

Figure 2.2: Performance of vectors in high dimensions. For each graph, 50 trials are run where we compare a random cue vector, v, with a noise vector, η, a noisy cue, $v + \eta$, and a cleaned vector, $C(v + \eta)$. The cleanup usually recovers v (only one mistake at n=10). Increasing the dimensionality of the vector, n, reduces the variability in these calculations leading to more reliable operations.

2.5.1 Reliable recovery

First, in order to demonstrate some basic properties of cleanup memories, we perform a few tests. Figure 2.2 shows the effects that dimensionality has on vector manipulations. Note how increasing n diminishes the amount of systemic noise, making it easier to distinguish vectors that are similar from those that are random. Furthermore, as n increases, two randomly chosen vectors become more and more orthogonal. This property will hold for vectors where the elements are identically distributed with a mean of zero.

Using the graphs in Figure 2.2 we can determine a threshold that separates vectors that are random from those that are correlated.

By inspection, we see that a similarity measure above 0.8 is usually produced by a cleanup operation and a measure below 0.5 is typically produced by two random vectors. More precisely, we note that the expected similarity between v_i and the

normalized noisy vector, $v_i + \eta$, is $\sqrt{1/2} < 0.8$.[3] Thus, if the similarity of a known vector and its cleaned up version is greater than 0.8, we can say that the vector is reliably recovered.

Given a particular encoding scheme, we can calculate a more precise threshold. Plate, for example, uses numerical techniques to find the optimal threshold for HRR vectors (Plate 2003, p. 100). The optimal value is not generally useful however, since it depends on the number of superpositions, k, which is usually unknown. Therefore, unless otherwise stated, a 0.8 demarcation is assumed throughout this thesis.

2.5.2 Superposition Capacity

A natural question to ask of this kind of representation is: how many vectors can one superpose and still reliably recover an item? We define the vector, u_k, as the superposition of a known clean vector, v, and k-1 random vectors, η_i:

$$u_k = v + \sum_{i=1}^{k-1} \eta_i \tag{2.11}$$

Since we expect $\|u_k\| \approx \sqrt{k}$, the expected normalized dot product is

$$\begin{aligned} \mathrm{E}\left[d(v, u_k)\right] &= \mathrm{E}\left[v \cdot u_k/\sqrt{k}\right] \\ &= \sqrt{1/k}\left(\mathrm{E}\left[v \cdot v\right] + \sum_{i=1}^{k-1} \mathrm{E}\left[v \cdot \eta_i\right]\right) \\ &= \sqrt{1/k} \end{aligned} \tag{2.12}$$

[3] This is proven later in eqn. (2.13) with $\alpha = \beta = \sqrt{1/2}$

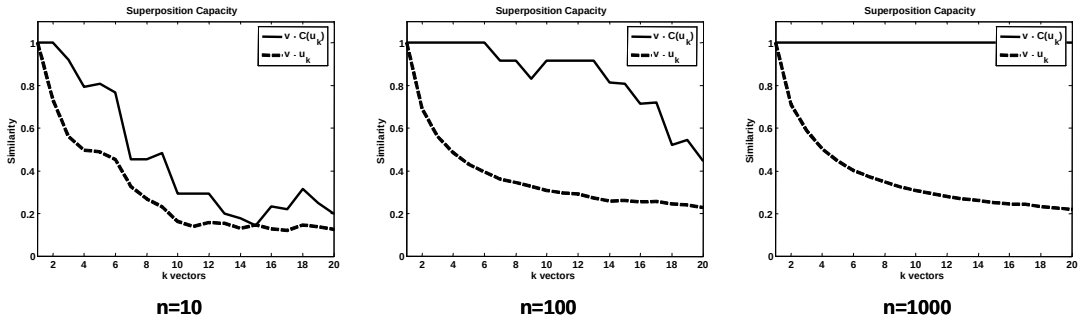

Figure 2.3: Superposition capacity of higher dimensional vectors. In each graph we compare the performance of the cleanup memory versus the similarity of an unclean vector, u_k.

since v and η_i are nearly orthonormal ($\mathrm{E}[v \cdot \eta_i] = 0$), and v is normalized ($\mathrm{E}[v \cdot v] = 1$).

The similarity of v with u_k and a clean $C(u_k)$ is compared in Figure 2.3. The results dramatically show how an efficient cleanup memory can enhance superposition capacity. For example at n=100, we can reliably recover (threshold=0.8) a vector that has been corrupted by the superposition of 14 vectors with cleanup, and only 2 without. There is even greater improvement for higher n.

2.5.3 Scalar Distortion

A more comprehensive performance measure can be determined by using scalar distortion. We can generically define a corrupt vector, $u_\alpha = \alpha v + \beta\eta$, where α and β are positive constants such that $\alpha^2 + \beta^2 = 1$, and η is a noise vector. This produces a vector which is roughly unit length and has a parameterized signal-to-noise term. Again, since two random vectors are nearly orthogonal (i.e. $\mathrm{E}[v \cdot \eta] = 0$),

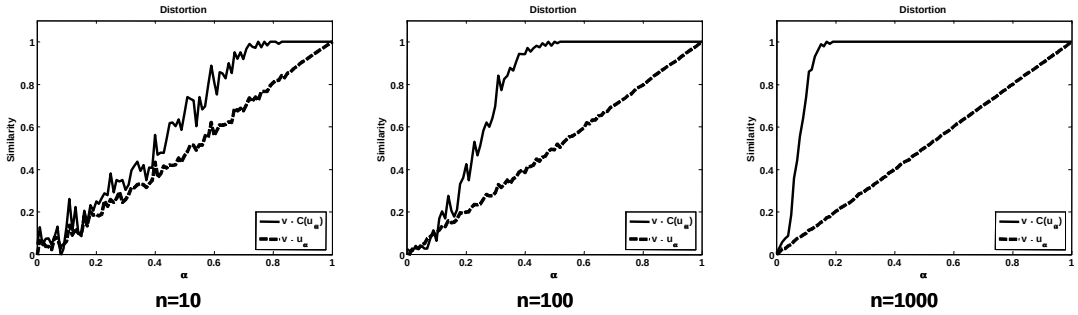

Figure 2.4: Scalar distortion tests. Each graph compares the similarity of a clean vector v with the vectors u_α and $C(u_\alpha)$. Each measurement is the mean of 100 trials.

the expected similarity between v and u_α is

$$\begin{aligned} \mathrm{E}[d(v, u_\alpha)] &= \mathrm{E}[v \cdot (\alpha v + \beta \eta)] \\ &= \alpha\, \mathrm{E}[v \cdot v] + \beta\, \mathrm{E}[v \cdot \eta] \\ &= \alpha. \end{aligned} \tag{2.13}$$

Figure 2.4 displays this linear relationship along with the performance improvement that comes with the cleanup operation in higher dimensions.

Unlike the superposition capacity measure, these graphs simultaneously capture the completely clean (α=1) and the completely noisy (α=0). Moreover, rather than applying tests at every integer step, we can measure the performance at any resolution. The graphs in Figure 2.4 are related to Figure 2.3 however. The number of random superpositions, k, implied by α is

$$k = 1/\alpha^2 \tag{2.14}$$

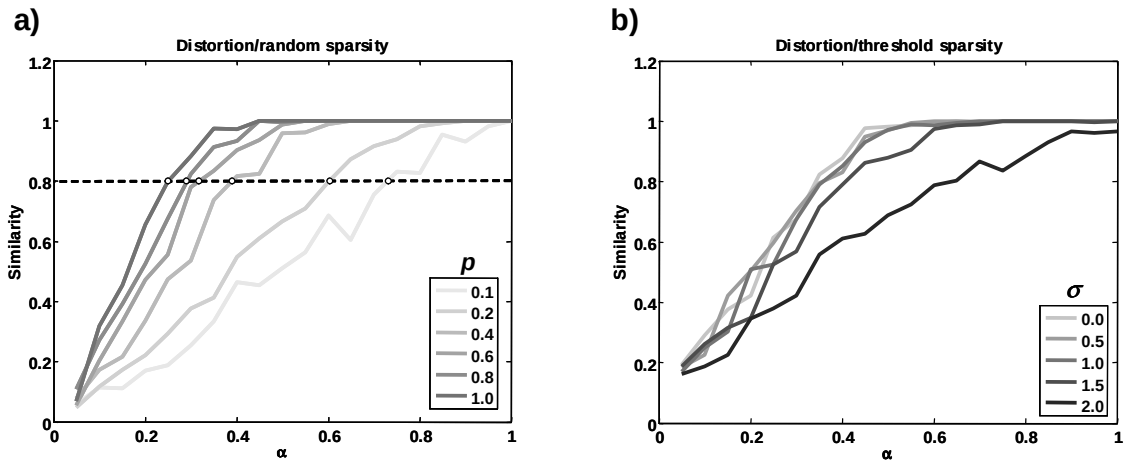

Figure 2.5: Performance under various sparsification methods and levels. HRR vectors are sparsified by **a)** randomly setting elements to zero with probability p (indicated in legend), or **b)** setting small elements to zero given a threshold (legend indicates the threshold as a fraction of $\sigma=\sqrt{1/n}$). The minimum intercepts of the threshold line (at 0.8) and the performance curve, can serve as an overall indicator of performance. In a), the cleanup is best with no sparsification. In b), a fraction of 1 gives a good tradeoff between sparsity (66%) and performance (near optimal).

2.5.4 Performance measures

With these tests defined, we can now conduct controlled experiments to determine optimal system parameters. For example, we can test the effects of sparsification on HRR vectors. Sparse vectors (those with many zero elements) are thought to be more energy efficient and biologically realistic. Vectors can be sparsified by either randomly setting elements to zero with probability p, or by setting small elements to zero given a threshold. Figure 2.5a shows, however, that sparsification should be avoided in some cases. (We will come back to these graphs and discuss sparsification in more detail in section 4.3).

Generally, it is desirable to use a single number to indicate performance rather

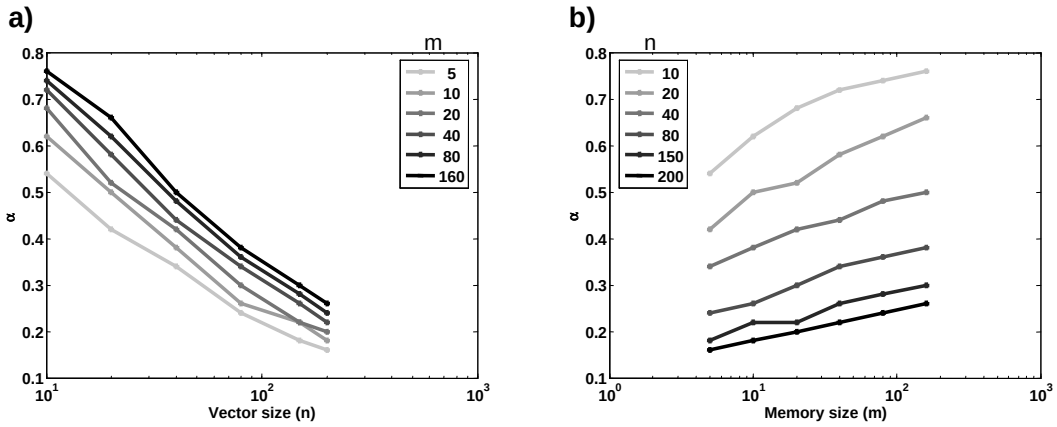

Figure 2.6: Scalability performance of max cleanup under a variety of dimensions, n, and number of clean vectors, m. Each sample point is the average α-intercept over 100 trials. **a)** Generally, performance improves with n and **b)** performance decreases slightly with m. Note that both graphs display the same data but at different viewpoints

than analyzing a set of curves. Assuming that, in the limit, the distortion curves are largely continuous and monotonically increasing, we can use the intercept of the performance curve with the recoverability threshold as an overall performance indicator (see Figure 2.5a).[4] Smaller α-intercepts indicate better cleanup performance.

The most generic parameters that influence the operation of a cleanup memory are the dimensionality of the vectors, n, and the number of clean vectors, m. Figure 2.6 uses the α-intercepts to study the impact of changing n and m. We can convert the α-axis into a superposition capacity axis using (2.14). Doing so confirms Plate's findings: the capacity, k, increases linearly with n and decreases slightly with m (Plate 2003, pg. 237).

[4]Since the curves in our simulations are neither completely smooth nor monotonic, we can have multiple intercepts. In these situations we simply take the smallest intercept.

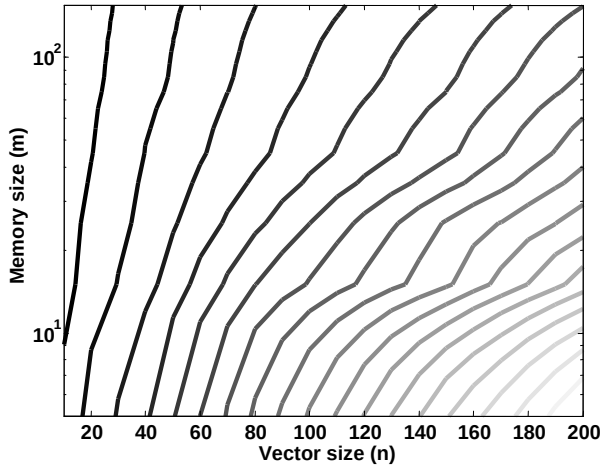

Figure 2.7: Scalability curves of max cleanup. Each line indicates a fixed superposition capacity, $k = 2, 4, ..., 16$ with lighter curves indicating larger k. The contours are roughly linear, suggesting that m is exponential in n.

Finally, taking the data from Figure 2.6, converting the α-axis to measure superposition capacity, and plotting the contour curves, yields Figure 2.7. This graph is useful because it characterizes the relationship between n and m across fixed k values. In this case, it suggests that m increases exponentially with n; i.e. we can store exponentially more vectors by increasing n linearly. Again, this exemplifies the benefits of computing in higher dimension.

2.6 Summary

Encoding concepts in memory using high-dimensional vectors is very different from traditional symbolic representation. Rather than using a single unit that stands for an object, this representation spreads information over many units. Distributed representations are tolerant to noise, applicable to many modalities, and conducive

to parallel processing. With the addition of binding and unbinding operators, they provide an elegant means for representing structure. As well, they support new forms of computing that can be a neurally plausible alternative to symbolic representation.

Many fixed-width schemes for representing structure have been invented that exploit the properties of distributed representations. All store information in very high dimensional vectors, all come with specific operations that can be applied to them, and all are susceptible to systemic noise. Effectively using these complex operations requires a cleanup memory; a structure that takes a noisy cue vector and returns the closest match to a set of known clean vectors. Such a structure is not only useful for noise reduction, but can also perform partial matching, pattern completion, and high-level object detection.

There are a number of forms of cleanup memories in the neural modeling field. Many are very specific to the particular architecture they support and none have been shown to be biologically plausible. Here, we define a very general cleanup memory that can be used with any scheme that employs fixed-width codevectors. The mathematical characterization of the problem admits a straightforward solution which has been implemented and tested. The results reiterate the power of computing in high dimensional spaces.

In this section, we have described a series of generic tools to test the effectiveness of cleanup operations. This will be useful later when we construct a variety of cleanup memories. These new memories, unlike the direct cleanup presented above, and unlike the cleanup memories developed elsewhere, will be based on biologically

realistic neurons.

Chapter 3

Neural Engineering Framework

3.1 Towards Neural Engineering

In an attempt to understand brain function, many investigators have turned to mathematical formalisms that are easily captured in computer algorithms. Popular attempts fall under the general category of Artificial Neural Networks. Adequately configured networks have shown success in the areas of pattern recognition, classification, and control (Rumelhart and McClelland 1986; Churchland and Seynowski 1994). With more 'neurons' and the right connections, it was hoped that these networks would function in a manner congruent to the human brain.

However, ANN neurons are not at all like real neurons and one should not expect too much from such networks, largely because it is unclear how to design them for many tasks. The brain, to put it mildly, is incredibly complicated. There are approximately 10^{10} neurons with at least 10^{13} connections—and there are thousands

of different *kinds* of neurons. Real neurons are dynamic, nonlinear devices operating in a noisy environment. They communicate with distinct action potentials (spikes) rather than continuous analog values. Ignoring these important implementational constraints may lead to cognitively unrealistic neural processing; difficult problems may be overlooked, easy implementations may be missed, and useful insight may remain obscured.

ANNs are popular because of their ease of implementation. Theoreticians, however, are becoming increasingly inspired by the mechanisms of real brains. More general and useful principles have started to shape our understanding of the design of neural systems. These new principles of representation, computation, and dynamics are a necessary step toward understanding cognitive functions. One synthesis of these principles is found in Eliasmith and Anderson (2003).

In this section, we describe the methodology of the Neural Engineering Framework (NEF) that is used here to construct biologically plausible neural networks for cleanup. We begin by defining the types of neurons modeled and the parameters that fall within biologically reasonable regimes. We then describe the characteristics of an ensemble (or population) of neurons and suggest a plausible population-level encoding and decoding relation with the relevant stimuli. Finally, we determine how to instantiate higher-level dynamics using this population of neurons.

The description of the NEF here is necessarily brief. For a fuller and broader explanation of the topics in this section consult Eliasmith and Anderson (2003), from which most of the derivations presented here originate. In addition, neural parameters that are particularly relevant for cleanup networks are highlighted.

3.2 Neural cell dynamics

To model sub-symbolic operations at a neural level, we must first convert a code-vector into an input current. This can be done with a generic linear transformation,

$$J_i(\mathbf{x}) = \alpha_i \left\langle \tilde{\phi}_i \cdot \mathbf{x} \right\rangle + J_i^{bias} + \eta_i \tag{3.1}$$

where $J_i(\mathbf{x})$ is the input current to neuron i, $\mathbf{x}$ is the codevector encoded by the neuron, α_i is a gain and conversion factor, J_i^{bias} is a bias current that accounts for background activity, and η_i models neural noise. The preferred direction vector, $\tilde{\phi}_i$, models the fact that each neuron is known to have varying sensitivity to particular dimensions (Georgopoulos et al. 1986).

The population of neurons in this model is a heterogeneous collection of leaky integrate-and-fire (LIF) neurons. The time course of the somatic voltage in response to the current in (3.1) evolves with the dynamics captured by (Koch 1998),

$$dV_i/dt = -\left(V_i - J_i(\mathbf{x})R\right)/\tau_i^{RC} \tag{3.2}$$

where V_i is the somatic voltage, R is the leak resistance, τ_i^{RC} is the RC time constant. The system is integrated until the membrane potential, V_i, crosses the neurons threshold, V_{th}, at which point a $\delta(t - t_{in})$ spike is generated and V_i is reset to zero for the duration of the refractory period, τ_i^{ref} (see Figure 3.1).

To model a diverse population of typical neurons, we randomly select the neural parameters. The gain and bias current, α_i and J_i^{bias}, are chosen such that the

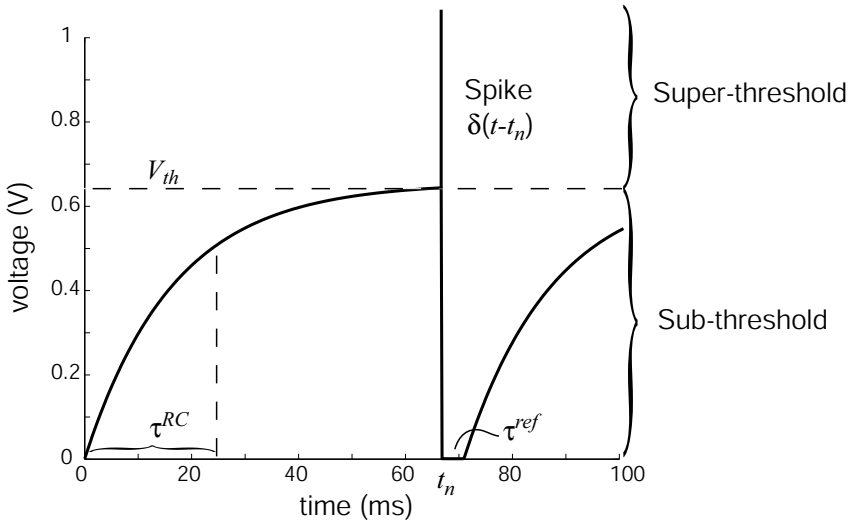

Figure 3.1: Time course of a leaky integrate-and-fire (LIF) neuron with constant input (from Eliasmith and Anderson (2003) ©The MIT Press, reproduced with permission).

maximum firing rates are between 200-400 Hz. The RC time constant is τ_i^{RC}=5 ms and the refractory period is set to τ_i^{ref}=1 ms. In addition, during the simulation, independent Gaussian noise, η_i=N(0, 0.1), is injected into the soma to account for various sources of neural noise.

The preferred direction vectors, $\tilde{\phi}_i$, are particularly important for the cleanup networks presented in this thesis. Generally the $\tilde{\phi}_i$'s are drawn randomly from a uniform distribution around the unit hyper-sphere. However, in some cases, neurons are known to act independently of others. To model this, independent neurons are set to have orthogonal preferred directions, while dependent neurons point along a shared vector. This effectively divides an ensemble into a number of subpopulations while increasing the representational accuracy of each independent component.

3.3 Population encoding and decoding

These parameters now define a biologically plausible, heterogeneous population of spiking neurons. Next we must address how these neurons can be used to represent and process information about codevectors. This can be done by characterizing 1) how such information is encoded into the population of neurons, and 2) what information can be decoded from the population.

Specifically, we understand communication between populations to be characterized in terms of a nonlinear encoding process and a linear decoding process. Encoding involves converting a quantity, $\mathbf{x}(t)$, into a spike train:

$$\sum_n \delta(t - t_{in}) = G_i \left[J_i(\mathbf{x}(t)) \right] \tag{3.3}$$

where $G_i[\cdot]$ is the nonlinear function describing the rate at which spikes are produced (see Figure 3.2 for typical LIF responses), and J_i is the current in the soma of the cell. Note that both the current and nonlinearity are explicitly described by (3.1) and (3.2) respectively. Equation (3.3) captures the full encoding process from a high-dimensional variable, $\mathbf{x}$, to a one dimensional soma current, J_i, to a train of spikes, $\delta(t - t_{in})$.

To understand how a neural system might use the information carried in a spike train, we must be able to characterize a neurally plausible decoding. To do so we need to understand how this information can be converted from spike trains back into a relevant quantity. Somewhat surprisingly, a plausible means of characterizing this decoding is as a *linear* transformation of the spike train (Eliasmith and

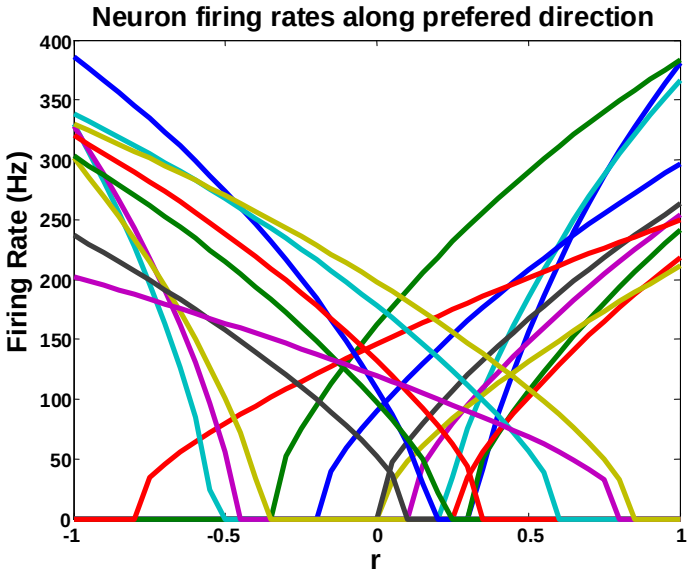

Figure 3.2: A random sampling of LIF tuning curves. Each curve describes the spiking response, $G_i[r]$, of an individual neuron.

Anderson 2003). Specifically, we can estimate the original stimulus vector $\mathbf{x}(t)$ by decoding an estimate, $\hat{\mathbf{x}}(t)$, using a linear combination of filters, $h_i(t)$, weighted by the decoding vector, ϕ_i:

$$\hat{\mathbf{x}}(t) = \sum_{in} \delta(t - t_{in}) * h_i(t)\phi_i = \sum_{in} h_i(t - t_{in})\phi_i \tag{3.4}$$

where '$*$' indicates convolution (see Figure 3.3). These $h_i(t)$ are thus linear decoding filters which, for reasons of biological plausibility, are taken to be the post-synaptic currents (PSCs) in the subsequent neuron (Eliasmith and Anderson 2003).

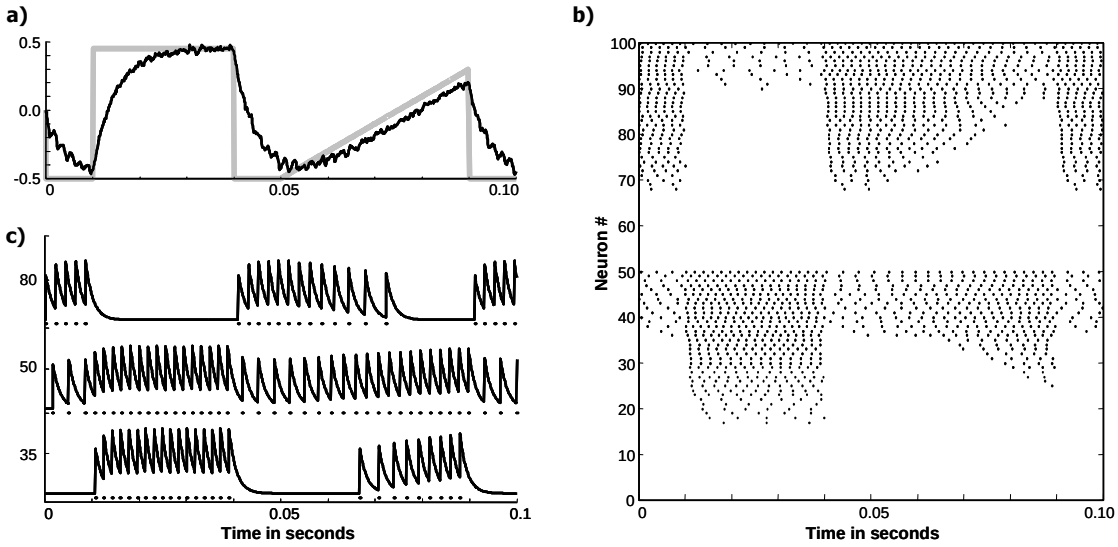

Figure 3.3: Population encoding and decoding of a square pulse followed by a ramp signal. **a)** The input (gray line) and the decoding estimate, $\hat{\mathbf{x}}(t)$ (black line), using a population of 100 one dimensional LIF neurons. **b)** The encoding produces a raster of spike trains, $\delta(t-t_{in})$, where t_{in} indicates the nth spike for neuron i. Neurons are separated at i=50 into 'on' and 'off' neurons ($\tilde{\phi}_i$=+1 and -1 respectively) and then sorted by firing onset (the value of $\mathbf{x}$ for which $G_i\left[J_i(\mathbf{x})\right]$=0). **c)** As an example, the rasters of neurons i=80, 50, and 35 are plotted with their PSC filtered counterpart, $h_i(t-t_{in}) = \delta(t-t_{in}) * h_i(t)$. Finally, the weighted sum all the filtered trains, $\sum_{in} h_i(t-t_{in})\phi_i$, yields the decoding estimate (black line in a)).

3.4 Neural computation

After (Eliasmith and Anderson 2003), the optimal linear population decoding vectors, ϕ_i, can be found by minimizing the error,

$$\begin{aligned} E &= \frac{1}{2}\left\langle [\mathbf{x}(t) - \hat{\mathbf{x}}(t)]^2 \right\rangle_{\mathbf{x},t} \\ &= \frac{1}{2}\left\langle \left[\mathbf{x}(t) - \sum_{in} \left(h_i(t - t_{in}) + \eta_i \right) \phi_i \right]^2 \right\rangle_{\mathbf{x},t,\eta} \end{aligned} \tag{3.5}$$

where $\langle \cdot \rangle_{\mathbf{x}}$ denotes integration over the range of $\mathbf{x}$, and η_i models the expected noise. By optimizing with random noise, we ensure that fine tuning is not a concern, since the decoding weights will be robust to minor fluctuations (i.e., 10% of the maximum spike frequency in this model).

In some of the cleanup networks presented here, we want a population to transform the signal it receives into something useful. Rather than encoding a variable, we'd like it to represent a function. We can calculate the appropriate decoders by simply minimizing over the required function (Eliasmith and Anderson 2003),

$$E = \frac{1}{2}\left\langle \left[f\left(\mathbf{x}(t)\right) - \sum_{in} \left(h_i(t - t_{in}) + \eta_i \right) \phi_i^{f(\mathbf{x})} \right]^2 \right\rangle_{\mathbf{x},t,\eta} \tag{3.6}$$

where we denote $\phi^{f(\mathbf{x})}$ as the decoding vector for computing $f(\mathbf{x})$. Notably, this calculation can be high dimensional. The details of this calculation are explained in Appendix A along with a sampling technique that the cleanup ensembles take advantage of.

3.5 Higher level dynamics

A population of neurons can represent variables through non-linear encoding and linear decoding, and perform computation with appropriate decoding weights. Now, we can place these populations together to form higher level dynamic systems (Eliasmith and Anderson 2003). Specifically, we can write the relevant dynamics of the network in control theoretic form, i.e., employing one of the state equations that comprises the foundation of modern control theory,

$$\dot{\mathbf{x}}(t) = \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t) \tag{3.7}$$

where $\mathbf{A}$ is the dynamics matrix, $\mathbf{B}$ is the input matrix, $\mathbf{u}(t)$ is the input or control vector, and $\mathbf{x}(t)$ is the codevector (see Figure 3.4a for a graphical depiction of this equation). In general, these matrices and vectors can describe a wide variety of dynamics.

Initially, this high-level characterization is divorced from neural-level, implementational considerations. However, it is possible to account for the intrinsic neural dynamics by converting this characterization into a neurally relevant one (Figure 3.4b). That is, assuming a model of postsynaptic currents (PSCs) given by $h(t) = \tau^{-1}e^{-t/\tau}$, we can derive the following relation between Figure 3.4a and Figure 3.4b (Eliasmith and Anderson 2003):

$$\begin{aligned} \mathbf{A}' &= \tau\mathbf{A} + \mathbf{I} \\ \mathbf{B}' &= \tau\mathbf{B} \end{aligned} \tag{3.8}$$

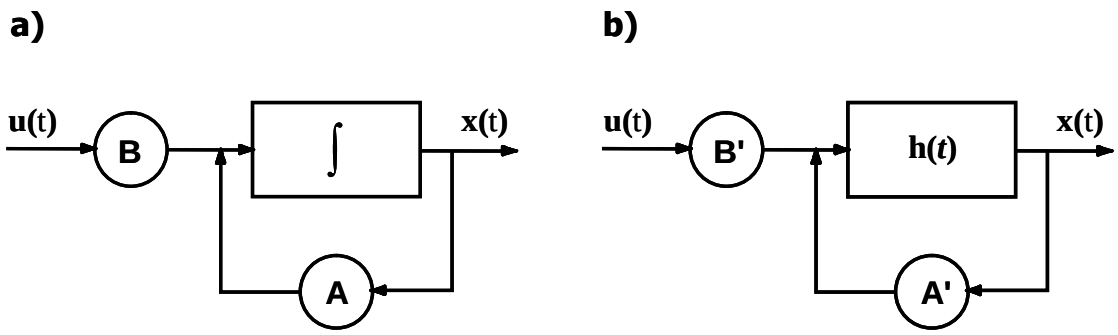

Figure 3.4: Control theoretic block diagram for time invariant linear systems. **a)** Diagram of the standard state equation, $\dot{\mathbf{x}}(t) = \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t)$. **b)** Diagram of neural state equation, $\mathbf{x}(t) = h(t) * [\mathbf{A}'\mathbf{x}(t) + \mathbf{B}'\mathbf{u}(t)]$. Note that the dot is dropped in the neural equation since the dynamics of the filter, $h(t)$, accounts for integration. We can convert a) into b) using (3.8).

So, in the time domain, our description of the high-level neurally plausible dynamics becomes

$$\mathbf{x}(t) = h(t) * [\mathbf{A}'\mathbf{x}(t) + \mathbf{B}'\mathbf{u}(t)] \,. \tag{3.9}$$

Notably, this transformation is general, and assumes nothing about the form of $\mathbf{A}$ or $\mathbf{B}$. So, given any behavioural system defined in the form of (3.7) (e.g. integrators, oscillators, attractors, controlled filters, etc...), it is possible to construct the neural counterpart by solving for $\mathbf{A}'$ and $\mathbf{B}'$.

To incorporate this high-level description of the dynamics with our previous characterization of the neural representation, we must combine the dynamics of (3.9), the encoding of (3.3), and the population decoding of $\mathbf{x}$ and $\mathbf{u}$ from (3.4). For this purpose we take $\hat{\mathbf{x}} = \sum_{jn} h_j(t - t_{jn})\phi_j^{\mathbf{x}}$ and $\hat{\mathbf{u}} = \sum_{kn} h_k(t - t_{kn})\phi_k^{\mathbf{u}}$, which

gives

$$\begin{aligned}
&\sum_n \delta(t - t_{in}) \\
=&G_i \left[\alpha_i \left\langle \tilde{\phi}_i \mathbf{x}(t) \right\rangle + J_i^{bias}\right] \\
=&G_i \left[\alpha_i \left\langle \tilde{\phi}_i h(t) * [\mathbf{A}'\hat{\mathbf{x}}(t) + \mathbf{B}'\hat{\mathbf{u}}(t)] \right\rangle + J_i^{bias}\right] \\
=&G_i \left[\alpha_i \left\langle \tilde{\phi}_i h(t) * \left[\mathbf{A}' \sum\nolimits_{jn} h_j(t - t_{jn})\phi_j^{\mathbf{x}} + \mathbf{B}' \sum\nolimits_{kn} h_k(t - t_{kn})\phi_k^{\mathbf{u}}\right] \right\rangle + J_i^{bias}\right].
\end{aligned} \tag{3.10}$$

It is important to keep in mind that the temporal filtering is only done once, despite this notation. That is, $h(t)$ is the same filter as that defining the decoding of both $\mathbf{x}(t)$ and $\mathbf{u}(t)$. More precisely, this equation should be written as

$$\begin{aligned}
&G_i \left[\alpha_i \left\langle \tilde{\phi}_i \left[\mathbf{A}' \sum\nolimits_{jn} h_j(t - t_{jn})\phi_j^{\mathbf{x}} + \mathbf{B}' \sum\nolimits_{kn} h_k(t - t_{kn})\phi_k^{\mathbf{u}}\right] \right\rangle + J_i^{bias}\right] \\
=&G_i \left[\sum\nolimits_{jn} \omega_{ij} h_j(t - t_{jn}) + \sum\nolimits_{kn} \omega_{ik} h_k(t - t_{kn}) + J_i^{bias}\right]
\end{aligned} \tag{3.11}$$

where $\omega_{ij} = \alpha_i \mathbf{A}' \phi_j^{\mathbf{x}} \tilde{\phi}_i$ and $\omega_{ik} = \alpha_i \mathbf{B}' \phi_k^{\mathbf{u}} \tilde{\phi}_i$ are the recurrent and input connection weights respectively. These weights will now implement the dynamics defined by the control theoretic structure from (3.7) in a neurally plausible network. Notably, this characterization is very general (e.g., Eliasmith (in press) provides a comprehensive account of controlled spiking attractor networks (i.e., point, line, ring, plane, cyclic, and chaotic attractor networks) using these methods).

In theory, such dynamics can be useful in cleanup memories with recurrent connections. However, simulations have shown that recurrent networks in high dimensions are generally unstable. If such networks settle, it is often at an undesirable

attractor point. Moreover, recurrent networks are incapable of multiassociation, if they are constructed to settle at predefined attractors. Thus, for the circuits presented here, we ignore the connections provided by matrix $\mathbf{A}'$ and employ the feedforward connections induced by $\mathbf{B}'$.

We note that the weights can take on negative values, which is not biologically realistic. Other work has developed a generic technique that employs interneurons to make all weights the same sign (Eliasmith and Anderson 2003; Parisien and Eliasmith forthcoming). However, this process is complex, does not substantially affect the results of the network, and would take us far afield. So, for the models presented here we have allowed negative weights.

Circuits derived in this manner were modeled using the Neural Engineering Simulator (NES), which is available as open source (http://www.sf.net/projects/nesim).

3.6 Summary

Constructing biologically plausible neural networks requires more than abstract characterization of neurons as found in ANNs; it requires an understanding of the principles of neural representation, computation, and dynamics. The Neural Engineering Framework described here embodies such principles. We have shown how an ensemble of neurons represents a variable through nonlinear encoding and linear decoding, can perform computation (or transformations) through suitable decoding weights, and can carry out a variety of dynamics defined by standard control theory.

This characterization presents new challenges, but as we will see, also offers new solutions to problems that are not available by conventional means.

Chapter 4

Cleanup Memory in NEF

We now apply the neural engineering principles discussed in Chapter 3 to the cleanup problem defined in Chapter 2. This union offers a few extra challenges not found in standard artificial approaches. First, in addition to the inherent noise of distributed representations, *more* noise will be added to simulate the background environment of the brain (e.g., spike jitter, thermal fluctuations, neurotransmitter variations, etc...).

Second, the network simulations evolve over time. Operations are not expected to occur instantaneously, but are delayed. Figure 4.1 displays the temporal and noisy aspects of neural representation. To compare with the direct cleanup memory, we take the 'answer' from these networks to be the final vector output. This answer can be made more stable with another ensemble acting as a low pass filter (Eliasmith in press), or using a broader filter (e.g. using a longer τ in the PSC filter, $h(t) = \tau^{-1}e^{-t/\tau}$). So all results presented here are lower bounds on achievable

accuracy. Also, the dot product calculations in (2.8) will vary, and is only accurate with respect to the amount of intrinsic and background noise in the system. Note that the approximation to the vector improves with the number of neurons. In fact, it has been shown (Eliasmith and Anderson 2003), that the error in the signal drops as $1/N$, where N is the total number of neurons used in representing a value.

Third, due to constraints on computational resources, there is a practical limit to the number of neurons per ensemble one can numerically simulate. The total number of neurons in an ensemble representing a vector can be written as $N = nN_e$, where n is the size of the vector and N_e corresponds roughly to the number of neurons per vector element. One then must make a tradeoff between the size of the vector and the accuracy of the representation of the elements. The effects of changing N_e are portrayed in Figure 4.1. In order to generate comparable simulations over many trials, a practical maximum of N for available computational resources is 1000 neurons; N_e is typically 10, leaving a maximum dimension of n=100. This dimension is noticeably smaller than that typically used, which is usually greater than 500.

Thus, due to time and memory constraints, these neural simulations will be of relatively low quality cleanup memories. However, by presenting systems that exhibit performance trends similar to the ideal solution at low dimensions, one can extrapolate similar performance at higher dimensions.

Like other frameworks, the NEF admits many solutions to a given problem. Here we present three networks that perform the cleanup operation: the ANN inspired butterfly network, the function network which calculates the cleanup, and

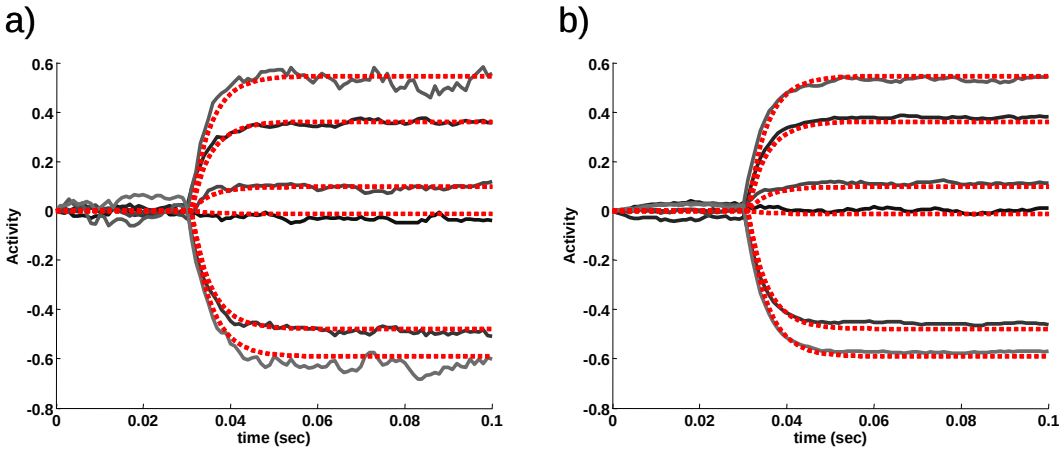

Figure 4.1: Neural ensemble representation of a 6-dimensional vector using **a)** N_e=5 and **b)** N_e=10 neurons per element. The total number of neurons is $N = 6N_e$. At t=0.03 s, the ensemble goes from representing the zero vector to a random HRR vector. At t=0.05, the signal has stabilized. We take the final output to be the last signal value, in this case, at t=0.1. Dashed lines indicate the expected signals.

the sparse network which produces sparse encodings of clean vectors.

4.1 Butterfly nets

We first present a neural network that is inspired by the traditional bottleneck or butterfly networks (Figure 4.2). Butterfly networks typically reduce the dimensionality of a vector by mapping it into a lower dimensional space. In this respect, they are like traditional autoencoders, which learn sparse representations using backpropagation. This circuit does not use any feedback, however, and we will return to the issue of learning in section 4.4.

4.1.1 Butterfly network

Recall that an approximate cleanup memory can be described as

$$C_\mu(c) = \mu(cM^T)M = \mu(s)M\,. \tag{4.1}$$

One can take this definition and convert it into a three layer feed-forward network as shown in Figure 4.2. This in turn can be converted to a spiking neural network, but we must note a few differences in terminology. The 'layers', in this case, are populations of neurons, the 'units' are vector elements represented by the neurons, and the 'weights' are connection strengths between vector elements. These weights are referred to as coupling weights to avoid confusion with neuron-to-neuron connection weights described in (3.11).

As illustrated in Figure 4.2, the memory matrix is used to transform the cue vector to and from the similarity space. Using M in this way effectively stores the memory in the coupling weights. The middle layer represents the similarity vector and also computes $\mu(s)$. The connection weights are easily derived using the NEF (see equation (3.11)): $\omega_{ij} = \alpha_i M^T \phi_j^{\mathbf{c}} \tilde{\phi}_i$ between ensembles $\mathbf{C}$ and $\mathbf{S}$, and $\omega_{jk} = \alpha_j M \phi_k^{\mu(\mathbf{s})} \tilde{\phi}_j$ between $\mathbf{S}$ and $\mathbf{X}$. The decoding vector, $\phi^{\mu(\mathbf{s})}$, can be calculated using (3.6).

The nature of this network offers two optimizations. First, the encoding of a vector element in each population does not depend on the encoding of any other element. Therefore, we can improve the overall representational accuracy by forcing the encoding to be independent. This is realized by simply adjusting the preferred

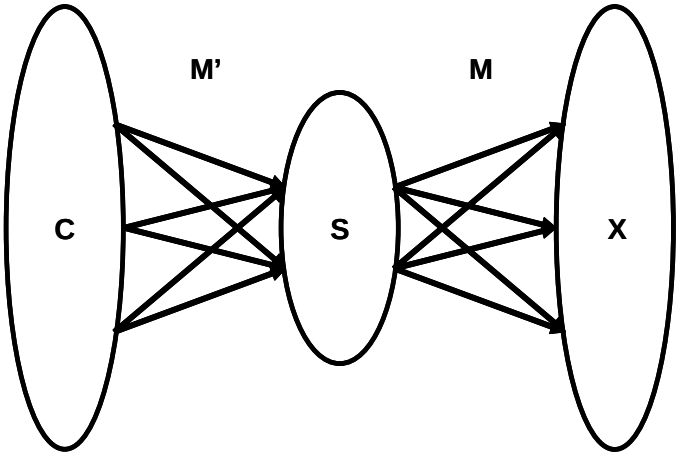

Figure 4.2: Butterfly cleanup memory. The n-dimensional cue ensemble, **C**, is connected to the similarity ensemble, **S**, using the memory matrix, M^T, as the coupling matrix. This effectively computes the dot product of the cue with every vector in M. Ensemble **S** not only represents this m-dimensional similarity vector but also computes the membership weight, $\mu(s)$. The second set of weights uses the function output of **S** to create the weighted sum. The result, stored in **X**, is the cleaned vector.

direction vectors, $\tilde{\phi}_i$, to only point along one particular dimension. Second, for HRR vectors, the distribution of each element is $N(0, 1/n)$. Hence, rather than minimizing (3.6) over the usual range, $[-1, 1]$, we can tune our neurons to better represent the values between $[-\sqrt{1/n}, \sqrt{1/n}]$.

The caliber of this network under various parameters is shown in Figure 4.3. The number of neurons in a population is significant up to a point. While the performance increases with N_e, there are diminishing returns. The performances for N_e=5, 10, 20 are nearly indistinguishable. Thus, setting N_e greater than 10, in these simulations, is unproductive.

It is important to realize that the performance of these networks are plagued by three distinct sources of noise: external noise, which we impose upon the sys-

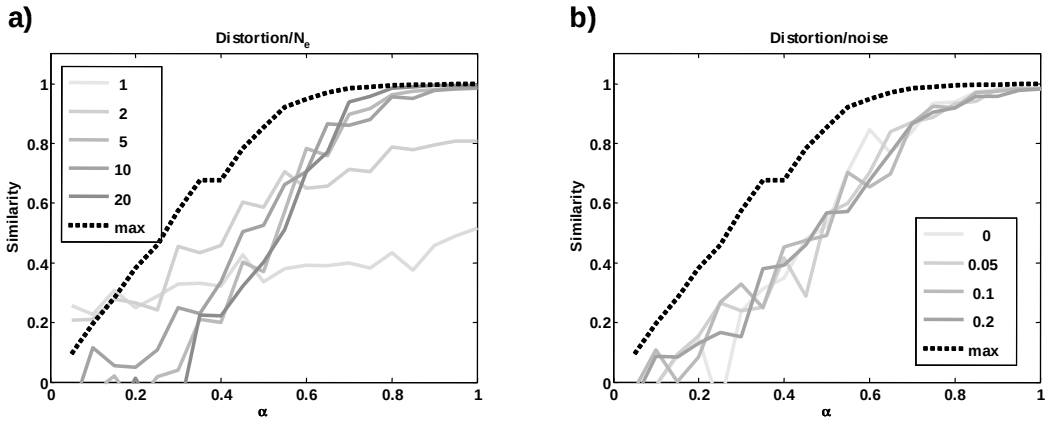

Figure 4.3: Butterfly network performance under varying N_e and noise. The optimal non-neural performance (using equation (2.6)) is marked with a dashed line. **a)** Cleanup quality converges toward the optimum with more neurons. N_e=10 will be sufficient in these simulations. **b)** The network performance is degraded but also robust to various levels of background noise.

tem; internal representation noise, which is the inaccuracies engendered by neural decoding; and internal SDR noise, which is the noise inherent in SDR operations.

As expected, external noise does diminish the overall performance of the system. One would further expect that increasing levels of noise would proportionally degrade these networks. Yet, for realistic noise values ($\sigma^2 \leq 20\%$) no significant trends were found. This is because the ensembles were designed to be robust to external noise (see eq. (3.5)), and hence show no markedly adverse affects (Figure 4.3b).

Usually, the effects of this external noise are diminished as $1/N_e$. However, using a fixed N_e with independent neurons incurs a fixed amount of external and internal representation noise in the network. This has a profound impact on the scalability of the network since they dominate the internal SDR noise that one

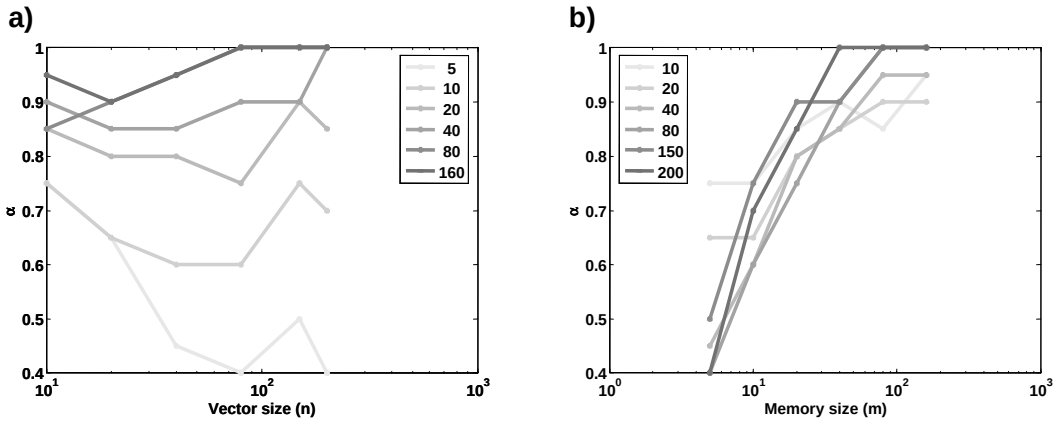

Figure 4.4: Scalability of butterfly network. Unlike SDR operations **a)** the performance does not improve with n, and **b)** performance is significantly impaired by increasing m.

expects to drop as $1/n$. Thus, while performing relatively well at low dimensions, it does not improve with higher dimensions (Figure 4.4a). Worse yet, the quality of the cleanup is destroyed by increasing m (Figure 4.4b). A look at the overall scalability in Figure 4.5 shows that the exponential relationship between n and m has become flat. Thus, in this circuit, the exponential memory capacity of HRR vectors does not hold when using plausible neurons. This suggests that this is not an appropriate architecture for cognitive systems.

4.2 Function Cleanup

The butterfly cleanup has strong ties to traditional neural networks. It is, in fact, formulated as a three layer feed forward network where the connections are coupling weights specified by the memory matrix, M. This network could have been an artificial neural network if one replaces the neurons with artificial neurons and

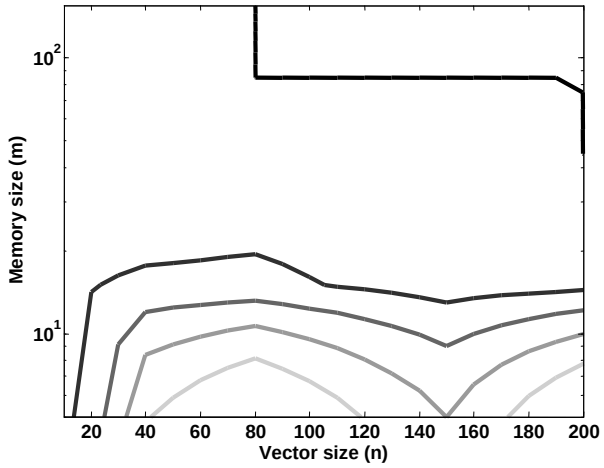

Figure 4.5: Storage capacity graph (see 2.7) of butterfly nets. Storage capacity does not increase with n. The normally exponential storage capacity is extinguished by network noise.

casts the membership function as an activation function.[1]

Now, we present a network that has no ANN counterpart, that adheres more closely to the principles of Neural Engineering, and performs better than the butterfly network.

4.2.1 Function sampling

The NEF allows for a more direct approach to constructing cleanup memories. Recall that an ensemble can calculate an arbitrary transformation using suitable weights. Therefore, by defining an appropriate cleanup function, we can calculate $\phi^{C(x)}$. Thus, rather than using three populations of neurons, we can combine the

[1]In fact, it could be argued that all ANNs can be redefined into a neurally plausible network. Of course, these artificial-cum-spiking neural networks would still have to deal with the added problems of noise.

entire butterfly network into a single ensemble.

Essentially, we require a mapping, C, such that, not only does $C(v_i) = v_i$, but also that $C(v_i + \eta) = v_i$, where η is a small noise vector. This function can be approximated using (3.6) and a sample of points (see Appendix A.2). The standard sampling method uses points that are randomly and uniformly distributed over the space of codevectors, thus producing a relatively full mapping of C. However, many sample points would be required to ensure that all v_i are reasonably approximated.

Alternatively, one could concentrate the sampling around the clean vectors, thereby improving the representational accuracy around the clean vectors. This approach suggests a learning technique that does not involve the calculation of $C(x)$, but only involves the presentation of the clean vectors. A comparison of these two sampling techniques (Figure 4.6) concludes that the concentrated sampling is superior.

Since the gaps in between vectors are not sampled, it would seem that they would be ill-defined. However, given the smoothness of the LIF tuning curves, the mapping of C in these regions will be smooth combination of the nearby points. Experimental results verify that novel compositions can be recovered; i.e. $C(v_i + v_j + \eta) = v_i + v_j$ even though the vector, $v_i + v_j$, is never explicitly stored in memory.

Figures 4.7 and 4.8a show the results of the scalability analysis. The performance is much better than the butterfly circuit, but is not as good as the optimal cleanup. Given that the network is calculating $C(x)$, the only source of discrepancy is internal representative noise. Indeed, the circuit compares well to the optimal cleanup with noise added to the cue (Figure 4.8b).

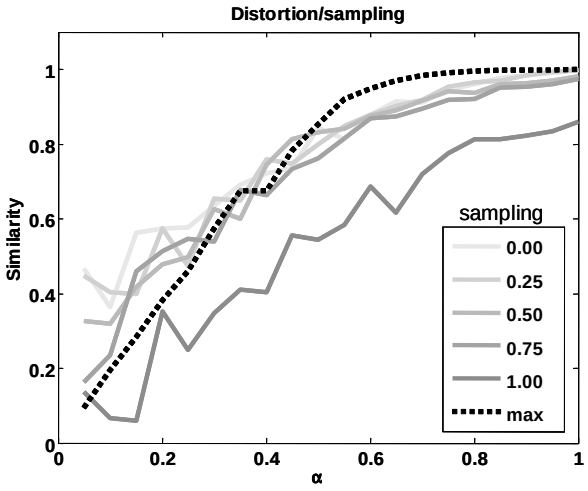

Figure 4.6: Function cleanup across two sampling techniques. The legend indicates the probability of sampling from one of two techniques. At 0, the sampling is concentrated around the clean vectors combination, while at 1 the sampling is random and uniform. The concentrated sample, even in small amounts, produces results very close to the optimal.

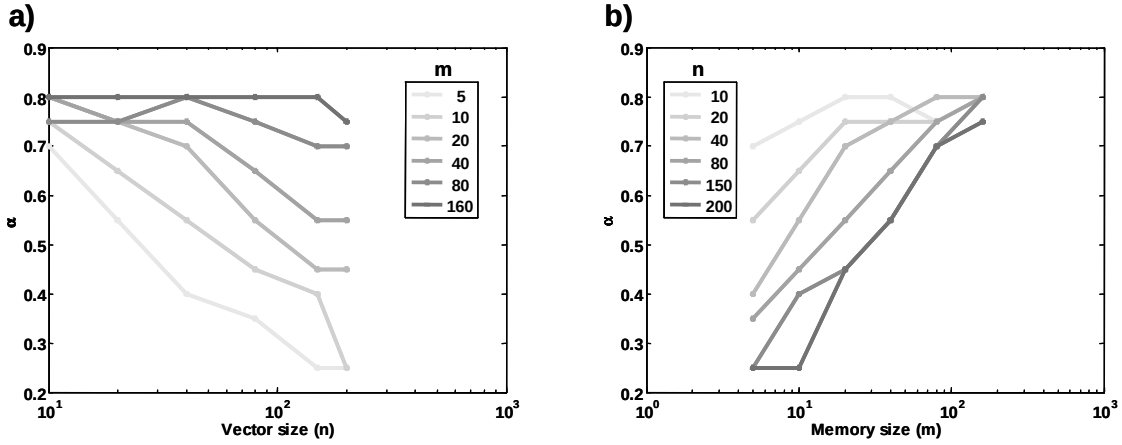

Figure 4.7: Function cleanup performance under varying n and m. Performance does improve with n, but is drastically curtailed by increasing m.

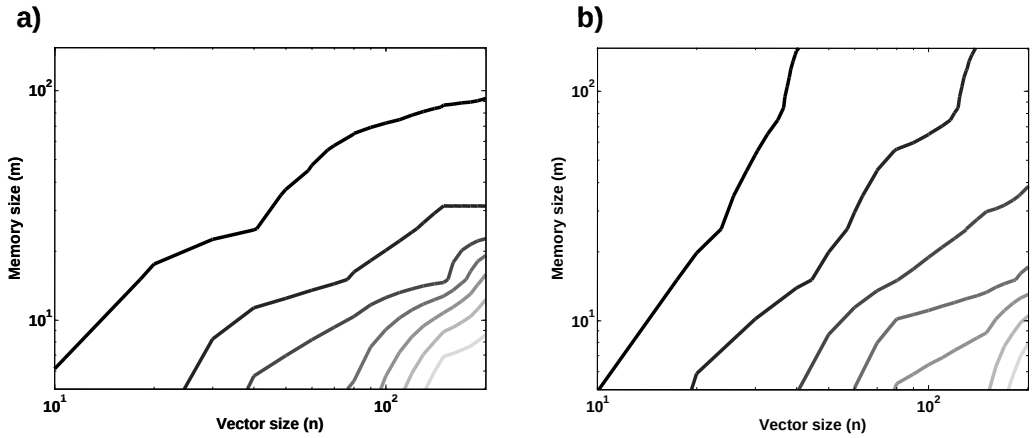

Figure 4.8: Scalability analysis of function cleanup. **a)** The curves suggest a linear rather than an exponential relationship between n and m. This is consistent with **b)** the optimal solution with 10% noise.

Aside from cleaning up, this approach can be used for heteroassociative networks as well. Rather than producing a clean version of the input, the system transforms the input into another vector; i.e. $C(x_i + \eta) = C(x_i) = y_i$, where x_i and y_i are associated pairs.

4.3 Sparse Cleanup

Recent research suggests that neural encodings are often sparse (for an review see Olshausen and Field (2004)). 'Spareseness' is used variously in the literature to refer to two separate concepts: sparse vectors and sparse coding. Sparse vectors, introduced earlier in section 2.5.4, are vectors with few non-zero elements. Sparse coding is a neurobiological term that implies that few neurons are active in representing a value. An extreme version of sparse coding is the localist representation, which uses a single neuron to represent an item.

It is generally assumed that sparse vectors imply sparse coding. However, this is not necessarily the case for biologically plausible neurons. Recall from (3.3) that the spiking activity is a function of the input current and the tuning curve

$$\begin{aligned}\sum_n \delta(t - t_{in}) &= G_i\left[J_i(\mathbf{x})\right] \\ &= G_i\left[\alpha_i \left\langle \tilde{\phi}_i \cdot \mathbf{x} \right\rangle + J_i^{bias} + \eta_i\right]\end{aligned} \tag{4.2}$$

Note, that it requires current to represent $\mathbf{x} = \mathbf{0}$ (the sparsest vector), since $J_i(\mathbf{0}) = J_i^{bias} + \eta$. Furthermore, even when $J_i(\mathbf{x}) = 0$, $G_i\left[0\right]$ may still produce spikes. Figure 3.2, in the previous chapter, depicts a few tuning curves that will spike with zero current.

Given a cleanup memory that is inundated with random DR vectors, we expect the typical output to be the zero vector. Hence, it would be very energy efficient to not generate spikes when representing zero. Naturally, under this setting, sparse coding will lead to a sparse vector and vice versa. Figure 4.9 exhibits a few tuning curves that do not cross the origin and hence rarely spike at 0. Sparseness (in both senses) can be increased by raising the minimum onset value. Figure 4.9 sets this value to 0.2. A reasonable tradeoff between performance and sparsity (recall Figure 2.5b) is one standard deviation of the element distribution, $\sigma{=}\sqrt{1/n}$ (making 66% of vector zero). When set, the ensemble is incapable of representing values in the range of $\pm\sigma$, thus simultaneously producing a sparse vector that is also sparsely coded.

In high dimensions, $E[\tilde{\phi}_i \cdot \mathbf{x}] = 0$, since $\tilde{\phi}_i$ and $\mathbf{x}$ are randomly chosen and the elements are distributed with a mean of 0. Thus most vectors, even clean ones, will

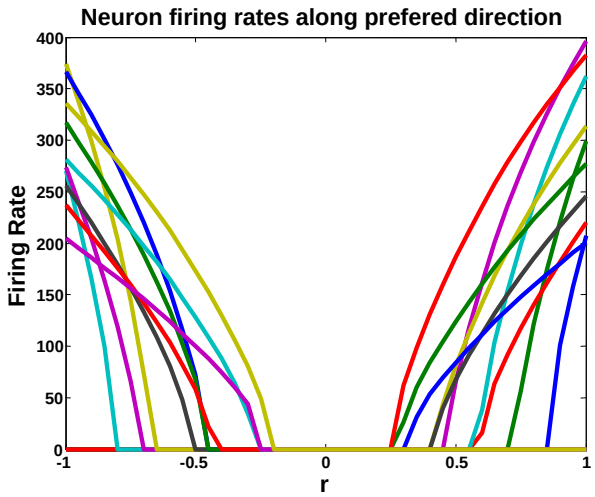

Figure 4.9: A sampling of LIF tuning curves that do not spike when $r \in [-0.2, 0.2]$. Such curves can be used to sparsify HRR vectors.

produce nearly indistinguishable firing patterns. We are free, however, to point $\tilde{\phi}_i$ in any desired direction. An immediate choice is to point them in the same direction as the clean vectors; i.e $\tilde{\phi}_i = v_j$, where j is chosen randomly. This works well since the neurons will fire maximally when presented with a clean vector, at which point, only a few neurons will be active. However, when the number of neurons is close to the number of clean vectors ($N = nN_e \approx m$), as in some of the simulations here, the output becomes highly localist, since only a few neurons are used to represent a vector. Such an encoding is brittle and does not represent its vector well under noise.

A less sparse choice is to set the preferred direction vectors to a random superposition of a few clean vectors; e.g. $\tilde{\phi}_i = v_j + v_k + v_l$. The output will be less sparse since more neurons will participate in the encoding, but it will also be less brittle, more resilient to noise, and produce lower firing rates.

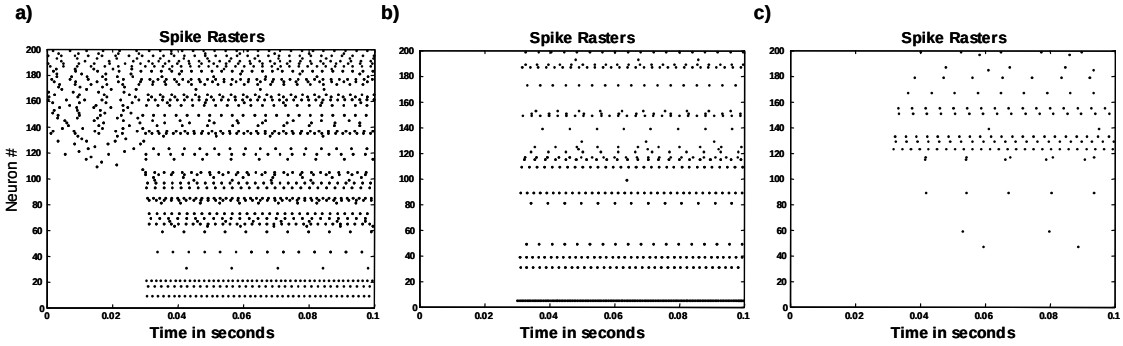

Figure 4.10: Sparsification of spike rasters. **a)** Typical spike raster of the function cleanup memory. Note the background firing during $t \leq 0.3$ s—even though it is representing **0**. **b)** The sparse cleanup, however, has no code for **0** and only a few neurons (22%) spike when presented with a clean vector. **c)** When presented with an unknown vector, there is virtually no activity. Only 16% produce spikes and they are at a very low frequency.

The cleanup performance of the sparse network is comparable to the function network (not shown). In addition to sparse coding, this network also leads to a natural learning strategy which we will discuss in the next section.

4.4 Learning

We have assumed, so far, an *a priori* lexicon of 'true' clean vectors. But where do these clean vectors come from? Presumably, they are either inborn or originate from experiences in the world. Individual experiences are too numerous, however, and it would be impractical to store them all in a single matrix. Indeed, such exhaustive storage is particularly pointless when the input is noisy or incomplete.

Also, the performance of the cleanup networks depend on the distribution of the stored codevectors. Vectors that are close together are harder to distinguish

from vectors that are far apart. In the tests performed earlier, these vectors were randomly and uniformly distributed, hence nearly orthogonal. We expect vectors that encode real world data to have some structure and will therefore be densely packed in some areas.

The ANN literature emphasizes the need for unsupervised learning. To resolve this issue, many learning rules have been found that capture the structure of the input data. Of course, these rules are based on unrealistic neurons, and hence invoke structures and methods with little biological basis.

In general, a learning rule updates connection weights between two populations, which in our networks is formulated as

$$\omega_{ij} = \alpha_i M \tilde{\phi}_i \phi_j. \tag{4.3}$$

Attempts at creating a neurally realistic, local learning rule have proven to be fiendishly difficult, however. With the exception of one special case (Eliasmith and Anderson 2003), the problem remains largely unsolved. Thus, this section proposes preliminary learning strategies for the cleanup memory that may provide an avenue for discovering such rules. Two approaches are proposed here to update ω_{ij}; one does so by updating M and the other by updating $\tilde{\phi}_i$.

4.4.1 Adaptive SVD

One solution to the problems of size, distribution, and experience acquisition is singular value decomposition (SVD) (Lawson and Hanson 1974). Briefly, SVD

decomposes M into a product of three matrices:

$$M = USV^T \approx U_k S_k V_k^T \tag{4.4}$$

where $U \in \mathbb{R}^{m \times m}$ and $V \in \mathbb{R}^{n \times n}$ are orthonormal matrices, and $S \in \mathbb{R}^{m \times n}$ is a pseudo-diagonal matrix containing the singular values. The k-subscripted matrices are the original matrices reduced to k columns; that is, $U_k \in \mathbb{R}^{m \times k}$, $V_k \in \mathbb{R}^{n \times k}$, and $S_k \in \mathbb{R}^{k \times k}$. The product of these reduced matrices form an MSE optimal, rank-k approximation of M.

Thus, SVD calculates a very good set of k orthonormal vectors that span the row-space of M. Therefore, rather than continually adding vectors to M, an adaptive SVD approximation can reduce experiences to principal components. These components become the clean vectors. Additionally, SVD produces orthonormal vectors, which is the ideal distribution for the cleanup memories defined here. In these circuits, we only require V_k, which replaces M in the butterfly network, and holds the clean vectors for the function network.

SVD is a complicated and expensive calculation, however. Is it possible that the brain could perform such a feat? The neural network community has, in fact, investigated this question for many years (for a unifying approach to these techniques see Fiori (2003)). These approaches do not try to fully calculate the SVD, but rather try to learn two orthonormal matrices, $A \in \mathbb{R}^{m \times k}$ and $B \in \mathbb{R}^{k \times n}$, such that they, respectively, span the column and row space of M. A simple learning rule for calculating these matrices follows from the coupled dynamics (Weingessel

and Hornik 1997),

$$\begin{cases} \Delta A = \kappa \left(MB - AB^T M^T A \right) \\ \Delta B = \kappa \left(M^T A - BA^T MB \right) \end{cases} \tag{4.5}$$

where κ is the learning rate. We are interested in B for the cleanup networks. Thus, the neural learning rule is

$$\Delta \omega_{ij} = \alpha_i \Delta B \tilde{\phi}_i \phi_j. \tag{4.6}$$

The matrix, B, then is stored in the connection weights. It is unclear, however, where A and M might be located.

4.4.2 Self Organizing Neurons

The sparse cleanup in Section 4.3 simulates an environment in which clean vectors are randomly added to preferred direction vectors. This random technique yields surprisingly good results. A learning rule that achieves this performance is

$$\Delta \tilde{\phi}_i = \kappa(t) r_i(t) x_t \tag{4.7}$$

where x_t is the incoming stimuli, $r_i(t) \in [0,1]^n$ is a sparse random vector, and $\kappa(t)$ is a learning rate. This rule obviates the need for the memory matrix, M. Unfortunately, the encoding becomes increasingly dense with each sampling of x_t as each $\tilde{\phi}_i$ gradually approaches $\sum_j v_j$, leading to very poor performance. The standard solution to this problem is to define a learning rate, $\kappa(t)$, that decays in time. This ensures that vectors are not overlearned.

The random selection rule, however, does not seem realistic since it is difficult to postulate a biological explanation for $r_i(t)$. There are better methods of updating $\tilde{\phi}_i$. We now propose a new learning rule, for the sparse network, based on Learning Vector Quantization (LVQ) (Kohonen 1988, 1995).

LVQs are a class of competitive, unsupervised learning algorithms that are typically used in compression algorithms (Gersho and Gray 1992). The goal is to find k representative "reconstruction" vectors, v_i that divide the input data, x_t, into a number of clusters such that the items in the same cluster are similar. The LVQ algorithm is straightforward. Find the closest v_i to x_t and move it, *and only it*, a little towards x_t. The other vectors are pushed away from x_t. The learning rule is simply,

$$\Delta v_i = \kappa(t)\mu_{v_i}(x_t)[x_t - v_i] \tag{4.8}$$

where, $\mu_{v_i}(x_t)$ is 1 if v_i is the closest to x_t, and -1 otherwise. Computational effort consists of calculating all $\mu_{v_i}(x_t)$ and updating each v_i accordingly. This burden need not be so heavy however, when implemented with neurons.

Two features of our neural networks make the LVQ approach computationally cheap. First, an important feature of SDRs is the ease of which structures can be learned. Learned SDRs are simply the superposition of input vectors. Thus unlike LVQ, we need not calculate $[x_t - v_i]$, but merely update the vector by a factor of $x(t)$; this automatically moves v_i towards x_t. Secondly, (4.2) reveals that the input similarity measure, $(\tilde{\phi}_i \cdot x_t)$, is proportional to the input current, $J_i(x_t)$. Therefore,

a natural mechanism for updating $\tilde{\phi}_i$ in realistic neurons is

$$\Delta\tilde{\phi}_i = \kappa(t)\mu_i(J_i(x_t))x_t. \tag{4.9}$$

Notably, we can employ the more convenient membership function, $\mu_i(\cdot)$, defined in (2.10). One can adjust the variance in μ_i (hence the size of membership neighbourhood) to regulate how receptive a neuron is to new data.

Effectively, this rule states that the weights are updated proportionally to the neurons firing rate. This is precisely the standard biological Hebbian learning captured by the moto, "fire together, wire together".

Additionally, clean vectors need not be (nearly) orthogonal, like the SVD solution. Setting $\tilde{\phi}_i$ of a large number of neurons to point in a particular direction, effectively increases the sensitivity of the ensemble in that direction. In these situations, two clean vectors that point in nearly the same direction will be easily distinguishable.

4.5 Summary

This chapter has presented three cleanup memories based on biologically plausible neurons. Of the three, the ANN inspired butterfly network was the poorest performer. However, it offers more than just cleanup. It provides two signal paths: one from the output layer and the other from the similarity layer. The similarity code is very sparse since we expect only a few elements in s to be non-zero. Also, the butterfly network transforms a vector from an n-dimensional space to an m-

dimensional one. If m is smaller than n, the encoding will be very compressed. Caution must be taken however, since too much sparsification and compression can lead to a nearly localist representation particularly if the neurons are defined to be independent.

The function cleanup is a much better network in terms of performance and simplicity of design. It is more accurate, scales better, and is comparable to the optimal cleanup memory with noise. Additionally, the method of calculating the weights is applicable to heteroassociative memories. However, the output is a very full vector, which is not energy efficient from a biological perspective.

By manipulating individual neuron parameters, the sparse cleanup is capable of sparse coding and cleanup. Despite activating fewer neurons, the circuit maintains the same level of performance as the function cleanup. Adjusting the preferred direction to fire along a superposition of clean vectors effectively embeds the power of high dimensional computation inside the neuron. The result is heightened sensitivity to clean vectors, very sparse encoding, and virtually no firing for unrecognized vectors.

Postulating a set of clean vectors or a memory matrix poses an epistemological problem which can be solved with an SVD-like deconstruction. These techniques capture the statistics of the environment to yield a small set of orthonormal vectors which are ideal for the cleanup circuits.

Alternatively, we can adaptively adjust the preferred direction vectors in response to the input it receives. Features of the neurons modeled in the NEF make it possible to embed an LVQ-like learning algorithm within the cell. The proposed

rule suggests a very natural method by which biologically plausible neural networks may learn cleanup. A general solution to learning cleanup networks remains unsolved, however.

Chapter 5

Conclusion and Future Work

Three realistic neural networks, that perform the cleanup task necessary for SDRs, have been presented. Each step has provided deeper insights into the architecture of neural devices.

The butterfly network shows that traditional ANN networks can be converted into neurally plausible networks. However, it performs poorly as a cleanup. More natural circuits can be constructed using the principles of neural engineering. The function cleanup is such a network that has not been proposed by the ANN community, yet is more plausible and performs better than the butterfly network. Furthermore, we find that we can improve the biological plausibility of the function cleanup through sparse coding. Surprisingly, this reveals a new approach to learning that embeds an old algorithm within cellular parameters. A fully neurobiologically plausible learning rule, however, remains elusive. Further research is necessary to find local rules that employ neurally realistic devices.

This thesis has also shown that while SDR operations can be computed within a plausible neural substrate, accurate performance of these networks were impeded by noise. Rather than exhibiting the exponential storage capacity of the optimal cleanup, the function and sparse cleanup showed linear capacity, and the butterfly cleanup had become flat. These networks can be greatly improved. More neurons in higher dimensions are required to adequately emulate the large vectors typically employed. However, these circuits are currently constrained by available computational resources. While faster computers with more memory may provide modest short-term gains, a more promising path is to research and develop newer algorithms and techniques for modeling realistic networks. In particular, we need faster and more memory efficient algorithms for solving the optimal decoders (A.6). Such improvements will allow for larger more complex networks.

Appendix A

Optimal Decoders

A.1 Function Decoding

Here we describe how to calculate the decoding vectors for approximating an arbitrary function. Recall that the optimal decoding weights, ϕ_i, for approximating the transformation, $f(\mathbf{x})$, are found by minimizing the error,

$$E = \frac{1}{2}\left\langle \left[f\left(\mathbf{x}(t)\right) - \sum_{in} \left(h_i(t - t_{in}) + \eta_i\right)\phi_i \right]^2 \right\rangle_{\mathbf{x},t,\eta} \tag{A.1}$$

where $\langle\cdot\rangle_{\mathbf{x}}$ denotes integration over the range of $\mathbf{x}$. First, we simplify this equation by noting that the minimization over $\mathbf{x}$, t and η can be minimized over the range of $\mathbf{x}$, and we write the filtered spike train in terms of its tuning curve,

$$h_i(t - t_{in}) + \eta_i = a_i(\mathbf{x}(t)) = a_i(\mathbf{x}) \tag{A.2}$$

Now, we can write our error term (A.1) as

$$E = \frac{1}{2} \int \left[f(\mathbf{x}) - \sum_{in} a_i(\mathbf{x}) \phi_i \right]^2 d\mathbf{x} \tag{A.3}$$

Following the derivation of Eliasmith and Anderson (2003, Appendix A), we take the derivative with respect to ϕ_i,

$$\begin{aligned} \frac{\partial E}{\partial \phi_i} &= -\frac{1}{2} \int 2 \left[f(\mathbf{x}) - \sum_{jn} a_j(\mathbf{x}) \phi_j \right] a_i(\mathbf{x}) d\mathbf{x} \\ &= -\int f(\mathbf{x})\, a_i(\mathbf{x}) d\mathbf{x} + \int \sum_{jn} a_i(\mathbf{x}) a_j(\mathbf{x}) \phi_j d\mathbf{x} \ , \end{aligned} \tag{A.4}$$

and setting to zero gives

$$\int f(\mathbf{x}) a_i(\mathbf{x}) d\mathbf{x} = \sum_{jn} \int a_i(\mathbf{x}) a_j(\mathbf{x}) \phi_j d\mathbf{x} \tag{A.5}$$

We can write this in matrix-vector notation as, $\Upsilon = \Gamma \phi$, where

$$\begin{aligned} \Gamma_{ij} &= \langle a_i(\mathbf{x}) a_j(\mathbf{x}) \rangle_{\mathbf{x}} \\ \Upsilon_i &= \langle a_i(\mathbf{x}) f(\mathbf{x}) \rangle_{\mathbf{x}} \end{aligned} \tag{A.6}$$

and solve, $\phi = \Gamma^{-1} \Upsilon$.

A.2 Point Sampling

For comprehensiveness, we make no assumptions about $f(x)$, thereby ruling out an analytic solution. Nevertheless, we can achieve close approximations with simple numerical techniques such as point sampling. Given a sample of points $\mathbb{P} = \{p_1, p_2, ..., p_n\}$, we can approximate the integral in (A.6) with matrix summations:

$$\begin{aligned} \Gamma_{ij} &= \sum_{p \in \mathbb{P}} a_i(p) a_j(p) \Delta \mathbf{x}_p \\ \Upsilon_i &= \sum_{p \in \mathbb{P}} a_i(p) f(p) \Delta \mathbf{x}_p. \end{aligned} \tag{A.7}$$

where $\Delta \mathbf{x}_p$ is the area represented by point p. Assuming all areas are the same, $\Delta \mathbf{x}_p = \Delta \mathbf{x}$, allows us to drop the term altogether when solving $\phi = \Gamma^{-1} \Upsilon$.

The points over the range of $\mathbf{x}$ can be regularly sampled as in standard numerical integration, randomly sampled as in Monte Carlo methods, or chosen at appropriate locations where accuracy is desired.

Bibliography

D. J. Chalmers. Why Fodor and Pylyshyn were wrong: The simplest refutation. In *Proceedings of The Twelfth Annual Conference of the Cognitive Science Society, Cambridge, MA, July 1990*, pages 340–347, Hillsdale, NJ, 1990. Lawrence Erlbaum Associates.

P. S. Churchland and T. J. Seynowski. *The Computational Brain.* MIT Press, Cambridge, MA, USA, 1994. ISBN 0262531208.

J. Conklin and C. Eliasmith. Transformations of structured distributed representations in a neurobiologically plausible network: An application to the wason card selection task. Describes neural circuits that perform HRR binding and unbinding, forthcoming.

C. P. Dolan. *Tensor manipulation networks : connectionist and symbolic approaches to comprehension, learning, and planning.* PhD thesis, UCLA Computer Science Department, 1989.

C. Eliasmith. The myth of the turing machine: The failings of functionalism and

related theses. *JETAI: Journal of Experimental & Theoretical Artificial Intelligence*, 14, 2002.

C. Eliasmith. Learning context sensitive logical inference in a neurobiological simulation. In S. Levy and R. Gayler, editors, *Compositional Connectionism in Cognitive Science*, pages 17–20. AAAI Fall Symposium, AAAI Press, 2004.

C. Eliasmith. A unified approach to building and controlling spiking attractor networks. *Neural Computation*, in press.

C. Eliasmith and C. H. Anderson. *Neural Engineering: Computational, Representation, and Dynamics in Neurobiological Systems*. MIT Press, Cambridge, MA, USA, 2003. ISBN 0262050714.

S. Fiori. Singular value decomposition learning on double stiefel manifold. *International journal of neural systems*, 13(3):155–170, Jun 2003.

J. Fodor. *Psychosemantics: The Problem of Meaning in the Philosophy of Mind*. The MIT Press, Cambridge, Massachusetts, 1987.

J. Fodor and B. P. McLaughlin. Connectionism and the problem of systematicity: Why smolensky's solution doesn't work. *Cognition*, 35(2):183–204, May 1990.

R. W. Gayler. Multiplicative binding, representation operators & analogy (workshop poster), Jan. 01 1998.
http://cogprints.ecs.soton.ac.uk/archive/00000502.

T. V. Gelder. Compositionality: A connectionist variation on a classical theme. *Cognitive Science*, 14, 1990.

A. Georgopoulos, A. Schwartz, and R. Kettner. Neuronal population coding of movement direction. *Science*, 233:1416–1419, 1986.

A. Gersho and R. M. Gray. *Vector Quantization and Signal Compression*. Kluwer Academic Publishers, Norwell, MA, USA, 1992. ISBN 0-7923-9181-0.

J. J. Hopfield. Neural networks and physical systems with emergent collective computational facilities. *Proceedings of the National Academy of Sciences of the USA*, 79:2554–2558, 1982.

P. Kanerva. *Sparse Distributed Memory*. MIT Press, Cambridge, MA, 1988.

P. Kanerva. Binary spatter-coding of ordered k-tuples. In *ICANN 96: Proceedings of the 1996 International Conference on Artificial Neural Networks*, pages 869–873, London, UK, 1996. Springer-Verlag. ISBN 3-540-61510-5.

P. Kanerva. Fully distributed representation. In *Proc. 1997 Real World Computing Symposium 1997 (Tokyo, Japan)*, pages 358–365, 1997.

P. Kanerva. Pattern completion with distributed representation. In *IEEE International Conference on Neural Networks (IJCNN'98)*, volume II, pages II–1416–II–1421, Anchorage, AK, July 1998. IEEE. Swedish Institute of Computer Science, .se.

C. Koch. *Biophysics of Computation, Information Processing in Single Neurons*. Oxford University Press, 1998.

T. Kohonen. *Associative Memory,a System-Theoretical Approach*. Springer, Berlin, 1 edition, 1977. ISBN 3-540-08017-1.

T. Kohonen. Learning Vector Quantization. *Neural Networks*, 1(Supplement 1): 303, 1988.

T. Kohonen. *Self-Organizing Maps*, volume 30 of *Springer Series in Information Sciences*. Springer, Berlin, Heidelberg, 1995. (Second Extended Edition 1997).

J. F. Kolen and J. B. Pollack. Multiassociative memory. In *The proceedings of the Thirteenth Annual Conference of the Cognitive Science Society*. Cognitive Science Society, Aug. 1991.

C. L. Lawson and R. J. Hanson. *Solving Least Squares Problems*. Series in Automatic Computation. 1974. ISBN 0-13-822585-0.

Y. LeCun and J. Denker. Natural versus universal probability complexity, and entropy. In *IEEE Workshop on the Physics of Computation*, pages 122–127. IEEE, 1992.

B. B. Murdock. A theory for the storage and retrieval of item and associative information. *Psychological Review*, 89(6):316–338, 1982.

B. B. Murdock. A distributed memory model for serial-order information. *Psychological Review*, 90(4):316–338, 1983.

A. Newell and H. A. Simon. Computer science as empirical inquiry: symbols and search. *Commun. ACM*, 19(3):113–126, 1976. ISSN 0001-0782.

B. A. Olshausen and D. J. Field. Sparse coding of sensory inputs. *Current opinion in neurobiology*, 14(4):481–487, Aug 2004. JID: 9111376; RF: 54; ppublish.

C. Parisien and C. Eliasmith. Negative weights and neural inhibition. A general solution to the negative weights problem, forthcoming.

T. Plate. Randomly connected sigma-pi neurons can form associator networks. *Network: Computation in Neural Systems*, 11, 2000.

T. A. Plate. *Holographic Reduced Representation: Distributed Representation for Cognitive Structures*. CSLI Publications, Stanford, CA, USA, 2003. ISBN 1575864290.

T. A. Plate. *Distributed Representations and Nested Compositional Structure*. PhD thesis, University of Toronto, 1994.

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

T. A. Plate. A common framework for distributed representation schemes for compositional structure. In F. Maire, R. Hayward, and J. Diederich, editors, *Connectionist Systems for Knowledge Representation and Deduction*, pages 15–34, Queensland University of Technology, 1997.

J. B. Pollack. Recursive distributed representations. *Artificial Intelligence*, 46(1-2): 77–105, Nov. 1990.

D. E. Rumelhart and J. L. McClelland. *PDP models and general issues in cognitive science*. MIT Press, Cambridge, MA, USA, 1986. ISBN 0-262-68053-X.

G. Sjödin. SICS strategic research program 1997. Technical Report R97-04, Swedish

Institute of Computer Science, 1997.
http://www.sics.se/sicsinfo/research_prog/rp97/srp4.html.

P. Smolensky. Tensor product variable binding and the representation of symbolic structures in connectionist systems. *Artificial Intelligence*, 46(1-2):159–216, 1990.

D. S. Touretzky. BoltzCONS: Reconciling connectionism with the recursive nature of stacks and trees. In *Proceedings of 8th Annual Conference of the Cognitive Science Society*, Amherst, MA, 1986. Lawrence Erlbaum Associates.

D. S. Touretzky. BoltzCONS: Dynamic symbol structures in a connectionist network. *Artificial Intelligence*, 46(1-2):5–46, Nov. 1990.

D. S. Touretzky and G. E. Hinton. Symbols among the neurons: Details of a connectionist inference architecture. *IJCAI*, 9:238–243, Aug. 1985.

D. S. Touretzky and G. E. Hinton. A distributive connectionist production system. *Cognitive Science*, 12(3):423–466, July 1988.

A. Weingessel and K. Hornik. SVD algorithms: APEX-like versus subspace methods. *Neural Processing Letters*, 5:177–184, 1997.

D. J. Willshaw, O. P. Buneman, and H. C. Longuet-Higgins. Nonholographic associative memory. *Nature*, 222:960–962, 1969.

L. A. Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965.